

A photograph of a large, two-story, light-colored building with a red-tiled roof, surrounded by green grass and trees. The building has many windows and a central entrance. The text is overlaid on the top half of the image.

**MAHARISHI UNIVERSITY of MANAGEMENT**

*Engaging the Managing Intelligence of Nature*

**Computer Science Department**

**CS401 Modern Programming  
Practices (MPP)  
Bright Gee Varghese**

# Lecture 8: Functional Programming in Java 8

*Commanding All the Laws of Nature from the Source*



# Wholeness Statement

The declarative style of functional programming makes it possible to write methods (and programs) just by declaring *what* is needed, without specifying the details of *how* to achieve the goal. Including support for functional programming in Java makes it possible to write parts of Java programs more concisely, in a more readable way, in a more threadsafe way, in a more parallelizable way, and in a more maintainable way, than ever before.

Maharishi's Science of Consciousness: Just as a king can simply *declare* what he wants – a banquet, a conference, a meeting of all ministers – without having to specify the details about how to organize such events, so likewise can one who is awake to the home of all the laws of nature, the “king” among laws of nature, command those laws and thereby fulfill any intention. The royal road to success in life is to bring awareness to the home of all the laws of nature, through the process of transcending, and live life established in this field.

# Imperative Programming

Begin  
Accept location from user input of either location name or ZIP code.  
Call OpenWeather's Geocoding API to convert location data into geographic coordinates.  
Call OpenWeather's Current Weather Data API.  
Send geographic coordinates to OpenWeather.  
Call OpenWeather's Daily Forecast 16 Days API.  
Resend geographic coordinates.  
Parse JSON returned by the APIs to extract current weather and forecast data.  
Return current weather and forecast.  
Display current weather and forecast to user.  
End

# Declarative Programming

Begin

Location submitted by user is location name or ZIP code.

Location is converted into geographic coordinates.

Weather data is retrieved for geographic coordinates.

Weather data is displayed for user.

End

# Outline

1. **Features and Benefits of Functional Programming**
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables




# Features of Functional Programming

1. Programs are declarative (“what”) rather than imperative (“how”). Makes code more *self-documenting* – the sequence of function calls mirrors precisely the requirements
2. Functions have *referential transparency* – two calls to the same method are guaranteed to return the same result

```
int x = 0;  
int mutate() {  
    ++x;  
    return x;  
}
```

← *lacks referential  
transparency;  
produces a side effect*

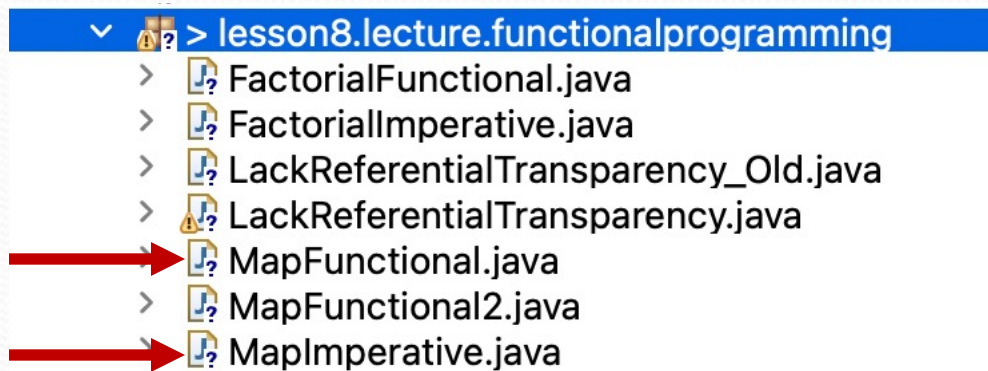
- 
3. Functions do not cause a change of state; in an OO language, this means that functions do not change the state of their enclosing object (by modifying instance variables). In general, functions do not have *side effects*; they compute what they are asked to compute and return a value, without modifying their environment (modifying the environment is a *side effect*). [See example, previous slide]
  4. Functions are *first-class citizens*. This means in particular that it is possible to use functions in the same way objects are used in an OO language: They can be passed as arguments to other functions and can be the return value of a function.



# The Functional Style of Programming (continued)

Demos show examples of adopting a Functional Programming style within Java SE 7. See `lesson8.lecture.functionalprogramming`.

These are not true functional programming examples because they rely on simpler methods that are not purely functional. But these examples illustrate the functional style at the top level. In Java SE 8, these techniques are supported in a truly functional way.



# Benefits of the Functional Style

- Programs are more compact, easier to write, and easier to read/understand
- Programs are thread-safe
- Easier to demonstrate correctness of functional programs
- Easier to test; less likely that a test of a subroutine will fail tomorrow if it passed today since there are no side effects



# Outline

1. Features and Benefits of Functional Programming
2. **Functional Aspects of Java and Functional Interfaces**
3. Lambda Expressions
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables



# Functional Programming in Java

- Java designers – in agreement with many experts – decided it was time to incorporate the many benefits of the functional style of programming into Java
- Java *continues to be an OO language*, with all the benefits of OO for building and maintaining large-scale applications
- For specialized needs – for instance, to extract data from a large collection using SQL-like queries – a functional approach is often more efficient, easier to read, and less error-prone
- Another potential benefit: Significant increase in performance because of automatic support for parallelizing code.

# How Java SE 7 Approximates “Functions As First-Class Citizens”

- Problem: Sort a list of Employee objects.

```
class Employee {  
    String name;  
    int salary;  
    public Employee(String n, int s) {  
        this.name = n;  
        this.salary = s;  
    }  
}
```

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

```
int compare(Employee e1, Employee e2) {  
    return e1.name.compareTo(e2.name);  
}
```



# How Java SE 7 Approximates “Functions As First-Class Citizens”

- Problem: Sort a list of Employee objects.

```
class Employee {  
    String name;  
    int salary;  
    public Employee(String n, int s) {  
        this.name = n;  
        this.salary = s;  
    }  
}
```

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

```
int compare(Employee e1, Employee e2) {  
    return e1.name.compareTo(e2.name);  
}
```

- Natural thing to try to do:

```
Collections.sort(list, compare)
```



# How Java SE 7 Approximates “Functions As First-Class Citizens”

- Problem: Sort a list of Employee objects.

```
class Employee {  
    String name;  
    int salary;  
    public Employee(String n, int s) {  
        this.name = n;  
        this.salary = s;  
    }  
}
```

- To sort, we need a criterion for comparing two Employee objects. We can define a function `compare` that tells us how to do this:

```
int compare(Employee e1, Employee e2) {  
    return e1.name.compareTo(e2.name);  
}
```

- Natural thing to try to do:

```
Collections.sort(list, compare)
```

BUT, *functions are not first-class citizens in Java*, so this cannot be done.

# How Java SE 7 Approximates “Functions As First-Class Citizens”

- The Java solution is to wrap a function or method in an object that implements a simple interface, whose only method is the method we wish to use. (A precise statement about this will be given in an upcoming slide.)
- In this example, the `Comparator` interface can be used as a way to represent the compare function.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```



# How Java SE 7 Approximates “Functions As First-Class Citizens”

- The `Comparator` interface is a *declarative wrapper* for the function `compare`.
- We can use the `compare` function to specify how `Employees` should be compared.
- An instance of this `Comparator` represents the “compare” function as an object.
- Though we can't pass the “compare” function as an argument, we can pass the `Comparator` object:

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
}  
  
public class EmployeeNameComparator  
    implements Comparator<Employee> {  
    @Override  
    public int compare(Employee e1, Employee e2) {  
        return e1.name.compareTo(e2.name);  
    }  
}
```

```
Comparator nameCompare =  
    new EmployeeNameComparator();
```

```
Collections.sort(list, nameCompare);
```



# Functional Interfaces and Functors

- A `Comparator` is called a *functional interface* because it has just one (abstract) method\* (for a precise statement see next slide).
- A class that implements it will contain just one function that implements an abstract method; any instance of this class will be an object that acts like a function.
- An implementation of a functional interface is called a *functor*. Example:

```
public class EmployeeNameComparator implements Comparator<Employee> {  
    @Override  
    public int compare(Employee e1, Employee e2) {  
        return e1.name.compareTo(e2.name);  
    }  
}
```

## (continued)

*Important Observation:* The essential feature of the class `EmployeeNameComparator` is that it represents functional behavior: It associates to each pair  $(e1, e2)$  of `Employees` an integer (indicating an ordering for  $(e1, e2)$ ).

$$(e1, e2) \mapsto \text{val where val} = \begin{cases} > 0 & \text{if } e1.name > e2.name \\ < 0 & \text{if } e1.name < e2.name \\ 0 & \text{if } e1.name \text{ equal to } e2.name \end{cases}$$

\**NOTE:* In reality, `Comparator` declares *two* abstract methods: `compare` and `equals`. However, `equals` already has an implementation in the `Object` class. The precise rule to determine whether an interface is a *functional* interface is that it must have exactly one abstract method, not counting any methods from `Object` that have been re-declared. See

<https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>



# Further Refinement of the Comparator Example

The implementation of the `Comparator` interface shown in the earlier slide has a limitation:

*If the comparison of `Employee` objects requires information stored elsewhere in the class in private variables, we will not be able to implement the `compare` method successfully since it will not have access to those variables.*

*Note:* The difficulty is not apparent in the example but if we change the requirements slightly, this limitation does become apparent (next slide).



# The Need for Closures

Example: If we want to have the choice of sorting by name or by salary, we will need two different Comparators.

```
public class EmployeeSalaryComparator implements
    Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        if(e1.salary == e2.salary) return 0;
        else if(e1.salary < e2.salary) return -1;
        else return 1;
    }
}

public class EmployeeNameComparator implements
    Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name);
    }
}
```

```

public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    private SortMethod method;

    public EmployeeInfo(SortMethod method) {
        this.method = method;
    }
    //Comparators are unaware of the value in method
    public void sort(List<Employee> emps) {
        if(method == SortMethod.BYNAME) {
            Collections.sort(emps, new EmployeeNameComparator());
        }
        else if(method == SortMethod.BYSALARY) {
            Collections.sort(emps, new EmployeeSalaryComparator());
        }
    }
}

```

**Exercise 8.1.** Think of a different way to implement the `sort` method so that only one type of `Comparator` is used. How can you define a `Comparator` so that it will be aware of the value stored in the `method` variable?

```
public void sort(List<Employee> employees) {
```

```
    Collections.sort(  
        employees,
```

```
        new Comparator<Employee>() {
```

```
            @Override
```

```
            public int compare(Employee o1, Employee o2) {
```

```
                if (sortMethod == SortMethod.BYNAME) {
```

```
                    return o1.getName().compareTo(o2.getName());
```

```
                } else {
```

```
                    retun o1.getSalary() - o2.getSalary();
```

```
                }
```

```
            }
```

```
        }
```

```
    );
```

```
}
```



# Creating a Comparator *Closure*

- A *closure* is a block of code that behaves like a function and has access to the variables in its surrounding class.
- In Java 7, instances of member, local, and anonymous inner classes are (essentially) closures, since they have full access to their enclosing object's state.
- As we will see, Java 8's lambda expressions are also (better) examples of closures.
- To solve the Exercise, we create a Comparator that acts as a closure – having full access to its environment.

- Implementing an `EmployeeComparator` using a local inner class allows us to use just one `Comparator`, embedded in the sort method itself:

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    public void sort(List<Employee> emps, final SortMethod method) {
        class EmployeeComparator implements Comparator<Employee> {
            @Override //Comparator is now aware of sort method
            public int compare(Employee e1, Employee e2) {
                if(method == SortMethod.BYNAME) {
                    return e1.name.compareTo(e2.name);
                } else {
                    if(e1.salary == e2.salary) return 0;
                    else if(e1.salary < e2.salary) return -1;
                    else return 1;
                }
            }
        }
        Collections.sort(emps, new EmployeeComparator());
    }
}
```



- NOTE: In Java 7 and before, the method argument

`SortMethod method`

must be declared final since it is referenced in the method body. In Java 8, this is no longer necessary but still the argument may not be modified in the method body.



# Another Functional Interface: Consumer

```
public interface Consumer<T> {  
    public void accept(T t);  
}
```

As discussed in Lesson 7, the `Consumer` interface, like `Comparator`, has just one abstract method. Therefore, it is also a functional interface.

```
public class ListInfo {  
    List<String> list = new ArrayList<>();  
    MyStringList strList = new MyStringList();  
  
    class MyConsumer implements Consumer<String> {  
        public void accept(String t) {  
            strList.add(t);  
        }  
    }  
}  
  
MyConsumer consumer = new MyConsumer();  
list.forEach(consumer);
```

The `consumer` object is another example of a closure. The `accept` method has access to `strList` – a private variable in the enclosing class

# Another Functional Interface: Supplier

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

```
public class EmployeeInfo {  
    private List<Employee> employees;  
    void aMethod() {  
        Supplier<String> supplier = new Supplier<String>() {  
            @Override  
            public String get() {  
                return employees.get(0).getName();  
            }  
        };  
    }  
    public EmployeeInfo(List<Employee> employees) {  
        this.employees = employees;  
    }  
}
```

supplier object is an another example for closure, since it has access to its enclosing class private variable, `employees`.



# Another Functional Interface: ActionListener

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent evt);  
}
```

Typical example from Swing (see Lesson 6 examples)

```
class BrowseButtonListener implements ActionListener {  
    public void actionPerformed(ActionEvent evt) {  
        int selectedRow = table.getSelectedRow();  
        String type = (String)table.getValueAt(selectedRow,0);  
        setVisible(false);  
        ProductListWindow c = new ProductListWindow(type);  
        c.setParentWindow(CatalogListWindow.this);  
        c.setVisible(true);  
    }  
}
```

This inner class is also a closure, and `table` is a variable that is part of the state of the environment.



# Summary: Approximating Functions As First-Class Citizens

- In Java, it is not possible to pass a function as a method argument (like `compare`)
- A realization of a functional interface – a *functor* – is a class that represents a function (recall that a functional interface names only one abstract function). Example: `EmployeeNameComparator` is a functor (implementing the functional interface `Comparator`) that is a wrapper for the function `compare`.
- A *closure* is a functor that is embedded in another class and that is aware of the state of its enclosing class. Example: local inner class `EmployeeComparator` is a wrapper for `compare` and can use data from its enclosing class to perform comparisons.

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    public void sort(List<Employee> emps, final SortMethod method) {
        class EmployeeComparator implements Comparator<Employee> {
            @Override //Comparator is now aware of sort method
            public int compare(Employee e1, Employee e2) {
                if(method == SortMethod.BYNAME) {
                    return e1.name.compareTo(e2.name);
                } else {
                    if(e1.salary == e2.salary) return 0;
                    else if(e1.salary < e2.salary) return -1;
                    else return 1;
                }
            }
        }
        Collections.sort(emps, new EmployeeComparator());
    }
}
```

# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. **Lambda Expressions**
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables



# Lambda Expressions

- Lambda Expressions are closures that may be viewed as syntax shortcuts for Java 7 closures (i.e. inner classes of various kinds). Every lambda expression can be rewritten (using a lot more code) using inner classes.
- Lambdas upgrade the status of (many) functions to first-class citizens.

# History of Lambda Notation

Lambda notation was an invention of the mathematician A. Church in his analysis of the concept of “computable function,” long before computers had come into existence (in the 1930s).

Several equivalent ways of specifying a (mathematical) function:

$f(x, y) = 2x - y$	// this version gives the function a name – namely ‘f’
$(x, y) \mapsto 2x - y$	// in mathematics, this is called “maps to” notation
$\lambda xy. 2x - y$	// Church’s lambda notation
$(x, y) \rightarrow 2 * x - y$	// Java SE 8 lambda notation

**NOTE:** In Church’s lambda notation, the function’s arguments are specified to the left of the dot, and output value to the right.



# Anatomy of a Lambda Expression

A Java lambda expression (like  $(x, y) \rightarrow 2 * x - y$ ) has three parts:

*parameters*      [zero or more]

$\rightarrow$       (arrow)

*code block*      [if more than one statement, enclosed in curly braces  $\{ \dots \}$  ]  
[may contain *free variables*; values for these supplied  
by local or instance variables]



# Using a Lambda Expression for a Consumer

Recall the `Consumer` interface is used with the `forEach` method of `Iterable`

```
public interface Consumer<T> {  
    public void accept(T t);  
}
```

```
Consumer<String> consumer = new Consumer<String>() {  
    @Override  
    public void accept(String s) {  
        System.out.println(s);  
    }  
};  
System.out.println("-----using new forEach method-----");  
l.forEach(consumer);
```

can be rewritten as

```
System.out.println("-----using new forEachMethod-----");  
l.forEach(s -> System.out.println(s));
```

We say that `s -> System.out.println(s)` *implements* the `Consumer` interface.

**NOTE:** Passing `s -> System.out.println(s)` as an argument to `l.forEach` is essentially the same as passing in the `accept` method.

# Example: the Function $(x,y) \rightarrow 2 * x - y$

Java SE 8 offers new functional interfaces to support many of the lambda expressions that could arise (though not all).

The `BiFunction<S, T, R>` interface has as its unique abstract method `apply()`, which returns the result of applying a function to its first two arguments (of type `S`, `T`) to produce a result (of type `R`).

```
public interface BiFunction<S, T, R> {  
    R apply(S s, T t);  
}
```



This code uses lambda notation to express functional behavior.

```
public static void main(String[] args) {  
    BiFunction<Integer, Integer, Integer> f =  
        (x,y) -> 2 * x - y;  
    System.out.println(f.apply(2, 3));    //output: 1  
}
```

The lambda `(x,y) -> 2 * x - y` implements the `BiFunction` interface

Without lambdas (Java 7 – style):

```
public static void main(String[] args) {  
    class MyBiFunction implements  
        BiFunction<Integer, Integer, Integer> {  
        public Integer apply(Integer x, Integer y) {  
            return 2 * x - y;  
        }  
    }  
    MyBiFunction f = new MyBiFunction();  
    System.out.println(f.apply(2, 3)); // output 1  
}
```

# Another Closure/Lambda Example:

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    private SortMethod sortMethod = SortMethod.BYNAME;
    public void sort(List<Employee> emps, final SortMethod method) {
        class EmployeeComparator implements Comparator<Employee> {
            @Override
            public int compare(Employee e1, Employee e2) {
                if(method == sortMethod) {
                    return e1.name.compareTo(e2.name);
                } else {
                    throw new IllegalArgumentException("Cannot compare");
                }
            }
        }
        Collections.sort(emps, new EmployeeComparator());
    }
}
```

//Using lambdas:

```
public class EmployeeInfo {
    static enum SortMethod {BYNAME, BYSALARY};
    private SortMethod sortMethod = SortMethod.BYNAME;
    public void sort(List<Employee> emps, SortMethod method) {
        Collections.sort(emps, (e1,e2) ->
        {
            if(method == sortMethod) {
                return e1.name.compareTo(e2.name);
            } else {
                throw new IllegalArgumentException("Cannot compare");
            }
        });
    }
}
```



# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. **Parameters and Free Variables**
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables

# Parameters and Free Variables

- *Parameters* are the arguments that are passed as arguments in the lambda expression: In  $(x, y) \rightarrow x * y$ , the variables  $x, y$  are parameters
- *Free variables* are variables in the lambda expression that are *not* parameters and *not* defined inside the block of code on the right hand side of the lambda expression

```
//parameters e1, e2; free variables sortMethod and method
```

```
Comparator c =(Employee e1, Employee e2) -> {  
    if (sortMethod.equals(method)) {  
        return e1.name.compareTo(e2.name);  
    } else {  
        throw new IllegalArgumentException("can't sort");  
    }  
}
```



## (continued)

- In order for a lambda expression to be evaluated, values for the free variables need to be supplied (either as arguments in the method in which the lambda expression appears or in the enclosing class).
- Values that are supplied to the free variables are said to be *captured by the lambda expression*

# More Examples

```
//handle method in EventHandler:  
//  parameter: evt, free vbles: out and username  
(ActionEvent evt) -> System.out.println(  
    "Hello " + (username != null ? username : "World") + "!" );
```

Note: `out` is a variable in the `System` class, which is a reference to standard output (typically, the console)

The value of `out` is supplied from outside the expression, so is also a free variable



```

JButton button = new JButton("ok");
button.addActionListener(
    new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            //Write the action
        }
    }
);

```



```

button.addActionListener(
    (eventObject) -> {
        //Write the action
    }
);

```

# Creating Your Own Functional Interface:

Demo: `lesson8.lecture.lambdaexamples.trifunction`

```
@FunctionalInterface
public interface TriFunction<S,T,U,R> {
    R apply(S s, T t, U u);
}

public static void main(String[] args) {
    TriFunction<Integer, Integer, Integer, Integer> f =
        (x, y, z) -> x + y + z;
    System.out.println(f.apply(2, 3, 4)); //output: 9
}
```

## Notes:

- The `@FunctionalInterface` annotation is checked by the compiler – if the interface does not contain exactly one abstract method, there is a compiler error.
- Not required, but strongly recommended, to enable compile-time checking.



# Exercise 8.2

1. Is this a functional interface?  
Explain.

```
public interface Example1 {  
    String toString();  
}
```

2. Does the following code compile?  
Explain

```
@FunctionalInterface  
public interface Example2 {  
    String toString();  
    void act();  
}
```

3. Define your own functional interface that could be used as a wrapper for a function that takes zero arguments and returns an integer. Then use it to define an object that produces random numbers. In a main method test your code; then try to represent your functional interface with a lambda

```
@FunctionalInterface  
interface MyIface {  
  
    }  
class MyRandGen implements MyIface {  
    ...  
    public static void main(String[] args) {  
        MyIface mrg = new MyRandGen();  
        System.out.println(mrg.produce());  
    }  
}
```

# Exercise 8.2 Solution

1. Is this a functional interface?

Explain.

```
public interface Example1 {  
    String toString();  
}
```

**No – we don't count methods that override methods from Object**

2. Does the following code compile?  
Explain

```
@FunctionalInterface  
public interface Example2 {  
    String toString();  
    void act();  
}
```

**Yes, there is just one abstract method not counting methods from Object**

3. Define your own functional interface that could be used as a wrapper for a function that takes zero arguments and returns an integer. Then use it to define an object that produces random numbers. In a main method test your code; then try to represent your functional interface with a lambda

```
public interface MyIface {  
    int produce();  
}
```

```
public class MyRandGen implements MyIface {  
    public int produce() {  
        Random r = new Random();  
        return r.nextInt();  
    }  
}
```



```
@FunctionalInterface
public interface MyIface {
    int produce();
}
```

```
import java.util.Random;

public class MyRandGenWithLambda {

    public static void main(String[] args) {
        MyIface myIface = () -> new Random().nextInt(10);
        System.out.println(myIface.produce());
    }
}
```

int value between 0 (inclusive) and the specified value (exclusive)

# Main Point 1

In Java, before Java SE 8, functions were not first-class citizens, which made the functional style difficult to implement. Prior to Java SE 8, Java approximated a function with a functional interface; when implemented as an inner class, objects of this type were close approximations to functions. In Java SE 8, these inner class approximations can be replaced by lambda expressions, which capture their essential functional nature: *Arguments mapped to outputs*. With lambda expressions, it is now possible to reap many of the benefits of the functional style while maintaining the OO essence of the Java language as a whole.

The “purification” process that made it possible to transform “noisy” one-method inner classes into simple functional expressions (lambdas) is like the purification process that permits a noisy nervous system to have a chance to operate smoothly and at a higher level. This is one of the powerful benefits of the transcending process.



# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. Parameters and Free Variables
5. **A Sample Application of Lambdas**
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables

# A Sample Application of Lambdas

Task: Extract from a list of names (Strings) a sublist containing those names that begin with a specified character, and transform all letters in such names to uppercase.

## Imperative Style (Java 7)

```
public List<String> findStartsWithLetterToUpper(  
    List<String> list, char c) {  
    List<String> startsWithLetter = new ArrayList<String>();  
    for(String name : list) {  
        if(name.startsWith("" + c)) {  
            startsWithLetter.add(name.toUpperCase());  
        }  
    }  
    return startsWithLetter;  
}
```



# A Sample Application of Lambdas (cont.)

## Using Lambdas and Streams (Java 8)

(Note: Streams will be discussed fully in Lesson 9)

```
public List<String> findStartsWithLetter(List<String> list, String letter) {  
    return list.stream() // convert list to stream  
        .filter(name -> name.startsWith(letter)) // returns filtered  
                                                    // stream  
        .map(name -> name.toUpperCase()) // maps each string to upper  
                                           // case string  
        .collect(Collectors.toList()); // organizes into a list  
}
```

```
// parallel processing  
public List<String> findStartsWithLetter(List<String> list, String letter) {  
    return list.parallelStream() // convert list to stream  
        .filter(name -> name.startsWith(letter)) // returns filtered  
                                                    // stream  
        .map(name -> name.toUpperCase()) // maps each string to upper  
                                           // case string  
        .collect(Collectors.toList()); // organizes into a list  
}
```

# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. **Naming Lambda Expressions**
7. Syntax Shortcuts and Method Expressions
8. Restricted Use of Free Variables



# Naming Lambda Expressions

1. We want to be able to reuse lambda expressions rather than rewriting the entire expression each time. To do so, we need to give it a name and a type.
2. Every object in Java has a type; the same is true of lambda expressions.

*The type of a lambda expression is any functional interface for which the lambda expression is an implementation!*

# Naming Lambda Expressions (cont.)

**Example:** Consider this lambda expression:

```
(Employee e1, Employee e2) ->  
    e1.getName().compareTo(e2.getName());
```

What functional interface has been implemented here?

**Analysis.** The lambda accepts two parameters – e1 and e2 – and has a return value of type Integer. We must find (or create) a functional interface whose encapsulated function has these characteristics.

**How to Search.** The new functional interfaces in Java 8 are contained in the package `java.util.function`. We can look at the JDK API docs at <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html> to locate a matching interface.



# Naming Lambdas (continued)

## Interface Summary

Interface	Description
<b>BiConsumer</b> <T,U>	Represents an operation that accepts two input arguments and returns no result.
<b>BiFunction</b> <T,U,R>	Represents a function that accepts two arguments and produces a result.
<b>BinaryOperator</b> <T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<b>BiPredicate</b> <T,U>	Represents a predicate (boolean-valued function) of two arguments.
<b>BooleanSupplier</b>	Represents a supplier of boolean-valued results.
<b>Consumer</b> <T>	Represents an operation that accepts a single input argument and returns no result.

# Naming Lambdas (continued)

## Interface Summary

Interface	Description
<b>BiConsumer</b> <T,U>	Represents an operation that accepts two input arguments and returns no result.
<b>BiFunction</b> <T,U,R>	Represents a function that accepts two arguments and produces a result.
<b>BinaryOperator</b> <T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<b>BiPredicate</b> <T,U>	Represents a predicate (boolean-valued function) of two arguments.
<b>BooleanSupplier</b>	Represents a supplier of boolean-valued results.
<b>Consumer</b> <T>	Represents an operation that accepts a single input argument and returns no result.

## First Candidate

```
BiFunction<Employee, Employee, Integer> lambda  
    = (Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```



# Naming Lambdas (continued)

- *Another Possibility.* Recall that the `compare` method of a `Comparator` accepts two arguments and returns an integer – positive integer means “greater than”, negative integer means “less than”.

## Second Candidate

```
Comparator<Employee> lambda  
    = (Employee e1, Employee e2) -> e1.getName().compareTo(e2.getName());
```

- The Second Candidate is also correct. The lambda can be typed in either way.

# Naming Lambdas (continued)

**Question:** Which of the two candidates should be used?

**Answer:** It depends on how you intend to use the lambda. In this case, it is likely that the lambda is designed as a `Comparator` for sorting. Only the `Comparator` type can be used for this purpose

```
Comparator<Employee> lambda1  
    = (e1, e2) -> e1.getName().compareTo(e2.getName());  
BiFunction<Employee, Employee, Integer> lambda2  
    = (e1, e2) -> e1.getName().compareTo(e2.getName());
```

```
public List<Employee> sort(List<Employee> list) {  
    Collections.sort(list, lambda1);  
    return list;  
}
```

↖  
This code works

```
public List<Employee> sort2(List<Employee> list) {  
    Collections.sort(list, lambda2);  
    return list;  
}
```

↖  
This code does *not* work



# Naming Lambdas (Continued)

- [Optional] NOTE: Although every lambda is a realization of a functional interface, the way in which the Java compiler translates a lambda into a realization of such an interface is *not* obvious. Historically, the possibility of simply translating the lambda into an anonymous inner class was considered by the Java engineers, but was rejected for a number of reasons. One reason is performance – inner classes have to be loaded separately by the class loader. Another is that tying lambdas to such an implementation would limit the possibility for evolution of new features of lambdas in future releases. You can verify that lambdas and anonymous inner classes are fundamentally different (even though very similar) by considering how the implicit object reference 'this' is interpreted by each type: In an anonymous inner class, 'this' refers to the inner class; in a lambda, 'this' refers to the surrounding class. See <http://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood>

# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. **Syntax Shortcuts and Method Expressions**
8. Restricted Use of Free Variables



# Syntax Shortcuts via Target Typing

## 1. If parameter types can be inferred, they can be omitted

```
Comparator<Employee> empNameComp = (e1, e2) -> {  
    if (method == SortMethod.BYNAME) {  
        return e1.name.compareTo(e2.name);  
    } else {  
        if (e1.salary == e2.salary) return 0;  
        else if (e1.salary < e2.salary) return -1;  
        else return 1;  
    }  
}
```

//sort expects a Comparator; since types in emps list are Employee, infer type Comparator<Employee>

```
Collections.sort(emps, (e1, e2) -> {  
    if (method == SortMethod.BYNAME) {  
        return e1.name.compareTo(e2.name);  
    } else {  
        if (e1.salary == e2.salary) return 0;  
        else if (e1.salary < e2.salary) return -1;  
        else return 1;  
    }  
});
```

See DEMO: [lesson8.lecture.lambdaexamples.comparator3](#)

# Syntax Shortcuts via Target Typing (cont.)

2. If a lambda expression has a single parameter with an inferred type, the parentheses around the parameter can be omitted.

```
Consumer consumer = str -> {    //instead of (str)
    System.out.println(str);
};
```

```
ActionListener<ActionEvent> listener = evt -> {
    System.out.println("Hello World");
};
```




# Syntax Shortcuts via Target Typing(cont.)

- 3. *Method References.* (See Lesson 9 for a fourth type of method reference – *constructor reference*)
- A. Type: *object::instanceMethod*. Given an object *ob* and an instance method *meth()* in *ob*, the lambda expression  
$$x \rightarrow ob.meth(x)$$
can be written as  
$$ob::meth$$

Example (see `SimpleButtonSwing` in  
`lesson8.lecture.methodreferences.objinstance.print`)

Rewrite

```
button.addActionListener(evt -> p.print(evt));  
as  
button.addActionListener(p::print);
```



```
List<String> laptops = Arrays.asList("Del", "Samsung", "NVIDIA");  
laptops.forEach(laptop -> System.out.println(laptop));  
//or  
laptops.forEach(System.out::println);
```



```
public class ShortDemo {  
    int myFun(int data) {  
        return data * 2;  
    }  
}
```

```
public static void main(String[] args) {  
    ShortDemo demo = new ShortDemo();  
    MyIFace face = data -> demo.myFun(data);  
    System.out.println(face.fun(23));  
}
```

```
    MyIFace face1 = demo::myFun;  
    System.out.println(face1.fun(2));  
}
```

```
@FunctionalInterface  
interface MyIFace {  
    double fun(int data);  
}
```

```
MyIFace face2 = new MyIFace() {  
    @Override  
    public double fun(int data) {  
        return demo.myFun(data);  
    }  
};  
System.out.println(face2.fun(23));
```

# Syntax Shortcuts via Target Typing(cont.)

- B. Type: *Class::staticMethod*. Given a class `ClassName` and one of its static methods `meth()`, the lambda expression

```
x -> ClassName.meth(x)
```

(or `(x, y) -> ClassName.meth(x, y)` if `meth` accepts two arguments)

can be rewritten as

```
ClassName::meth
```

## Example

(see `MethodRefMath` demo in `lesson8.lecture.methodreferences.classmethod.math`)

Rewrite

```
BiFunction<Integer, Integer, Double> f  
    = (x, y) -> Math.pow(x, y);
```

as

```
BiFunction<Integer, Integer, Double> f = Math::pow;
```



# Syntax Shortcuts via Target Typing(cont.)

- C. Type: *Class::instanceMethod*. Given a class *ClassName* and one of its instance methods *meth()*, the lambda expression

$$(x, y) \rightarrow x.meth(y)$$

can be rewritten as

$$ClassName::meth$$

Example:

$$(str1, str2) \rightarrow str1.compareToIgnoreCase(str2)$$

can be written as

$$String::compareToIgnoreCase$$

```
List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);  
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
```

*//Or*

```
Collections.sort(numbers, Integer::compareTo);
```

```
Function<String, String> function = s -> s.toUpperCase();
```

*// or*

```
// Function<String, String> function = String::toUpperCase;  
System.out.println(function.apply("mpp"));
```



# Outline

1. Features and Benefits of Functional Programming
2. Functional Aspects of Java and Functional Interfaces
3. Lambda Expressions
4. Parameters and Free Variables
5. A Sample Application of Lambdas
6. Naming Lambda Expressions
7. Syntax Shortcuts and Method Expressions

# Exercise 8.3

## *Lambda and Method Reference Exercises*

(see PDF in package lesson8.exercise\_3 in InClassEx's)

Assign each lambda expressions to a variable of the appropriate type and then express as a method reference. Indicate which type of method reference you are using.

```
Example: (String x) -> x.toUpperCase()
```

```
Function<String, String> toUpper1 = (String x) -> x.toUpperCase();  
Function<String, String> toUpper2 = String::toUpperCase;  
    Method reference type: Class::instanceMethod
```



# Connecting the Parts of Knowledge With the Wholeness of Knowledge

*Declarative programming and command of all the laws of nature*

1. In Java SE 7, the only first-class citizens are objects, created from classes. The valuable techniques of functional programming and a declarative style can be approximated using functional interfaces.
  2. In Java SE 8, functions – in the form of lambda expressions – have become first-class citizens, and can be passed as arguments and occur as return values. In this new version, the advantage of functional programming with its declarative style is now supported in the language
- 
3. **Transcendental Consciousness:** TC, which can be experienced in the stillness of one's awareness through transcending, is where the laws of nature begin to operate – it is the *home of all the laws of nature*
  4. **Impulses Within the Transcendental Field:** As TC becomes more familiar, more and more, intentions and desires reach fulfillment effortlessly, because of the hidden support of the laws of nature.
  5. **Wholeness moving within Itself:** In Unity Consciousness, one finally recognizes the universe in oneself – that all of life is simply the impulse of one's own consciousness. In that state, one effortlessly commands the laws of nature for all good in the universe.