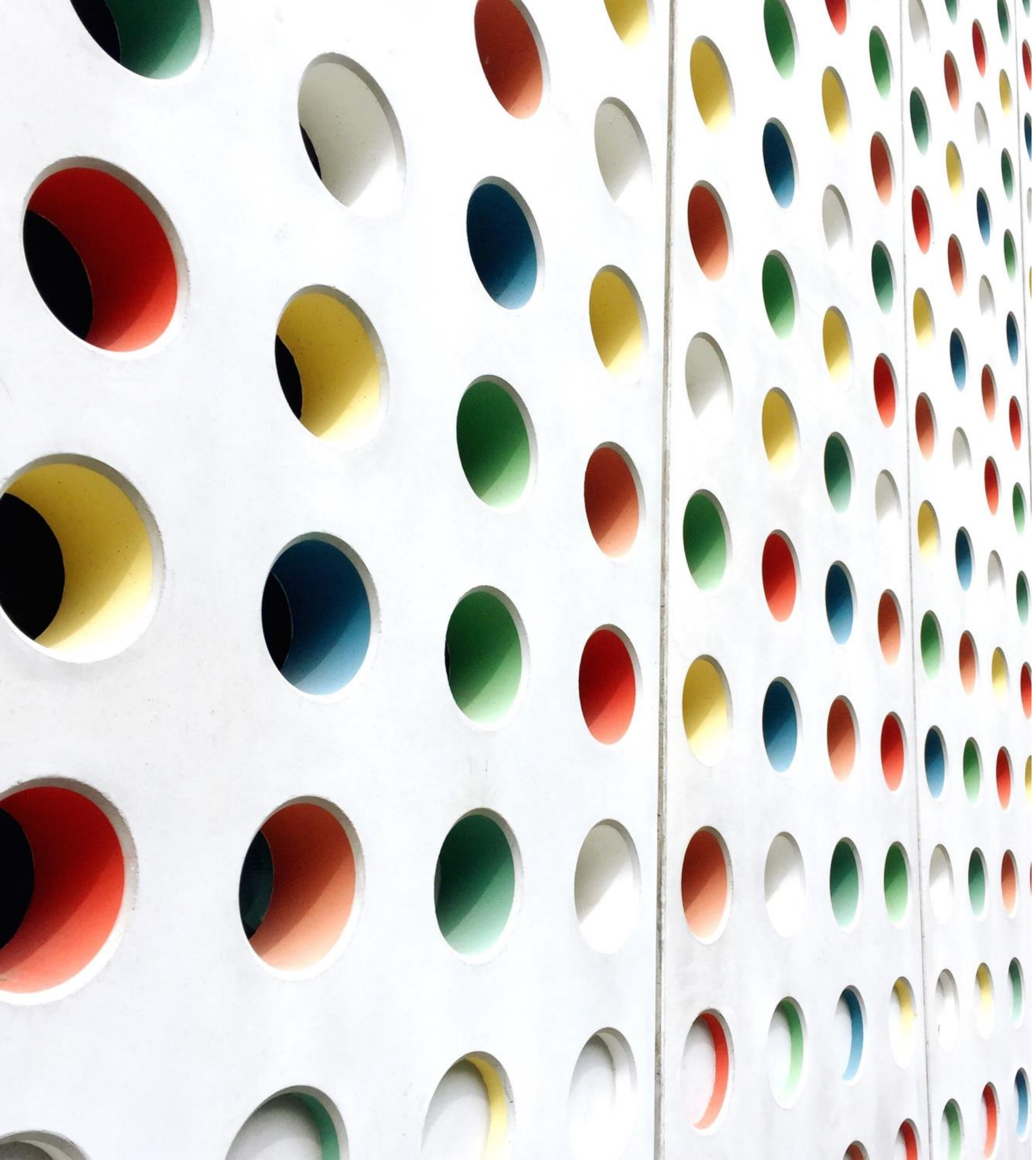


Introduction to Programming Paradigms

CHANDREYEE CHOWDHURY

Programming Languages

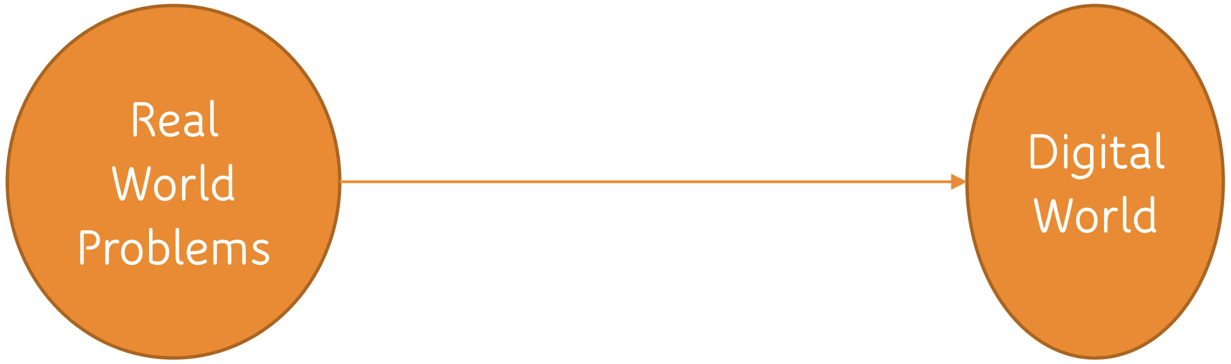
- Programming languages began by imitating and abstracting the operations of a computer
- The kind of computer for which they were written had a significant impact on their design
- A notation for describing computation in machine-readable and human-readable form

Real World Problems
Digital World

Abstraction

An abstraction is a notation or way of expressing ideas that makes them concise, simple, and easy for human mind to grasp

In the case of Assembly language, the programmer must still do the hard work of translating the abstract ideas of a problem domain to the concrete and machine dependent notation of a program



Platform Independence

- Though the programs written in high level languages were independent of the makes and models of the computers, the languages still echoed the underlying architecture of the Von Neumann model of a machine
- There is an area of memory where both programs and data are stored and a separate CPU sequentially executes instructions fetched from memory

Computational Paradigms

- Features of a language based on the Von Neumann model
  - Variable represent memory locations
  - Assignment allows to operate on those memory locations
- An "imperative language"
  - Sequential execution of instructions
  - Variables representing memory locations
  - Assignment to change values of variables
- It is called imperative language because its primary feature is a sequence of statements that represents commands or imperatives

Computational Paradigms

- The requirement that computation can be described as a sequence of instructions, each operating on a single piece of data is referred to as Von Neumann bottleneck.
- Problems
  - Non deterministic computation
  - Parallelism

Computational Paradigms

Object-oriented Programming

1. Based on "object" = memory locations + operations on them.
2. Objects represent reusable codes that mimic the behavior of objects in real world
3. Objects are grouped into classes – having same properties
4. An object is an "instance" of a class.
5. Object oriented paradigm is an extension of imperative paradigm as it primarily relies on the idea of sequential execution with changing set of memory locations
6. Difference-the resulting program consists of a large no of small pieces whose interactions are carefully controlled and easily managed
7. Object interactions could be abstracted as message passing that will map to parallel processors with private memory space

Industrial Age to Information Age

- ❑ Following Moore’s Law, the hardware speeds increased by a factor of 2 every 18 months
- ❑ More abstractions in programming languages could be afforded
- ❑In early 2000, this language abstraction and hardware performance eventually ran into separate roadblocks
- ❑ On the hardware side, the engineers began to reach the limits predicted by Moore’s Law
- ❑ To mitigate this problem, multicore architectures were introduced-CPU now consists of processors with own private memory and a shared memory
- ❑ The roadblock has been broken by the collaboration of the cores to carry out computations in parallel

Breaking the roadblocks

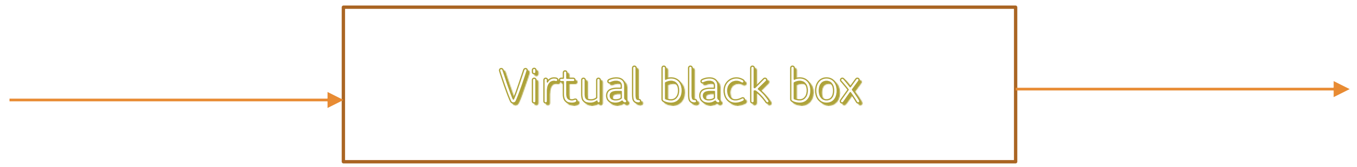
- ❑No higher level abstractions could solve the following problems
  - ❑ the model of computation that relied upon changes to the values of the variables became increasingly difficult to debug
  - ❑ The concept of sequential flow of instructions (with a single processor and memory space) could not be naturally mapped to the multi-core architectures

We need programming language design that is not biased by any particular model of hardware
It should support models of computation suitable for various styles of problem solving

- ❑ Alonzo Church developed such a computational model in the late 1930's

Functional Programming

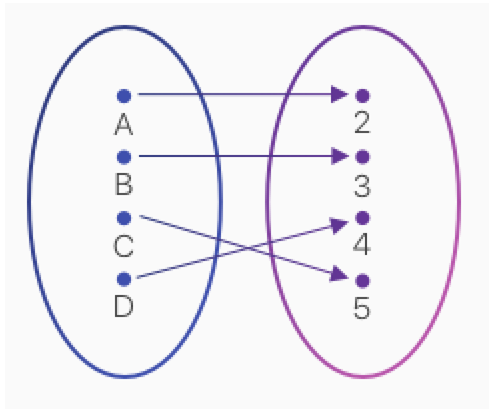
- ❑ A program is a description of a specific computation
- ❑ "WHAT" of the computation + "HOW" of the computation



- 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



Mathematics vs imperative programming

1. Variables stand for actual values
2. No concept of memory location and assignment -values cannot be changed

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

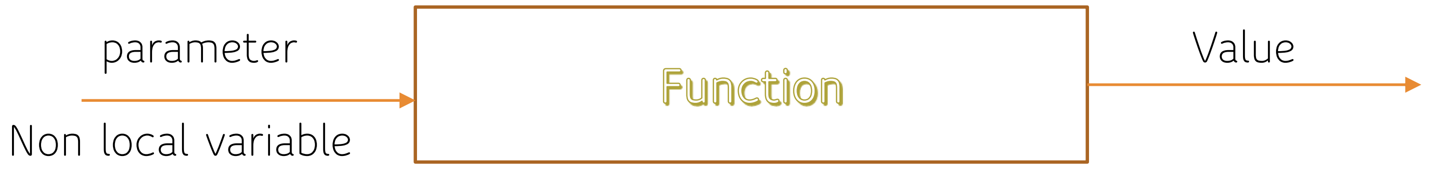
First class data values

- value semantics
  - Opposite to OOP where computation proceeds by changing the local state of the objects- reference semantics
- 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 (an action) 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

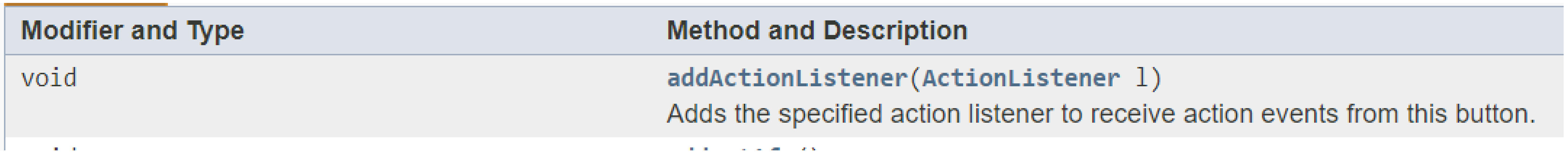
(x) →x+1

Returns x+1

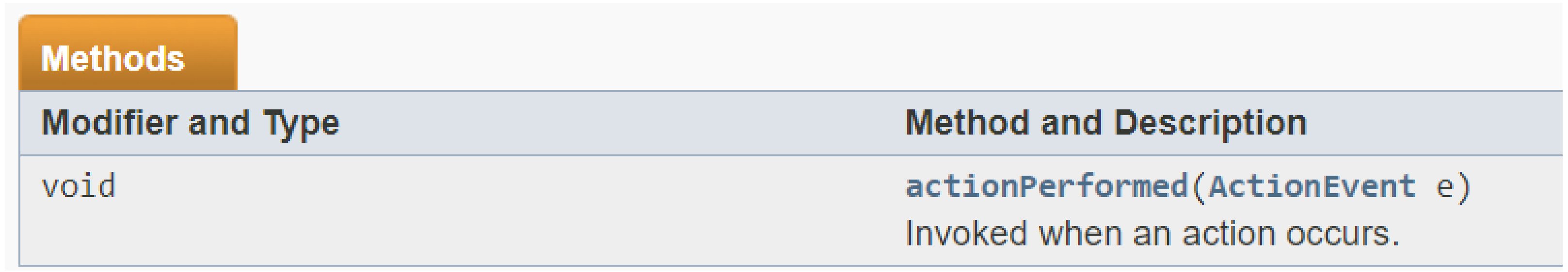
```
Runnable noArguments = () -> System.out.println("Hello World");
```

Lambda Expressions

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



Modifier and Type	Method and Description
void	`addActionListener(ActionListener l)` Adds the specified action listener to receive action events from this button.



Methods

Modifier and Type	Method and Description
void	`actionPerformed(ActionEvent e)` Invoked when an action occurs.

Defining Lambda Expression

A *lambda expression* can be understood as a concise representation of an anonymous function that can be passed around: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown.

- ***Anonymous***— We say *anonymous* because it doesn't have an explicit name like a method would normally have: less to write and think about!
- ***Function***— We say *function* because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- ***Passed around***— A lambda expression can be passed as argument to a method or stored in a variable.
- ***Concise***— You don't need to write a lot of boilerplate like you do for anonymous classes.

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

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())
5. (Integer i) -> return "Alan" + i;

```
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

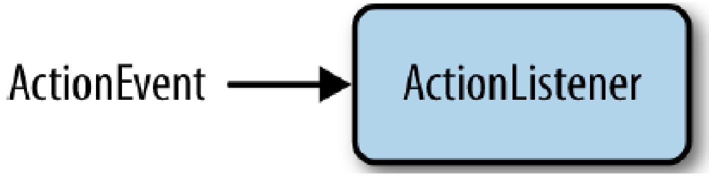
- *Free variables can be captured*
  - `int portNumber = 1337;`
  - `Runnable r = () -> System.out.println(portNumber);`

  `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

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

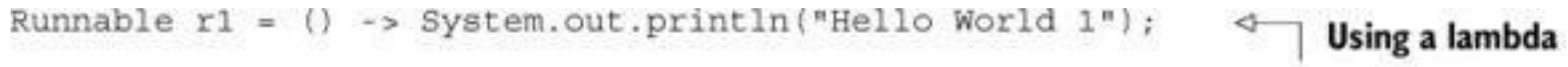
Functional Interfaces

- 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



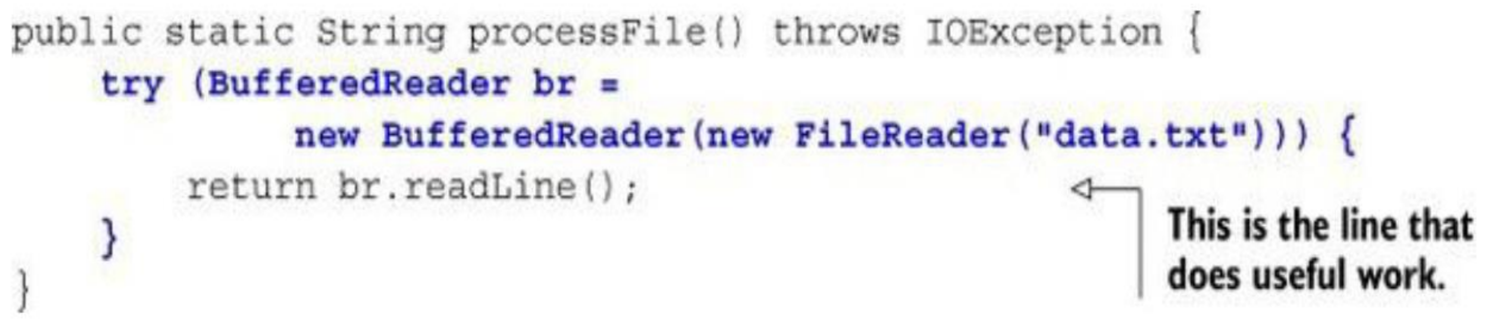
Runnable

Functional Interfaces



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

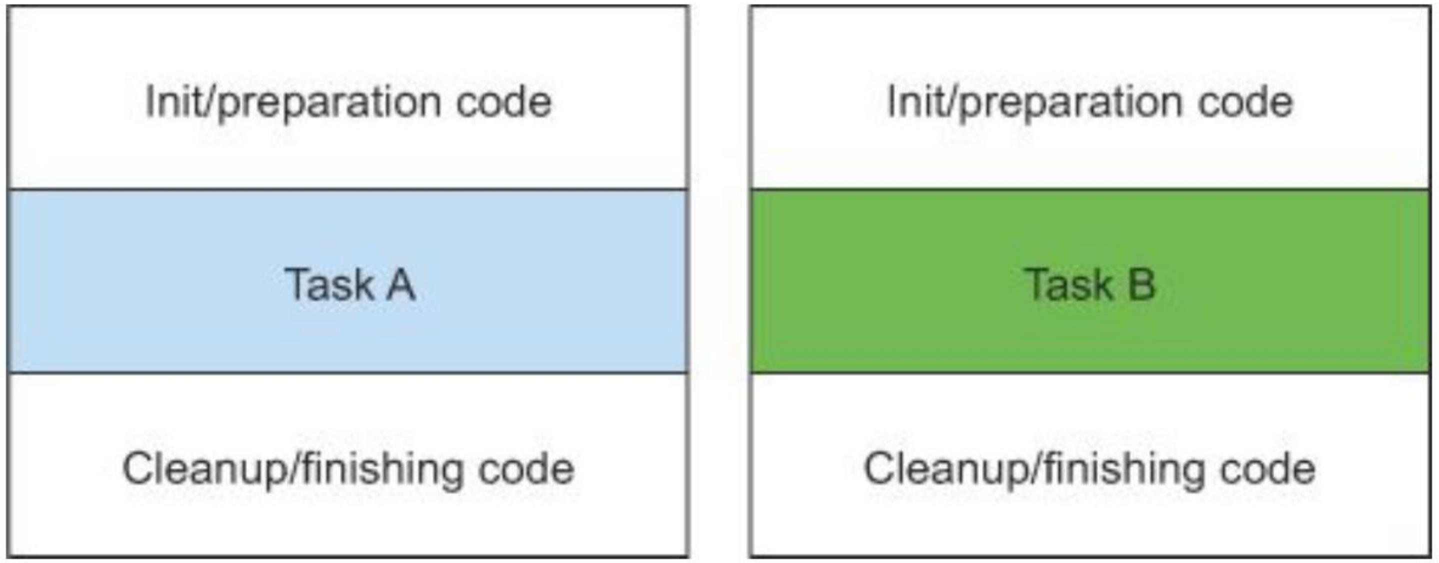
Using a lambda

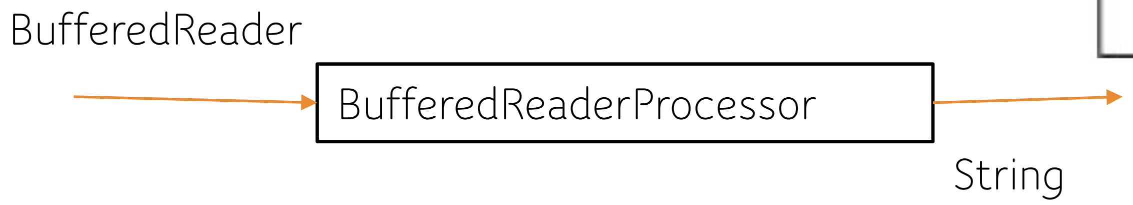


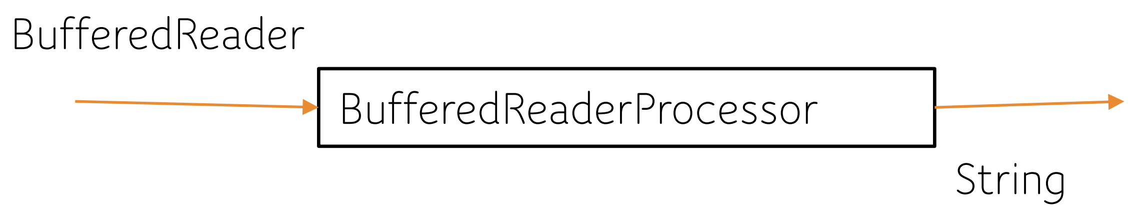
```
public static String processFile() throws IOException {
    try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

This is the line that does useful work.

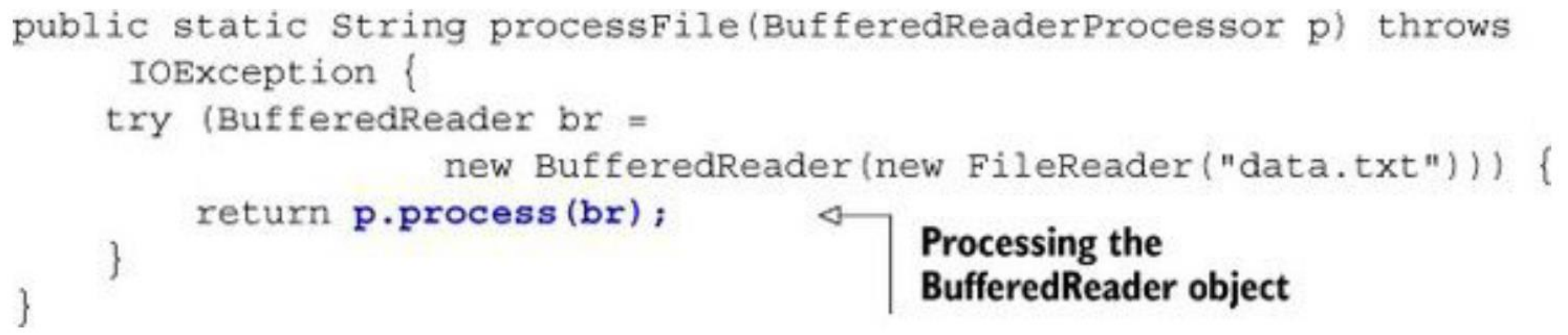
```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```







Execute Around Pattern



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

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

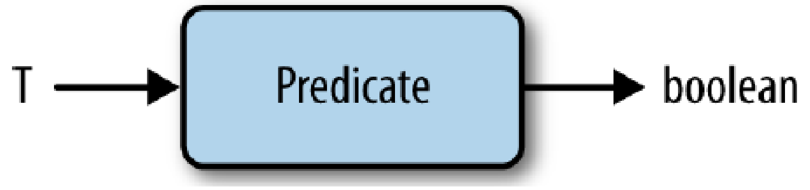
```
String twoLines =
   processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

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

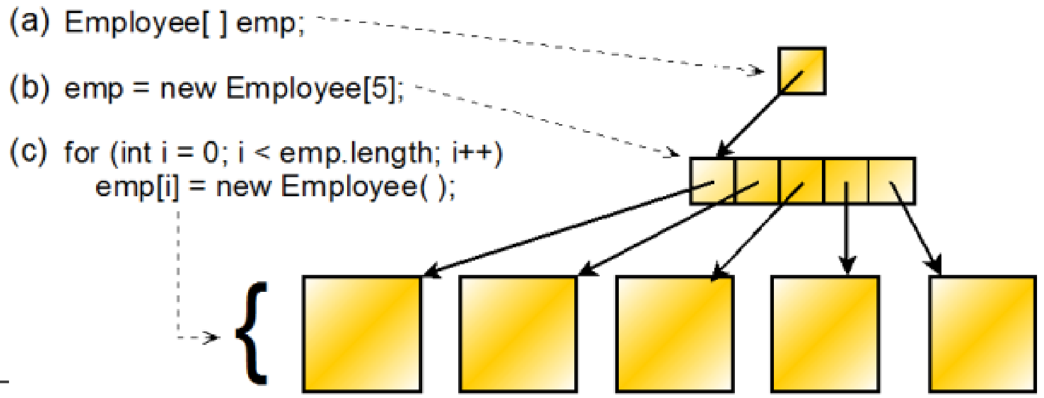
Functional Interfaces

- 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();

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

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



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

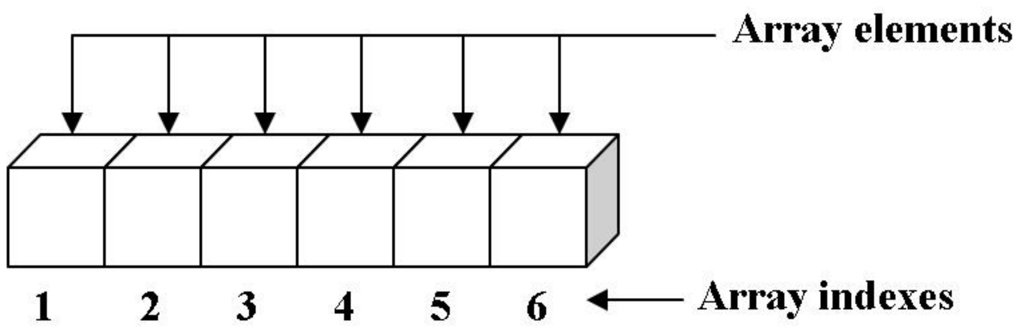

(a) Employee[ ] emp;
(b) emp = new Employee[5];
(c) for (int i = 0; i < emp.length; i++)
emp[i] = new Employee( );

Boxing and Unboxing

- 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

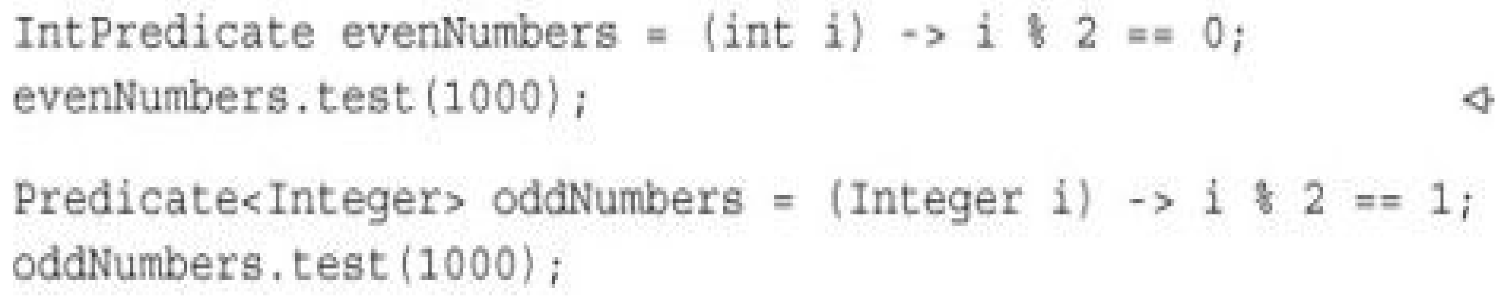
ToIntFunction<T>

int apply(<T>)



One-dimensional array with six elements

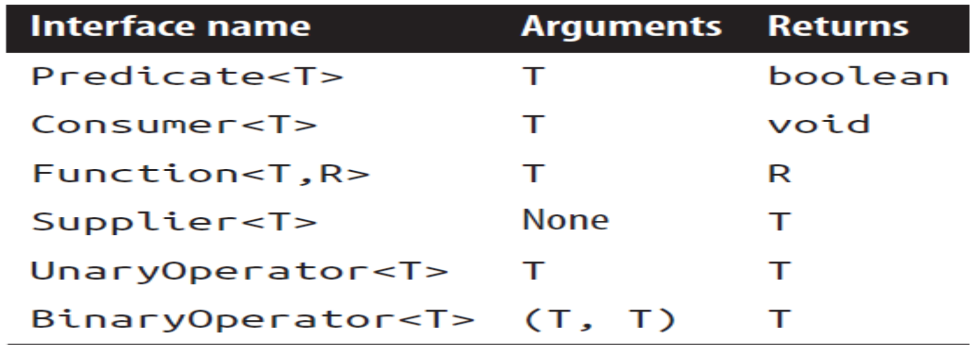
Boxing vs Unboxing



```
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;

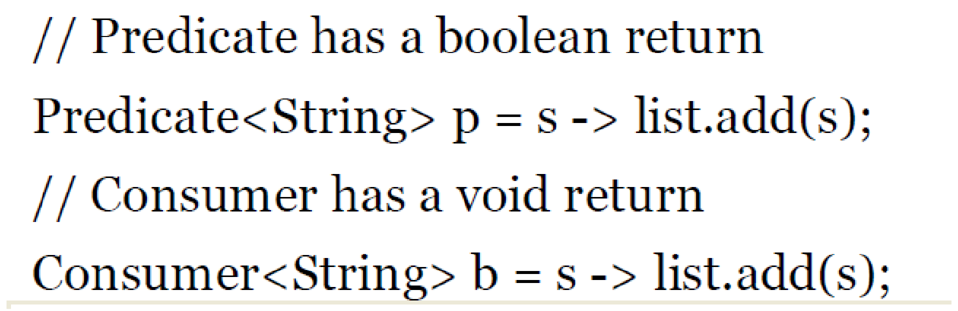


Interface name	Arguments	Returns
Predicate<T>	T	boolean
Consumer<T>	T	void
Function<T,R>	T	R
Supplier<T>	None	T
UnaryOperator<T>	T	T
BinaryOperator<T>	(T, T)	T

Target Typing

```
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).



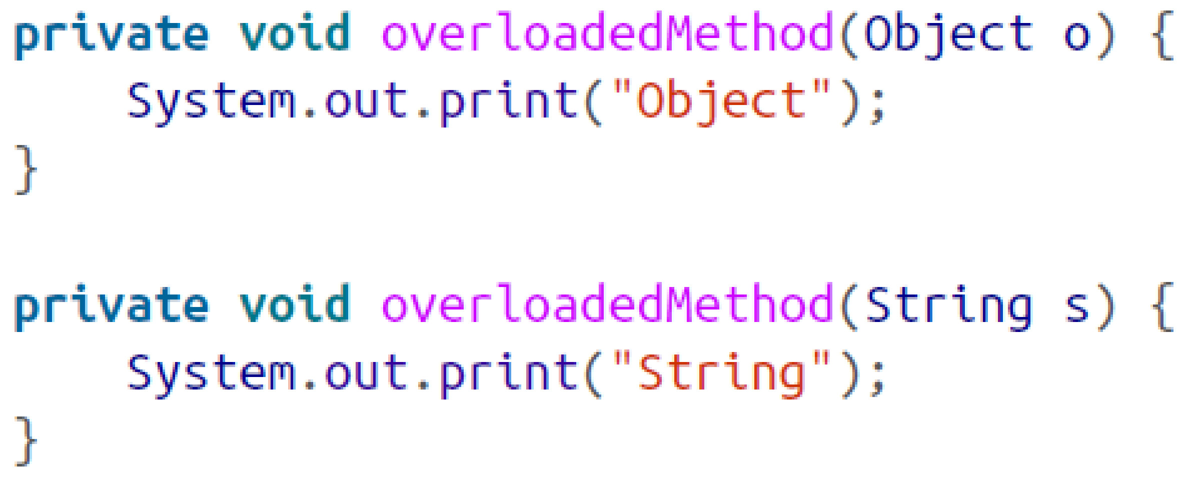
```
// Predicate has a boolean return
Predicate<String> p = s -> list.add(s);
// Consumer has a void return
Consumer<String> b = s -> list.add(s);
```

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

```
boolean test(String s) {

     return list.add(s);

}

void method1(String s) {

    list.add(s);

    return;

}
```

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