Lecture 7
summary of Java SE

presentation

Java Programming – Software App Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 7 – Summary of JSE

Lambda Expressions, Method References, Functional Interfaces, Default method, Stream API, Anonymous, Inner classes, Call-back, Closure

Java 8 Features and Pre-requisites

1. Java 8 Features

Java 8 New Features:

JAVA 8 (aka JDK 1.8) is a major release of JAVA programming language development. Its initial version was released on 18 March 2014.

With the Java 8 release, Java provided support for functional programming, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

- **Lambda expression** – Adds functional processing capability to Java.
- **Method references** – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- **Default method** – Interface to have default method implementation.
- **New tools** – New compiler tools and utilities are added like 'jdeps' to figure out dependencies.
- **Stream API** – New stream API to facilitate pipeline processing.
- **Date Time API** – Improved date time API.
- **Optional** – Emphasis on best practices to handle null values properly.
- **Nashorn, JavaScript Engine** – A Java-based engine to execute JavaScript code.

Along with these new featuers, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

1. Java 8 Features

Why languages evolve?

- To meet developers expectations
- To remain relevant
- To keep up-to-date with hardware advancements
- Security fixes
  Better approaches to perform certain task

Why you should learn Java 8?

- To embrace functional programming paradigm
  - Lambdas
  - Declarative data processing
- New and improved API's and technology
  - Nashorn JavaScript Scripting Engine
  - Date and Time API
  - Stream API
  - Concurrency utilities
- Improved support for clean API design
  - Optional
  - Interface default and static methods

1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature – Pre-requisite for Closure and Callbacks:

With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses **invokedynamics** feature, introduced in Java 7 to improve performance.

jjs

For Nashorn engine, JAVA 8 introduces a new command line tool, **jjs,** to execute javascript codes at console.

```
//sample.js
print('Hello World!');
```

Open console and use the following command.

```
$jjs sample.js
```

It will produce the following output:

```
Hello World!
```

1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature:

jjs in Interactive Mode

Open the console and use the following command.

```
$jjs

jjs> print("Hello, World!")
Hello, World!

jjs> quit()

>>
```

Calling JavaScript from Java

Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.

1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature:

```
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;

public class Java8JSTester {
   public static void main(String args[]) {
      ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
      ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");

      String name = ”Hello name ";
      Integer result = null;

      try {
         nashorn.eval("print('" + name + "')");
         result = (Integer) nashorn.eval("10 + 2");

           } catch(ScriptException e) {
                 System.out.println("Error executing script: "+ e.getMessage());
         }
      System.out.println(result.toString());
   }
}
```

1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature:

```
import java.io.FileReader;
import java.nio.file.Path;
import java.nio.file.Paths;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class HelloWorldJSFile {
  public static void main(String[] args) throws Exception {
    ScriptEngineManager m = new ScriptEngineManager();
    // Sets up Nashorn JavaScript Engine
    ScriptEngine e = m.getEngineByExtension("js");
    // Nashorn JavaScript syntax.
   e.eval("print('Hello, ')");
    // world.js contents: print('World!\n');
    Path p1 =
Paths.get("/home/stud/javase/lectures/c06/src/nashornjs/word.
js");
    e.eval(new FileReader(p1.toString()));
  }
}
```

1.1 Java 8 Nashorn JavaScript Engine

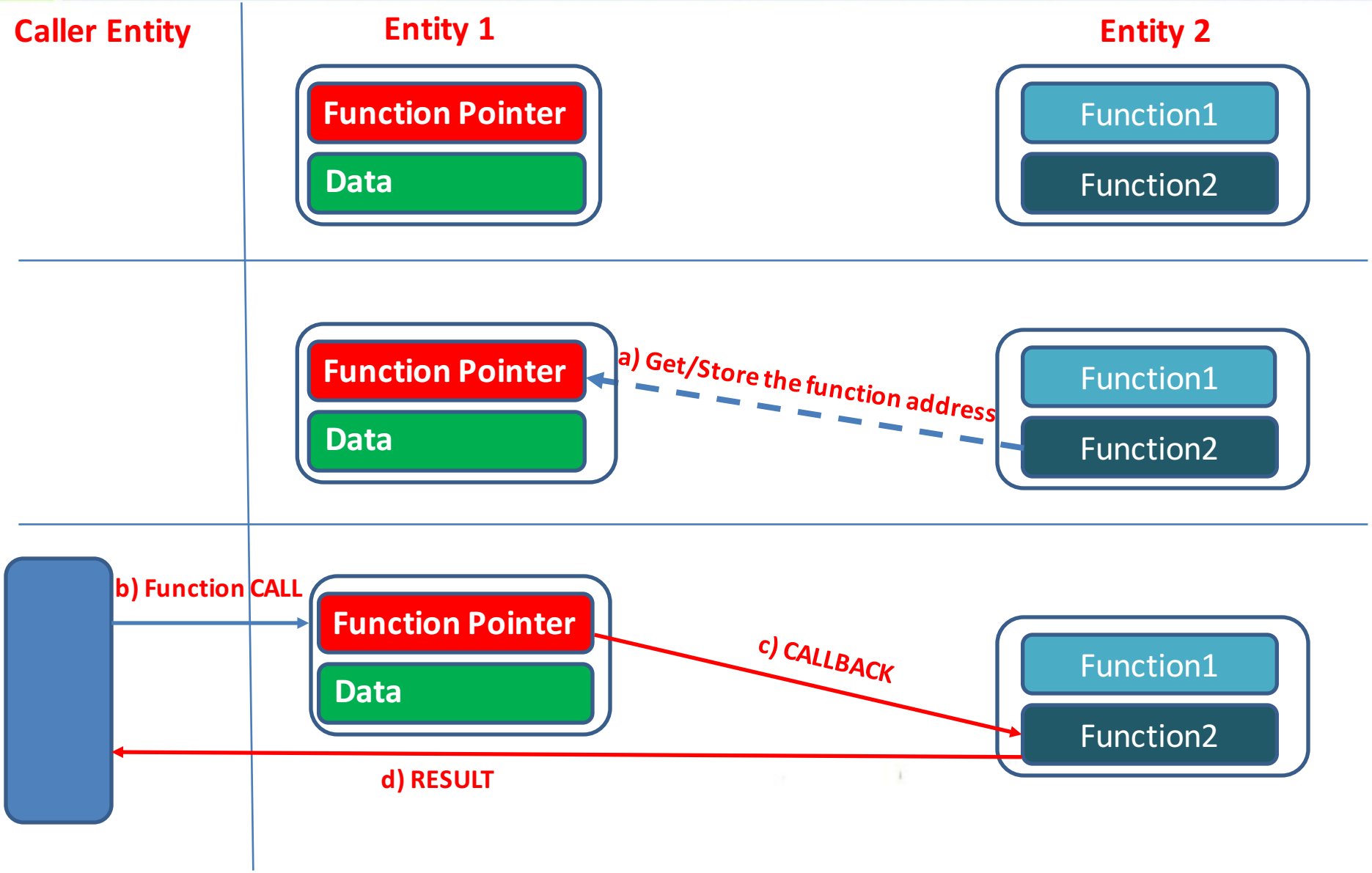
Java 8 JavaScript Nashorn Feature:

Calling Java from JavaScript

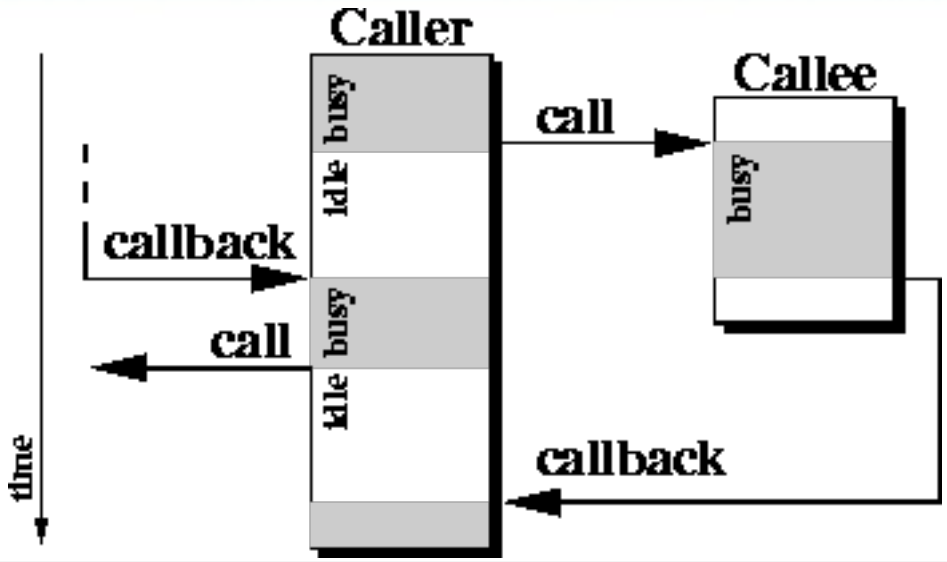
The following example explains how to import and use Java classes in java script –

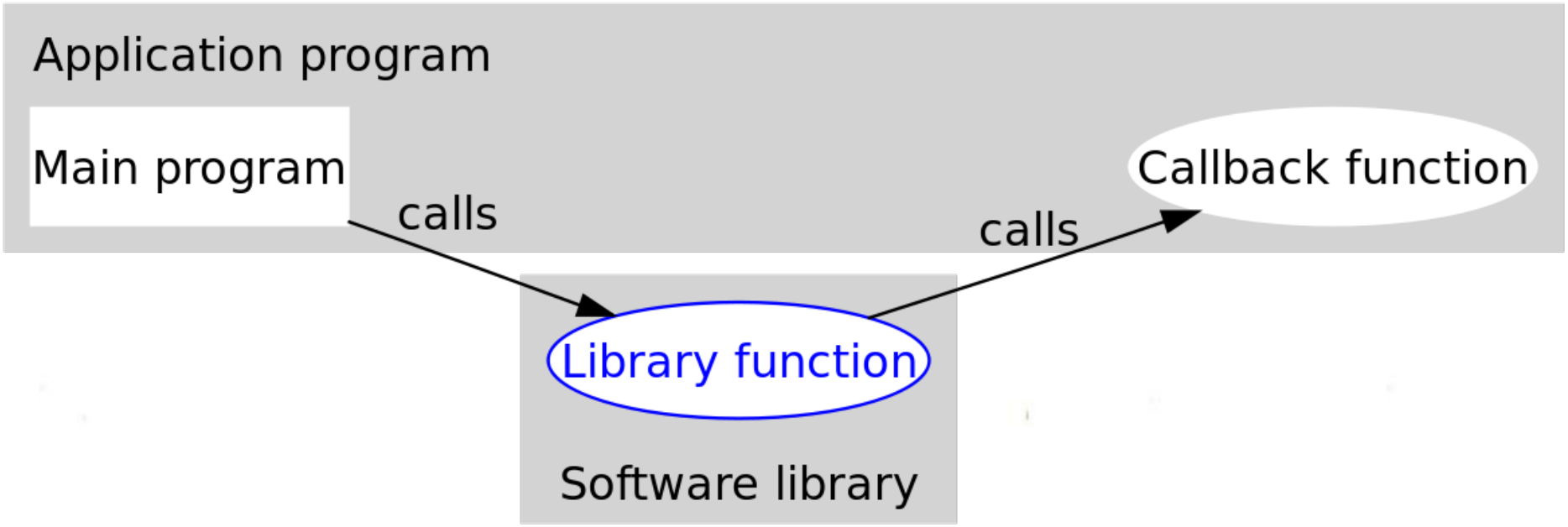
```
//sampleBigDecimal.js
var BigDecimal = Java.type('java.math.BigDecimal');

function calculate(amount, percentage) {

    var result = new BigDecimal(amount).multiply(
    new BigDecimal(percentage)).divide(new BigDecimal("100"), 2,
BigDecimal.ROUND_HALF_EVEN);

    return result.toPlainString();
}

var result = calculate(5680000000000000023,13.9);
print(result);
```

1.1 Callback Overview



1.1 Callback Overview





1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature – Pre-requisite for Closure and Callbacks:

```
// define our function with the callback argument
function some_function1(arg1,  arg2, callback) {
        // this generates a random number between
        // arg1 and arg2
        var my_number = arg1 + arg2;
        // then we're done, so we'll call the callback and
        // pass our result
        callback(my_number);
}

function fcalled2(num) {
        // this anonymous function will run when the
        // callback is called
        print("callback called! " + num);
}

// call the function
some_function1(5,  15, fcalled2);
```

Closure is how the one is building it, **the callback** is how the one is **using it**.

A callback can be implemented (build):

- **as a closure (in languages that have them) or;**
- **an implementation of an interface (in Java, as an anonymous inner class or a regular class).**

1.1 Java 8 Nashorn JavaScript Engine

Java 8 JavaScript Nashorn Feature – Pre-requisite for Closure and Callbacks:

```
// define our function with the callback argument
function some_function(arg1,  arg2, callback) {
        // this generates a random number between
        // arg1 and arg2
        var my_number = arg1 + arg2;
        // then we're done, so we'll call the callback and
        // pass our result
        callback(my_number);
}

// call the function (callback usage) by closure implementation
some_function(5,  15, function(num)  {
        // this anonymous function will run when the
        // callback is called
        print("callback called! " + num);
});
```

Closure is how the one is building it, **the callback** is how the one is **using it**.

A callback can be implemented (build):

- **as a closure (in languages that have them) or;**
- **an implementation of an interface (in Java, as an anonymous inner class or a regular class).**

1.1 Java 8 Nashorn JavaScript Engine

Java POJO Callbacks!?:

```
interface CallBack {
    void methodToCallBack();
}

class CallBackImpl implements CallBack {
    public void methodToCallBack() {
        System.out.println("I've been called back");
    }
}

class Caller {

    public void register(CallBack callback) {
        callback.methodToCallBack();
    }

    public static void main(String[] args) {
        Caller caller = new Caller();
        CallBack callBack = new CallBackImpl();
        caller.register(callBack);
    }
}
```

1.2 Java Inner Class

Java inner class or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class {
 //code
 class Java_Inner_class {
  //code
 }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

1.2 Java Nested Classes

Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

1. Non-static nested class(inner class)
    - a)Member inner class
    - b)Annomynous inner class
    - c)Local inner class
2. Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1.2 Java Inner Class

Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1 {
 private int data=30;
 class Inner {
  void msg(){System.out.println("data is "+data);}  //it will print 30
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

1.2 Java Inner Class

Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

- Class (may be abstract or concrete).
- Interface

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Java anonymous inner class example using class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

1.2 Java Inner Class

Java Local inner class

A class i.e. created inside a method is called local inner class in Java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Rules:

Local variable can't be private, public or protected.

Local inner class cannot be invoked from outside the method.

Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Example of local inner class with local variable

```
class localInner2{
 private int data=30;//instance variable
 void display(){
  int value=50;//local variable must be final till jdk 1.7 only
  class Local{
   void msg(){System.out.println(value);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner2 obj=new localInner2();
  obj.display();
 }
}
```

1.2 Java Inner Class

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

1. Java static nested class example with instance method

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }// In this example, you need to create the
   // instance of static nested class because
} // it has instance method msg().
```

2. Java static nested class example with static method

```
class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg();//no need to create the instance of static nested class
  } // If you have the static member inside static nested class, you don't
    // need to create instance of static nested class.
}
```

1.2 Java Inner Class

Java Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

1.2 Java Inner Class

Example of nested interface which is declared within the interface	Example of nested interface which is declared within the class
In this example, we are going to learn how to declare the nested interface and how we can access it.	In this example, we see how can we define an interface inside the class and how can we access it.

```
interface Showable {

  void show();

  interface Message {
   void msg();
  }
}

class TestNestedInterface8 implements Showable.Message {

 public void msg() { System.out.println("Hello nested interface"); }

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface8();//upcasting here
  message.msg();
 }

}
```

```
class A {
  interface Message {
   void msg();
  }
}

class TestNestedInterface9 implements A.Message {
 public void msg() { System.out.println("Hello nested interface"); }

 public static void main(String args[]) {
  A.Message message = new TestNestedInterface9();//upcasting here
  message.msg();
 }
}
```

1.2 Java Inner Class

You van use them sometimes as a syntax hack for Map instantiation:

```
Map map = new HashMap() {{ put("key", "value"); }};
```

vs

```
Map map = new HashMap(); map.put("key", "value");
```

It saves some redundancy when doing a lot of put statements. However, I have also run into problems doing this when the outer class needs to be serialized via remoting.

To be clear, the first set of braces is the anonymous inner class (sub-classing HashMap). The second set of braces is an instance initializer (rather than a static one) which then sets the values on your HashMap subclass.

1. Java Inner Class and Lambda Usage

When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

As mentioned in the section Nested Classes, nested classes enable you to logically group classes that are only used in one place, increase the use of encapsulation, and create more readable and maintainable code.

Local classes, anonymous classes, and lambda expressions also impart these advantages; however, they are intended to be used for more specific situations:

- Local class: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

- Anonymous class: Use it if you need to declare fields or additional methods.

- Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.
    - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

- Lambda expression:
    - Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
    - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

Section Conclusion

Fact: **Java 8 Features**

In few **samples** it is simple to remember: Java 8 new features.

Lambda Expressions, Method References, Functional Interfaces, Default Methods, Streams, Optional Class, Nashorn JavaScript, New Date/Time API, Base64, New I/O

Java 8 Features Details

2.1 Java 8 Method Reference

Method references help to point to methods by their names. A method reference is described using :: (double colon) symbol. A method reference can be used to point the following types of methods –

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

```
import java.util.List;
import java.util.ArrayList;

public class Java8TesterMethRef {
   public static void main(String args[]){
     List<String> names = new ArrayList<String>();

   names.add("Diana");
    names.add("Mariah");
   names.add("Dan");
   names.add("Steve");
   names.add("Mike");

      names.forEach(System.out::println);
  }
}
```

2.2 Java 8 Default Methods

Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8. For example, 'List' or 'Collection' interfaces do not have 'forEach' method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

Syntax

```
public interface vehicle {
   default void print(){
      System.out.println("I am a vehicle!");
   }
}
```

2.2 Java 8 Default Methods

Multiple Defaults

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.

```
public interface vehicle {
   default void print(){
      System.out.println("I am a vehicle!");
   }
}

public interface fourWheeler {
   default void print(){
      System.out.println("I am a four wheeler!");
   }
}
```

First solution is to create an own method that overrides the default implementation.

```
public class car implements vehicle, fourWheeler {
   default void print(){
      System.out.println("I am a four wheeler car vehicle!");
   }
}
```

Second solution is to call the default method of the specified interface using super.

```
public class car implements vehicle, fourWheeler {
   default void print(){
      vehicle.super.print();
   }
}
```

2.2 Java 8 Default Methods

Static Default Methods

An interface can also have static helper methods from Java

```
public interface vehicle {
   default void print(){
      System.out.println("I am a vehicle!");
   }

   static void blowHorn(){
      System.out.println("Blowing horn!!!");
   }
}
```

Default Method Example

Example to get more clarity on **default method**. Please write the following program in an code editor, understand and verify the results.

```
public class Java8Tester {
   public static void main(String args[]){
      Vehicle vehicle = new Car();
      vehicle.print();
   }
}

interface Vehicle {
   default void print(){
      System.out.println("I am a vehicle!");
   }

   static void blowHorn(){
      System.out.println("Blowing horn!!!");
   }
}

interface FourWheeler {
   default void print(){
      System.out.println("I am a four wheeler!");
   }
}

class Car implements Vehicle, FourWheeler {
   public void print(){
      Vehicle.super.print();
      FourWheeler.super.print();
      Vehicle.blowHorn();
      System.out.println("I am a car!");
   }
}
```

2.3 Java 8 Lambda Expressions – Functional Programming

"**Lambda expressions (λEs),** also known as closures, provide a means to represent anonymous methods. Supported by Project Lambda, λEs allow for the creation and use of single method classes. These methods have a basic syntax that provides for the omission of modifiers, the return type, and optional parameters. The specification for λEs is set out in JSR 335, which is divided into seven parts: functional interfaces, lambda expressions, method and constructor references, poly expressions, typing and evaluation, type inference, and default methods.

"Modern IDEs have features to convert anonymous inner classes to lambda expressions."

Microsoft: "Functional programming is a form of **declarative programming**. In contrast, most mainstream **languages**, including object-oriented **programming** (OOP) **languages** such as C#, C++, and Java, were designed to primarily support **imperative** (procedural) **programming"**

Bruce Eckel: ***"I don't see an OOP versus FP (functional programming) debate here; that is not my intention. Indeed, I don't really see a "versus" issue. OO is good for abstracting over data (and just because Java forces objects on you doesn't mean that objects are the answer to every problem), while FP is good for abstracting over behavior. Both paradigms are useful, and mixing them together has been even more useful for me, both in Python and now in Java 8."***

2.3 Java 8 Lambda Expressions – Functional Programming

Programming paradigms

- A programming paradigm is a fundamental style of computer programming.
  - Imperative, declarative, functional, object-oriented, procedural, logical, etc.
- A programming language can follow one or more paradigms.

Imperative vs Functional Paradigm

- Imperative programming paradigm
  - Computation: Uses statements that change a program's state.
  - Program: Consists of sequence of commands that tell a program how it should achieve the result.
- Functional programming paradigm
  - Computation: Evaluation of expressions
  - Expression: Formed by using functions to combine basic values.
  - Focusses on what the program should accomplish rather than how

2.3 Java 8 Lambda Expressions – Functional Programming

Functional Programming

- Functional programming was invented in 1957
  - Before Object Oriented programming
  - Before structured programming
- Memory was too expensive to make functional programming practical
- "*Functional programming is so called because a program consists entirely of functions.*" - John Hughes, Why Functional Programming Matters

What is a function?

- A side effect free computation that always result in same value when passed with same argument.

Why should we embrace functional programming? (This does NOT mean to not use OOP)

- Functional programs are simpler
- No side effects
- Fewer concurrency issues

2.3 Java 8 Lambda Expressions – Functional Programming

Functional Languages

- Haskell / ML / Erlang / Clojure
- F# - (Hybrid)
- Scala - (Hybrid)
- Python (Hybrid)
- Java 8 - (Hybrid)

Key Functional Programming

- Function
  - A side effect free computation that always result in same value when passed with same argument.
- Higher order function
  - function that takes function as argument or return functions.
- Referential transparency
  - allows you to replace function with its value
- Recursion
  - function calls itself
- Lazy evaluation
  - function is not evaluated until required

2.3 Java 8 Lambda Expressions – Functional Programming

Lambda Expressions

- A new feature introduced in Java 8
- A representation of anonymous function that can be passed around.
- Allows you to pass behavior as code that can be executed later
- Allows you to encapsulate changing behavior in a lambda expression
- Earlier this was achieved via the use of anonymous inner classes

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

- The (first, second) are parameters of the compare method
- first.length() - second.length() is the function body
- -> is the lambda operator

2.3 Java 8 Lambda Expressions

Java 8 Lambda Programming Style:

Lambda expressions typically include a parameter list, a return type, and a body.

(parameter list) -> { statements; }

Examples of λEs include:

```
() -> 66
(x,y) -> x + y
(Integer x, Integer y) -> x*y
(String s) -> { System.out.println(s); }
```

2.3 Java 8 Lambda

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.

Syntax

A lambda expression is characterized by the following syntax –

```
(parameter list) -> { statements; }
```

Following are the important characteristics of a lambda expression –

- **Optional type declaration** – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

2.3 Java 8 Lambda & FI – Functional Interface

"**λEs must have a functional interface (FI).** An FI is an interface that has one abstract method and zero or more default methods. FIs provide target types for lambda expressions and method references, and ideally should be annotated with @FunctionalInterface to aid the developer and compiler with design intent."

```
//sort using java 7
private void sortUsingJava7(List<String> names) {
  Collections.sort(names, new Comparator<String>() {
    @Override
         public int compare(String s1, String s2) {
      return s1.compareTo(s2);
    }
  });
}
```

```
@FunctionalInterface
public interface Comparator<T>{
        // Only one abstract method allowed
        int compare(T o1, T o2);
        // Overriding allowed
        boolean equals(Object obj);
        // Optional default methods allowed
}
```

```
//sort using java 8
private void sortUsingJava8(List<String> names){
 Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
}
```

2.3 Java 8 Lambda and FI Sample

```
public class Java8Tester {

   final static String salutation = "Hello! ";

   public static void main(String args[]){
      GreetingService greetService1 = message ->
      System.out.println(salutation + message);
      greetService1.sayMessage(”John");
  }

  @FunctionalInterface
  interface GreetingService {
      void sayMessage(String message);
  }
}
```

2.3 Java 8 Lambda and FI Sample

```
class LengthComparator implements Comparator<String> {
     public int compare(String first, String second) {
        return Integer.compare(first.length(), second.length());
     }
  }

Arrays.sort(strings, new LengthComparator());
```

What are first and second? They are both strings! Java is a strongly typed language, and we must specify that as well:

```
Integer.compare(first.length(), second.length())
```

```
(String first, String second)
     -> Integer.compare(first.length(), second.length())
```

The lambda expression from above is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters. Had he known about the Java API, he would have written:

```
λfirst.λsecond.Integer.compare(first.length(), second.length())
```

Java 8 Programming Style:

```
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class Java8TesterLambdaFullComparatorFI {
   public static void main(String args[]){

      List<String> names1 = new ArrayList<String>();
      names1.add("Mary "); names1.add("Sam ");
      names1.add("Robert ");names1.add("Nick ");

      List<String> names2 = new ArrayList<String>();
      names2.add("Mary "); names2.add("Sam ");
      names2.add("Robert "); names2.add("Nick ");
      Java8Tester tester = new Java8Tester();
      System.out.println("Sort using Java 7 syntax: ");

      tester.sortUsingJava7(names1);
      System.out.println(names1);
      System.out.println("Sort using Java 8 syntax: ");

      tester.sortUsingJava8(names2);
      System.out.println(names2);
   }
```

Java 8 New Features:

```
//sort using java 7
private void sortUsingJava7(List<String> names){
    Collections.sort(names, new Comparator<String>() {
  @Override
        public int compare(String s1, String s2) {
    return s1.compareTo(s2);
    }
  });
}

//sort using java 8
private void sortUsingJava8(List<String> names){
    Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
}
}
```

2.3 Java 8 Lambda and Functional Interface Sample

```
public class Java8TesterLambda {
   public static void main(String args[]){
      Java8Tester tester = new Java8Tester();
      //with type declaration
      MathOperation addition = (int a, int b) -> a + b;
      //with out type declaration
      MathOperation subtraction = (a, b) -> a - b;
      //with return statement along with curly braces
      MathOperation multiplication = (int a, int b) -> { return a * b; };
      //without return statement and without curly braces
      MathOperation division = (int a, int b) -> a / b;

      System.out.println("10 + 5 = " + tester.operate(10, 5, addition));
      System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));
      System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));
      System.out.println("10 / 5 = " + tester.operate(10, 5, division));

      //with parenthesis
      GreetingService greetService1 = message -> System.out.println("Hello " + message);
      //without parenthesis
      GreetingService greetService2 = (message) -> System.out.println("Hello " + message);

      greetService1.sayMessage("Jake");
      greetService2.sayMessage("John");
   }
...

}
```

```
interface MathOperation {
    int operation(int a, int b);
}

interface GreetingService {
  void sayMessage(String message);
}

private int operate(int a, int b, MathOperation
mathOperation){
  return mathOperation.operation(a, b);
}
```

2.3 Java 8 Functional Interface

Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method '**compareTo**' is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions.

Following is the list of several functional interfaces defined in java.util.Function package.

Samples with pre-defined functional interface such as: Function<T, R> and Predicate<T>

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
Runnable	none	void	run	Runs an action without arguments or return value	
Supplier<T>	none	T	get	Supplies a value of type T	
Consumer<T>	T	void	accept	Consumes a value of type T	andThen
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U	andThen
Function<T, R>	T	R	apply	A function with argument of type T	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A Boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A Boolean-valued function with two arguments	and, or, negate

2.3 Java 8 Functional Interface Sample

Predicate <T> interface is a functional interface with a method test(Object) to return a Boolean value. This interface signifies that an object is tested to be true or false.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Java8TesterFI {
   public static void main(String args[]){
   List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

      // Predicate<Integer> predicate = n -> true
      // n is passed as parameter to test method of Predicate interface
      // test method will always return true no matter what value n has.

      System.out.println("Print all numbers:");

      //pass n as parameter
   eval(list, n->true);

   // Predicate<Integer> predicate1 = n -> n%2 == 0
      // n is passed as parameter to test method of Predicate interface
      // test method will return true if n%2 comes to be zero
         ...
  }
}
```

2.3 Java 8 Functional Interface Sample

```
    //Predicate<Integer> predicate1 = n -> n%2 == 0
        // n is passed as parameter to test method of Predicate interface
        // test method will return true if n%2 comes to be zero

        System.out.println("Print even numbers:");
    eval(list, n-> n%2 == 0 );

        // Predicate<Integer> predicate2 = n -> n > 3
        // n is passed as parameter to test method of Predicate interface
        // test method will return true if n is greater than 3.

        System.out.println("Print numbers greater than 3:");
    eval(list, n-> n > 3 );
  }

    public static void eval(List<Integer> list, Predicate<Integer> predicate) {
        for(Integer n: list) {

      if(predicate.test(n)) {
        System.out.println(n + " ");
      }
    }
  }
}
```

2.3 Java 8 Functional Interface & Lambda Sample

```
// AreLambdasClosures.java
import java.util.function.*;

public class AreLambdasClosures {
    public Function<Integer, Integer> make_fun() {
        // Outside the scope of the returned function:
        int n = 0;
        return arg -> {
            System.out.print(n + " " + arg + ": ");
            arg += 1;
            // n += arg; // Produces error message
            return n + arg;
        };
    }
    public void try_it() {
        Function<Integer, Integer>
            x = make_fun(),
            y = make_fun();
        for(int i = 0; i < 5; i++)
            System.out.println(x.apply(i));
        for(int i = 10; i < 15; i++)
            System.out.println(y.apply(i));
    }
    public static void main(String[] args) {
        new AreLambdasClosures().try_it();
    }
}
```

2.3 Java 8 Functional Interface & Lambda Sample

Type Inference

- The act of inferring a type from context is called Type Inference.
- In most cases, javac will infer the type from context.
- If it can't resolve then compiler will throw an error.

```
Comparator comparator = (first, second) -> first.length() - second.length();
```

Cannot resolve method 'length()'

Lambdas are typed

```
Comparator<String> nameComparator = (first, second) -> second.length() - first.length();
```

- Type of a lambda expression is an interface
- Only interfaces with single abstract method can be used
- These interfaces are called functional interfaces
- You can use @FunctionalInterface annotation to mark your interface a functional interface

2.4 Java 8 Processing Streams

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement –

SELECT max(salary), employee_id, employee_name FROM Employee

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

2.4 Java 8 Processing Streams

Stream Processing

- Why we need a new data processing abstraction?
- Stream vs Collection Using Stream API
- Using Stream API

Why?

- Collection API is too low level
- Developers needed a higher level declarative data processing API
- Processing data in parallel

2.4 Java 8 Processing Streams

What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream :

• **Sequence of elements** – A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

• **Source** – Stream takes Collections, Arrays, or I/O resources as input source.

• **Aggregate operations** – Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

• **Pipelining** – Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. **collect()** method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

• **Automatic iterations** – Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Stream is a sequence of elements from a source supporting data processing operations

- source -> collection with elements
- data processing operations -> filter, map, etc.

2.4 Java 8 Processing Streams vs. Collections

Collection	Stream
Read and write operation	Only read operations. You can't add or remove elements.
Eagerly evaluated	Lazily evaluated
Collections are about data	Streams are for performing computations on data
Client has to iterate over collection >> External iteration	internal iteration
You can iterate over collection multiple times	You can only process a stream once

2.4 Java 8 Processing Streams API

Generating Streams

With Java 8, Collection interface has two methods to generate a Stream:

- **stream()** – Returns a sequential stream considering collection as its source.
- **IntStream()**, **LongStream()**, **DoubleStream()** –working with numeric streams (e.g. methods – of, range, rangeClosed, iterate, generate)
- **parallelStream()** – Returns a parallel Stream considering collection as its source. It creates an instance of parallel stream by calling the parallel operator.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect( Collectors.toList() );
```

Stream Operations

- **Intermediate operations** >> which results in another stream
  - **Map (Function), filter (Predicate), sorted (Comparator), distinct**
- **Terminal operations** >> which produce a non-stream result
  - **collect, forEach**

2.4 Java 8 Processing Streams

Java 8 Data Processing – without Streams

```
public static void main(String[] args) {
    List<Task> tasks = getTasks();

    List<Task> readingTasks = new ArrayList<>();
    for (Task task : tasks) {
        if(task.getType() == TaskType.READING){
            readingTasks.add(task);
        }
    }

    Collections.sort(readingTasks, (t1,t2) -> t1.getCreatedAt().compareTo(t2.getCreatedAt()));
    for (Task readingTask : readingTasks) {
        System.out.println(readingTask.getTitle());
    }


}
```

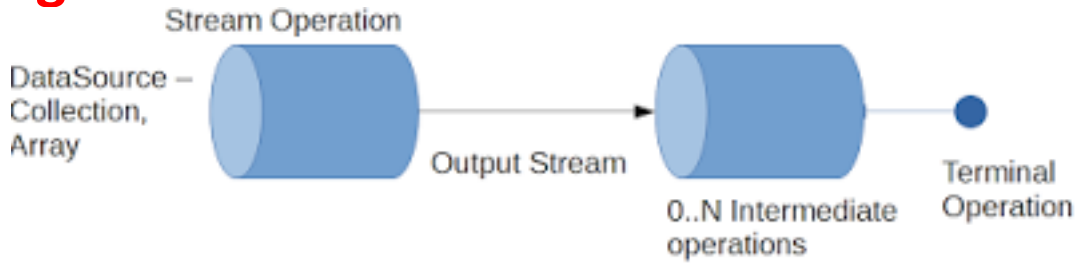
Java 8 Data Processing – with Streams

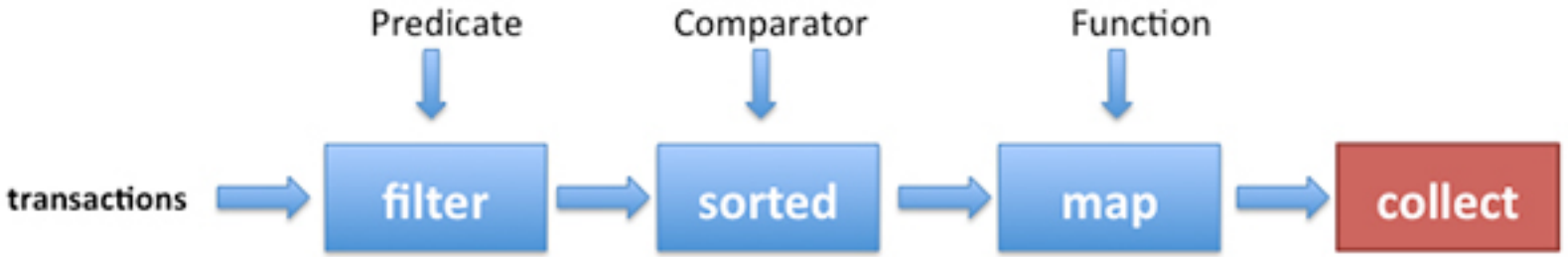
```
public static void main(String[] args) {
    List<Task> tasks = getTasks();
    tasks.stream().
            filter(task -> task.getType() == TaskType.READING).
            sorted().
            map(Task::getTitle).
            forEach(System.out::println);
}
```

// Copyright: https://github.com/shekhargulati/java8-the-missing-tutorial

2.4 Java 8 Processing Streams

Understanding Java 8 Streams Code






```
public static void main(String[] args) {
    List<Task> tasks = getTasks();
    tasks.stream().
            filter(task -> task.getType() == TaskType.READING).
            sorted().
            map(Task::getTitle).
            forEach(System.out::println);
}
```

Creates a stream on the source collection

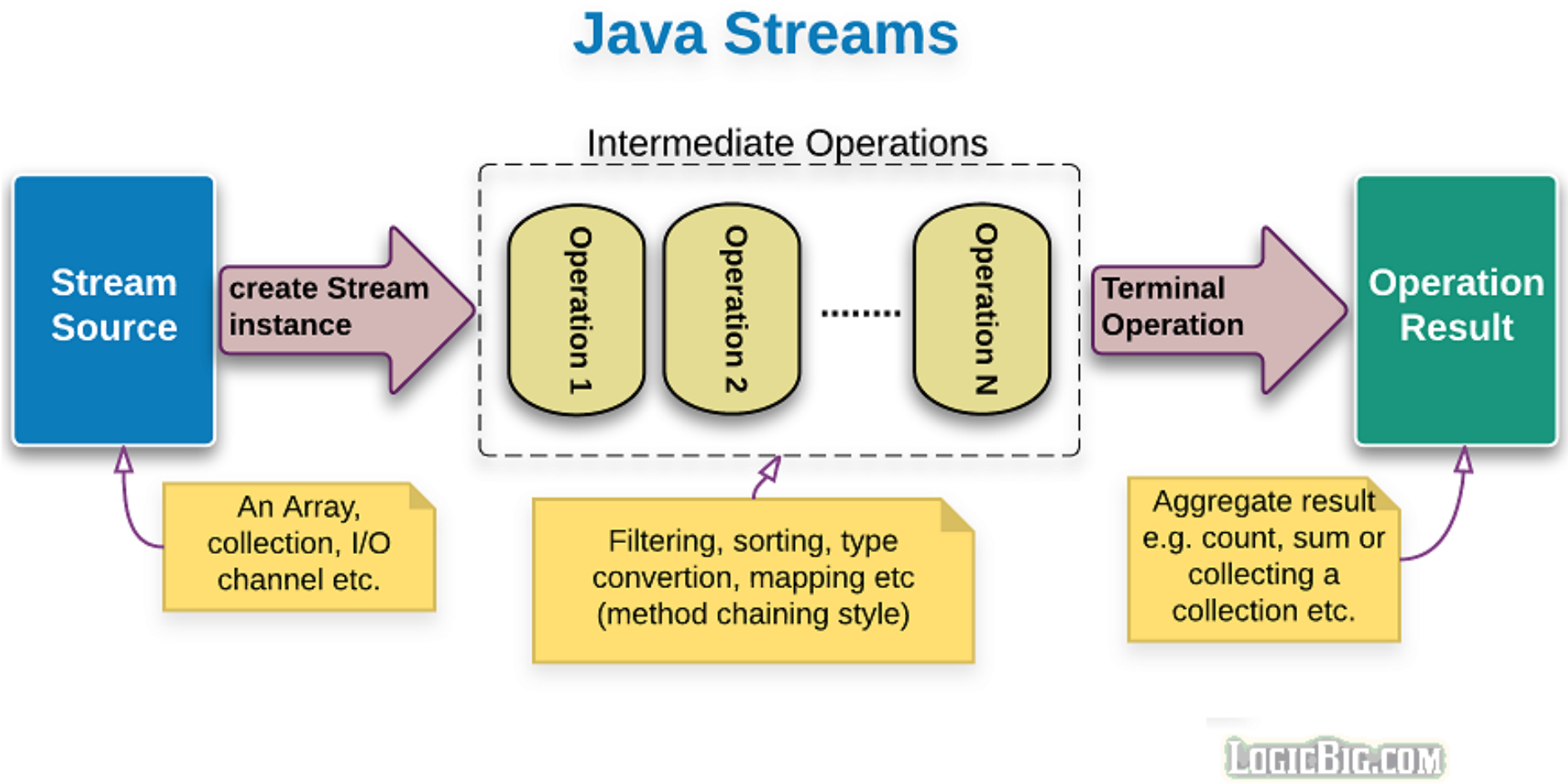
perform data processing operations. Each produce a stream

finally stream is evaluated and each element is printed on the console

// Copyright: https://github.com/shekhargulati/java8-the-missing-tutorial

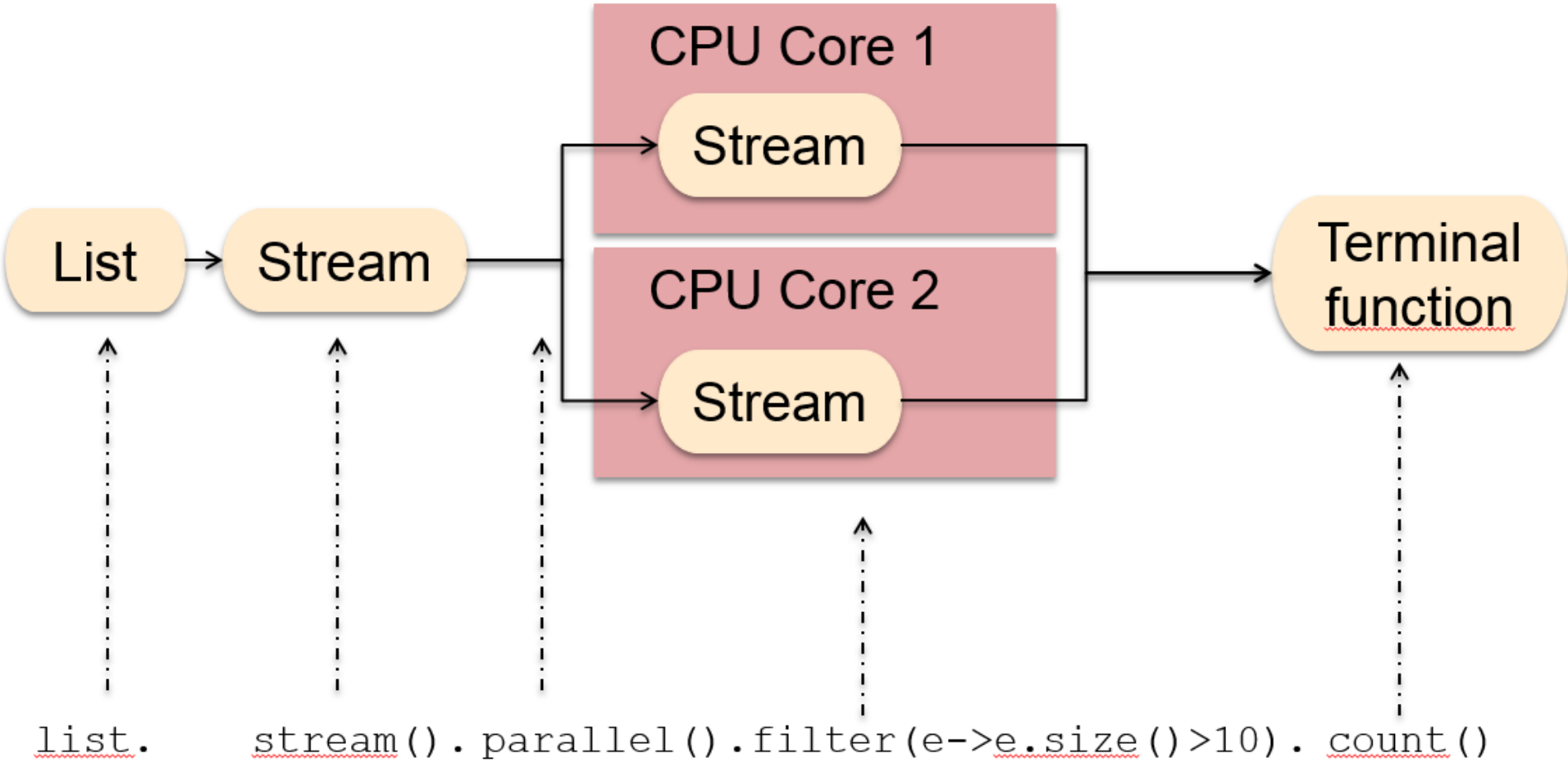
2.4 Java 8 Processing Streams

Understanding Java 8 Streams Code

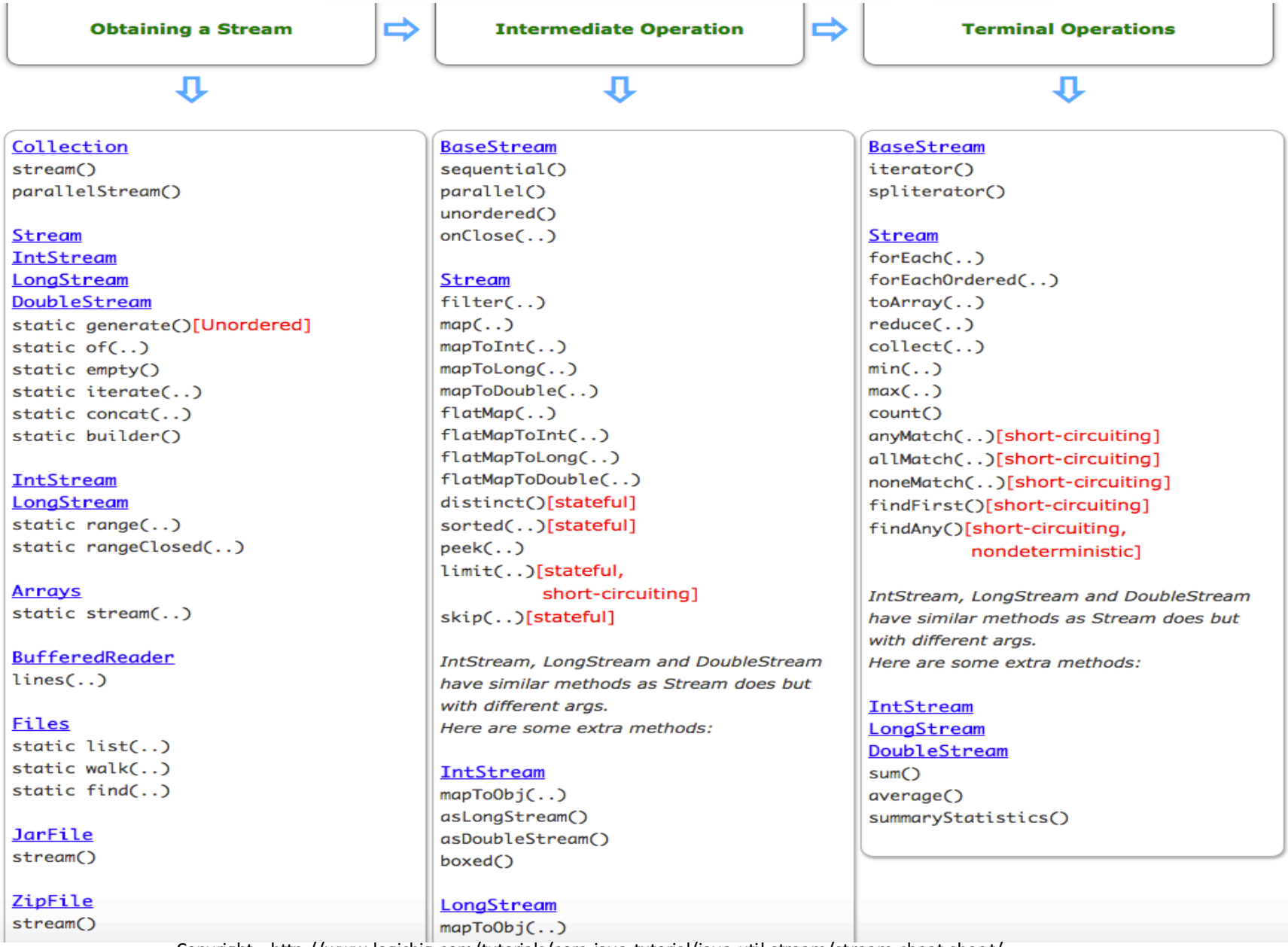


2.4 Java 8 Processing Streams

Understanding Java 8 Streams Code

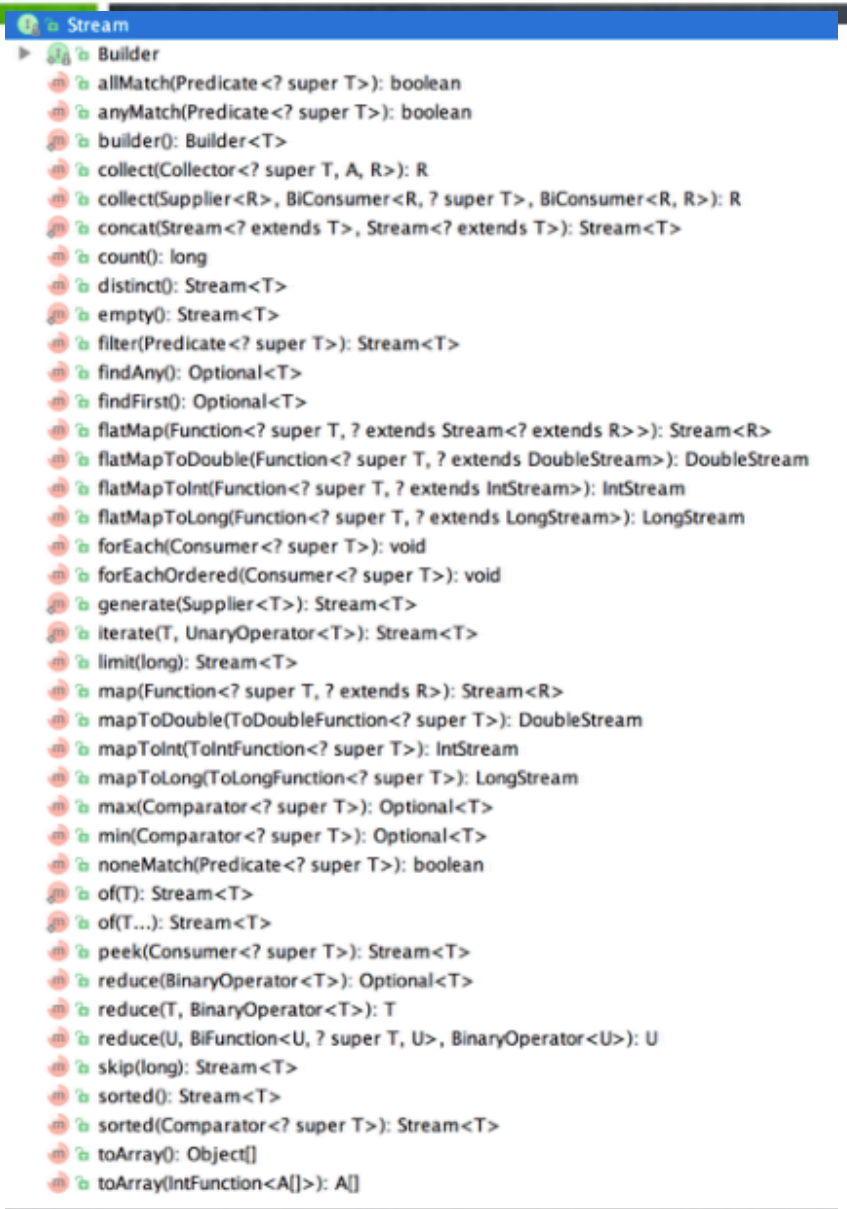


2.4 Java 8 Processing Streams API



Copyright: http://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-cheat-sheet/

2.4 Java 8 Processing Streams API



forEach – Terminal Operation

Stream has provided a new method 'forEach' to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

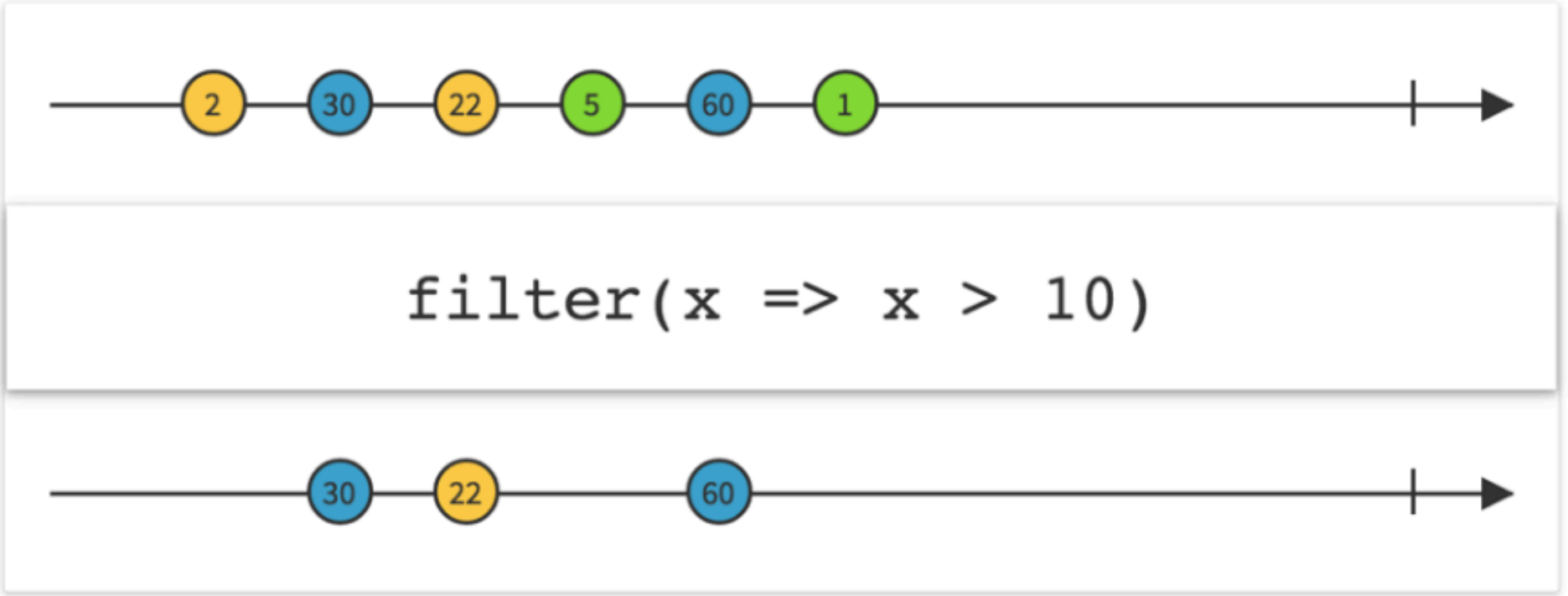
```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

2.4 Java 8 Processing Streams API

Filter – Intermediate Operations : Predicate

*The 'filter' method (**filtering**) is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.*

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl"); //get # of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```



Filter – Intermediate Operations : Predicate

Sample operations (please see the source code from the lecture):

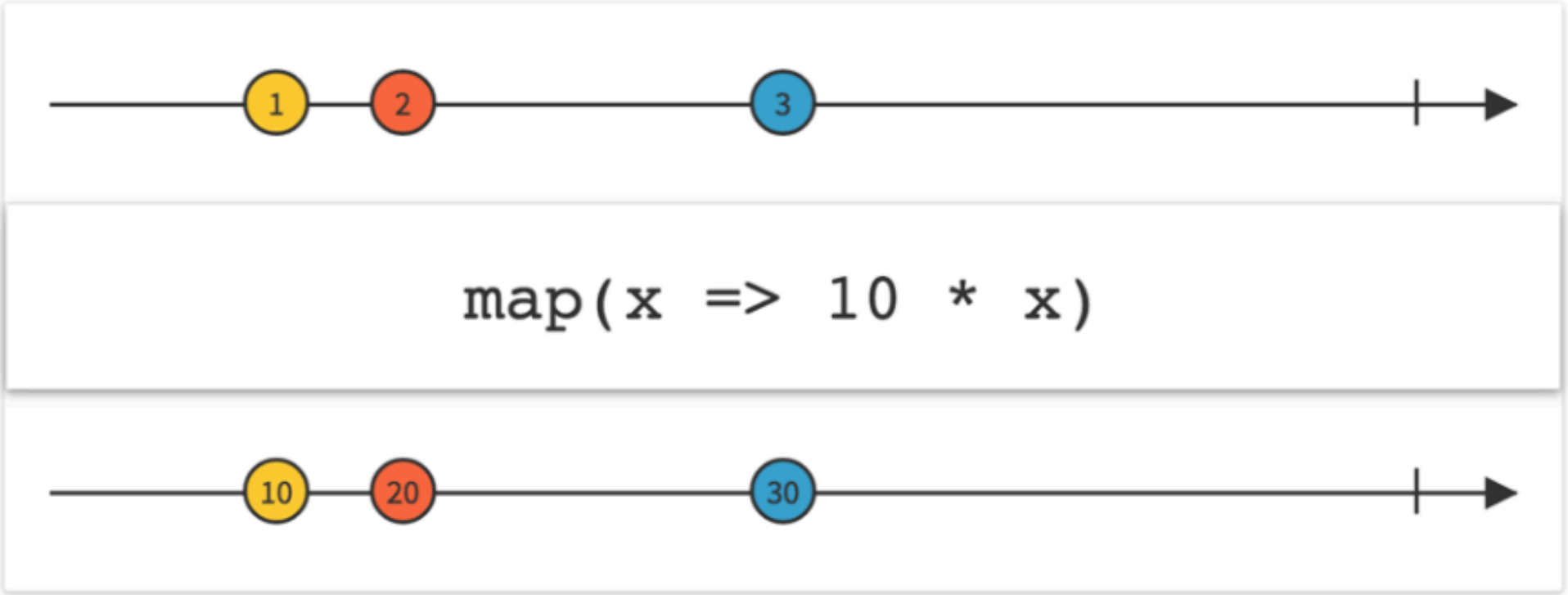
- Find all the reading tasks sorted by their creation date
- Find all distinct tasks and print them
- Find top 5 reading tasks sorted by their creation date
  - Pagination with Skip and Limit Count all reading tasks
- Count all reading tasks

2.4 Java 8 Processing Streams API

map – Intermediate Operations : Function

The 'map' method (mapping) is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5); //get list of unique squares
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```



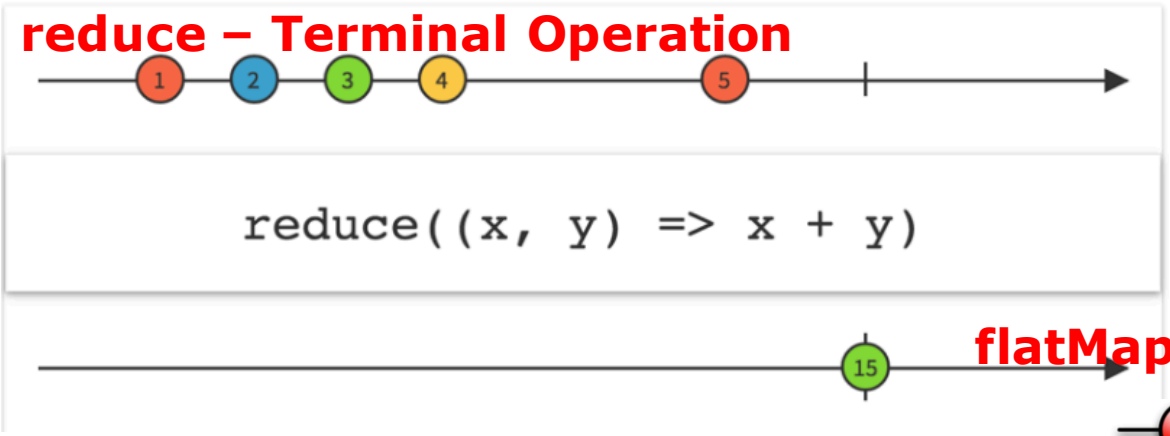
2.4 Java 8 Processing Streams API

limit – Intermediate Operation

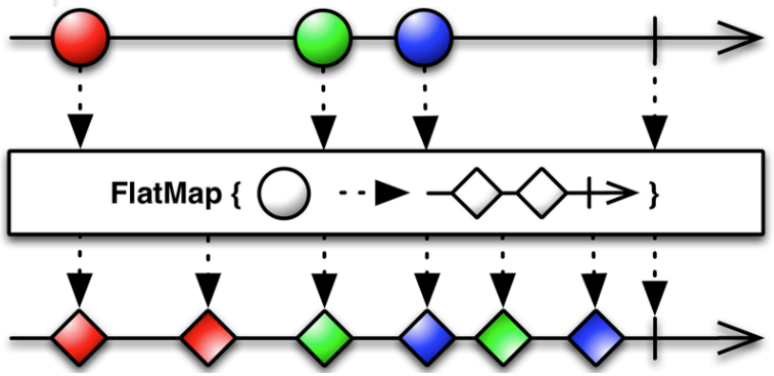
The 'limit' method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

reduce – Terminal Operation



flatMap – Intermediate Operation



2.4 Java 8 Processing Streams API

sorted - Intermediate Operations : Comparator

The 'sorted' method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.out::println);
```

Parallel Processing

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl"); //get # of empty string
int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

It is very easy to switch between sequential and parallel streams.

2.4 Java 8 Processing Streams

Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","",  "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));
System.out.println("Merged String: " + mergedString);
```

Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

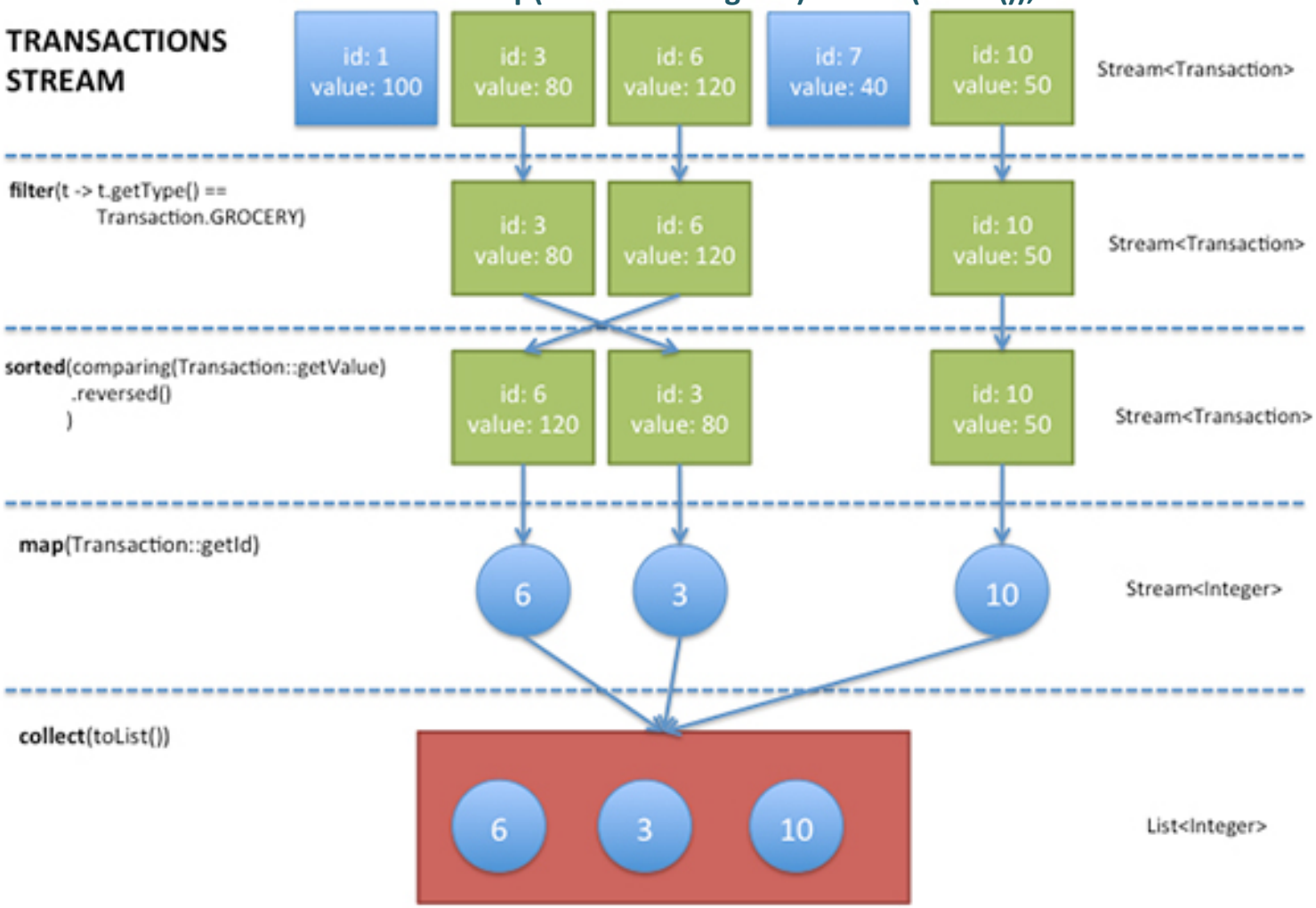
```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest  number in List : " + stats.getMax());
System.out.println("Lowest  number in List : " + stats.getMin());
System.out.println("Sum  of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

2.4 Java 8 Processing Streams

Sample Java 8 Code

Let's say we need to find all transactions of type grocery and return a list of transaction IDs sorted in decreasing order of transaction value.

```
List<Integer> transactionsIds = transactions.stream()
.filter(t -> t.getType() == Transaction.GROCERY)
.sorted(comparing(Transaction::getValue).reversed())
.map(Transaction::getId).collect(toList());
```



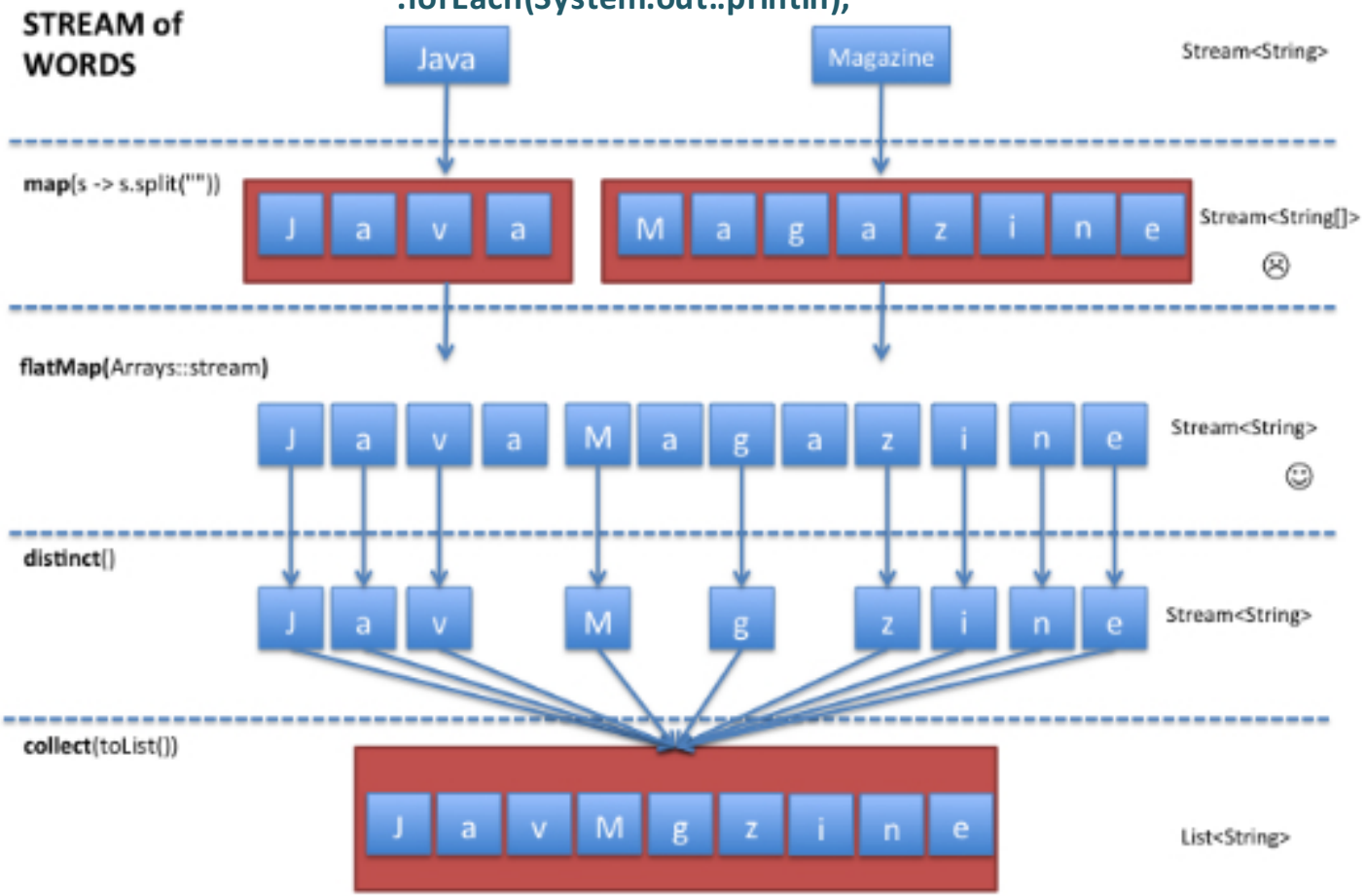
// Copyright:http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

2.4 Java 8 Processing Streams

Sample Java 8 Code

Using the **flatMap** method has the effect of replacing each generated array not by a stream but by the contents of that stream. In other words, all the separate streams that were generated when using map(Arrays::stream) get amalgamated or "flattened" into one single stream.

```
Files.lines(Paths.get("stuff.txt"))
.map(line -> line.split("\\s+")) // Stream<String[]>
.flatMap(Arrays::stream) // Stream<String>
.distinct() // Stream<String>
.forEach(System.out::println);
```



// Copyright: http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html

2.4 Java 8 Processing Streams – Power of Collectors

Java 8 Streams Collectors

- Reduce a stream to a single value
  - collect is a reduction operation
- They are used with collect terminal method

What you can do with collect?

- Reducing stream to a single value
- Group elements in a stream
- Partition elements in a stream

```
List<String> readingTasks = tasks.stream().
        filter(task -> task.getType() == TaskType.READING).
        map(Task::getTitle).
        collect(Collectors.toList());
```

2.4 Java 8 Processing Streams – Power of Collectors

Collectors class

- A utility class that contains static factory methods for most common collectors
  - that collect to a Collection
  - Grouping
  - partitioning

Reducing to a single value

- Single value could be
  - numeric type
  - domain object
  - a collection

Grouping elements

- Group elements by key
- You can do both single level and multilevel grouping

Partitioning

- It is a special case of grouping
- Groups source collection into at most two partitioned by a predicate
- Returned map has following syntax
  - Map<Boolean, List<Task>>

2.4 Java 8 Optional<T>

How many of you have experienced NullPointerException?

Null History

- It was designed by Sir Tony Hoare in 1965
  - creator of Quicksort
- He was designing Algol W language
- null was designed to signify absence of value
- Most programming languages like Java, C++, C#, Scala, etc all have Null
- He called it was a Billion Dollar Mistake
- http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

What could possible go wrong in the source code below?

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

2.4 Java 8 Optional<T>

NullPointerException

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

Task could be null

```
public String taskAssignedTo(String taskId) {
  return taskRepository.find(taskId).getAssignedTo().getUsername();
}
```

User could be null

2.4 Java 8 Optional<T>

We end up writing …

```
public String taskAssignedTo(String taskId) {
  Task task = taskRepository.find(taskId)
  if(task != null){
    User user = task.getAssignedTo() ;
    if(user != null){
        return user.getUsername();
    }
  }
  return null;
}
```

Biggest problem with code

Absence of value is not visible in the API

Possible solutions

- Null Object Pattern << Before Java 8
- Optional << From Java 8

2.4 Java 8 Optional<T>

Optional

- A container that may or may not contain a value
- If a function returns Optional then the client would know that value might not be present
- Common concept is functional languages
  - MayBe, Nothing >> Haskell
  - Option, Some, None >> Scala
  - Optional<T> >> Java 8
  - Type? / Type! >> Swift 2/3
  - Optional >> Guava
- Optional is a container object which is used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values. It is introduced in Java 8 and is similar to what **Optional** is in Guava.

2.4 Java 8 Optional<T>

S. No.	Method & Description
1	**static <T> Optional<T> empty()**Returns an empty Optional instance.
2	**boolean equals(Object obj)**Indicates whether some other object is "equal to" this Optional.
3	**Optional<T> filter(Predicate<? super <T> predicate)**If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.
4	**<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional.
5	**T get()**If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
6	**int hashCode()**Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
7	**void ifPresent(Consumer<? super T> consumer)**If a value is present, it invokes the specified consumer with the value, otherwise does nothing.
8	**boolean isPresent()**Returns true if there is a value present, otherwise false.
9	**<U>Optional<U> map(Function<? super T,? extends U> mapper)**If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result.
10	**static <T> Optional<T> of(T value)**Returns an Optional with the specified present non-null value.
11	**static <T> Optional<T> ofNullable(T value)**Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
12	**T orElse(T other)**Returns the value if present, otherwise returns other.
13	**T orElseGet(Supplier<? extends T> other)**Returns the value if present, otherwise invokes other and returns the result of that invocation.
14	**<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)** Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier.
15	**String toString()**Returns a non-empty string representation of this Optional suitable for debugging.

2.4 Java 8 Optional<T> Sample

```
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed parameter to be null.
      Optional<Integer> a = Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }
   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      //Optional.isPresent - checks the value is present or not

      System.out.println("First parameter is present: " + a.isPresent());
      System.out.println("Second parameter is present: " + b.isPresent());

      //Optional.orElse - returns the value if present otherwise return
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

Compiling the source code....
$javac Java8Tester.java 2>&1

Executing the program....
$java -Xmx128M -Xms16M Java8Tester

```
First parameter is present: false
Second parameter is present: true
10
```

2.4 Java 8 Optional<T> Another Sample

```
@Override
public boolean equals(Object o) {
    return Optional.ofNullable(o)
            .filter(that -> that instanceof Task)
            .map(that -> (Task) that)
            .filter(that -> Objects.equals(this.title, that.title))
            .filter(that -> Objects.equals(this.type, that.type))
            .isPresent();
}
```

2.4 Java 8 New Date/Time API

What does the below program prints?

```
public class DatePain {

    public static void main(String[] args) {
        Date date = new Date(12, 12, 12);
        System.out.println(date);
    }
}
```

Sun Jan 12 00:00:00 IST 1913

- Which 12 is for date field?
- 12 is for December right?? No. It's January
- Year 12 is 12 CE?? Wrong 1913.. starts from 1900
- Hmmm. why there is time in date??
- There is time-zone as well?? Who asked??
- **Existing Date API**
  - Date API was introduced in JDK in 1996. **There are many issues with:**
    - Mutability
    - Date is not date but date with time
    - Separate Date class hierarchy for SQL
    - No concept of time-zone
    - Boilerplate friendly

Existing Calendar API

- Still mutable
- Can't format a date directly
- You can't perform arithmetic operations on date.
- Calendar instance does not work with formatter

2.4 Java 8 New Date/Time API

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API –

- **Not thread safe** – java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
- **Poor design** – Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
- **Difficult time zone handling** – Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package **java.time**. Following are some of the important classes introduced in java.time package –

- **Local** – Simplified date-time API with no complexity of time-zone handling.
- **Zoned** – Specialized date-time API to deal with various time-zones.

2.4 Java 8 New Date/Time API

New API — Getting Started

- Developed as part of JSR 310
- Heavily inspired by Joda-Time library
- New package — java.time

New types for humans

- ***LocalDate*** - a date with no time or timezone
- ***LocalTime*** - a time with no date or timezone
- ***LocalDateTime*** - LocaDate + LocalTime

All types are immutable

New Type >> Instant

- A machine friendly way to describe date and time
- Number of seconds passed since epoch time
- Nanosecond precision
- Useful to represent event timestamp

```
public static void main(String[] args) {
    Instant instant = Instant.ofEpochSecond(3);
    System.out.println(instant);//1970-01-01T00:00:03Z

    Instant now = Instant.now();
    System.out.println(now);//2015-09-19T17:18:18.425Z

    System.out.println(Instant.parse("2015-09-12T10:15:30.00Z"));
}
```

2.4 Java 8 New Date/Time API

Duration and Period

- Duration represents quantity or amount of time in seconds or nano-seconds like 10 seconds
  - Duration d = Duration.between(dt1, dt2);
- Period represents amount or quantity of time in years, months, and days
  - Period p = Period.between(ld1, ld2);

Temporal Adjuster

When you need to do advance date-time manipulation then you will use it.

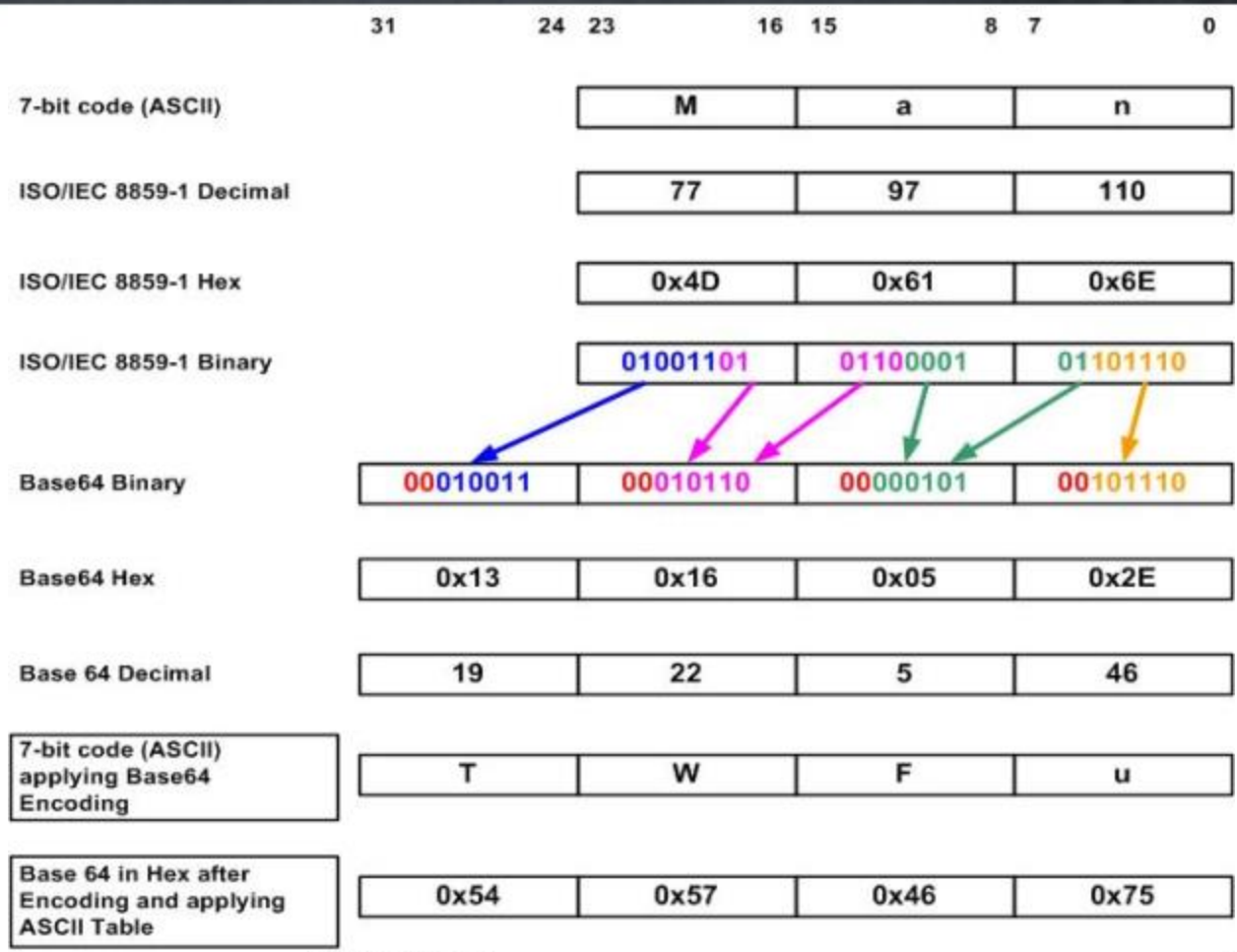
Examples

- adjust to next Sunday
- adjust to first day of next month
- adjust to next working day

2.4 Java 8 Base64

Base64 encoding is used in practice usually for transport over the network and heterogeneous environments binary code such as pictures or executable code. The techniques is very simple: to transform each 3 bytes values into 4 bytes value in order to avoid to obtain values greater then 127 per byte.

For instance, if the scope is to encode the word “Man” into Base64 encoding then it is encoded as “TWFu”. Encoded in ASCII (in ISO 8859-1, one value per byte), M, a, n are stored as the bytes 77 (0x4D), 97 (0x61), 110 (0x6E), which are 01001101, 01100001, 01101110 in base 2.



As this example illustrates, the encoding converts 3 not encoded bytes (in this case, ASCII characters) into 4 encoded ASCII characters. If the output is not multiple of 4 bytes then the '=' sign is put in order to supply the padding. Must be considered the padding with = (64 value) in case of multiple 4 bytes number.

Base 64 Code

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51

0	1	2	3	4	5	6	7	8	9	+	/	=
52	53	54	55	56	57	58	59	60	61	62	63	64 (64 is used as padding)

2.4 Java 8 Base64

With Java 8, Base64 has finally got its due. Java 8 now has inbuilt encoder and decoder for Base64 encoding. In Java 8, we can use three types of Base64 encoding –

- **Simple** – Output is mapped to a set of characters lying in **A-Za-z0-9+/**. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
- **URL** – Output is mapped to set of characters lying in **A-Za-z0-9+_**. Output is URL and filename safe.
- **MIME** – Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters each, and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. No line separator is present to the end of the encoded output.

2.4 Java 8 Base64

Base64 - Nested Classes

S. No.	Nested class & Description
1	**static class Base64.Decoder** This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
2	**static class Base64.Encoder** This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Methods

S. No.	Method Name & Description
1	**static Base64.Decoder getDecoder()**Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
2	**static Base64.Encoder getEncoder()**Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.

2.4 Java 8 Base64

Methods

S. No.	Method Name & Description
3	**static Base64.Decoder getMimeDecoder()**Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
4	**static Base64.Encoder getMimeEncoder()** Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
5	**static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)**Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
6	**static Base64.Decoder getUrlDecoder()**Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
7	**static Base64.Encoder getUrlEncoder()**Returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

Example without Base64 from Java 8

From the discussion here, and especially this answer, this is the function I currently use (* *Also, look into the javax.xml.bind.DatatypeConverter.parseBase64Binary() and printHexBinary()*):

```
final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();
public static String bytesToHex(byte[] bytes) {
 char[] hexChars = new char[bytes.length * 2];
 for ( int j = 0; j < bytes.length; j++ ) {
   int v = bytes[j] & 0xFF;
   hexChars[j * 2] = hexArray[v >>> 4];
   hexChars[j * 2 + 1] = hexArray[v & 0x0F];
 }
 return new String(hexChars);
}
```

"My own tiny benchmarks (a million bytes a thousand times, 256 bytes 10 million times) showed it to be much faster than any other alternative, about half the time on long arrays. Compared to the answer I took it from, switching to bitwise ops --- as suggested in the discussion --- cut about 20% off of the time for long arrays. (Edit: Performance is equivalent to Commons Codec, which uses very similar code.)"

2.4 Java 8 Base64

Example with Base64 from Java 8

```
public static void main(String args[]){
   try {

      // Encode using basic encoder
      String base64encodedString = Base64.getEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));
      System.out.println("Base64 Encoded String (Basic) :" + base64encodedString);

      // Decode
      byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);

      System.out.println("Original String: " + new String(base64decodedBytes, "utf-8"));
      base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));
      System.out.println("Base64 Encoded String (URL) :" + base64encodedString);

      StringBuilder stringBuilder = new StringBuilder();

      for (int i = 0; i < 10; ++i) {
         stringBuilder.append(UUID.randomUUID().toString());
      }

      byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");
      String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes);
      System.out.println("Base64 Encoded String (MIME) :" + mimeEncodedString);

   }catch(UnsupportedEncodingException e){
      System.out.println("Error :" + e.getMessage());
   }
```

Section Conclusions

Method references, Default Methods, Lambda and Functional Interfaces, Streams for processing

Why Java 8 code is better?

- **Developer's intent is clear**
- **Declarative >> What over How**
- **Reusable chainable higher level construct**
- **Unified language for data processing**
- **No boilerplate code.!! Yay :)**

Java 8 Functional Programming
for easy sharing

Share knowledge, Empowering Minds

Communicate & Exchange Ideas

Questions & Answers!

But wait…

There's More!

Thanks!

Java SE Programming
End of Lecture 7 – summary of Java SE