PA2

```
import java.util.Scanner;
import java.util.function.Supplier;
import java.util.Random;
import java.util.stream.Stream;

class Main {
private static final Random RNG_REST = new Random(3L);
private static final Random RNG_REST_PERIOD = new Random(4L);
private static final double SERVER_REST_RATE = 0.1;

static double genRestPeriod() {
    return -Math.log(RNG_REST_PERIOD.nextDouble()) / SERVER_REST_RATE;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    ImList<Pair<Double,Supplier<Double>>> inputTimes =
        new ImList<Pair<Double,Supplier<Double>>>();

    int numOfServers = sc.nextInt();
    int qmax = sc.nextInt();
    int numOfCustomers = sc.nextInt();
    double probRest = sc.nextDouble();

    inputTimes = new ImList<Pair<Double,Supplier<Double>>>(
            Stream.<Pair<Double, Supplier<Double>>>generate(() ->
                new Pair<Double, Supplier<Double>>(
                    sc.nextDouble(), () -> sc.nextDouble()))
            .limit(numOfCustomers)
            .toList());

    Supplier<Double> restTimes = () ->
```

```
        RNG_REST.nextDouble() < probRest ? genRestPeriod() : 0.0;

    Simulator sim = new Simulator(numOfServers, qmax, inputTimes, restTimes);
    System.out.println(sim.simulate());
    sc.close();
}
}
```

```
import java.util.function.Supplier;

class ServerBalancer {
    private final int numOfServers;
    private final int qmax;
    private final ImList<Server> servers;
    private final Supplier<Double> restTimes;

    ServerBalancer(int numOfServers, int qmax, Supplier<Double> restTimes) {
        this.numOfServers = numOfServers;
        this.qmax = qmax;
        this.restTimes = restTimes;

        ImList<Server> servers = new ImList<Server>();

        for (int i = 0; i < this.numOfServers; i++) {
            servers = servers.add(new Server(i + 1, this.qmax, this.restTimes));
        }

        this.servers = servers;
    }
}
```

```
import java.util.Comparator;

class EventComparator implements Comparator<Event> {
    public int compare(Event e1, Event e2) {
        if (e1.getEventTime() == e2.getEventTime()) {
            if (e2.getPriority() == e1.getPriority()) {
                return e1.getCustomer().getCustomerNumber()
                        - e2.getCustomer().getCustomerNumber() > 0 ? 1 : -1;
            }
            return e2.getPriority() - e1.getPriority() > 0 ? 1 : -1;
        } else {
            return e2.getEventTime() - e1.getEventTime() > 0
                    ? -1
                    : 1;
        }
    }
}
```

```java
abstract class Event {
    protected final Customer customer;
    protected final int priority;
    protected final boolean isTerminalEvent;
    protected final double eventTime;
    protected final boolean readyToExecute;
    protected static final int LOW_PRIORITY = 0;
    protected static final int MID_PRIORITY = 1;
    protected static final int HIGH_PRIORITY = 2;

    abstract Pair<Event, ServerBalancer> getNextEvent(ServerBalancer serverBalancer);

    double getEventTime() {
        return this.eventTime;
    }

    protected Customer getCustomer() {
        return this.customer;
    }

    protected boolean isTerminalEvent() {
        return this.isTerminalEvent;
    }

    protected int getPriority() {
        return this.priority;
    }

    String getFormattedEventTime() {
        return String.format("%.3f", this.eventTime);
    }

    void maybePrintEvent() {
        if (this.readyToExecute && !this.isTerminalEvent) {
            System.out.println(this);
        }
    }

    boolean isReadyToExecute() {
        return this.readyToExecute;
    }
}
```

```java
package java.util;

import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;
import java.util.stream.Stream;
```

```java
public final class Optional<T> {
    private static final Optional<?> EMPTY = new Optional<>();
    private final T value;


    private Optional() {
        this.value = null;
    }

    public static<T> Optional<T> empty() {
        @SuppressWarnings("unchecked")
        Optional<T> t = (Optional<T>) EMPTY;
        return t;
    }

    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }
    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }

    public static <T> Optional<T> ofNullable(T value) {
        return value == null ? empty() : of(value);
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    public boolean isPresent() {
        return value != null;
    }

    public boolean isEmpty() {
        return value == null;
    }

    public void ifPresent(Consumer<? super T> action) {
        if (value != null) {
            action.accept(value);
        }
    }

    public void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction) {
        if (value != null) {
            action.accept(value);
        } else {
            emptyAction.run();
        }
    }


    public Optional<T> filter(Predicate<? super T> predicate) {
        Objects.requireNonNull(predicate);
        if (!isPresent()) {
            return this;
        } else {
            return predicate.test(value) ? this : empty();
        }
    }


    public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {
        Objects.requireNonNull(mapper);
        if (!isPresent()) {
            return empty();
        } else {
            return Optional.ofNullable(mapper.apply(value));
        }
    }

    public <U> Optional<U> flatMap(Function<? super T, ? extends Optional<? extends U>> mapper) {
        Objects.requireNonNull(mapper);
        if (!isPresent()) {
            return empty();
        } else {
            @SuppressWarnings("unchecked")
            Optional<U> r = (Optional<U>) mapper.apply(value);
            return Objects.requireNonNull(r);
        }
    }

    public Optional<T> or(Supplier<? extends Optional<? extends T>> supplier) {
        Objects.requireNonNull(supplier);
        if (isPresent()) {
            return this;
        } else {
            @SuppressWarnings("unchecked")
            Optional<T> r = (Optional<T>) supplier.get();
            return Objects.requireNonNull(r);
        }
    }

    public Stream<T> stream() {
        if (!isPresent()) {
            return Stream.empty();
        } else {
            return Stream.of(value);
        }
    }

    public T orElse(T other) {
        return value != null ? value : other;
    }

    public T orElseGet(Supplier<? extends T> supplier) {
        return value != null ? value : supplier.get();
    }

    public T orElseThrow() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X {
        if (value != null) {
            return value;
        } else {
            throw exceptionSupplier.get();
        }
    }


    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }

        if (!(obj instanceof Optional)) {
            return false;
        }

        Optional<?> other = (Optional<?>) obj;
        return Objects.equals(value, other.value);
    }


    @Override
    public int hashCode() {
        return Objects.hashCode(value);
    }

    @Override
    public String toString() {
        return value != null
            ? String.format("Optional[%s]", value)
            : "Optional.empty";
    }
}
```

Streams

```
Stream.of("hello\nworld", "ciao\nmondo", "Bonjour\nle monde", "Hai\ndunia")
    .map(x -> x.lines()) // returns a stream of streams


Stream.of("hello\nworld", "ciao\nmondo", "Bonjour\nle monde", "Hai\ndunia")
    .flatMap(x -> x.lines()) // return a stream of strings

Stream.iterate(0, x -> x + 1).takeWhile(x < 5);
Stream.iterate(0, x -> x + 1).peek(System.out::println).takeWhile(x < 5)
  .forEach(x -> {});

Stream.of(1, 2, 3).reduce(0, (x, y) -> x + y);

boolean isPrime(int x) {
  return IntStream.range(2, x)
      .noneMatch(i -> x % i == 0);
}

IntStream.iterate(2, x -> x+1)
    .filter(x -> isPrime(x))
    .limit(500)
    .forEach(System.out::println);

List<String> letters = Arrays.asList("a", "b", "c", "d", "e");
String result = letters
  .stream()
  .reduce("", (partialString, element) -> partialString + element);
assertThat(result).isEqualTo("abcde");

List<String> letters = Arrays.asList("a", "b", "c", "d", "e");
String result = letters
  .stream()
  .reduce("", (partialString, element) -> partialString + element);
assertThat(result).isEqualTo("abcde");

String result = letters
  .stream()
  .reduce(
    "", (partialString, element) -> partialString.toUpperCase() + element.toUpperCase());
assertThat(result).isEqualTo("ABCDE");
```