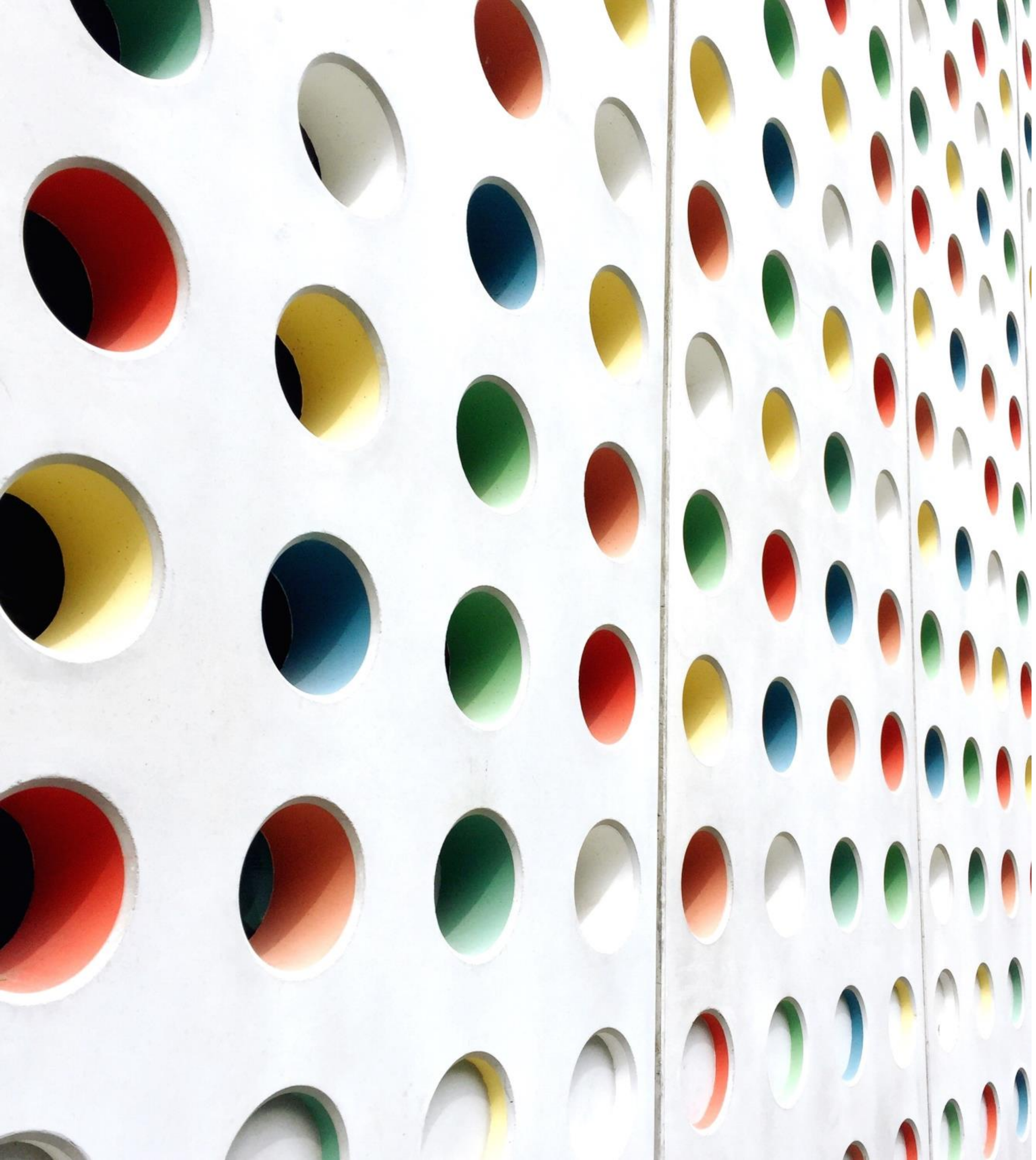
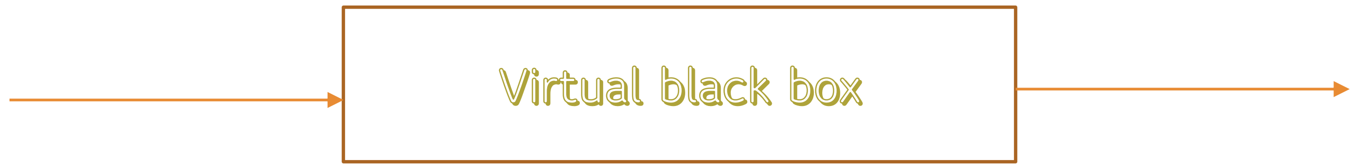


# Functional Programming

CHANDREYEE CHOWDHURY

# Functional Programming

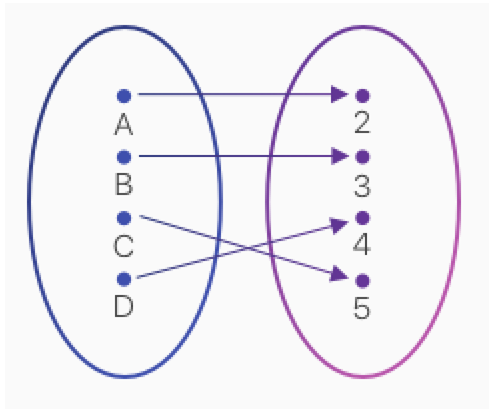
- ❑ A program is a description of a specific computation
- ❑ "WHAT" of the computation + "HOW" of the computation



- A program becomes equivalent to a mathematical function
- Programs, procedures and functions as functions
- It only distinguishes between input and output

# Mathematical function

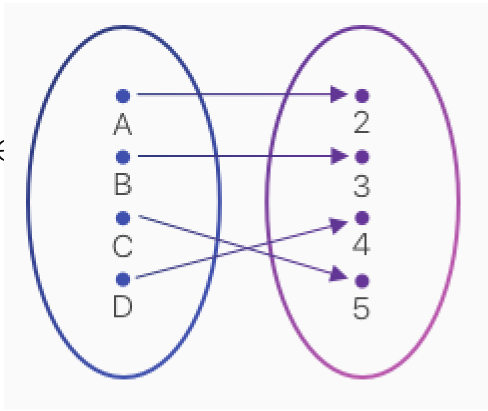
- A function is a rule that associates to each `x` from some set `X` of values a unique `y` from a set `Y` of values
  - `y=f(x)`
  - `f:X→Y`          `g:X→Y`



- Partial vs total participation
- In programming languages
  - Function definition describes how a value is computed using formal parameters
  - Function application is function call using actual parameters
- independent variable `x` in `f` is a parameter w.r.t programming

## Extensional vs Intensional

- Two functions are *extensionally equal if* they have the same input-output behavior
  - Function as sets
- Two functions are *intensionally equal if* they are given by (essentially) the same formula, that is the same definition
  - Function as rules
- To find *f*(*n*), first add 5 to *n*, then multiply by 2. return (n+5)*2
- To find *g*(*n*), first multiply *n* by 2, then add 10. return (n*2)+10
- two functions are Intensionally equivalent nd only if they assign the same values to the same arguments at every world
- highest-mountain-on-earth
- highest-mountain-in-the-Himalayas



## Mathematics vs imperative programming

1. Variables stand for actual values
2. No concept of memory location and assignment

1. Variables refer to memory locations to store values
2. New values can be assigned

- ❑ Pure functional programs adopt a strictly mathematical approach to variables
- ❑ No loops

# Referential transparency

```
void function(int u, int v, int *x)
{

   int y,t,z;

   z=u;y=v;

   while(y!=0) {

       t=y;

       y=z%y;

       z=t;      }

   *x=z;

}
```

```
int function (int u, int v) {

   if(v==0)

     return u;

   else

     return function(v, u%v);
```

$$function(u,v) = \begin{cases} u & \text{if } v = 0 \\ function(v, u\%v) & \text{Otherwise} \end{cases}$$

## Referential transparency

- Output of any function depends only on
  - arguments
  - Non local variables
  - Irrespective of order of evaluation of its arguments
- some function inherently depends on
  - State of the machine
  - Previous call to itself
- A referentially transparent function with no parameters must always return the same value
  - no different than a constant
  - NOT a function in purely functional languages

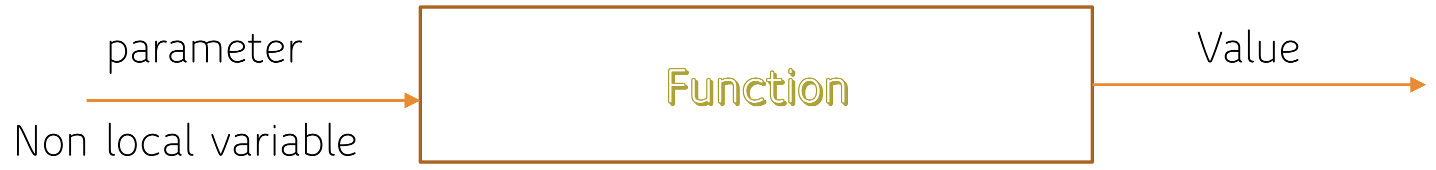
## First class data values

- value semantics
  - Opposite to OOP where computation proceeds by changing the local state of the objects
- functions must be viewed as values themselves
- values can be computed by functions
- can be passed as parameters to other functions
- x=f(g())
- G(x)=f(h(y,z))

# Higher order functions

- composition is itself a function that takes one or more functions as arguments and returns another function

- `if f:X→Y and g:Y→Z then gof: X→Z is given by`
- `(gof)(x)=g(f(x))`



- `F((g(x))`

# Functional Programming in Java

STREAMS AND LAMBDA EXPRESSIONS

## Lambda Expressions

- Way to represent anonymous functions
- behaviour parameterization

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

- The construction is obscure as we want to pass behaviour but we pass objects instead

```
Runnable multiStatement = () -> {
    System.out.print("Hello");
    System.out.println(" World");
};
```

## Lambda Expressions

arguments

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Body of the lambda

- ❑ Instead of passing an object of an interface a function without a name is passed
- ❑ → separates the parameter from the body of the lambda expression
- ❑ javac infers the type of the variable (event)
  - ❑ signature of addActionListener()
- ❑ Using null is another type of type inference

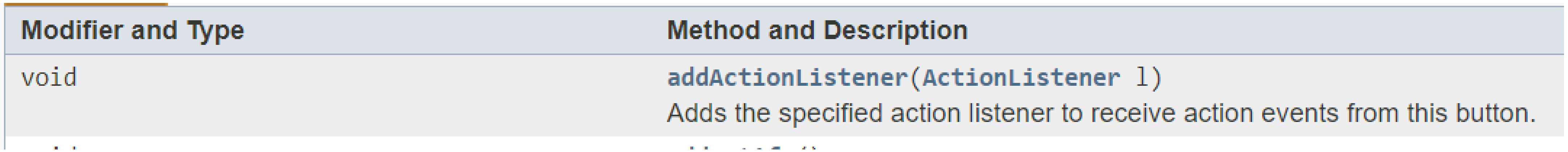
```
(x) →x+1
```

Returns x+1

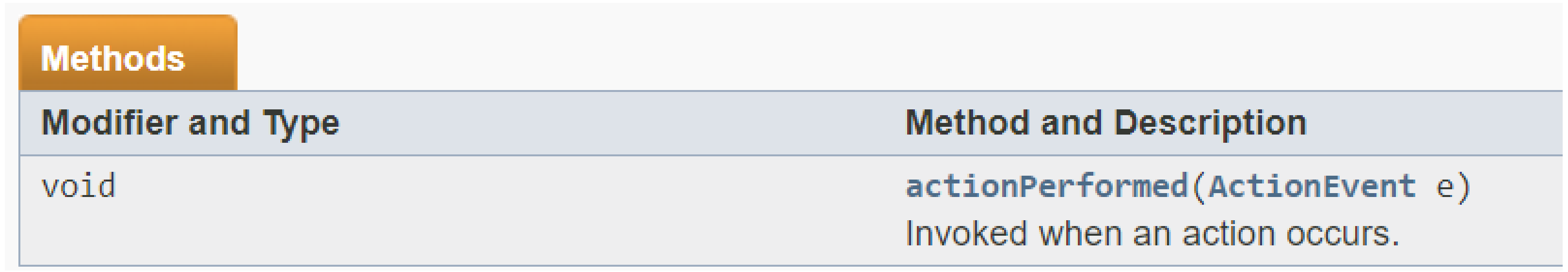
```
Runnable noArguments = () -> System.out.println("Hello World");
```

## Lambda Expressions

```
button.addActionListener(event -> System.out.println("button clicked"));
```



| Modifier and Type | Method and Description |
|---|---|
| void | addActionListener(ActionListener l)<br>Adds the specified action listener to receive action events from this button. |



Methods

| Modifier and Type | Method and Description |
|---|---|
| void | actionPerformed(ActionEvent e)<br>Invoked when an action occurs. |

## Lambda Expressions

- it allows functions to be treated as data values
- Features
  - do not have a specific name
  - not associated with any class unlike a java method
  - can be passed as an argument to a method or stored as a variable (passed around)
  - concise syntax – not verbose like inner classes
- `(parameters)→ expressions;`
- `(parameters)→ {statements;}`

- `()→ {return "CR";}`
- `()→ "CR"`

## Lambda Expressions

```
()→ return "Iron Man"
```

- Lambdas can be used to
  - create objects
  - writing Boolean expressions
  - extracting data from an object
  - combine two values
  - compare two objects

1. ()→ new Mask(10)
2. (String s) → s.length()
3. (List<String> list) → list.isEmpty()
4. (Mask m1, Mask m2)→ m1.getLayers().compareTo(m2.getLayers())

```
final String name = getUserName();
button.addActionListener((event)->System.out.println("hi" +
name));
```

{

## Lambda Expressions

```
BinaryOperator <Long> addExplicit = ( Long x, Long y)→ x+y;
```

- ❑ Immutable values
  - ❑ Anonymous inner classes can only access final (local) variables of their surrounding methods
  - ❑ Free variables captured by lambda should be effectively final
- ❑ This explains closure
  - ❑ Lambdas close over values rather than variables

# *Capturing Lambdas*

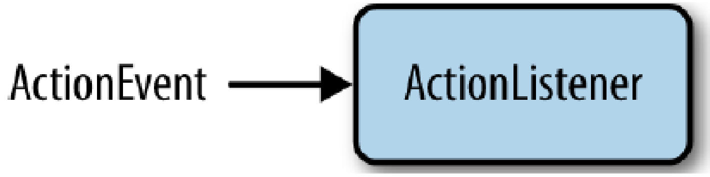
- *Free variables can be captured*
  - `int portNumber = 1337;`
  - `Runnable r = () -> System.out.println(portNumber);`

    **`portNumber=1554;`**
- Lambdas can be passed as argument to methods and can access variables outside their scope
- variables have to be implicitly final
- Allowing capture of mutable local variables opens new thread-unsafe possibilities, which are undesirable
  - close over values rather than variables
- instance variables are fine because they live on the heap, which is shared across threads

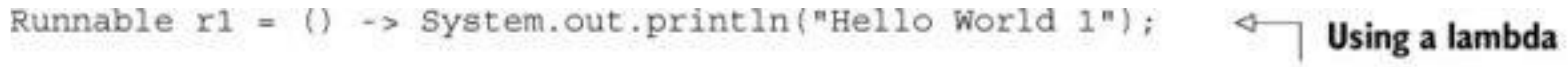
```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

## Functional Interfaces

- An interface with a single abstract method that is used as the type of the lambda expression
- May use more than 1 parameters
- may return a value
- may use generics
- signature of the lambda expression should be same as the method of the functional interface
- the type checking for lambda expressions are performed by the compiler
- More example:- Runnable, Comparator

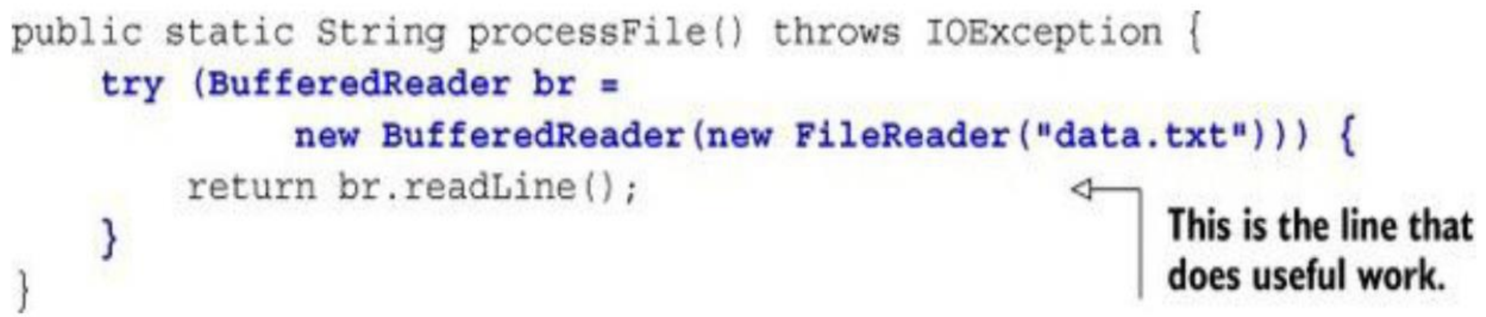


## Functional Interfaces



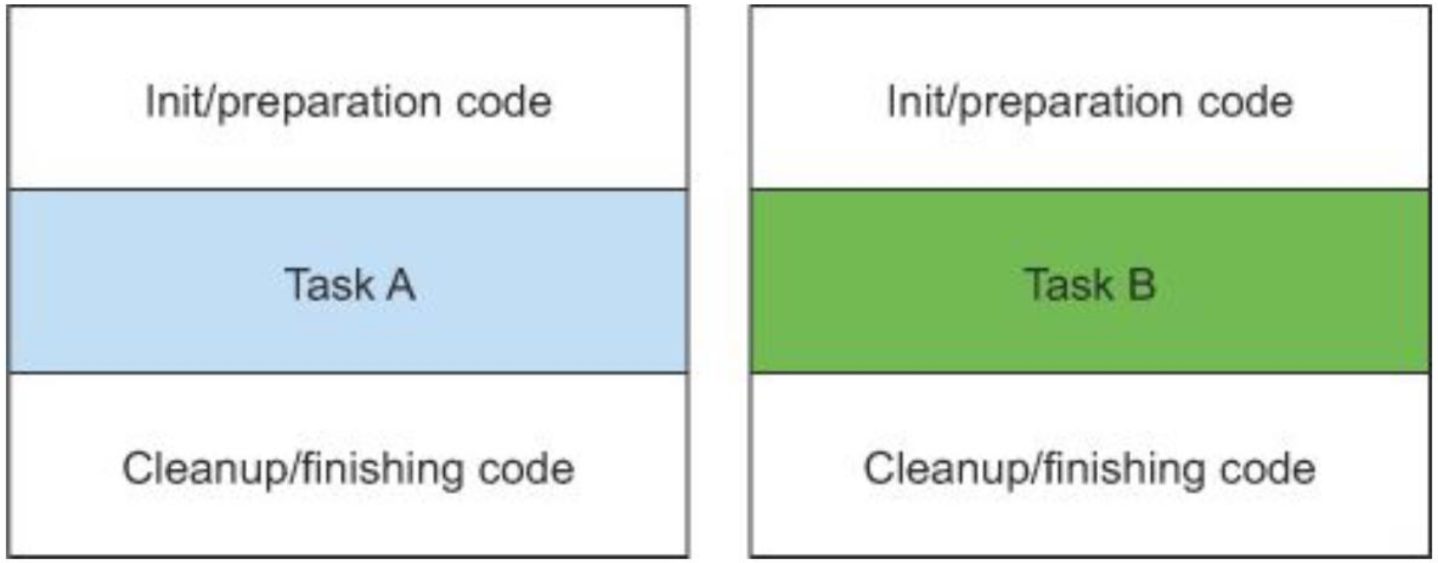
```java
Runnable r1 = () -> System.out.println("Hello World 1");
```

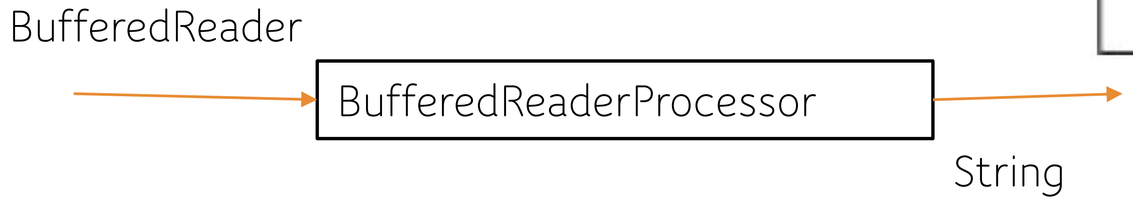
Using a lambda

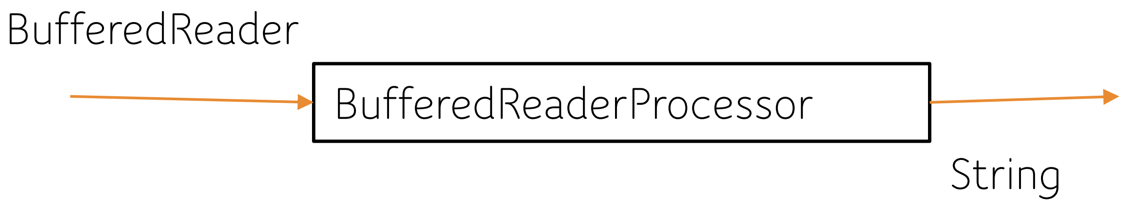


```
public static String processFile() throws IOException {
    try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

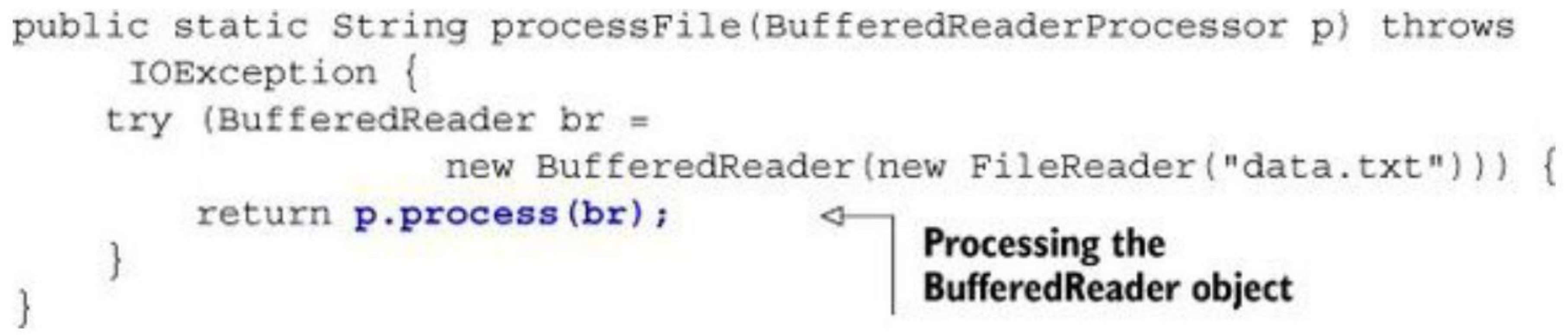
```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```







## Execute Around Pattern



```
public static String processFile(BufferedReaderProcessor p) throws
     IOException {
    try (BufferedReader br =
                     new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);
    }
}
```

**Processing the BufferedReader object**

```
String oneLine =
   processFile((BufferedReader br) -> br.readLine());
```

```
String twoLines =
   processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

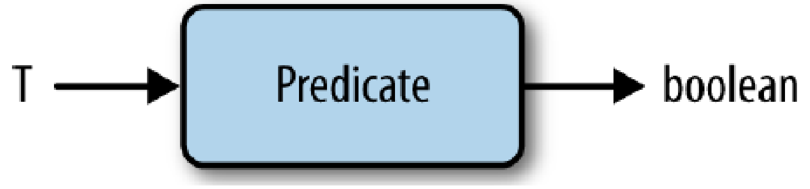
| Interface name | Arguments | Returns |
|---|---|---|
| Predicate<T> | T | boolean |
| Consumer<T> | T | void |
| Function<T,R> | T | R |
| Supplier<T> | None | T |
| UnaryOperator<T> | T | T |
| BinaryOperator<T> | (T, T) | T |

## Functional Interfaces

- New functional interfaces are defined
- Function<T,R> {
-     <R> apply(<T>) ;
- }
- X→X+1;
- X→X==1;
- (X,Y)→X+1;
- (String s)→s.length();

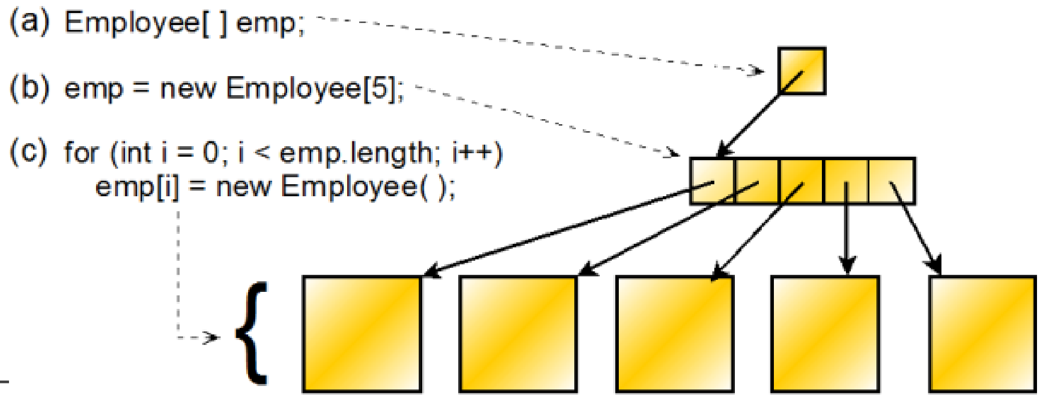
```
Predicate<Integer> atLeast5 = x -> x > 5;

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



***Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();***

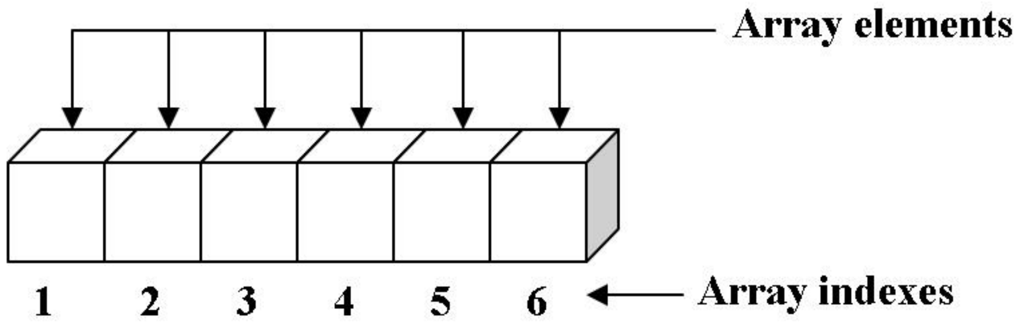
## Boxing and Unboxing


(a) Employee[ ] emp;
(b) emp = new Employee[5];
(c) for (int i = 0; i < emp.length; i++)
emp[i] = new Employee( );

- Boxing converts- mechanism to convert a primitive type into a corresponding reference type
- Unboxing converts
- Autoboxing automatically performs boxing and/or unboxing
- Each element of a primitive array is the size of the primitive
- Boxed values use more memory
- require additional memory lookups to fetch the wrapped primitive value

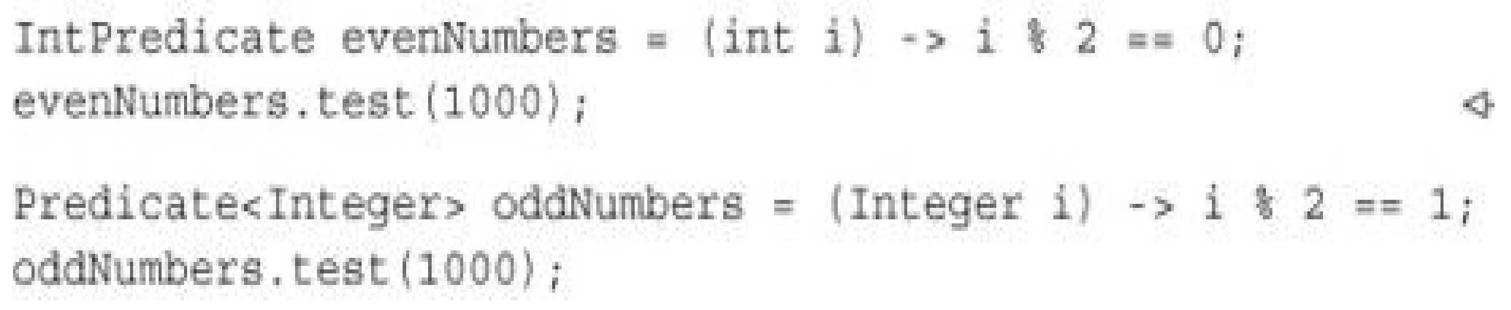
ToIntFunction<T>

int apply(<T>)



One-dimensional array with six elements

## Boxing vs Unboxing



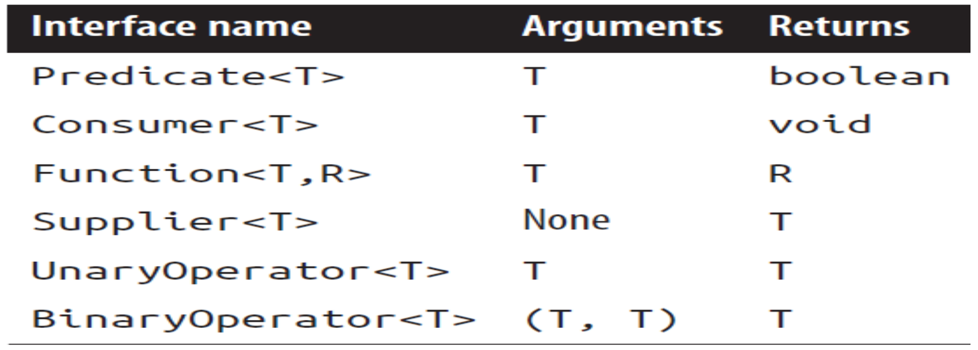
```
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);

Predicate<Integer> oddNumbers = (Integer i) -> i % 2 == 1;
oddNumbers.test(1000);
```

ToIntFunction<T>

IntToDoubleFunction f=a->a+1;

## Target Typing



| Interface name | Arguments | Returns |
|---|---|---|
| Predicate<T> | T | boolean |
| Consumer<T> | T | void |
| Function<T,R> | T | R |
| Supplier<T> | None | T |
| UnaryOperator<T> | T | T |
| BinaryOperator<T> | (T, T) | T |

```
Function<Integer,Boolean> f=a->a==1;

Predicate<Integer> p1=a->a==1;
```

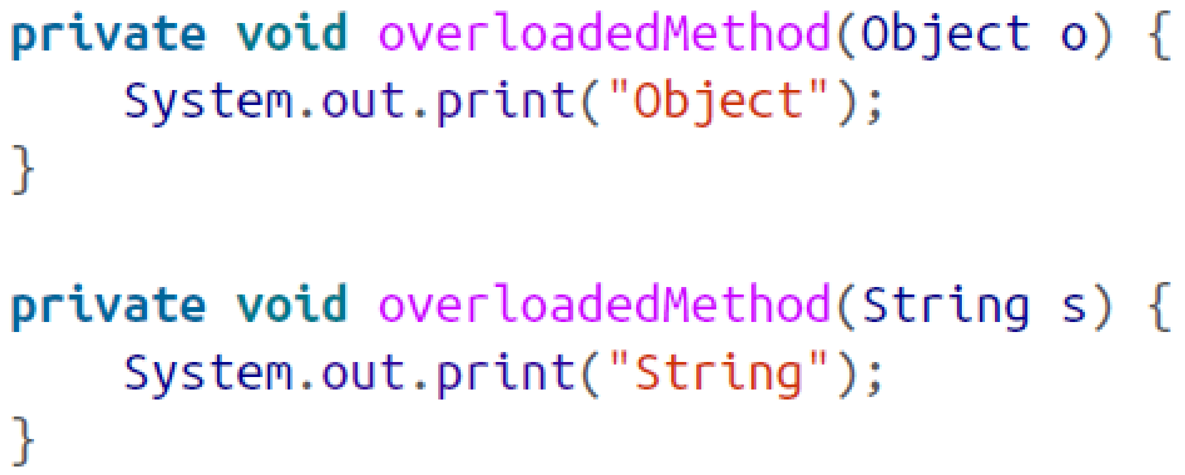
If a lambda has a statement expression as its body, it's compatible with a function descriptor that returns void (provided the parameter list is compatible too).

```
// Predicate has a boolean return
Predicate<String> p = s -> list.add(s);
// Consumer has a void return
Consumer<String> b = s -> list.add(s);
```

1. T -> R
2. (int, int) -> int
3. T -> void
4. () -> T

target type can be decided from an assignment context, method invocation context (parameters and return), and a cast context

## Overloading



```
private void overloadedMethod(Object o) {
    System.out.print("Object");
}

private void overloadedMethod(String s) {
    System.out.print("String");
}
```

- OverloadedMethod(“abc”);
- Javac will refer to the most specific type

## Overloading Resolution

Javac will fail to compile this as there is no such most specific target type

```
overloadedMethod((x) -> true);

private interface IntPredicate  {
    public boolean test(int value);
}

private void overloadedMethod(Predicate<Integer> predicate) {
    System.out.print("Predicate");
}

private void overloadedMethod(IntPredicate predicate) {
    System.out.print("IntPredicate");
}
```

## Overloading Rules

1. If there is a single specific target type javac infers ….
2. If there are several specific target types, ….
3. If there are several specific target types and no most specific type,…

## Identifying a Functional Interface

- java.io.Closeable
  - If an object is closeable, it must hold a file object –a handle that can be closed
  - mutating state

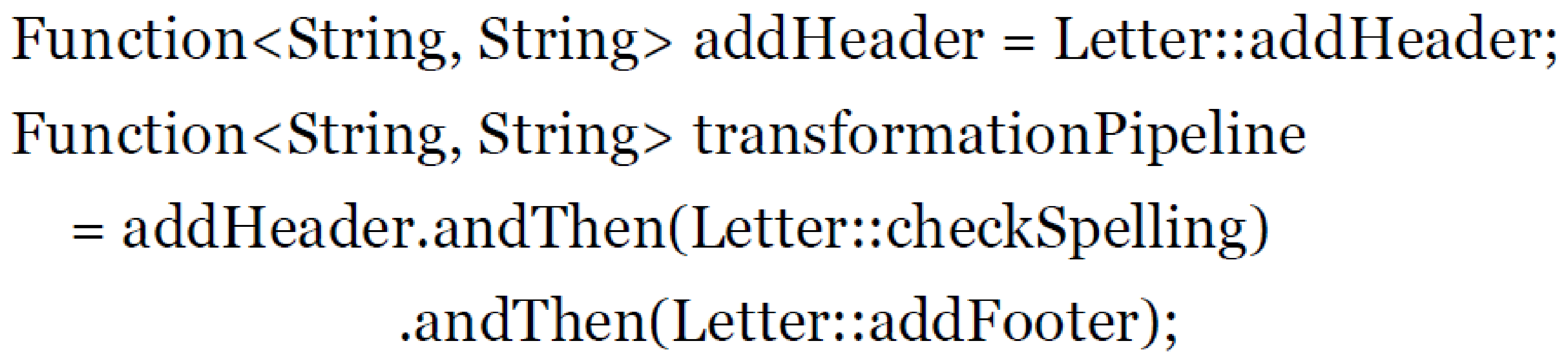
## Composing *Functions*

The Function interface comes with two default methods for this, andThen and compose

Integer::parseInt

D->Integer.parseInt(d)

addHeader=l->l.addHeader()



```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
    = addHeader.andThen(Letter::checkSpelling)
                   .andThen(Letter::addFooter);
```

Course Outcomes

1. Be familiar with functional languages
2. Be exposed to logic programming
3. Be familiar with Lambda Calculus – universal model of computation
4. Being able to map design concepts of Lambda Calculus to modern language features
5. Be familiar with design issues of object oriented languages such as Ruby and Java
6. Apply proper programming paradigm or a mix of it depending on problem description