Functional Programming in Lambda Expressions

Shing-Chi Cheung
Computer Science and Engineering
HKUST

Functional Programming

- Several important features have been added to Java since 8
- A very important one is Lambdas (expressions)
- Background:
  - Rising demands to support functional programming for big data and cloud applications
  - Rising demands to leverage the parallel computation power supported by GPU cards and multi-core CPUs in servers and mobile devices
  - Such demands are so high that lambda is recently supported by all major programming languages (Python, C#, Java, JavaScript, Go, Kotlin, C++, PHP)
    - https://en.wikipedia.org/wiki/Higher-order_function

Origin of Lambda

- Two important computation models were proposed two computer scientists in 1936
  - Alan Turing proposed computation using the Turing machine → von Neumann architecture that leads to imperative programming
  - Alonzo Church proposed computation using lambda-definable functions whose values can be obtained by repeated substitution → lambda calculus that leads to functional programming

Alan Turing

Alonzo Church

https://plato.stanford.edu/entries/church-turing/#CompTuriChurAppr

Functional programming in lambda expressions

- In Java, a lambda (expression) is an anonymous function to be performed by an object that implements a functional interface. We call the object a function object
- Mathematically, lambdas (a.k.a. closures) take the form:
  - $\lambda xy.t$ where x, y are variables and t is a lambda term
  - Example: $\lambda xy.x^2+y^2$
- In Java, a lambda expression takes the form:
  - (x, y) -> t where *x, y* are variables and *t* is a lambda term
  - Example: (x, y) -> x*x + y*y;

Waiving two paradigms into one language

Functional Programming

A lambda that defines an anonymous function to be performed by an object that implements the functional interface EventHandler<ActionEvent>

- Concise syntax
  - More readable than anonymous inner classes

```
btUp.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
    }
  }
);
```

(a) Anonymous inner class event handler

```
btUp.setOnAction(e -> {
  text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
});
```

(b) Lambda expression event handler

Functional Programming

- Automatic type inference
  - Type of function object and its parameter(s) are inferred.
  - No need to change the client code that uses a library API (e.g., EventHandler and ActionEvent) even their names are changed.

```java
btUp.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
    }
  }
);
```

(a) Anonymous inner class event handler

```java
btUp.setOnAction(e -> {
  text.setY(text.getY() > 10 ? text.getY() - 5 : 10);
});
```

The highlighted code defines and creates a function object (or lambda) that implements a functional interface

(b) Lambda expression event handler

Type Inference of Lambda Expression

A slide extracted from Event Handling #30

```
btUp.setOnAction(e -> {text.setY(text.getY() > 10 ? …);});
```

1. Java compiler infers that the argument's type must be EventHandler<ActionEvent>
2. Java compiler infers that the concerned lambda expression must define the abstract method, which is handle(ActionEvent evt) in the interface

```
interface EventHandler<ActionEvent> {
  public abstract void handle(ActionEvent evt);
}
```

3. Java compiler infers that e in the lambda expression must corresponds to evt in handle's method parameter
4. Java compiler infers that the type of e in the lambda expression is ActionEvent, and {text.setY(text.getY() > 10 ? …);} defines the overriding handle method's body

Functional Programming

- A key programming construct that supports big data and cloud programming
  - Allow programming of popular higher-order functions such as map, reduce and filter
  - Facilitates the use of powerful built-in functional interfaces
- Support streams and lazy evaluation
  - Subject to parallel processing by GPUs and multi-cores
  - for (Frame f: video) f.process(); *// requires whole video available*
  - video.parallelStream().forEach(f->process()); *// stream-based*

From Anonymous Innerclass to Lambda

```
myName( new Names() { // anonymous inner class
@Override
public void sayName(String n) {System.out.println("My Name is " + n);} }, “John”);
```

- Omit interface and method names

```
myName( (String n) -> {System.out.println("My Name is " + n);}, “John”);
```

- Omit parameter types

```
myName((n) -> {System.out.println("My Name is " + n);}, “John”);
```

- Drop parentheses if single parameter or single statement

```
myName(n -> System.out.println("My Name is " + n), “John”);
```

From Anonymous Innerclass to Lambda

- Anonymous innerclass before Java 8

```
new Thread( new Runnable() { // anonymous inner class
  @Override
  public void run() {
   processSomeImage(imageName);
  }
});
```

()->processSomeImage(ImageName) is a lambda that creates a Runnable instance. We call the instance a function object because it is an object of a class that implements a functional interface. **A lambda is treated as a function object**.

- Lambda since Java 8

no argument

```
new Thread(() -> processSomeImage(imageName));
```

function object

How is Lambda Used?

```
btUp.setOnAction(e -> {text.setY(text.getY() > 10 ? …);});
```

- A lambda provides a value for a variable or parameter that expects a function object (i.e., an instance of a class that implements a functional interface)
- Previously, we mainly illustrate the use of lambdas to define event handlers, where lambda variables reference event objects
- We will illustrate examples on how lambdas can be used to define higher-order functions
- We will define methods that take lambdas (i.e., function objects) as arguments (cf. passing function pointers to C++ methods)

Higher-Order Functions

- A higher-order function is a function that takes other functions (i.e., function objects) as arguments

```
interface Func {
  public double eval(double x);
}
public class LambdaTest {
  public double calc(Func f, double x) {
    return f.eval(x);
  }
}
```

We provide a lambda that defines f.eval() each time we call calc()

Examples of using calc():

```
System.out.println(calc(x->x+x, 3));
System.out.println(calc(x->x*x, 3));
System.out.println(calc(x->x/x, 3));
```

Output:

```
6.0
9.0
1.0
```

LambdaTest.java

Terminologies

- A function is 0-order if it does not take any functions as arguments
- A function is (n+1)-order if the highest order of functions that it takes as arguments is n where n ≥ 0
- A function is higher-order if its order is greater than 0
- A higher-order function may return a function as output
- Examples:
  - 0-order: eval(double x) *// Func is 0-order*
  - 1-order: calc(Func f, double x)

```java
interface Func {
  public double eval(double x);
}
```

Example 1: Numerical Integration

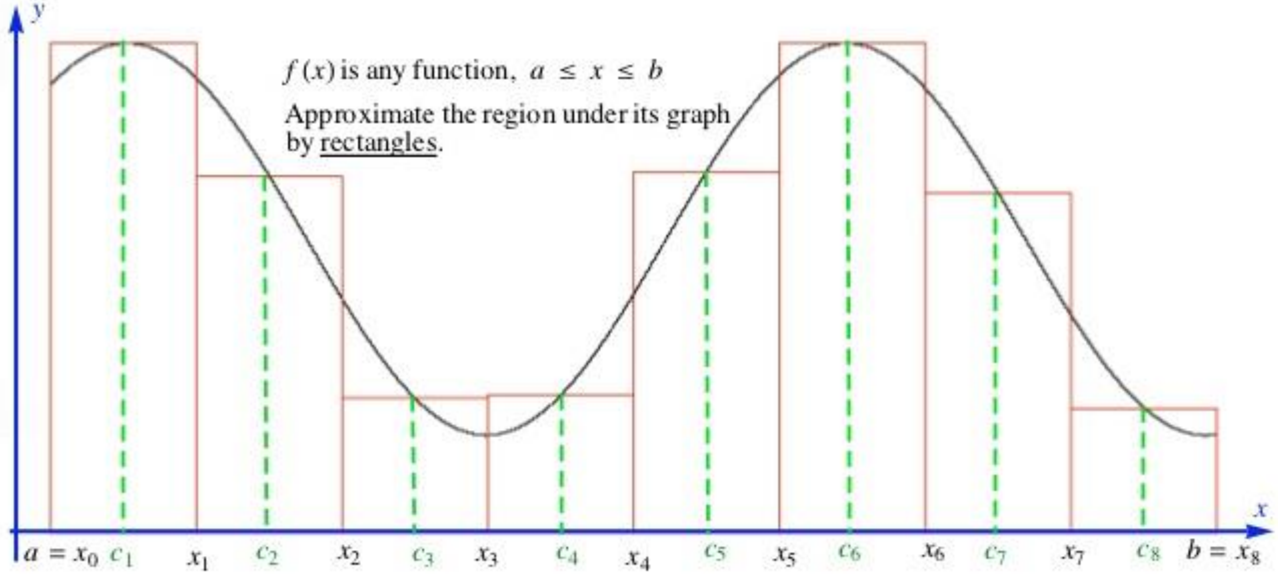
- Define a functional interface with a method

```
interface Integrable {
  double eval(double x);
}
```

```
interface Func {
  public double eval(double x);
}
```

semantically the same

The Midpoint Rule



- Divide $[a, b]$ into n subintervals of equal length $\Delta x = \dfrac{b-a}{n}$
- In each subinterval take the midpoint: $c_1, c_2, \dots c_n$
- $\int_a^b f(x)\,dx \approx M_n$ where

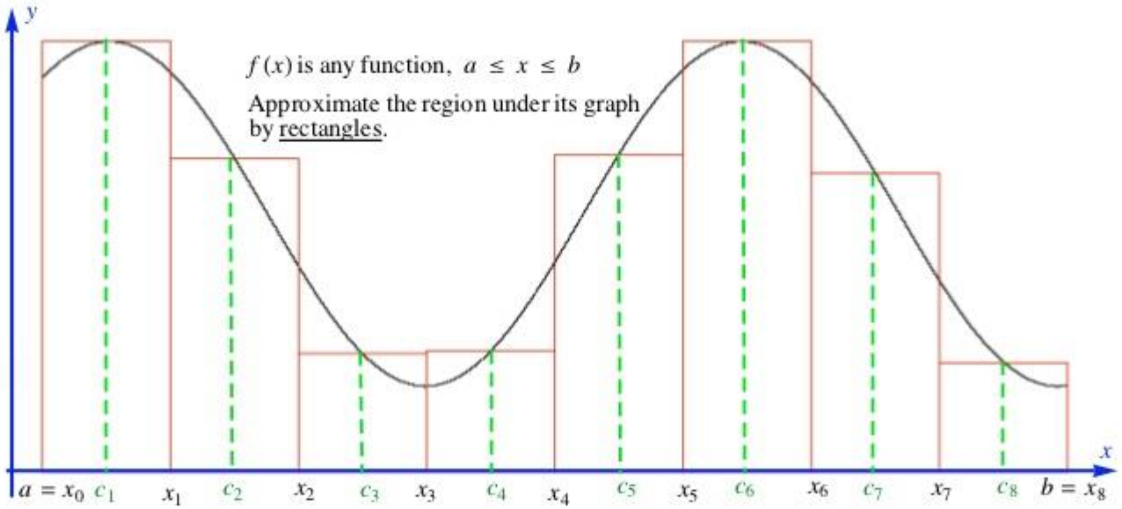
$M_n =$ the sum of the areas of the above rectangles

Example 1: Numerical Integration

```
public static double integrate(Integrable f, double a, double b, int numSlices) {
  if (numSlices < 1) numSlices = 1;
  var delta = (b-a)/numSlices;
  var start = a + delta/2;
  var sum = 0.0;
  for (var i=0; i<numSlices; i++)
    sum += delta*f.eval(start+delta*i);
  return sum;
}

interface Integrable {
  double eval(double x);
}
```

f is a (function) object that implements Integrable, which supports eval(double)



- Divide $[a, b]$ into n subintervals of equal length $\Delta x = \dfrac{b-a}{n}$

LambdaIntegration

Example 1: Numerical Integration

```java
public static void integrationTest(Integrable f, double a, double b) {
  for (var i=1; i<7; i++) {
    var numSlices = (int)Math.pow(10, i);
    var result = integrate(f, a, b, numSlices);
    System.out.printf(" For numSlices =%,10d result = %,.8f%n",numSlices, result);
  }
}
public static void main(String args[]) {
  System.out.println("Estimate integral of x^2 from 10 to 100");
  integrationTest(x->x*x, 10, 100);
  System.out.println("Estimate integral of sin(x) from 0 to PI");
  integrationTest(x->Math.sin(x), 0, Math.PI);
}
```

LambdaIntegration.java

Example 2: Measure Runtime Utilization of a Calculation

- Cannot implement the method *measure* as:
  - measure(*someCalculation()*);
  - *someCalculation* is computed before measure's body is executed.
- Implement the method with a function parameter so that it can be used as:
  - measure(()->*someCalculation*));
  - measure() runs *someCalculation* inside its body and thereby measures the runtime overhead.

Example 2: Measure Runtime Utilization of a Calculation

```java
private static final double ONE_BILLION = 1_000_000_000;
public static void measure(Op operation) {
  long startTime = System.nanoTime();
  operation.runOp();
  long endTime = System.nanoTime();
  double elapsedSeconds = (endTime-startTime)/ONE_BILLION;
  System.out.printf("  Elapsed time: %.3f seconds.\n", elapsedSeconds);
}
public static void main(String[] args) {
  measure(()->LambdaIntegration.integrationTest(x->x*x,10,100));
  measure(()->LambdaIntegration.integrationTest(x->Math.sin(x),0,Math.PI));
  measure(()->LambdaIntegration.main(null));
}
```

```java
interface Op {
  void runOp();
}
```

LambdaTimeOp.java

@FunctionalInterface Annotation

```java
@FunctionalInterface
interface Op {
    void runOp();
}
```

- Tell compiler that it is a functional interface
- Tell other developers that this is a functional interface to be implemented by lambdas
- They should not declare additional abstract methods to this interface

:: Method References (Since Java 8)

- Can we simplify an lambda expression if its functionality is mapped to a method or constructor?
  - Yes – static/instance method references and constructor references
- Java supports a notation for static method references ::
  - (args) -> ClassName.staticMethod(args) *// must refer to the set of arguments*
  - ClassName::staticMethod *// treat as a static method reference*
- Examples:

✓ integrationTest(x->Math.sin(x), 0, Math.PI); → integrationTest(Math::sin, 0, Math.Pi);

✗ measure(()->LambdaIntegration.main(null)); → measure(LambdaIntegration::main);
// un-matching args

no arg *one arg*

:: Method References (Since Java 8)

- Java supports a notation for instance method references ::
  - (args) -> obj.instanceMethod(args)
  - obj::instanceMethod *// treat as a instance method reference*
- Examples
  - public double square(double x) { return x*x; }
  - integrationTest(x->o.square(x)), 10, 100);
  - integrationTest(o::square, 10, 100); // ok

LambdaMethodReference.java

Ambiguities Arising from Method References?

Suppose two overloading methods are defined:

- **public double** square(**double** x) { **return** x*x; }
- **public double** square(**double** r, **double** i) { **return** r*r + i*i; } *// complex num*

Which method will be referenced?

- integrationTest(o::square, 10, 100); *// no arguments specified in o::square*
- Compiler resolves it by the abstract method in the functional interface type for the Lambda argument.
- **public static void** integrationTest(Integrable f, **double** a, **double** b) {…}
- interface Integrable { **double** eval(**double** x); } *// matches the first square*

Advantages of Method References?

- Concise and more natural
  - integrationTest(o::square, 10, 100); *// drops arguments*
  - integrationTest(x->o.square(x), 10, 100);
- Maintenance friendly – we can keep the reference (e.g., o::square) even we have changed the functional interface

```
interface Integrable {
  double eval(double x);
}
```

⇩

```
interface Integrable {
  double eval(double r, double i);
}
```

- Suppose we want to define an integration for complex numbers:
  - **interface** Integrable { **double** eval(**double** r, **double** i); }
  - **public static void** integrationTest(Integrable f, **double** a, **double** b) {…}
    - $\underbrace{\text{integrationTest(o::square, 10, 100);}}_{\textit{No need to change}}$ becomes integrationTest((r, i)->o.square(r, i), 10, 100);

Scope of Lambdas

Is the following code in a method body legal?

```
…
double x = 0;
f(x->x+x);
…
```

Illegal: x already defined in the scope

Scope of Lambdas

Is the following code legal?

```
void foo () {
  double x = 0;
  f(y->{ double x = 3.4; … });
  …
}
```

Illegal: repeated variable declaration in lambda and its embedding method

Scope of Lambdas

Is the following code legal?

```
void foo () {
  double x = 0;
  f(y -> x = 3.4);
  …
}
```

Illegal: Variables used in lambda should be final or effectively final

Scope of Lambdas

Is the following code legal?

```
class MyClass {
  private double x = 0;
  void foo () {
    f(y -> x = 3.4);
  }
}
```

Legal: modifying instance variable x

Scope of Lambdas

LambdaScope.java

Is the following code legal? What does this refer to?

```
class MyClass {
  private double x = 0;
  void foo () {
    f(x -> x + this.x);
  }
}
```

Summary:

- The scope of lambda variables is their embedding method (e.g. foo() in the example).
- A lambda can access local variables and all variables/methods defined in its outer class
- Local variables used by lambda must be effectively final.
- “this” in a lambda (function object) refers to the instance of its embedding method. No separate .class files are generated for lambdas.

Legal: lambda variable name matches instance variable name

Generic Programming using Lambdas (Since Java 8)

- We will see how functional interfaces provide a power facility for generic and parallel programming
- To facilitate that, Java supports a set of built-in functional interfaces
  - Predicate<T>: *T -> boolean*
  - Function<T, R>: *T -> R*
  - Consumer<T>: *T -> void*
  - Supplier<T>: *() -> T*
  - …

Built-in Functional Interfaces: Predicate<T>

- java.util.function.Predicate<T>

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

- For creating a function object to test if a condition holds
- Commonly used to filter from a list (or stream) those elements that satisfy a condition. The condition test can be provided on-the-fly by a function object of type Predicate

Built-in Functional Interfaces: Predicate<T>

- java.util.function.Predicate<T>

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

If a functional interface has only one type parameter, it tends to describe the lambda variable type and the lambda returns either void or a primitive type

- For creating a function object to test if a condition holds
- Commonly used to filter from a list (or stream) those elements that satisfy a condition. The condition test can be provided on-the-fly by a function object of type Predicate

Using Predicate<T> functional interface

- If we want to support a few queries on an USTEmployee list.
  - findEmployeeByFirstName
  - findEmployeeByAge
  - findEmployeeBySalary
  - …
- Prior to Java 8, we need to write a separate method for each query.

findEmployeebyFirstName and findEmployeebyAge

```java
public static USTEmployee findEmployeebyFirstName
(List<USTEmployee> employees, String firstName) {
  for (USTEmployee e: employees) {
    if (e.getFirstName().equals(firstName)) { return e; }
  }
  return null;
}
```

What do you think?

```java
public static USTEmployee findEmployeebyAge
(List<USTEmployee> employees, int ageCutoff) {
  for (USTEmployee e: employees) {
    if (e.getAge() >= ageCutoff) { return e; }
  }
  return null;
}
```

Differences are highlighted in red. **List** is a commonly used Java interface in Lambdas. Classes implementing **List** include ArrayList, LinkedList, Stack, …

LambdaEmployee.java

findEmployeebyFirstName and findEmployeebyAge

```java
public static USTEmployee findEmployeebyFirstName
(List<USTEmployee> employees, String firstName) {
  for (USTEmployee e: employees) {
    if (e.getFirstName().equals(firstName)) { return e; }
  }
  return null;
}
```

I hate it! Tedious and difficult to maintain!!

Suppose our application has 3 data lists (or tables), each has 10 attributes. To support various queries on these attributes, we will need to write **3,069** (=3x(2^{10}-1)) find methods, which need to be consistently maintained in future.

```java
public static USTEmployee findEmployeebyAge
(List<USTEmployee> employees, int ageCutoff) {
  for (USTEmployee e: employees) {
    if (e.getAge() >= ageCutoff) { return e; }
  }
  return null;
}
```

LambdaEmployee.java
findWithoutLambda()

Using Predicate<T> functional interface

- We can write a generic function that finds an element for any criteria in any type of list using Predicate.

This generic function can be written independently from the LambdaEmployee program

```
public static <T> T find(List<T> dataset, Predicate<T> predicate) {
  for (var e: dataset)
    if (predicate.test(e))
      return e;
  return null;
}
```

Exempt the need to write separate find methods:

- *FindEmployeeByFirstName:*
  find(employees, e->e.getFirstName().equals(“David”))
- *FindEmployeeByAge:*
  find(employees, e->e.getAge() >= 42)
- *FindEmployeeByGender:*
  find(employees, e->e.getGender() == “F”)
- *FindCarByMileage:*
  find(cars, c->c.getMileage() < 1000)

LamdaEmployee.java
findWithLambda()

Exercise 1

- Can you modify the generic method find(…) to findAll(…) so that it finds all elements in a dataset?

```
public static <T> List<T> findAll(List<T> dataset, Predicate<T> predicate) {
    List<T> matches = new ArrayList<>();  // diamond operator
    for (T e: dataset)
        if (predicate.test(e))
            matches.add(e);
    return matches;
}
```

LamdaEmployee.java
findAllWithLambda()

Built-in Functional Interface: Function<T,R>

- Helps apply the function apply(T t) iteratively on each element of type T in a list with a result of type R

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

If a functional interface has more than one type parameter, the last one tends to describe the output type and the remaining describes the lambda variable(s) type(s)

- Example: We apply a function to raise each employee's salary. The function takes an instance of **USTEmployee (T)** and returns an instance of **Double (R)**

```java
Function<USTEmployee, Double> raise = e -> e.getSalary()*1.1;
for (var e: employees)
  e.setSalary(raise.apply(e));
```

raise references a function object that implements apply(T t) using the lambda expression

LambdaEmployee.java
raiseSalaryUsingFunction()

Built-in Functional Interface: Function<T,R>

- We can build on top of Function<T,R> and write a generic function that calculates the sum of all elements in a list of any type

```
public static <T> double mapSum(List<T> list, Function<T, Double> mapper) {
  var sum = 0.0;
  for (var entry : list) {
    sum += mapper.apply(entry);
  }
  return sum;
}
```

$$mapsum(\mathcal{L}, mapper) = \sum_{e \in \mathcal{L}} mapper(e)$$

Built-in Functional Interface: Function<T,R>

- We can now use mapSum to sum up a numeric property of each element in a given list.
- To sum up the total salary from a list of *employees*

```
mapSum(employees, e -> e.getSalary());
```

- To sum up the total balance from a list of purchased *items*

```
mapSum(items, Item::getPrice);
```

LambdaEmployee.java
mapSumTest()

Built-in Functional Interface: Consumer<T>

- Function<T,R> assumes a function that returns a value of type R.
- What if a functions that returns nothing but introduces side effect to the input object t?
- Consumer<T>: Let us make a "function" that takes an object of type T and does some side effect to it (with no return value)

```
public interface Consumer<T> {
  public void accept(T t); // makes side effect to t
}
```

Built-in Functional Interface: Consumer<T>

```java
public void accept(Employee e) { e.setSalary(e.getSalary()*1.1); }
```

```java
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
for (var e: employees) raise.accept(e);
```

- Create a function object implementing an accept method that raises an input employee's salary by 10%.
  - *Note: "e.setSalary(e.getSalary()*1.1)" returns nothing*
- Apply the method to each employee in employees using a for-loop.

LambdaEmployee.java
raiseSalaryUsingConsumerWithoutForEach()

Built-in Functional Interface: Consumer<T>

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
for (var e: employees) raise.accept(e);
```

The above for-statement can be replaced by:

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
employees.forEach(raise);
```

What are the advantages of the replacement?

It removes the need to specify the “accept(e)” method in the for-loop.

Built-in Functional Interface: Consumer<T>

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
employees.forEach(raise);
```

- forEach(*Consumer<? super E>*) returns void

We may cast raise to a supertype and then iterate it.

  - An *important* default method defined by interfaces Iterable<E> and Stream<E>
    - employees is an instance of List<USTEmployee>
    - List<E> extends Collection<E>, which extends Iterable<E>

forEach() is optimized to these two data structures

  - forEach takes a Consumer function object and returns nothing
  - forEach allows lists to be processed as unbounded streams if needed:
    - employees.stream().forEach(raise); // stream is a method of Collection that returns a Stream
  - forEach method is applied to stream elements once they are available
  - Stream<E> is a class introduced in Java 8 for (parallel) processing of big data

Built-in Functional Interface: Consumer<T>

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
employees.forEach(raise);
```

- A more common way to code this example by Java practitioners:

```
employees.forEach(e -> e.setSalary(e.getSalary()*1.1));
```

1. *compiler automatically infers that it is a Consumer function object.*
2. *The generic type T of Consumer<T> binds to the type of lambda variable e (which is USTEmployee in the example).*

LambdaEmployee.java
raiseSalaryUsingConsumerWithForEach()

Built-in Functional Interface: Consumer<T>

```
employees.forEach(e -> e.setSalary(e.getSalary()*1.1));
```

- Turn a list into a sequential stream processing:

```
employees.stream().forEach(e -> e.setSalary(e.getSalary()*1.1));
```

- Turn a list into a parallel stream processing:

```
employees.parallelStream().forEach(e -> e.setSalary(e.getSalary()*1.1));
```

To learn more on stream operations: https://www.youtube.com/watch?v=bzO5GSujdqI

Type Casting of Functional Interfaces

- functional interfaces

My Consumer interface

```
interface Consumer<T> {void accept(T t);}
interface MyInterface {void processEmployee(USTEmployee e);}
```

- incompatible types cannot be automatically converted

```
Consumer<USTEmployee> raise = e -> e.setSalary(e.getSalary()*1.1);
MyInterface mi = e -> e.setSalary(e.getSalary()*1.1);
mi = raise; // error: types are not compatible
```

Same lambda

```
mi = (MyInterface) raise; // Can we cast Consumer<USTEmployee> to MyInterface?
```

Type Casting of Functional Interfaces

LambdaEmployee.java
TypeCasting.testTypeCasting()

- functional interfaces

```
interface Consumer<T> {void accept(T t);}
interface MyInterface {void processEmployee(USTEmployee e);}
```

- Cannot resolve setSalary and getSalary. Why?

```
var o = e -> e.setSalary(e.getSalary()*1.1); // error
```

Tell java compiler the lambda's type

```
var o = (MyInterface) e -> e.setSalary(e.getSalary()*1.1);
```

```
var o = (Consumer<USTEmployee>) e -> e.setSalary(e.getSalary()*1.1);
```

Useful built-in functional interfaces for future references (optional)

- Lookup the online Java tutorial or other web resources on the use of other built-in functional interfaces:
  - BinaryOperator<T>: *(T, T) -> T*
  - Block<T>: *T -> void*
  - Combiner<T, U, V>: *(T, U) -> V*
  - Factory<T>: *() -> T*
  - Mapper<T, U>: *T -> U*
  - Supplier<T>: *() -> T*
  - UnaryOperator<T>: *T -> T*
  - …

https://docs.oracle.com/javase/tutorial/

Function Composition (by Predicate default methods)

- So far, we pass non-function objects (e.g., USTEmployee) to the method defined in the built-in functional interfaces

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

```
// anonymous innerclass
Predicate<USTEmployee> isYoung = new Predicate() {
  boolean test(USTEmployee e) {
    return e.getAge() < 30;
  }
}
```

  - Predicate<USTEmployee> isYoung = e -> e.getAge()<30;
  - Predicate<USTEmployee> isRich = e -> e.getSalary()>=30000;
  - In both cases, we pass a USTEmployee object e to the test() method

Function Composition (by Predicate default methods)

```
Predicate<USTEmployee> isYoung = e -> e.getAge()<30;
Predicate<USTEmployee> isRich = e -> e.getSalary()>20000;
```

- Can we use the Predicate<T> to logically compose these two lambdas?
  - E.g., Find employees who are both young and rich.
- Yes, we can compose these two lambdas using default methods.
- Example:
  - query = find(employees, isRich.and(isYoung));

take a function object as parameter

LambdaEmployee.java
composeFunctions()

compose two lambdas using the and( ) default method in the Predicate interface

Function Composition by Predicate default methods

- and: *Predicate<? super T> -> Predicate<T> // isYoung -> isRich&isYoung*
  - a *default method* that accept a predicate *p* and returns a composed predicate whose test method is true if both the current predicate and *p* is true.
- or: *Predicate<? super T> -> Predicate<T>*
  - a *default method* that accepts a predicate *p* and returns a composed predicate whose test method is true if either the current predicate or *p* is true.
- negate: *() -> Predicate<T>*
  - a *default method* that returns a predicate negating the current one.
- isEqual: *Object -> Predicate<T>*
  - a *static method* that accepts an object and returns a predicate whose test method is true if the current predicate’s argument equals to the object.

Predicate default Methods

- and, or, negate, isEqual are default methods defined by the built-in Predicate interface to facilitate predicate composition.
- They cannot be abstract methods because otherwise:
  - Predicate functional interface contains more than one abstract method, violating the language rule.
  - Developers need to implement these methods for each Predicate function object, which is tedious.
  - Java 8 provides default implementation of these methods.
  - and, or, negate can be overridden by Predicate function objects.

Examples of and, or, negate, isEqual

- Predicate<USTEmployee> isYoung = e -> e.getAge()<30;
- Predicate<USTEmployee> isRich = e -> e.getSalary()>20000;
- findAll(employees, isRich.and(isYoung))
- findAll(employees, isRich.or(isYoung))
- findAll(employees, isRich.negate())

- Predicate<USTEmployee> sameAsFirst =
  Predicate.isEqual(employees.get(0));
- findAll(employees, sameAsFirst)

Exercise 2:

- Write a generic function *conjunct* that accepts variable Predicate arguments and return a predicate that conjunct all arguments?

  E.g., conjunct(w->w.contains(“h”), w->w.contains(“l”), w->w.length()<=4)
  is evaluated to: w->true.and(w->w.contains(“h”).and(w->w.length()<=4))

```
public static <T> Predicate<T> conjunct(Predicate<T>... predicates) {
    Predicate<T> result = e->true; // a trivial tautology if no input arguments
    for (var e: predicates)
        result = result.and(e);
    return result;
}
```

Exercise 3:

- Write a generic function *findAll* that accepts a list and variable Predicate arguments, and returns all elements that satisfy all the predicate arguments from the list? Assume the *findAll* in Exercise 1 and *conjunct* in Exercise 2.

E.g., findAll(words, w->w.contains(“h”), w->w.contains(“l”), w->w.length()<=4)

```java
public static <T> List<T> findAll(List<T> dataset, Predicate<T>... predicates) {
    Predicate<T> compositePredicate = conjunct(predicates);
    return findAll(dataset, compositePredicate);
}
```

Exercise 4: Using the findAll with multiple predicates

```
private static List<String> words = Arrays.asList("hi", "hello", "hola",
"bye", "goodbye", "adios");
public static void findAllExample() {
  List<String> hWords = Util.findAll(words, w->w.contains("h"));
  System.out.println("Words with h: "+hWords);
  List<String> hlWords = Util.findAll(words, w->w.contains("h"),
    w->w.contains("l"));
  System.out.println("Words with h and l: "+hlWords);
  List<String> hlShortWords = Util.findAll(words, w->w.contains("h"),
    w->w.contains("l"), w->w.length()<=4);
  System.out.println("Words with h and l and length <= 4: "+hlShortWords);
}
```

LambdaEmployee.java

Word.findAllExample()

Function Default Methods

- Function<T, R> provides two useful default methods to compose functions.

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

- compose: *Function -> Function*
  - a *default method* where f1.compose(f2) means to pass the argument to f2.apply and then the result to f1.apply. Mathematically, f1(f2(x)) or $f1 \circ f2$.
- identity: *() -> Function*
  - a *static method* returning a function that always echoes its input argument (i.e., e->e).

Exercise 5: Composing Multiple Functions

- Write a generic function to compose all the functions in its arguments.

```java
public static<T> Function<T,T> composeAll(Function<T,T>... functions) {
  Function<T,T> result = Function.identity(); // a base fn for compositn
  for (var f: functions)
    result = result.compose(f);
  return result;
}

Function<Double,Double> f = Util.composeAll(Math::rint, Math::sqrt);
```

Stream Operations and Their Composition

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
numbers.stream()
        .map(v -> v*2)
        .forEach(System.out::println);
```

- Stream<T> operations are composed into a pipeline
- They are lazily evaluated
- Seven popular default methods supported by Stream objects:
  - filter(Predicate), map(Function), reduce(BinaryOperator)
  - forEach(), anyMatch(), allMatch(), count()

<R> map: (Function<? super T, ? extends R>) -> Stream<R>

```
public static void main(String[] args) {
    var numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
    applyMap(numbers, v->v*2);
}

public static <T, R> void applyMap(List<T> list, Function<T, R> f) {
    list.stream()
          .map(f)
          .forEach(System.out::println);
}
```

StreamOperation.java
applyMap

filter: Predicate<? super T> -> Stream<T>

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
numbers.stream()
        .filter(v -> v%2==0)
        .forEach(System.out::println);
```

StreamOperation.java
applyFilter

```
List<String> names = Arrays.asList("Bob", "Tom", "Jeff", "Jennifer","Steve");
final Function<String, Predicate<String>> startsWithLetter =
        letter->name->name.startsWith(letter); // lambda with 2 function objs
names.stream()
        .filter(startsWithLetter.apply("J"))
        .forEach(System.out::println);
```

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

<T> reduce(base, (T,T) -> T)

base value to be returned for an empty stream

StreamOperation.java
applyReduce

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
int sum = numbers.stream().reduce(0, (i, j)->i+j);
System.out.println(sum);
```

```
List<String> names = Arrays.asList("Bob", "Tom", "Jeff", "Jennifer", "Steve");
String longestName = names.stream().reduce("", (name1, name2) ->
                    name1.length()>=name2.length()?name1:name2);
System.out.println(longestName);
```

```
double totalSalary = employees.stream().map(e->e.getSalary()).reduce(0.0, (s1, s2)->s1+s2);
System.out.println("Total salary = " + totalSalary);
```

<T> reduce(base, (T,T) -> T)

base value to be returned for an empty stream

```
double totalSalary = employees.stream().map(e->e.getSalary()).reduce(0.0, (s1, s2)->s1+s2);
System.out.println("Total salary = " + totalSalary);
```

```
double totalSalary =
  employees.stream()
                  .map(e->e.getSalary())
                  .reduce(0.0, (s1, s2)->s1+s2);
System.out.println("Total salary = " + totalSalary);
```

anyMatch(), allMatch(), count()

```java
boolean anyMatch = employees.stream().anyMatch(e -> e.getAge()>42);
System.out.println("Is there an employee older than 42? "+anyMatch);

boolean allMatch = employees.stream().allMatch(e -> e.getAge()>42);
System.out.println("Are all employees older than 42? "+allMatch);

long n = employees.stream().filter(e -> e.getAge()>42).count();
System.out.printf("There are %d employees older than 42.\n", n);
```

StreamOperation.java

Making Streams

■ From List ❑ aList.stream()	employees.stream()
■ From object array ❑ Stream.of(anArray)	Employee[] workers = …; Stream.of(workers)
■ From individual values ❑ Stream.of(val1, val2, …)	Employee e1 = …; Employee e2 = …; Stream.of(e1, e2, …)

Converting Streams back to Data Structures

- To List
  - aStream.collect(Collectors.toList())
  - employees = employeeStream.collect(Collectors.toList());
- To object array
  - aStream.toArray(EntryType[]::new)
  - employeeArray = employeeStream.toArray(USTEmployee[]::new);

Computing lambdas with multi-cores and GPUs

```
static void tryParallelStreamOperation() {

  List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
  numbers.parallelStream().forEach(System.out::println);

   employees.parallelStream()
     .filter(e->e.getAge() >= 42)
     .map(e->e.getFirstName()+"")
     .forEach(System.out::println);
}
```

replace stream() with parallelStream()

LambdaEmployee.java
tryParallelStreamOperation

Computing lambdas with multi-cores and GPUs

- The idea is to define computation in lambdas as parallel streams and let the JVM run it on multi-cores and GPUs.
  - http://semiaccurate.com/2013/11/11/amd-charts-path-java-gpu/
  - https://dzone.com/articles/whats-wrong-java-8-part-iii ← *Possible implementation limitation*
- Java 8 is supported since Android Studio 2.1 and Android N.
  - Allow Android apps to leverage the multi-cores and GPU on mobile devices.
  - http://www.androidpolice.com/2016/04/26/android-studio-v2-1-released-to-stable-channel-with-support-for-android-n-java-8-language-features-and-more/

Further Notes on Default Methods

Motivation:

- *Default methods* add new functionality to an existing interface without breaking its implementing classes.

Why default methods in Java 8?

- With the introduction of stream processing, any Iterable objects (e.g., lists, sets, …) can be processed as a stream using forEach:

```
forEach( E e: list ) { … }
```

Further Notes on Default Methods

- Iterable<E> is an interface, which is a super-type of List<E>.

```
public interface Iterator<T> {
  … // abstract methods defined in Java 7
 public default void forEach(Consumer<? super T> action) { … }
  … // other default methods in Java 8
}
```

- Default methods add stream processing functionality to Iterable without breaking all existing classes that have implemented Iterable<T> or its descendants (e.g., List<T>) for the sake of backward compatibility.
- Default methods add functional composition to Predicate<T> using logical operations
- Default methods add functional composition to Function<T,R>
- …

Further Notes on Default Methods

```
interface Vehicle {
  default void print() {
    System.out.println("I am a vehicle.");
  }
}
```

```
class Car implements Vehicle {
  void foo() {
    print();
  }
}
```

- Default methods must be public.
- Default methods cannot be final.

Further Notes on Default Methods

```
interface Vehicle {
  default void print() {
    System.out.println("I am a vehicle.");
  }
}
```

```
class Car1 implements Vehicle {
  public void print() {
    System.out.print("I am a car and ");
    Vehicle.super.print();
  }
}
```

- Default methods can be overridden.
- Default method can be called using InterfaceName.super.

Ambiguous Default Methods – Resolve by Overriding

```java
interface Vehicle {
 default void print() {
  System.out.println("I am a vehicle.");
 }
}
```

```java
interface FourWheeler {
 default void print() {
  System.out.println("I am a four wheeler.");
 }
}
```

```java
class Car3 implements Vehicle, FourWheeler {
 public void print() {
  Vehicle.super.print();
 }
}
```

- Multiple inheritance of default methods can cause ambiguity, which needs to resolve.

Ambiguous Default Methods – Resolve by Overriding

```
interface Vehicle {
 default void print() {
  System.out.println("I am a vehicle.");
 }
}
```

```
interface FourWheeler {
 default void print() { helperFunction(); }
 private void helperFunction() {
  System.out.println("I am a four wheeler.");
 }
}
```

private interface method

```
class Car2 implements Vehicle, FourWheeler {
 public void print() {
  System.out.print("I am a four wheeler car vehicle.\n");
 }
}
```

- Java interface supports private methods to provide helper functions for other interface methods

Static Interface Methods

```
interface Vehicle {
  default void print() {
    System.out.println("I am a vehicle.");
  }
  static void blowHorn() {
    System.out.println("Blowing horn!");
  }
}
```

```
…
Vehicle.blowHorn();
…
```

- Non-abstract interface methods can be static
- They can be called like regular static methods in classes

LambdaDefaultMethod

Static Interface Methods

```java
interface Vehicle {
  default void print() {
    System.out.println("I am a vehicle.");
  }
  static void blowHorn() {
    System.out.println("Blowing horn!");
  }
}
```

```java
…
Vehicle.blowHorn();
…
```

- Interface static methods cannot be inherited
- Why?

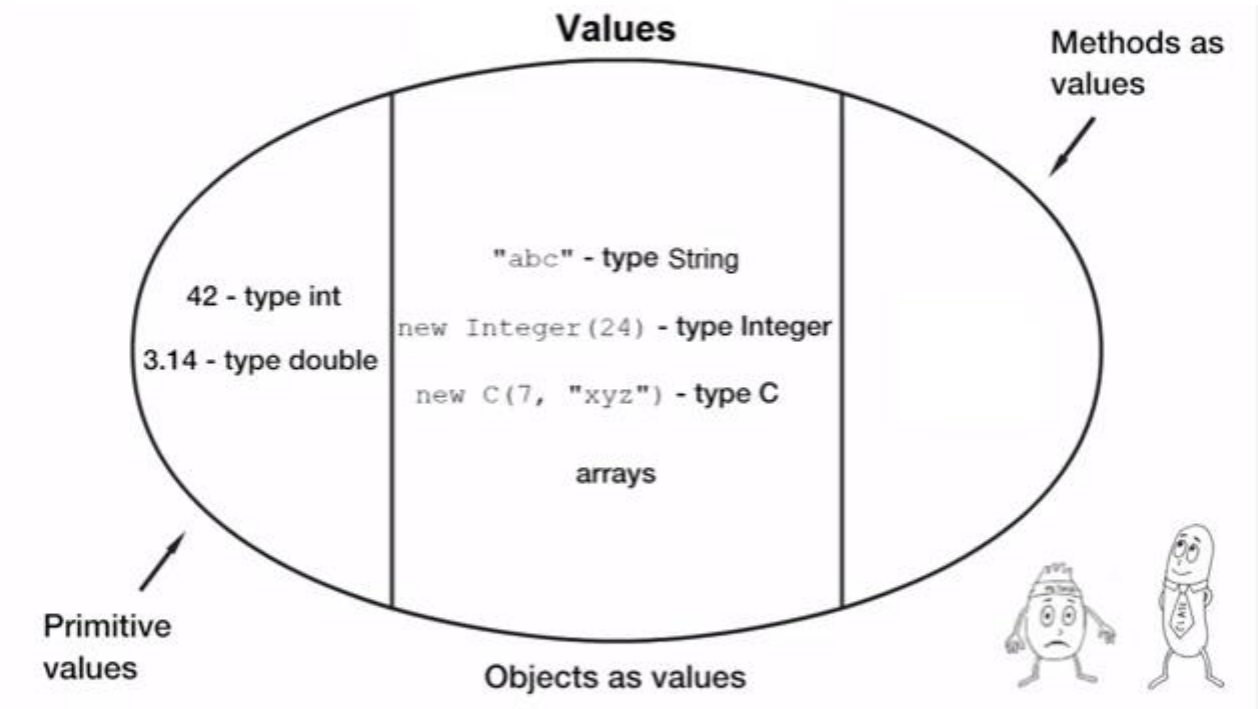
Summary

```java
employees.forEach(e -> e.setSalary(e.getSalary()*1.1));
```

- Functional programming: Define function object in lambda
- Built-in functional interfaces in Java 8
- Generic programming using lambdas
- Higher order functions: composing lambdas using default methods in built-in functional interfaces

http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Video: First-class functions in Java



https://www.youtube.com/watch?v=Rd-sqHjmfB0

THE END OF JAVA LECTURES