LESSON – 8

Functional Programming in Java

- **Wholeness of the Lesson:** 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.

The Functional Style of Programming

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

Functional Style Programming Principles

1. Programs are Declarative

Programs are declarative (“what”) rather than imperative (“how”). Makes code more *self-documenting* – the sequence of function calls mirrors precisely the requirements.

```
// Program to get the list of names starts with "M"

List<String> names = Arrays.asList(“Joe", “Sandy", “Andy", “John",“Bruen");
```

Imperative style

```
List<String> list = new ArrayList();
for (String name : names) {
if (name.startsWith(“J"))
list.add(name);
}
```

Declarative Style

```
List<String> filternames = names.stream()   // Convert list into stream
.filter(name -> name.startsWith(“J"))       // return the filtered stream
.collect(Collectors.toList());              // organizes into a list
```

Demo : java8featurepack.imperativedeclarative Package

2. Avoid Mutable states

- The principle is the use of *immutable* values.
- Example : famous *Pythagorean equation*
  - $x^2 + y^2 = z^2$ (apply $x = 3$, $y = 4$ and get $z = 5$)
- Benefits
  - If we make a value immutable, the synchronization problem disappears in multithreaded programming. Concurrent reading is harmless, so multithreaded programming becomes far easier.
  - Immutable values relates to program correctness. Testing becomes easy.
- Functional programming encourages us to think strategically about when and where mutability is necessary.

3. Functions as First class values

- Prior Java 8
  - In Java, we are accustomed to passing objects and primitive values to methods, returning them from methods, and assigning them to variables. This means that objects and primitives are *first-class values* in Java. Methods and classes are second class values.
- In Java 8
  - Possible to pass functions as values. So Functions are first class value.

4. Side effect free functions

- 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. 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*).
- Functions have *referential transparency* – two calls to the same method are guaranteed to return the same result.
- In mathematics, functions never have side effects, meaning they are *side-effect-free*. For example, no matter how much work sin(x) has to do, its entire result is returned to the caller. No external state is changed.
- "renuka".replace("r", "R"); // always produce Renuka. It is Referential Transparency.
- Random.nextInt() always produce different result. It is not a Referential Transparency.

5. Design with Higher order functions

- There is a special term for functions that take other functions as arguments or return them as results: *higher-order functions*.
- Java methods are limited to primitives and objects as arguments and return values, but we can mimic this feature with our Function interfaces.
- Higher-order functions are a powerful tool for building abstractions and composing behavior.
- Example : Functional Interface

Benefits

- Allows us to write easier-to-understand, more declarative, more concise programs than imperative programming
- Allows us to focus on the problem rather than the code
- Facilitates parallelism
- Thread safe.

Introducing Lambda Expressions

- Lambda notation was an invention of the mathematician Alonzo Church. 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
- NOTE: In Church's lambda notation, the function's arguments are specified to the left of the dot, and output value to the right.
- $\lambda xy.2x - y$ //Church's lambda notation
- (x,y) -> 2*x – y // Java SE 8 lambda notation

Lambdas

- A lambda expression is a block of code with a list of formal parameters and a body. Sometimes a lambda expression is simply called a *lambda*.
- The body of a lambda expression can be a block statement or an expression.
- An arrow (->) is used to separate the list of parameters and the body.
- The term “lambda” in "lambda expression" has its origin in Lambda calculus that uses the Greek letter lambda “λ” to denote a function abstraction.
- A lambda expression describes an anonymous function. The general syntax for using lambda expressions is very similar to declaring a method. The basic syntax of a lambda is either

  (parameters) -> expression ;

  or (note the curly braces for statements)

  (parameters) -> { statements; }

- Examples

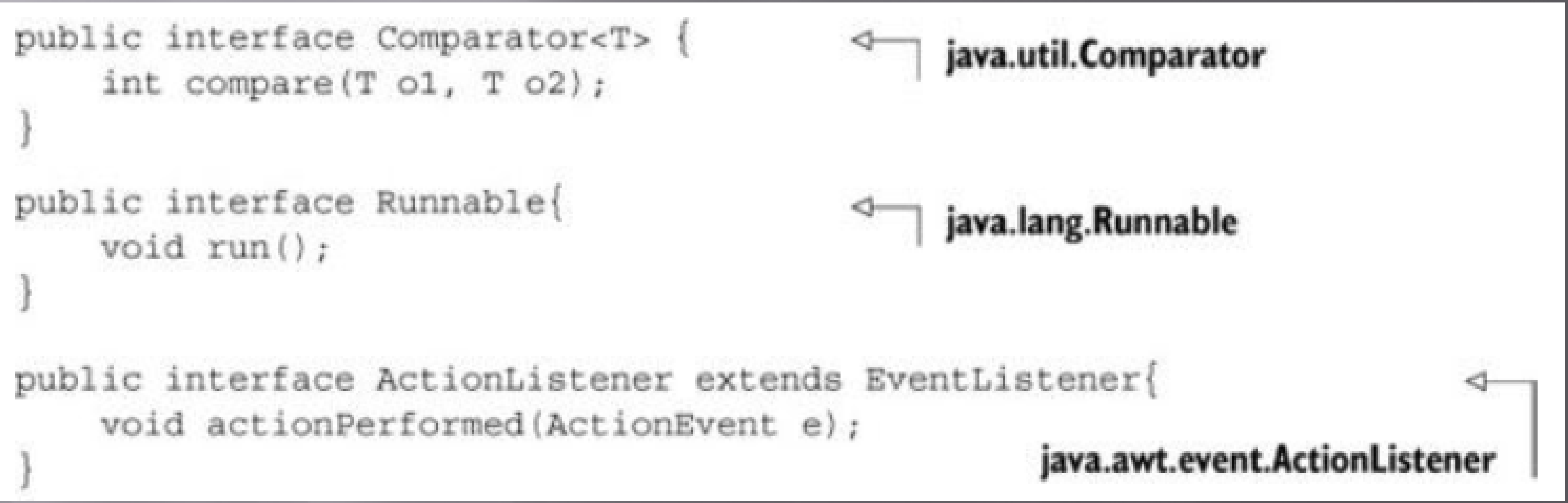
```
// Takes an int parameter and returns the parameter value
incremented by 1
(int x) -> x + 1;   (or)   (x) -> x+1;
// Takes two int parameters and returns their sum
(int x, int y) -> x + y;  (or) (x,y) ->x + y;
// Takes two int parameters and returns the maximum of the
two
(int x, int y) -> { int max = x > y ? x : y; }
// Takes no parameters and returns a string "OK"
() -> "OK“
// Takes a parameter and prints it on the standard output
(msg) -> System.out.println(msg)
// Takes a String parameter and returns its length
(String str) -> str.length()
```

Lambdas	Equivalent Java method
`() -> 6;`	`int getValue(){ return 6; }`
`(String s1, String s2) -> s1+s2;`	`String join(String s1, String s2) { return s1 + s2;}`
`n -> n % 2 != 0;`	`boolean Even(int n){ if (n % 2 !=0) return false; else return true; }`
`(String s) -> { System.out.println(s); };`	`void print(String s){ System.out.println(s); }`

Note : In Lambdas Parameter comes in left side, body of the part comes as right side, separated by -> symbol

Functional Interface Implementation using Lambdas

- ***A functional interface*** is an interface that specifies exactly one abstract method.
- You already know several other functional interfaces in the Java API such as Comparator and Runnable



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

public interface Runnable{
    void run();
}

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

- Every expression in Java has a type; so does a lambda expression. The type of a lambda expression is a functional interface type.
- When the abstract method of the functional interface is called, the body of the lambda expression is executed.
- See Demo : java8featurepack.lambda

Example

Anonymous Way

```
Collections.sort(obj,new Comparator<Person>(){
@Override
public int compare(Person o1, Person o2) {
return o1.getName().compareTo(o2.getName());
}
});
```

Using Lambdas

```
Collections.sort(obj, (e1,e2) ->e1.getName().compareTo(e2.getName()));
```

Example

```
// Without Lambda
button.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
 System.out.println("Process Print");
 } });
// Using Lambda
button.addActionListener(
evt -> {
                System.out.println("Process Print");
        });
```

Example

```
@FunctionalInterface
interface StringToIntMapper {
int map(String str);
}
```

StringToIntMapper mapper = (String str) -> str.length();

- In this statement, the compiler finds that the right-hand side of the assignment operator is a lambda expression.
- To infer its type, it looks at the left-hand side of the assignment operator that expects an instance of the StringToIntMapper interface; it verifies that the lambda expression conforms to the declaration of the map() method in the StringToIntMapper interface;
- finally, it infers that the type of the lambda expression is the StringToIntMapper interface type.
- When you call the map() method on the mapper variable passing a String, the body of the lambda expression is executed as shown in the following snippet of code:

```
StringToIntMapper mapper = (String str) -> str.length();
String name = "Kristy";
int mappedValue = mapper.map(name);
System.out.println("name=" + name + ", mapped value=" + mappedValue);
```

Target Typing

- Functional interfaces that allow for "target typing"; they provide enough information for the compiler to infer argument and return types.
- The general improvements to type inference in Java 8 mean that lambdas can infer their type parameters; so rather than use

```
(Integer x, Integer y) -> x + y;
```

- you can drop the Integer type annotation and use the following instead.

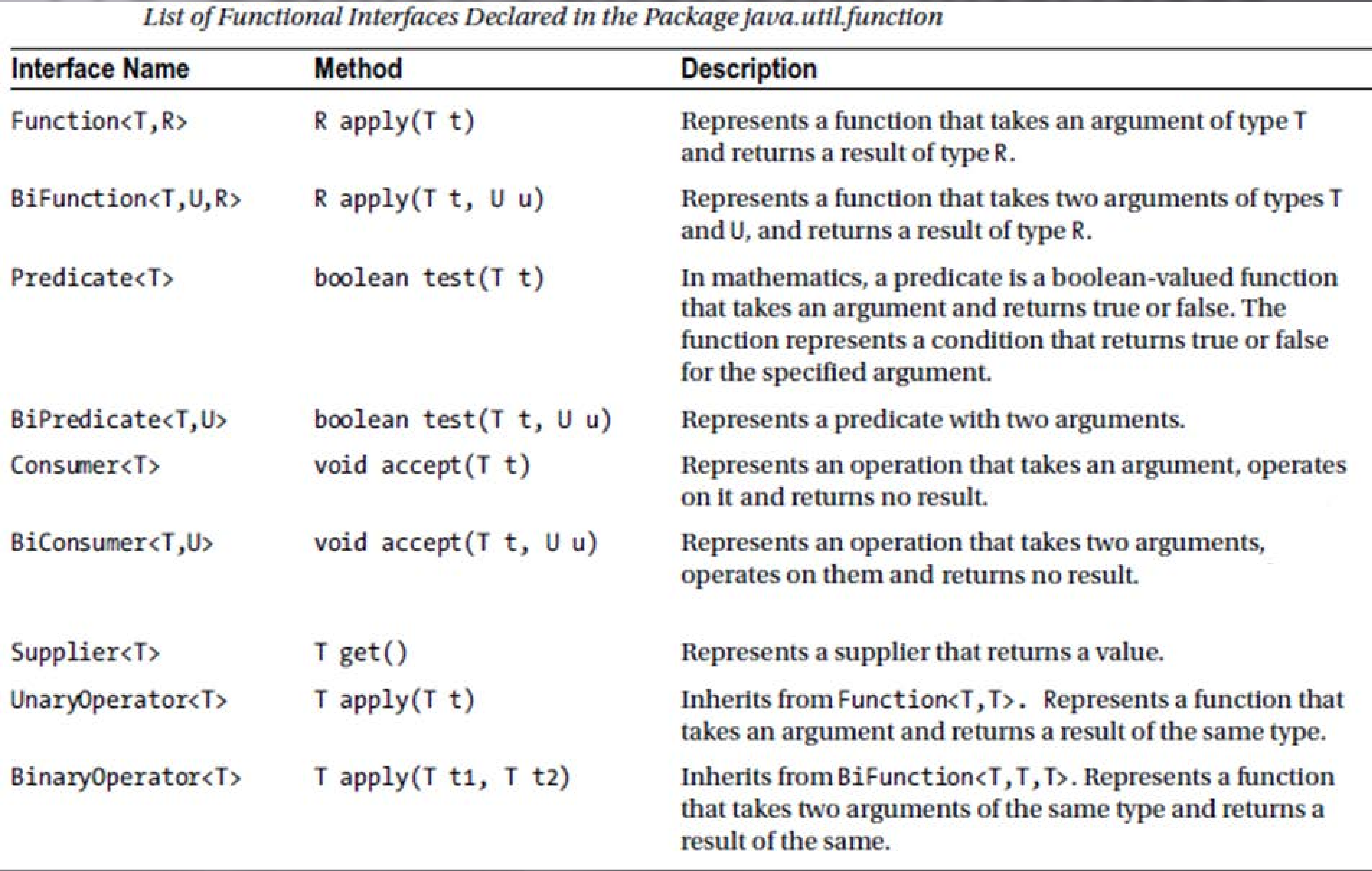
```
(x, y) -> x + y;
```

- This is because the functional interface describes the types, it gives the compiler all the information it needs.
- For example, if we take an example functional interface.

```
@FunctionalInterface
interface Calculation {
Integer apply(Integer x, Integer y);
}
```

- When a lambda is used in-lieu of the interface, the first thing the compiler does is work out the "target" type of the lambda.

Pre-Defined Functional Interfaces



List of Functional Interfaces Declared in the Package java.util.function

Interface Name	Method	Description
`Function<T,R>`	`R apply(T t)`	Represents a function that takes an argument of type `T` and returns a result of type `R`.
`BiFunction<T,U,R>`	`R apply(T t, U u)`	Represents a function that takes two arguments of types `T` and `U`, and returns a result of type `R`.
`Predicate<T>`	`boolean test(T t)`	In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument.
`BiPredicate<T,U>`	`boolean test(T t, U u)`	Represents a predicate with two arguments.
`Consumer<T>`	`void accept(T t)`	Represents an operation that takes an argument, operates on it and returns no result.
`BiConsumer<T,U>`	`void accept(T t, U u)`	Represents an operation that takes two arguments, operates on them and returns no result.
`Supplier<T>`	`T get()`	Represents a supplier that returns a value.
`UnaryOperator<T>`	`T apply(T t)`	Inherits from `Function<T,T>`. Represents a function that takes an argument and returns a result of the same type.
`BinaryOperator<T>`	`T apply(T t1, T t2)`	Inherits from `BiFunction<T,T,T>`. Represents a function that takes two arguments of the same type and returns a result of the same.

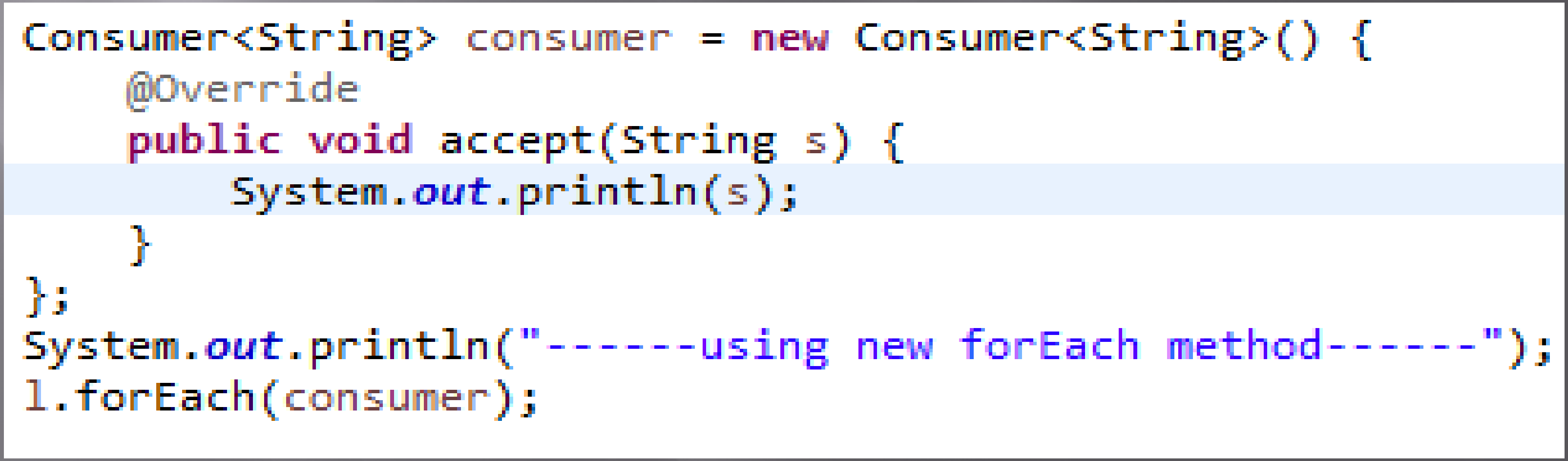
See More on https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html Demo : java8featurepack.lambda.function package

Examples

Example 1 : Function Interface

```
// Lambda Implementation for apply method
Function<Integer, Double> milesToKms = (input) -> 1.6 * input;
        int miles = 3;
        double kms = milesToKms.apply(miles);
        System.out.printf("%d miles = %3.2f kilometers\n", miles, kms);
```

Example 2 : Consumer Interface



```
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);
```

Example 3 : Predicate Interface

```
public class PredicateDemo1 {
    public static void main(String[] args) {
        Predicate<String> numbersOnly = (input) -> {
            for (int i = 0; i < input.length(); i++) {
                char c = input.charAt(i);
                if ("0123456789".indexOf(c) == -1) {
                    return false;
                }
            }
            return true;
        };

        System.out.println(numbersOnly.test("12345"));// true
        System.out.println(numbersOnly.test("100a")); // false
    }
}
```

Quiz 1

1. Choose any one suitable Functional Interface to reverse a String and do implementation using lambdas.

Quiz 2 : Say Valid or Invalid Functional Interface

```
@FunctionalInterface
interface B {
abstract void apply();
String toString();
}
```

```
@FunctionalInterface
interface FunctionalInterfaceExample2 {
void apply();
void illegal();
}
```

```
@FunctionalInterface
interface FunctionalInterfaceExample1 {
void show();
}
```

```
public interface Calculator {

   double calculate(int a, int b);
}
```

```
@FunctionalInterface
interface A {
abstract void apply();
}
```

```
interface C {
void apply();
int hashCode();
}
```

Lambdas and closures

- Lambdas
  - JDK 8 will introduce a syntax for defining *anonymous functions*, also called *lambdas.* It comes from the use of the Greek lambda symbol λ to represent functions in *lambda calculus*.

- **Free Variables and Closures**
  - Free variables are variables that are *not* parameters and *not* defined inside the block of code (on the right hand side of the lambda expression – Lambda body)
  - In order for a lambda expression to be evaluated, values for the free variables need to be supplied (either by the method in which the lambda expression appears or in the enclosing class). These values are said to be *captured by the lambda expression.*
  - A *closure* in Java can be defined to be a block of code on the right hand side of a lambda expression, together with the values of the free variables in that block.

Examples

```
//compare in Comparator: two parameters e1, e2; 1 free variable method
 (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;
}
}
//accept in Consumer: one parameter str; no free vbles
(String str) -> System.out.println(str);
//handle in EventHandler: one parameter evt, one free vble username
(ActionEvent evt) ->
System.out.println("Hello " + (username != null ? username : "World") + "!");
```

Find Free Variables

Example 1 :

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
      while (count > 0) {
        count--; // Error: Can't mutate captured variable
        System.out.println(text);
        Thread.yield();
      }
    };
    new Thread(r).start();
  }
```

Example 2 :

```
      double  c = 1.6;
      Function<Integer, Double> milesToKms = (input) -> c * input;
```

Example 3 :

```
StringToIntMapper mapper = (String str) -> str.length();
```

Sorting – Various Implementation

- Example: Suppose we want to sort a list of Employee objects.

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

Comparator Interface

- The Comparator interface is a *declarative wrapper* for the function compare.

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

- It is called a *functional interface* because it has just one (abstract) method. So a class that implements it will have in effect just one implemented function; it 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);
        }
}
```

NOTE: Though EmployeeNameComparator is a class, it is essentially just a function that associates to each pair (e1,e2) of Employees an integer (indicating an ordering for e1, e2).

Using Inner Class

- The implementation of the Comparator interface shown in the previous slide has a limitation: If the way the compare method acts depends on the state of the class that is attempting to sort Employee objects, our Comparator implementation will never be aware of this fact. (This is not a big problem in this case but can be in more complex settings.)

  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);
        }
}
```

EmployeeInfo Class

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

        public EmployeeInfo(SortMethod method) {
                this.method = method;
        }
        //The Comparators are unaware of the choice of sort 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());
                }
        }

        public static void main(String[] args) {
                List<Employee> emps = new ArrayList<>();
                emps.add(new Employee("Joe", 100000));
                emps.add(new Employee("Tim", 50000));
                emps.add(new Employee("Andy", 60000));
                EmployeeInfo ei = new
      EmployeeInfo(EmployeeInfo.SortMethod.BYNAME);
                ei.sort(emps);
                System.out.println(emps);
                ei = new EmployeeInfo(EmployeeInfo.SortMethod.BYSALARY);
                ei.sort(emps);
                System.out.println(emps);
        }
}
```

Creating a Comparator *Closure*

- A *closure* is a functor embedded inside another class, that is capable of remembering the state of its enclosing object. 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.

  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());
        }
        public static void main(String[] args) {
                List<Employee> emps = new ArrayList<>();
                emps.add(new Employee("Joe", 100000));
                emps.add(new Employee("Tim", 50000));
                emps.add(new Employee("Andy", 60000));
                EmployeeInfo ei = new EmployeeInfo();
                ei.sort(emps, EmployeeInfo.SortMethod.BYNAME);
                System.out.println(emps);
                //same instance
                ei.sort(emps, EmployeeInfo.SortMethod.BYSALARY);
                System.out.println(emps);
        }
}
```

Creating a Comparator using Lambda

Example 1 :

```
public void sort(List<Employee> emps,  SortMethod method) {
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;
}
});
```

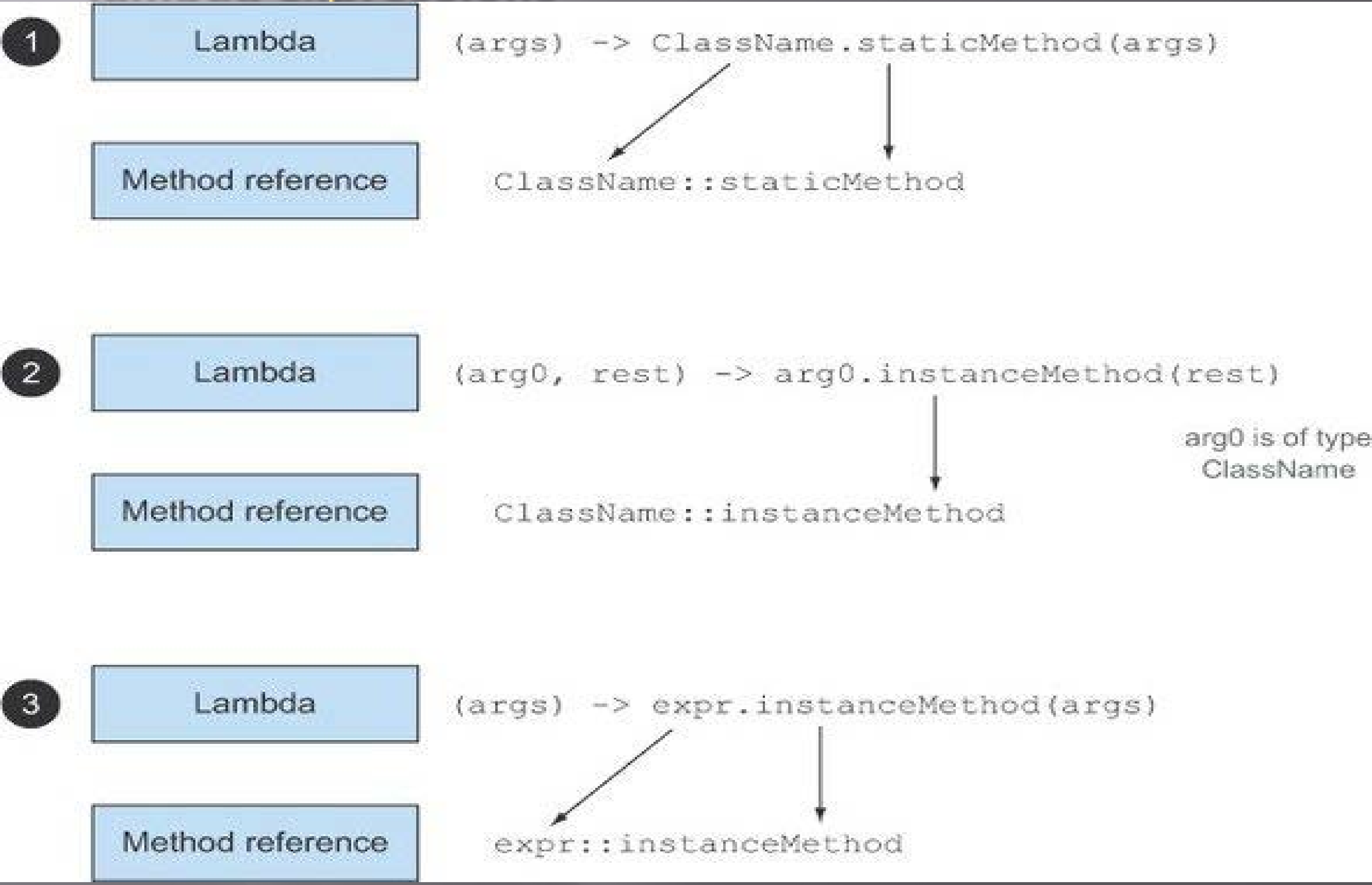
Example 2 :

```
Collections.sort(product, PRICE_COMPARATOR);
// Lambda Implementation
System.out.println("\nSorting objects in decreasing order of price,
using lambdas");
Collections.sort(product, (c1, c2) -> c2.price().compareTo(c1.price()));
System.out.println(product);
```

Method References

- A lambda expression represents an anonymous function that is treated as an instance of a functional interface.
- A method reference is a shorthand to create lambda expression using an existing method.
- Using method references makes your lambda expressions more readable and concise; it also lets you use the existing methods.
- If a lambda expression contains a body that is an expression using a method call, you can use a method reference in place of that lambda expression.

Method references for three different types of lambda expressions


1
Lambda
(args) -> ClassName.staticMethod(args)
Method reference
ClassName::staticMethod
2
Lambda
(arg0, rest) -> arg0.instanceMethod(rest)
arg0 is of type ClassName
Method reference
ClassName::instanceMethod
3
Lambda
(args) -> expr.instanceMethod(args)
Method reference
expr::instanceMethod

Cont…

- There are three main kinds of method references:

1. A method reference to a *static method* (for example, the method parseInt of Integer, written

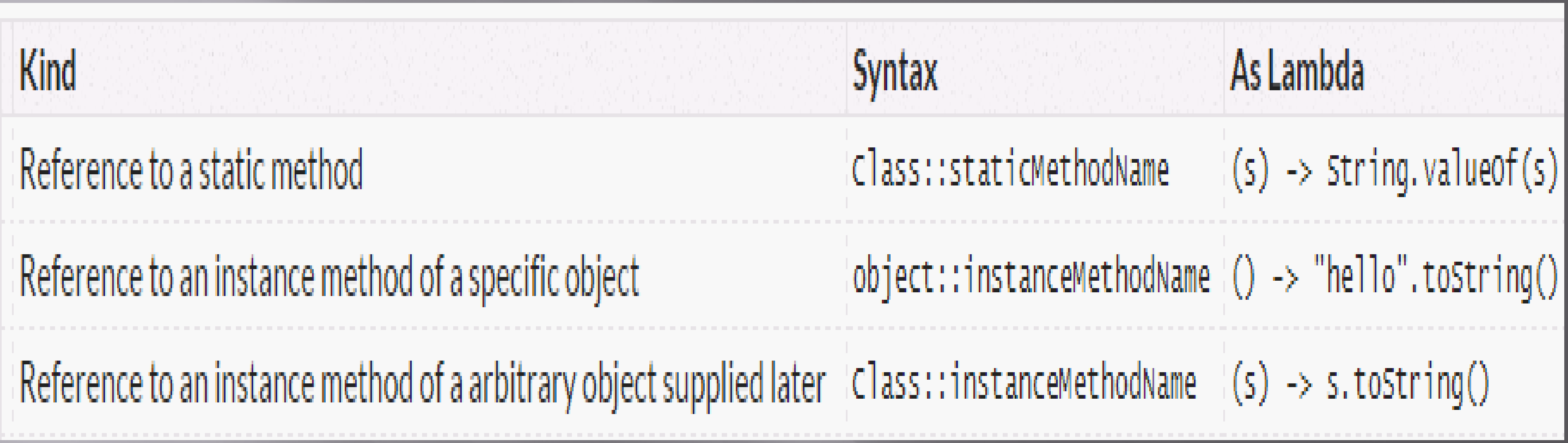
Integer::parseInt)

2. A method reference to an *instance method of an arbitrary type* (for example, the method

length of a String, written **String::length**)

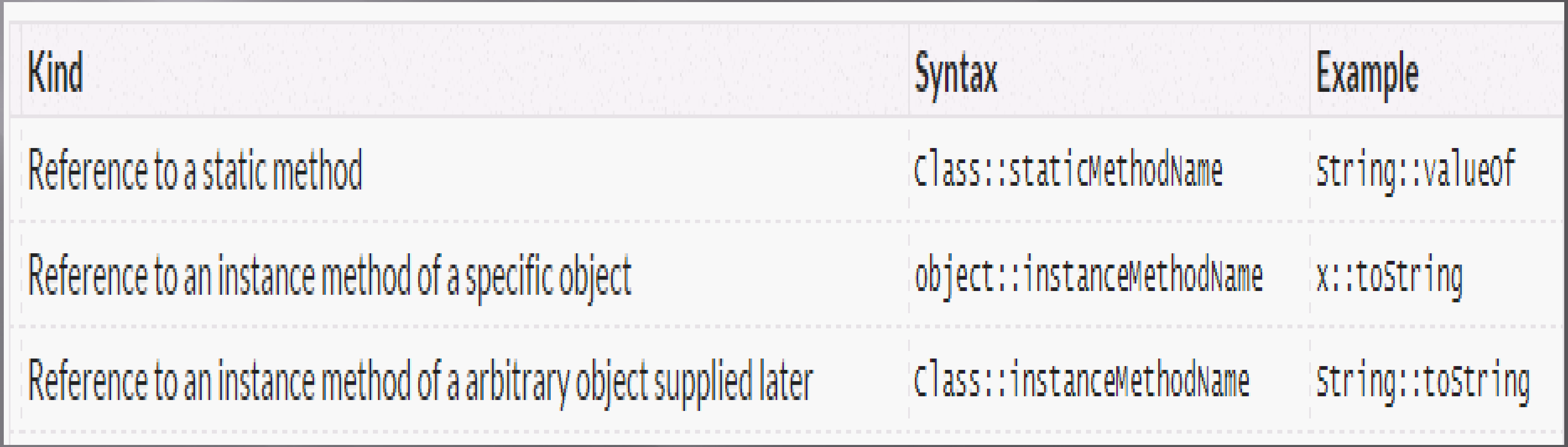
3. A method reference to an *instance method of an existing object* (for example, suppose you have a local variable expensiveTransaction that holds an object of type Transaction, which supports an instance method getValue; you can write **expensiveTransaction::getValue**)

Method calls using Lambdas



Kind	Syntax	As Lambda
Reference to a static method	`Class::staticMethodName`	`(s) -> String.valueOf(s)`
Reference to an instance method of a specific object	`object::instanceMethodName`	`() -> "hello".toString()`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`(s) -> s.toString()`

Method calls using Method References



Kind	Syntax	Example
Reference to a static method	`Class::staticMethodName`	`String::valueOf`
Reference to an instance method of a specific object	`object::instanceMethodName`	`x::toString`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`String::toString`

Example 1 :

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
```

The above code can be written using Method reference as

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort(String::compareToIgnoreCase);
```

Example : 2

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction = str -> str.length();
String name = "Ellen";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ", length = " + len);
```

The above code can be written using Method reference as

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction = String::length;
String name = "Ellen";
int len = lengthFunction.applyAsInt(name);
System.out.println("Name = " + name + ", length = " + len);
```

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 upper case.

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;
}
```

Using Lambdas and Streams (Java 8)

```
 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
          }
```