Java Programming: A Comprehensive Overview

This one offers a comprehensive overview of Java, covering everything from foundational programming concepts to advanced topics. It is ideal for new programmers and students seeking to master Java's powerful capabilities.

Basic Programming Constructs

1 Variables and Data Types

Understand Java's fundamental data types like int, float, boolean, and char.

2 Operators

Learn arithmetic, logical, assignment, and bitwise operators for computations.

3 Control Flow

Master if-else statements and switch cases for decision-making logic.

4 Loops

Utilize for, while, and do-while loops for repetitive tasks. For example, calculate factorials.

Object-Oriented Programming (OOP)

Class and Object

Differentiate between a class (blueprint) and an object (instance).

Inheritance

Extend class functionality by creating subclasses.

Polymorphism

Explore method overloading and overriding for diverse behaviors.

Encapsulation

Implement data hiding and control access to class members.

Abstraction

Simplify complex systems by focusing on essential features.

Example: Design a Vehicle class with Car and Motorcycle subclasses.

Packages, Strings, and Exception Handling

Packages

Organize classes efficiently using namespaces.

Strings

Understand string immutability and common operations.

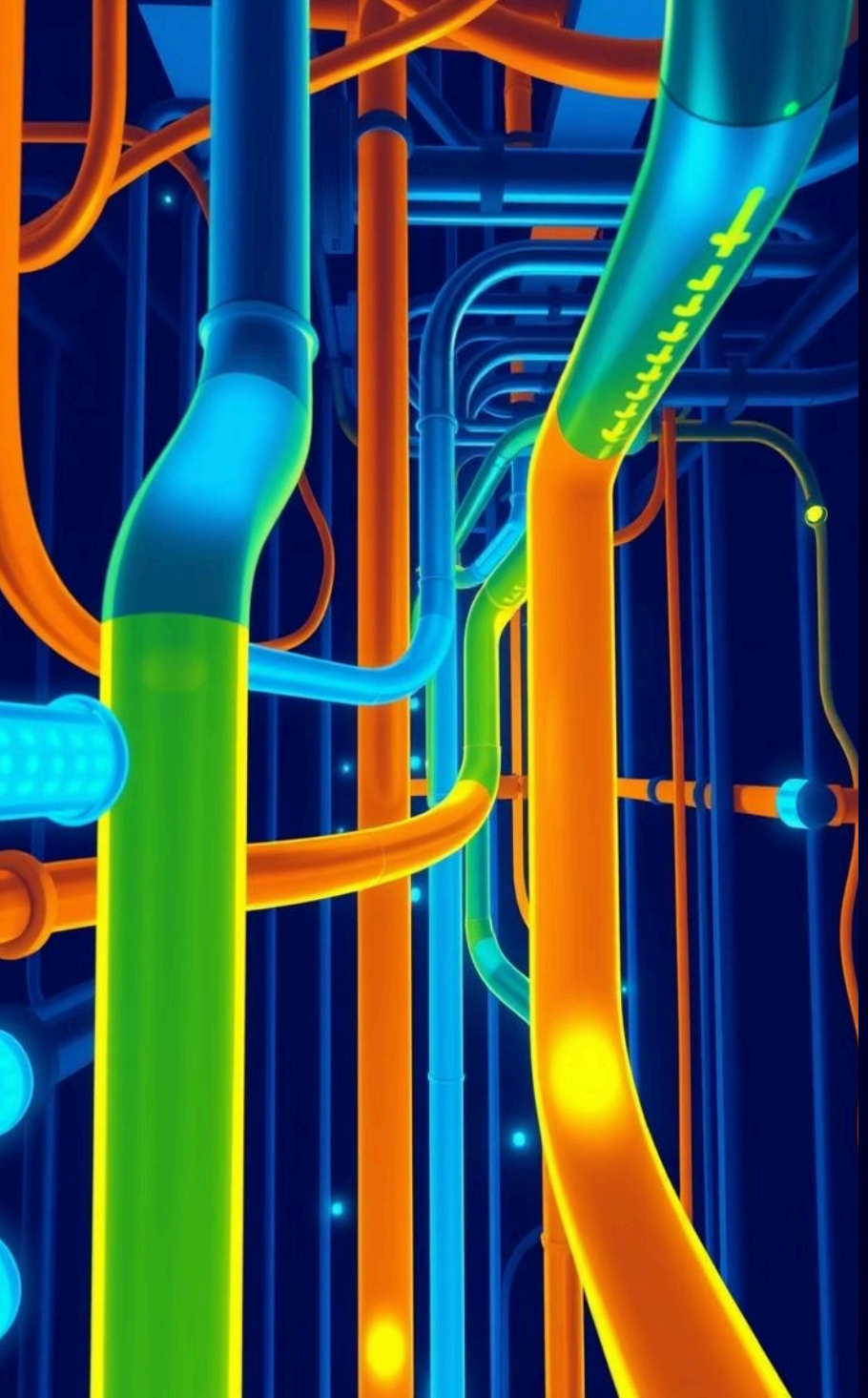
Exception Handling

Implement try-catch blocks and use the throw keyword.

Checked vs Unchecked

Distinguish between checked and unchecked exceptions.

Example: Create a custom exception for invalid input data to ensure robustness.



Input/Output (I/O) Streams

Byte Streams

Utilize InputStream and OutputStream for raw byte data.

Character Streams

Work with Reader and Writer for character-based data.

File I/O

Learn to read from and write data to various files.

Buffered Streams

Improve I/O performance significantly with buffering.

Example: Develop code to efficiently read and write data to a text file and perform some business

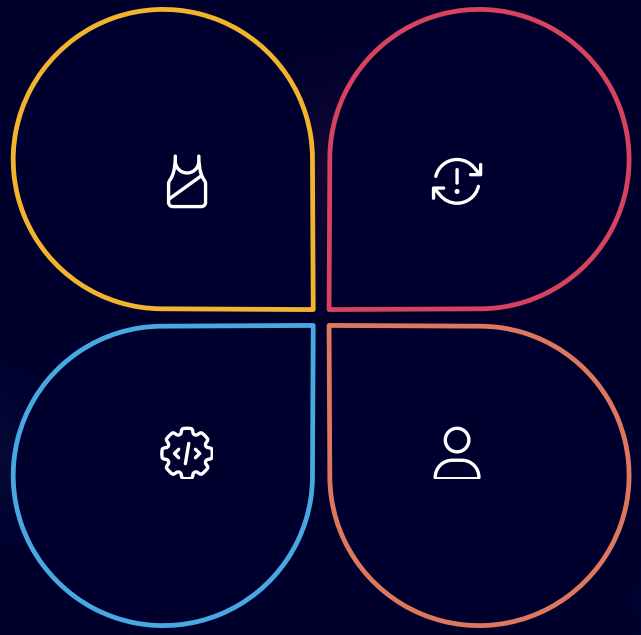
Multithreading

Threads

Create and effectively manage multiple threads for concurrent execution.

Concurrency

Achieve true parallelism in Java applications.



Synchronization

Prevent race conditions using synchronization techniques.

Thread Pools

Manage threads efficiently with pre-initialized thread pools (ExecutorService - java.util.concurrent)

Example: Implement multiple threads to process large datasets concurrently.

Collection Framework

Lists	ArrayList, LinkedList	Ordered collections for sequences
Sets	HashSet, TreeSet	Unique element storage
Maps	HashMap, TreeMap	Key-value pair storage
Iterators	Standard way to traverse	Efficiently navigate collections

Example: Store and retrieve student records efficiently using a HashMap.

Reflection API

Examine Structure

Inspect class structure, methods, and fields at runtime.

Instantiate Dynamically

Create objects dynamically without knowing their class at compile time.

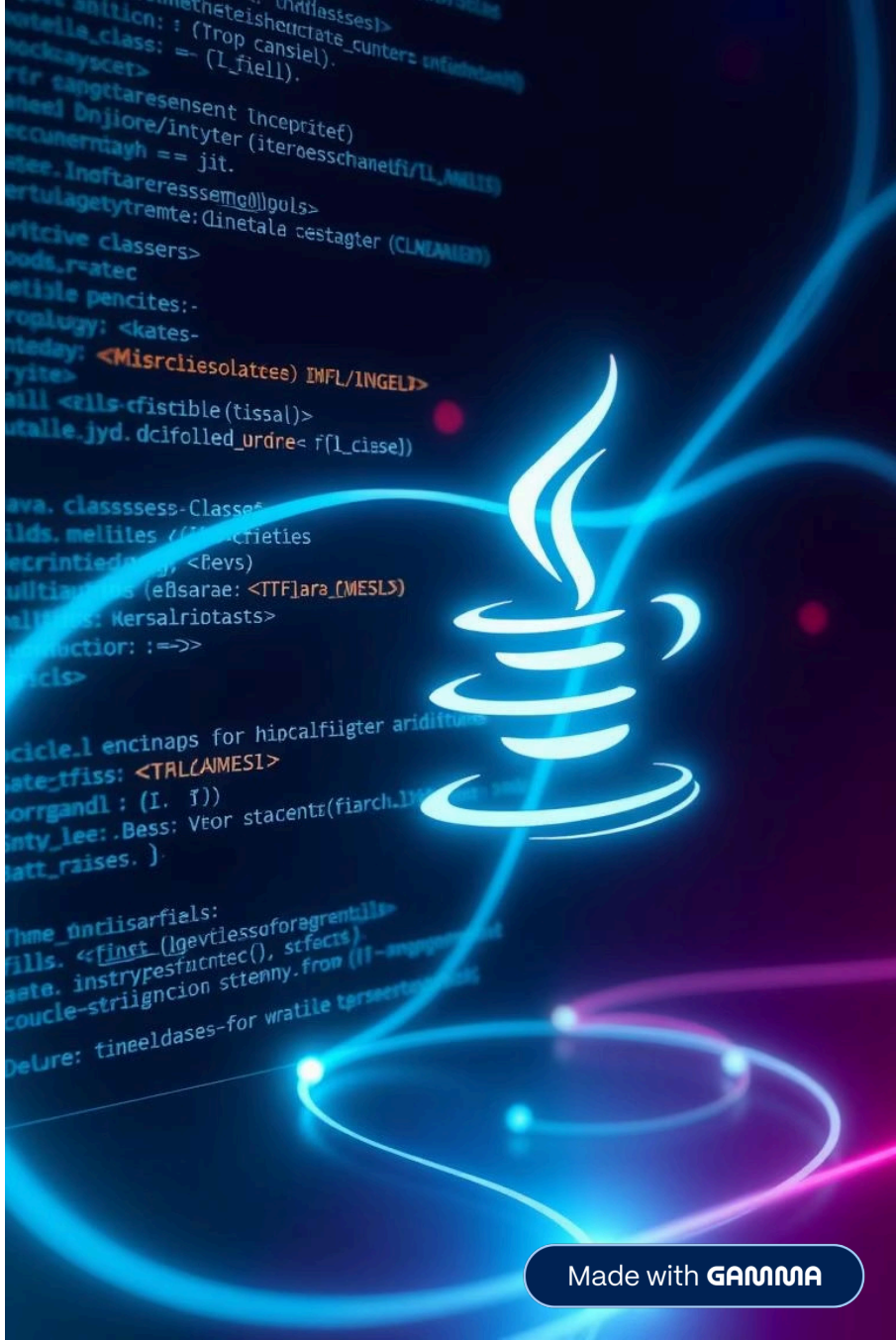
Invoke Methods

Access and invoke methods and fields programmatically.

Use cases: Powerful for frameworks, dependency injection, and advanced testing. Example: Use reflection to inspect a class and its methods.

Unveiling Java Reflection API

It explores the Java Reflection API. Learn how to analyze and manipulate code dynamically at runtime. Discover its power for advanced use cases.



Introduction to Reflection

Runtime Inspection

Reflection enables dynamic analysis of Java components. It allows programs to examine classes, methods, and fields at runtime.

Frameworks & Testing

It's crucial for frameworks, like Spring, and for testing tools. Serialization libraries also heavily rely on reflection for object conversion.

Core Package

The `java.lang.reflect` package contains core components. This package provides all necessary classes for reflective operations.

The getClass() Method

Object Instance

The simplest way uses an object instance.

```
MyObject obj = new MyObject();
```

```
Class<?> clazz = obj.getClass();
```

.class Literal

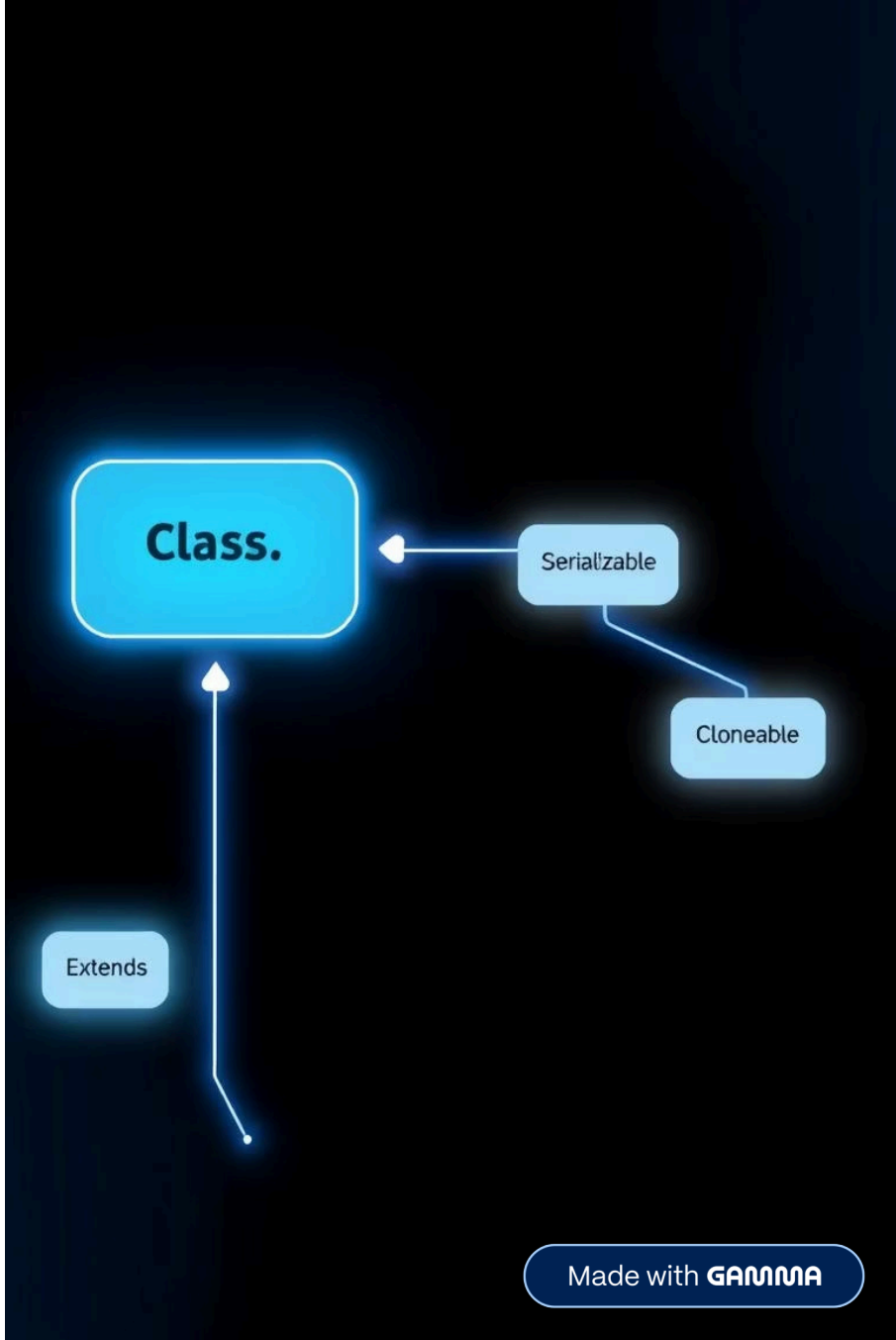
For known types, use the `.class` literal. Example:

```
Class<MyObject> clazz = MyObject.class;
```

Class.forName()

Dynamically load classes with `Class.forName()`. Example:

```
Class.forName("com.example.MyClass");
```

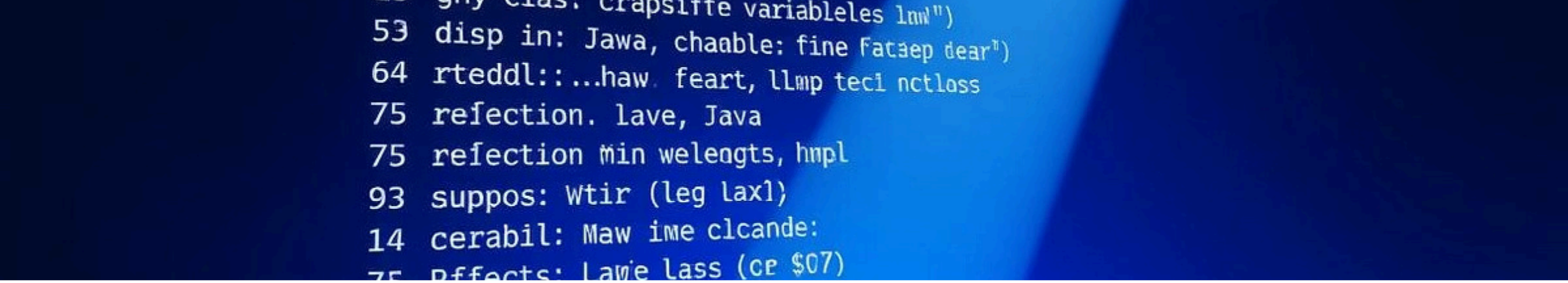


Exploring Method Objects

Retrieving Methods

- `getMethods()`: Public methods.
- `getDeclaredMethods()`: All methods.
- `getMethod(name, ...)`: Specific public method.
- `getDeclaredMethod(name, ...)`: Specific any method.

Method objects represent class methods and constructors. You can obtain them using various retrieval methods. `getDeclaredMethods()` is key for private methods.



Working with Field Objects

getFields()	Returns all public fields.	clazz.getFields();
getDeclaredFields()	Returns all fields (public, private, protected).	clazz.getDeclaredFields();
getField(name)	Returns a specific public field.	clazz.getField("id");
getDeclaredField(name)	Returns a specific field (any access).	clazz.getDeclaredField("name");

Field objects represent class fields. Use getDeclaredField() to access private fields directly. This method bypasses typical access modifiers.

Accessing Private Data

Bypassing Access Control

Reflection allows access to private fields and methods. Use `setAccessible(true)` to override standard Java security checks.

Manipulating Private Fields

Once accessible, `Field` objects enable direct read and write operations. This bypasses encapsulation for instance variables.

Invoking Private Methods

Similarly, `Method` objects can invoke private methods. This allows execution of internal logic not exposed publicly.

Accessing Private Data

Bypass Restrictions

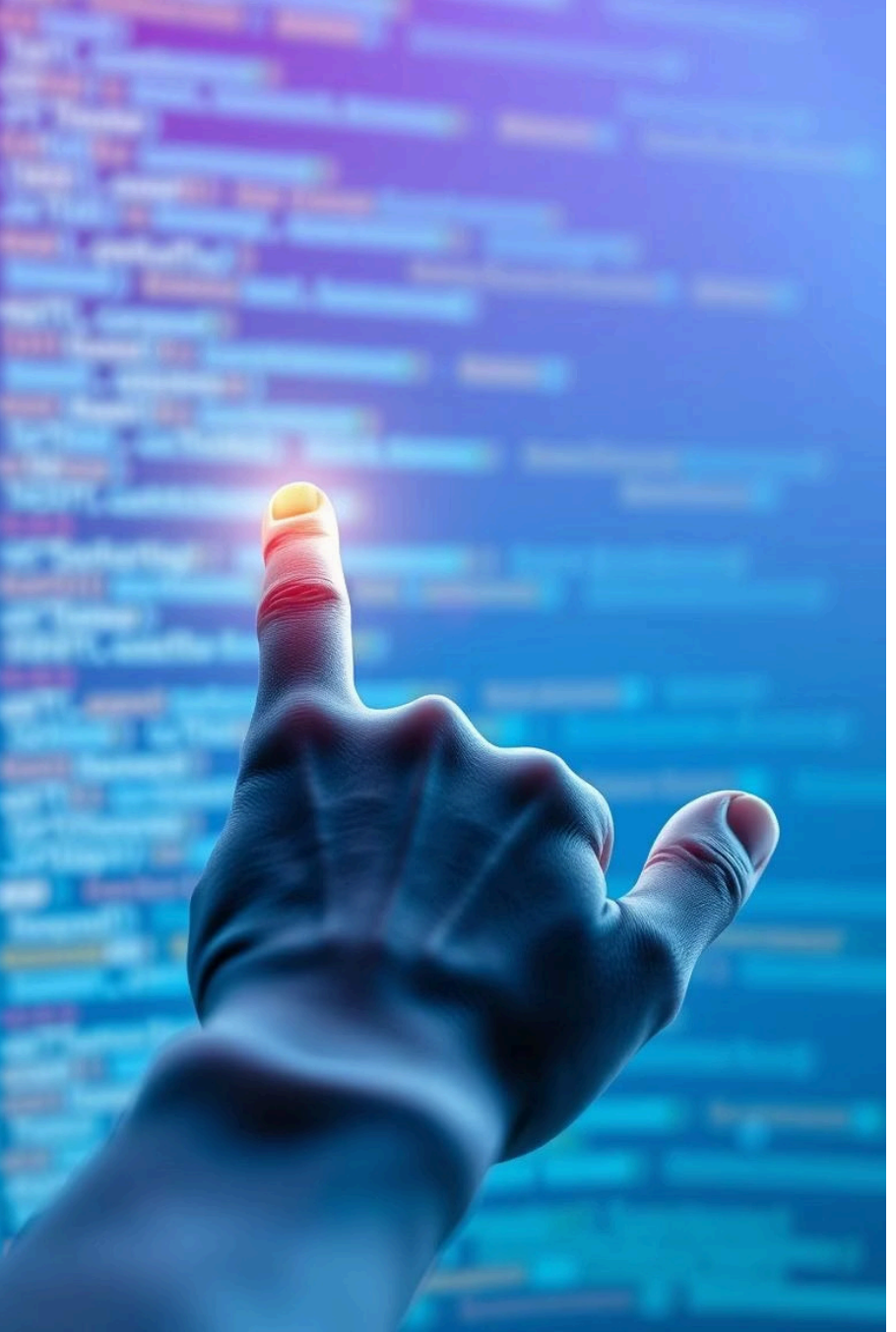
Reflection allows bypassing Java's access restrictions. This grants powerful control over private members.

Set Accessible

Call `setAccessible(true)` on the `Field` or `Method` object. This enables access to private members.

Use Judiciously

While powerful, this breaks encapsulation. Use reflection cautiously, as it can lead to fragile code.



Invoking Methods Dynamically

Get the Method

First, obtain a `Method` object. Use `getMethod()` or `getDeclaredMethod()`.

Prepare Arguments

Gather necessary arguments for the method. Pass them as an array or varargs.

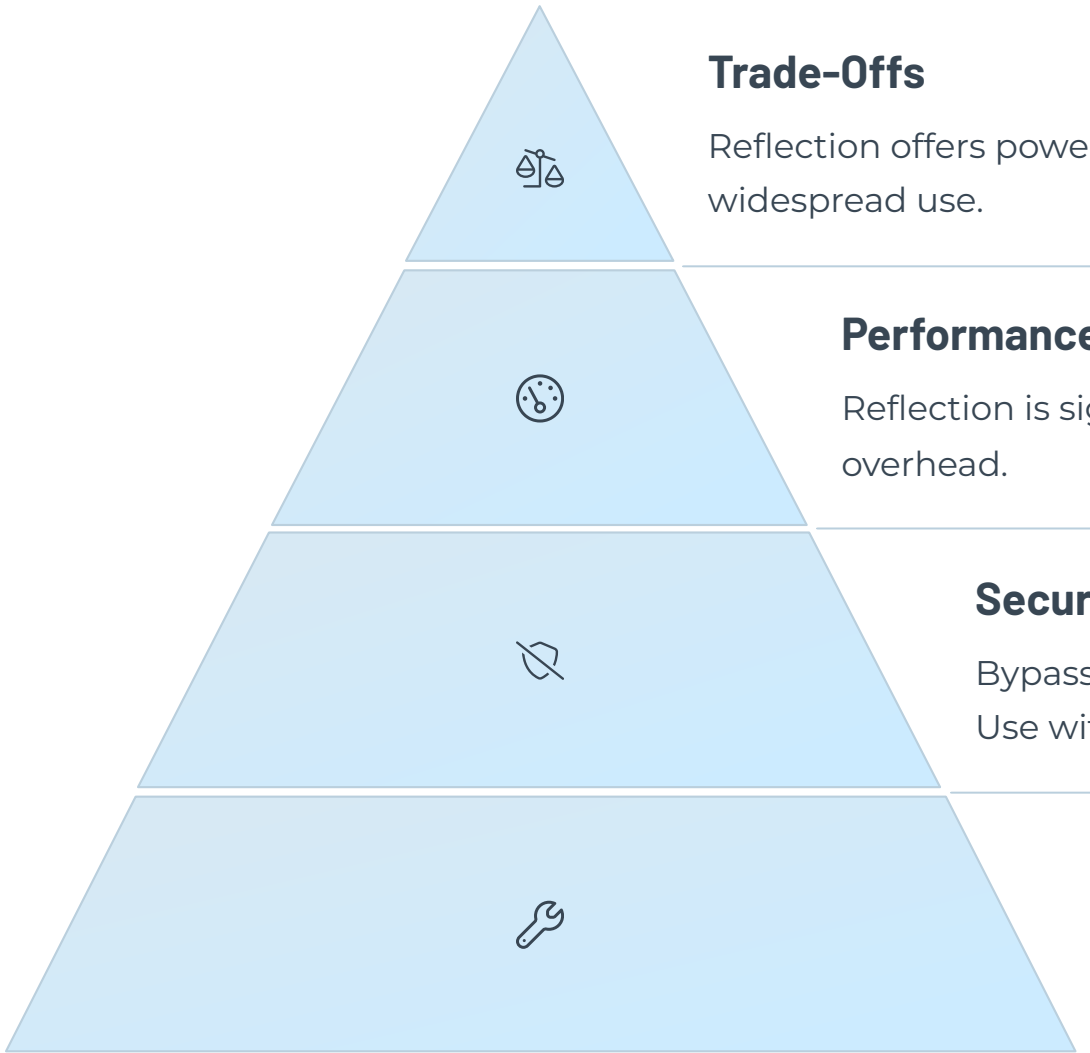
Invoke Method

Call `method.invoke(object, args...)`. This executes the method at runtime.

Handle Exceptions

Be prepared for `InvocationTargetException` and `IllegalAccessException`.

Reflection API - Use with Caution



Trade-Offs

Reflection offers power but comes with costs. Understand its implications before widespread use.

Performance Impact

Reflection is significantly slower than direct calls. Expect a performance overhead.

Security Risks

Bypassing access controls can introduce security vulnerabilities. Use with extreme care.

Maintainability

Code using reflection is harder to read and debug. It can increase complexity and reduce clarity.

Stream API
and
Lambda Expression

Basics Revisited

- Interfaces
- Collections Framework
- Anonymous Inner Classes

Interfaces

"Ordinary" interfaces

Marker interfaces (e.g. `Serializable` or `Runnable`)

Functional interfaces (annotated with `@FunctionalInterface`)

`@interface` classes (Annotations)

New in Java 8:

`static` methods

`default` methods

Default methods[1]

"A default method is a method that is declared in an interface with the `default` modifier; its body is always represented by a block. It provides a default implementation for any class that implements the interface without overriding the method. Default methods are distinct from concrete methods (§8.4.3.1), which are declared in classes."

[1][Gosling 2015, p. 288]

Collections Framework

Two things to remember:

1. There are Lists, Sets, and Maps:

```
List<T>:        ArrayList<T>  or  LinkedList<T>
Set<T>:         TreeSet<T>  or  HashSet<T>
Map<K, V>:      TreeMap<K, V>  or  HashMap<K, V>
```

2. Use “loosly coupled” references:

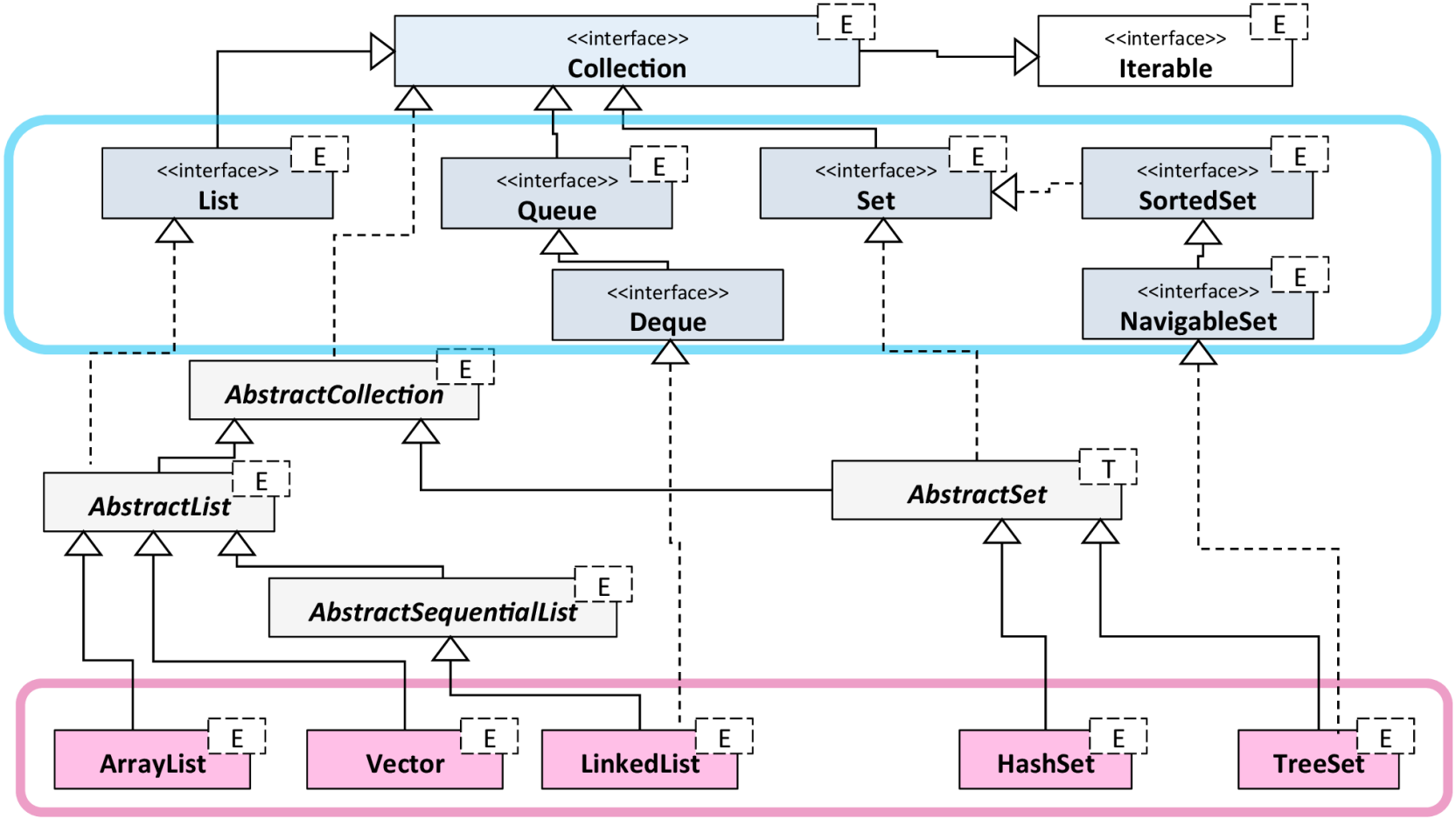
```
List<String> = new ArrayList<>();
```

Collections Framework[1]

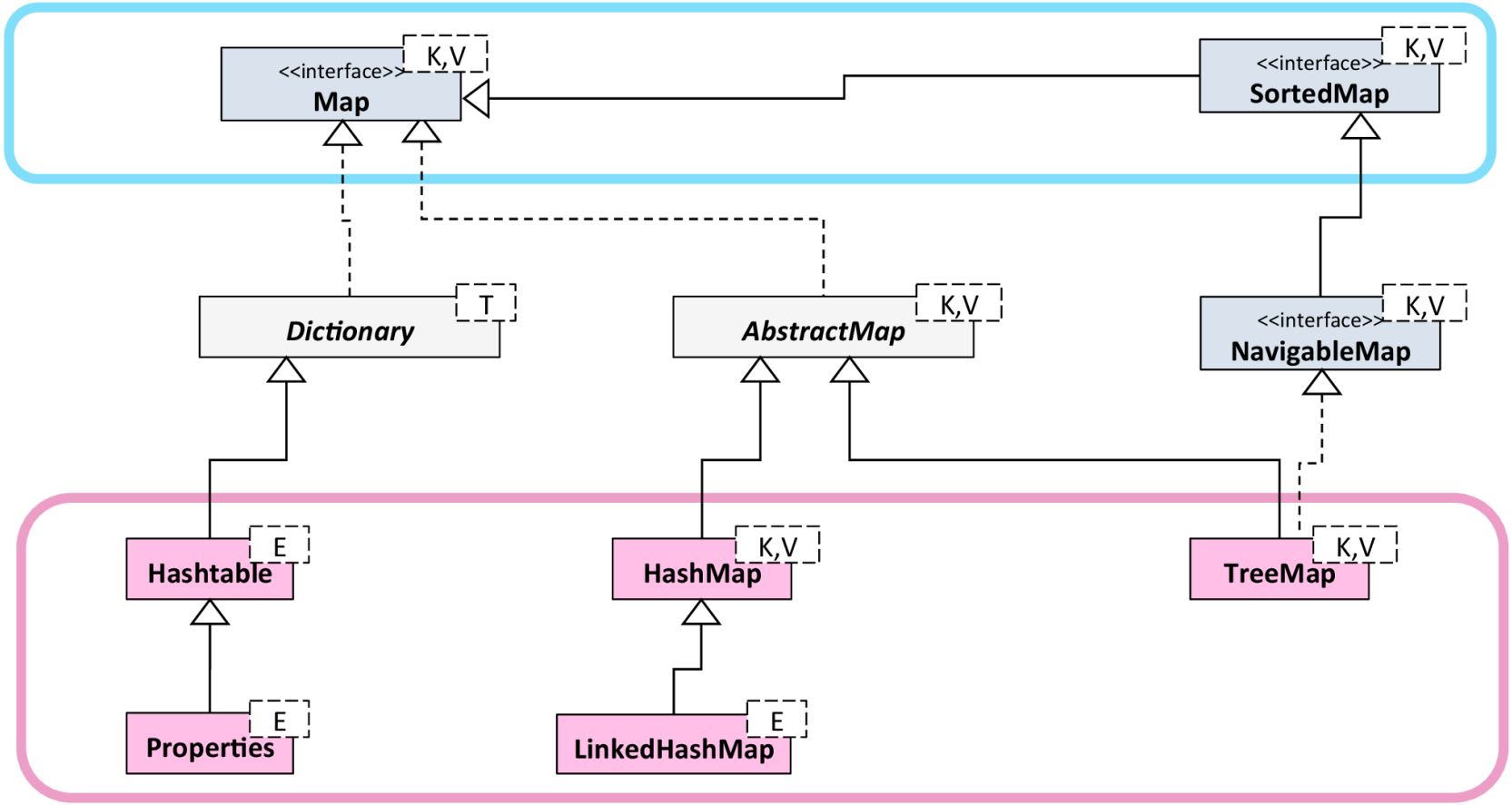
“The collections framework is a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. […]”

[1][Oracle Corp. 2016]

Collection interfaces (abridged)



Map interfaces (abridged)



Utilities in class Collections

```
public static void reverse(List<?> list)

public static <E> Collection<E> checkedCollection(Collection<E> c,
                                                  Class<E> type)

public static <T> List<T> nCopies(int n, T o)

public static int frequency(Collection<?> c, Object o)

public static void shuffle(List<?> list)

public static void rotate(List<?> list, int distance)

public static void reverse(List<?> list)

public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)

public static <T> T min(Collection<? extends T> coll,
                        Comparator<? super T> comp)

// others:

Arrays.asList(Object... o)

Arrays.stream(T[] array)

Stream.of(T[] array)
```

Anonymous Inner Classes

```
public interface Comparator<T> { int compare (T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names, new Comparator<String>() {
   @Override public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
});

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Streams

Java Streams: What are they (not)?

- Not related to Java IO Streams!
  - (FileInputStream, InputStreamReader, etc. -- none of those guys)

- A basic way to do *functional* programming in Java

Streams

- are not data structure
- do not contain storage for data
- are “pipelines” for streams of data (i.e. of objects)
- while in the pipeline data undergo transformation (without changing the data structure holding it)
- wrap collections (lists, set, maps)
- read data from it
- work on copies

What is this... “functional” programming?

- Basic idea: issue **methods** as arguments to other methods,
  - there, execute them with local data as arguments

But what is a “Stream”?

- Think “Streaming Collection of Elements”
- Can have different sources
  - Java Collections
  - Arrays
  - A sequence of individual objects
- A sequence of operations can be applied
- Results not available until “terminal” operation

How do data get into streams?

Streams are mainly generated based on collections:

```
List<String> names = Arrays.asList("John", "George", "Sue");


Stream<String> stream1 = names.stream();

Stream<String> stream2 = Stream.of("Tom", "Rita", "Mae");

Stream<String> stream3;
stream2 = Arrays.stream( new String[]{"Lisa", "Max", "Anna"} );
```

Or with builder pattern:

```
Stream<String> stream4 = Stream.<String>builder()
                              .add("Mike")
                              .add("Sandra").build();
```

How do data get out of streams?

- The Streaming API provides so called “finalizing” methods (i.e. methods that do not return stream objects)

```
forEach
toArray
collect
reduce
min
max
count
anyMatch
noneMatch
findFirst
findAny
```

Streaming example

"Take all names from the stream that start with the letter "J", map the names into capital letters, skip one, and collect them into a new set"

```
List<String> names = Stream.of("John","George","Joe","Sue","James");


Stream<String> stream1 = names.stream();

______ = stream1.filter( ___________ )
                .map( ___________ )
                .skip( ___________ )
                .collect( ___________ );
```

```
Set<String> result = Stream.of("John", "George", "Joe", "Sue", "James")
        .filter(name -> name.startsWith("J"))
        // Filter names starting with "J"
        .map(String::toUpperCase) // Convert names to uppercase
        .skip(1) // Skip one name
        .collect(Collectors.toSet()); // Collect into a Set


System.out.println(result);
```

Lambda Expressions and Functional Interfaces

- Lambdas
- Functional Interfaces

Lambdas or Closures

"Lambda" = "closure" = record storing a function (functionality, method) and its environment (but without a class or method name)

Roughly: anonymous method

Lambdas represent source code – not data and not object state!

Syntax:

```
( parameter list ) -> { expression(s) }
```

Examples:

```
(int x, int y) -> { return x + y; }
```

```
(long x) -> { return x * 2; }
```

```
() -> { String msg = "Lambda"; System.out.println(msg); }
```

For details on "functional programming" cf. [Huges 1984] or [Turner 2013]

Lambdas and functional interfaces

Functional interfaces are so called *SAM* types (single abstract method)

A functional interface has exactly one abstract method, e.g. `Runnable`, `Comparator<T>`, or `Comparable<T>`

Functional interfaces can define 0..* default methods and 0..* static methods

Using the `@FunctionalInterface` annotation on classes the compiler is required to generate an error message if it is no interface and it doesn't define exactly one SAM.

```
@FunctionalInterface
public interface Counter<T> {
    int count(T obj);
}
```

```
Counter<String> strCount = (String s) -> { return s.length(); };
```

[https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html]

Your first lambda with Comparator<T>

```
@FunctionalInterface
public interface Comparator<T> { int compare(T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

Your task:
Prepare your first lambda expression
to compare two String objects by length

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names,   ????   );

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Functional interfaces in JDK

Selection of most used interfaces from package `java.util.function`:

```
// Computes a single input with no result
public interface Consumer<T>  {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {...}
}

// Represents a supplier of results.
interface Supplier<T> {
   T get();
}

// Computes a single output, prodeces output of different type
interface Function<T,R>  {
   R apply(T t);
   default <V> Function<T,V> andThen(Function<? super R,? extends V> after)  {...}
   default <V> Function<V,R> compose(Function<? super V,? extends T> before) {...}
}

// Represents a predicate (boolean-valued function) of one argument
interface Predicate<T> {
   boolean test(T t);
   default Predicate<T> and(Predicate<? super T> other) {...}
   default Predicate<T> negate() {...}
   // ...
}
```

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>

[https://docs.oracle.com/javase/8/docs/api/java/util/function/package-amountmary.html]

Type inference

Lambda expressions allow for minimal syntax if compiler can deduct type information (so called *type inference*), e.g.:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

R = Integer could be inferred from return type of expression t.length() + u.length() → Integer (e.g. when used directly in typed method call)

```
BiFunction<String, Object, Integer> bf;
bf = (String txt, Object obj) -> { return t.length() + u.hashCode() ); }
// can be even shorter:
bf = (txt, obj) -> t.length() + u.hashCode(); // se below
```

compiler can infer types from declaration of f

Further syntax shortening examples:

for single return statements keyword return together with {}-pair can be dropped

```
(int x, int y) -> { return x * y; }  // shorter:   (x, y) -> x * y
(long x) -> { return x * 2; }        // shortest:  x -> x * 2
```

()-pair can be dropped with only one parameter

Lambda as parameters and return types

Further examples for type inference using `Comparator<T>` as functional interface:

```
List<String> names = Arrays.asList("Ahab", "George", "Sue");
Collections.sort(names, (s1, s2) -> s1.length() - s2.length() );
```

*T :: **String** can be deducted from names being a List<**String**>*

Signature of Collections::sort method is:
*public static <**T**> void sort(List<**T**> list, Comparator<? super **T**> c)*

```
Collections.sort(names, createComparator() );
```

```
public Comparator<String> createComparator() {
    return (s1, s2) -> s1.length() - s2.length();
}
```

Statement returned here is of functional interface type Comparator<String>

Method references (“function pointers”)

Syntax: `Classname::methodName` `objectReferenceName::methodName`

Lambdas can be replaced by method references whenever there would not further actions within the the lambda

Examples:

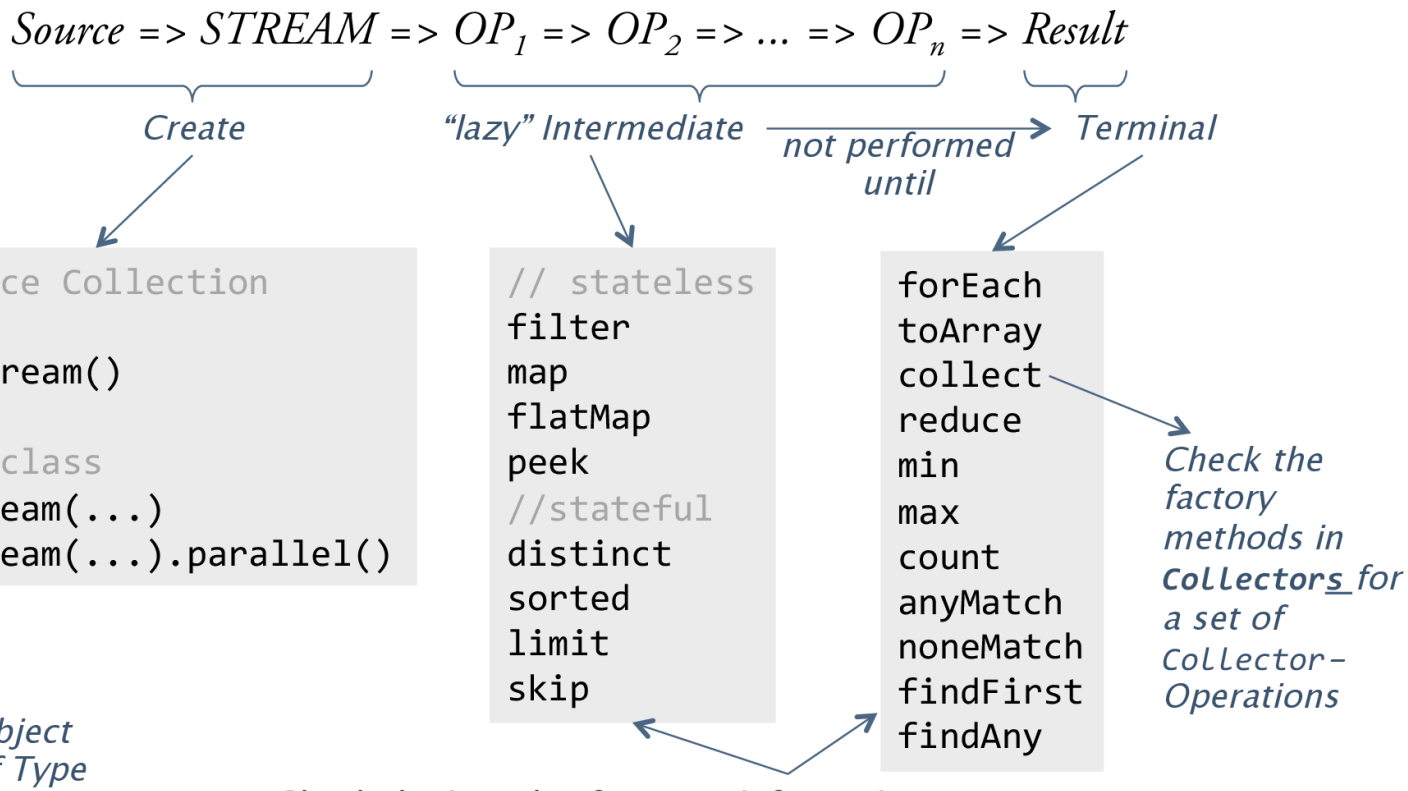
Reference	Method reference…	…replacing Lambda
static method	`String::valueOf`	`obj -> String.valueOf(obj)`
instance method (via class)	`String::compareTo`	`(s1, s2) -> s1.comapreTo(s2)`
instance method (via object ref)	`person::getName`	`() -> person.getName()`
Constructor	`ArrayList::new`	`() -> new ArrayList<>()`

[Table taken from Inden 2015, p. 812]

Streaming API

- Creating Streams
- Fluently working with streams
- Finalize Streams

Stream operations



[Inden 2015, p. 825]

How to make streams?

- Import Stream-related things from java.util.stream
  - `import java.util.stream.*` imports everything related
- Method 1: build from a static array or individual objects using Stream.of
  - ```
    String[] menuItemNames = {"Grits", "Pancakes", "Burrito"};
    Stream.of(menuItemNames); // returns a stream, so needs "=" before it
    ```
  - `Stream.of("Hedgehog", "Kitten", "Fox"); // arbitrary argument count`
- Method 2: call the `stream()` method of any `Collection`
  - `List<String> menuItemNameList = Arrays.asList(menuItemNames);`
  - `menuItemNameList.stream();`
- Method 3: use the StreamBuilder class and it's "accept" method.
    ```

forEach

- ***Intuition*** → iterate over elements in the stream
- Lambda has one argument, return value is ignored
- Terminal operation: does not return another stream!
- ```
  Stream.of(users).forEach(e -> e.logOut());
  ```
  - Logs out all users in system
  ```

forEach

- Loops over stream elements, calling provided function on each element
  - `Stream.of("hello", "world").forEach(word -> System.out.println(word));`
  - A lambda argument is passed

- Can also pass “method references”
  - `Stream.of("hello", "world").forEach(System.out::println);`
  - Syntax: `class::method`

Some More Common Stream Operations

map

Applies a function to each element

filter

Removes elements that don't satisfy a custom rule

sorted

Sorts elements

limit

Return the first N elements

distinct

Removes duplicates

collect

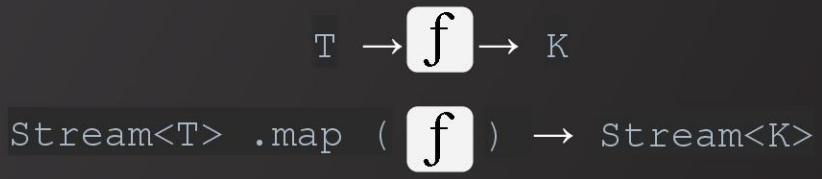
Gets elements out of the stream once we're done (terminal operation)

collect (Basics)

- Also a terminal method.
- Let's say we start with
  - `Stream<Integer> stream = Arrays.asList(3,2,1,5,4,7).stream();`
- Some basic examples: just output all elements as a collection.
  - `List<Integer> list = stream.collect(Collectors.toList());`
  - `Set<Integer> list = stream.collect(Collectors.toSet());`
- Lots more useful goodies,
  - like `Collectors.groupingBy(`  `)` and `Collectors.reducing(`  `);`

map

- ***Intuition*** → modifies the elements of the stream
- The function takes an element of type T and returns an element of type K.



- Example:

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```

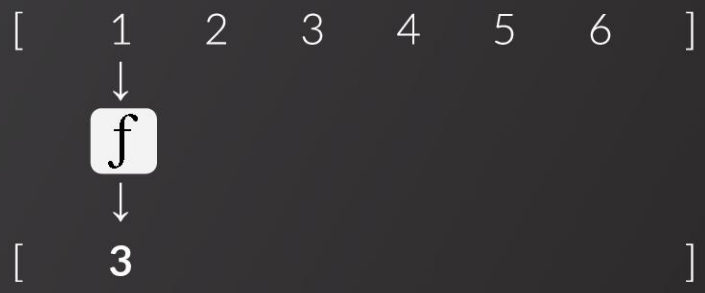
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```

[1 2 3 4 5 6]

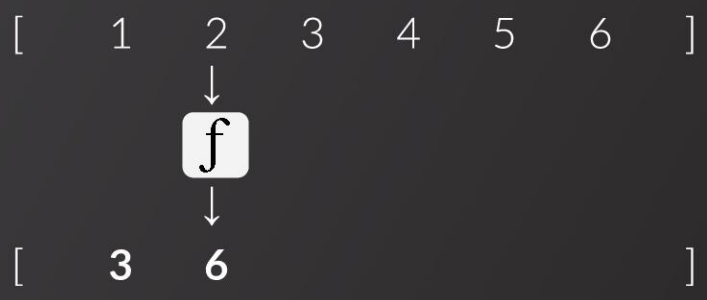
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



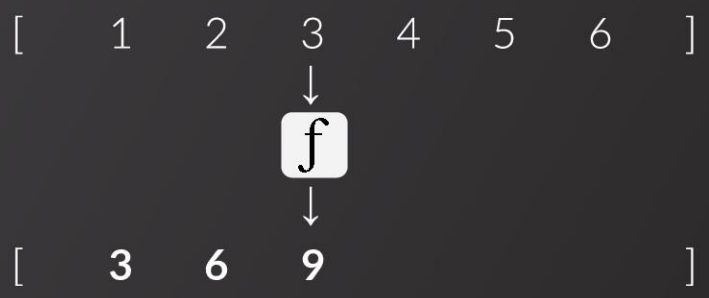
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



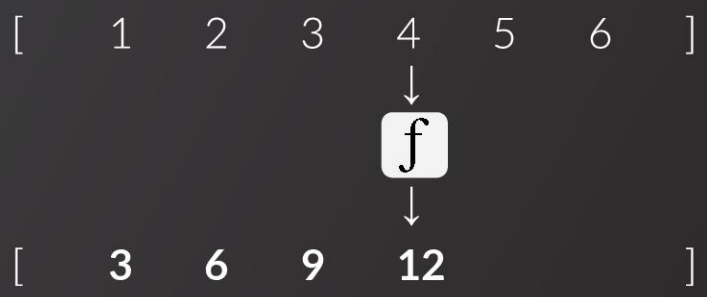
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



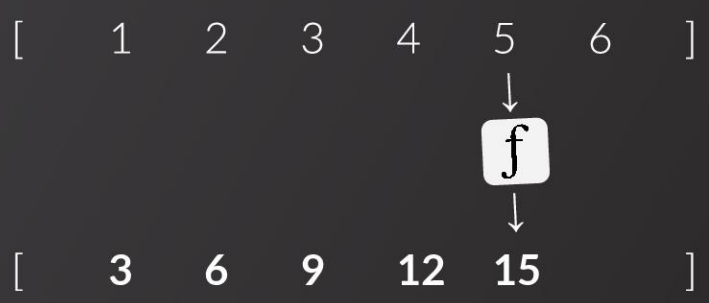
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



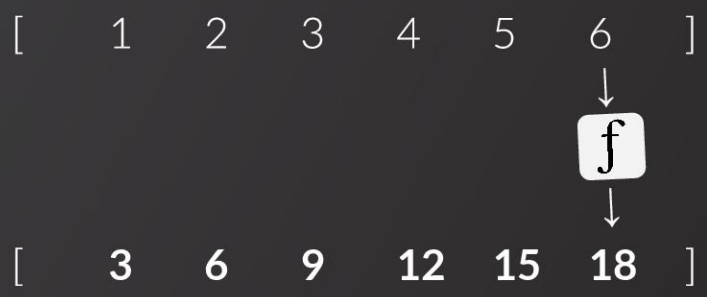
map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



map

```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());
```



map

The function  can be a...

- One-liner lambda expression

```
.map(x -> x/2)
```

- More complex lambda expression

```
.map(x -> {
    … some code …
    return something;
})
```

- Just any function

```
.map(String::toUpperCase)
```

filter

- ***Intuition*** → keeps elements satisfying some condition
- Lambda has one argument and produces a boolean
- Value of boolean determines whether item should be kept

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 2015 2020 2025]

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 2015 2020 2025]

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[**2000** 2005 2010 2015 2020 2025]
^

`y != 2020` evaluates to true

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[**2000** 2005 2010 2015 2020 2025]
^

`y != 2020` evaluates to true

[2000

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 **2005** 2010 2015 2020 2025]

^

`y != 2020` evaluates to true

[2000 2005

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 **2010** 2015 2020 2025]

^

`y != 2020` evaluates to true

[2000 2005 2010

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 **2015** 2020 2025]
^

`y != 2020` evaluates to true

[2000 2005 2010 2015

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 2015 **2020** 2025]
^

`y != 2020` evaluates to false

[2000 2005 2010 2015

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 2015 2020 **2025**]
^

`y != 2020` evaluates to true

[2000 2005 2010 2015 2025

For each element `y`, what does `y != 2020` evaluate to?

filter

```
List<Integer> goodYears = years
        .stream().filter(y -> y != 2020).collect(toList());
```

[2000 2005 2010 2015 2020 2025]

Result: new stream only containing values satisfying `y != 2020`

[2000 2005 2010 2015 2025]

filter

- No requirement to have simple or one-liner condition

```
List<Integer> leapYears =
        years.stream().filter(y -> {
            if (y % 400 == 0) return true;
            if (y % 100 == 0) return false;
            if (y %   4 == 0) return true;
            return false;
        }).collect(toList());
```

- Reminder: lambda is anonymous class implementing functional interface
- Implements `Predicate<T>` which has `boolean test(T t)`

sorted

```
var numbers = Arrays.asList(3, 2, 1, 5, 4, 7);
numbers.stream().sorted().forEach(System.out::println);
```

[3 2 1 5 4 7]

Result: new stream only containing values

[1 2 3 4 5 7]

distinct

```
var numbers = Arrays.asList(3,3,1,1,4,7,8);
numbers.stream().distinct().forEach(System.out::println);
```

[3 3 1 1 4 7 8]

Result: new stream only containing values

[3 1 4 7 8]

limit

```
var numbers = Arrays.asList(3,2,2,3,7,3,5);
numbers.stream().limit(4).forEach(System.out::println);
```

[3 2 2 3 7 3 5]

Result: new stream only containing values

[3 2 2 3]

collect (Reductions)

- `Stream.collect()` allows us to “reduce” a stream to a single output
- This process is called a “reduction”

Some scenarios:

- A list of vote counts in many districts of a state for two candidates can be ***reduced*** to an **aggregate vote count** for each candidate.
- A list of heights for athletes in a basketball team can be ***reduced*** to an **average height** for the whole team.
- A list of ages of students in a class can be ***reduced*** to the **maximum (oldest) age** in the class.

collect (Reductions)

- Create a list of heights (in inches) of team members on a Basketball team

```
List<Integer> teamHeights = List.of(73, 68, 75, 77, 74);
```

- Collect using a "reducer" created with `Collectors.reducing`
- Collectors.reducing() accepts initial accumulator value and a function with two parameters: current value of accumulator and current stream element value

```
int totalHeight = teamHeights.stream().collect(
    Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
);
```

- `System.out.println(totalHeight);`
  - Prints: `367`

collect (Reductions)

```
Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
```

```
[ 73, 68, 75, 77, 74 ]
  ^
```

```
Accumulator value:       0

Current stream element:  73

New accumulator value:   73
```

collect (Reductions)

```
Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
```

```
[ 73, 68, 75, 77, 74 ]
        ^
```

```
Accumulator value:        73

Current stream element:   68

New accumulator value:    141
```

collect (Reductions)

```
Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
```

```
[ 73, 68, 75, 77, 74 ]
```

```
          ^
```

```
Accumulator value:       141

Current stream element:  75

New accumulator value:   216
```

collect (Reductions)

```
Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
```

[73, 68, 75, **77**, 74]

^

Accumulator value: 216

Current stream element: 77

New accumulator value: 293

collect (Reductions)

```
Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
```

```
[ 73, 68, 75, 77, 74 ]
                   ^
```

```
Accumulator value:        293

Current stream element:   74

New accumulator value:    367   (Final result)
```

Some More Common Stream Operations

count

Counts all elements in a stream (terminal)

skip

Gets rid of the first N elements

findFirst

Gets the first stream element wrapped in Optional (terminal)

toArray

Return elements as an array (terminal)

flatMap

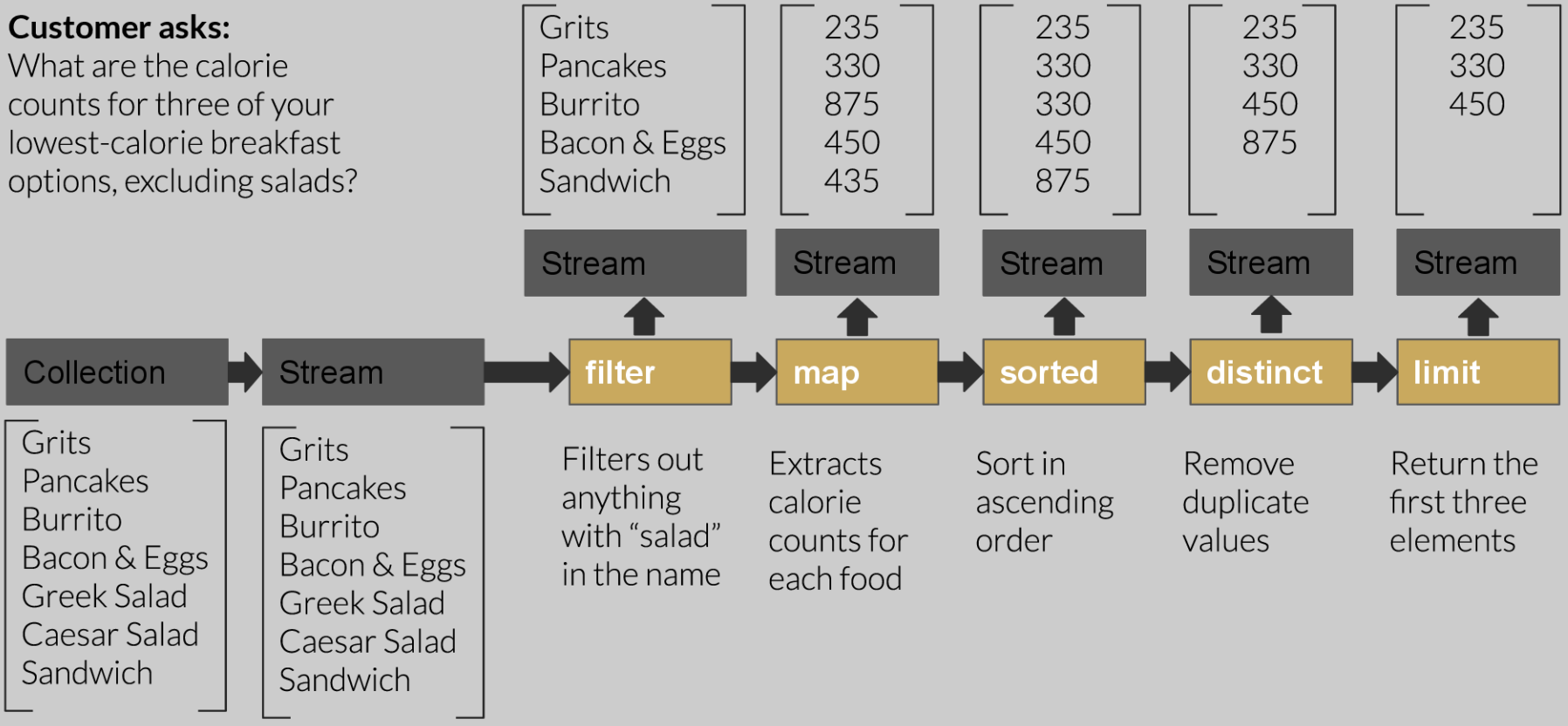
Flatten the data structure (e.g. on stream consisting of Lists)

peek

Do something with each item (like forEach, but not terminal)

Restaurant Example

Customer asks:
What are the calorie counts for three of your lowest-calorie breakfast options, excluding salads?



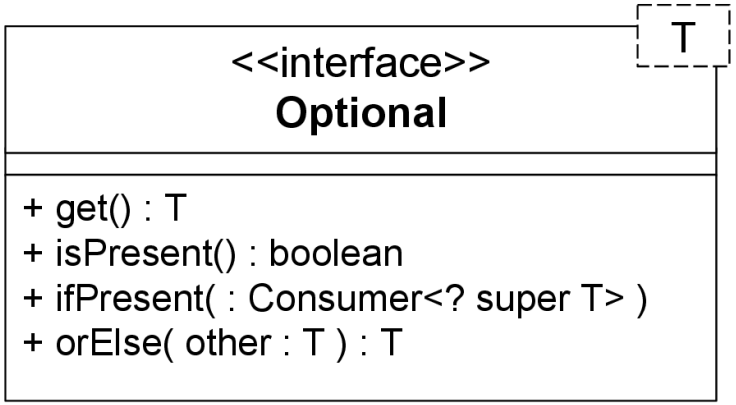
```
// Step 1: Create the list of breakfast items with their respective calories
List<MenuItem> menu = Arrays.asList(
new MenuItem("Grits", 235),
new MenuItem("Pancakes", 330),
new MenuItem("Burrito", 875),
new MenuItem("Bacon & Eggs", 450),
new MenuItem("Greek Salad", 350), // Exclude as it is a salad
new MenuItem("Caesar Salad", 300), // Exclude as it is a salad
new MenuItem("Sandwich", 435)
);

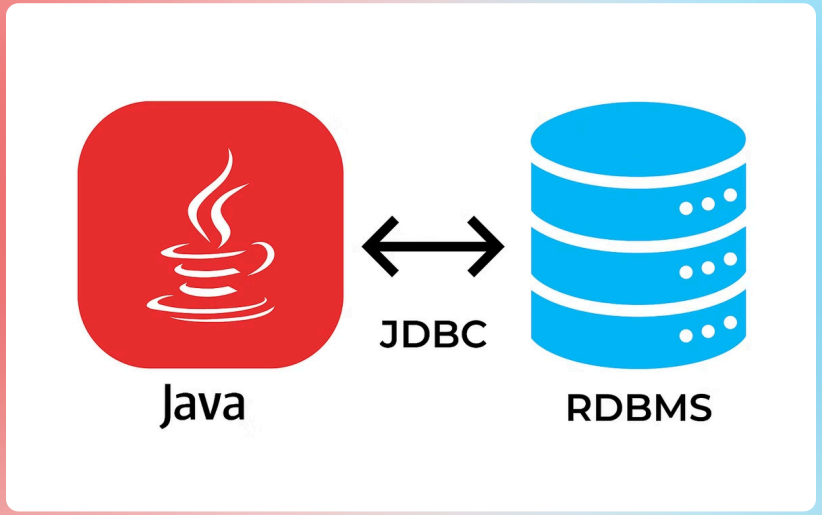
// Step 2: Use Stream to process the menu and get the calorie counts of the lowest
three breakfast items (excluding salads)
List<Integer> lowestCalories = menu.stream()
.filter(item -> !item.getName().toLowerCase().contains("salad")) // Exclude salads
.map(MenuItem::getCalories) // Extract calorie count
.sorted()         // Sort in ascending order
.distinct()        // Remove duplicates if any
.limit(3)         // Get the first 3 items
.collect(Collectors.toList()); // Collect the result

// Step 3: Output the result
System.out.println("Three lowest-calorie breakfast options: " + lowestCalories);
```

Class `Optional<T>`

The class Optional<T> is a container wrapped around an object and is useful if you do not know whether its content is `null` or not (e.g. when using in fluent programming style).





JDBC: Java Database Connectivity

JDBC connects Java applications to relational databases. It enables data-driven Java development. This presentation covers key concepts, drivers, API, and best practices for JDBC usage.

Types of JDBC Drivers

Type 1: JDBC-ODBC Bridge

Platform-dependent, uses ODBC driver, deprecated for production.

Type 2: Native-API Driver

Uses vendor libraries, better performance, platform-dependent.

Type 3: Network Protocol Driver

Middleware server based, platform-independent, good for web apps.

Type 4: Thin Driver

Direct database protocol, platform-independent, best performance.

Core JDBC API Overview

DriverManager

Manages all registered JDBC drivers.

Connection

Represents a database session.

Statements

Execute SQL queries and updates.

ResultSet

Holds query result data.

SQLException

Handles database errors.

Steps to Connect to a Database

- Load the driver class using Class.forName.
- Establish connection with DriverManager.getConnection using URL, user, and password.
- Create a Statement object to execute queries.
- Execute the query and get ResultSet.
- Process ResultSet data with looping.
- Close connection using try-with-resources for safety.

Statement Interfaces Explained

Statement

Executes static SQL; prone to SQL injection; basic methods include executeQuery.

PreparedStatement

Precompiled SQL with placeholders; prevents injection; efficient for repeated queries.

CallableStatement

Used to invoke stored procedures with input and output parameters.

Batch Processing & Transaction Management

Batch Processing

Combine multiple SQL commands using addBatch and executeBatch for efficiency.

Transaction Management

Control commit and rollback with setAutoCommit, ensuring ACID properties for data integrity.

Example: Secure fund transfers between accounts.

Metadata and CallableStatement Usage

DatabaseMetaData

Fetches details about database structure like tables and driver info.

ResultSetMetaData

Provides details about query results such as column names and types.

CallableStatement

Prepare and execute stored procedures; set parameters and retrieve outputs.

Best Practices and Conclusion

Use connection pooling to improve resource management.

Always handle exceptions with try-catch and close resources properly.

Choose the right JDBC driver for your platform needs.

Use PreparedStatement to safeguard against SQL injection.

Batch processing boosts performance for bulk SQL operations.

Manage transactions carefully to maintain data integrity.

JDBC bridges Java and databases, vital for enterprise apps.

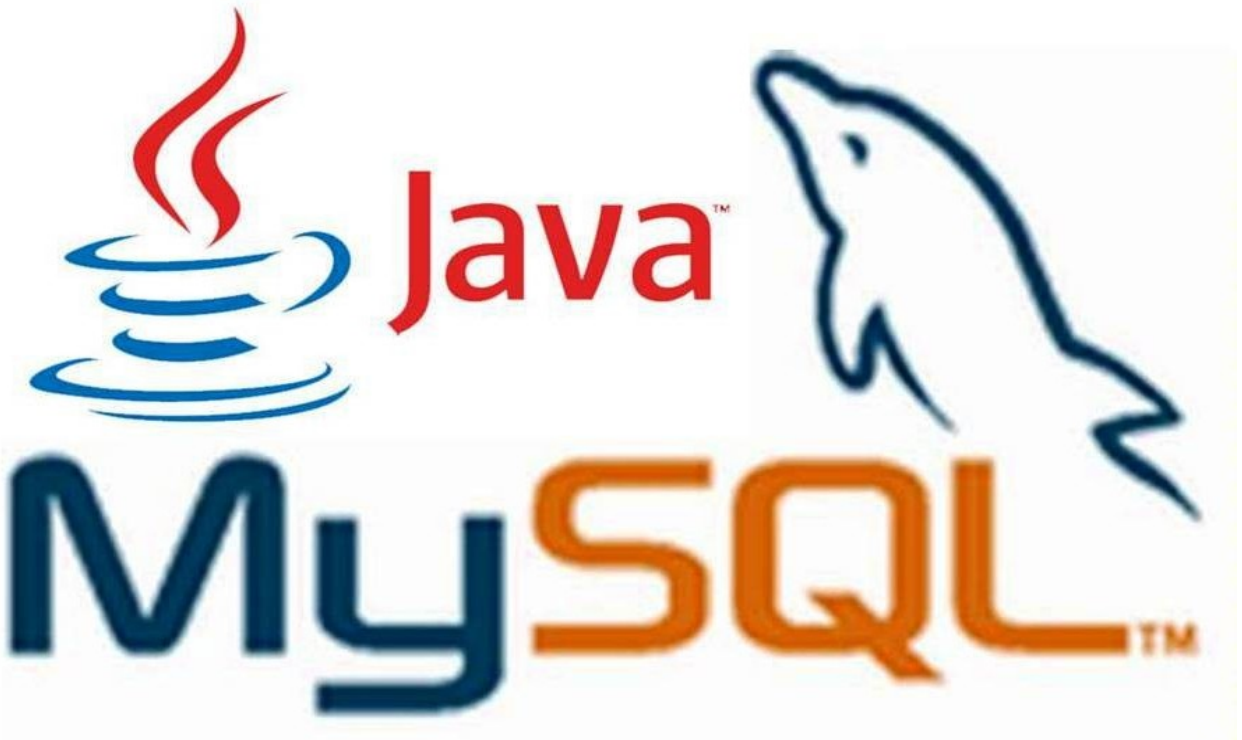
Thank You

Feel free to reach out with any questions.

JDBC

Sadhu Sreenivas

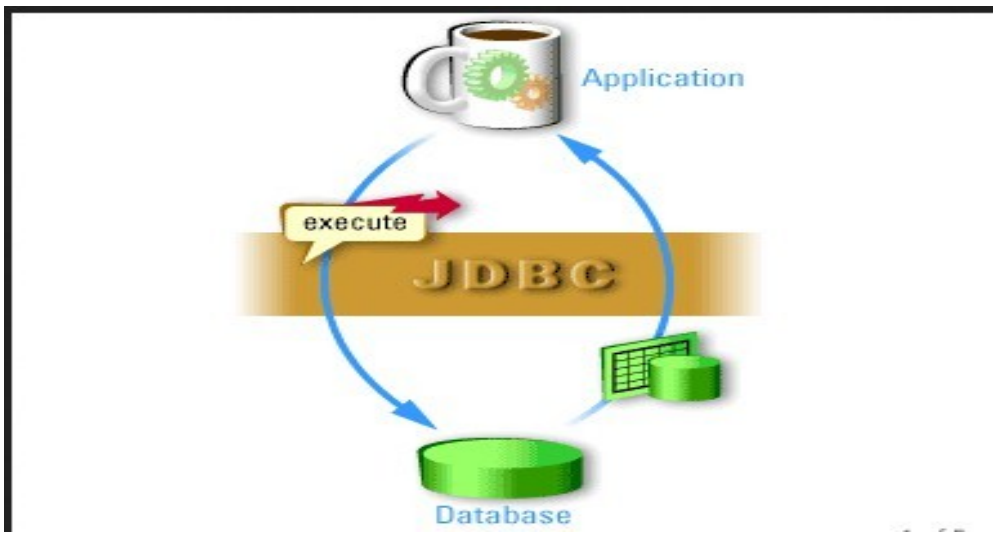
C -DAC Hyderabad



What is JDBC?

Definition

- JDBC is a Java-based data access technology (Java Standard Edition platform) from Oracle Corporation.
- This technology is an API for the Java programming language that defines how a client may access a database.
- It provides methods for querying and updating data in a database.



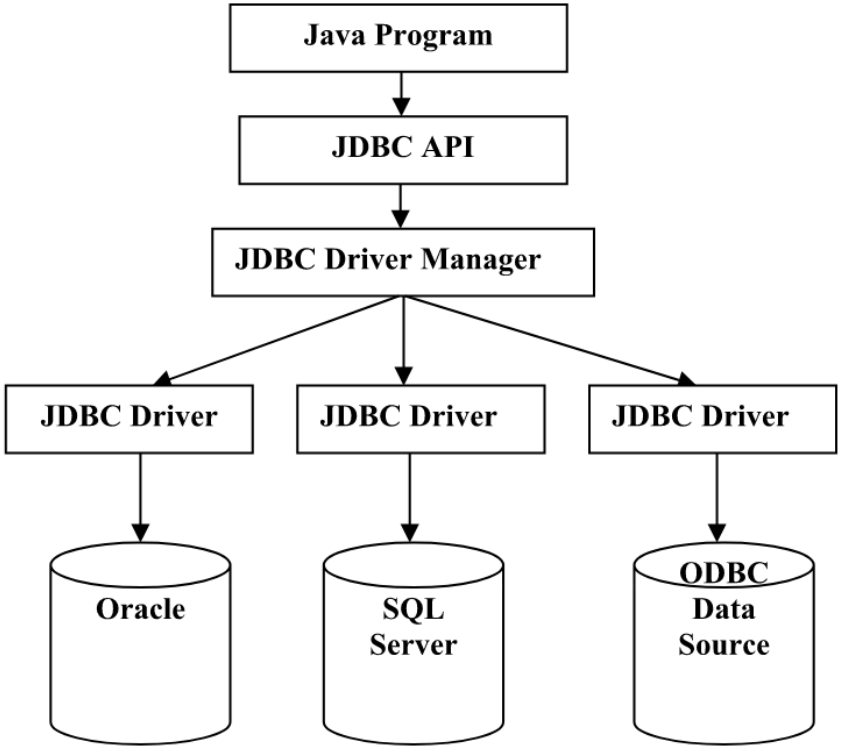
JDBC History

JDBC History

- Before JDBC, ODBC API was used to connect and execute query to the database.
- But ODBC API uses ODBC driver that is written in C language which is platform dependent and unsecured.
- Sun Micro System has defined its own API (JDBC API) that uses JDBC driver written in Java language.

JDBC History

- Sun Microsystems released JDBC as part of JDK 1.1 on February 19, 1997.
- The JDBC classes are contained in the Java package java.sql
- Some classes and interfaces are available javax.sql



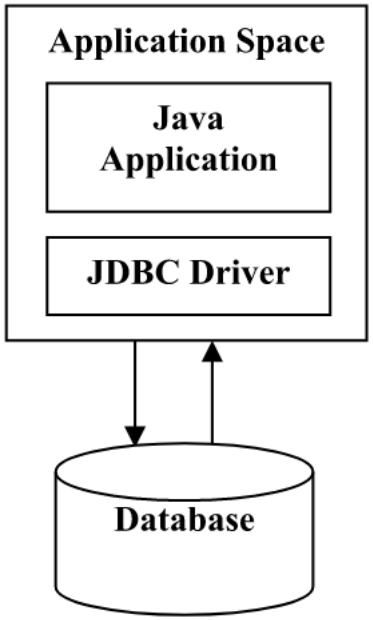
Layers of JDBC Architecture

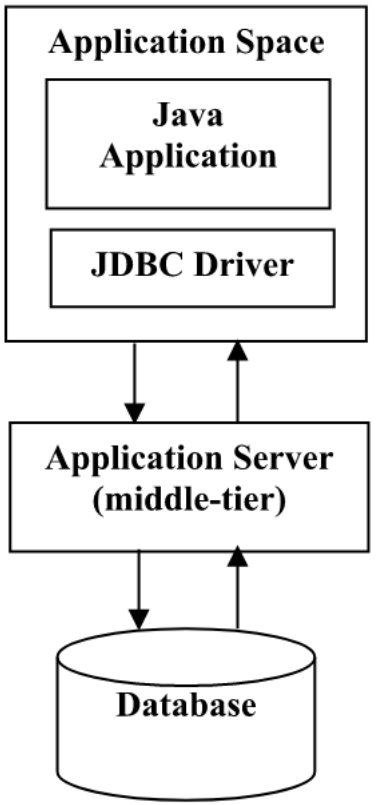
What is API

API

- The Java API is the set of classes included with the Java Development Environment. These classes are written using the Java language and run on the JVM. The Java API includes everything from collection classes to GUI classes.

- JDBC is also an API.





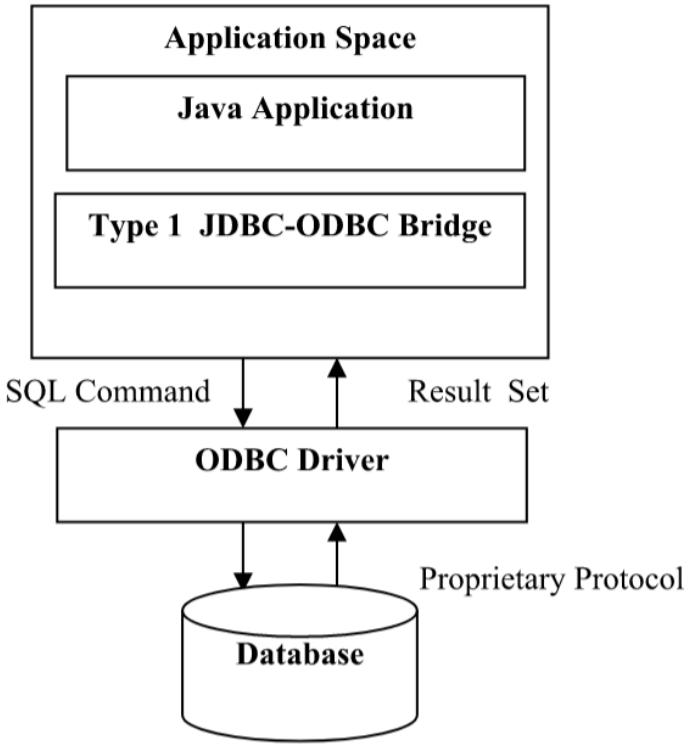
JDBC Drivers

JDBC Drivers

- *JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:*
  - Type 1: JDBC-ODBC bridge driver
  - Type 2: Native-API driver (partially java driver)
  - Type 3: Network Protocol driver (fully java driver)
  - Type 4: Thin driver (fully java driver)

Type 1:
JDBC-ODBC Bridge Driver

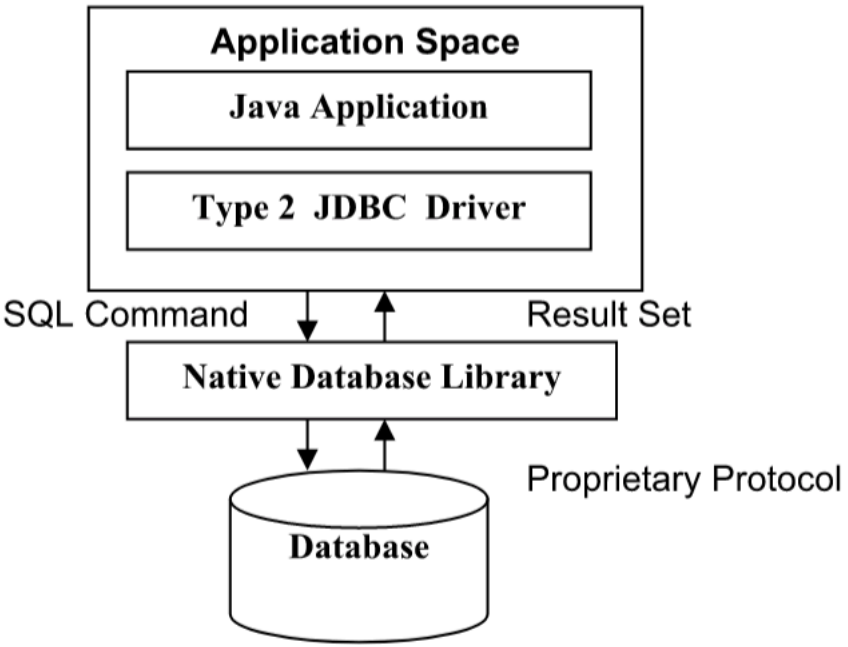
- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. This is now discouraged because of thin driver.



Type 1 JDBC-ODBC Bridge

Type 2:
Native-API Driver

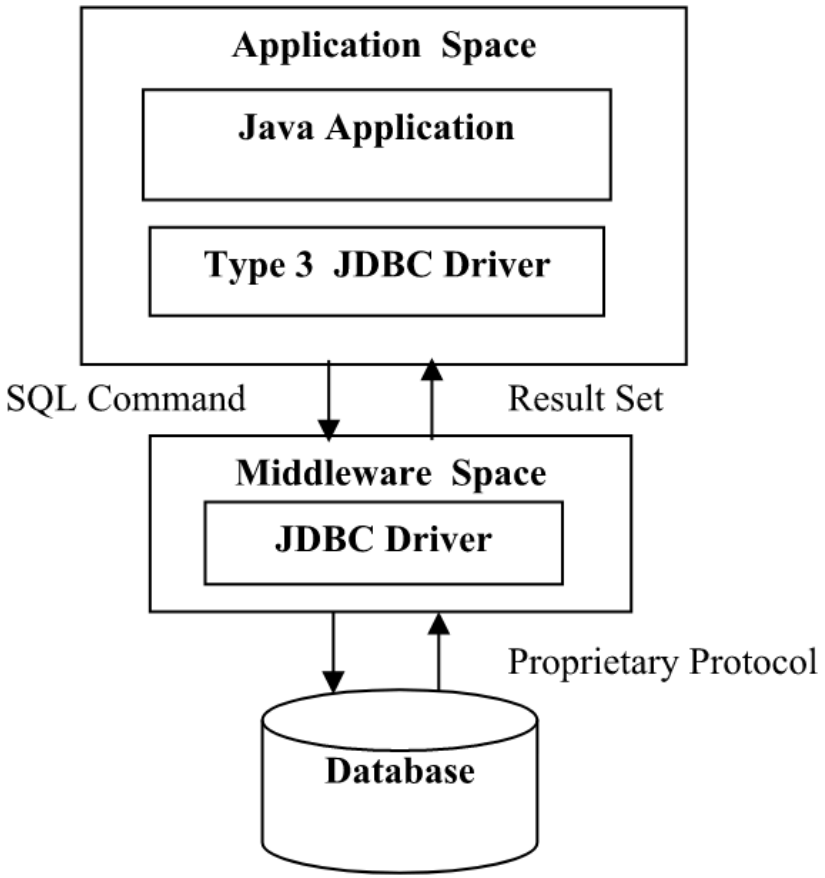
- The Native API driver uses the client-side libraries of the database.
- It is not written entirely in Java.



Type 2 Native-API JDBC driver

Type 3:
Network Protocol Driver

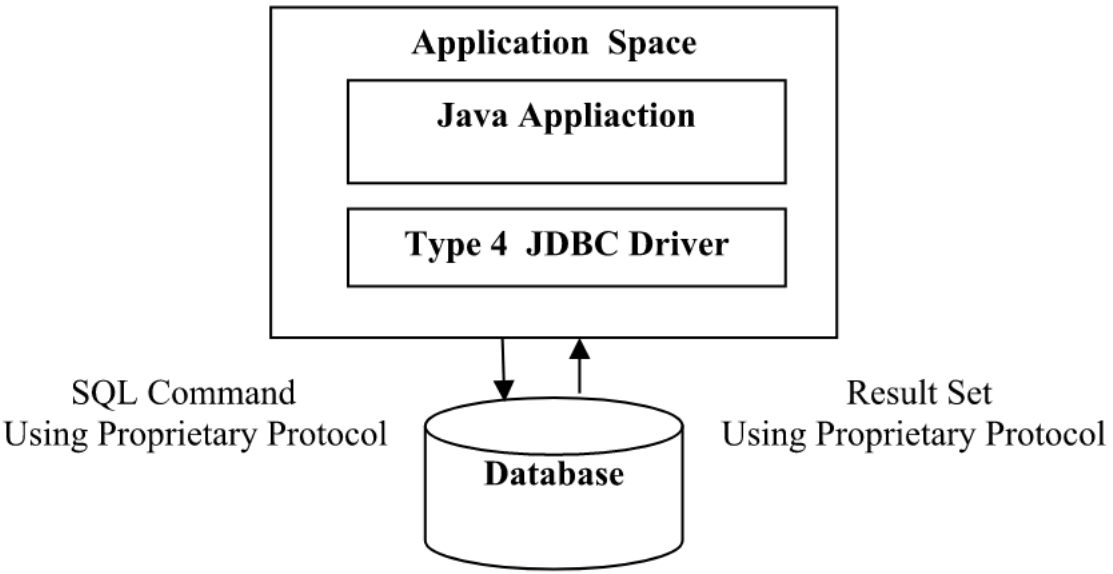
- The Network Protocol driver uses middle ware (application server).
- It is fully written in Java.



Type 3 JDBC-Net driver

Type 4: Thin Driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol.
  - It is known as thin driver. It is fully written in Java
  - Better performance than all other drivers.
  - No software is required at client side or server side.
  - *Disadvantage:* Drivers depends on the Database.



Type 4 native-protocol JDBC driver

Steps to Connect to the Database in Java

Steps to Connect to Database

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

Registering the Driver

- The forName() method of Class class is used to register the driver class.
- Class.forName("com.mysql.jdbc.Driver");

Creating Connection Object

- The getConnection() method of DriverManager class is used to establish connection with the database.
- Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306","root","password");

Creating Statement Object

- The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.
- Statement stmt=con.createStatement();

Execute Query

- The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.
- *ResultSet rs=stmt.executeQuery("select * from emp");*

```
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Closing Connection

- By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.
- con.close();

Connecting to the MySQL Database

Connect to MySQL Database

- Driver class: com.mysql.jdbc.Driver.
- Connection URL: jdbc:mysql://localhost:3306/db_name
- Username: The default username for the mysql database is root.
- Password: Given by the user at the time of installing the mysql database
- *Example:*

```
Connection
con=DriverManager.getConnection( “jdbc:mysql://
localhost:3306/cdac","root","admin");
```

Loading the .jar

- Download the MySQL connector.jar from mysql.com

  Paste the mysqlconnector.jar in the lib folder of source directory.

  Set the classpath

DriverManager class

- The DriverManager class acts as an interface between user and drivers.
- It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- *Connection con = null;*
  con=DriverManager.getConnection("jdbc:mysql://localhost:3306","root","password");
- *DriverManager.registerDriver().*

Connection Interface

- A Connection is the session between Java application and database.
- The Connection interface is a factory of Statement and PreparedStatement.
- Object of Connection can be used to get the object of Statement and PreparedStatement.

Connection Interface

Methods of Connection interface:

- public Statement createStatement(): creates a statement object that can be used to execute SQL queries.
- public void commit(): saves the changes made since the previous commit/rollback permanent.
- public void close(): closes the connection and Releases a JDBC resources immediately.

Statement Interface

- The Statement interface provides methods to execute queries with the database.
- The statement interface is a factory of ResultSet.
- It provides factory method to get the object of ResultSet.

Statement Interface

Methods of Statement interface:

- public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.
- public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
- public boolean execute(String sql): is used to execute queries that may return multiple results.

Statement Interface

Example:

- ```
  Statement stmt=con.createStatement();
  int result=stmt.executeUpdate("delete from table where id=xy");
  System.out.println(result+" records affected");
  con.close();
  ```
  ```

ResultSet interface

- The object of ResultSet maintains a cursor pointing to a particular row of data.
- Initially, cursor points to before the first row.

ResultSet Interface

Methods of ResultSet interface:

- **public boolean next():** is used to move the cursor to the one row next from the current position.
- **public boolean previous():**is used to move the cursor to the one row previous from the current position.
- **public boolean first():** is used to move the cursor to the first row in result set object.
- **public boolean last():**is used to move the cursor to the last row in result set object.

ResultSet Interface

Methods of ResultSet interface:

- **public int getInt(int columnIndex):** is used to return the data of specified column index of the current row as int.
- **public int getInt(String columnName):** columnName): is used to return the data of specified column name of the current row as int.
- **public String getString(int columnIndex):** is used to return the data of specified column index of the current row as String.
- **public String getString(String columnName):** is used to return the data of specified column name of the current row as String.

ResultSet Interface

Example:

```
ResultSet rs=stmt.executeQuery("select * from table");

//getting the record of 3rd row

rs.absolute(3);

System.out.println(rs.getString(1)+" "+rs.getString(2)+"
"+rs.getString(3));


con.close();
```

PreparedStatement Interface

- The PreparedStatement interface is a sub interface of Statement.
- It is used to execute parameterized query.
- Example of parameterized query:
  - String sql="insert into emp values(?,?,?)";

PreaparedStatement Interface

Methods of PreparedStatement:

- *public void setInt(int paramIndex, int value): sets the integer value to the given parameter index.*
- *public void setString(int paramIndex, String value): sets the String value to the given parameter index.*
- *public void setFloat(int paramIndex, float value): sets the float value to the given parameter index.*

PreparedStatement Interface

Methods of PreparedStatement:

- public void setDouble(int paramIndex, double value): sets the double value to the given parameter index.
- public int executeUpdate(): executes the query. It is used for create, drop, insert, update, delete etc.
- public ResultSet executeQuery(): executes the select query. It returns an instance of ResultSet.

PreparedStatement Interface

Example:

- PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
  stmt.setInt(1,101);//1 specifies the first parameter in the query

  stmt.setString(2,”ABC");

  int i=stmt.executeUpdate();

  System.out.println(i+" records inserted");

Questions

Write a program to implement CRUD in a table.

Write a program to implement PreparedStatement?

Thank You

C -DAC Hyd

SQL Window Functions Overview

Essential tools for advanced data analysis in SQL

Ranking Functions: row_number(), rank(), dense_rank()

row_number()

Unique sequential number per row

rank()

Assigns rank with gaps for ties

dense_rank()

Ranks without gaps in ties

The OVER() Clause

- Defines window for functions
- Used with ranking and analytic functions
- Enables frame specification

Partitioning Data: PARTITION BY

- Divides data into groups for functions
- Resets computation per partition
- Improves targeted analysis precision

TABLE: EMP

Empid	empname	**empphone**	**empsal**	**deptno**
101	Ajay	234567	20000	25
102	Vijay	654378	15000	30
103	Ramesh	345678	10000	25
104	Ram	346279	15000	10

Ordering Within Windows: ORDER BY

- Defines row order inside partitions
- Impacts ranking and lead/lag computations
- Supports ascending and descending modes

Lead and Lag Functions: lead() & lag()

lead()

Accesses next row value

- Useful for comparisons
- Lookahead calculations

lag()

Accesses previous row value

- Lookback calculations
- Detect changes over time

Value Retrieval Functions: nth_value() & first_value()

first_value()

Returns first row value in window

- Anchor point for comparison

nth_value()

Returns nth row value in window

- Accesses any position in ordered set

Key Takeaways & Applications

- Window functions enable complex analytics
- Partition and order tailor analysis scope
- Lead, lag, and rank unlock powerful insights