Java Functional Programming

RAHUL

Why Java 8 should we used

- Lambda Expressions: Simplify the development process and improve the readability of code by enabling functional programming.
- Stream API: Allows for efficient and easy processing of collections of objects.
- New Date and Time API: Provides a more comprehensive and improved way of handling date and time.
- Default Methods: Enable the addition of new methods to interfaces without breaking existing implementations.

RAHUL

Lambda Expression

- Lambda provide a clear and concise way to represent an anonymous function (i.e., a function without a name) and simplify the syntax for writing instances of single-method interfaces (functional interfaces).

```
() -> ()
```

Parameters

Token

Body

RAHUL

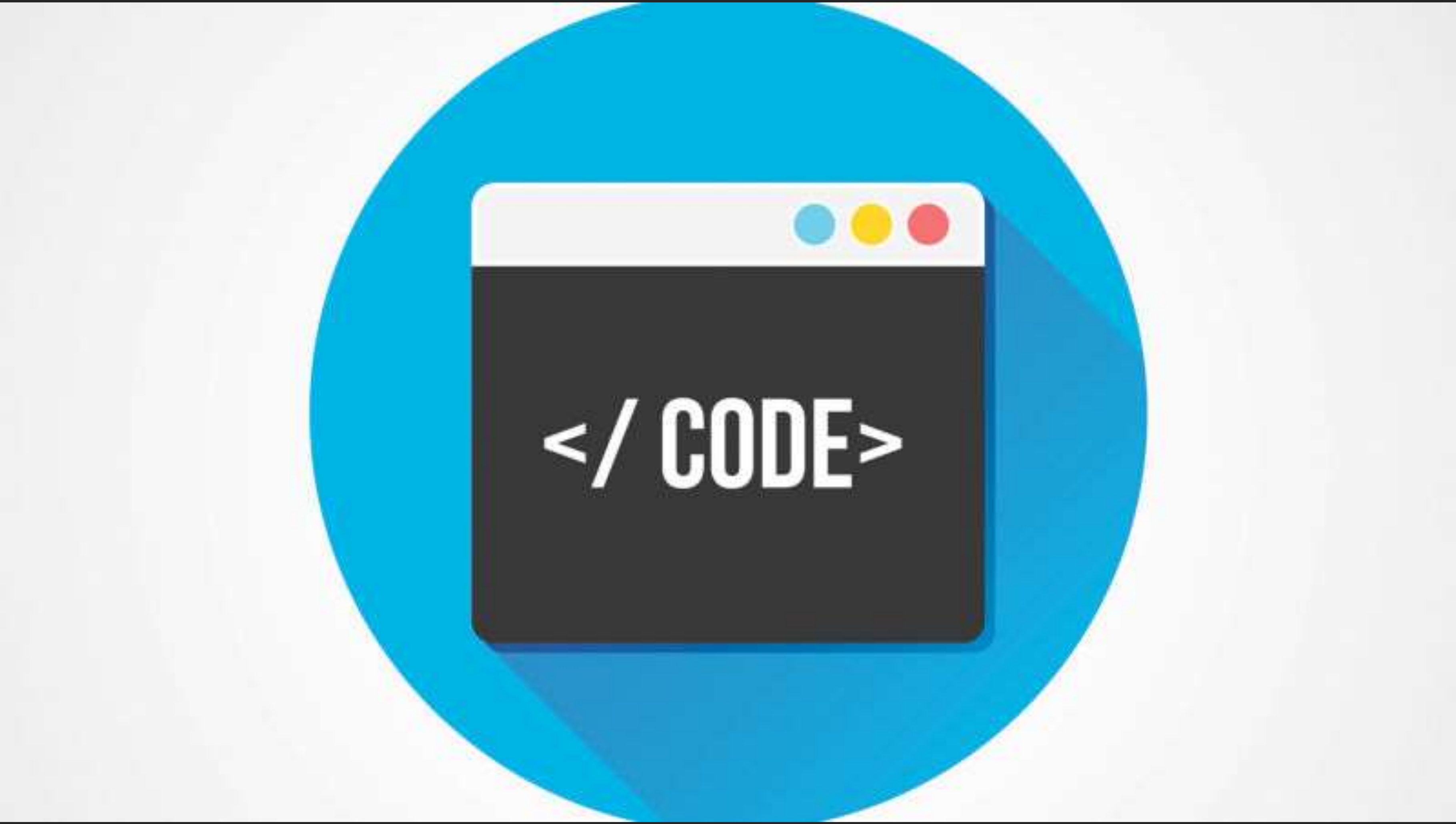
Lambda Expression

Lambda Expression can be written in below type

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

Example

- No Parameters:
```
() -> System.out.println("Hello, World!");
```
- Single Parameter:
```
x -> x * x
```
- Multiple Parameters:
```
(x, y) -> x + y
```
- Block of Statements:
```
(x, y) -> {
    int sum = x + y;
    return sum;
}
```



Lets Write Code

RAHUL

Functional Interface

- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.

- It can have default and static methods.

- *@FunctionalInterface* annotation are optional if interface is having only one method.

RAHUL

Functional Interface

Some Built-in Java Functional Interfaces

- Runnable –> This interface only contains the run() method.
- Comparable –> This interface only contains the compareTo() method.
- ActionListener –> This interface only contains the actionPerformed() method.
- Callable –> This interface only contains the call() method.

Newly Introduced

- Consumer – Bi Consumer
- Predicate – Bi Predicate
- Function – Bi Function, Unary Operator, Binary Operator
- Supplier

RAHUL

Consumer Interface

- The Consumer interface is a functional interface that represents an operation that accepts a single input argument and returns no result. It is typically used to perform some operations on the given argument.

Syntax

```
Consumer< T > consumer1 = ( T ) -> {body};

consumer1.accept(parameters)
```

T = Type of arguments

Bi-Consumer Interface

- The Bi-Consumer interface is a functional interface that represents an operation that accepts a two input arguments and returns no result. It is typically used to perform some operations on two given arguments.

Syntax

```
Consumer< T, U > consumer2 = (T, U) -> {body};

consumer2.accept(parameters)
```

T = Type of first arguments

U = Type of second arguments

Combine Two Consumer

- Both Consumer and Bi-Consumer provide a default method andThen that allows combining multiple consumers.

Syntax

```
Consumer< T > consumer1 = ( T ) -> {body};

consumer1.accept(parameters)
```

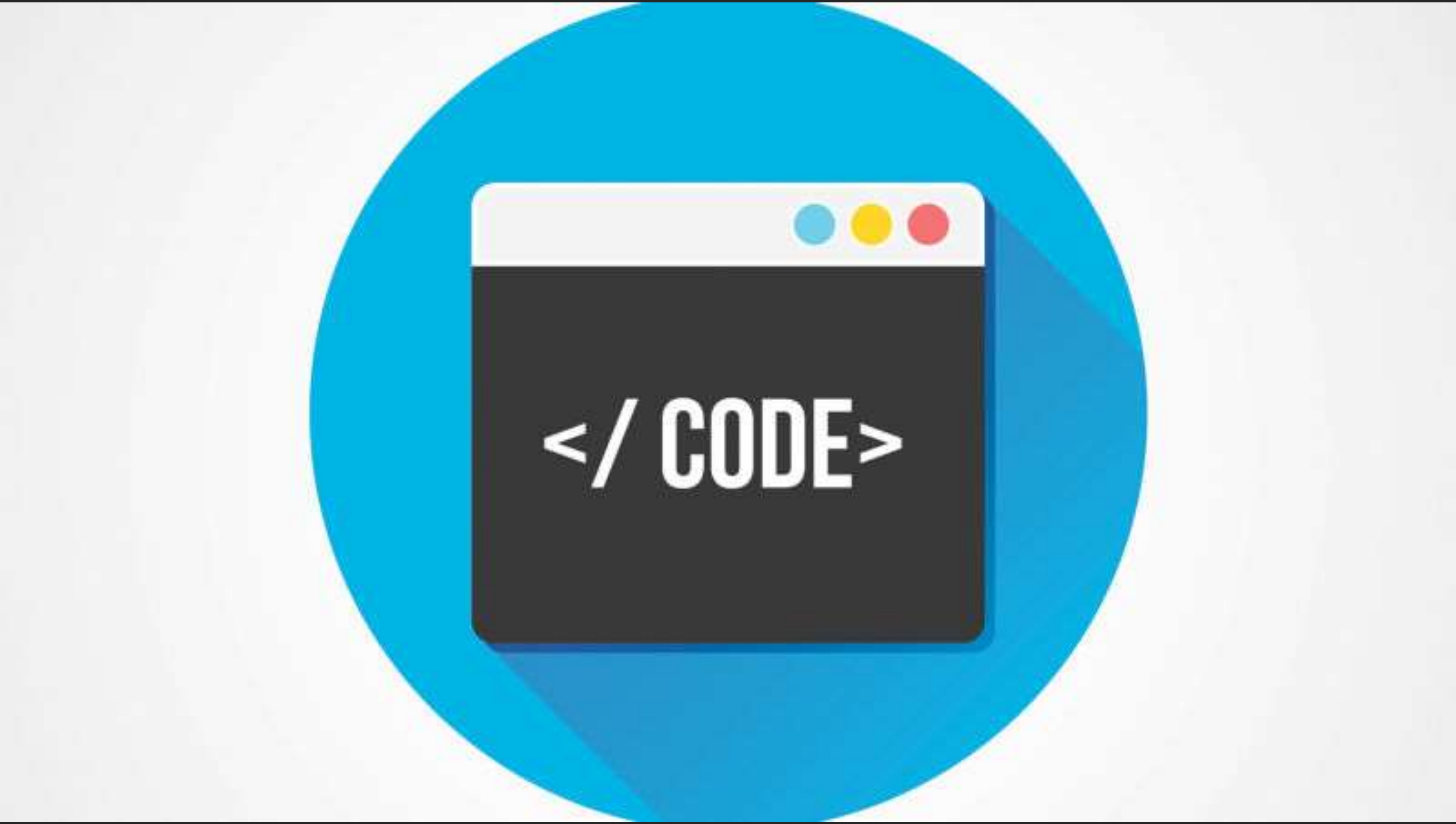
```
Consumer< T, T > consumer2 = (T, T) -> {body};

consumer2.accept(parameters)
```

Combine Two Consumer

```
consumer1.andThen(consumer2).accept(p1,p2);
```

<u>Note: When we combine two consumer both should accept same number of parameters.</u>



Lets Write Code

Predicate Interface

- The Predicate interface is a functional interface that represents a single argument function that returns a Boolean value. It is used to evaluate a condition on a given argument.

Syntax

```
Predicate< T > p1 = ( T ) -> {body};

p1.test(T)
```

T = Type of arguments

<u>Combining Predicates</u>

You can combine multiple predicates using default methods like and, or, and negate.

Bi-Predicate Interface

- The BiPredicate interface is a functional interface that represents a predicate (boolean-valued function) of two arguments.

Syntax

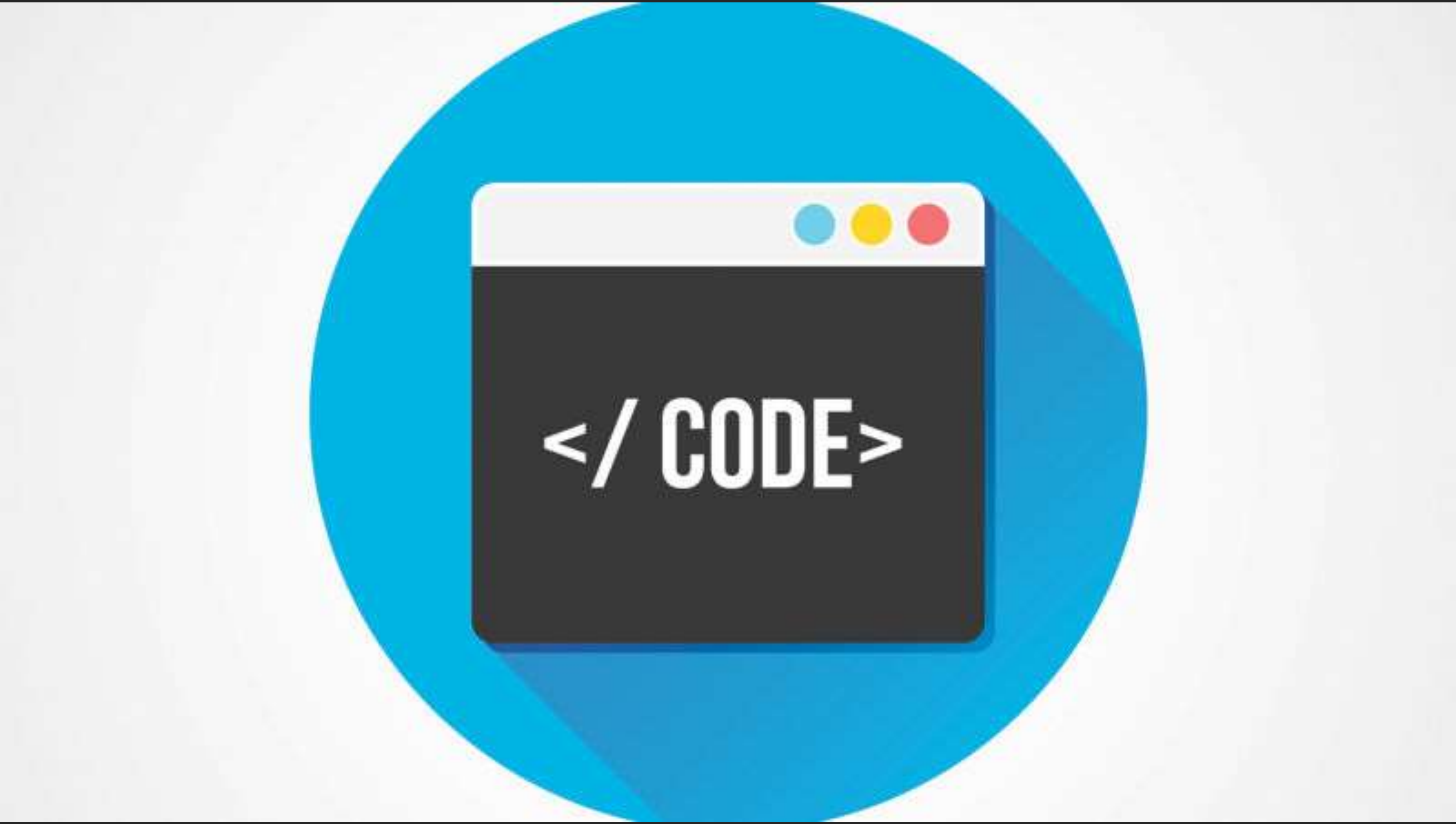
Predicate< T1,T2 > p1 = (T1,T2) -> {body};

p1.test(T1,T2)

T = Type of arguments

Combining Predicates

You can combine multiple predicates using default methods like and, or, and negate.



Lets Write Code

RAHUL

Function Interface

- The Function interface is a functional interface that represents a function that accepts one argument and produces a result. It is commonly used for defining transformations or mappings of data.

Syntax

```
Function< T,R > f1 = (T ) -> {body};

f1.apply(T)
```

T = Type of input arguments
R = Type of result arguments

Other methods

andThen

Bi-Function Interface

- The BiFunction interface is a functional interface that represents a function that accepts two arguments and produces a result. It is useful for operations that involve two inputs.

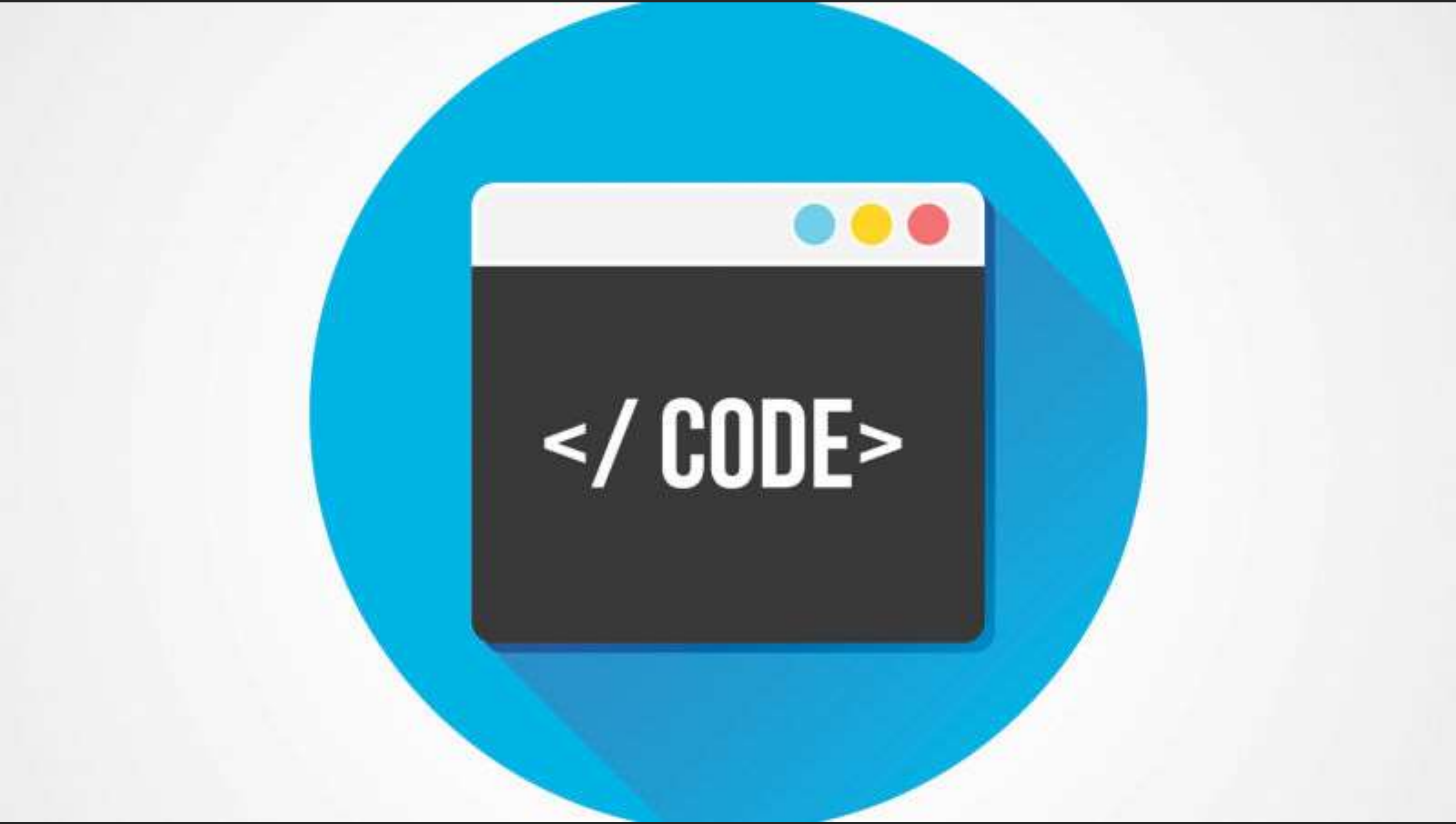
Syntax

```
BiFunction< T,T,R > f1 = (T,T ) -> {body};

f1.apply(T,T)
```

T = Type of input arguments
R = Type of result arguments

RAHUL



Lets Write Code

RAHUL

UnaryOperator Interface

- The UnaryOperator interface is a specialized form of the Function interface. It represents an operation on a single operand that produces a result of the same type as its operand. This is a functional interface with the following method:

Method

R apply(T t): Applies this function to the given argument and returns the result.

T = Type of input arguments
R = Type of result arguments

RAHUL

BinaryOperator Interface

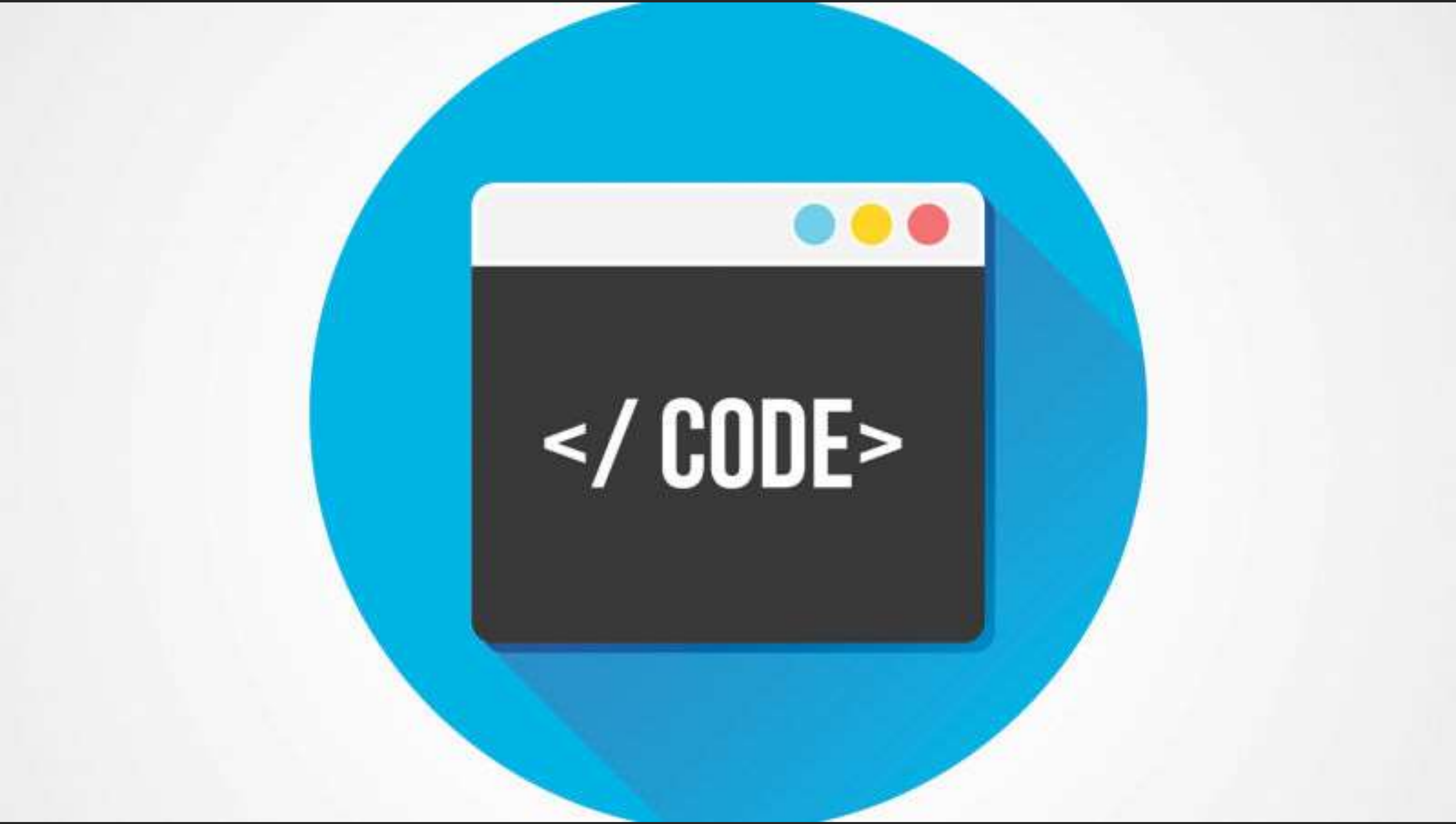
- The BinaryOperator interface is a specialized form of the BiFunction interface. It represents an operation upon two operands of the same type, producing a result of the same type as the operands.

Method

R apply(T t, U u): Applies this function to the given argument and returns the result.

T = Type of input arguments
R = Type of result arguments

We can combine Operator using default method: andThen



Lets Write Code

RAHUL

Supplier Interface

- The Supplier interface is a functional interface introduced in Java 8.
- It represents a supplier of results, meaning it produces a result of a specified type but doesn't take any arguments.

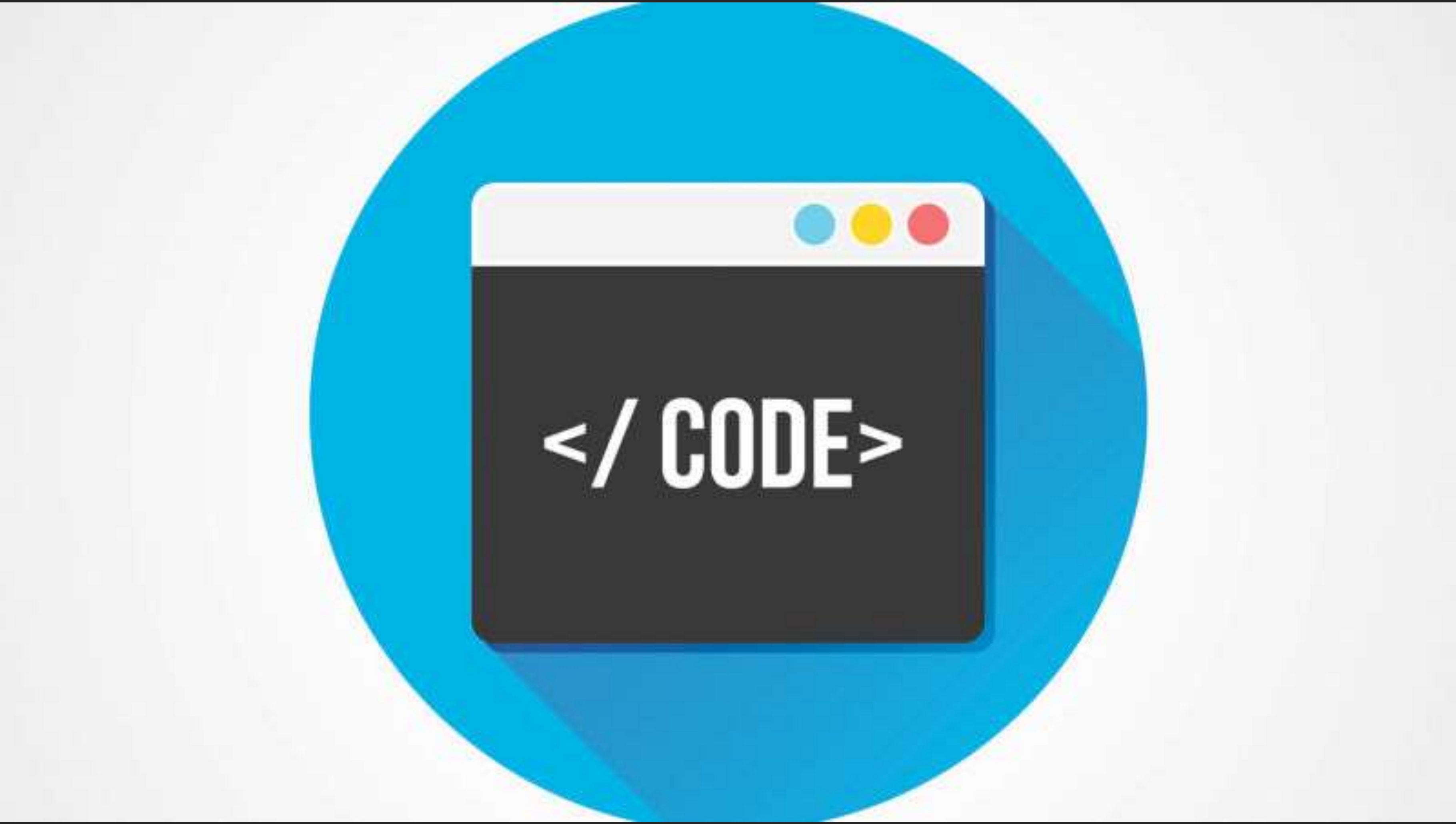
Method

T get(): Gets a result.

T = Type of input arguments
R = Type of result arguments

We can combine Operator using default method: andThen

RAHUL



Lets Write Code

RAHUL

Method Reference

AND

Constructor Reference

RAHUL

Method/Constructor Reference

- Java 8 introduced method references and constructor references as a shorthand notation to create instances of functional interfaces.

Method References

- Method references can be used to refer to a method without invoking it. They are used primarily to simplify lambda expressions.

Four Types

- 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

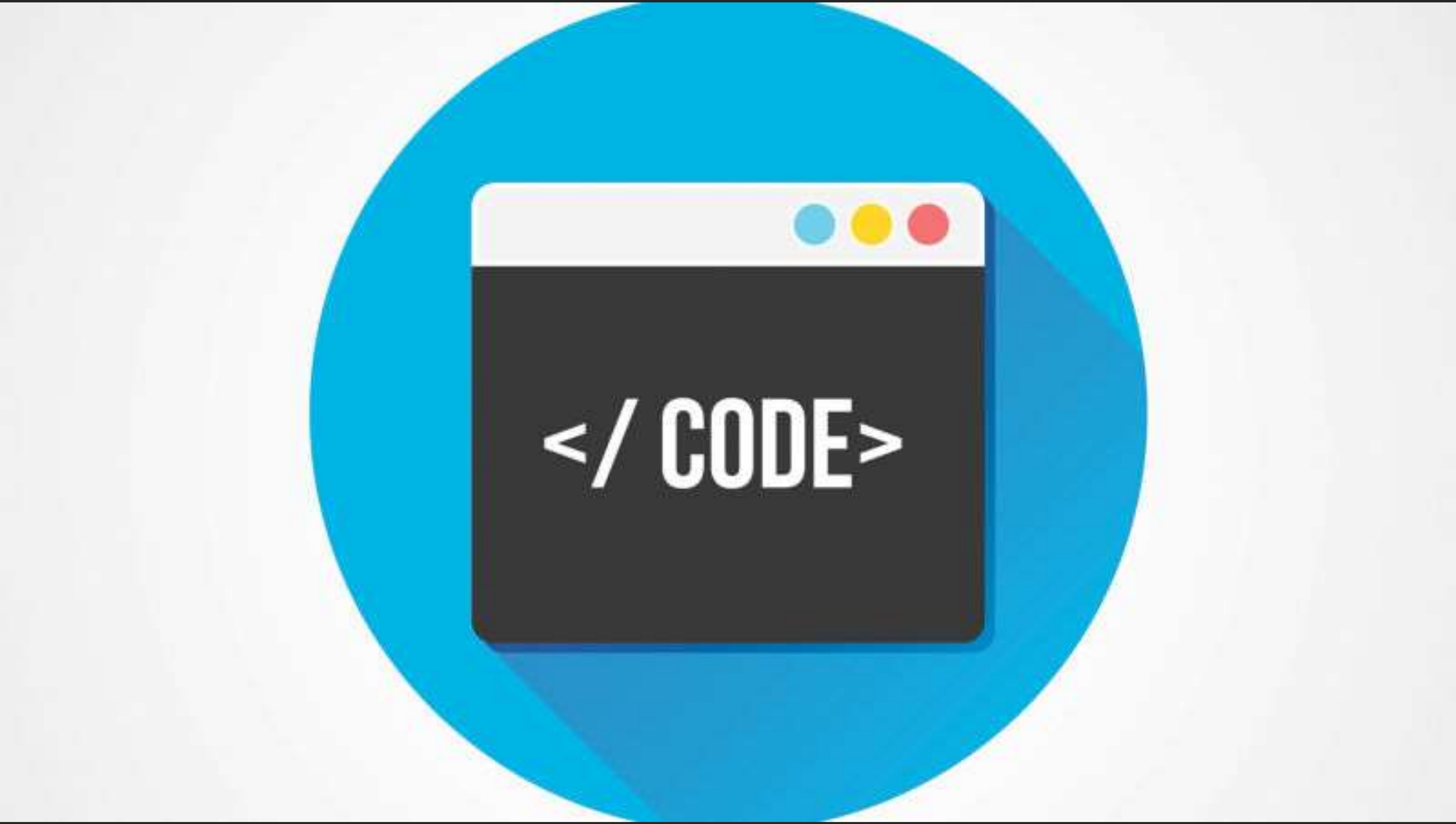
RAHUL

Method Reference

Syntax

- Static method: ClassName::staticMethodName
- Instance method of a particular object: instance::instanceMethodName
- Instance method of an arbitrary object of a particular type: ClassName::instanceMethodName
- Constructor: ClassName::new

R A H U L



Lets Write Code

RAHUL

Constructor Reference

Overview

- Constructor references are a special kind of method reference that can be used to create new objects. They provide a concise way to instantiate objects without explicitly invoking the constructor.

Lets See the Code

RAHUL

Lambda Local Variables

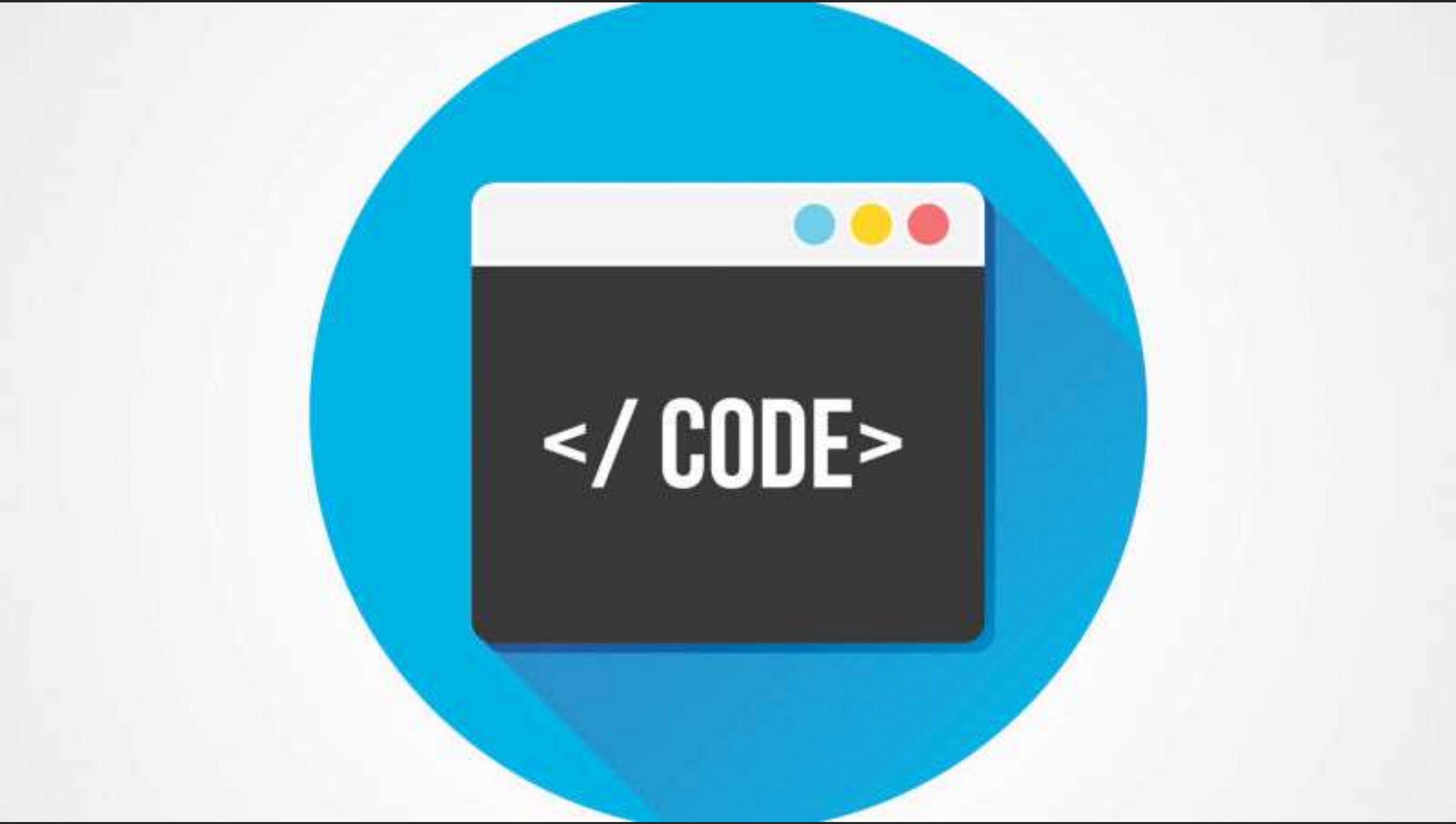
Rules for Local Variables in Lambda Expressions

- Effectively Final Variables: Local variables used in lambda expressions must be effectively final. This means that the variable should not be reassigned after its initial assignment.

- Scope: A lambda expression can access local variables from its enclosing scope, but it cannot modify them. The variables must be final or effectively final.

- Shadowing: Lambda expressions cannot define local variables with the same name as a local variable in its enclosing scope.

RAHUL



Lets Write Code

RAHUL

Difference between Stream and collections

Feature	Collections	Streams
Purpose	Store and manage groups of objects	Process sequences of elements
Storage	In-memory, elements are stored	No storage, elements are computed on demand
Evaluation	Eager	Lazy
Modification	Mutable	Immutable
Iteration	External (using iterators and loops)	Internal (using pipelines)
Size	Finite	Can be finite or infinite
Operations	Directly on elements (add, remove, update)	Functional-style operations (map, filter, reduce)
Parallel Processing	Manual (requires additional handling for concurrency)	Built-in support for parallel processing

Difference between Stream and collections

Use Cases

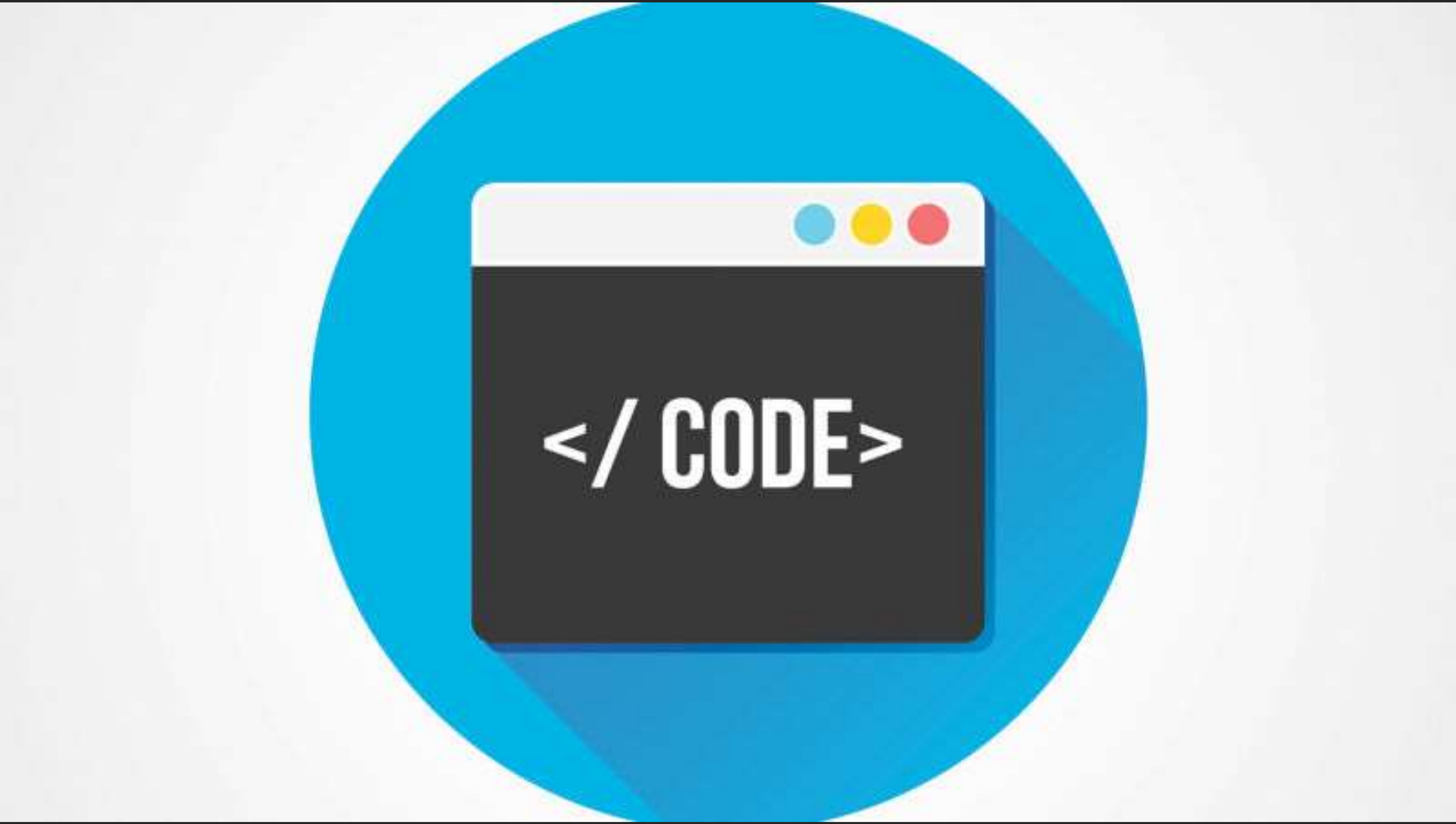
- Collections:
  - Use collections when you need to store, manage, and modify a group of objects.
  - Suitable for scenarios where direct access and modification of elements are required.
- Streams:
  - Use streams when you need to perform complex data processing and transformations.
  - Ideal for scenarios involving bulk operations, functional-style programming, and parallel processing.

Creating stream from List, Set, Map and Arrays

Stream can be created using below method

- **List:** Use list.stream().
- **Set:** Use set.stream().
- **Map:** Use map.keySet().stream(), map.values().stream(), or map.entrySet().stream().
- **Arrays:** Use Arrays.stream(array) or Stream.of(array).

RAHUL



Lets Write Code

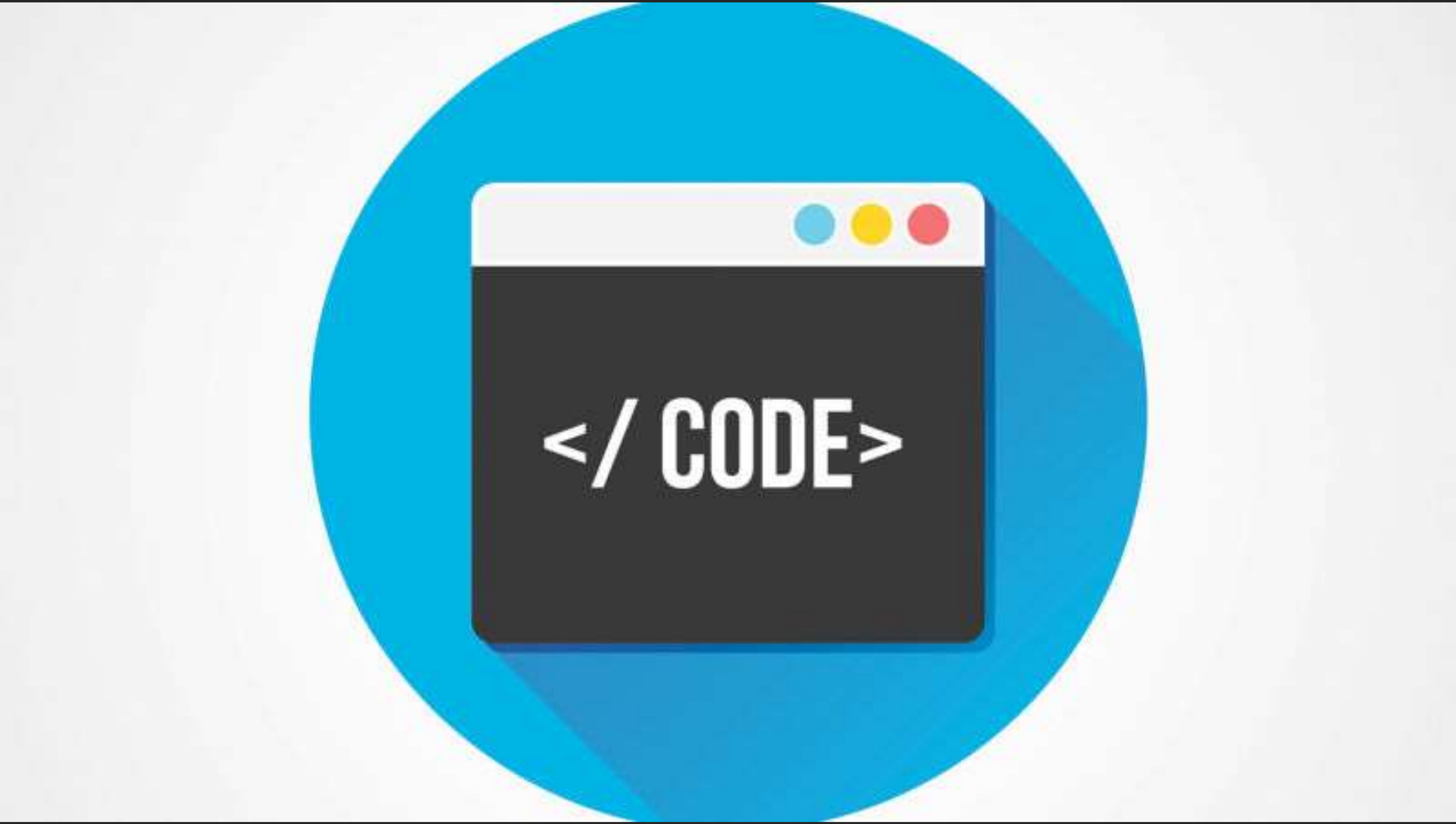
RAHUL

Intermediate Operations

Filtering: Using filter to remove elements based on conditions.

- List: Use list.stream().filter(predicate).collect(Collectors.toList()).
- Set: Use set.stream().filter(predicate).collect(Collectors.toSet()).
- Map: Use map.entrySet().stream().filter(predicate).collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)).
- Arrays: Use Arrays.stream(array).filter(predicate).toArray() or DoubleStream.of(array).filter(predicate).toArray() for primitive arrays.

RAHUL



Lets Write Code

RAHUL

Intermediate Operations

Mapping: Transforming elements using map and flatMap.

- The map method applies a function to each element in the stream and returns a new stream consisting of the results of applying the function to the elements of the original stream.

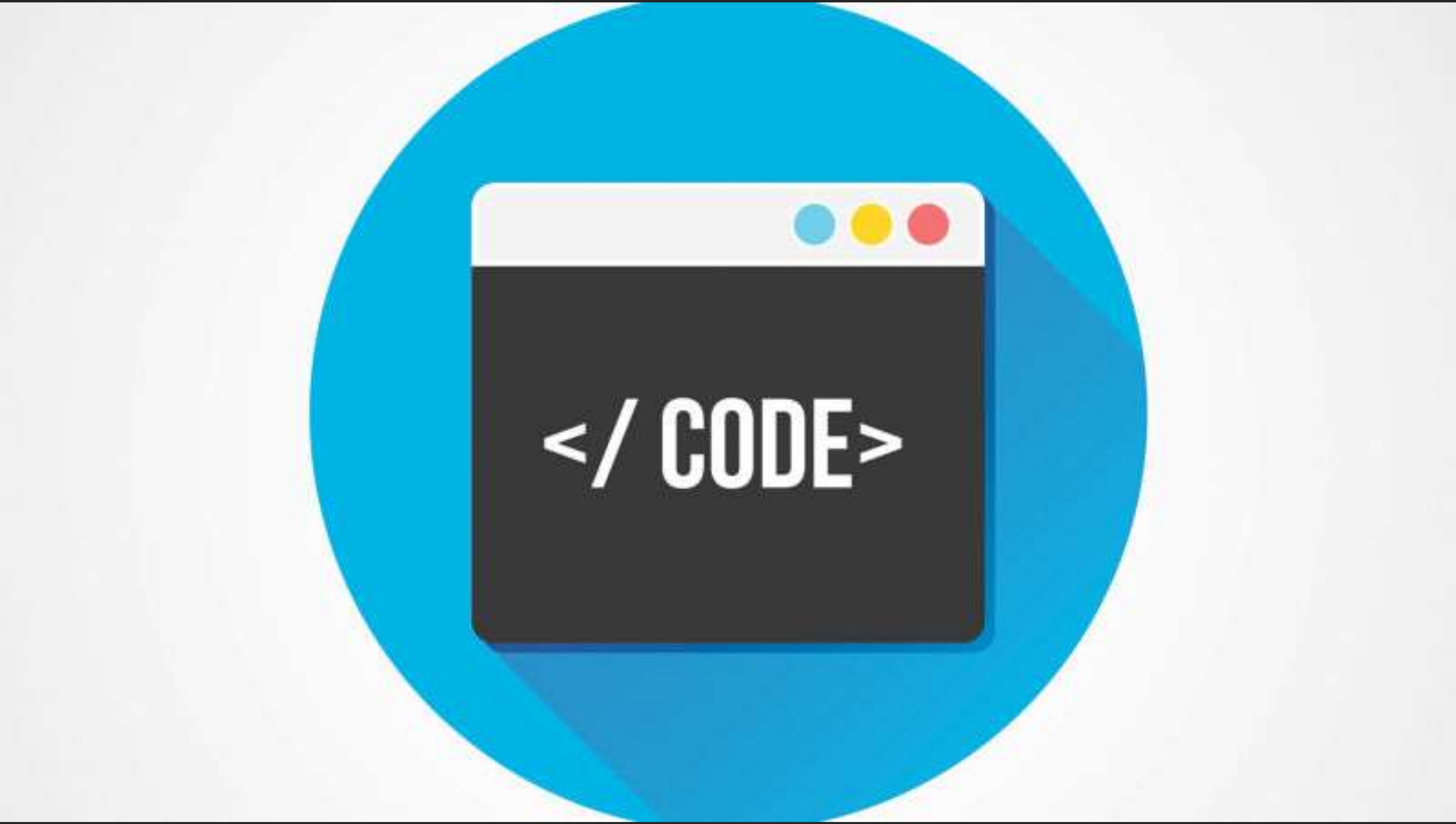
RAHUL

Intermediate Operations

Mapping: Transforming elements using map and flatMap.

- The flatMap method is used to transform each element into a stream of elements and then flatten these streams into a single stream. This is particularly useful when you have nested structures and want to flatten them.

RAHUL



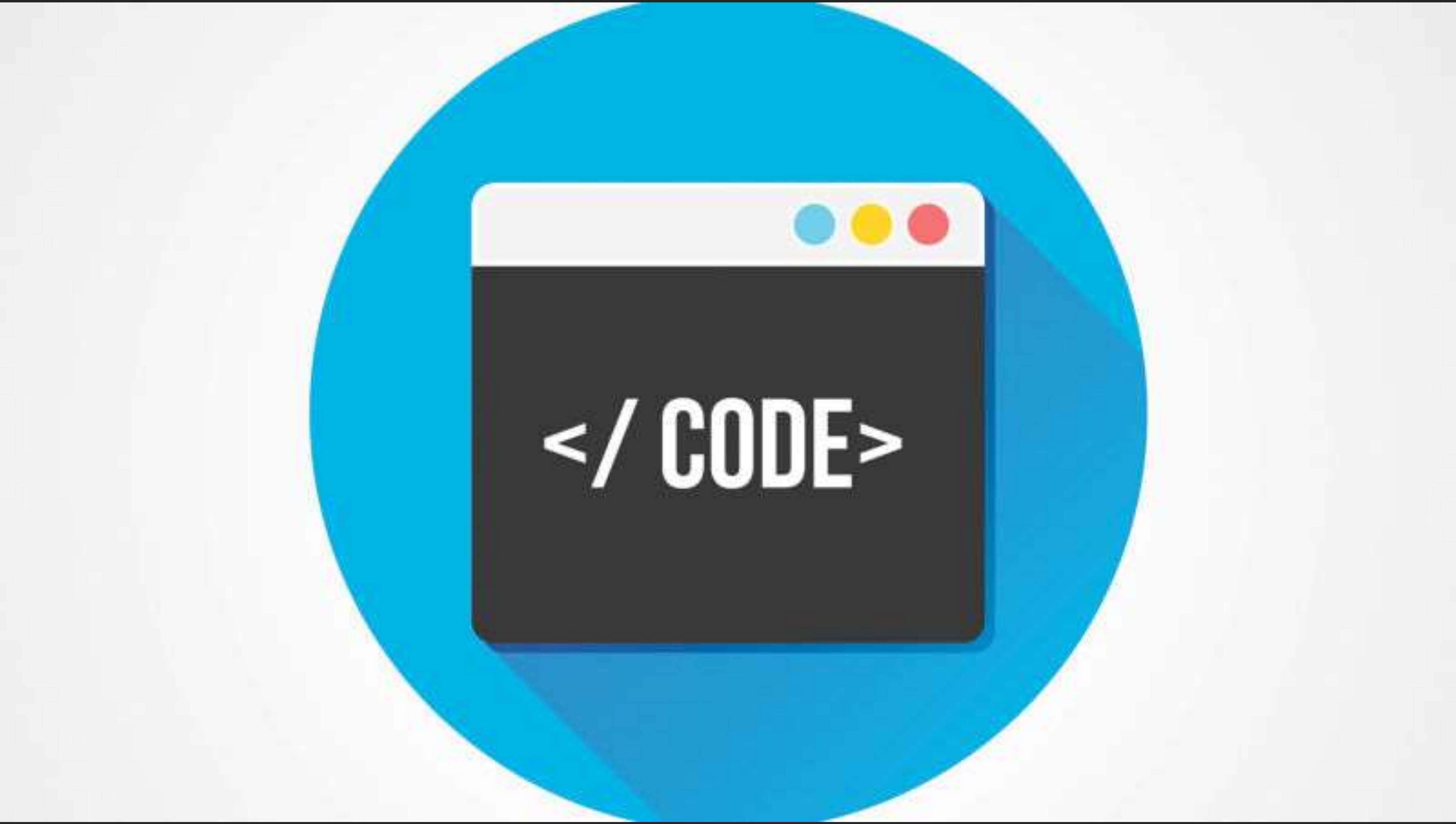
Lets Write Code

RAHUL

Intermediate Operations

Sorting: Using sorted to sort elements.

- In Java streams, the sorted method is used to sort elements. This method can sort elements in their natural order or according to a custom comparator.



Lets Write Code

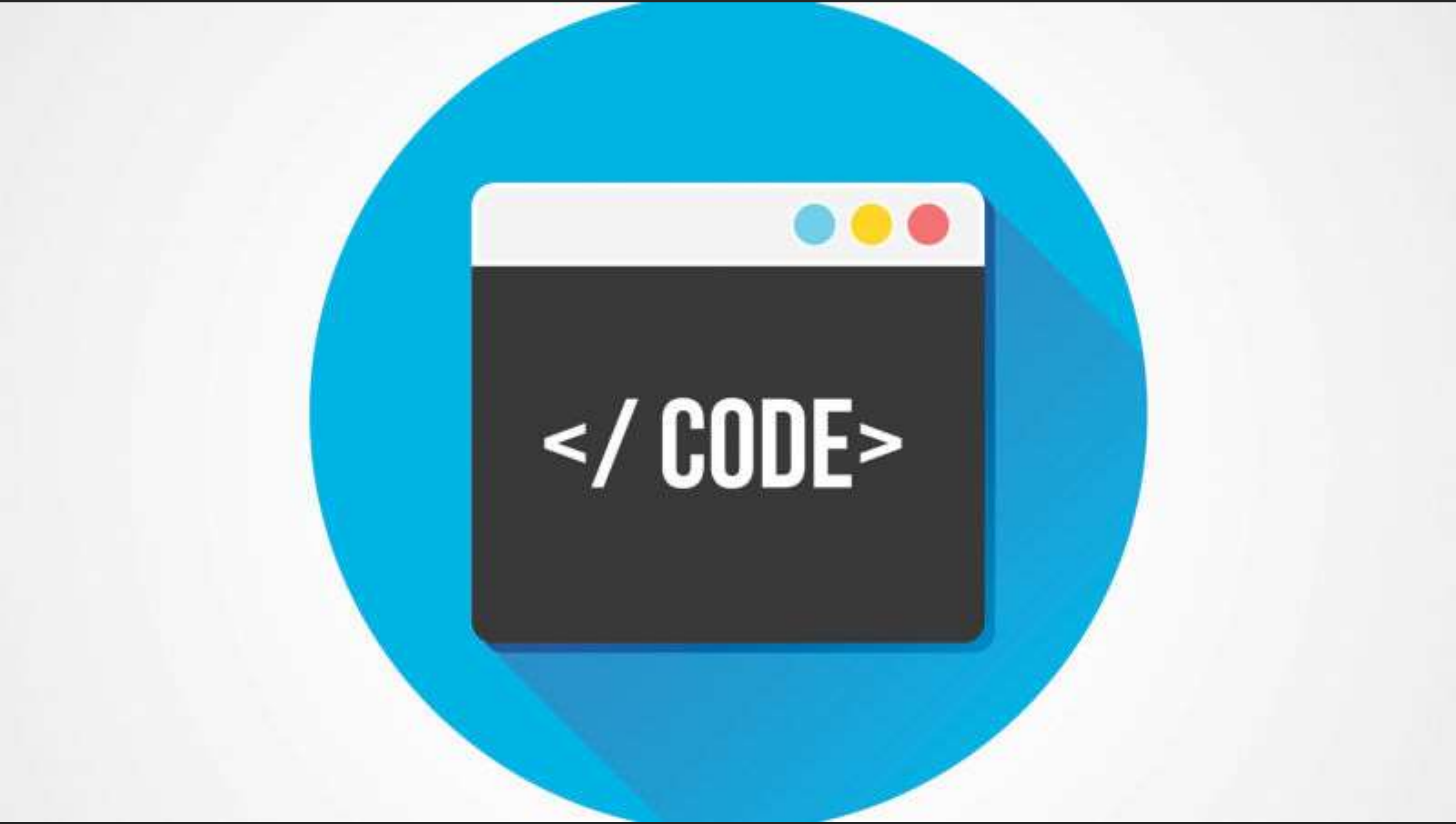
RAHUL

Intermediate Operations

Distinct: Removing duplicates with distinct.

- In Java 8, the distinct method of the Stream interface is used to remove duplicate elements from a stream.

RAHUL



Lets Write Code

RAHUL

Intermediate Operations

Limit and Skip:
Controlling the size of the stream with limit and skip

- In Java 8, the Stream API provides the limit and skip methods to control the size of the stream.
- The skip(n) method is an intermediate operation that discards the first n elements of a stream. The n parameter can't be negative, and if it's higher than the size of the stream, skip() returns an empty stream.
- The *limit(n)* method is another **intermediate operation that returns a stream not longer than the requested size**. As before, the *n* parameter can't be negative.

Terminal Operations

Collecting:
Using collect to gather elements into collections, strings, or other types.

- Collecting to List
- Collecting to Set
- Collecting to Map in key,value

- One Amazing Class : IntSummaryStatistics

RAHUL

Terminal Operations

Reducing:
Using reduce to combine elements into a single result.

- In Java 8, the **reduce** method is a terminal operation of the Stream interface that combines elements of the stream into a single result. This operation is also known as reduction.

- It is useful for performing **aggregation operations** such as summing numbers, concatenating strings, or finding the maximum or minimum value.

RAHUL

Terminal Operations

Counting: Using count to count elements

- In Java 8, the **count** method is a **terminal operation** of the Stream interface that returns the count of elements in the stream.

- This method is **useful when you need to know the number of elements that match certain criteria** or simply count the total elements in the stream.

RAHUL

Terminal Operations

Matching:
Using anyMatch, allMatch, and noneMatch to check conditions.

- In Java 8, the Stream interface provides several methods **to check if elements of a stream match given conditions.** The methods anyMatch, allMatch, and noneMatch are used for this purpose.

- **anyMatch** - Returns **true** if any elements of the stream match the provided predicate.
- **allMatch** - Returns **true** if all elements of the stream match the provided predicate.
- **noneMatch** - Returns **true** if no elements of the stream match the provided predicate.

RAHUL

Terminal Operations

Finding:
Using findFirst, MinBy,MaxBy and findAny to retrieve elements

- **findFirst:** Useful for obtaining the **first element** in a sequential stream.
- **findAny:** Efficient in parallel streams **for getting any element** without concern for order.
- **min and max:** Utilized to find the **smallest and largest elements** based on custom comparison logic.

RAHUL

Advance Stream Operations

Advance Stream Operations

Grouping:
Using Collectors.groupingBy to group elements

- In Java 8, the **Collectors.groupingBy** method is used to group elements of a stream based on a classifier function.

- This method is part of the Collectors utility class and provides a powerful way to group elements in a stream into a Map where the *keys are the result* of applying the classifier function, and the *values are Lists of items that match the key*.

Advance Stream Operations

Partitioning

- In Java 8, the ***Collectors.partitioningBy*** method is used to partition the elements of a stream into two groups based on a *predicate*.
- This method is part of the Collectors utility class and *returns a Map with Boolean keys*, where the value *true* corresponds to elements that satisfy the predicate, and *false* corresponds to elements that do not.

RAHUL

Advance Stream Operations

Joining

- In Java 8, the *Collectors.joining* method is used to concatenate strings from a stream into a single String. This method is part of the Collectors utility class and provides several overloads to specify *delimiters, prefixes, and suffixes*.

RAHUL

Parallel Streaming

Advance Stream Operations

Sequential Processing

- In sequential processing, the elements of a stream are processed one at a time in a ***single*** thread. This is the default mode of stream processing.

Advance Stream Operations

Parallel Processing

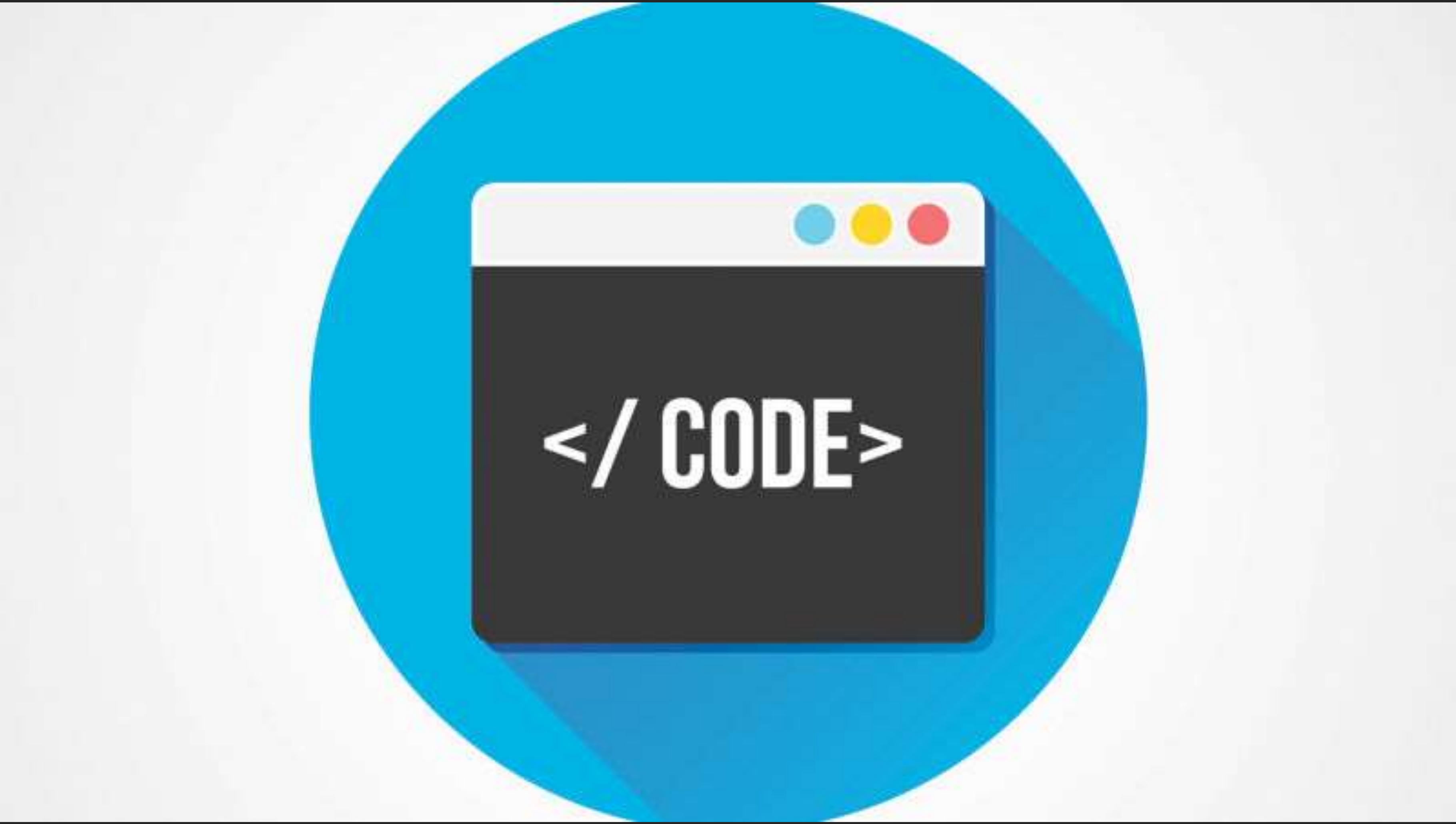
- In parallel processing, the stream is divided into *multiple segments*, and each segment is processed concurrently in different threads.
- This can lead to significant *performance improvements* for large data sets, especially on multi-core processors.

RAHUL

Advance Stream Operations

Comparision

Points	Sequential	Parallel
Performance	• Operations are performed one after the other in a single thread. • It can be slower for larger datasets or computationally intensive operations.	• Operations are divided and executed concurrently across multiple threads. • Speed up processing for large datasets
Order of Execution	The order of operations is guaranteed, maintaining the sequence of elements as they are processed.	The order of operations is not guaranteed, as elements may be processed in different threads and in different orders. Use forEachOrdered if you need to maintain order.
Overhead	Minimal overhead as only one thread is used	Higher overhead due to thread creation, management, and synchronization
Use Cases	Suitable for small datasets or simple operations	Suitable for large datasets or complex operations



Lets Write Code

RAHUL

Optional

Optional

What is Optional?

- **Optional** is a container object which *may or may not contain a non-null* value. It was introduced in Java 8 to avoid *null* references and provide a more expressive way to handle optional values.

- Use Cases:
  - To avoid *NullPointerException*.
  - To express that a *value might be absent* without using null.

RAHUL

Optional

Topic covered

- Creating Optional Instances
- Accessing Values in Optional
    - Use the get() method
- Handling Absence of Value
    - Using orElse, orElseGet, orElseThrow
- Transforming Values

RAHUL

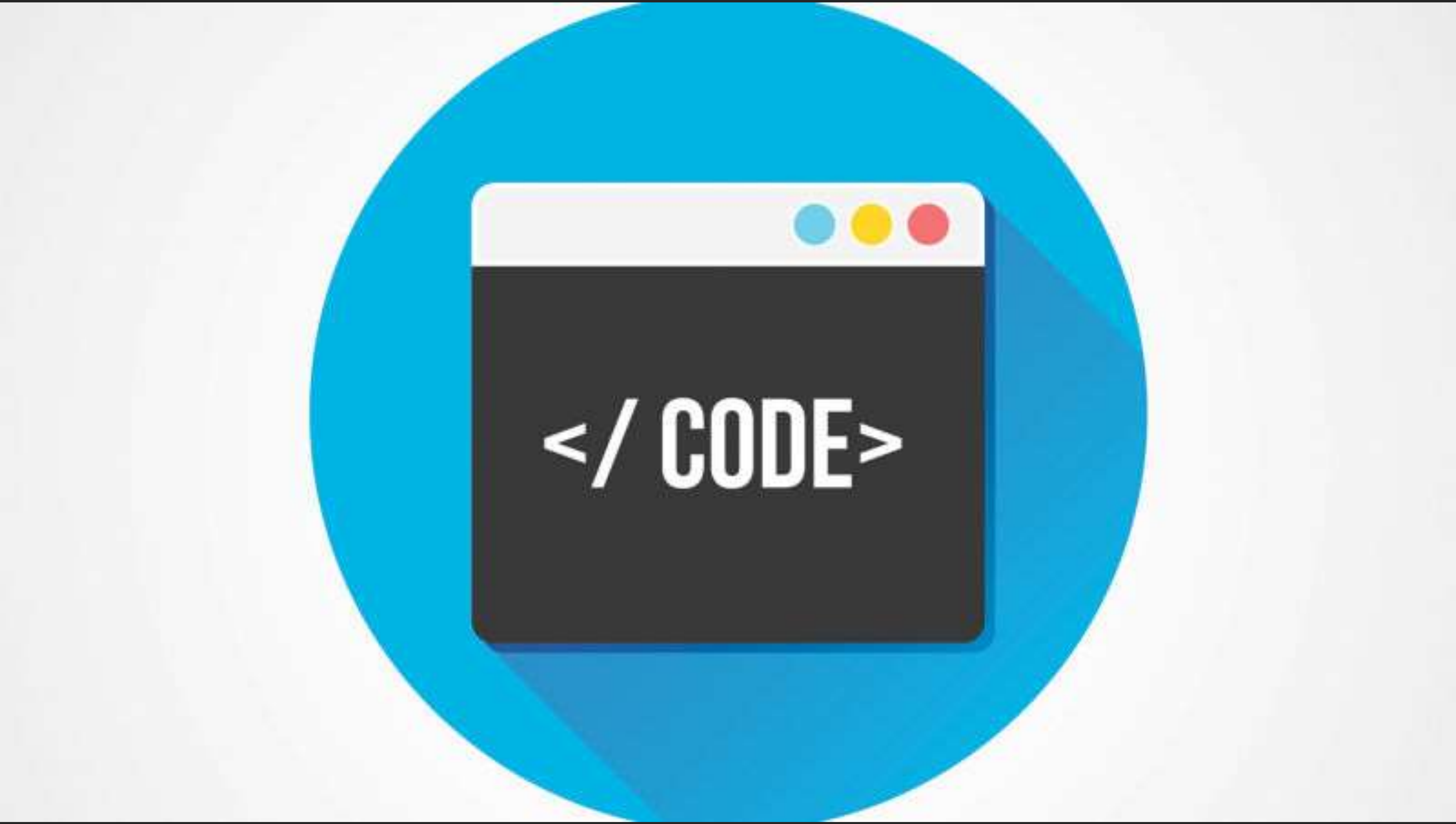
New Date Time API

New Date Time API

Key Classes in the New Date-Time API

- LocalDate: Represents a date *without time*. (e.g. 2024-07-19)
- LocalTime: Represents a time *without a date*. (e.g. 15:30:00)
- LocalDateTime: Represents a date and time *without a time zone*. (e.g. 2024-07-19T15:30:00)
- ZonedDateTime: Represents a date and time *with* a time zone. (e.g. 2024-07-19T15:30:00+01:00[Asia/Kolkata]
- Instant: Represents a specific moment in time (*timestamp*). (e.g. 2024-07-19T14:30:00Z)
- Duration: Represents a *time-based* amount of time (e.g., days, hours). (e.g. 2 hours, 30 minutes)
- Period: Represents a *date-based* amount of time (e.g., years, months). (e.g. 1 year, 2 months)

RAHUL



Lets Write Code

RAHUL

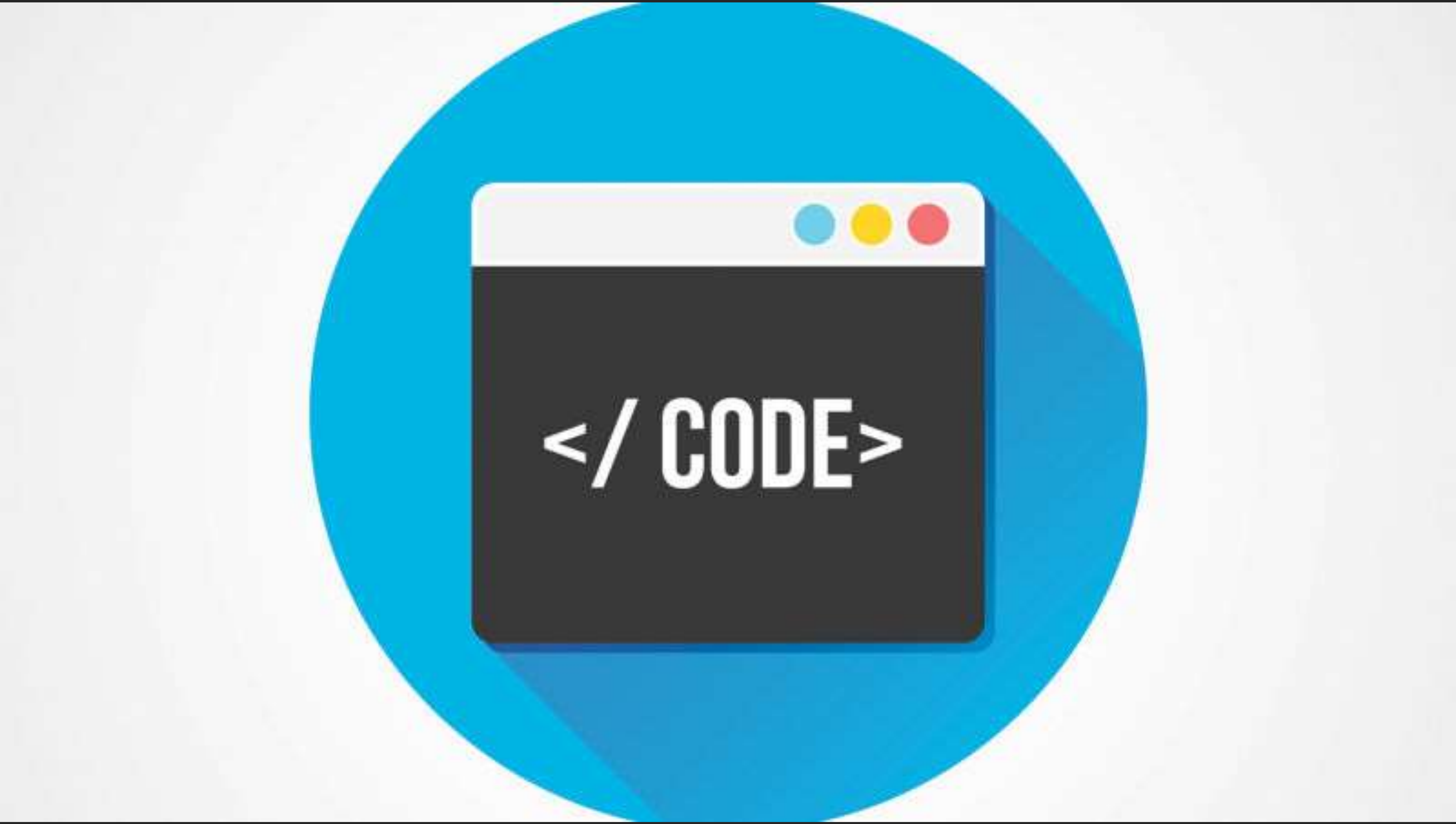
Error Handling in Streams

Error Handling in Streams

Handling exceptions in Java 8 streams

- Wrapper Function: Wrap functional interfaces to *handle exceptions and rethrow* them as unchecked exceptions.
- Using Optional: Handle exceptions gracefully by using Optional to filter out erroneous values.
- Collecting Errors Separately: Collect errors in a separate list for later processing or logging.
- Try-Catch Inside Stream Operations: Directly handle exceptions within the stream operations using try-catch blocks.

RAHUL



Lets Write Code

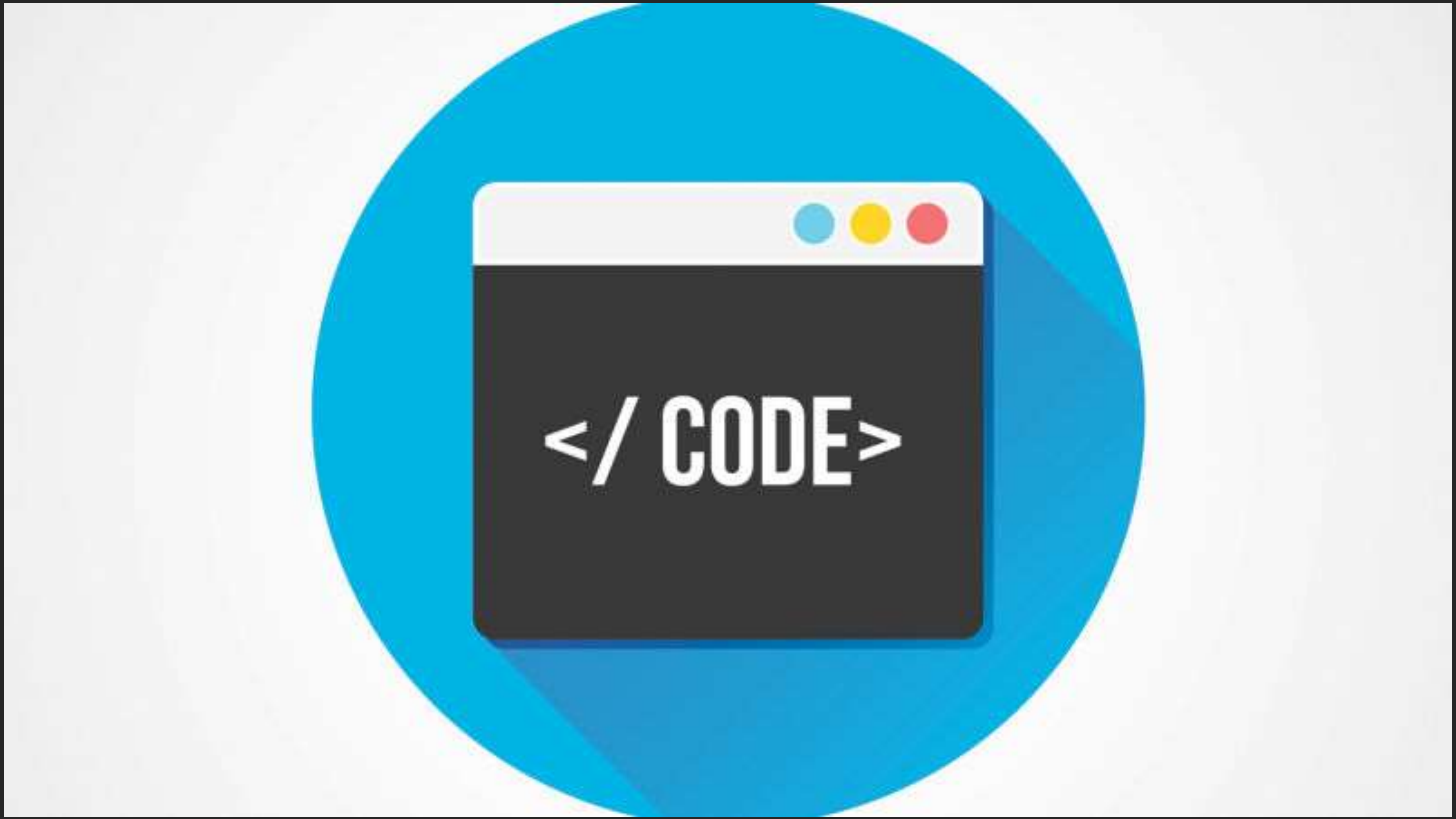
RAHUL

Performance Considerations

Performance Considerations

- Sequential vs Parallel Streams: Use parallel streams for CPU-bound operations with large datasets, but measure performance as overhead may negate benefits.
- Avoid Unnecessary Operations: Minimize *intermediate operations* to reduce overhead.
- Use toArray for Simple Conversions: Prefer *toArray* over collect for converting to arrays.
- Use Primitive Streams: Prefer *IntStream*, *LongStream*, and *DoubleStream* to avoid boxing and unboxing.
- Minimize Terminal Operations: Avoid *multiple terminal operations* on the same stream to prevent re-evaluation.

RAHUL



Lets Write Code

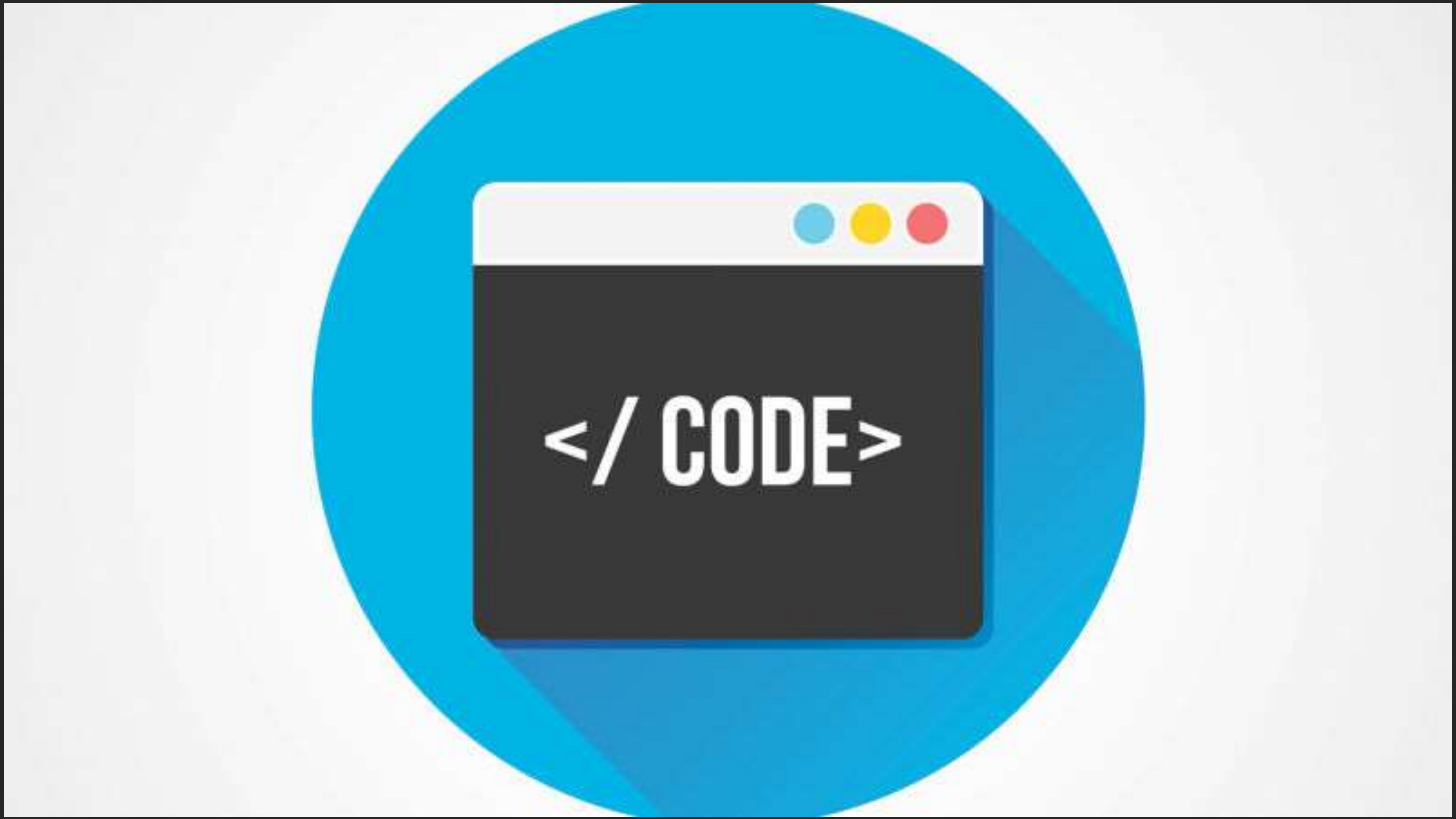
RAHUL

Common Pitfalls and Best Practices

Common Pitfalls

- Overusing Parallel Streams: Measure performance before using.
- Ignoring Exception Handling in Streams: Use wrapper functions or Optional.
- Using Stateful Operations: Avoid external mutable state.
- Forgetting to Close Resources: Use try-with-resources.
- Using Streams for Simple Iterations: Use traditional loops for simplicity.

RAHUL

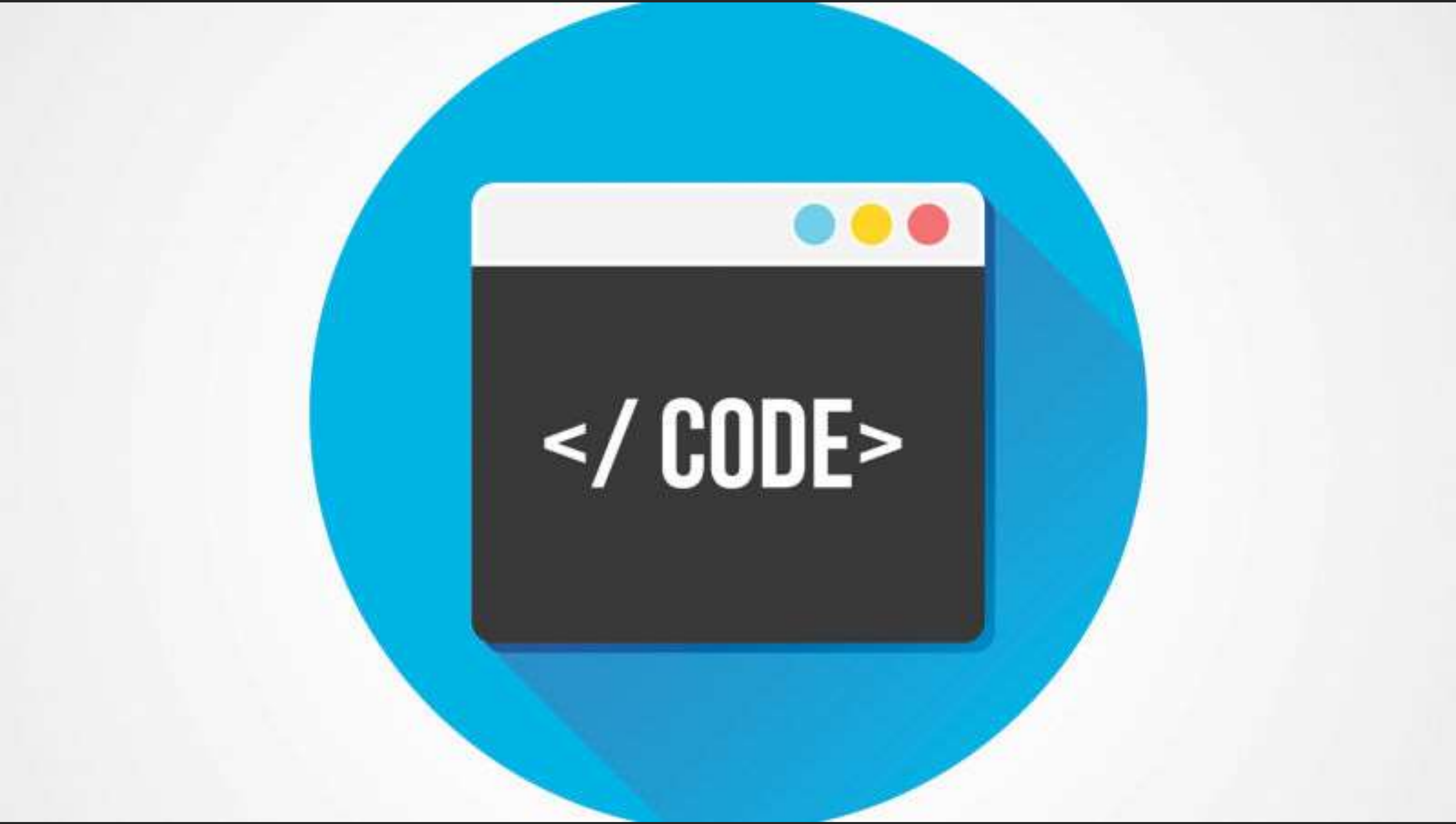


Lets Write Code

RAHUL

Best Practices

- Prefer Method References Over Lambda Expressions
- Use Collectors Efficiently
- Leverage Primitive Streams
- Optimize Performance with Parallel Streams
- Handle Exceptions Gracefully
- Avoid Modifying the Source of the Stream
- Use Optional Effectively
- Consider Readability and Maintainability



Lets Write Code

RAHUL