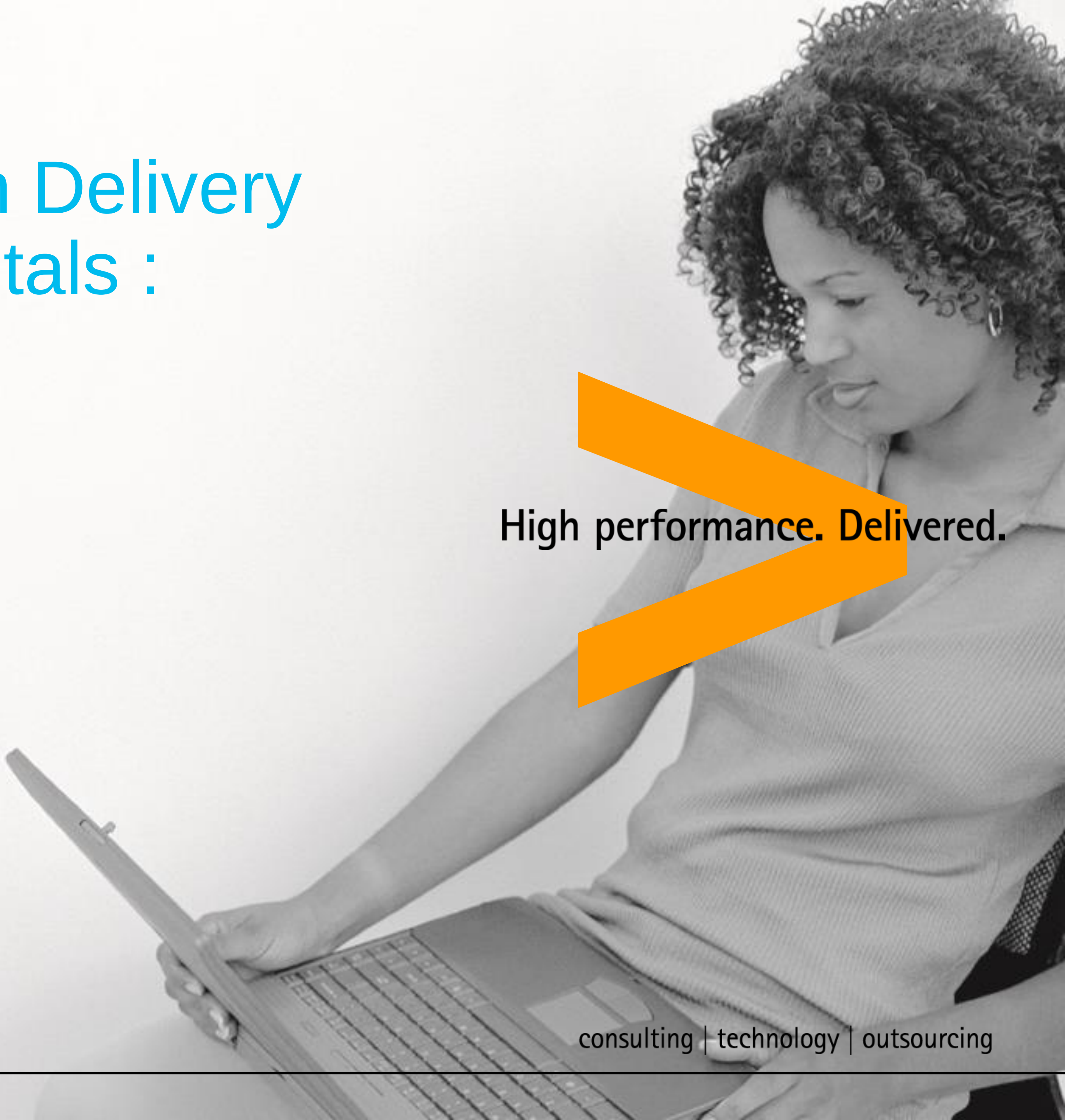
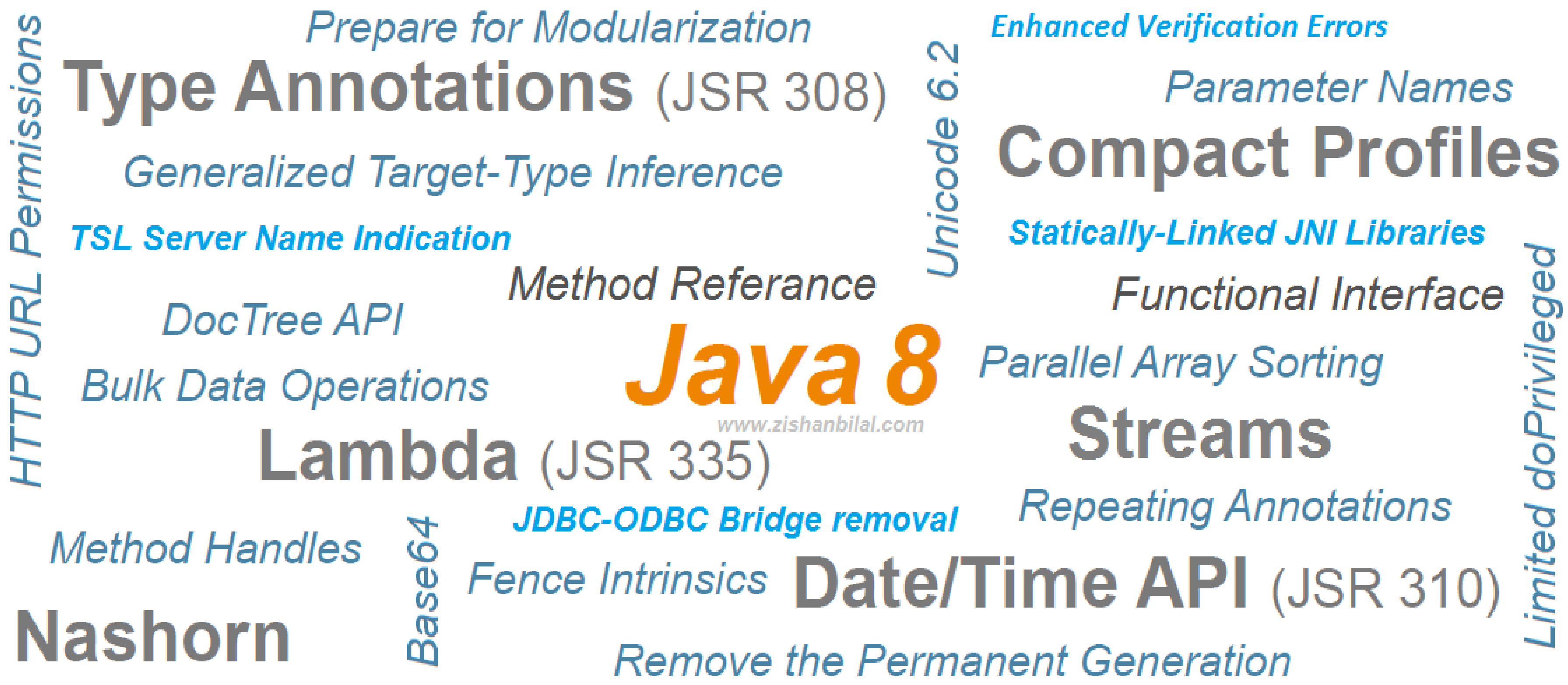
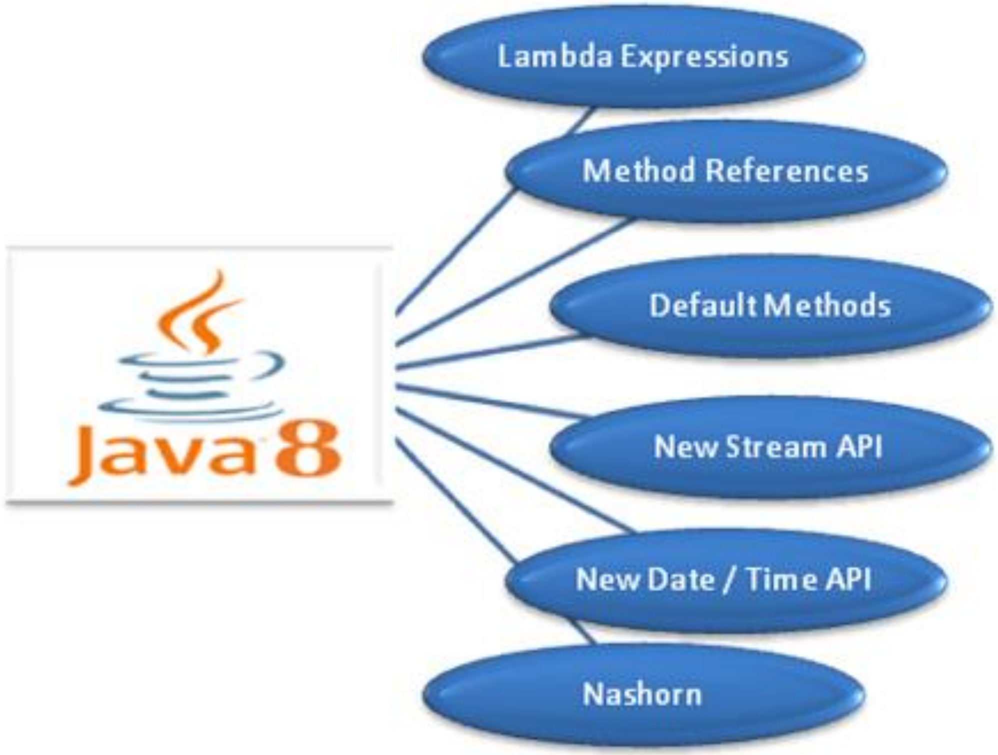
Application Delivery Fundamentals : Java

Java 8


High performance. Delivered.
consulting | technology | outsourcing

Java 8


HTTP URL Permissions
Prepare for Modularization
Type Annotations (JSR 308)
Generalized Target-Type Inference
TSL Server Name Indication
DocTree API
Bulk Data Operations
Method Referance
Java 8
www.zishanbilal.com
Lambda (JSR 335)
Method Handles
Nashorn
Base64
JDBC-ODBC Bridge removal
Fence Intrinsics
Date/Time API (JSR 310)
Remove the Permanent Generation
Unicode 6.2
Enhanced Verification Errors
Parameter Names
Compact Profiles
Statically-Linked JNI Libraries
Functional Interface
Parallel Array Sorting
Streams
Repeating Annotations
Limited doPrivileged


Java 8
Lambda Expressions
Method References
Default Methods
New Stream API
New Date / Time API
Nashorn

Java 8 Goals

- Lambda expressions
- Method references
- Functional interfaces
- Accumulators(Double Acc, Double Adder Long Acc, Long Adder)
- Stream API
- Default methods
- Base64 Encode Decode
- Static methods in interface,
- Optional class

Java 8 Goals

- Collectors class,
- ForEach() method,
- Parallel array sorting,
- Nashorn JavaScript Engine,
- Parallel Array Sorting,
- Type and Repating Annotations,
- IO Enhancements,
- Concurrency Enhancements,
- JDBC Enhancements etc.

Java 8 Goals

- Java 8 Security Enhancements
- Javadoc Enhancements (Doc Tree API)

Java Lambda Expressions

- Lambda expression is a new and important feature provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface.
- It saves a lot of code.
- Java lambda expression is treated as a function.

Java Lambda Expressions

- Lambda expressions is to treat functionality as method argument, or code as data.
- Java 8 Lambda Expressions can be defined as methods without names i.e anonymous functions.
- Like methods, they can have parameters, a body, a return type and possible list of exceptions that can be thrown.
- But unlike methods, neither they have names nor they are associated with any particular class.

Lambda Syntax

(Parameters) -> Expression

OR

(Parameters) -> { Statements }

- Lambda syntax consist of three parts – list of parameters, an arrow mark and a body.
- The body of a lambda can be an expression or a set of statements.
- If it is set of statements, they must be enclosed within curly braces { }.
- Return type and possible list of exceptions that can be thrown are not explicitly mentioned in a lambda.
- They are implicitly applied.

Where To Use Lambda Expressions?

- Lambda expressions are used where an instance of functional interface is expected.
- Functional interface is an interface which has only one abstract method.
- Functional interfaces can have any number of default methods.
- But, they must have only one abstract method. Comparator, Runnable And ActionListener are some examples of functional interfaces.

Lambda Expressions

Lambda expression vs method in Java

- A method (or function) in Java has these main parts:
- 1. Name
- 2. Parameter list
- 3. Body
- 4. return type.

Lambda Expressions

A lambda expression in Java has these main parts:

- Lambda expression only has body and parameter list.
- 1. No name – function is anonymous so we don’t care about the name
- 2. Parameter list
- 3. Body – This is the main part of the function.
- 4. No return type – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly.

Functional Interface

- Lambda expression provides implementation of *functional interface*.
- An interface which has only one abstract method is called functional interface.
- Java provides an annotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Functional Interface

- Single Abstract Method Type

Functional Interfaces

Example

```
@FunctionalInterface
public interface Runnable {
    public void run();
}
Runnable r = () -> System.out.println(“CITI Team!");
```

Java Lambda Expression Syntax

- (argument-list) -> {body}
- Java lambda expression is consisted of three components.
- **1) Argument-list:** It can be empty or non-empty as well.
- **2) Arrow-token:** It is used to link arguments-list and body of expression.
- **3) Body:** It contains expressions and statements for lambda expression.

Java Lambda Expression Syntax

- Lambdas bring anonymous function types in Java (JSR 335):

```
(parameters) -> {body}
```

- Example:

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

Java Lambda Expression Syntax

- Examples of such functional interfaces:

java.lang.Runnable -> run()

java.util.concurrent.Callable -> call()

java.security.PrivilegedAction -> run()

java.util.Comparator -> compare(T o1, T o2)

java.awt.event.ActionListener ->

    actionPerformed (ActionEvent e)

java.lang.Iterable ->

    forEach(Consumer<? super T> action)

Java Lambda Expression Syntax

- Examples of such functional interfaces:

java.lang.Runnable -> run()

java.util.concurrent.Callable -> call()

java.security.PrivilegedAction -> run()

java.util.Comparator -> compare(T o1, T o2)

java.awt.event.ActionListener ->

actionPerformed (ActionEvent e)

java.lang.Iterable ->

forEach(Consumer<? super T> action)

Java Lambda Expression Syntax

```
@FunctionalInterface
public interface Comparator
{
    int compare(T o1, T o2);       //Only one abstract method
}

@FunctionalInterface
public interface Runnable
{
    public abstract void run();   //Only one abstract method
}

@FunctionalInterface
public interface ActionListener extends EventListener
{
    public void actionPerformed(ActionEvent e);  //Only One abstract method
}
```

Java Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Java Lambda Expression Example: Single Parameter

```
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
s.o.p (s1.say(“casper”);
 }
}
```

Java Lambda Expression

- Java Lambda Expression Example: Multiple Parameters
- Java Lambda Expression Example: with or without return keyword
- Java Lambda Expression Example: For each Loop
- Java Lambda Expression Example: Creating Thread
- Java Lambda Expression Example: Comparator

```
Collections.sort(list,(p1,p2)->{
        return p1.name.compareTo(p2.name);
        });
```

How To Use Lambda Expressions?

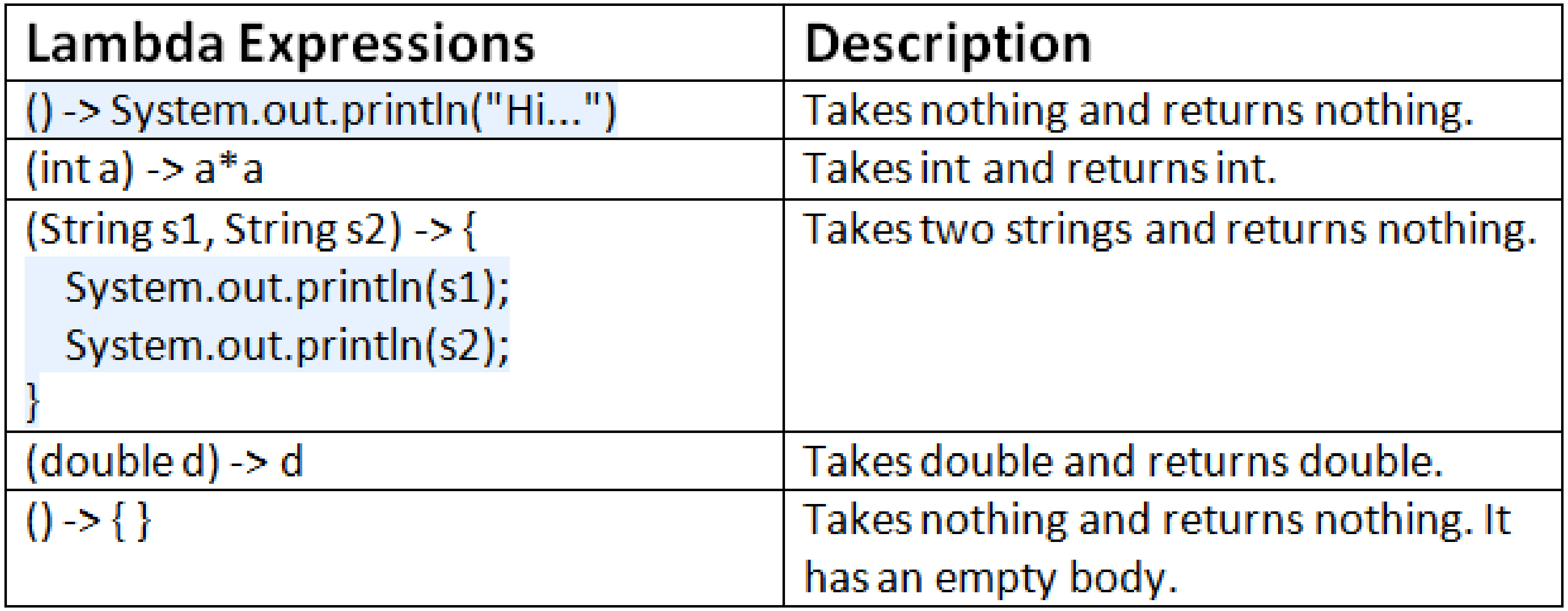
- Lambda expressions are used to implement functional interfaces.
- Before Java 8, anonymous inner classes are used to implement functional interfaces.
- Before Java 8 : Implementation of Comparator interface using anonymous inner class

```
Comparator<Student> idComparator = new Comparator<Student>() {
            @Override
            public int compare(Student s1, Student s2) {
                return s1.getID()-s2.getID();
            }
        };
```

How To Use Lambda Expressions?

- After Java 8 : Implementation of Comaparator interface using lambda expressions
- Comparator<Student> idComparator = (Student s1, Student s2) -> s1.getID()-s2.getID();

Valid Lambda Expressions With Description

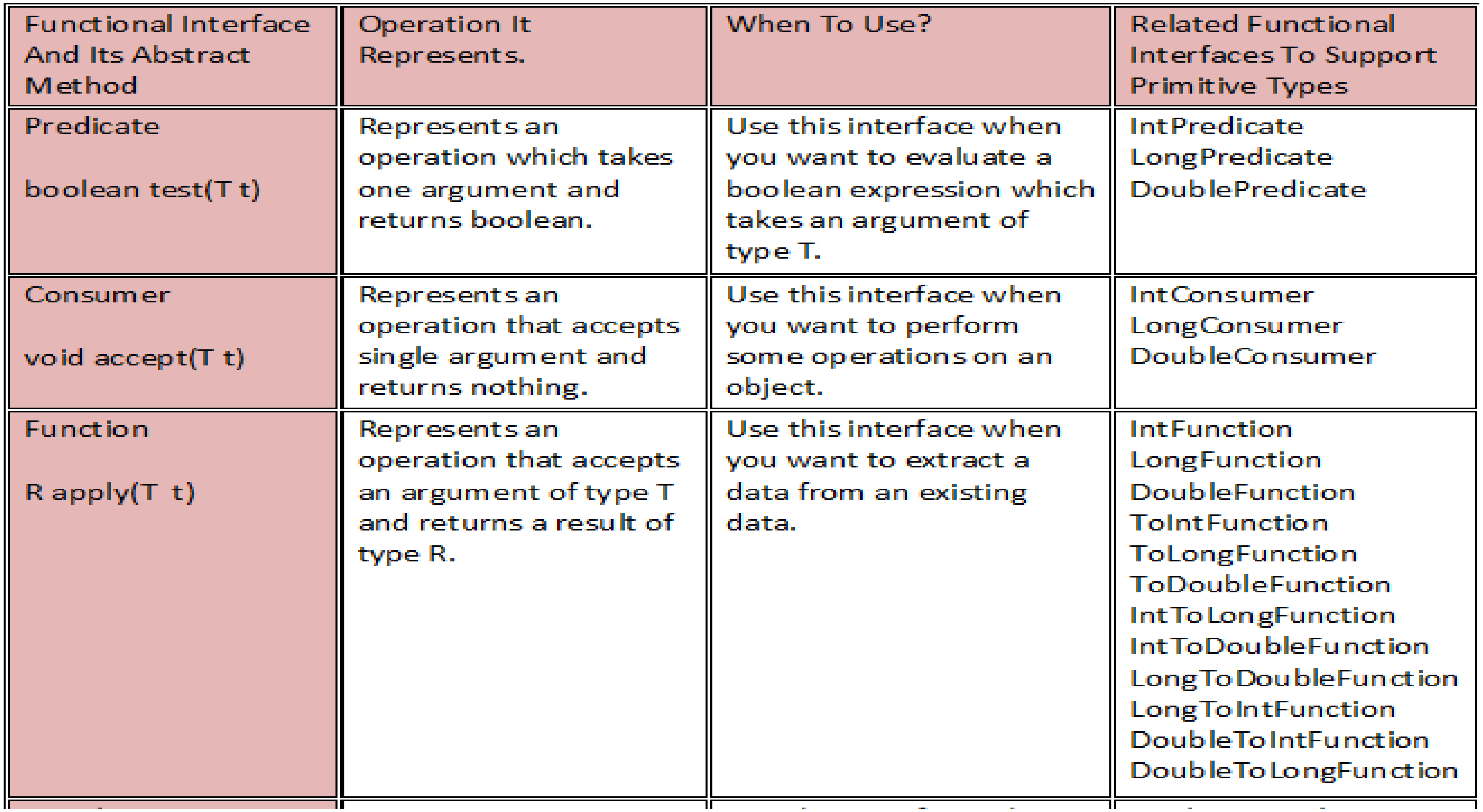


Lambda Expressions	Description
() -> System.out.println("Hi...")	Takes nothing and returns nothing.
(int a) -> a*a	Takes int and returns int.

(String s1, String s2) -> {
   System.out.println(s1);
   System.out.println(s2);
}
	Takes two strings and returns nothing.

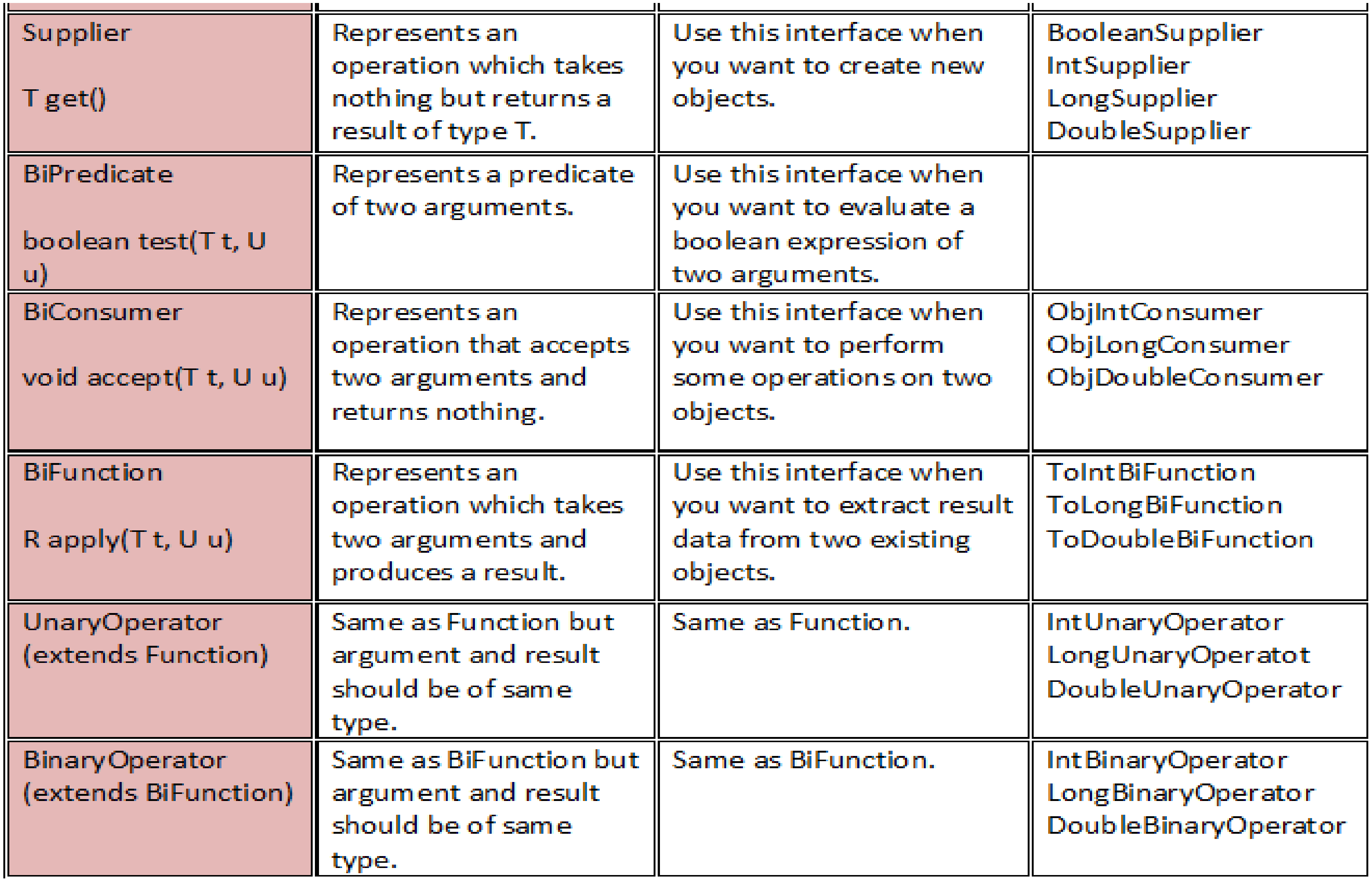
(double d) -> d	Takes double and returns double.
() -> { }	Takes nothing and returns nothing. It has an empty body.

Java 8 Functional Interfaces java.util.function package



Functional Interface And Its Abstract Method	Operation It Represents.	When To Use?	Related Functional Interfaces To Support Primitive Types
Predicate boolean test(T t)	Represents an operation which takes one argument and returns boolean.	Use this interface when you want to evaluate a boolean expression which takes an argument of type T.	IntPredicate LongPredicate DoublePredicate
Consumer void accept(T t)	Represents an operation that accepts single argument and returns nothing.	Use this interface when you want to perform some operations on an object.	IntConsumer LongConsumer DoubleConsumer
Function R apply(T t)	Represents an operation that accepts an argument of type T and returns a result of type R.	Use this interface when you want to extract a data from an existing data.	IntFunction LongFunction DoubleFunction ToIntFunction ToLongFunction ToDoubleFunction IntToLongFunction IntToDoubleFunction LongToDoubleFunction LongToIntFunction DoubleToIntFunction DoubleToLongFunction

Java 8 Functional Interfaces java.util.function package



Supplier T get()	Represents an operation which takes nothing but returns a result of type T.	Use this interface when you want to create new objects.	BooleanSupplier IntSupplier LongSupplier DoubleSupplier
BiPredicate boolean test(T t, U u)	Represents a predicate of two arguments.	Use this interface when you want to evaluate a boolean expression of two arguments.	
BiConsumer void accept(T t, U u)	Represents an operation that accepts two arguments and returns nothing.	Use this interface when you want to perform some operations on two objects.	ObjIntConsumer ObjLongConsumer ObjDoubleConsumer
BiFunction R apply(T t, U u)	Represents an operation which takes two arguments and produces a result.	Use this interface when you want to extract result data from two existing objects.	ToIntBiFunction ToLongBiFunction ToDoubleBiFunction
UnaryOperator (extends Function)	Same as Function but argument and result should be of same type.	Same as Function.	IntUnaryOperator LongUnaryOperatot DoubleUnaryOperator
BinaryOperator (extends BiFunction)	Same as BiFunction but argument and result should be of same type.	Same as BiFunction.	IntBinaryOperator LongBinaryOperator DoubleBinaryOperator

Built-in Functions

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single double-valued operand that produces a double-valued result.

Built-in Functions

Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two int-valued operands and producing an int-valued result.
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.
IntPredicate	Represents a predicate (boolean-valued function) of one int-valued argument.
IntSupplier	Represents a supplier of int-valued results.
IntToDoubleFunction	Represents a function that accepts an int-valued argument and produces a double-valued result.
IntToLongFunction	Represents a function that accepts an int-valued argument and produces a long-valued result.
IntUnaryOperator	Represents an operation on a single int-valued operand that produces an int-valued result.
LongBinaryOperator	Represents an operation upon two long-valued operands and producing a long-valued result.
LongConsumer	Represents an operation that accepts a single long-valued argument and returns no result.
LongFunction<R>	Represents a function that accepts a long-valued argument and produces a result.
LongPredicate	Represents a predicate (boolean-valued function) of one long-valued argument.
LongSupplier	Represents a supplier of long-valued results.

Built-in Functions

LongToDoubleFunction	Represents a function that accepts a long-valued argument and produces a double-valued result.
LongToIntFunction	Represents a function that accepts a long-valued argument and produces an int-valued result.
LongUnaryOperator	Represents an operation on a single long-valued operand that produces a long-valued result.
ObjDoubleConsumer<T>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
ObjIntConsumer<T>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.
ObjLongConsumer<T>	Represents an operation that accepts an object-valued and a long-valued argument, and returns no result.
Predicate<T>	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	Represents a supplier of results.
ToDoubleBiFunction<T, U>	Represents a function that accepts two arguments and produces a double-valued result.
ToDoubleFunction<T>	Represents a function that produces a double-valued result.
ToIntBiFunction<T,U>	Represents a function that accepts two arguments and produces an int-valued result.
ToIntFunction<T>	Represents a function that produces an int-valued result.
ToLongBiFunction<T,U>	Represents a function that accepts two arguments and produces a long-valued result.
ToLongFunction<T>	Represents a function that produces a long-valued result.
UnaryOperator<T>	Represents an operation on a single operand that produces a result of the same type as its operand.

Java Method References

- Method reference is used to refer method of functional interface.
- It is compact and easy form of lambda expression.
- Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.
- Java 8 method references can be defined as shortened versions of lambda expressions calling a specific method.
- Method references are the easiest way to refer a method than the lambdas calling a specific method.
- Method references will enhance the readability of your code.

Types of Method References

- If your lambda expression is like this:
  - **str -> System.out.println(str)**
- then you can replace it with a method reference like this:
  - **System.out::println**

Types of Method References

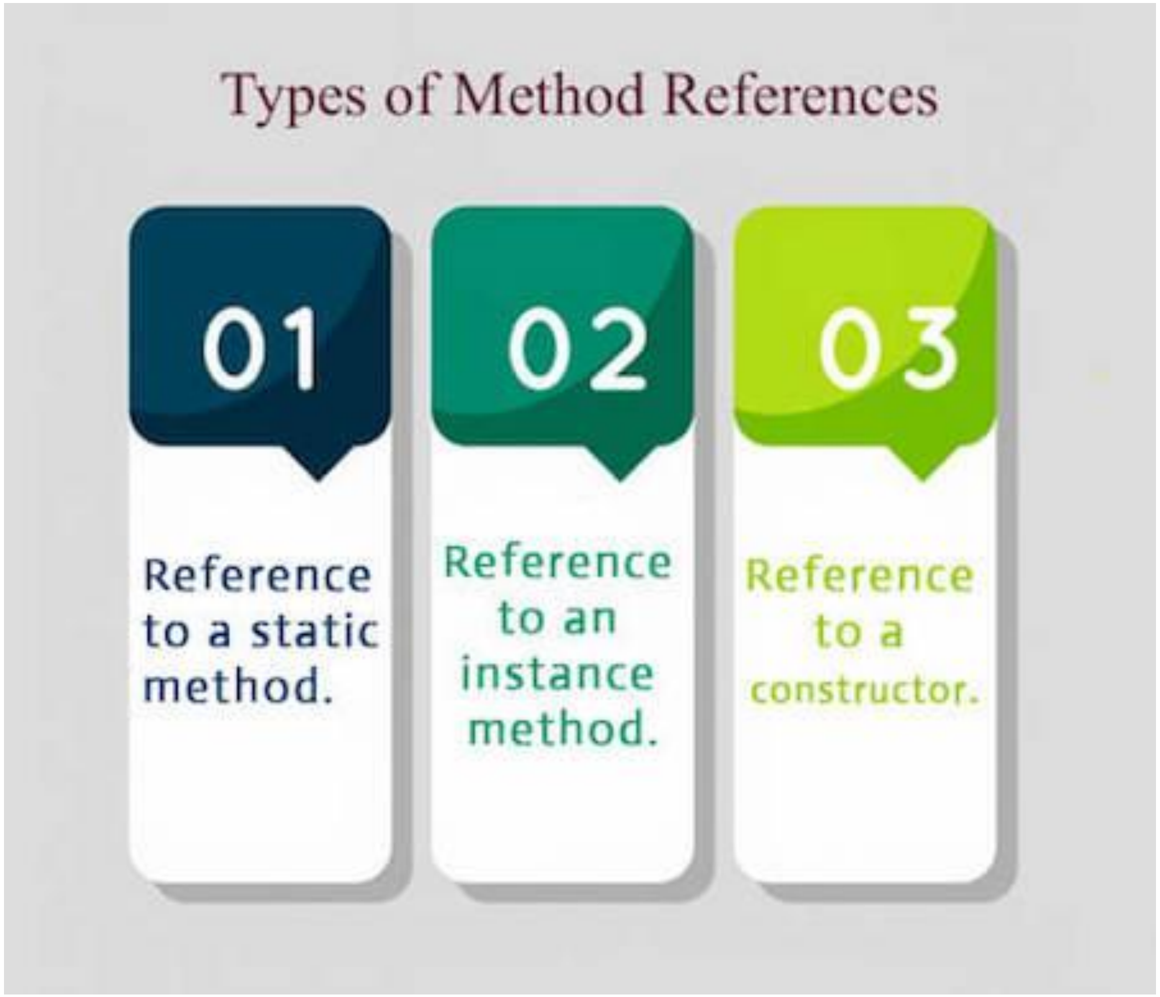
- Four types of method references
  1. Method reference to an instance method of an object – object::instanceMethod

  2. Method reference to a static method of a class – Class::staticMethod

  3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod

  4. Method reference to a constructor – Class::new

Types of Method References


Types of Method References

01
Reference to a static method.

02
Reference to an instance method.

03
Reference to a constructor.

Static Reference

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

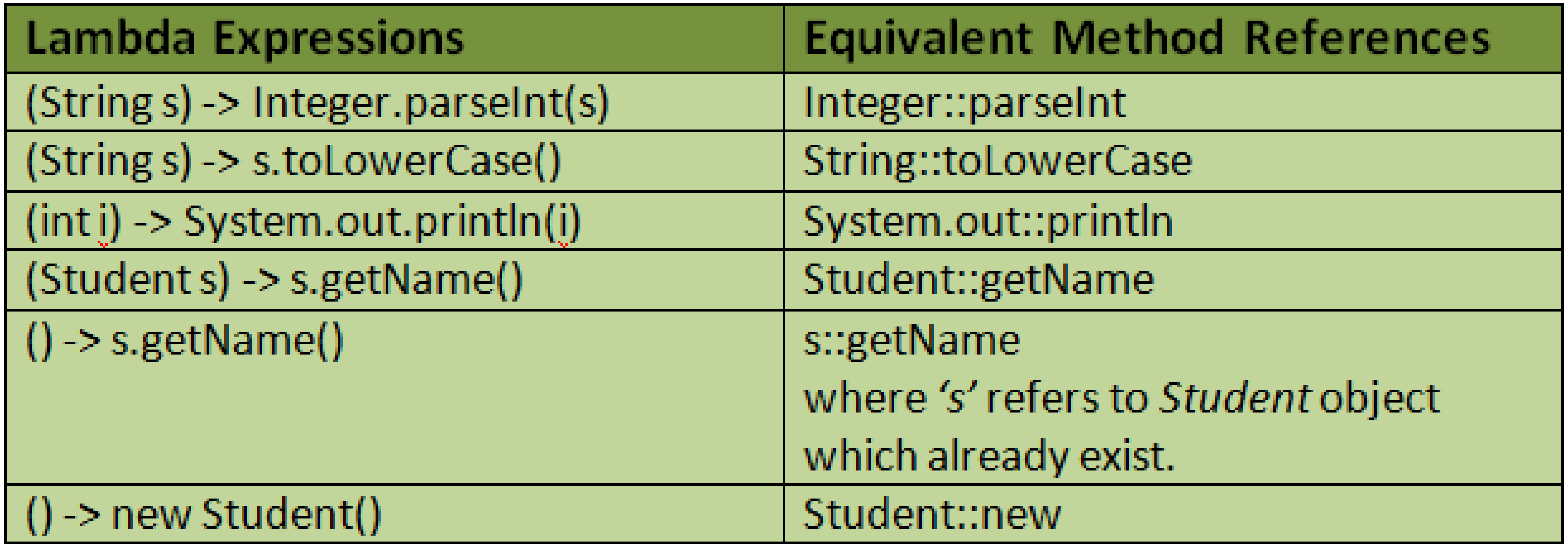
Instance Method Reference

```
interface Sayable{
    void say();  }
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
 // Creating  object
        InstanceMethodReference methodReference = new InstanceMethodReference();
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();          }  }
```

Reference to a Constructor

```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

Method References



Lambda Expressions	Equivalent Method References
(String s) -> Integer.parseInt(s)	Integer::parseInt
(String s) -> s.toLowerCase()	String::toLowerCase
(int i) -> System.out.println(i)	System.out::println
(Student s) -> s.getName()	Student::getName
() -> s.getName()	s::getName where *'s'* refers to *Student* object which already exist.
() -> new Student()	Student::new

Atomic Integer

- The package java.concurrent.atomic contains many useful classes to perform atomic operations.
- Internally, the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs.
- Those instructions usually are much faster than synchronizing via locks.
- So my advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.

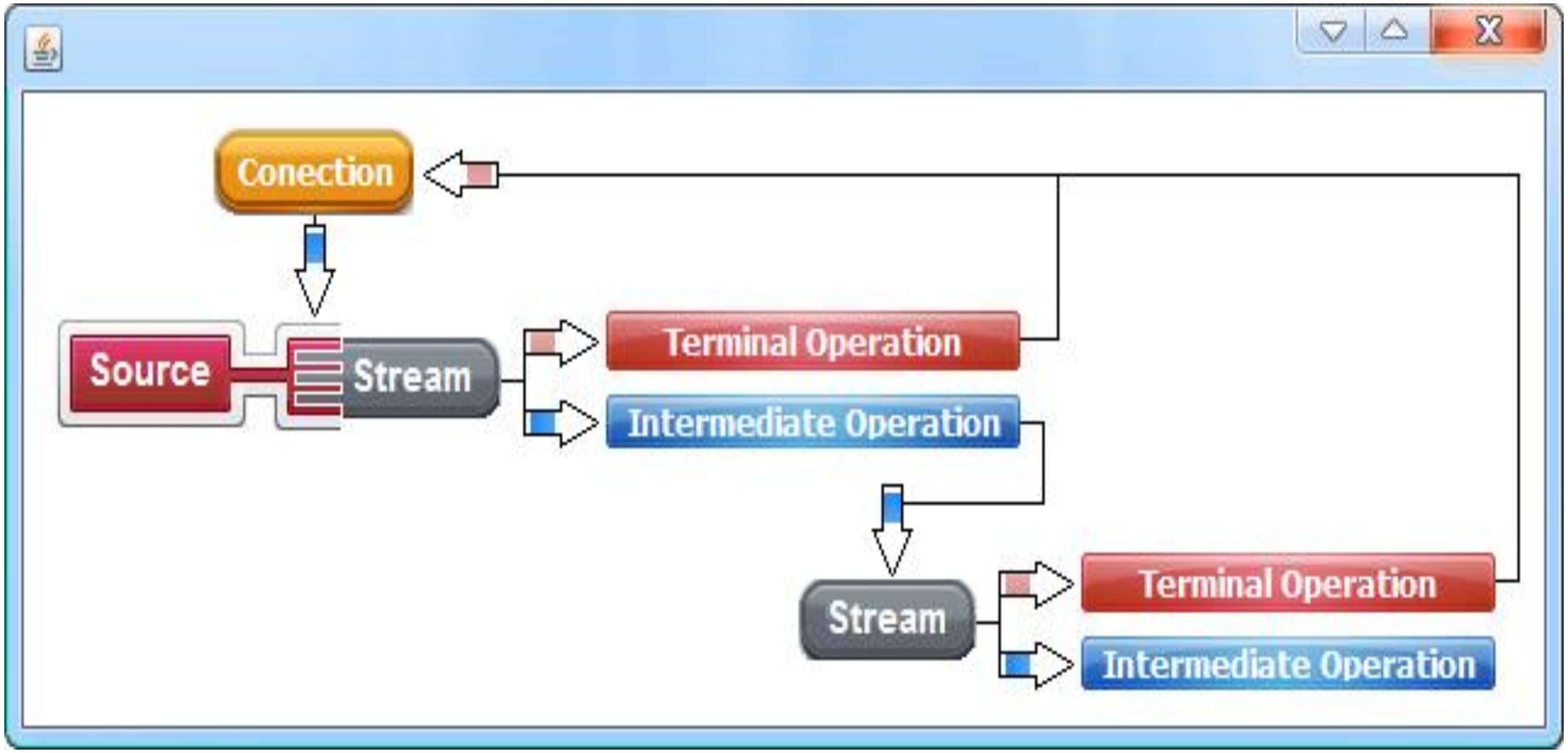
Java 8 Stream

- Java provides a new additional package in Java 8 called java.util.stream.
- This package consists of classes, interfaces and enum to allows functional-style operations on the elements.
- You can use stream by importing java.util.stream package.

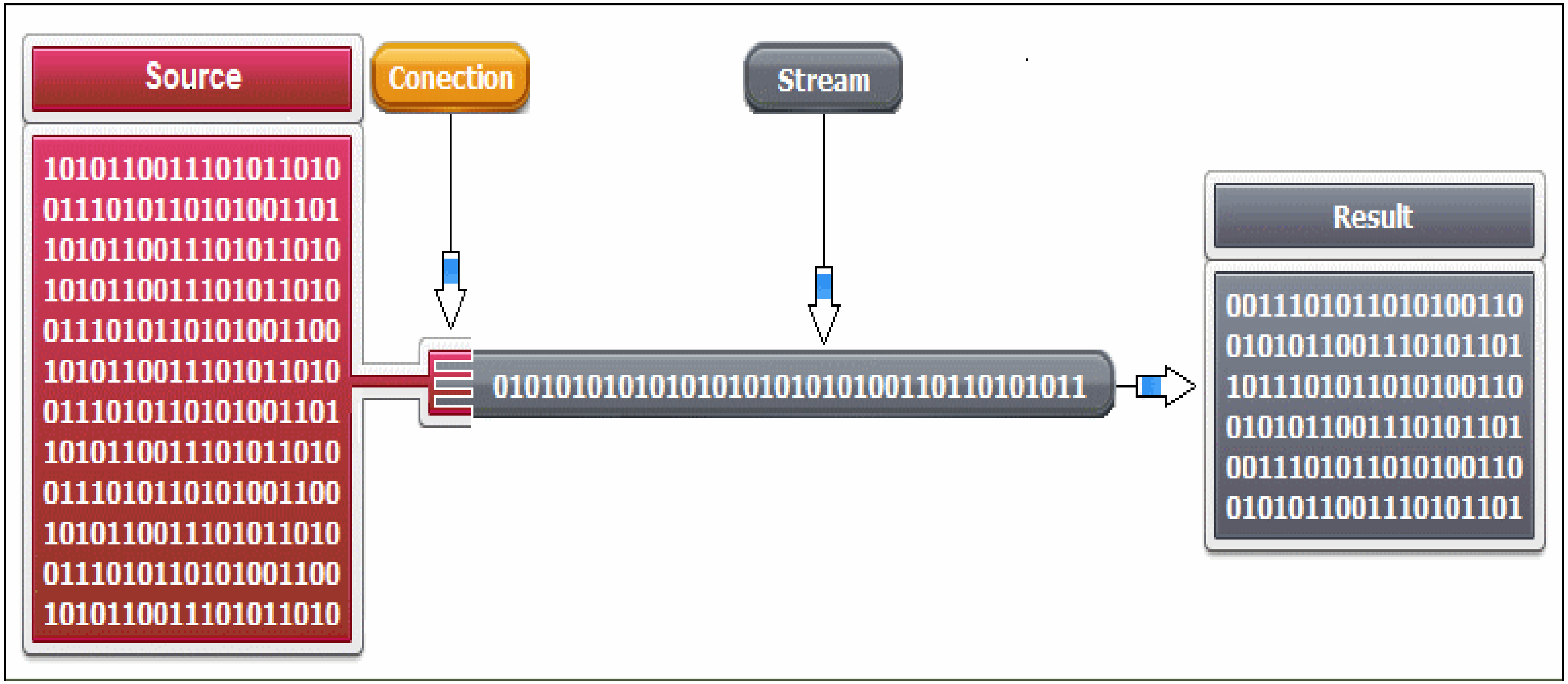
Java 8 Stream

- Stream provides following features:
- Stream does not store elements.
- It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature.
- Operations performed on a stream does not modify it's source.
- For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream.
- Like an Iterator, a new stream must be generated to revisit the same elements of the source.
- Stream can be used to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

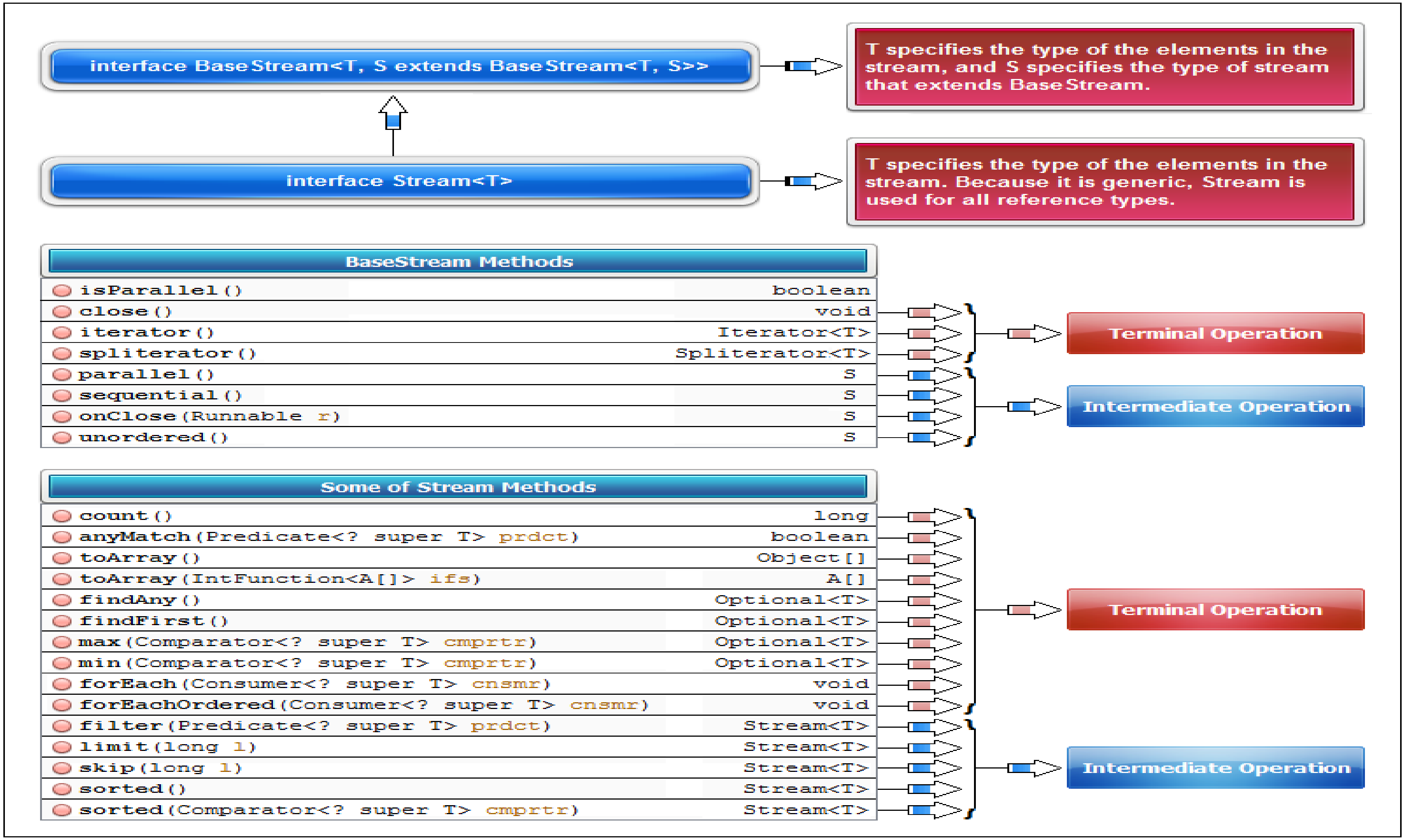
Java 8 Stream



Java 8 Stream



Java 8 Stream



interface BaseStream<T, S extends BaseStream<T, S>>

T specifies the type of the elements in the stream, and S specifies the type of stream that extends BaseStream.

interface Stream<T>

T specifies the type of the elements in the stream. Because it is generic, Stream is used for all reference types.

BaseStream Methods	
isParallel()	boolean
close()	void
iterator()	Iterator<T>
spliterator()	Spliterator<T>
parallel()	S
sequential()	S
onClose(Runnable r)	S
unordered()	S

close(), iterator(), spliterator(): Terminal Operation

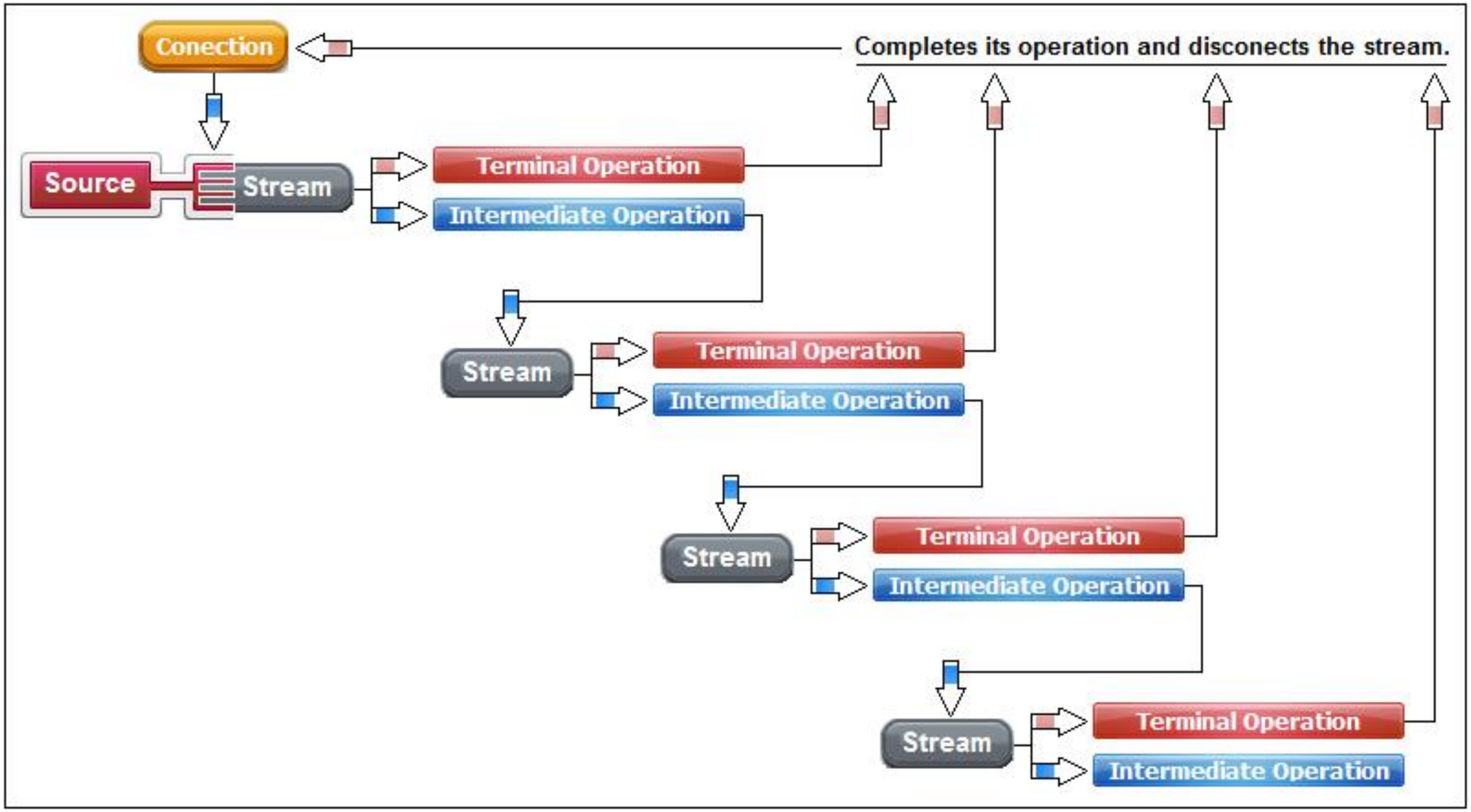
parallel(), sequential(), onClose(Runnable r), unordered(): Intermediate Operation

Some of Stream Methods	
count()	long
anyMatch(Predicate<? super T> prdct)	boolean
toArray()	Object[]
toArray(IntFunction<A[]> ifs)	A[]
findAny()	Optional<T>
findFirst()	Optional<T>
max(Comparator<? super T> cmprtr)	Optional<T>
min(Comparator<? super T> cmprtr)	Optional<T>
forEach(Consumer<? super T> cnsmr)	void
forEachOrdered(Consumer<? super T> cnsmr)	void
filter(Predicate<? super T> prdct)	Stream<T>
limit(long l)	Stream<T>
skip(long l)	Stream<T>
sorted()	Stream<T>
sorted(Comparator<? super T> cmprtr)	Stream<T>

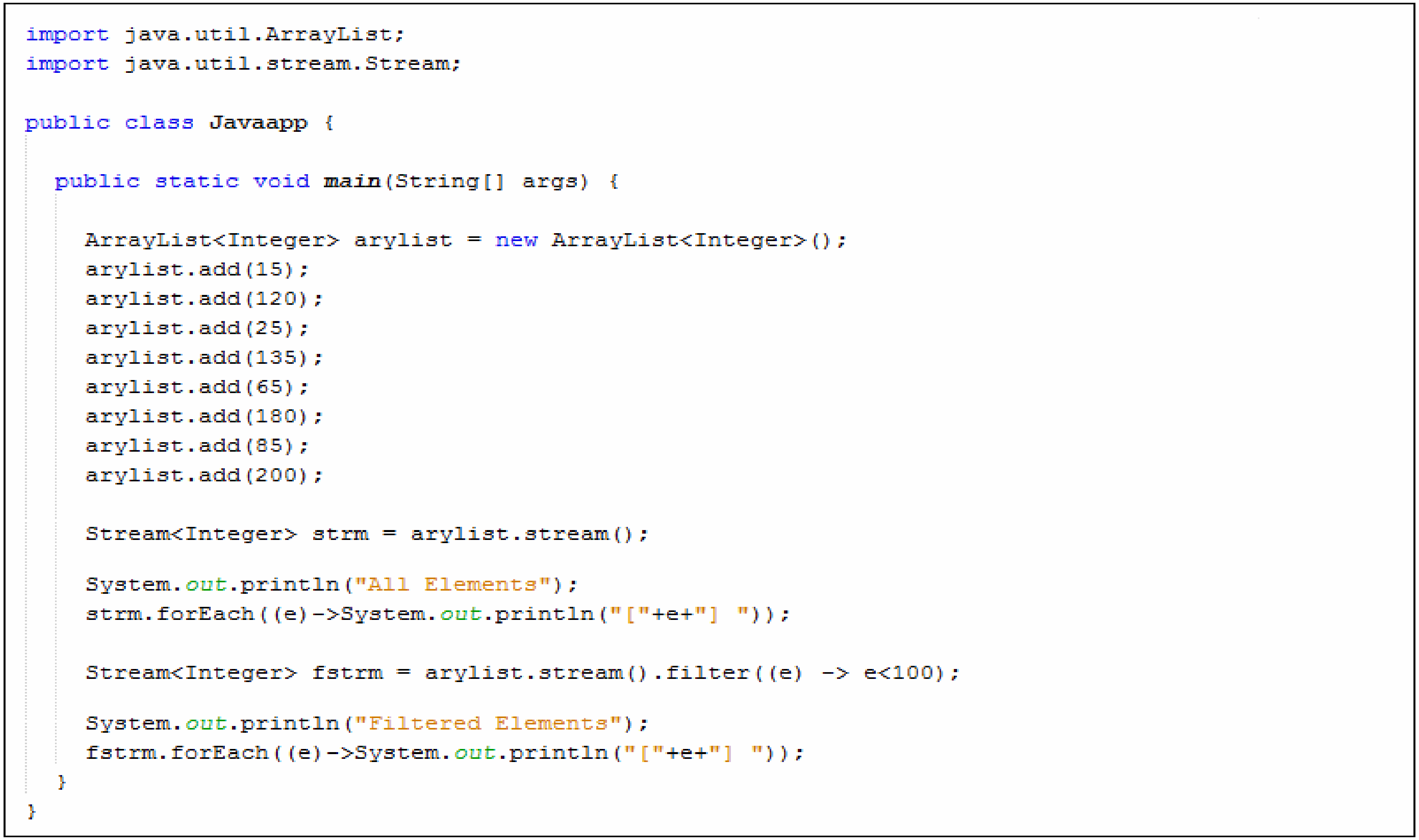
count() … forEachOrdered(Consumer<? super T> cnsmr): Terminal Operation

filter(Predicate<? super T> prdct) … sorted(Comparator<? super T> cmprtr): Intermediate Operation

Java 8 Stream



Java 8 Stream Animation



```
import java.util.ArrayList;
import java.util.stream.Stream;

public class Javaapp {

  public static void main(String[] args) {

    ArrayList<Integer> arylist = new ArrayList<Integer>();
    arylist.add(15);
    arylist.add(120);
    arylist.add(25);
    arylist.add(135);
    arylist.add(65);
    arylist.add(180);
    arylist.add(85);
    arylist.add(200);

    Stream<Integer> strm = arylist.stream();

    System.out.println("All Elements");
    strm.forEach((e)->System.out.println("["+e+"] "));

    Stream<Integer> fstrm = arylist.stream().filter((e) -> e<100);

    System.out.println("Filtered Elements");
    fstrm.forEach((e)->System.out.println("["+e+"] "));
  }
}
```

Java Stream

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
boolean anyMatch(Predicate<? super T> predicate)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
static <T> Stream.Builder<T> builder()	It returns a builder for a Stream.
<R,A> R collect(Collector<? super T,A,R> collector)	It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.

Java Stream

static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)	It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T> findAny()	It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T> findFirst()	It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.

Java Stream

<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)	It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

Java Stream

DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)	It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)	It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
void forEach(Consumer<? super T> action)	It performs an action for each element of this stream.
void forEachOrdered(Consumer<? super T> action)	It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T> generate(Supplier<T> s)	It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

Java Stream

static <T> Stream<T> iterate(T seed,UnaryOperator<T> f)	It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T> limit(long maxSize)	It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R> map(Function<? super T,? extends R> mapper)	It returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)	It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream mapToInt(ToIntFunction<? super T> mapper)	It returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream mapToLong(ToLongFunction<? super T> mapper)	It returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T> max(Comparator<? super T> comparator)	It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.
Optional<T> min(Comparator<? super T> comparator)	It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.

Java Stream

boolean noneMatch(Predicate<? super T> predicate)	It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
@SafeVarargs static <T> Stream<T> of(T... values)	It returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T> of(T t)	It returns a sequential Stream containing a single element.
Stream<T> peek(Consumer<? super T> action)	It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T> reduce(BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T reduce(T identity, BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T> skip(long n)	It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.

Java Stream

Stream<T> sorted()	It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
Stream<T> sorted(Comparator<? super T> comparator)	It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[] toArray()	It returns an array containing the elements of this stream.
<A> A[] toArray(IntFunction<A[]> generator)	It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Java 8 Stream

```
List<Float> productPriceList2 =productsList.stream()
                            .filter(p -> p.price > 30000)// filtering data
                            .map(p->p.price)      // fetching price
                            .collect(Collectors.toList()); // collecting as list


Stream.iterate(1, element->element+1)
      .filter(element->element%5==0)
      .limit(5)
      .forEach(System.out::println);
   }

 productsList.stream()
                       .filter(product -> product.price == 30000)
                       .forEach(product -> System.out.println(product.name));
```

Java 8 Stream

```
// This is more compact approach for filtering data
        Float totalPrice = productsList.stream()
                    .map(product->product.price)
                    .reduce(0.0f,(sum, price)->sum+price);   // accumulating price
        System.out.println(totalPrice);
        // More precise code
        float totalPrice2 = productsList.stream()
                .map(product->product.price)
                .reduce(0.0f,Float::sum);   // accumulating price, by referring method of Float class
```

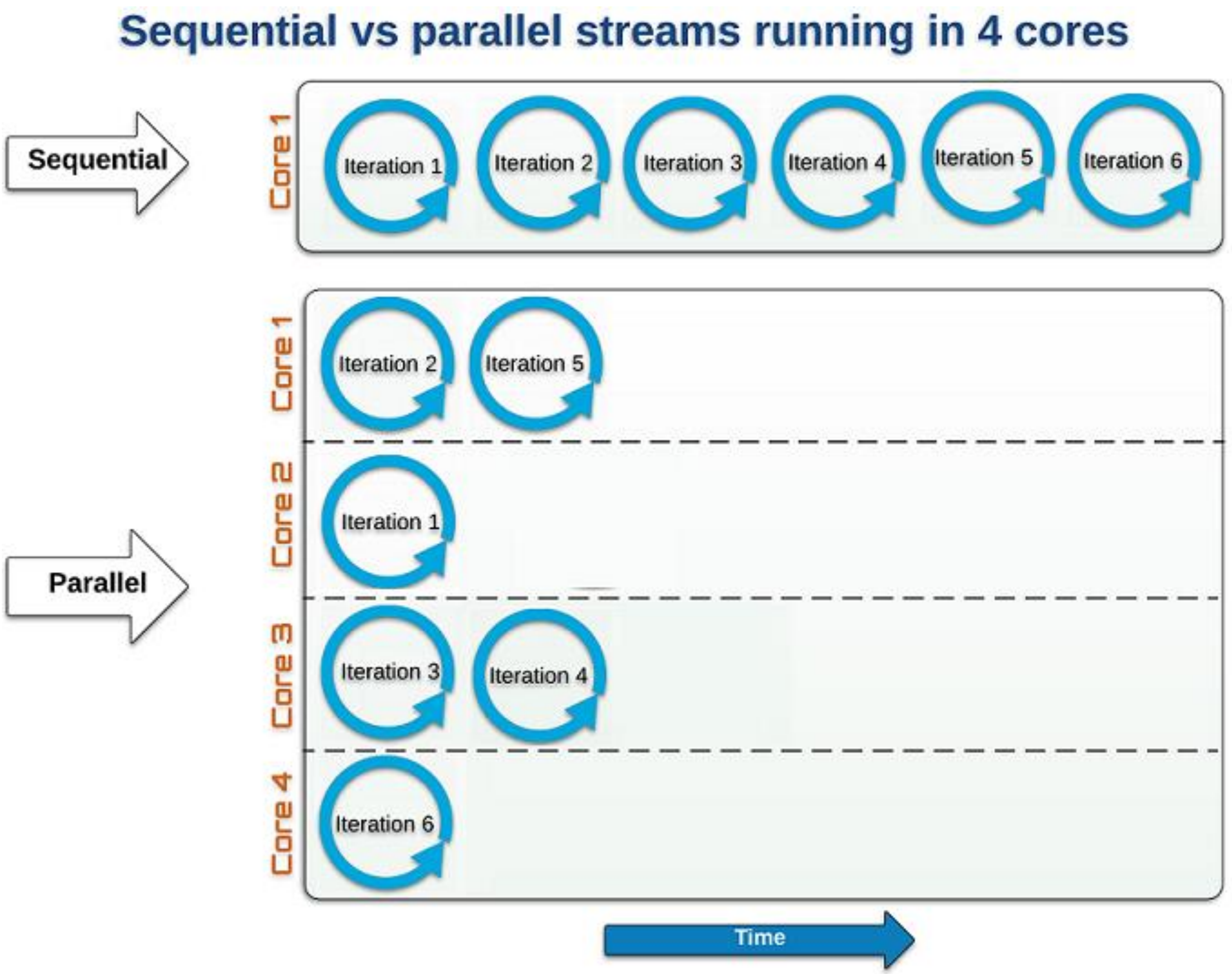
Java 8 Stream

```
// // max() method to get max Product price
        Product productA = productsList.stream()
                        .max((product1, product2)->
                        product1.price > product2.price ? 1: -1).get();

        System.out.println(productA.price);
        // min() method to get min Product price
        Product productB = productsList.stream()
                .max((product1, product2)->
                product1.price < product2.price ? 1: -1).get();
```

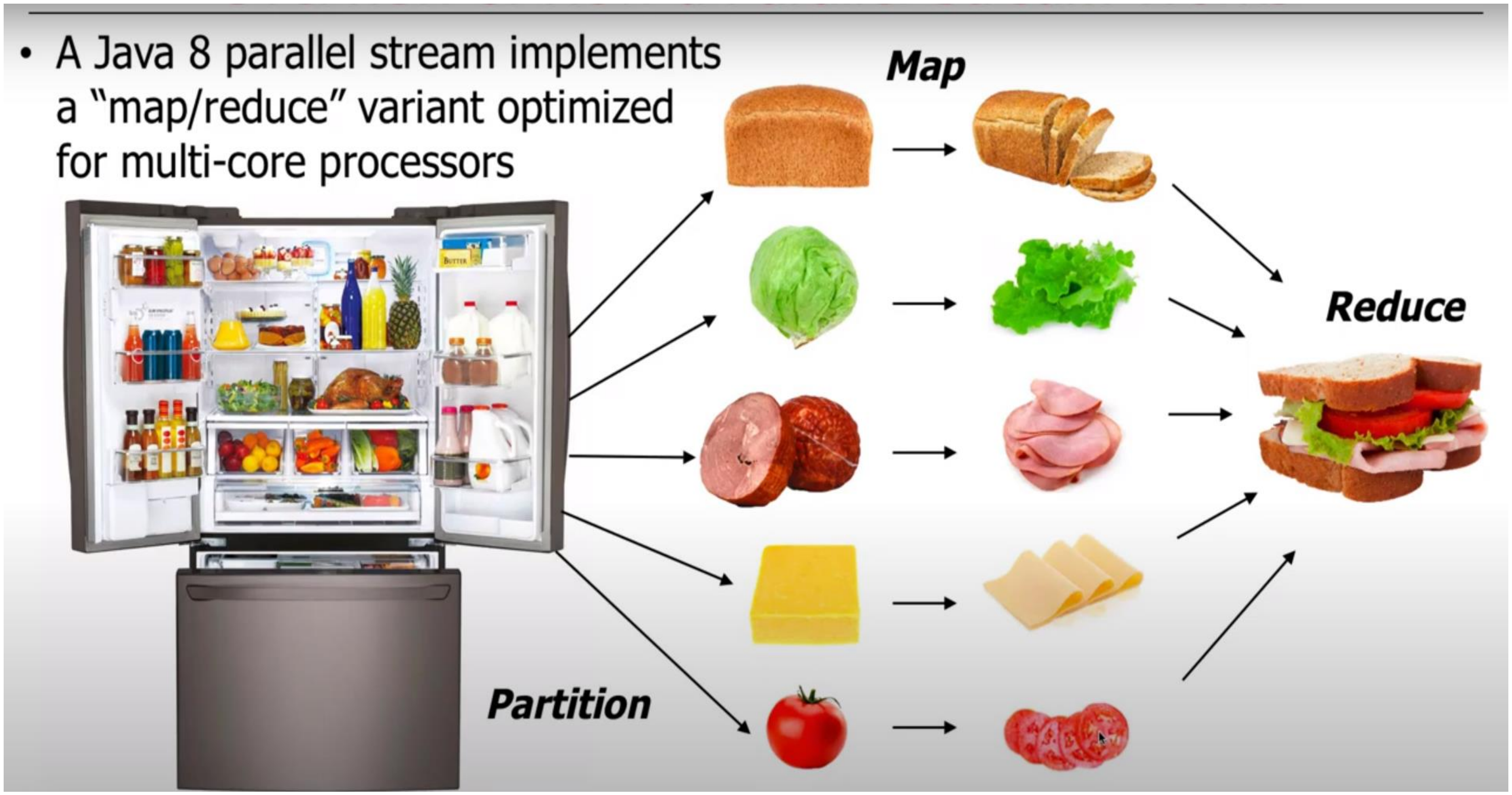
Java 8 Stream

```
// count number of products based on the filter
        long count = productsList.stream()
                    .filter(product->product.price<30000)
                    .count();
// Converting Product List into a Map
        Map<Integer,String> productPriceMap =
            productsList.stream()
                        .collect(Collectors.toMap(p->p.id, p->p.name));

               List<Float> productPriceList =
                productsList.stream()
                            .filter(p -> p.price > 30000) // filtering data
                            .map(Product::getPrice)         // fetching price by referring getPrice method
                            .collect(Collectors.toList());  // collecting as list
```

Streams


Sequential vs parallel streams running in 4 cores
Sequential
Core 1
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Parallel
Core 1
Iteration 2
Iteration 5
Core 2
Iteration 1
Core 3
Iteration 3
Iteration 4
Core 4
Iteration 6
Time

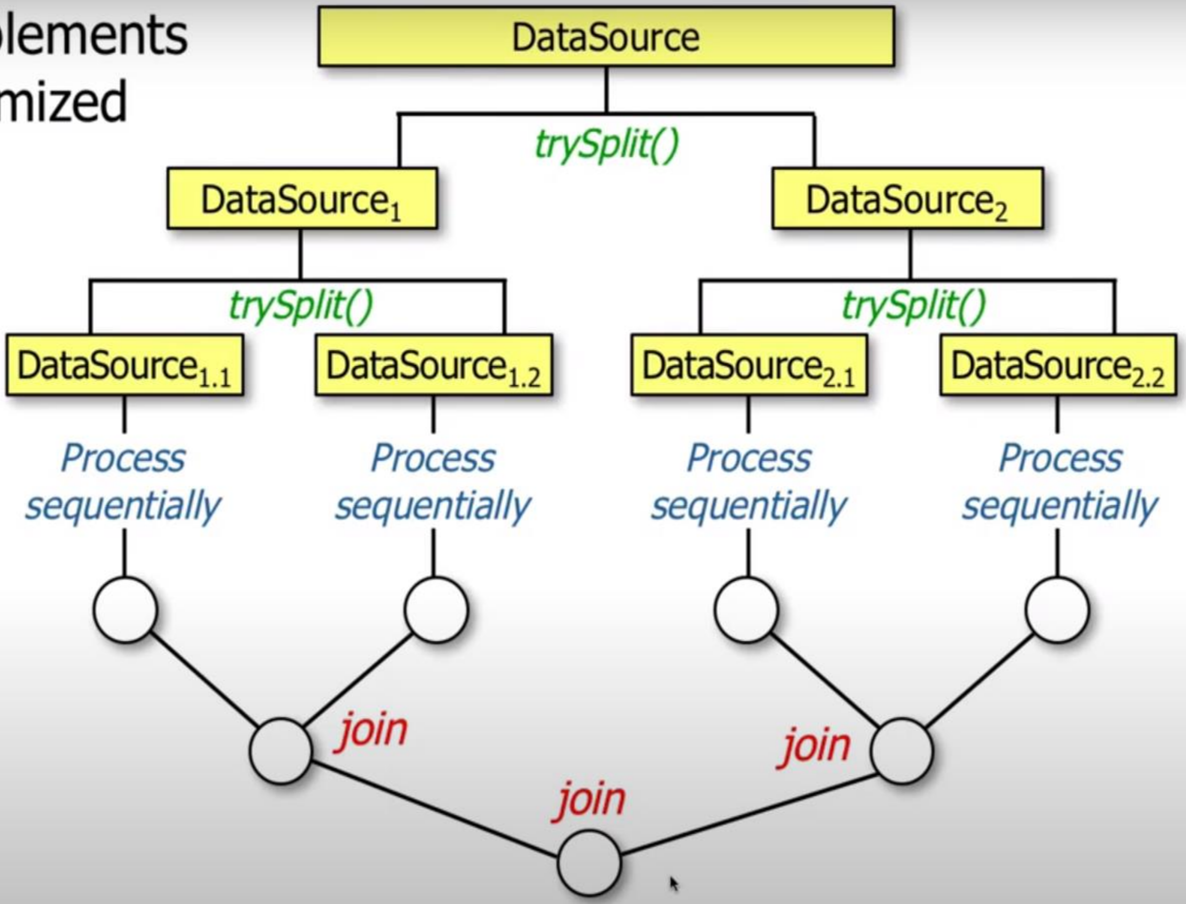
Parallel Streams



- A Java 8 parallel stream implements a "map/reduce" variant optimized for multi-core processors

Parallel Streams

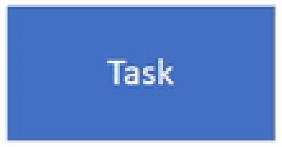
- A Java 8 parallel stream implements a "map/reduce" variant optimized for multi-core processors
  - It's actually more like the "split-apply-combine" data analysis strategy



Fork Join Pool

- The fork/join framework was designed to speed up the execution of tasks that can be divided into other smaller subtasks, executing them in parallel and then combining their results to get a single one.
- For this reason, the subtasks must be independent of each other and the operations must be stateless, making this framework not be the best solution for all problems.
- Applying a divide and conquer principle, the framework recursively divides the task into smaller subtasks until a given threshold is reached. This is the fork part.
- Then, the subtasks are processed independently and if they return a result, all the results are recursively combined into a single result. This is the join part.

Fork Join Pool



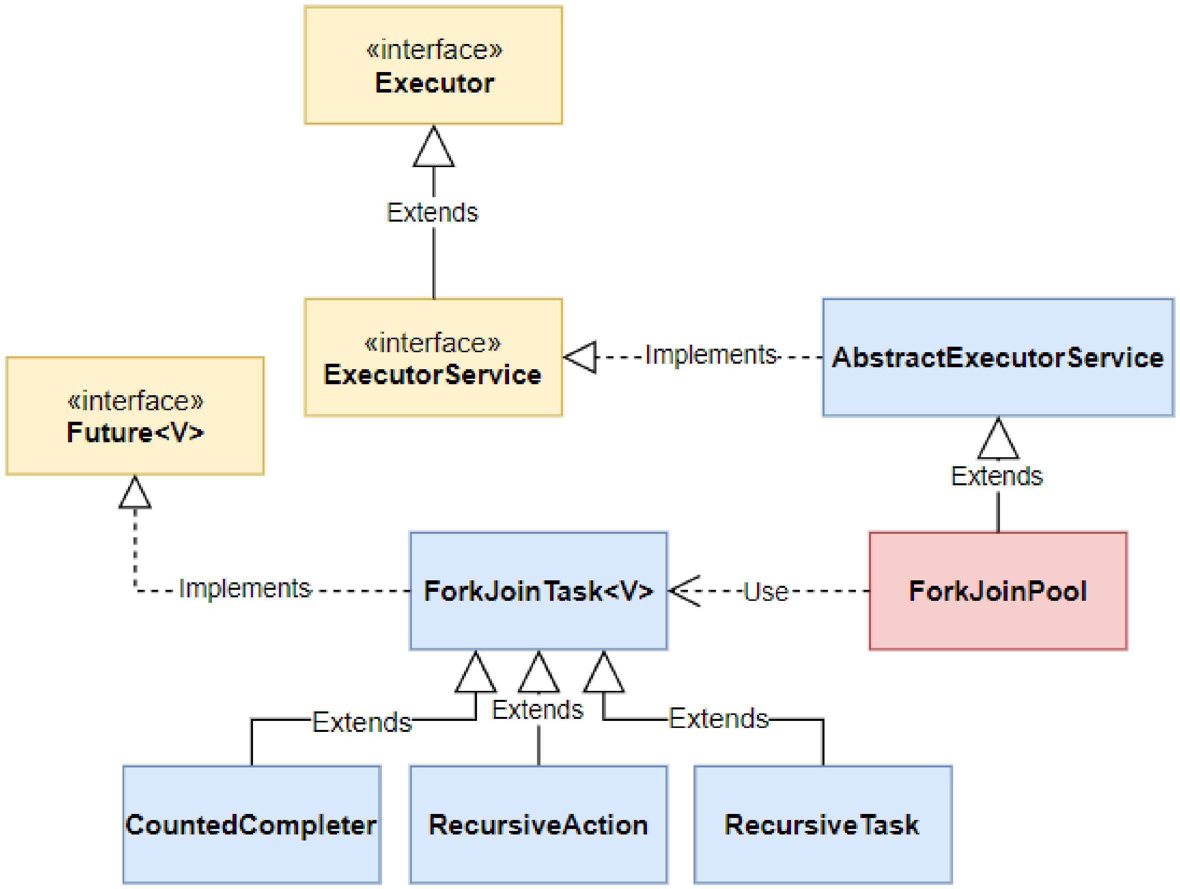
Fork Join Pool

- To execution the tasks in parallel, the framework uses a pool of threads, with a number of threads equal to the number of processors available to the Java Virtual Machine (JVM) by default.
- Each thread has its own double-ended queue (deque) to store the tasks that will execute.
- A deque is a type of queue that supports adding or removing elements from either the front (head) or the back (tail). This allows two things:
  - A thread can execute only one task at a time (the task at the head of its deque).
  - A work-stealing algorithm s implemented to balance the thread’s workload.

Fork Join Pool

- With the work-stealing algorithm, threads that run out of tasks to process can steal tasks from other threads that are still busy (by removing tasks from the tail of their deque).
- This approach makes processing more efficient by increasing throughput when there are many tasks to process or when one task diverges into many subtasks.

Fork Join Pool



What streams are good for

- They allow functional programming style using bindings.
- They allow for better performance by removing iteration. Iteration occurs with evaluation. With streams, we can bind dozens of functions **without iterating**.
- They allow easy parallelization for task including long waits.
- Streams may be infinite (since they are lazy). Functions may be bound to infinite streams without problem. Upon evaluation, there must be some way to make them finite. This is often done through a short circuiting operation.

When to use Parallel Streams?

- They should be used when the output of the operation is not needed to be dependent on the order of elements present in source collection (i.e. on which the stream is created)
- Parallel Streams can be used in case of aggregate functions
- Parallel Streams quickly iterate over the large-sized collections
- Parallel Streams can be used if developers have performance implications with the Sequential Streams
- If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in

What streams are not good for

- Streams should be used with high caution when processing intensive computation tasks.
- In particular, by default, all parallel streams will use the same ForkJoinPool, configured to use as many threads as there are cores in the computer on which the program is running.
- If evaluation of one parallel stream results in a very long running task, this may be split into as many long running sub-tasks that will be distributed to each thread in the pool.
- From there, no other parallel stream can be processed because all threads will be occupied.

Case Study

- Parallel streams to increase the performance of a time-consuming save file tasks.
- This Java code will generate 10,000 random employees and save into 10,000 files, each employee save into a file.
- For normal stream, it takes 27-29 seconds.
- For parallel stream, it takes 7-8 seconds.
- P.S Tested with i7-7700, 16G RAM, WIndows 10

What streams are not good for

- A parallel stream has a much higher overhead compared to a sequential one.
- Coordinating the threads takes a significant amount of time. Use sequential streams by default and only consider parallel ones if you have a massive amount of items to process (or the processing of each item takes time and is parallelizable)
- parallelism also often exposes nondeterminism in the computation that is often hidden by sequential implementations;

What streams are not good for

- In reality, sometimes parallelism will speed up your computation, sometimes it will not, and sometimes it will even slow it down.
- It is best to develop first using sequential execution and then apply parallelism where (A) you know that there's actually benefit to increased performance and (B) that it will actually deliver increased performance.
- (A) is a business problem, not a technical one. If you are a performance expert, you'll usually be able to look at the code and determine (B), but the smart path is to measure.

Java Base64 Encode and Decode

- Java provides a class Base64 to deal with encryption.
- You can encrypt and decrypt your data by using provided methods.
- You need to import java.util.Base64 in your source file to use its methods.
- This class provides three different encoders and decoders to encrypt information at each level.

Java Base64 Encode and Decode

- Basic Encoding and Decoding
- It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations.
- The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

Java Base64 Encode and Decode

- URL and Filename Encoding and Decoding
- It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

Java Base64 Encode and Decode

Class	Description
Base64.Decoder	This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
Base64.Encoder	This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Java Base64 Encode and Decode

- Base64 Methods

Methods	Description
public static Base64.Decoder getDecoder()	It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
public static Base64.Encoder getEncoder()	It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
public static Base64.Decoder getUrlDecoder()	It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
public static Base64.Decoder getMimeDecoder()	It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
public static Base64.Encoder getMimeEncoder()	It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)	It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
public static Base64.Encoder getUrlEncoder()	It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

Java Base64 Encode and Decode

- Base64 Methods

Methods	Description
public static Base64.Decoder getDecoder()	It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
public static Base64.Encoder getEncoder()	It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
public static Base64.Decoder getUrlDecoder()	It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
public static Base64.Decoder getMimeDecoder()	It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
public static Base64.Encoder getMimeEncoder()	It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)	It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
public static Base64.Encoder getUrlEncoder()	It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

Java Default Methods

- Java provides a facility to create default methods inside the interface.
- Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered().

It is used to iterate elements in the order specified by the stream.

Signature:

```
void forEachOrdered(Consumer<? super T> action)
```

```
  gamesList.stream().forEachOrdered(System.out::println);
```

Java Optional Class

- java introduced a new class Optional in jdk8.
- It is a public final class and used to deal with Null PointerException in Java application.
- You must import java.util package to use this class.
- It provides methods which are used to check the presence of value for particular variable.

Java Optional Class

Methods	Description
public static <T> Optional<T> empty()	It returns an empty Optional object. No value is present for this Optional.
public static <T> Optional<T> of(T value)	It returns an Optional with the specified present non-null value.
public static <T> Optional<T> ofNullable(T value)	It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
public T get()	If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
public boolean isPresent()	It returns true if there is a value present, otherwise false.
public void ifPresent(Consumer<? super T> consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
public Optional<T> filter(Predicate<? super T> predicate)	If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.

Java Optional Class

public <U> Optional<U> map(Function<? super T,? extends U> mapper)	If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional.
public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper)	If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
public T orElse(T other)	It returns the value if present, otherwise returns other.
public T orElseGet(Supplier<? extends T> other)	It returns the value if present, otherwise invoke other and return the result of that invocation.
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable	It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier.
public boolean equals(Object obj)	•Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:It is also an Optional and; •Both instances have no value present or; •the present values are "equal to" each other via equals().

Java Optional Class

public int hashCode()	It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present.
public String toString()	It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.

Java Nashorn

- Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.
- You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

Java Nashorn

- Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.
- You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

Java Parallel Array Sorting

- Java provides a new additional feature in Array class which is used to sort array elements parallel.
- New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.
- The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

Java Parallel Array Sorting

Methods	Description
public static void parallelSort(byte[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(byte[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(char[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(char[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(double[] a)	It sorts the specified array into ascending numerical order.

Java Parallel Array Sorting

public static void parallelSort(double[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(float[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(float[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(int[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(int[] a,int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

Java Parallel Array Sorting

public static void parallelSort(long[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(long[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(short[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(short[] a,int fromIndex,int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static <T extends Comparable<? super T>> void parallelSort(T[] a)	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

Java Parallel Array Sorting

public static <T7gt; void parallelSort(T[] a,Comparator<? super T> cmp)	It sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array).
public static <T extends Comparable<? super T>> void parallelSort(T[] a,int fromIndex, int toIndex)	It sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in this range must implement the Comparable interface. Furthermore, all elements in this range must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).
public static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp)	It sorts the specified range of the specified array of objects according to the order induced by the specified comparator. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in the range must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the range).

Method Parameter Reflection

- Java provides a new feature in which you can get the names of formal parameters of any method or constructor.
- The java.lang.reflect package contains all the required classes like Method and Parameter to work with parameter reflection.

Method Parameter Reflection

Method	Description
public boolean equals(Object obj)	It compares this Method against the specified object. It returns true if the objects are the same. Two Methods are the same if they were declared by the same class and have the same name and formal parameter types and return type.
public AnnotatedType getAnnotatedReturnType()	It returns an AnnotatedType object that represents the use of a type to specify the return type of the method/constructor.
public <T extends Annotation> T getAnnotation(Class<T> annotationClass)	It returns this element's annotation for the specified type if such an annotation is present otherwise returns null. NullPointerException - if the given annotation class is null
public Annotation[] getDeclaredAnnotations()	It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0. The caller of this method is free to modify the returned array. it will have no effect on the arrays returned to other callers.

Method Parameter Reflection

public Class<?> getDeclaringClass()	It returns the Class object representing the class or interface that declares the executable represented by this object.
public Object getDefaultValue()	It returns the default value for the annotation member represented by this Method instance.
public Class<?>[] getExceptionTypes()	It returns an array of Class objects that represent the types of exceptions declared to be thrown by the underlying executable represented by this object.
public Type[] getGenericExceptionTypes()	It returns an array of Type objects that represent the exceptions declared to be thrown by this executable object. It returns an array of length 0 if the underlying executable declares no exceptions in its throws clause. It throws following exceptions: **GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if the underlying executable's throws clause refers to a non-existent type declaration. **MalformedParameterizedTypeException** - if the underlying executable's throws clause refers to a parameterized type that cannot be instantiated for any reason.

Method Parameter Reflection

public Type[] getGenericParameterTypes()	It returns an array of Type objects that represent the formal parameter types. It throws following exceptions:**GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if any of the parameter types of the underlying executable refers to a non-existent type declaration. **MalformedParameterizedTypeException** - if any of the underlying executable's parameter types refer to a parameterized type that cannot be instantiated for any reason.
public int getModifiers()	It returns the Java language modifiers for the executable represented by this object.
public String getName()	It returns the name of the method represented by this Method object as a String.
public Annotation[][] getParameterAnnotations()	It returns an array of arrays that represent the annotations on the formal and implicit parameters, in declaration order, of the executable represented by this object.
public int getParameterCount()	It returns the number of formal parameters for the executable represented by this object.
public Class<?>[] getParameterTypes()	It returns an array of Class objects that represent the formal parameter types. in declaration order, of the executable represented by this object. It returns an array of length 0 if the underlying executable takes no parameters.

Method Parameter Reflection

public Class<?> getReturnType()	It returns a Class object that represents the formal return type of the method represented by this Method object.
public TypeVariable<Method>[] getTypeParameters()	It returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order. It throws GenericSignatureFormatError, if the generic signature of this generic declaration does not conform to the format specified in The Java Virtual Machine Specification
public int hashCode()	It returns a hashcode for this Method. The hashcode is computed as the exclusive-or of the hashcodes for the underlying method's declaring class name and the method's name.

Method Parameter Reflection

public Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException	It invokes the underlying method represented by this Method object, on the specified object with the specified parameters. If the underlying method is static, the specified obj argument is ignored. It may be null. If the number of formal parameters required by the underlying method is 0, the supplied args array may be of length 0 or null. If the underlying method is an instance method, it is invoked using dynamic method lookup as documented in The Java Language Specification. If the underlying method is static, the class that declared the method is initialized if it has not already been initialized. If the method completes normally, the value it returns is returned to the caller of invoke.
public boolean isBridge()	It returns true if this method is a bridge method. otherwise returns false.

Method Parameter Reflection

public boolean isDefault()	It returns true if this method is a default method otherwise returns false. A default method is a public non-abstract instance method, that is, a non-static method with a body, declared in an interface type.
public boolean isSynthetic()	It returns true if this executable is a synthetic construct; returns false otherwise.
public boolean isVarArgs()	It returns true if this executable was declared to take a variable number of arguments; returns false otherwise.
public String toGenericString()	It returns a string describing this Method, including type parameters.
public String toString()	It returns a string.

Method Parameter Reflection

Methods	Description
public boolean equals(Object obj)	It compares based on the executable and the index.
public AnnotatedType getAnnotatedType()	It returns an AnnotatedType object that represents the use of a type to specify the type of the formal parameter represented by this Parameter.
public <T extends Annotation> T getAnnotation(Class<T> annotationClass)	It returns this element's annotation for the specified type if such an annotation is present, else null. It throws NullPointerException, if the given annotation class is null.
public Annotation[] getAnnotations()	It returns annotations that are present on this element. If there are no annotations present on this element, the return value is an array of length 0.
public <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass)	It returns annotations that are associated with this element. If there are no annotations associated with this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation. It throws NullPointerException, if the given annotation class is null.

Method Parameter Reflection

public <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass)	It returns this element's annotation for the specified type if such an annotation is directly present, else null. This method ignores inherited annotations. It throws NullPointerException, if the given annotation class is null.
public Annotation[] getDeclaredAnnotations()	It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0.
public <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass)	It returns this element's annotations for the specified type if such annotations are either directly present or indirectly present. This method ignores inherited annotations. If there are no specified annotations directly or indirectly present on this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getDeclaredAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation if one is present. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers. It throws NullPointerException, if the given annotation class is null

Method Parameter Reflection

public Executable getDeclaringExecutable()	It returns the Executable which declares this parameter.
public int getModifiers()	It returns the modifier flags for the parameter represented by this Parameter object.
public String getName()	It returns the name of the parameter. If the parameter's name is present, this method returns the name provided by the class file. Otherwise, this method synthesizes a name of the form argN, where N is the index of the parameter in the descriptor of the method which declares the parameter.
public Type getParameterizedType()	It returns a Type object that identifies the parameterized type for the parameter represented by this Parameter object.
public Class<?> getType()	It returns a Class object that identifies the declared type for the parameter represented by this Parameter object.
public int hashCode()mul int arg0 int arg1 add int arg0 int arg1	It returns a hash code based on the executable's hash code and the index.
public boolean isImplicit()	It returns true if this parameter is implicitly declared in source code. Otherwise, returns false.

Java Type and Repeating Annotations

- Java Type Annotations
- @NonNull String str;
- @NonNull List<String>
- List<@NonNull String> str
- Arrays<@NonNegative Integer> sort
- @Encrypted File file
- @Open Connection connection

New Date-Time API in Java 8

- New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :
- **Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn’t have setter methods.
- **Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

New Date-Time API in Java 8

- Java 8 under the package java.time introduced a new date-time API, most important classes among them are :
- **Local :** Simplified date-time API with no complexity of timezone handling.
- **Zoned :** Specialized date-time API to deal with various timezones.

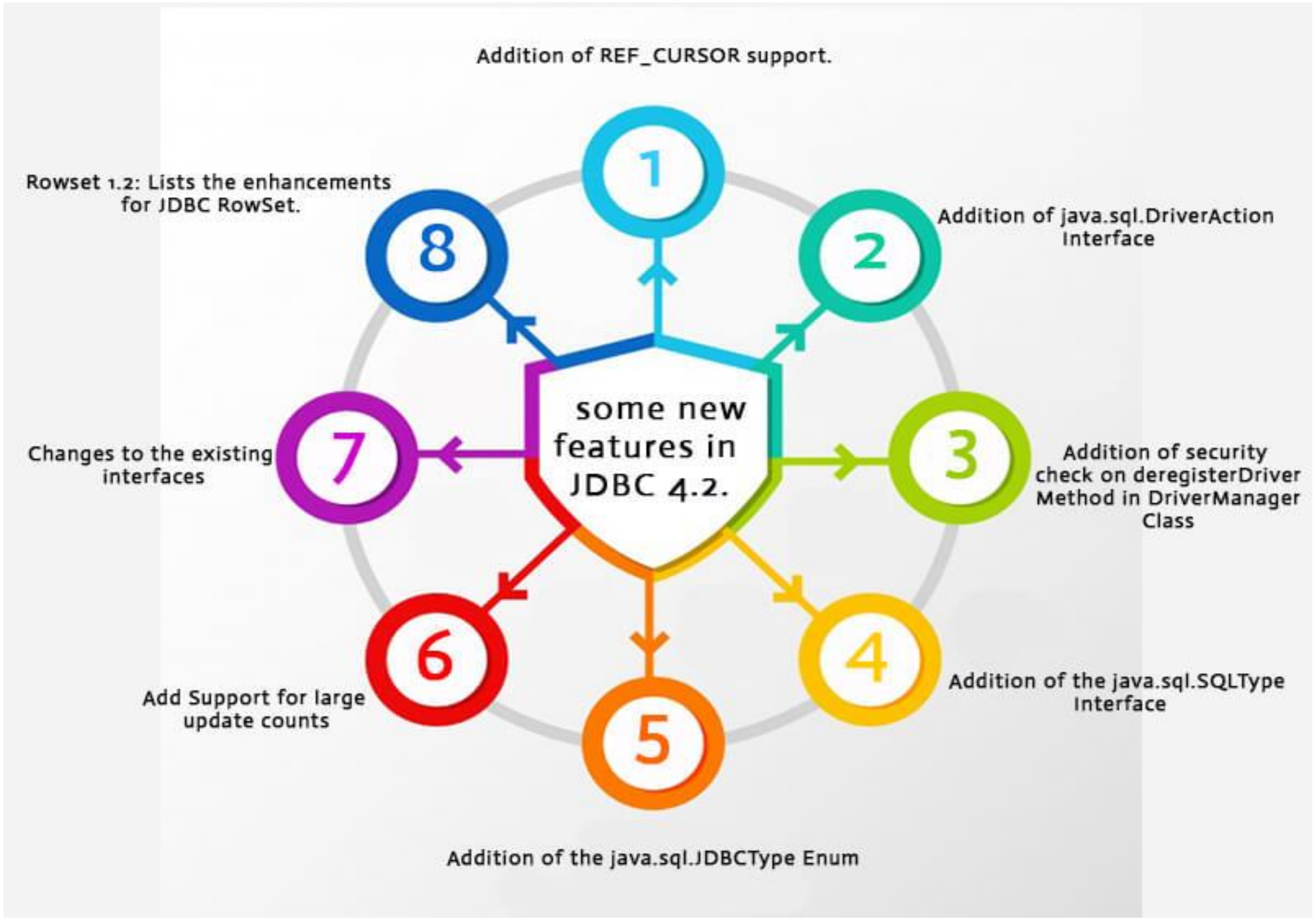
Java 8 JDBC Improvements

- The JDBC-ODBC Bridge has been removed.
- Oracle does not support the JDBC-ODBC Bridge. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Java 8 JDBC Improvements

- Added some new features in JDBC 4.2.
- Java JDBC 4.2 introduces the following features:
- Addition of REF_CURSOR support.
- Addition of java.sql.DriverAction Interface
- Addition of security check on deregisterDriver Method in DriverManager Class
- Addition of the java.sql.SQLType Interface
- Addition of the java.sql.JDBCType Enum
- Add Support for large update counts
- Changes to the existing interfaces
- Rowset 1.2: Lists the enhancements for JDBC RowSet.

Java 8 JDBC Improvements


Addition of REF_CURSOR support.
Addition of java.sql.DriverAction Interface
Addition of security check on deregisterDriver Method in DriverManager Class
Addition of the java.sql.SQLType Interface
Addition of the java.sql.JDBCType Enum
Add Support for large update counts
Changes to the existing interfaces
Rowset 1.2: Lists the enhancements for JDBC RowSet.
some new features in JDBC 4.2.
1
2
3
4
5
6
7
8

Questions

Module Summary

- Lambdas
- Stream API
- Base 64
- Nashorn
- Reflection
- Date Time API
- Repeated Annotations
- JDBC Improvements