Faculté des Sciences
Aix*Marseille Université

Année universitaire

Site : ☐ Luminy ☐ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☒ Session 2 Durée de l'épreuve : 4h
Examen de : L2 Nom du diplôme : Licence d'informatique
Code du module : SIN3U02 Libellé du module : Programmation 2
Calculatrices autorisées : NON Documents autorisés : OUI

---

1 Correction de l'examen

Vous trouverez ci-dessous une correction possible de l'examen. C'est une des corrections possibles mais pas la seule.

2 La tâche de calcul : classe `Job` (1 point)

```
package cluster;

public class Job {
  private final int memory;
  private final int flop;
  private final int id;
  private static int jobCount = 0;

  public Job(int memory, int flop) {
    if (memory <= 0){
      throw new IllegalArgumentException("Cannot create a job with negative memory : " + getMemory()
    }
    if (flop <= 0){
      throw new IllegalArgumentException("Cannot create a job with a negative number of operations
    }
    this.memory = memory;
    this.flop = flop;
    id = jobCount++;
  }

  public int getFlop() {
    return flop;
  }

  public int getMemory() {
    return memory;
  }

  public int getId() {
    return id;
  }

  public static void resetJobCount() {
    jobCount = 0;
  }

  public static int getJobCount() {
    return jobCount;
```

```
  }

  public String toString(){
    return "Job "+ getId()
            + " (" + getFlop() + "FLOP, "
            + getMemory() + "o)";
  }
}
```

3 Tester la classe `Job` : classe `TestJob` (2 points)

```
package cluster;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static org.assertj.core.api.AssertionsForClassTypes.assertThat;

public class TestJob {

    @Test
    void testToString(){
        // Test with Junit
        Job job1 = new Job(100, 100);
        String expectedStringRepresentation1 = "Job " + job1.getId() + " (100FLOP, 100o)";
        assertEquals(expectedStringRepresentation1, job1.toString());

        // Test with assertJ
        Job job2 = new Job(1, 10);
        String expectedStringRepresentation2 = "Job " + job2.getId() + " (10FLOP, 1o)";
        assertThat(job2.toString()).isEqualTo(expectedStringRepresentation2);
    }

    @Test
    void testResetJobCount(){
        Job.resetJobCount();
        Job job = new Job(10, 1000);
        assertThat(job.getId()).isEqualTo(0);
    }

    @Test
    void testGetId(){
        int[] memories = {10, 1000};
        for(int memory : memories){
            Job job = new Job(memory, 1000);
            assertThat(job.getMemory()).isEqualTo(memory);
        }
    }

    @Test
    void testGetJobCount(){
        Job.resetJobCount();
        for(int jobCount = 1; jobCount < 4; jobCount++){
            Job job = new Job(10, 1000);
            assertThat(Job.getJobCount()).isEqualTo(jobCount);
        }
    }
```

```
}
```

4 Les nœuds de la ferme de calcul : classe `Node` (1 point)

```
package cluster;

import java.util.ArrayList;
import java.util.List;

public class Node {
  private final String name;
  private final int memoryCapacity;
  private final int flopCapacity;

  private int availableMemory;
  private int availableFlop;
  private final List<Job> assignedJobs = new ArrayList<>();

  Node(String name, int memoryCapacity, int flopCapacity) {
    this.name = name;
    this.memoryCapacity = memoryCapacity;
    this.flopCapacity = flopCapacity;
    availableMemory = memoryCapacity;
    availableFlop = flopCapacity;
  }

  boolean canAccept(Job job){
    return availableMemory >= job.getMemory() &&
            availableFlop >= job.getFlop();
  }

  boolean canHandle(Job job){
    return memoryCapacity >= job.getMemory() &&
            flopCapacity >= job.getFlop();
  }

  void accept(Job job){
    assignedJobs.add(job);
    availableMemory -= job.getMemory();
    availableFlop -= job.getFlop();
  }

  int usedMemory(){
    return memoryCapacity - availableMemory;
  }

  int usedFlop(){
    return flopCapacity - availableFlop;
  }

  public String toString(){
    return "Node " + name + " (" + usedFlop() + "/" + flopCapacity + "FLOP, " + usedMemory() + "/" +
  }

  void printJobs(){
    System.out.println(this);
```

```
    for(Job job : assignedJobs){
      System.out.println(job);
    }
  }
}
```

5 Exception NotEnoughMemoryException (2 points)

```
package cluster;

public abstract class NotEnoughResourceException extends Exception {
  private final Node node;
  private final Job job;

  public NotEnoughResourceException(Node node, Job job) {
    this.node = node;
    this.job = job;
  }

  @Override
  public String getMessage() {
    return node + getSentenceMessage() + job;
  }

  public abstract String getSentenceMessage();
}
```

```
package cluster;

public class NotEnoughRemainingMemoryException extends NotEnoughResourceException{
  public NotEnoughRemainingMemoryException(Node node, Job job) {
    super(node, job);
  }

  @Override
  public String getSentenceMessage() {
    return " has not enough remaining memory to handle ";
  }
}
```

```
package cluster;

public class NotEnoughTotalMemoryException extends NotEnoughResourceException {
  public NotEnoughTotalMemoryException(Node node, Job job) {
    super(node, job);
  }

  @Override
  public String getSentenceMessage() {
    return " has not enough total memory to handle ";
  }
}
```

```
package cluster;

public class NotEnoughTotalFlopException extends NotEnoughResourceException{
  public NotEnoughTotalFlopException(Node node, Job job) {
```

```
    super(node, job);
  }

  @Override
  public String getSentenceMessage() {
    return " has not enough total number of operations to handle ";
  }
}
```

```
package cluster;

public class NotEnoughRemainingFlopException extends NotEnoughResourceException {
  public NotEnoughRemainingFlopException(Node node, Job job) {
    super(node, job);
  }

  @Override
  public String getSentenceMessage() {
    return " has not enough remaining FLOP to handle ";
  }
}
```

```
public class Node {
void accept(Job job) throws NotEnoughResourceException{
    if(memoryCapacity < job.getMemory()){
      throw new NotEnoughTotalMemoryException(this, job);
    }
    if(availableMemory < job.getMemory()){
      throw new NotEnoughRemainingMemoryException(this, job);
    }
    if(flopCapacity < job.getFlop()){
      throw new NotEnoughTotalFlopException(this, job);
    }
    if(availableFlop < job.getFlop()){
      throw new NotEnoughRemainingFlopException(this, job);
    }
    assignedJobs.add(job);
    availableMemory -= job.getMemory();
    availableFlop -= job.getFlop();
  }
}
```

6 Interface `JobGenerator` et classe `UniformJobGenerator` (1 points)

```
package cluster;

public interface JobGenerator {
  Job generateJob();
}
```

```
package cluster;

import java.util.Random;

public class RandomJobGenerator implements JobGenerator{
  private final int minMemory;
  private final int maxMemory;
```

```
  private final int minFlop;
  private final int maxFlop;
  private final Random random;

  public RandomJobGenerator(int minMemory, int maxMemory, int minFlop, int maxFlop, long seed) {
    this.minMemory = minMemory;
    this.maxMemory = maxMemory;
    this.minFlop = minFlop;
    this.maxFlop = maxFlop;
    this.random = new Random(seed);
  }

  private int nextInt(int min, int max){
    return min + random.nextInt(max - min + 1);
  }

  public Job generateJob(){
    return new Job(nextInt(minMemory, maxMemory), nextInt(minFlop, maxFlop));
  }
}
```

7 Classe `RandomJobGenerator` (2 points)

```
package cluster;

import java.util.Random;

public class RandomJobGenerator implements JobGenerator{
  private final int minMemory;
  private final int maxMemory;
  private final int minFlop;
  private final int maxFlop;
  private final Random random;

  public RandomJobGenerator(int minMemory, int maxMemory, int minFlop, int maxFlop, long seed) {
    this.minMemory = minMemory;
    this.maxMemory = maxMemory;
    this.minFlop = minFlop;
    this.maxFlop = maxFlop;
    this.random = new Random(seed);
  }

  private int nextInt(int min, int max){
    return min + random.nextInt(max - min + 1);
  }

  public Job generateJob(){
    return new Job(nextInt(minMemory, maxMemory), nextInt(minFlop, maxFlop));
  }
}
```

8 Ordonnanceur : interface `Scheduler` (1 point)

```
package cluster;

import java.util.List;
```

```java
public interface Scheduler<J extends Job> {
  List<J> scheduleJobs(List<J> jobs, List<Node> nodes);
}
```

9 Ordonnanceur aléatoire : classe `RandomScheduler` (2 points)

```java
package cluster;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RandomScheduler implements Scheduler<Job>{
  private Random random = new Random();

  public Node randomNode(List<Node> nodes){
    return nodes.get(random.nextInt(nodes.size()));
  }

  @Override
  public List<Job> scheduleJobs(List<Job> jobs, List<Node> nodes) {
    List<Job> nonScheduledJobs  = new ArrayList<>();
    for(Job job : jobs){
      try{
        randomNode(nodes).accept(job);
      }
      catch (NotEnoughMemoryException e){
        nonScheduledJobs .add(job);
      }
    }
    return nonScheduledJobs ;
  }
}
```

10 Classe `PriorityJob` (2 points)

```java
package cluster;

public class PriorityJob extends Job implements Comparable<PriorityJob>{
  private final int priority;

  public PriorityJob(int memory, int flop, int priority) {
    super(memory, flop);
    this.priority = priority;
  }

  @Override
  public int compareTo(PriorityJob job) {
    return job.priority - this.priority;
  }
}
```

11 Ordonnanceur avec priorité : classe `PriorityScheduler` (2 points)

```java
package cluster;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PriorityScheduler implements Scheduler<PriorityJob> {
  @Override
  public List<PriorityJob> scheduleJobs(List<PriorityJob> jobs, List<Node> nodes) {
    List<PriorityJob> nonScheduledJobs = new ArrayList<>();
    Collections.sort(jobs);
    for (PriorityJob job : jobs) {
      if(!scheduleJob(job, nodes)){
        nonScheduledJobs.add(job);
      }
    }
    return nonScheduledJobs;
  }


  private boolean scheduleJob(Job job, List<Node> nodes){
    for (Node node : nodes) {
      try {
        node.accept(job);
        return true;
      }
      catch (NotEnoughResourceException e){}
    }
    return false;
  }
}
```

12 Classe `Controller` (2 points)

```java
import java.util.ArrayList;
import java.util.List;

public class Controller<J extends Job> {
  private List<J> nonScheduledJobs = new ArrayList<>();
  private final List<Node> nodes = new ArrayList<>();
  private final String name;
  private final Scheduler<J> scheduler;

  void addNode(Node node){
    nodes.add(node);
  }

  void submitJob(J job){
    nonScheduledJobs.add(job);
  }

  Controller(String name, Scheduler<J> scheduler){
    this.name = name;
    this.scheduler = scheduler;
```

```
  }

  void printNodesAndNonScheduledJobs(){
    System.out.println(this);
    printNodes();
    System.out.println("Non scheduled jobs");
    printNonScheduledJobs();
  }

  private void printNodes(){
    for (Node node : nodes){
      node.printJobs();
      System.out.println();
    }
  }

  private void printNonScheduledJobs(){
    for (Job job : nonScheduledJobs){
      System.out.println(job);
    }
  }

  public void scheduleJobs(){
    nonScheduledJobs = scheduler.scheduleJobs(nonScheduledJobs, nodes);
  }

  @Override
  public String toString() {
    return "Controller " + name;
  }
}
```