Computer System Design & Application
计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn
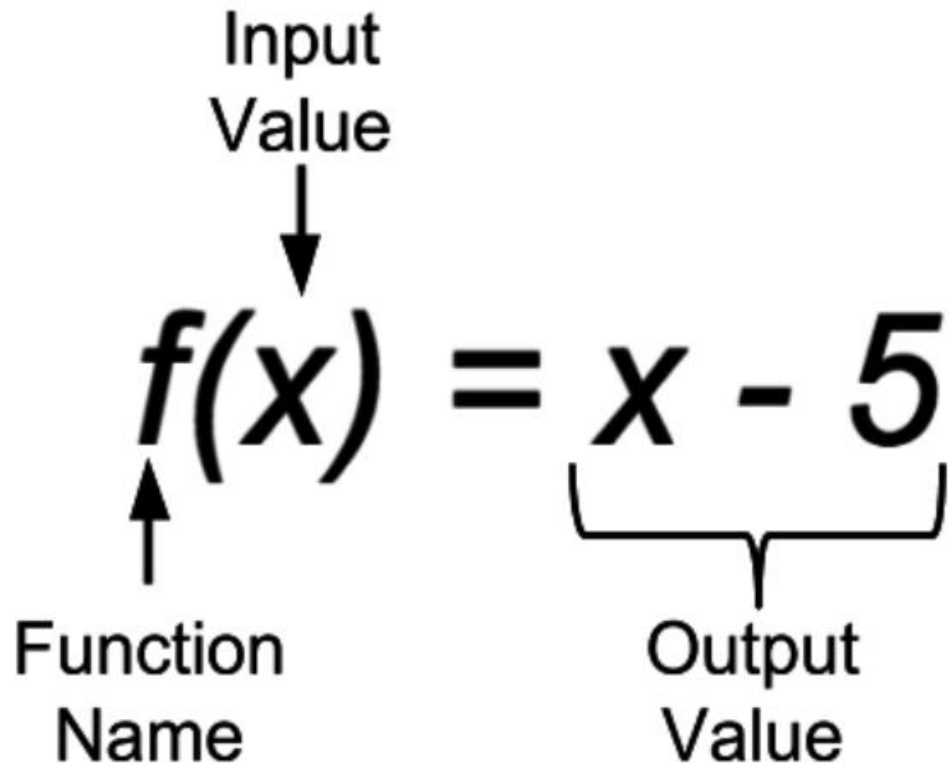
Lecture 3

- Functional Programming
- Lambda Expressions

What is a function?

- [Mathematics] A function from a set X to a set Y assigns to each element of X exactly one element of Y
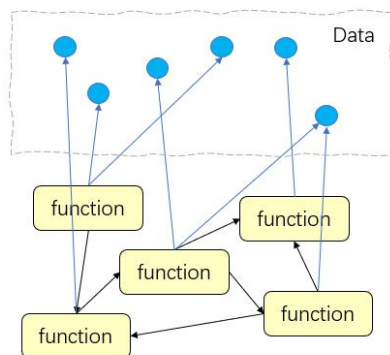- Maps input to output

Input Value

$$f(x) = x - 5$$

Function Name

Output Value

x	f(x)
1	–4
2	–3
3	–2
4	–1

What is Functional Programming?

What types of programming paradigms have we seen so far?

Procedural Design

High coupling. Reduced information hiding.
Hard to make changes and to scale.

Object-oriented Design

Traffic Control System

Data
function1
function2
......

Data
function1
function2
......

Data
function1
function2
......

High cohesion. Good information hiding.
Easier to maintain and extend.

TAO Yida@SUSTECH   49

- Functional programming is a **programming paradigm** (编程范式)
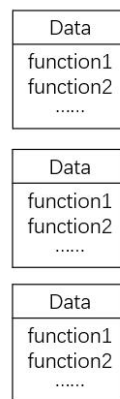- A programming paradigm refers to the way of thinking about things and the way of solving problems

编程范式好比武功门派，博大精深且自成体系。
--摘自《冒号课堂：编程范式与OOP思想》

Programming Paradigms

```
                    ┌─────────────────────┐
                    │    Programming      │
                    │    Paradigms        │
                    └─────────────────────┘
                              │
        ┌─────────────────────┼─────────────────────┐
        │                     │                     │
┌───────────────┐   ┌───────────────┐   ┌───────────────┐
│ Object Oriented│   │  Procedural   │   │  Functional   │
│  Programming   │   │  Programming  │   │  Programming  │
└───────────────┘   └───────────────┘   └───────────────┘
```

Organize data and logics in **objects** (fields and methods)

Organize code as sequential **procedures** (may change same data)

Organize code as **functions** (behaviors are separated from the data, which won't be changed)
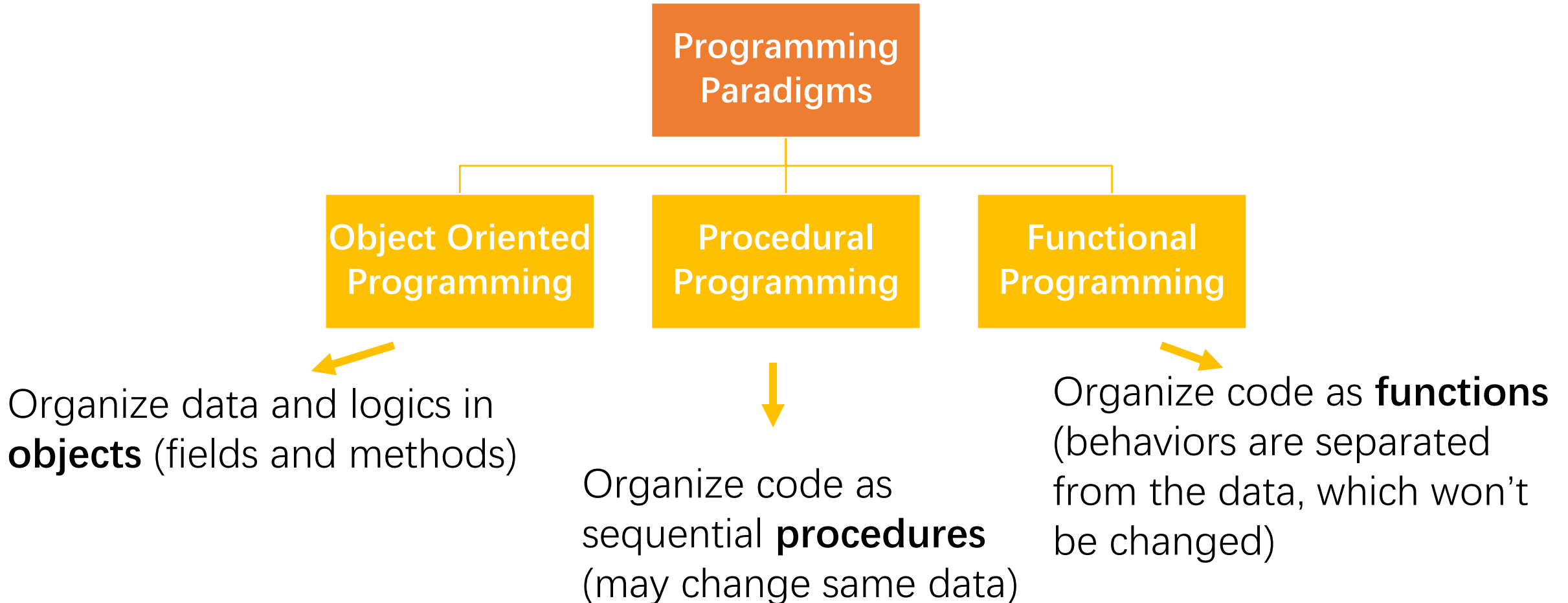
Functional Programming
(函数式编程)

- Basic unit of computation is function
- Primary characteristics
  - Functions are treated as first-class citizens
  - No side effects
  - Immutability
  - Recursion

First-class functions

Functions are treated like any other variables

Functions can be passed as arguments

Functions can be assigned to a variable

Functions can be returned

Image source : https://www.webtips.dev/webtips/javascript-interview/first-class-functions

Higher Order Functions

- A higher order function
  - Takes another function as argument
  - Returns another function as result

- A common usage scenario: data processing
  - `map()`, `filtering()`, `reduce()`, etc.
  - More on these later

No Side Effects

- Side effects: Events that are caused by a system with a **limited scope**, whose effects are felt **outside of that scope**
- Pure functions have no side effects (cannot change external states)

Impure function	Side Effects
writeFile(filename)	External files are changed
updateDatabase(table)	External database table is changed
sendAjaxRequest(request)	External server state is changed

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Impure function	Input	Possible Output
writeFile(filename)	filename	Success or failures given file state
queryDatabase(table)	table	Different results given table state
sendAjaxRequest(request)	request	Different responses given server state

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Is this a pure function?

```
global_list = []
def append_to_list(x):
    global_list.append(x)
    print(global_list)
```

Side effects

- `global_list` is implicitly changed even though it is not declared as the input to the function

- The output of the function changed even though the input remains the same

Immutability

- Variables, once defined, never change their values (eliminate side effects)
- Pure functional programs do not have assignment statements

```
x = x + 10
```

Non-functional style

```
int plusTen(int x)
{
        return x+10;
}
```

Functional style

What about loops?

- No "while" or "for" loop in functional programming
- How do we perform iterations though?

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

```
def factorial(n):
    fact = 1
    while n >= 1:
        fact = fact * n
        n = n - 1
    return fact
```

Using loops

```
def factorial(n):
    if n <= 0: return 1
    return n * factorial(n-1)
```

Using recursive functions (which invoke themselves)
Or higher order functions (e.g., map)

Advantages of Functional Programming

- Easy to debug, test, and parallelize
  - Same input => same output (deterministic)
  - No side effects
  - Data are immutable
- Complexity is dramatically reduced (architectural simplicity)
  - The only interaction with the external system is via the argument and return value (API)

OO style: Object methods interact with the object states

Procedural style: external state is often manipulated from within the function

Functional Programming Languages

- Lisp, Erlang, Haskell, Clojure, Scala, F#, Python, Javascript, Kotlin, Rust, Swift, etc.

- FP is used in big companies
  - **Whatsapp** needs only 50 engineers for its 900M users, using Erlang
  - **Huawei** adopts Rust to engineer trustworthy software systems
  - **NVIDIA** uses Haskell for the backend development of its GPUs
  - **Facebook** uses Haskell to fight spams
  - ......

Functional Programming in Java

- Different programming paradigms are not necessarily mutually exclusive
  - Python also supports OOP
  - Java also supports the functional styles of programming

- Java 8 introduces functional programming abilities
  - Lambda expressions
  - Streams API

Lecture 3

- Functional Programming

- Lambda Expressions
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - Java Functional Interfaces

Java Lambda Expressions

- Introduced in Java 8
  - Java's first step into functional programming

- A Java lambda expression
  - is an anonymous function with no name/identifier
  - can be created without belonging to any class
  - can be passed as a parameter to another function
  - are callable anywhere in the program

Lambda Syntax

Arrow token

`(param1, param2) -> {expressions or statements}`

Left part – function parameters

- No function name
- Parentheses could be omitted for a single parameter
- Multiple parameters are separated by comma (,)
- `()` is used if no parameter is needed

Right part – function body

- Curly braces could be omitted for a single expression
- Multiple statements are separated by a semi-colon (;)
- Can have a `return` statement
- Local assignments and control structures (`if`, `for`) are allowed (but probably less common).

Lambda Parameters

- Zero Parameter

```
() -> {for (int i = 100; i >= 0; i--) System.out.println(i);}
```

- One Parameter

```
(param) -> System.out.println("1 parameter:"+ param)
param -> System.out.println("1 parameter:"+ param)
```

- Multiple Parameters

```
(first, second) -> first.length() - second.length()
```

Lambda Function Body

- One statement

```
(param) -> System.out.println("1 parameter:"+ param)
```

- Multiple statements (with curly braces)

```
(first, second) ->
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

- Return

```
(param1, param2) -> {return param1 > param2;}
(param1, param2) -> param1 > param2;
```

Lambda Usage

- Lambda: a shortcut to define an implementation of a functional interface

- Functional interface is an interface with a single abstract method (e.g., `Comparator<T>` interface only has one abstract method `int compare(T o1, T o2)`)

- We can supply a lambda expression whenever an object of a function interface is expected

Example: sorting a string list by element's length

```java
public class StringComparator implements Comparator<String>{
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
}
Collections.sort(strList, new StringComparator());
```

Classic way
- Explicitly creating a class that implements the Comparator interface
- Verbose

```java
Collections.sort(strList, new Comparator<String>() {
        public int compare(String s1, String s2) {
            return Integer.compare(s1.length(), s2.length());
        }
});
```

Using the anonymous class
- Don't have to declare a name for it
- Declare and instantiate the class at the same time
- Anonymous class can be used only once
- Shorter code, but still verbose

Using lambda in Java 8
```java
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Matching Lambdas to Functional Interfaces

```
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

```
Collections.sort(List<T> list,        Comparator<? super T> c  )
```
1. Lambda is matched to the Comparator interface

```
int compare(T o1, T o2)
```

2. Lambda is matched to `Comparator.compare(T o1, T o2)` method, which is the only abstract(unimplemented) method in the `Comparator` interface

3. The parameter and return type are deduced by compiler using type inference

Matching Lambdas to Functional Interfaces

1. Matching lambda to interface
   - By comparing caller method's parameter type

2. Matching lambda to interface's abstract method
   - The interface must have only one abstract (unimplemented) method

3. Matching method parameters and return type
   - Parameter types and return type must match

All of these must be satisfied to replace a functional interface with lambda successfully!

Matching lambda to interface's abstract method

- From Java 8, an interface could have both default methods and static methods (method with implementations/concrete methods)
- A functional interface must have only one abstract (unimplemented) method, but it can have multiple default and static methods

```java
public interface MyInterface {

    void printIt(String text);

    default public void printUtf8To(String text, OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException("Error writing String as UTF-8 to OutputStream", e);
        }
    }

    static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

Example
- **This interface can be implemented by lambda**
- **Although it has 3 methods, only 1 is unimplemented**

Source: http://tutorials.jenkov.com/java/lambda-expressions.html

Type Inference

- Compiler obtains most of the type information from *generics*
- Compiler won't be able to infer types if raw type (e.g., `List`) is used instead of the parameterized type (e.g., `List<String>`)

> ⚠ List is a raw type. References to generic type List<E> should be parameterized

```
List strList = new ArrayList();
strList.add("abc");
strList.add("bcd");
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

> ⓧ The method length() is undefined for the type Object

Use `List<String> strList = new ArrayList<String>()` to remove all the warnings and errors!

Variable Scope for Lambda

```
String s1 = "";
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

⊗ Lambda expression's parameter s1 cannot redeclare another local variable defined in an enclosing scope.

```
String str = "";
Comparator<String> comp = (s1, s2) -> {   str = str + " test";
                                          return s1.length() - s2.length();};
```

⊗ Local variable str defined in an enclosing scope must be final or effectively final

```
String str = "";
Comparator<String> comp = (s1, s2) -> { System.out.println(str);
                                        return s1.length() - s2.length();};
```

√ The local variable could be accessed without changing its value

More Use Cases

- Use Case I: create & "instantiate" a functional interface

```java
public interface MyInterface{

    // abstract method
    double getPiValue();

}
```

- The lambda expression implements the abstract method; therefore, we could "instantiate" the interface

- Strictly, we are instantiating an anonymous class that implements MyInterface

```java
MyInterface ref = () -> 3.1415;
System.out.println("Pi = " + ref.getPiValue());}
```

More Use Cases

- Use Case II: executing the same operation when iterating elements

```java
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```

```java
default void forEach(Consumer<? super T> action)
```

Only 1 abstract method

```java
@FunctionalInterface
public interface Consumer<T>

void accept(T t)
```

Performs this operation on the given argument.

Lecture 3

- Functional Programming

- **Lambda Expressions**
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - Java Functional Interfaces

Method Reference

- Sometimes, a lambda expression does nothing but call a <u>one existing</u> method

- Method reference allows us to refer to this one method <u>by name</u>, which is often (but not always) easier to read

```java
public interface MyInterface{
        public void print(String s);
}
```

```java
// Using lambda expression
MyInterface ref = s -> System.out.println(s);

// Using method reference
MyInterface ref = System.out::println;
```

Class that owns the method (receiver)

The double colon indicates that this is a method reference

The method name

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Static Method
- Methods declared with the `static` keyword
- Can be called without creating an object of a class (no need to `new` an instance)
- E.g., `Integer.parseInt`, `Math.min`

Instance Method
- Methods declared not with the static keyword
- Requiring an object of its class to be created before it can be called

Method Reference

- **Static method**
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Syntax
```
ClassName::staticMethod
```

Example
```
Integer::parseInt
```

Lambda Equivalent
```
str -> Integer.parseInt(str)
```

Method Reference

- Static method

- **Instance method (Bound)**

- Instance method (Unbound)
- Constructor

```
System.out::println
```

executes `println` on a sepcific instance of
`PrintStream`, which is the `System.out` instance

Syntax
```
InstanceName::instanceMethod
```

- Here the **InstanceName (object reference)** represents any object/instance.
- We call the **InstanceName** bounded receiver since the receiver is bounded to the instance.

Method Reference

- Static method
- **Instance method (Bound)**
- Instance method (Unbound)
- Constructor

```
public interface Deserializer {
    public int deserialize(String v1);
}
```

```
public class StringConverter {
    public int convertToInt(String v1){
        return Integer.valueOf(v1);
    }
}
```

```
StringConverter stringConverter = new StringConverter();

Deserializer des = stringConverter::convertToInt;
```

Example from http://tutorials.jenkov.com/java/lambda-expressions.html

- **convertToInt** and **deserialize** has the same signature
- The lambda uses a **StringConverter** instance **stringConverter** and refer to its **convertToInt** method

Method Reference

- Static method
- Instance method (Bound)

- **Instance method (Unbound)**
- Constructor

Syntax
```
ClassName::instanceMethod
```

- **ClassName** is the name of the class, such as **String**, **Integer**.
- We call **ClassName** unbounded receiver since the receiver instance is bounded later.
- Unbound receivers allow us to use instance methods <u>as if they were static methods</u>; However, the creation of instances are still required, but are deferred (decided later)

Method Reference

- Static method
- Instance method (Bound)
- **Instance method (Unbound)**
- Constructor

```java
interface Transformer{
    String transform(String s);
}
```

```java
Transformer transformer = (s) -> s.toLowerCase();
```

⬇

```java
Transformer transformer = String::toLowerCase;

String res = transformer.transform("Hello Java");
```

Method Reference

- Static method
- Instance method (Bound)

Instance method (Unbound)

- Constructor

```
(arg0, rest) ->
arg0.instanceName(rest)
```
can be
```
ClassName::instanceName
```
(arg0 is an instance of type
ClassName)

```
interface Finder{
    int find(String s1, String s2);
}
```

Finder finder = (s1, s2) -> s1.indexOf(s2);

⬇

Finder finder = String::indexOf;

int index = finder.find("this is a test", "test");

Method Reference

- Static method
- Instance method (Bound)
- Instance method (Unbound)

- **Constructor**

Syntax
```
ClassName::new
ArrayTypeName[]::new
```

Example
```
String::new
Person[]::new
```

Lambda Equivalent
```
() -> new String()
size -> new Person[size]
```

Lecture 3

- Functional Programming

- **Lambda Expressions**
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - **Java Functional Interfaces**

Why do we need functional interfaces?

- Think of it as simply "functions"

- A functional interface defines the structure of that function (i.e., parameter types and return type)

- Functions can be used as parameters, and simplified using lambdas, thus making the code more compact

https://stackoverflow.com/questions/28417262

Built-in Functional Interfaces

- The `java.util.function` package

- Well defined set of general-purpose functional interfaces
  - All have only one abstract method
  - Lambda can be used wherever these interfaces are used
  - They are used extensively in Java class libraries, especially with the Streams API (later)

Consumer<T>

represents a function that takes an argument of type T and returns nothing (consume it)

```java
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```

```java
public void forEach(Consumer<? super E> action)
```

void accept(T t)

Performs this operation on the given argument.

Supplier<T>

The `Supplier` interface has one abstract method `T get()`

represents a function that takes no argument and returns (supplies) a value of type T

```java
Supplier<String> textSupplier = () -> "Hello!";
Supplier<Integer> integerSupplier = () -> 123;
Supplier<Double> randomSupplier = () -> Math.random();
Supplier<Double> randomSupplierMR = Math::random;

System.out.println(textSupplier.get());
System.out.println(integerSupplier.get());
System.out.println(randomSupplier.get());
System.out.println(randomSupplierMR.get());
```

Predicate<T>

The `Predicate` interface has one abstract method `boolean test(T t)`

represents a function that takes a value of type T and returns a boolean

```
default boolean removeIf(Predicate<? super E> filter)
```

Removes all of the elements of this collection that satisfy the given predicate.

```java
List<String> list = new ArrayList<>();
list.add("This");
list.add("is");
list.add("a");
list.add("Java");
list.add("Course");
list.removeIf(e -> e.length()<3);
list.forEach(System.out::println);
```

Function<T, R>
represents a function that takes a value of type T and returns a value of type R

```
Function<String, Integer> strLenFunction = String::length;
String inputString = "Hello, World!";
int length = strLenFunction.apply(inputString);
```

```
Function<Integer,int[]> arrayCreator = int[]::new;
int[] intArray = arrayCreator.apply(5);
System.out.println(Arrays.toString(intArray)); // [0,0,0,0,0]
```

Common Functional Interfaces

The Operator interfaces represent functions whose result and argument types are the same

The Predicate interface represents functions who take an argument and return a boolean

The Function interface represents functions whose result and argument types could differ

Interface	Function Signature	Example
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

Table from "Effective Java"

More Use Cases

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach5

Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, su unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a na class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-me

This section covers the following topics:

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

Next Lecture

- Java 8 Stream API