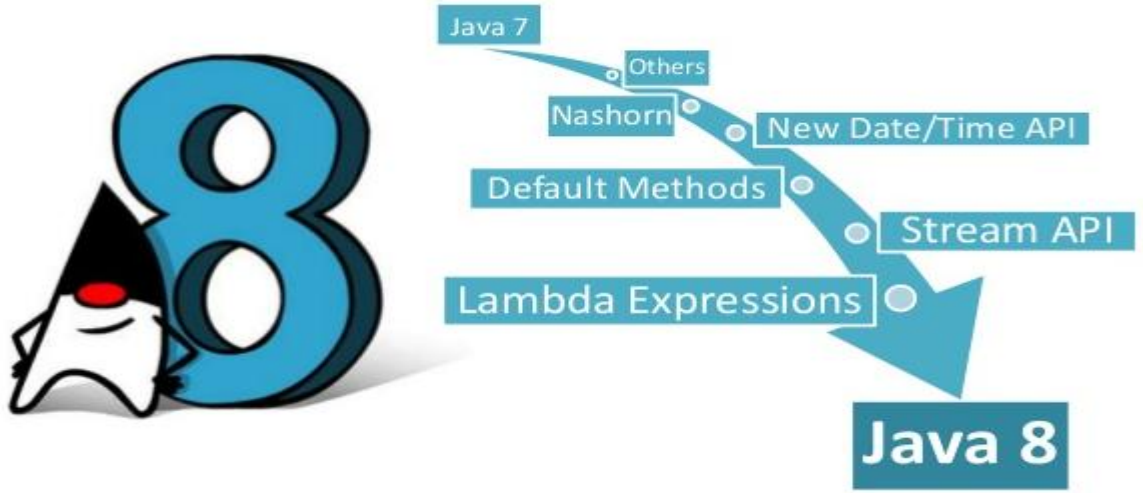
Java 8 - Features

Introduction to Java 8

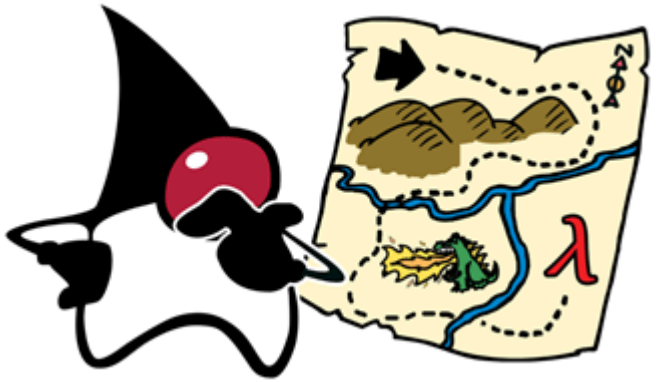
- Java 8 was released in early 2014.
- In java 8, most talked about feature was lambda expressions.
- It has many other important features as well such as default methods, stream API and new date/time API.



New Features of Java 8

New Features in Java language

- Lambda Expression
- Functional Interface
- Interface's default and Static Methods
- Method References



New Features in Java libraries

- Stream API
- Date/Time API

- **Nashorn, JavaScript Engine** – A Java-based engine to execute JavaScript code.

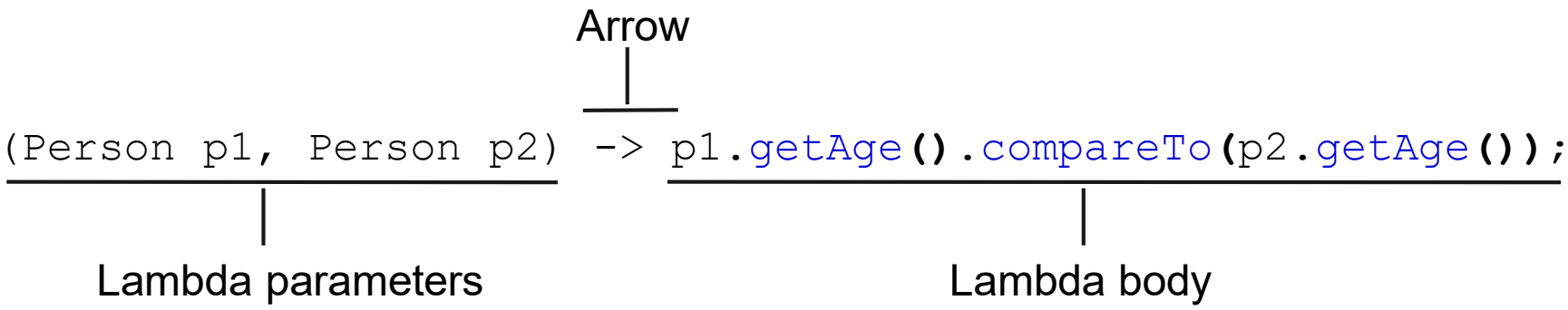
Lambda Expressions

- In programming, a Lambda expression (or function) is just an anonymous function.
- i.e., a function with no name and without being bounded to an identifier.
- They are written exactly in the place where it's needed, typically as a parameter to some other function.
- Unnamed block of code (or an unnamed function) with a list of formal parameters and a body.
- Lambda expression facilitates functional programming, and simplifies the development a lot.
- Its helpful in Collections, to iterate, filter & extract data.

The basic syntax of a lambda expression

```
(parameters) -> expression
or
(parameters) -> { statements; }
or
() -> expression
```

- Optional type declaration
- Optional parenthesis around parameter
- Optional curly braces
- Optional return keyword



Lambda - Examples

```
(int a, int b) ->    a * b       // takes two integers and
                                  returns their multiplication
(a, b) ->   a - b                // takes two numbers and returns
                                       their difference
() -> 99                        // takes no values and returns 99
(String a) -> System.out.println(a)   // takes a string, prints
                  its value to the console, and returns nothing
a -> 2 * a          // takes a number and returns the result of
                       doubling it
c -> { //some complex statements } // takes a collection and do
                                       some processing
```

Rules for writing lambda expressions

- A lambda expression can have zero, one or more parameters.
- The type of the parameters can be explicitly declared or it can be inferred from the context.
- Multiple parameters are enclosed in mandatory parentheses and separated by commas.
- Empty parentheses are used to represent an empty set of parameters.
- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a*a.

Rules for writing lambda expressions

- The body of the lambda expressions can contain zero, one or more statements.
- If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
- When there is more than one statement in body than these must be enclosed in curly brackets.

Lambda Expressions

Examples:

```
1.(String  s)  -> s.length()
2.(Person  p)  -> p.getAge()  > 20
3. ()  -> 92
4.      (int x,   inty)  ->x +y
5.      (int x,   inty)  ->      {
        System.out.println("Result:");
        System.out.println(x  +  y);
    }
```

What is a Functional interface?

- New term of Java 8
- A functional interface is an interface with only one *abstract* method.

```
public interface Runnable {

   run();
};
```

```
public interface Comparator<T> {

   int compare(T t1, T t2);
};
```

Functional Interface

- Single Abstract Method interfaces (SAM Interfaces) is not a new concept.
- It means interfaces with only one single method.
- In java, we already have many examples of such SAM interfaces.
- From java 8, they will also be referred as functional interfaces as well.
- 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.

Functional Interface

- Methods from the Object class don't count.

```
public interface MyFunctionalInterface {

    someMethod();

    /**
     * Some more documentation
     */
    equals(Object o);
};
```

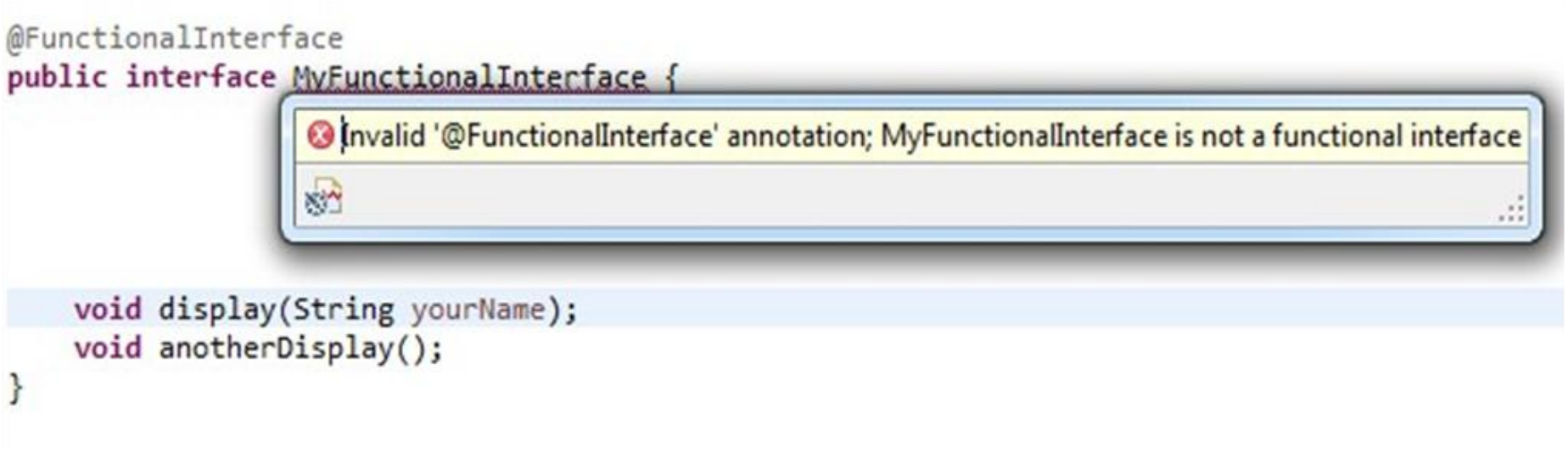
Annotation in Functional Interface

- A functional interface can be annotated.
- It's optional.

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void display(String yourName);
}
```

Functional Interface

- Show compile error when define more than one abstract method.



Lambda vs Functional Interface

- We know that Lambda expressions are anonymous functions with no name and they are passed (mostly) to other functions as parameters.
- In java method parameters always have a type and this type information is looked for to determine which method needs to be called in case of method overloading or even simple method calling.
- So, basically every lambda expression also must be convertible to some type to be accepted as method parameters.
- Well, that type in which lambda expressions are converted, are always of functional interface type.

Where should we use Lambda?

If we have to write a thread which will print “howtodoinjava” in console then simplest code will be:

```
new Thread(new Runnable() {
    @Override
    public void run() {
System.out.println("howtodoinjava")
;
    }
}).start();
```

If we use the lambda expression for this task then code will be :

```
new Thread(
        () ->   {
                System.out.println("My
Runnable");
                }
        ).start();
```

Lambda vs FI

- In simple words, a lambda expression is an instance of a functional interface.
- But a lambda expression itself does not contain the information about which functional interface it is implementing;
- That information is deduced from the context in which it is used.

Lambda Expression

Runnable:

```
// Anonymous class
Runnable r1 = new Runnable(){
    @Override
    public void run(){
        System.out.println("Hello world one!");
    }
};
// Lambda Runnable
Runnable r2 = () -> System.out.println("Hello world two!");
```

Lambda Expression

Iterator:

```
List<String> features = Arrays.asList("Lambdas", "Default
Method", "Stream API","Date  and Time API");
//Prior to Java 8 :
for (String feature : features) {
    System.out.println(feature);
}
//In Java 8:
Consumer<String> con = n -> System.out.println(n)
features.forEach(con);
```

Why Lambda Expressions ?

Enable to treat functionality as a method argument, or code as data.

A function that can be created without belonging to any class.

A lambda expression can be passed around as if it was an object and executed on demand.

Predefined Functional Interface

- Java 8 introduced a set of predefined functional interfaces within the java.util.function package.
- It facilitates functional programming with lambda expressions and method references.
- These interfaces provide common functional patterns, reducing the need to define custom functional interfaces for every scenario.

Java 8 Method Reference

- In Java, we can use references to objects, either by creating new objects or by using existing objects.

```
List list = new ArrayList();
List list2 = list;
```

- If we only use a method of an object in another method, we still have to pass the full object as an argument.
- Wouldn't it be more practical to just pass the method as an argument.

```
isFull(list.size);
```

Method References

- It is a Shorthand notation of a Lambda expression to call a method
- If your Lambda expression is like this:

str ->System.out.println(str)

- Then you can replace it with method reference like this:

System.out::println

- The :: operator is used in method reference to separate the class or object from the method name.

Java 8 Method Reference Operator - ::

- In Java 8, thanks to lambda expressions, we can use methods as if they were objects, or primitive values.
- A method reference is the shorthand syntax for a lambda expression that executes just ONE method.
- Here's the general syntax of a method reference:

Object :: methodName

- In a method reference, you place the object (or class) that contains the method before the **::** operator and the name of the method after it without arguments.

Consumer<String> c = System.out::println;

Method Reference

- So to use a method reference, you first need a lambda expression with one method.
- And to use a lambda expression, you first need a functional interface, an interface with just one abstract method.

```
list.forEach(System.out::println);
```

Method References

- There are four kinds of method references:
  - Reference to a static method
  - Reference to an instance method of a particular object
  - Reference to an instance method of an arbitrary object of a particular type
  - Reference to a constructor

Method References

- Reference to a static method

The syntax: **ContainingClass::staticMethodName**

```
package java8.methodreferences;

public class MetRefThread {

    public static void runBody() {
        for (int i = 0; i < 10; i++) {
            System.out.println("square of i is " + (i * i));
        }
    }

    public static void main(String[] args) {
        new Thread(MetRefThread::runBody).start();
    }
}
```

Method References

- Reference to an instance method of a particular object

The syntax: **containingObject::instanceMethodName**

```
public class ObjectMetRef {
    void startsWith(String s, String b) {
        System.out.println(String.valueOf(s.charAt(0)));
        System.out.println(String.valueOf(b.charAt(0)));
    }
    public static void main(String[] args) {
        ObjectMetRef something = new ObjectMetRef();
        Converter<String, String> converter = something::startsWith;
        converter.convert("Java", "8");    // "J" and "8"
    }
}
```

Method References

- Reference to an instance method of an arbitrary object of a particular type

The syntax: **ContainingType::methodName**

```
package java8.methodreferences;

import java.util.Arrays;

public class ArraySort {
    public static void main(String[] args) {
        String[] strArray = { "abe", "adb", "deb", "abc", "ghi", "acd", "acg","acb" };
        Arrays.sort(strArray, String::compareToIgnoreCase);
        for (String str : strArray) {
            System.out.print(str + " ");
        }
    }
}
```

Method References

- Reference to a constructor

The syntax: **ClassName::new**

```
public class ConstructorReference {

    private String content;

    public ConstructorReference() {
        this.content = "created by constructor reference";
    }

    public String getContent() {
        return content;
    }

    public static void main(String... args) {
        ConstructorReferenceArg<ConstructorReference>
            constructorSam = ConstructorReference::new;
        System.out.println("\n\n\tcontent = "
                + constructorSam.whatEverMethodName().getContent());
    }
}
```

```
public interface ConstructorReferenceArg<T> {
    T whatEverMethodName();
}
```

Method References

- Method references as Lambdas Expressions

Syntax	Example	As Lambda
ClassName::new	String::new	() -> new String()
Class::staticMethodName	String::valueOf	(s) -> String.valueOf(s)
object::instanceMethodName	x::toString	() -> "hello".toString()
Class::instanceMethodName	String::toString	(s) -> s.toString()

Java Predicate Interface

- It is a functional interface which represents a predicate (boolean-valued function) of one argument.
- It is defined in the java.util.function package.
- It contains test() a functional method.
- It improves manageability of code, helps in unit-testing them separately

Java Predicate Interface Methods

Methods	Description
boolean test(T t)	It evaluates this predicate on the given argument.
default Predicate<T> and(Predicate<? super T> other)	It returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. When evaluating the composed predicate, if this predicate is false, then the other predicate is not evaluated.
default Predicate<T> negate()	It returns a predicate that represents the logical negation of this predicate.
default Predicate<T> or(Predicate<? super T> other)	It returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. When evaluating the composed predicate, if this predicate is true, then the other predicate is not evaluated.
static <T> Predicate<T> isEqual(Object targetRef)	It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

Java Predicate Interface Example

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

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

        Predicate<Integer> pr = a -> (a > 18); // Creating predicate
        System.out.println(pr.test(10));    // Calling Predicate method
    }
}
```

Java 8 - Default Methods

- Java provides a facility to create default methods inside the interface.
- Java 8 allows you to add non-abstract methods in interfaces.
- These methods must be declared default methods.
- Default methods were introduces in java 8 to enable the functionality of lambda expression.
- Methods which are defined inside the interface and tagged with default are known as default methods.

Default Methods

- These methods are non-abstract methods.
- This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8.
- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Default Methods - Example

```
public interface Moveable {
    default void move(){
        System.out.println("I am moving");
    }
}
```

- Moveable interface defines a method move() and provided a default implementation as well.
- If any class implements this interface then it need not to implement it’s own version of move() method. It can directly call instance.move()

Default Methods - Example

```
public class Animal implements Moveable{
    public static void main(String[] args){
        Animal tiger = new Animal();
        tiger.move();
    }
}
```

- If class willingly wants to customize the behavior of move() method then it can provide it’s own custom implementation and override the method.

Static Methods

- The interfaces can have static methods as well which is similar to static method of classes.

Java 8 – Streams API

- A **Stream in Java** can be defined as a sequence of elements from a source that supports aggregate operations on them.
- The source here refers to a Collections or Arrays who provides data to a Stream.
- Stream keeps the ordering of the data as it is in the source.
- The **aggregate operations** or **bulk operations** are operations which allow us to express common manipulations on stream elements easily and clearly.

Java 8 – Streams API

- Java provides a new additional package in Java 8 called **java.util.stream**.
- Another major change introduced Java 8 is Streams API.
- It provides a mechanism for processing a set of data in various ways.
- It include filtering, transformation, or any other way that may be useful to an application.
- Streams API in Java 8 supports a different type of iteration.
- Stream does not store elements.
- It defines methods for the set of items to be processed, the operation(s) to be performed on each item, and where the output of those operations is to be stored.

Java Stream vs. Collection

- A **Collection is an in-memory data structure**, which holds all the values that the data structure currently has—every element in the Collection has to be computed before it can be added to the Collection.
- A **Stream is a conceptually fixed data structure, in which elements are computed on demand**.
- This gives rise to significant programming benefits.
- The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—as and when required.

Characteristics of Stream

- Not a data structure
- Designed for lambdas
- Do not support indexed access
- Can easily be outputted as arrays or lists
- Lazy access supported
- Parallelizable

Lazy Access

- Lazy Class Loading
  - JRE has built-in lazy initialisation of classes.
  - Classes load into memory only when they are first referenced.
  - It reduces memory usage
- Lazy Object Creation
  - Objects are created at run time whenever it is required.

Stream

Add text here...

- Not store data
- Designed for processing data
- Not reusable
- Can easily be outputted as arrays or lists

Java 8 Stream API - Types

Streams are designed for processing collections of data in a functional style and are distinct from I/O streams.

Sequential Streams: Process elements in a collection or data source sequentially in a single thread. This is the default behavior when creating a stream.

Parallel Streams: Process elements concurrently using multiple threads, leveraging multi-core processors for potentially faster execution, especially with computationally intensive tasks.

Primitive Streams: Specialized streams for primitive data types to avoid autoboxing/unboxing overhead.

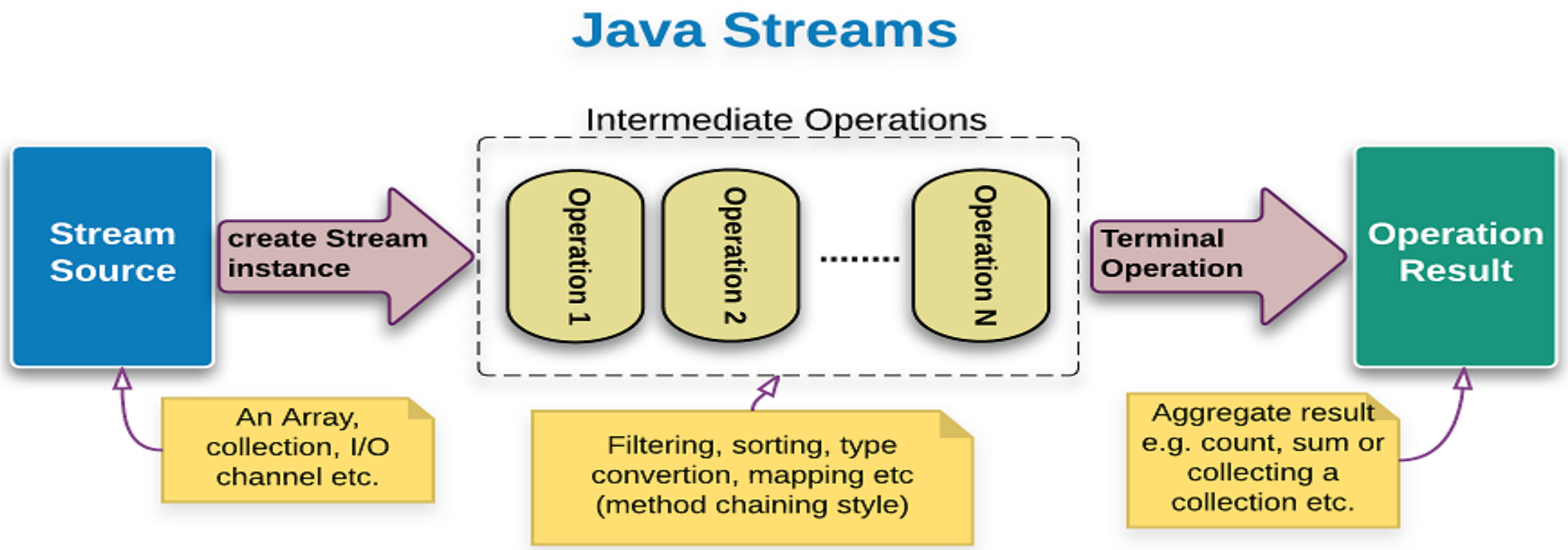
- IntStream: For int values.
- LongStream: For long values.
- DoubleStream: For double values.

Infinite Streams: Streams that can produce an unbounded number of elements, often created using Stream.iterate() or Stream.generate().

Stream - How to use

1. Build a stream
2. Transform stream
3. Collect result

Working of Streams



Stream - build streams

Build streams from collections

```
List<Dish> dishes = new ArrayList<Dish>();
....(add some Dishes)....
Stream<Dish> stream = dishes.stream();
```

stream() – Returns a sequential stream considering collection as its source.

Stream - build streams

Build streams from arrays

```
Integer[] integers =
   {1, 2, 3, 4, 5, 6, 7, 8, 9};
Stream<Integer> stream = Stream.of(integers);
```

Different Operations On Streams

- **Intermediate Operations**
  - Each intermediate operation is lazily executed and returns a stream as a result.
  - Hence various intermediate operations can be pipelined.
- **Terminal Operations**
  - Terminal operations mark the end of the stream and return the result.

Stream Methods -Intermediate Operations

1.map: The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

2.filter: The filter method is used to select elements as per the Predicate passed as argument.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

3.sorted: The sorted method is used to sort the stream.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

Stream Methods - Terminal Operations

1.collect: The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

2.forEach: The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

3.reduce: The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

Java 8 – Date & Time API

- Java 8 introduces a new date-time API under the package **java.time.**
- It simply change the way we have been handling dates in java applications.

- Important classes introduced in java.time package
  - **Local –** Simplified date-time API with no complexity of timezone handling.
  - **Zoned –** Specialized date-time API to deal with various timezones.

Date & Time API

- Date class has even become obsolete. The new classes intended to replace Date class are :

The **LocalDate** class represents a date. There is no representation of a time or time-zone.

The **LocalTime** class represents a time. There is no representation of a date or time-zone.

The **LocalDateTime** class represents a date-time. There is no representation of a time-zone.

Date & Time Example

```
LocalDate localDate = LocalDate.now();
LocalTime localTime = LocalTime.of(12, 20);
LocalDateTime localDateTime = LocalDateTime.now();
OffsetDateTime offsetDateTime = OffsetDateTime.now();
ZonedDateTime zonedDateTime =
ZonedDateTime.now(ZoneId.of("Europe/Paris"));
```

For date functionality with Zone information, Timezone offset can be represented

Date and Time API (JSR 310)

- Provides a new date, time and calendar API for the Java SE platform
- Examples:

```
Clock clock = Clock.systemUTC();
ZoneId zone = ZoneId.systemDefault();
ZoneId customZone = ZoneId.of("Europe/Berlin");
Clock customClock = Clock.system(customZone);
DateTimeFormatter formatter = DateTimeFormatter.
        ofPattern("MM-DD-YYYY");
LocalDate date = LocalDate.now();
String formattedDate = date.format(formatter);
System.out.println(formattedDate);
```

Java 8 - Optional Class

- Java introduced a new class Optional in jdk8.
- Optional is a container object used to contain not-null objects.
- It is a public final class and used to deal with NullPointerException 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.
- It avoids any runtime NullPointerExceptions and supports us in developing clean and neat Java APIs or Applications.

Advantages of Java 8 Optional

- Null checks are not required.
- No more NullPointerException at run-time.
- We can develop clean and neat APIs.
- No more Boiler plate code

Optional Class Methods

- **Optional.ofNullable()** method returns a Non-empty Optional if a value present in the given object. Otherwise returns empty Optional.
- **Optionaal.empty()** method is useful to create an empty Optional object.
- **Optional.filter()** If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.
- **Optional.isPresent()** returns true if the given Optional object is non-empty. Otherwise it returns false.
- **Optional.ifPresent()** performs given action if the given Optional object is non-empty. Otherwise it returns false.
- **Optional.orElse()** returns the value if present in Optional Container. Otherwise returns given default value.

Java 8 - Nashorn JavaScript

- With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino.
- Nashorn is a JavaScript engine developed in the Java language.
- Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM.
- Nashorn uses invoke dynamics feature, introduced in Java 7 to improve performance.
- A command line tool (**jjs**) for running javascript without using Java is also provided .

Calling JavaScript from Java

- Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.
- It is implemented by importing following packages,constructors:

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

```
ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");
```

Java 8 - Base64

- The lack of Base64 encoding API in Java is, included a decent API for it in the java.util package. .
- Java 8 now has inbuilt encoder and decoder for Base64 encoding.
- In Java 8, we can use three types of Base64 encoding.
- *Base64* is a binary-to-text encoding scheme that represents binary data in a printable ASCII string format.
- Each Base64 digit represents exactly 6 bits of binary data.
- Base64 is used to prevent data from being modified while in transit through information systems, such as email, that might not be 8-bit clean (they might garble 8-bit values).

Methods

- **static Base64.Decoder getDecoder()** - Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.

- **static Base64.Encoder getEncoder() -** Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.

Java Volatile Keyword

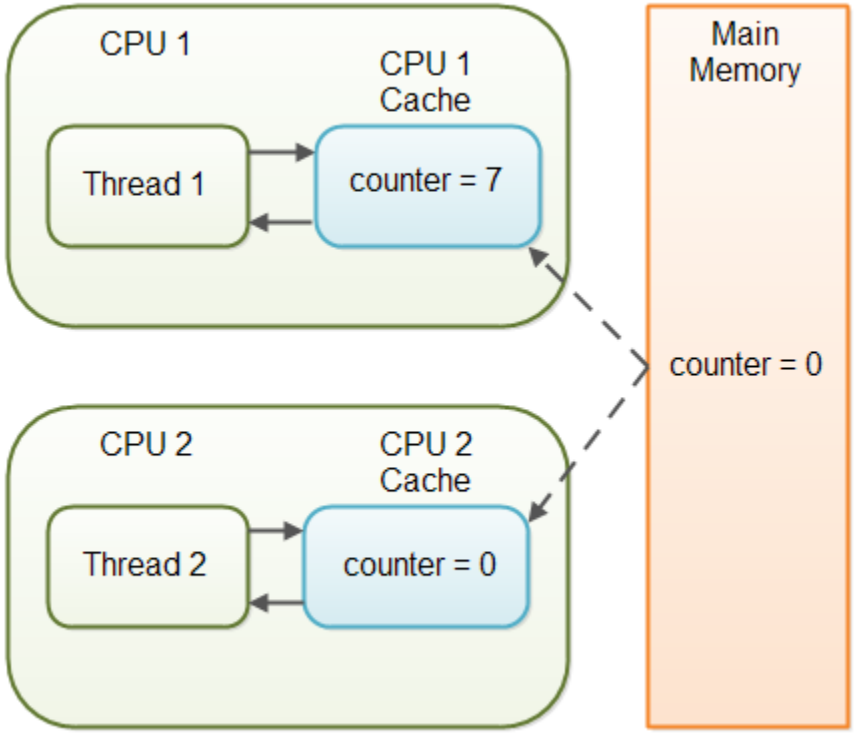
- The Java volatile keyword is used to mark a Java variable as "being stored in main memory".
- Every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache.
- Every write to a volatile variable will be written to main memory, and not just to the CPU cache.
- Since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory.

Volatile

- The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

- Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Volatile



- With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems .

```
public class SharedObject {

    public int counter = 0;

}
```

Volatile Visibility Guarantee

- The Java volatile keyword is intended to address variable visibility problems.
- By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately.
- Also, all reads of the counter variable will be read directly from main memory.
- Declaring a variable volatile thus guarantees the visibility for other threads of writes to that variable.

```
public class SharedObject {
  public volatile int counter = 0;  }
```

END