CS106 Data Structure & Java Fundamentals

BitTiger丨来自硅谷的终身学习平台

Claire & Bob

BITTIGER

版权声明

BITTIGER

Copyright Policy

All content included on the Site or third-party platforms as part of the class, such as text, graphics, logos, button icons, images, audio clips, video clips, live streams, digital downloads, data compilations, and software, is the property of BitTiger or its content suppliers and protected by copyright laws.

Any attempt to redistribute or resell BitTiger content will result in the appropriate legal action being taken.

We thank you in advance for respecting our copyrighted content.For more info see https://www.bittiger.io/termsofuse and https://www.bittiger.io/termsofservice

Project - VehicleBuilderService

Project - VehicleBuilderService

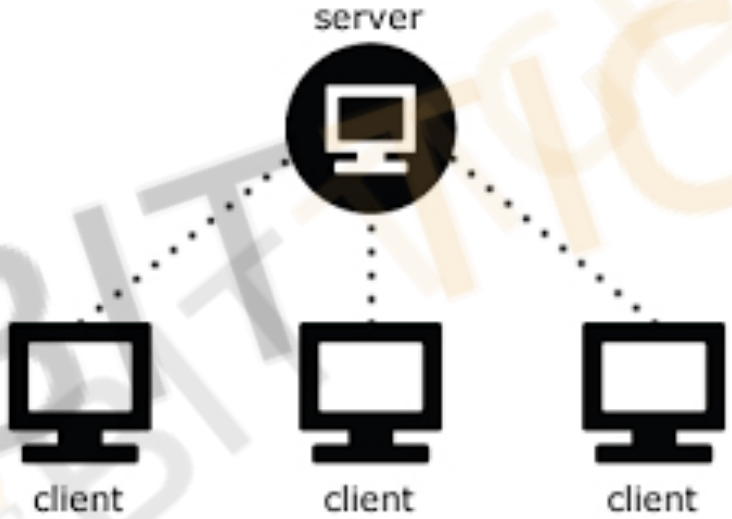
- Company A wants to build a online vehicle config service
- Features
  - create/update/delete/display vehicles
  - Company A stores and manages the data
- Size of the whole service might be huge
- Interact with internal data storage of Company A

Project - VehicleBuilderService

- Client-Server Architecture
- Interaction with User
  - Command Line Interface

Client-Server

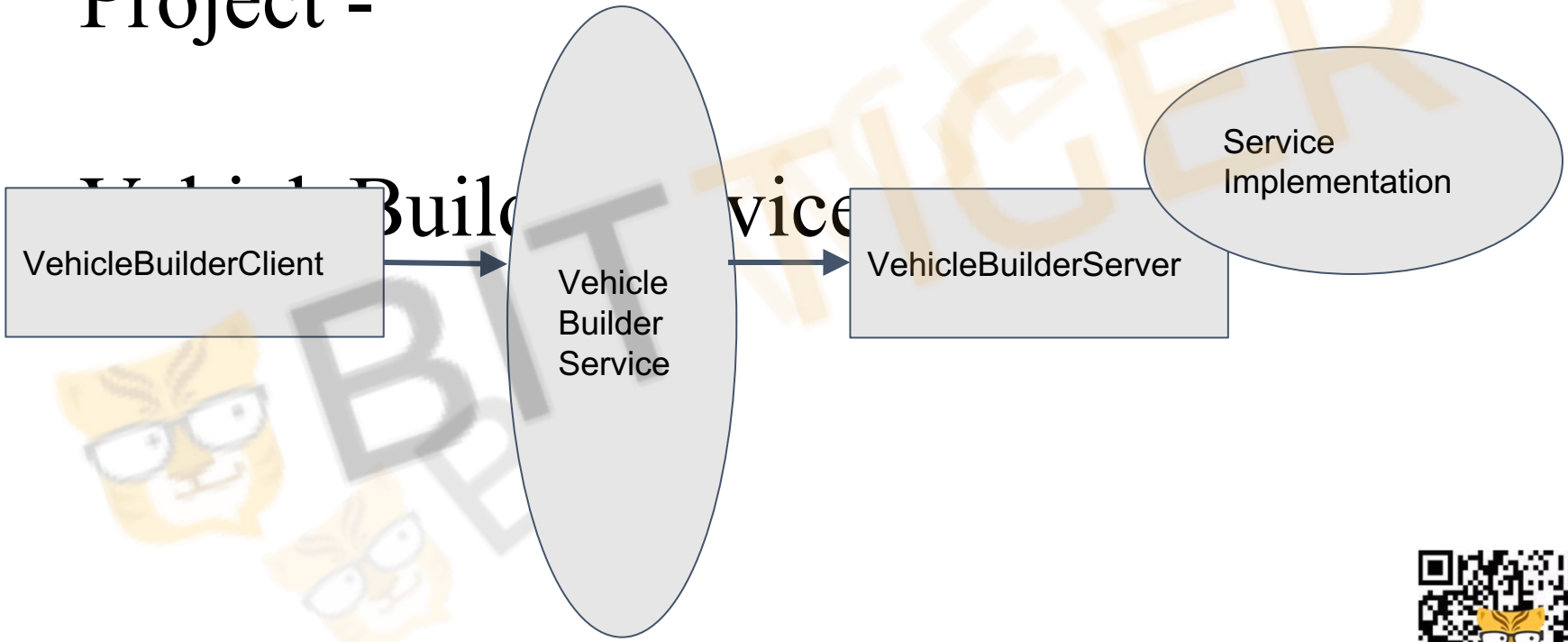
Archite



Client-Server Architecture

- Usually on separate hardware
- Communicate over computer network
- Client
  - Make request
  - Don't share resource with other clients
- Server
  - Centralized system that serves multiple clients
  - Provide service

Project -


VehicleBuilderClient
Vehicle
Builder
Service
VehicleBuilderServer
Service
Implementation

Socket

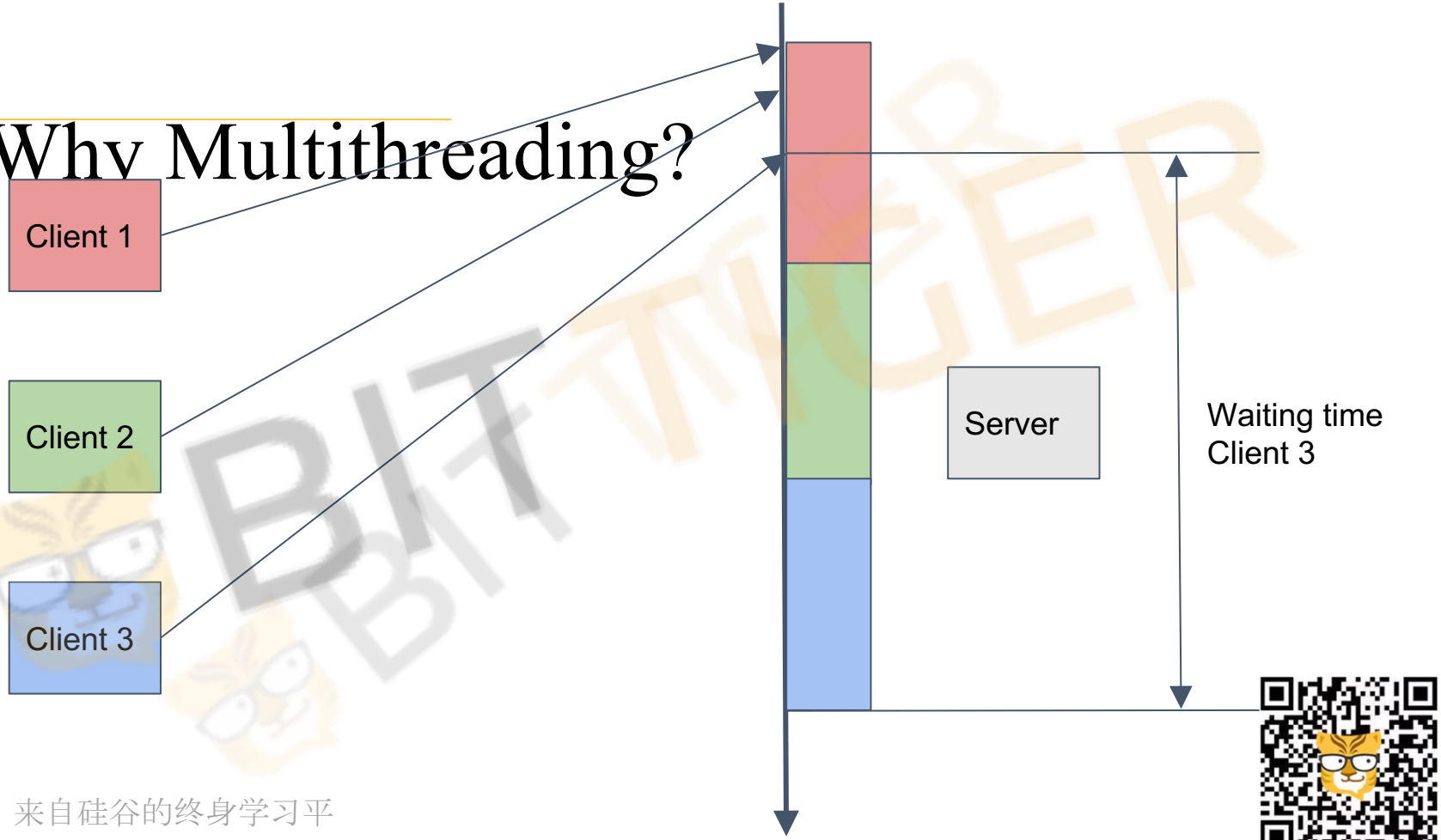
- Endpoint of a two-way communication programs
  - Ip address, port
- Between two programs running on the network
- java.net
  - Socket
  - ServerSocket

Socket

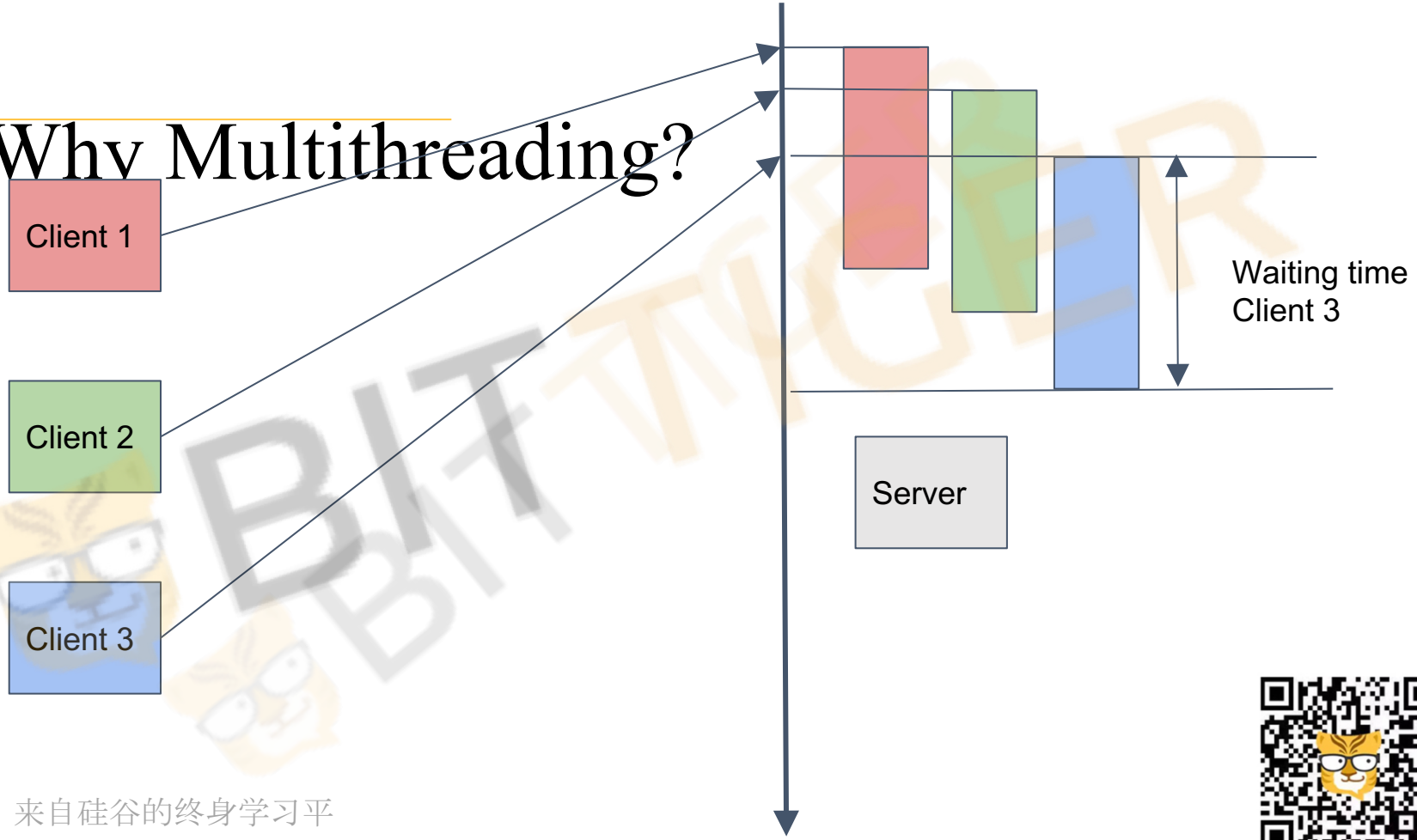
- Server initiates ServerSocket and waiting for requests
- Client initiates Socket, connecting to server by specifying server name and port
- Server accepts the connection
- Can start two-way communication

Any Problems?

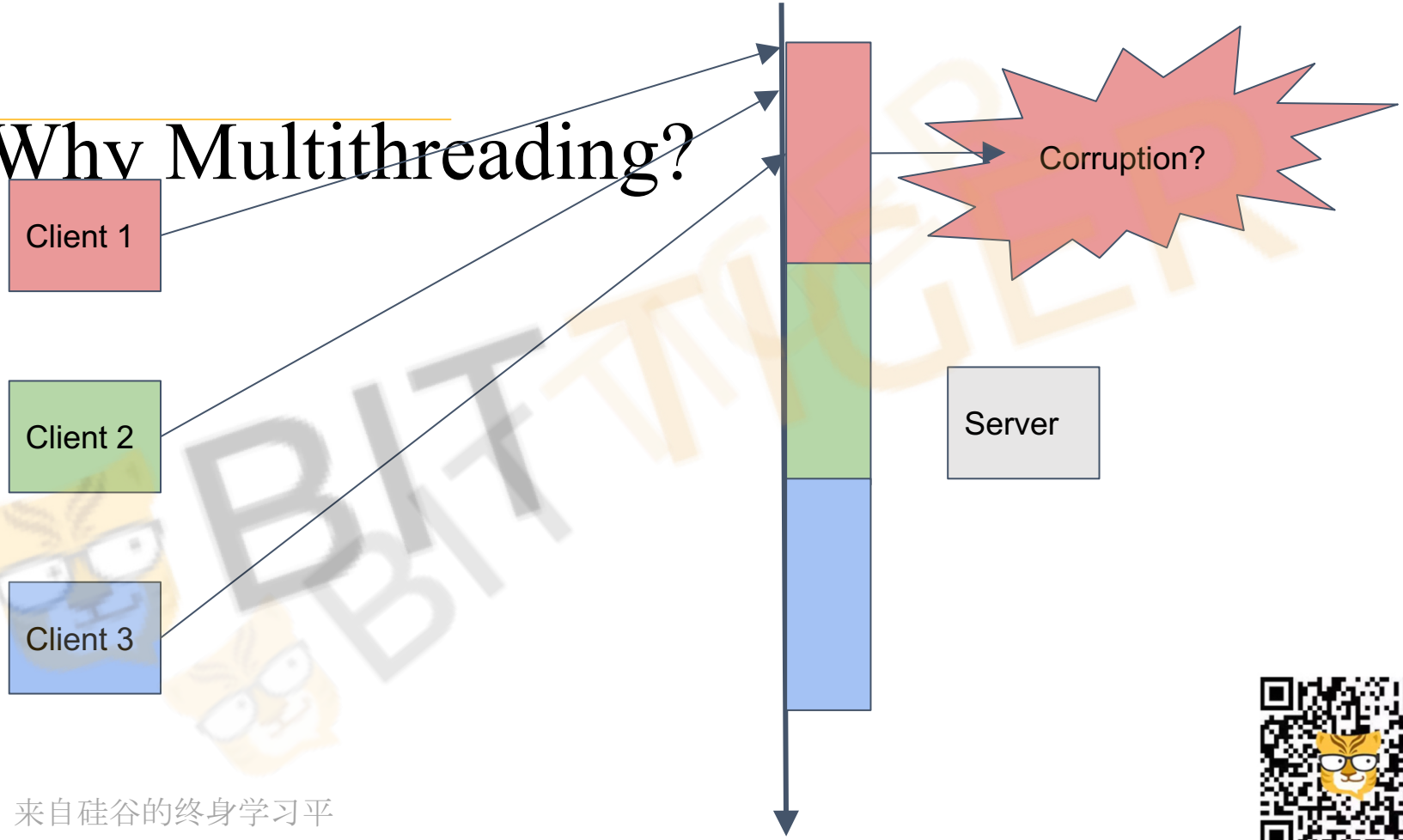
Why Multithreading?



Why Multithreading?



Why Multithreading?



Java 8 & 9 New Features

Java 8

- **Comparator**
- **Lambda Expression** “->”
- **Functional Interface** *Predicates, Comparator, Runnable*
- **Method References** “ :: “
- **Default Interface**
- **Streams** *map/reduce/count/filter/flatMap/Collectors*
- **Map**

Comparator

- Comparator?
- Comparable?

Comparator

- Comparator<T>
  - int compare(T o1, T o2);
  - boolean equals(Object obj); // optional
- Comparable<T>
  - int compareTo(T o)

-1 or less ⇒ “<”
0 ⇒ “==”
1 or greater ⇒ “>”

Comparable

- Description: A class instance is able to compare with other instance.

Comparable

```
public class Employee implements Comparable<Employee> {

    String name;
    int age;

    @Override
    public int compareTo(Employee that) {
        return this.age - that.age;
    }
}
```

Comparator

- Description: a kind of class which could compare two instances.

Comparator

```
public class AgeComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getAge() -  e2.getAge();
    }
}
```

Comparator -- Java 8

- Methods
  - reversed()
  - thenComparing()
- Static Method
  - nullFirst()
  - nullLast()
  - naturalOrder()
  - reverseOrder()

Comparator -- Java 8

```
Employee e1 = new Employee("Xiuting", 16);
Employee e2 = new Employee("Mingche", 65);
Employee e3 = new Employee("Qinyuan", 50);
Employee e4 = new Employee("Chang", 18);

List<Employee> employeeList = new ArrayList<>();
employeeList.add(e1);
employeeList.add(e2);
employeeList.add(e3);
employeeList.add(e4);

Collections.sort(employeeList, new AgeComparator());
```

>> [Xiuting, 16, Chang, 18, Qinyuan, 50, Mingche, 65]

Comparator

```
Collections.sort(list, new AgeComparator());
// [[Xiuting, 16], [Chang, 18], [Qinyuan, 50], [Mingche, 65]]

Collections.sort(list, new AgeComparator().reversed());
// [[Mingche, 65], [Qinyuan, 50], [Chang, 18], [Xiuting, 16]]
```

Comparator

```
Collections.sort(list, new NameComparator());
// [[Chang, 18], [Mingche, 65], [Qinyuan, 50], [Xiuting, 16]]

Collections.sort(list, new AgeComparator().thenComparing(new
NameComparator()));
// [[Xiuting, 16], [Chang, 50], [Mingche, 50], [Qinyuan, 50]]

Collections.sort(list, Comparator.nullsLast(new AgeComparator()));
// [[Xiuting, 16], [Mingche, 50], [Qinyuan, 50], [Chang, 50], null]
```

Lambda Expression

An anonymous function -- no name, in-line

A -----> B

```
str -> str.isEmpty()

(p1, p2) -> p1.getAge() - p2.getAge()

() -> System.out.println("hello, world!");
```

Lambda Expression

```
Employee e1 = employeeFactory.create("Xiuting", 16);
Employee e4 = employeeFactory.create("Mingche", 65);
Employee e3 = employeeFactory.create("Qinyuan", 50);
Employee e2 = employeeFactory.create("Chang", 18);

List<Employee> employeeList = new ArrayList<>();
employeeList.add(e1);
employeeList.add(e2);
employeeList.add(e3);
employeeList.add(e4);

Collections.sort(employeeList, new EmployeeAgeComparator());
System.out.println(employeeList);
```

Lambda Expression

```
class EmployeeAgeComparator implements Comparator<Employee> {

  @Override
  public int compare(Employee o1, Employee o2) {
    return o1.getAge() - o2.getAge();
  }
}

Collections.sort(employeeList, new EmployeeAgeComparator());
System.out.println(employeeList);
```

>> [Xiuting, 16, Chang, 18, Qinyuan, 50, Mingche, 65]

Lambda Expression

```
class EmployeeAgeComparator implements Comparator<Employee> {

  @Override
  public int compare(Employee o1, Employee o2) {
    return o1.getAge() - o2.getAge();
  }
}
```

Lambda Expression

```
class EmployeeAgeComparator implements Comparator<Employee> {

  (Employee o1, Employee o2) {
    return o1.getAge() - o2.getAge();
  }
}
```

Lambda Expression

```
Comparator<Employee> comp = {

  (Employee o1, Employee o2) {
    return o1.getAge() - o2.getAge();
  }
}
```

Lambda Expression

```
Comparator<Employee> comp = (o1, o2) { o1.getAge() - o2.getAge() }
```

Lambda Expression

```
Comparator<Employee> comp = (o1, o2) -> { o1.getAge() - o2.getAge() }
```

Lambda Expression

```
Comparator<Employee> comp = (o1, o2) -> o1.getAge() - o2.getAge()
```

```
Collections.sort(employeeList, (o1, o2) -> o1.getAget() - o2.getAge());
System.out.println(employeeList);
```

```
>> [Xiuting, 16, Chang, 18, Qinyuan, 50, Mingche, 65]
```

Functional Interface

```
Comparator<Employee> comp = (o1, o2) -> o1.getAge() - o2.getAge();

Predicate<String> pre = str -> str.isEmpty();

Runnable run = () -> System.out.println("Hello, World");

Function<Integer, Integer> resize = (a) -> {int b = 2 * a; return b;};
```

Functional Interface

```
Comparator<Employee> comp = (o1, o2) -> o1.getAge() - o2.getAge();

Predicate<String> pre = str -> str.isEmpty();

Runnable run = () -> System.out.println("Hello, World");

Function<Integer, Integer> resize = (a) -> {int b = 2 * a; return b;};
```

Any interface with one abstract method could be used as functional interface.

Functional Interface

```
@FunctionalInterface
public interface Runnable {
        public abstract void run();
}

@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);
}

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

Functional Interface

```
@FunctionalInterface
public interface EmployeeFactory {
        Employee create(String name, int age);
}

EmployeeFactory employeeFactory = (name, age) -> new Employee(name, age);

Employee e1 = employeeFactory.create("Xiuting", 16);
```

Common Functional Interface

- Supplier<R>
- Comsumer<T>
- Function<T, R>
- Predicate
- BinaryOperator
- Runnable
- Comparator<T>
-

Method Reference

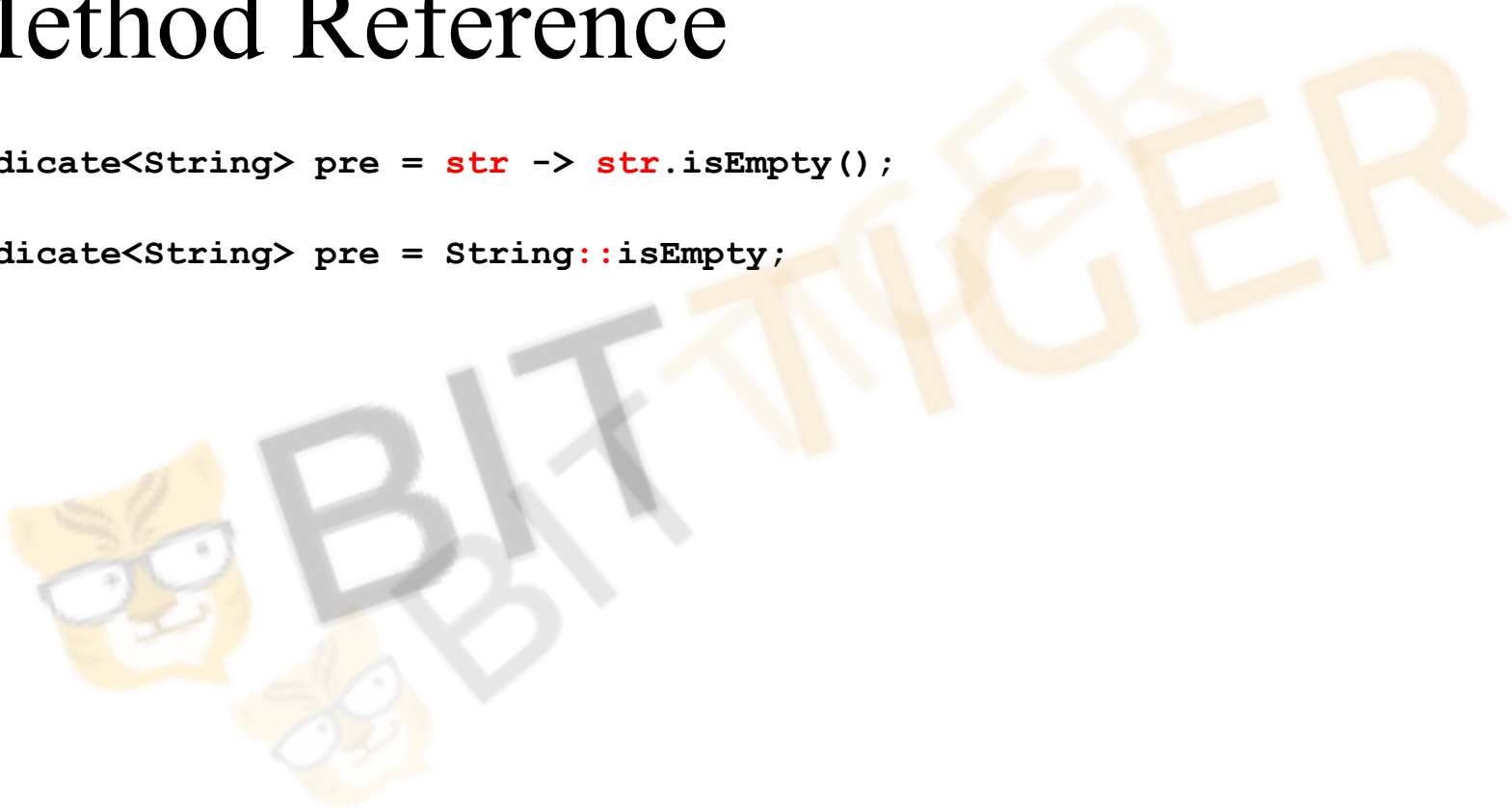
```
Predicate<String> pre = str -> str.isEmpty();
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;
```



Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;

Consumer<String> hello = str -> System.out.println(str);
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;

Consumer<String> hello = str -> System.out.println(str);

Consumer<String> hello = System.out::println;
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;

Consumer<String> hello = str -> System.out.println(str);

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

EmployeeFactory<Employee> factory = (name, age) -> new Employee(name, age);
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;

Consumer<String> hello = str -> System.out.println(str);

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

EmployeeFactory<Employee> factory = (name, age) -> new Employee(name, age);

EmployeeFactory<Employee> factory = Employee::new;
```

Method Reference

```
Predicate<String> pre = str -> str.isEmpty();

Predicate<String> pre = String::isEmpty;

Consumer<String> hello = str -> System.out.println(str);

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

EmployeeFactory<Employee> factory = (name, age) -> new Employee(name, age);

EmployeeFactory<Employee> factory = Employee::new;
```

IntelliJ

Practice

```
public class MyUpperCase implements Function<String, String>
     @Override
     String apply(String input) {
          return input.toUpperCase();
     }
}

public void converter(String str, Function<String, String> func) {
     return func.apply(str);
}
```

```
converter("hello", String::toUpperCase);
```

Default Methods

```
@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);
}
```

Default Methods

```
@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
```

New Feature: default methods **implementation** in interface

Default Methods

```
interface Person {

  String getName();

  default String greeting() {
    return String.format("Hello, I am %s", getName());
  }
}
```

Default Methods

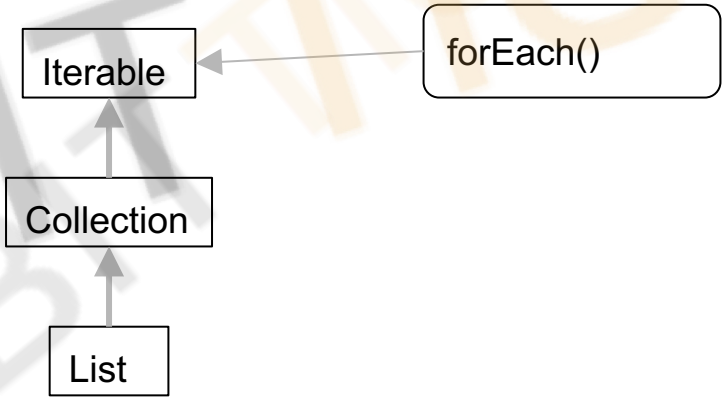
```
public class Employee implements Person {

  private String name ;
  private int age;

  public Employee(String name, int age) {
    this.name = name;
    this.age = age;
  }

  @Override
  public String getName() {
    return name;
  }
}

System.out.println(e1.greeting());
```

```
interface Person {

  String getName();

  default String greeting() {
    return String.format("Hello, I am %s", getName());
  }
}
```

>> Hello, I am Xiuting

Default Method

- **Purpose:** This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8



Stream

```
Employee e1 = employeeFactory.create("Xiuting", 16);
Employee e4 = employeeFactory.create("Mingche", 65);
Employee e3 = employeeFactory.create("Qinyuan", 50);
Employee e2 = employeeFactory.create("Chang", 18);

List<Employee> employeeList = new ArrayList<>();
employeeList.add(e1);
employeeList.add(e2);
employeeList.add(e3);
employeeList.add(e4);
```

1. Select all adult employees
2. Sort them by age
3. Convert their name to uppercase
4. Join them using # and output one string

Stream

```
Collections.sort(employeeList, (o1, o2) -> o1.getAge() - o2.getAge());
StringBuilder sb = new StringBuilder();
for (int i = 0; i < employeeList.size(); i++) {
  Employee emp = employeeList.get(i);
  if (emp.getAge() >= 18) {
    sb.append(emp.getName().toUpperCase());
    if (i != employeeList.size() - 1) {
      sb.append("#");
    }
  }
}
System.out.println(sb.toString())
```

```
>> CHANG#QINYUAN#MINGCHE
```

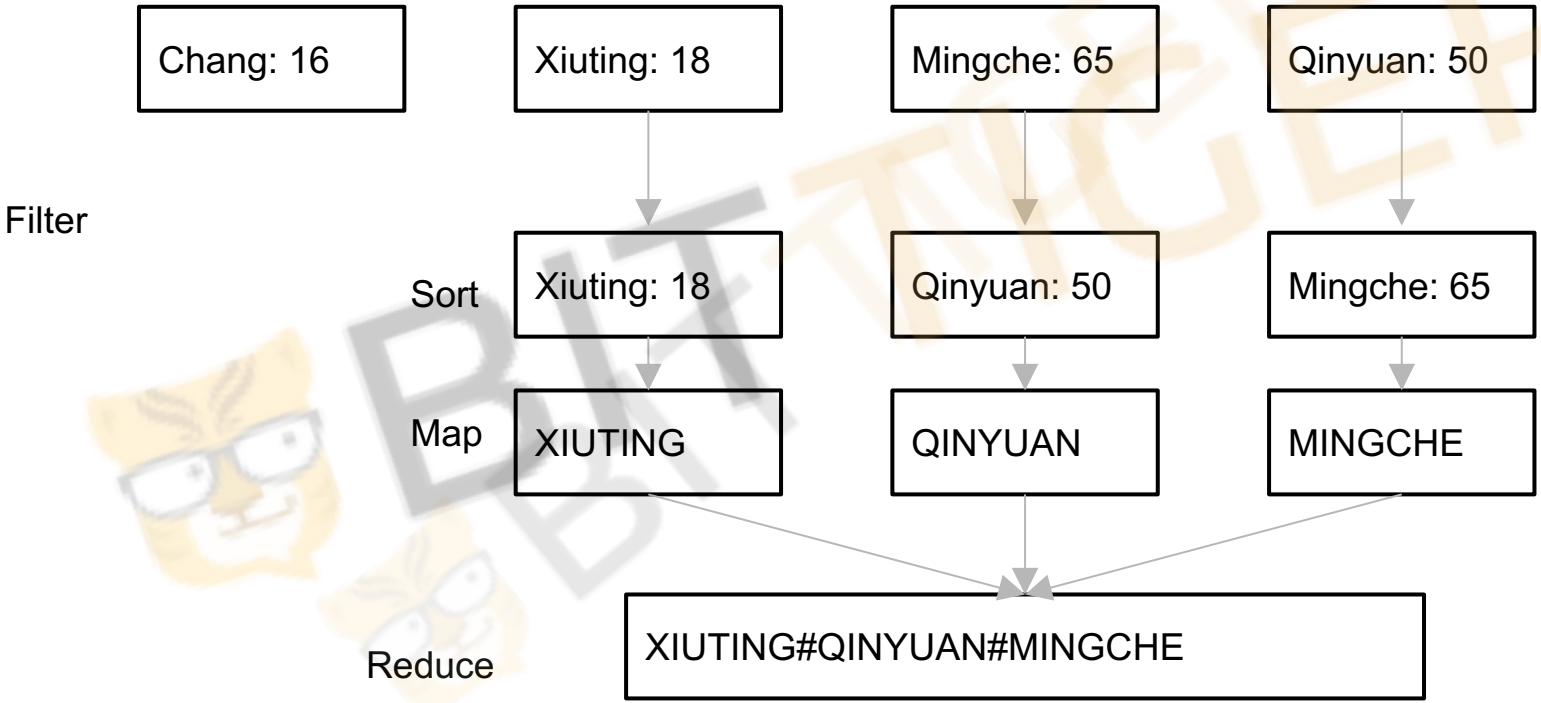
Stream

```
Optional<String> longName = employeeList.stream()
    .filter((e) -> e.getAge() >= 18)
    .sorted((o1, o2) -> o1.getAge() - o2.getAge())
    .map(e -> e.getName().toUpperCase())
    .reduce((o1, o2) -> o1 + "#" + o2);
System.out.println(longName.get());
```

```
>> CHANG#QINYUAN#MINGCHE
```

Stream

```
Optional<String> longName = employeeList.stream()
        .filter((e) -> e.getAge() >= 18)
        .sorted((o1, o2) -> o1.getAge() - o2.getAge())
        .map(e -> e.getName().toUpperCase())
        .reduce((o1, o2) -> o1 + "#" + o2);
```



Stream

```
Optional<String> longName = employeeList.stream()
        .filter((e) -> e.getAge() >= 18)
        .sorted((o1, o2) -> o1.getAge() - o2.getAge())
        .map(e -> e.getName().toUpperCase())
        .reduce((o1, o2) -> o1 + "#" + o2);
```

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    Stream<T> filter(Predicate<? super T> predicate);

    Stream<T> sorted(Comparator<? super T> comparator);

    <R> Stream<R> map(Function<? super T, ? extends R> mapper);

    Optional<T> reduce(BinaryOperator<T> accumulator);

}
```

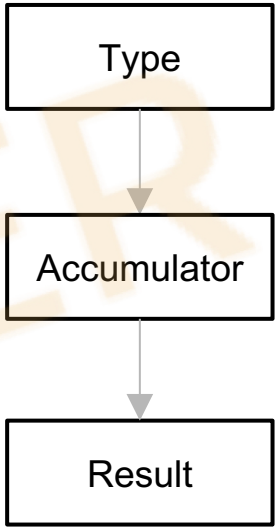
Stream

T: type
A: accumulator
R: result

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    long count();

    Stream<T> distinct();

    void forEach(Consumer<? super T> action);

    <R, A> R collect(Collector<? super T, A, R> collector);

}
```

Stream: Collectors

```
List<String> nameList = employeeList.stream()
      .filter((e) -> e.getAge() >= 18)
      .sorted((o1, o2) -> o1.getAge() - o2.getAge())
      .map(e -> e.getName().toUpperCase())
      .collect(Collectors.toList());
 System.out.println(nameList);
```




```
>> [CHANG, QINYUAN, MINGCHE]
```

Stream: Collectors

```
String longName2 = employeeList.stream()
      .filter((e) -> e.getAge() >= 18)
      .sorted((o1, o2) -> o1.getAge() - o2.getAge())
      .map(e -> e.getName().toUpperCase())
      .collect(Collectors.joining("#"));
 System.out.println(longName2);
```

```
>> CHANG#QINYUAN#MINGCHE
```

Stream: Collectors

```
int ageSum = employeeList.stream()
    .collect(Collectors.summingInt(Employee::getAge));
System.out.println(aveAge)
```

```
>> 149
```

Stream: Collectors

```
IntSummaryStatistics summary = employeeList.stream()
    .map(Employee::getAge)
    .collect(Collectors.summarizingInt(p -> p));

System.out.println(summary.getAverage());
System.out.println(summary.getCount());
System.out.println(summary.getMax());
System.out.println(summary.getMin());
System.out.println(summary.getSum());
```

```
>> 37.25
>> 4
>> 65
>> 16
>> 149
```

Stream: Practice

```
Employee e1 = employeeFactory.create("Xiuting", 16);
Employee e4 = employeeFactory.create("Mingche", 65);
Employee e3 = employeeFactory.create("Qinyuan", 50);
Employee e2 = employeeFactory.create("Chang", 18);

List<Employee> employeeList = new ArrayList<>();
employeeList.add(e1);
employeeList.add(e2);
employeeList.add(e3);
employeeList.add(e4);
```

Get the name of youngest employee!

Stream: Practice

```
employeeList.stream()
    .min((o1, o2) -> o1.getAge() - o2.getAge()))
    .ifPresent(System.out::println);
```

>> Xiuting, 16

Java8: Map

```
Map<String, Integer> counters = new HashMap<>();
for (Employee e : employeeList) {
  Integer count = counters.get(e.getName());
  if (count != null) {
    counters.put(e.getName(), count + 1);
  } else {
    counters.put(e.getName(), 1);
  }
}
System.out.println(counters);
```

Java8: Map

```
Map<String, Integer> counters = new HashMap<>();
for (Employee e : employeeList) {
  counters.computeIfPresent(e.getName(), (k, v) -> v + 1);
  counters.putIfAbsent(e.getName(), 1);
}
System.out.println(counters);
```

Java8: Map

```
Map<String, Integer> counters = new HashMap<>();
for (Employee e : employeeList) {
  counters.computeIfPresent(e.getName(), (k, v) -> v + 1);
  counters.putIfAbsent(e.getName(), 1);
}
System.out.println(counters);

counters.getOrDefault("Xiuting", 0);
```

Java 8: other updates

- Performance improvement: HashMap, link ⇒ Tree
- Optional Class
  - Optional<String> str = …
  - str.get()
  - str.ifPresent(Consumer<String> com)
  - str.isPresent();
- java.util.Time
  - from Joda time

Summary

- **Comparable<T>**: Ability to compare with others
- **Comparator<T>**: A new data type, compare two, Java 8 comparator
- **Lambda**: (arguments) -> { statements};
- **Functional Interfaces**: Comparator, Function, Predicates
- **Method Reference:** Class::method
- **Default Method**: implementation in Interface.
- **Stream**: map, reduce, sort, filter.
- **Collectors**: Joining, toList, summingInt

Java 9

- **Private method in interface**
- **Java 9 shell**
- **Reactive streams**
- **...**