IT5100B

Industry Readiness

Stream Processing

LECTURE 1

Course Introduction

FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

COURSE ADMINISTRATION

ABOUT IT5100B

INDUSTRY READINESS: STREAM PROGRAMMING

The global availability of data has reached a level where **aggregating data into generic, general-purpose “stores” is no longer feasible**. Having data collections statically available for querying by interested parties on demand is increasingly becoming the way of the past.

Instead, a new paradigm, called **Data Streaming**, has emerged recently. In this paradigm, data is bundled into high-throughput “streams" that are sharded efficiently across a large number of network nodes...

ABOUT IT5100B

INDUSTRY READINESS: STREAM PROGRAMMING

Consumers, sometimes counted in hundreds of thousands, or millions, “**subscribe**” to data subsets and are notified when new data becomes available, being under the obligation to process it immediately, or lose it.

Consequently, data storage is **no longer centralized**, but rather distributed into many smaller-sized abstract collections. This new approach to “Big Data” requires a new set of tools, platforms, and solution patterns...

ABOUT IT5100B

INDUSTRY READINESS: STREAM PROGRAMMING

In this course we propose to explore several facets of this new paradigm:

- The **Stream paradigm** introduced in Java 8.
- Platforms that implement Data Streaming, such as **Kafka**, and the Java bindings in the library **KafkaConnect**.
- Computing paradigms for stream processing, such as Reactive Programming, and the library **RxJava**.
- High-performance stream computing platforms, such as **Flink**

The course will be using **Java** as the main vehicle for introducing concepts and showcasing examples.

YOUR INSTRUCTOR

Mr **FOO** Yong Qi, B.Eng., M.Sc.

Instructor | NUS School of Computing

Ph.D. Student

yongqi@nus.edu.sg

COM2-02-27

https://yongqi.foo/

HOBBIES

Teaching :: Music :: Exercise :: Programming Languages and Software Engineering

CLASS LINKS

POLL EVERYWHERE / TELEGRAM

https://pollev.com/yongqifoo

https://t.me/+EaDk2xa6jH1kOWU1

COURSE SCHEDULE

TENTATIVE CLASS SCHEDULE

DATE	LECTURE	REMARKS
19 Jan	Course Introduction	
26 Jan	Generics	
2 Feb	Reactive Programming	
9 Feb	Event Streaming	Video due to CNY
16 Feb	Working with Events	
23 Feb	High Throughput Stream Processing	
1 Mar	Course Conclusion	Held at COM1-0206

CONTINUAL ASSESSMENT

TENTATIVE GRADED ITEMS

ITEM	WEIGHTAGE
Assignment 1: Java Programming	20%
Assignment 2: Reactive Programming	20%
Assignment 3: Event Streaming	20%
Project	40%

CONTINUAL ASSESSMENT

NOTICE

- Assignment deadlines are flexible within reason (job interviews etc), **contact me** for deadline extension
- Assignments are mostly coding-based

No code sharing

"Can you help me debug this code?"

Can discuss **concepts** and ideas:

"Based on lecture 4, you can subscribe to the event bus using the KStreams API"

CONTINUAL ASSESSMENT

NOTICE

You are allowed to **use ChatGPT with no restrictions for learning**, therefore:

You are allowed to use ChatGPT for assignments, but

ALL SUBMISSIONS YOU MAKE MUST BE ENTIRELY YOURS

Copying wholesale from ChatGPT or any other source constitutes plagiarism

ChatGPT is **not required** for this course (all knowledge required in assessed items are taught and/or freely available online)

CONTINUAL ASSESSMENT

NOTICE

NUS takes a strict view of plagiarism and cheating

Disciplinary action will be taken against students who violate NUS Student Code of Conduct

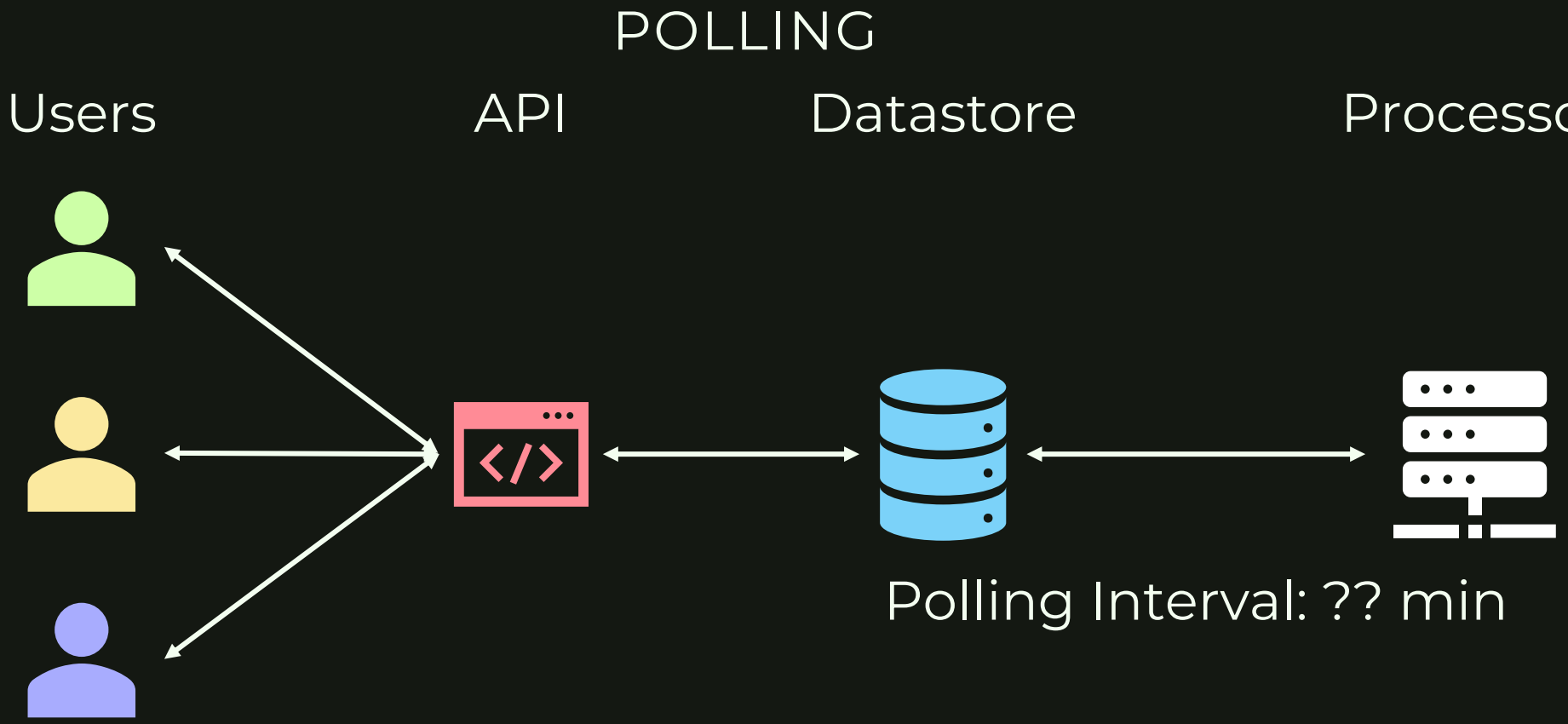
No part of your assignment can come from any other source

No discussion and sharing of solutions during quizzes

See NUS Plagiarism Notice

WHAT/WHY STREAMS?

WHAT/WHY STREAMS?

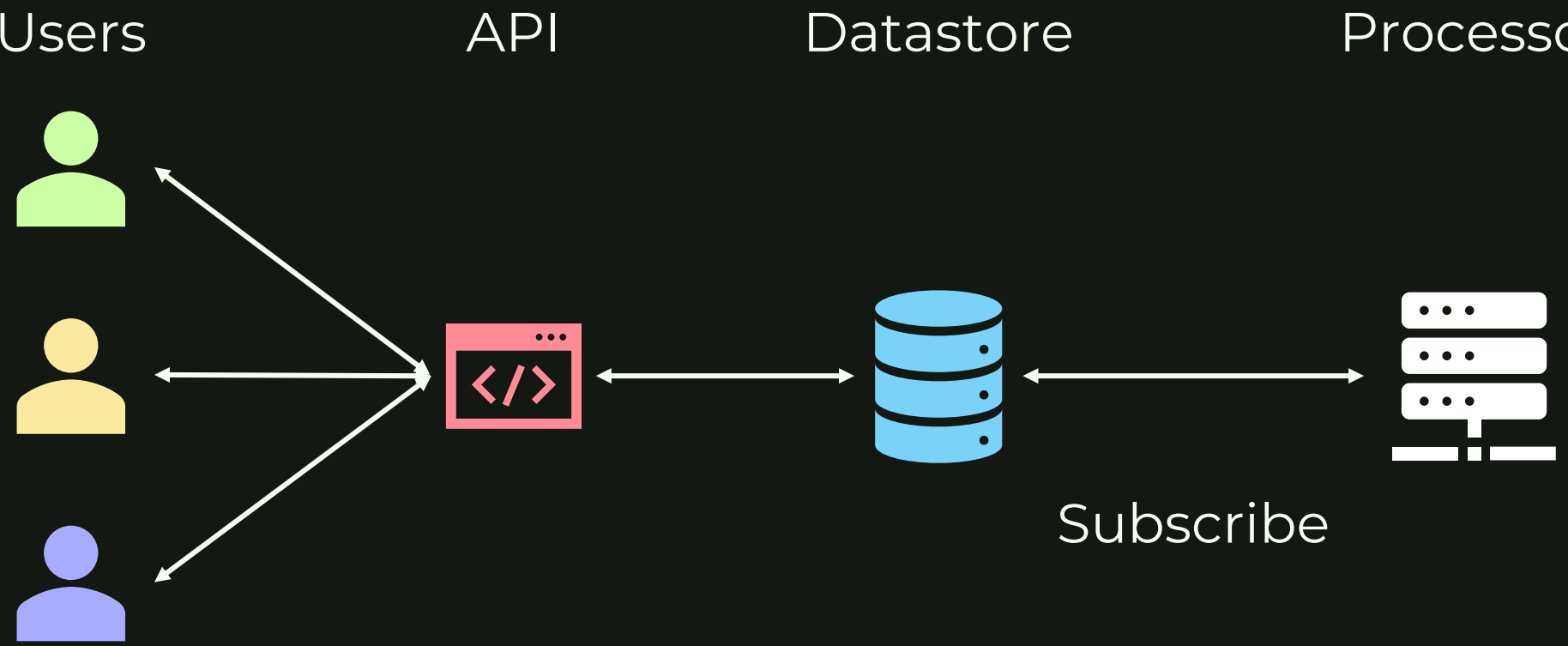


Smaller polling interval: lower latency but potentially more wasted resources

Higher polling interval: higher latency, less wasted resources

WHAT/WHY STREAMS?

A PARADIGM SHIFT



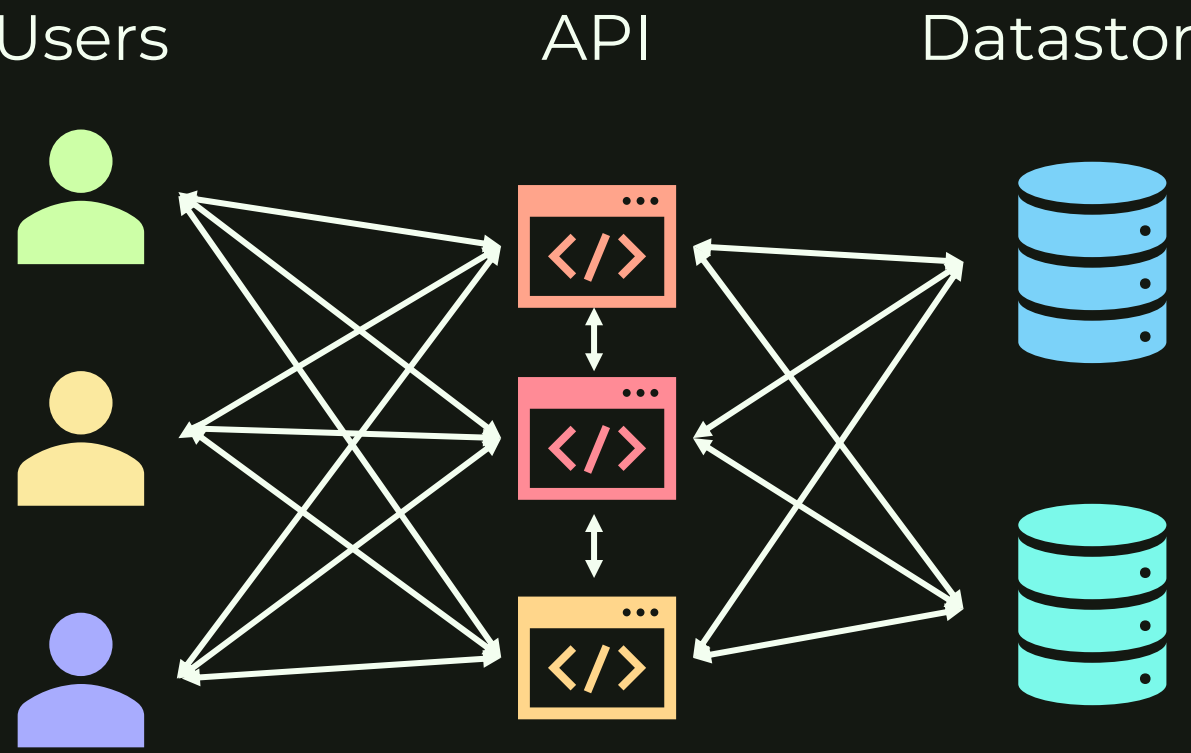
Processor **subscribes** to state changes; reactive programming; data moves in real time

KEY POINT #1

Instead of pulling data from a data source, **subscribe** to a data source to receive real-time changes

WHAT/WHY STREAMS?

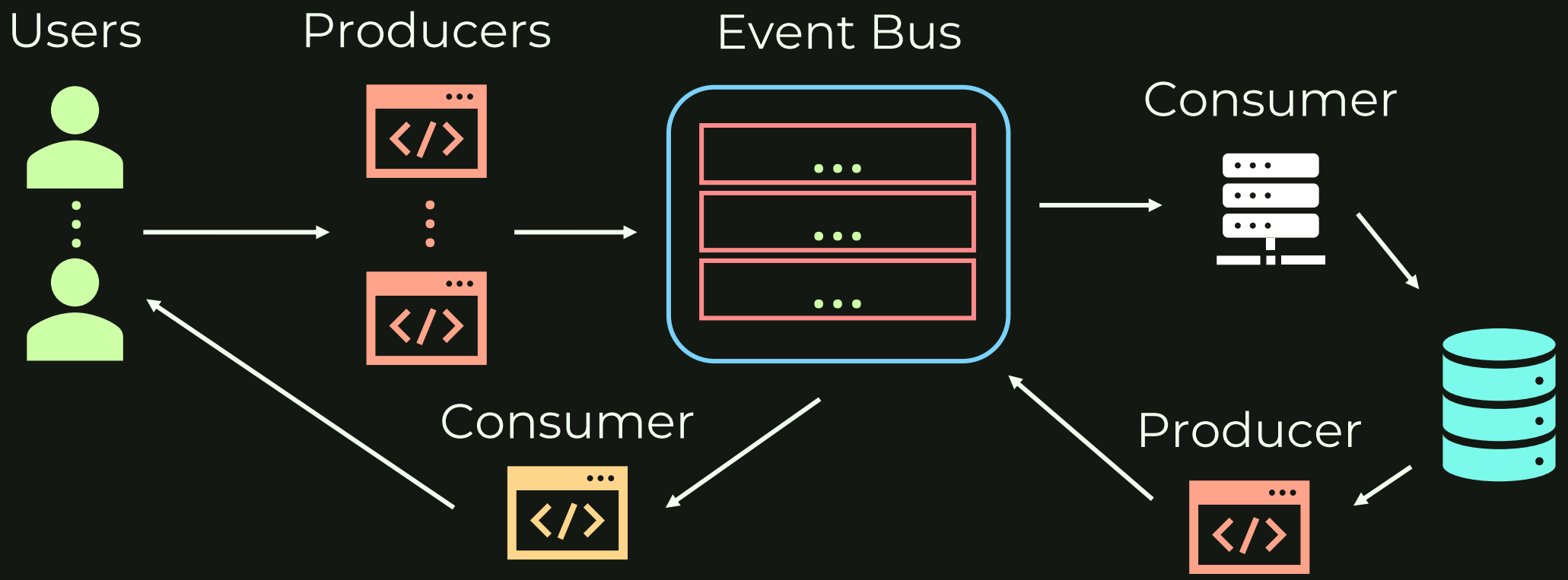
DISTRIBUTED STATE CHANGE



Dealing with large amounts of data at scale is incredibly challenging; tight coupling of systems that produce and consume state change

WHAT/WHY STREAMS?

DISTRIBUTED EVENT LOG



Things that happen are **events**; when things happen, push event to **event log**; when events appear in event log, do stuff with them

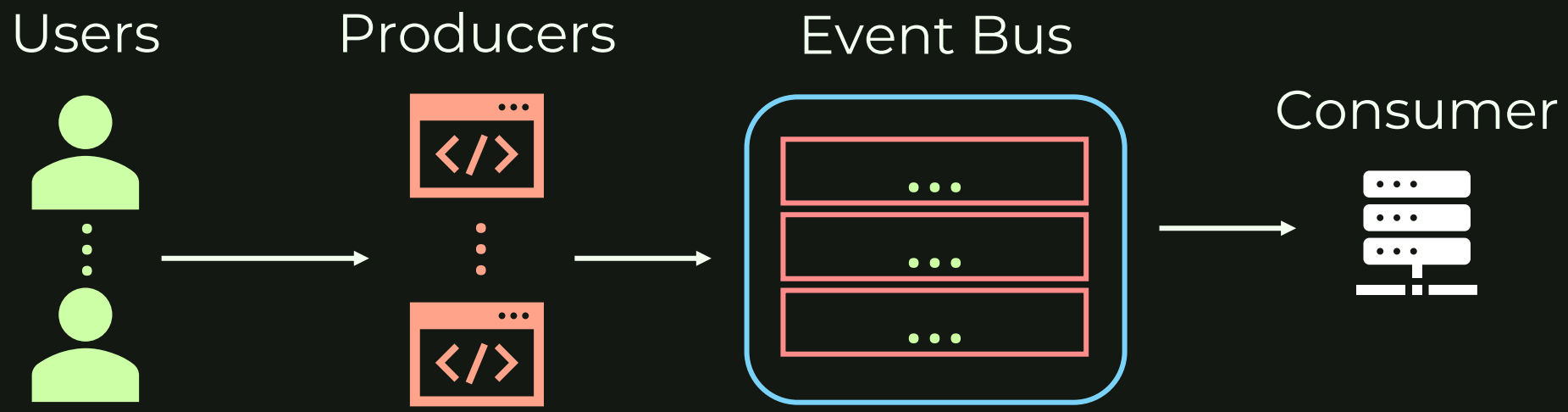
Keep things simple for easier decoupling and scaling

KEY POINT #2

Everything that happens in an application can be described by **events**, use an event bus/store to decouple producers and consumers of **event streams**

WHAT/WHY STREAMS?

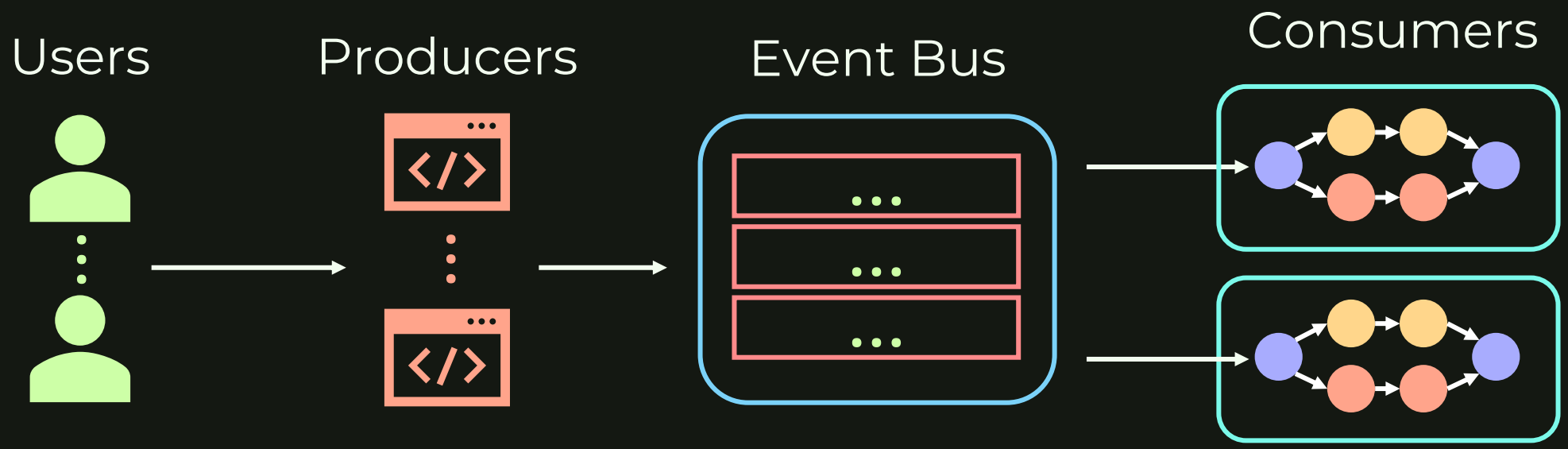
SINGLE-MACHINE PROCESSOR



Event stream processors/consumers process a lot of data; running process on one machine not scalable (especially if stateful!)

WHAT/WHY STREAMS?

DISTRIBUTED STREAM PROCESSING



Distribute process into subprocesses, split subprocesses among nodes in compute cluster

Can provision multiple copies of processor, each cluster processes one distinct partition of event log

KEY POINT #3

Distribute event stream processing into **multiple nodes** in **multiple computing clusters** for **high throughput**

KEY TECHNOLOGIES

TOOLS WE SHOULD SEE IN THIS COURSE

Reactive Programming: **Reactor** (alternative to RxJava)
Message broker/event bus: **Apache Kafka**
Stream Processor: **Apache Flink**

Others: Docker, PostgresQL, KafkaConnect, Debezium etc.

Language of choice: **Java**

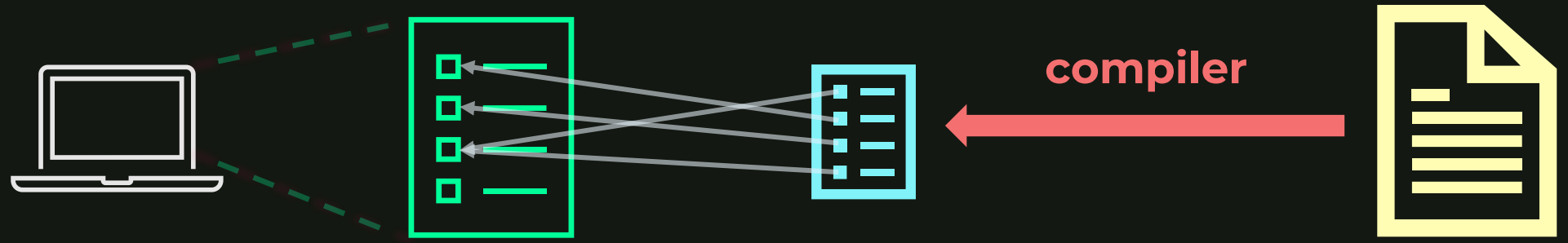
JAVA FUNDAMENTALS

WHAT IS JAVA?

A **cross-platform**, **high-level**, **multi-threaded**, **Ahead-of-Time (AOT)** and **Just-In-Time (JIT) compiled**, **statically-typed**, **Object-Oriented** (class-based inheritance), **general purpose** **programming language** created by James Gosling in 1991

WHAT IS JAVA?

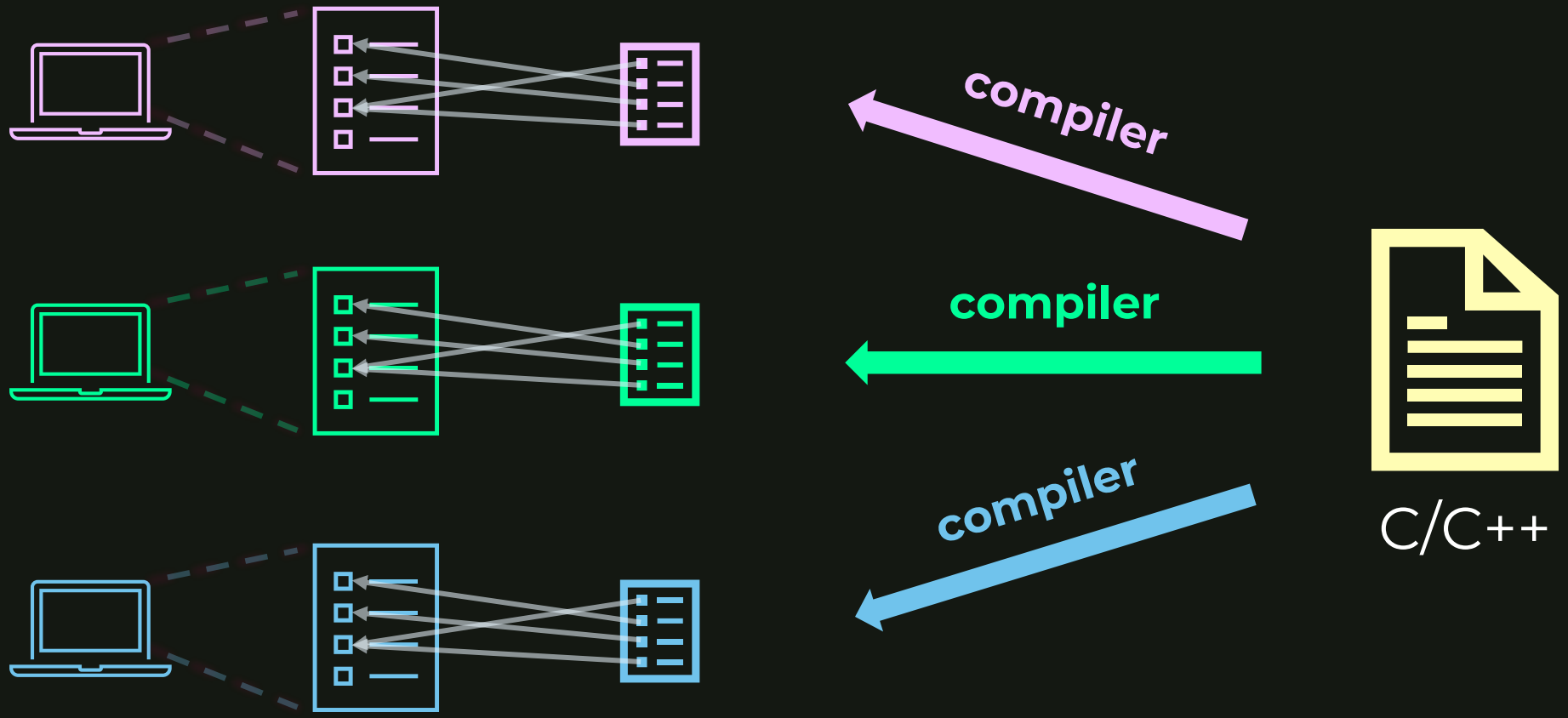
COMPILED LANGUAGES



(Among other things), compilation is the process of **translating code** in one language (usually of higher-level) into code of another (lower-level)

WHAT IS JAVA?

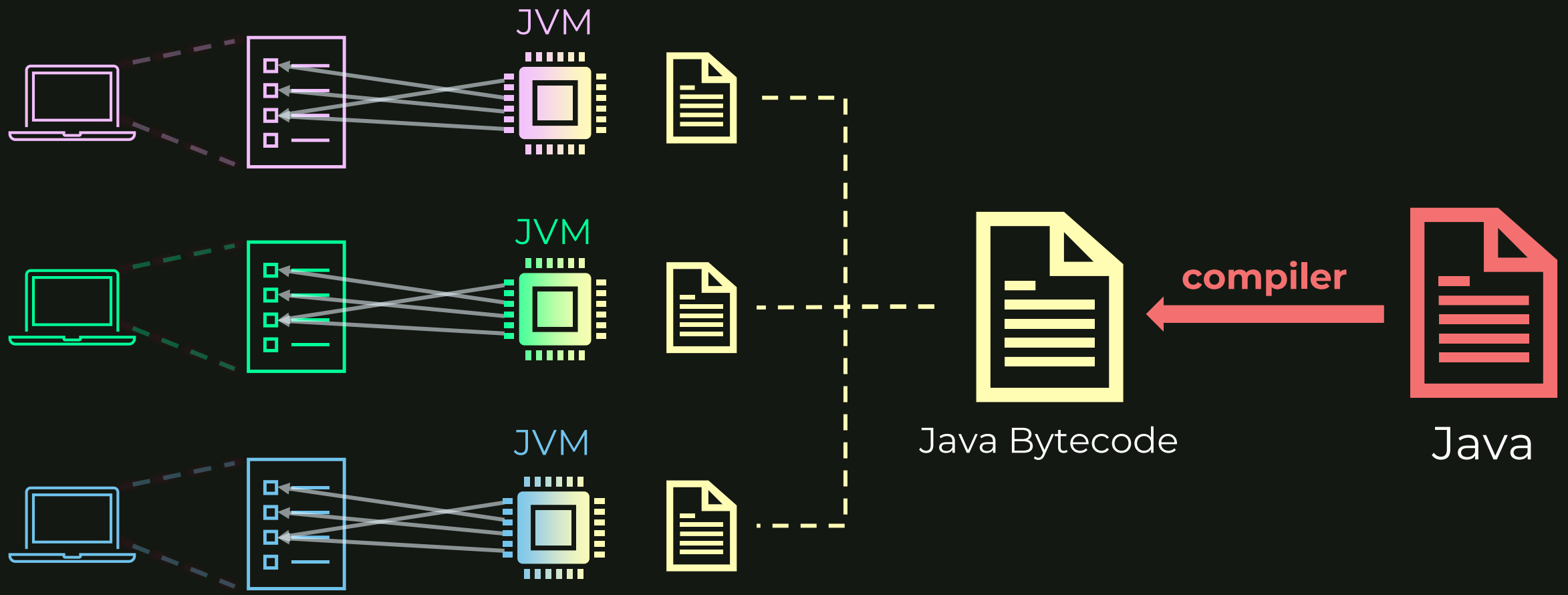
COMPILED LANGUAGES



Compilation affords the advantage of **serving multiple platforms** from a **single codebase**

WHAT IS JAVA?

CROSS-PLATFORM



Write Once, Run Anywhere (WORA)

WHAT IS JAVA?

HIGH-LEVEL

```
01010101010111010001010101001010101010
10101010101010101010101010101010010101
01010101010010101100101010101010000011
00101111010101010110000101101101101100
01011111010100101000110100101010100101
        0110111010101010101...
```

Machine code

```java
class Main {
    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        System.out.println(x + y);
    }
}
```

High-level code

WHAT IS JAVA?

HIGH-LEVEL

```java
class Main {
    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        System.out.println(x + y);
    }
}
```

High-level code

- Low level details abstracted away (ease of development)
- No control of low-level details (abstraction penalty)

ease of development > abstraction penalty

WHAT IS JAVA?

STATIC TYPING

```python
x = 1
s = 'Hello World!'
a = []
x = 'not an int' # ok
```

Python (dynamically-typed)

```java
int x = 1;
String s = "Hello world!";
int[] a = new int[5];
x = "not an int"; // error
```

Java (statically-typed)

Dynamic Typing: types of entities are determined at runtime

Static Typing: types of entities are determined at compile time

KEY POINT 4

Java is a **compiled** and **statically-typed** language

WORKING WITH JAVA

Things you'll need: either

1) Text editor (Visual Studio Code)
2) The Java Development Kit (JDK)

Integrated Development Environment (IDE) (IntelliJ IDEA)

EDIT-COMPILE-RUN

A TYPICAL WORKFLOW DEMONSTRATION

JSHELL

THE JAVA REPL

```
jshell> 1 + 1
$2 ==> 2
jshell> $2 + 4
$3 ==> 6
jshell> "a" + $3
$4 ==> "a6"
jshell> /exit
|  Goodbye
```

Like the Python shell, it is great for quick testing

PROGRAM ENTRY POINT

Python

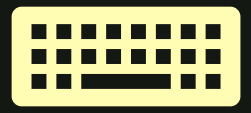
```python
def main():
    print('Hello World!')

if __name__ == '__main__':
    main()
```

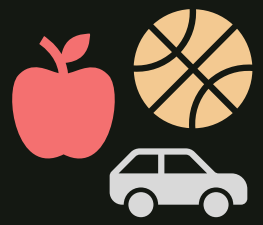
Java

```java
class Main {
    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        System.out.println(x + y);
    }
}
```

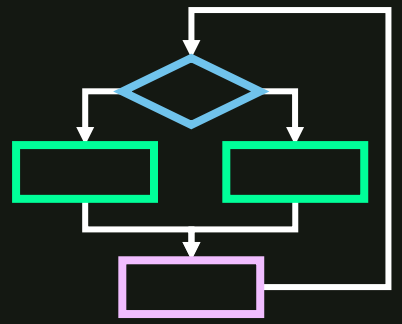
JAVA PROGRAMMING CONSTRUCTS



Console I/O



Classes & Objects



Control Structures

$x = 1$

Variables and Types

$\lambda x.\, \lambda y.\, x + y$

Lambda Expressions

CONSOLE OUTPUT

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World!");
    System.out.print("no newline here!");
    System.out.printf("x = %d", 1 + 2);
  }
}
```

`println` works just like Python's `print`

`print` is similar to `println` but doesn't print a `\n`

`printf` is similar to `print` but allows formatting

CONSOLE INPUT

```java
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int i = sc.nextInt();
    double d = sc.nextDouble();
    String line = sc.nextLine();
    String word = sc.next();
  }
}
```

1. Import the `Scanner` class
2. Create a new `Scanner` object
3. Call one of the `next` methods to obtain the next token from input

STRING FORMATTING

```
>>> f'x = {3:04d}, y = {0.5:.4f}'
'x = 0003, y = 0.5000'
```

Python

```
jshell> String.format("x = %04d, y = %.4f", 3, 0.5)
$1 ==> "x = 0003, y = 0.5000"
```

Java

COMMENTS

```java
public class Main {
    /**
     * A javadoc comment
     * @param args command-line arguments
     *             to this program
     */
    public static void main(String[] args) {
        int x = 1; // a single-line comment
        /*
         * A multi-line
         * comment
         */
    }
}
```

- Javadoc comments are commonly used to generate Java documentation (Javadocs)
- Multi-line comments are opened with `/*` and closed with `*/`

VARIABLES

```
1 public class Main {
2     public static void main(String[] args) {
3         int x; // a variable declaration
4         x = 1; // assignment
5         int y = 2; // variable declaration
6                    // and initializer
7     }
8 }
```

All variables must be declared with a type

PRIMITIVE TYPES

TYPE	LITERAL	EXAMPLES		DESCRIPTION
`byte`	Any int, short or char literal	`1`	`0x6F`	8 bits
`short`	Any int or char literal	`1`	`0x6F`	16-bit integer
`char`	Any int or char literal	`'c'`	`'\u006F'`	16-bit Unicode character
`int`	Any integer in decimal or hexadecimal	`1`	`0x6F`	32-bit integer
`long`	Any integer in decimal or hexadecimal, appended with `L`	`1L`	`0x6FL`	64-bit integer
`float`	Any floating point number (potentially appended with `F`)	`1.5`	`1.5F`	32-bit IEEE-754 floating point number
`double`	Any floating point number (potentially appended with `D`)	`1.5`	`1.5D`	64-bit (double precision) IEEE-754 floating point number
`boolean`	`true` or `false`	`true`	`false`	True or false

TYPECASTING

```java
public class Main {
    public static void main(String[] args) {
        int x = 1;
        double y = (double) x;
        char c = (char) x;
        boolean b = (boolean) x; // error!
    }
}
```

Syntax: (<type>) <expr>

Not all types can be casted to another

Can be unsafe if care is not taken

```
jshell> 1 + 1
$1 ==> 2

jshell> 2 - 3
$2 ==> -1

jshell> 3 * 4
$3 ==> 12

jshell> 9 / 4
$4 ==> 2

jshell> -9 / 4
$5 ==> -2

jshell> 9.0 / 4
$6 ==> 2.25

jshell> 9 / 4.0
$7 ==> 2.25
```

MATHEMATICAL OPERATIONS

Note the difference between integer division and floating point division

```
jshell> 9.0 / 4
$6 ==> 2.25

jshell> 9 / 4.0
$7 ==> 2.25

jshell> 4 % 3
$8 ==> 1

jshell> Math.pow(2, 3)
$9 ==> 8.0

jshell> "a" + "b"
$10 ==> "ab"

jshell> 4 + "ab"
$11 ==> "4ab"
```

MATHEMATICAL OPERATIONS

- Use `Math.pow(a, b)` for exponentiation
- String concatenation only requires one operand to be a `String`

```
jshell> int i = 0;
i ==> 0

jshell> i++
$1 ==> 0

jshell> i
i ==> 1

jshell> ++i
$2 ==> 2

jshell> i
i ==> 2

jshell> i--
$3 ==> 2

jshell> i
```

PRE/POSTFIX IN/DECREMENTATION

MIGHT BE CONFUSING

++: increase by 1 (increment)

--: decrease by 1 (decrement)

Prefix: in/decrease first, then return new value

Postfix: return current value, then in/decrease

```
jshell> ++i
$2 ==> 2

jshell> i
i ==> 2

jshell> i--
$3 ==> 2

jshell> i
i ==> 1

jshell> --i
$4 ==> 0

jshell> i
i ==> 0
```

PRE/POSTFIX IN/DECREMENTATION

MIGHT BE CONFUSING

++: increase by 1 (increment)

--: decrease by 1 (decrement)

Prefix: in/decrease first, then return new value

Postfix: return current value, then in/decrease

```
jshell> true && false // and
$1 ==> false

jshell> true || false // or
$2 ==> true

jshell> true ^ false // xor
$3 ==> true

jshell> !true // not
$4 ==> false

jshell> 1 == 2
$5 ==> false

jshell> 1 != 2
$6 ==> true

jshell> 1 > 3
$7 ==> false
```

LOGICAL OPERATORS

Note the different syntax for and and or

Exclusive OR (XOR): true if and only if exactly one of the operands is true

```
$4 ==> false

jshell> 1 == 2
$5 ==> false

jshell> 1 != 2
$6 ==> true

jshell> 1 > 3
$7 ==> false

jshell> 1 < 3
$8 ==> true

jshell> 2 <= 2
$9 ==> true

jshell> 2 >= 2
$10 ==> true
```

LOGICAL OPERATORS

Other logical operators are the same as in Python

IF-ELSE STATEMENTS

```java
public class Main {
    public static void main(String[] args) {
        int a = 1, b = 1, c;
        if (a != b) {
            c = 2;
        } else if (a > b)
            c = 3;
    }
}
```

Syntax:

`if (<expr>) <statement or block>`

or

`if (<expr>) <statement or block> else <statement or block>`

where `statement` can be any statement (including if-else statements), `block` is multiple statements enclosed in curly braces

WHILE LOOPS

```java
public class Main {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.println(x++);
        }
    }
}
```

Syntax:

`while (<expr>) <statement or block>`

`<statement or block>` is optional

DO-WHILE LOOPS

```java
public class Main {
    public static void main(String[] args) {
        int x = 0;
        do {
            System.out.println(x++);
        } while (x < 10);
    }
}
```

Do first, check later

Syntax:

do <statement or block> while (<expr>)

FOR LOOPS (PRIMITIVE)

```java
public class Main {
    public static void main(String[] args) {
        int x = 0;
        for (int i = 0; i < 10; ++i) {
            System.out.println(i);
        }
    }
}
```

Syntax:

```
for (<init>; <cond>; <update>)
<statement or block>
```

`<init>`, `<cond>`, `<update>` and `<statement or block>` are all optional

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

STORE

x: 0

i: 0

EXECUTION FLOW

<init>

CONSOLE

~ $ java Main

Upon entering the loop, <init> is executed

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Then repeat the following:
evaluate <cond>. If false, exit loop.
otherwise,

STORE

x: 0

i: 0

EXECUTION FLOW

<init>

<cond>

CONSOLE

```
~ $ java Main
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <statement or block>

STORE

x: 0

i: 0

EXECUTION FLOW

<init>

<cond>

<body>

CONSOLE

```
~ $ java Main
0
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <update>

STORE

x: 0

i: 1

EXECUTION FLOW

<init>

<cond>

<body>

<updt>

CONSOLE

```
~ $ java Main
0
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

STORE

x: 0

i: 1

EXECUTION FLOW

<init> <cond>

<cond>

<body>

<updt>

CONSOLE

```
~ $ java Main
0
```

Repeat! Evaluate <cond>. If false, exit loop.
otherwise,

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <statement or block>

STORE

x: 0

i: 1

EXECUTION FLOW

<init> <cond>

<cond> <body>

<body>

<updt>

CONSOLE

```
~ $ java Main
0
1
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <update>

STORE

x: 0

i: 2

EXECUTION FLOW

<init> <cond>

<cond> <body>

<body> <updt>

<updt>

CONSOLE

```
~ $ java Main
0
1
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

STORE

x: 0

i: 2

EXECUTION FLOW

<init> <cond> <cond>
<cond> <body>
<body> <updt>
<updt>

Repeat! Evaluate <cond>. If false, exit loop.
otherwise,

CONSOLE

```
~ $ java Main
0
1
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <statement or block>

STORE

x: 0

i: 2

EXECUTION FLOW

<init> <cond> <cond>
<cond> <body> <body>
<body> <updt>
<updt>

CONSOLE

```
~ $ java Main
0
1
2
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

Execute <update>

STORE

x: 0

i: 3

EXECUTION FLOW

<init> <cond> <cond>
<cond> <body> <body>
<body> <updt> <updt>
<updt>

CONSOLE

```
~ $ java Main
0
1
2
```

FOR LOOPS (PRIMITIVE)

OPERATIONAL SEMANTICS

```
1 public class Main {
2     public static void main(String[] args) {
3         int x = 0;
4         for (int i = 0; i < 10; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```

STORE

x: 0

i: 10

EXECUTION FLOW

<init> <cond> <cond>
<cond> <body> <body>
<body> <updt> <updt>
<updt> ...

CONSOLE

```
6
7
8
9
~ $
```

Eventually come to a point where <cond> evaluates to false, so we exit loop.

FOR LOOPS (ENHANCED)

ALSO KNOWN AS FOR-EACH LOOP

```java
public class Testing {
    public static void main(String[] args) {
        int[] a = new int[] {1, 2, 3, 4, 5};
        for (int i : a) {
            System.out.println(i);
        }
    }
}
```

Syntax:

```
for (<var declarator> : <iterable>) <statement or block>
```

ARRAYS

```java
public class Main {
    public static void main(String[] args) {
        int[] a = new int[10];
        a = new int[] {1, 2, 3, 4, 5};
        String[][] b = new String[5][];
        System.out.println(a[2]); // 3
        System.out.println(a.length) // 5
    }
}
```

Fixed-size mutable sequence of elements

Empty array: `new Type[n]`

Initialized array: `new Type[] {a, b, c}`

Index with nonnegative integers

CLASSES

(ALMOST) EVERYTHING IN JAVA

```java
class Dog {
    String name;
    int id;
    void bark() {
        System.out.printf(
          "%s says woof", name);
    }
}
```

- Blueprint for creating objects
- Contains what instances of the class have (attributes) and can do (methods)
- **All** instances of the class have attributes and methods listed in the class declaration

ATTRIBUTES

```
1 class Dog {
2     String name;
3     int id;
4 }
```

- Describes the data that objects of the class have
- Attributes must be of a fixed type throughout
- Syntax is the same as variable declarations

METHODS

```
1 class User {
2     String username;
3     String password;
4     boolean authenticate(String passwordAttempt) {
5         return passwordAttempt == this.password;
6     }
7 }
```

- Parameter types must be fixed
- Return type must be fixed
- Use `this` to refer to self (`this` does not need to be passed in as a parameter)

"FUNCTIONS"

```java
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        System.out.println(add(a, b));
    }
    public static int add(int a, int b) {
        return a + b;
    }
}
```

`static` methods are class-level methods (to be discussed in the future)

CONSTRUCTORS

```java
class User {
    String username;
    String password;
    User(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

- Name of constructor is the type itself
- No need for constructor to return itself

```java
class Main {
    public static void main(String[] args) {
        User u = new User("Bob", "1234");
        System.out.println(u.authenticate("1235"));
    }
}
```

- Constructors are called using the `new` keyword

LAMBDA EXPRESSIONS

```java
import java.util.stream.IntStream;
class Main {
    public static void main(String[] args) {
        int sumSquares =
            IntStream.rangeClosed(1, 10)
                .map(x -> x * x)
                .filter(x -> x % 2 == 0)
                .reduce((x, y) -> x + y);
        System.out.println(sumSquares);
    }
}
```

Syntax: `(<args...>) -> <expr>`

Parentheses around `<args...>` are optional if there is only one parameter

`map`, `filter` and `reduce` are methods attached to `Streams` (we will discuss in the future)

```java
class Dog {
    static int numOfDogs = 0;
    String name;
    // factory method, aka "named constructor"
    static Dog getBob() {
        return new Dog("Bob");
    }
    Dog(String name) {
        numOfDogs++;
        this.name = name;
    }
}

class Main {
    public static void main(String[] args) {
        Dog bob = Dog.getBob();
        Dog alice = new Dog("Alice");
        System.out.println(Dog.numOfDogs); // 2
        alice.numOfDogs++;
        System.out.println(Dog.numOfDogs); // 3
    }
}
```

CLASS-LEVEL ATTRIBUTES/METHODS

- `static` binds the method/attribute to the class itself
- Can call/refer to it from instance, but there is only one copy of that method/attribute

FINAL VARIABLES/ATTRIBUTES

```java
class Dog {
    static final String SPECIES = "Dog";
    final String name;
    Dog(final String name) {
        this.name = name;
    }
}
```

- `final` makes a variable/attribute immutable
- We use `static` and `final` attributes to store constants

ABSTRACT CLASSES AND INTERFACES

```
1  abstract class HTMLElement {
2      DOM d;
3      int id;
4      // concrete method
5      public String describe() {
6          return String.format("%s: %d", d, id);
7      }
8      // abstract method with no implementation
9      abstract void render();
10 }
```

`abstract` classes are “general” classes that define what its subclasses are like

You **cannot directly instantiate abstract classes** because there are unimplemented methods

ABSTRACT CLASSES AND INTERFACES

```java
interface Renderable {
    // the only attributes interfaces can
    // have are constants
    public static final String FORMAT =
        "%s: %d";
    // all classes that use this interface
    // will have this method
    void render();
}
```

- `interfaces` are **contracts** between users and implementing classes, saying that all classes that implement this interface will **have the methods specified in it**
- Similarly, you **cannot directly instantiate** instances of interfaces
- Interfaces (should) only have abstract methods
- Interfaces cannot have attributes except constants

ACCESS MODIFIERS

```java
public class Button {
    private final DOM d;
    private final int id;
    private Geometry g;
    public Button(DOM d, int id) {
        this.d = d;
        this.id = id;
        this.g = getAvailablePosition(d);
    }
    public Pair<DOM, Integer> describe() {
        return new Pair<>(d, id);
    }
    private Geometry getAvailablePosition(DOM d) {
        // ...
    }
}
```

"You can talk to me without knowing what underwear I'm wearing"

Assign **access modifiers** to attributes/methods (and even classes) to remove them from part of the abstraction

MODIFIER	ACCESSIBILITY
`private`	Only the class itself
`<nothing>`	Package private, i.e. only members of the same package
`protected`	Members of same package and subclasses/subinterfaces
`public`	Anyone

INHERITANCE

Letting one class/interface inherit from another

```
1 interface IntA {  }
2 interface IntB extends IntA {  } // ok
3 abstract class ClsC {  }
4 class ClsD extends ClsC implements IntA {  } // ok
5 abstract class ClsE extends ClsD, ClsC {  } // not ok!
6 interface IntF extends ClsE {  } // not ok!
```

- an interface can only extend interfaces (any number)
- a class (abstract or otherwise) can only extend one other class but can implement any number of interfaces
- an abstract class cannot extend a concrete (normal) class

OVERRIDING VS OVERLOADING

```
1  class Jetliner {
2    int cabinPressure(int altitude) {
3      return altitude * 5000;
4    }
5  }
6  class OldJetliner extends Jetliner {
7    @Override
8    int cabinPressure(int altitude) {
9      return (int) (super
10       .cabinPressure(altitude) * 0.8);
11   }
12 }
```

- Method overriding: providing an alternative implementation for a method defined in a superclass / interface
- When overriding, number and types of parameters and return type must be the same (effectively*)
- Use `super` to refer to superclass definition of attribute/method (like `super()` in Python)

*See the Java Language Specification for the exact definition of override-equivalence. Most notably, the return type of the overriding method can be a subtype of that of the overridden method

OVERRIDING VS OVERLOADING

```
1  class Dog {
2      String name;
3      Dog(String name) {
4          this.name = name;
5      }
6      Dog() {
7          this.name = "Dog";
8      }
9  }
```

- Creating a method (or constructor) with the **same name** but **different method signature** (number of parameters and their types)
- Two or more methods of the same name and signature cannot be defined in the same class (otherwise there will be ambiguity)
- If a method has the same name and signature as one in a superclass, it must be overriding

JAVA PROGRAMMING CONSTRUCTS

AND MANY MORE!

Java supports more features such as:

- Other OO constructs (which we will see)
- Type Parameters (which we will see)
- Annotations
- Method references
- Functional Interfaces
- Reflection
- ...

KEY POINT #5

Most programming constructs are semantically similar but syntactically different to their equivalents in Python; you'll get used to them by practice

JAVA API

APPLICATION PROGRAMMING INTERFACE

java.lang.Object

java.lang.Math

java.lang.String

java.util.Scanner

WORKED EXAMPLE

You are implementing a simple Point-of-Sales system that accepts incoming orders and obtains the total price of all orders

Write a POS class and Order class to support this system. A POS system should be able to print all orders and obtain the total price of all orders. An order is a customizable burger that can be changed.

WORKED SOLUTION

```
1 interface Order {
2     abstract int getPrice();
3 }
```

Create the abstraction first

```java
class AlaCarteBurger implements Order {
    private String ingredients;
    public AlaCarteBurger(String ingredients) {
        this.ingredients = ingredients;
    }
    public int getPrice() {
        int price = 0;
        for (int i = 0; i < ingredients.length(); ++i) {
            if (ingredients.charAt(i) == 'B')
                price += 50;
            // ...
            else if (ingredients.charAt(i) == 'M')
                price += 90;
        }
        return price;
    }
    @Override
    public String toString() {
        return String.format("Burger %s", ingredients);
    }
}
```

WORKED SOLUTION

Then create the concrete class

```
class POS {
    private static final String ITEM_FORMAT = "%d. %s: %s\n";
    private static String formatPrice(int price) {
        return String.format("$%d.%02d", price / 100, price % 100);
    }
    private static Order[] expandOrderArray(Order[] arr) {
        Order[] res = new Order[arr.length * 2];
        for (int i = 0; i < arr.length; ++i) {
            res[i] = arr[i];
        }
        return res;
    }
    private int i = 0;
    private Order[] orders = new Order[10];
    public void addOrder(Order o) {
        if (i == orders.length) {
            orders = expandOrderArray(orders);
        }
        orders[i++] = o;
    }
    public void printOrders() {
        int totalPrice = 0;
        for (int j = 0; j < i; ++j) {
            int p = orders[j].getPrice();
```

```
9               res[i] = arr[i];
10          }
11          return res;
12      }
13      private int i = 0;
14      private Order[] orders = new Order[10];
15      public void addOrder(Order o) {
16          if (i == orders.length) {
17              orders = expandOrderArray(orders);
18          }
19          orders[i++] = o;
20      }
21      public void printOrders() {
22          int totalPrice = 0;
23          for (int j = 0; j < i; ++j) {
24              int p = orders[j].getPrice();
25              System.out.printf(ITEM_FORMAT, j + 1, orders[j], formatPrice(p));
26              totalPrice += p;
27          }
28          System.out.printf("Total price: %s\n", formatPrice(totalPrice));
29      }
30  }
```

WORKED EXAMPLE

Now you realize there are also meals in this system. A meal is also an order, but consists of a burger and some other items. There are three sizes of meals, small, medium and large.

```java
class Meal implements Order {
    public static final String SMALL = "Small";
    public static final String MEDIUM = "Medium";
    public static final String LARGE = "Large";
    private AlaCarteBurger burger;
    private String size;
    public Meal(AlaCarteBurger burger, String size) {
        this.burger = burger;
        this.size = size;
    }
    public int getPrice() {
        if (size == LARGE)
            return burger.getPrice() + 300;
        if (size == MEDIUM)
            return burger.getPrice() + 200;
        return burger.getPrice() + 100;
    }
    @Override
    public String toString() {
        return String.format("%s meal with %s", this.size, this.burger);
    }
}
```

Thanks to good design, you only need to create one class, and you're done!

CONTENTS

- Course Administration
- What and Why of Streaming
- Java Fundamentals

KEY POINTS

- Instead of pulling data from a data source, **subscribe** to a data source to receive real-time changes
- Everything that happens in an application can be described by **events**, use an event bus/store to decouple producers and consumers of **event streams**
- **Distribute** event stream processing into **multiple nodes** in **multiple computing clusters** for **high throughput**
- Java is a **compiled** and **statically-typed** language
- Most programming constructs are semantically similar but syntactically different to their equivalents in Python; you'll get used to them by practice