TEST

Software Systems: Programming

course code: 202001024
date: 28 January 2021
time: 13:45 – 17:45

SOLUTIONS

General

- You may use the following (unmarked) materials when making this test:
  - Module manual.
  - Slides of the Programming topics.
  - The book
    David J. Eck. *Introduction to Programming Using Java*. Version 8.1.2, December 2020.
  - A dictionary of your choice.
  - IntelliJ and/or Eclipse.
  - Documentation of Java 11
    `https://docs.oracle.com/en/java/javase/11/docs/api/`

  You are *not* allowed to use any of the following:
  - Solutions of any exercises published on Canvas (such as recommended exercises or old tests);
  - Your own materials (copies of (your) code, solutions of lab assignments, notes of any kind, etc.).
- When you are asked to write Java code, follow code conventions where they are applicable. Failure to do so may result in point deductions. It is recommended that you use IntelliJ and/or Eclipse to write code. Remember you can copy code from Remindo to IntelliJ and/or Eclipse.
- You do *not* have to add Javadoc or comments, unless explicitly asked to do so. Invariants, preconditions and postconditions should be given only when they are explicitly asked.
- You are not allowed to leave the room during the first 30 minutes or the last 15 minutes of the exam.
- Place your student ID card on the table as well as documentation that grants extra time (if applicable).

Question 1 (5 points)

Describe the difference between **overriding** and **overloading** with respect to the method signature. Give an example of overriding and an example of overloading to illustrate your explanation.

Answer to question 1

- (+1) for mentioning that overriding has the **same** method signature and overloading has a **different** method signature
- (+2) for a correct example of overriding (subclass with same method signature)
- (+2) for a correct example of overloading involving multiple methods in the same class with the same name and different parameter types

Question 2 (7 points)

a. *(4 pts)* Explain the difference between unit testing and integration testing and describe an example of unit testing and an example of integration testing.

b. *(3 pts)* Imagine you find a bug in your program. How do you approach solving the bug if you use **test driven development**?

Answer to question 2

a. (+2) for the difference that unit testing tests a unit (like a class or method) and integration testing tests multiple units at the same time, typically checking that they work well together
(+1) for an appropriate example of unit testing
(+1) for an appropriate example of integration testing

b. (+3) for describing that you first write a test case that is red (failing), then you fix the bug, and afterwards the test case is green (passing). Give 2 points for an answer that says you have to make a test connected to the bug, but that does not say you have to make the test before you fix the bug. Give 1 point for something that says you have to test to find the bug, but does not specifically say you have to make a test that fails if the bug is there and succeeds if the bug is not there.

Question 3 (8 points)

Consider the following Java classes:

```
public class A {
    private A() {
    }
    public final int a() {
        return 4;
    }
}
public class ExampleClass extends A {
    protected final int a() {
        return b();
```

```
    }
    private final int b() {
        return c();
    }
    protected static int c() {
        return 4;
    }
    public final static void main(String[] args) {
        A ex = (A) new ExampleClass();
        (ExampleClass) ex.b();
    }
}
```

This code fragment has four compile-time and/or runtime errors. Explain these errors by referring to what is wrong, why that is wrong, and how it could be fixed.

Answer to question 3

Only the first four mentioned errors count. If the answer includes more, only grade the first four. Give just 1 point if the error is correct but the solution is wrong. Be careful: just copy-pasting the error from IntelliJ/Eclipse does not demonstrate that the student *understands* why it is an error.

- (+2) the constructor in A is **`private`** and because of that, you can't construct instances of `ExampleClass` since the (only) constructor in A may not be called from subclasses. Make it **`public`**/**`protected`** instead.
- (+2) `A.a()` is **`final`** and so you cannot override it in `ExampleClass`. Remove the **`final`** keyword.
- (+2) `ExampleClass.a()` is **`protected`**, while it overrides a method that is **`public`**. This is not allowed, so make it **`public`** instead (alternative: make `A.a` **`protected`**).
- (+2) Parentheses are missing in `((ExampleClass) ex).b();` simply add them... (making ex of type ExampleClass works but introduces another bug, give that just +1)

Question 4 (10 points)

Consider the following situation. You find the source code of the "firmware" (a software system) running inside a water sensor. It has a web-based interface, protected by a simple password. In the source code, you find the following declaration:

```
private static final String adminPasswordSHA256 =
        "e5f0d2bc06f4abc1cdcc51b99159a8285b9bf83aeb70e682bb168448d96613eb";
```

From this, you figure that the SHA-256 cryptographic hash function is used to protect the password.

a. *(3 pts)* Why is it a bad idea to use SHA-256 for this purpose? What is a better alternative?

From a reliable source, you hear that the admin password for this device is very simple. The length is always 4 or 5 characters long and only a single upper or lower character is used. So, examples of passwords include: "AAAA", "AAAAA", "FFFFF", "wwww" and "ccccc". Write a small program that tries to find the admin password. To get you started, we provide you with the following two snippets of Java code:

```
public static String hex(byte[] bytes) {
    StringBuilder result = new StringBuilder();
    for (byte aByte : bytes) {
        result.append(String.format("%02x", aByte));
    }
```

```
        return result.toString();
    }

    MessageDigest md = MessageDigest.getInstance("SHA-256");
    byte[] hash = md.digest(text.getBytes(StandardCharsets.UTF_8));
```

b. *(7 pts)* Give both your program and the password of the device.

Answer to question 4

a. (+2) SHA-256 is built for speed, making brute-forcing the password very easy, should the hash be leaked (as in this case).
(+1) Alternative: a purposely made slow hash function (e.g., PBKDF2**, scrypt or Argon2)

b. (+2) The answer was ”uuuuu”
(+1) Code has an appropriate loop to generate the possible passwords of length 4
(+1) Code has an appropriate loop to generate the possible passwords of length 5
(+3) Code computes the hex of the sha256 of possible passwords

Example code:

```
private static String hex(byte[] bytes) {
    StringBuilder result = new StringBuilder();
    for (byte aByte : bytes) {
        result.append(String.format("%02x", aByte));
    }
    return result.toString();
}

private static byte[] sha256(String text) {
    try {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(text.getBytes(StandardCharsets.UTF_8));
        return hash;
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);
    }
}

private static final String adminPasswordSHA256 =
        "e5f0d2bc06f4abc1cdcc51b99159a8285b9bf83aeb70e682bb168448d96613eb";

public static void main(String[] args) {
    char[] chars = ("abcdefghijklmnopqrstuvwxyz"+
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
    for (char c : chars) {
        String s4 = String.valueOf(c).repeat(4);
        String s5 = String.valueOf(c).repeat(5);

        if (hex(sha256(s4)).equals(adminPasswordSHA256)) {
            System.out.println(s4);
        }
        if (hex(sha256(s5)).equals(adminPasswordSHA256)) {
            System.out.println(s5);
        }
    }
}
```

Question 5 (10 points)

Imagine we have a file like the following example:

```
10
930
1116
218
491
682
178
905
192
558
283
851
```

Write a method **void** `processFile(String filename)` that reads all numbers from a file like the example and finds any two numbers a and b such that $a+b=2021$. The method must print these two numbers to standard output. For the example file these would be the numbers 1116 and 905. You don't need to print all possible numbers a and b, just the first one is enough.

Answer to question 5

- (+2) The file is actually opened somehow, e.g., **new** `Scanner(filename)` is wrong, but almost all other methods work.
- (+1) The file is closed after use (for example try-with-resources, or if using try-catch, then with close in the finally block)
- (+2) The numbers are somehow parsed into some collection (list, set) or array. Amount of points depends on how straightforward this is done. Full points for using a `Scanner` or parsing line by line with a BufferedReader where each line is fed to `Integer.parseInt` and added to the collection or array.
- (+2) Exceptions are handled inside the method appropriately (by printing to standard output/error a proper message, not just the default printStackTrace), or are thrown outside `processFile`.
- (+3) There is (a) a nested loop or similar construction that (b) finds the two numbers a and b such that $a+b=2021$ and (c) prints the result to standard output. Each step is 1 point.

Example Java code:

```
void processFile(String filename) {
    List<Integer> numbers = new ArrayList<>();
    try (Scanner sc = new Scanner(new File(filename))) {
        while (sc.hasNextInt()) numbers.add(sc.nextInt());
    } catch (FileNotFoundException e) {
        System.err.println("The␣file␣" + filename + "␣is␣not␣found!");
    }

    for (int n : numbers) {
        for (int m : numbers) {
            if ((n+m) == 2021) {
                System.out.println("numbers:␣" + n + "␣and␣" + m);
                return; // only first one needed
            }
        }
    }
}
```

Alternative:

```
void processFile(String filename) {
    try {
        Scanner sc = new Scanner(new File(filename));
        List<Integer> numbers = new ArrayList<>();
        while (sc.hasNextInt()) numbers.add(sc.nextInt());
        sc.close();

        for (int n : numbers) {
            for (int m : numbers) {
                if ((n+m) == 2021) {
                    System.out.println("numbers: " + n + " and " + m);
                    return; // only first one needed
                }
            }
        }
    } catch (FileNotFoundException e) {
        System.err.println("The file " + filename + " is not found!");
    }
}
```

Question 6 (10 points)

Imagine we are playing a guessing game. In this question, we are going to guess a number between 0 and 100. We make two classes, `Guesser` and `Challenger`. The Challenger comes up with a random number between 0 and 100. The guesser must repeatedly try to guess the number, and the reply is either 0 (you guessed correct), 1 (your guess was too high) or -1 (your guess was too low).

a. *(5 pts)* Implement a class `Challenger` which computes a random number between 0 and 100 (including 0 but excluding 100) inside the constructor. You can use the `nextInt(int)` method of the `java.util.Random` class to obtain such a number. Implement a method `int guess(int number)` (in the `Challenger` class) which returns -1, 0 or 1 according to the description.

b. *(5 pts)* Implement a class `Guesser` which can solve the challenge. Implement a recursive method `int solve(Challenger challenger, int lowest, int highest)` which guesses a number `x` where `lowest <= x && x <= highest`. Your implementation must be recursive and may not contain any loops.

Answer to question 6

a. (+1) a **private** field to store the number and a **public** method `guess`
(+1) the **constructor** initializes the number to a random number x, $0 \leq x < 100$, or is given a random number (the original question was ambiguous). Using `Math.random()` is also fine.
(+3) method `guess` returns the correct result. Also correct is using `Integer.compare` (which IntelliJ suggests)
To some extent, this question also tested the ability to use Javadoc.

b. (+2) there is a single call to `guess` of the Challenger (only 1 point if the answer calls `guess` multiple times)
(+1) if `guess` returns 0, the guessed value is returned
(+1) if `guess` returns -1, `solve` is **correctly** called (and the result returned)

(+1) if `guess` returns 1, `solve` is **correctly** called (and the result returned); only give 0.5 for `return solve(challenger, x, highest)` if it may result in an infinite recursion.
A solution that only checks whether `guess` returns 0 or not but is otherwise correct is worth 4 points: technically correct but not making good use of the return values of `guess`. Check what happens in corner cases, such as the number being 0 or 99 and if no infinite recursion occurs in those cases.

```
public class Challenger {
    private final int number;

    public Challenger() {
        number = new Random().nextInt(100);
    }

    public int guess(int number) {
        if (this.number == number) return 0;
        else if (this.number < number) return 1;
        else return -1;
    }
}

public class Guesser {
    public static int solve(Challenger challenger, int lowest, int highest) {
        int mid = (lowest+highest)/2;
        switch (challenger.guess(mid)) {
            case 0:
                return mid;
            case -1:
                // your guess was too low
                return solve(challenger, mid+1, highest);
            case 1:
                // your guess was too high
                return solve(challenger, lowest, mid-1);
            default:
                throw new RuntimeException("Unexpected result from challenger");
        }
    }
}
```

Question 7 (20 points)

In this question, we are concerned with ordering the perfect pizza. Assume we have the following toppings:

```
public enum Topping {
    ONIONS,
    MOZZARELLA,
    SALAMI,
    MUSHROOMS,
    SALMON,
    OLIVES,
    PINEAPPLE,
    SPINACH,
    PEPPERS,
    CHILI,
```

```
    GARLIC
}
```

We want to calculate the **subjective value** of a pizza: how good do we estimate that a pizza with certain toppings would taste? Example rules are:

- contains both onions and salami: multiply value by $1.25\times$
- contains mushrooms: multiply value by $1.2\times$
- contains pineapple: multiply value by $0.2\times$

If we consider a pizza with onions, salami and pineapple, and we start with an initial value of 10.0, application of the rules would result in the value 12.5 after applying the first rule, then 12.5 after applying the second rule, and finally 2.5 after the third rule.

a. *(2 pts)* Define an interface `ToppingRule` with a method **`double`** `apply(Collection<Topping> toppings, double value)`. The purpose of the `apply` method will be to check if the rule (for example: if contains onions, multiply by 2) matches the given toppings, and if so, to apply the rule to the given value, returning the result (either the updated value or the unchanged value).

b. *(3 pts)* Define an implementation of `ToppingRule` called `OneToppingRule` that checks in `apply` whether the given toppings include **one** topping and applies a (**`double`**) modifier if the topping is present. Don't forget to include an appropriate constructor.

c. *(3 pts)* Define an implementation of `ToppingRule` called `TwoToppingsRule` that checks of **two** toppings and applies a (**`double`**) modifier if both toppings are present. Don't forget to include an appropriate constructor.

d. *(3 pts)* Implement a class `ToppingRules`. This class must have some data structure to store your `ToppingRule`s. Implement a method **`void`** `addRule(ToppingRule rule)` which you will use to add your rules.

e. *(3 pts)* Implement a method in the `ToppingsRules` class called **`double`** `evaluate(Collection<Topping> toppings)` that applies the rules to a collection of toppings and returns the result. As the initial value, use the number 10.

f. *(6 pts)* A new pizza place has opened right around the corner, and it allows selecting exactly three toppings from the list of toppings. Write a program that finds the best combination of toppings, according to the rules added to a `ToppingsRules` object and prints the result to standard output. You must include the given example rules above, but are free to add additional rules of your choice.

Answer to question 7

a. (+2) Full points for model answer.
(-1) adding **`public`** to the method is wrong.
(-1) turning the interface into some weird Java Generics thing is also wrong.
(-1) including the enum `Topping` in the interface is also wrong.
(-1) adding a method body to `applyRule` is wrong.

```
public interface ToppingRule {
    double applyRule(Collection<Topping> toppings, double value);
}
```

b. (+1) appropriate fields (both private, correct types, sensible names)
(+1) a constructor that is public and sets the two fields
(+1) a correct implementation of `applyRule`

```
public class OneToppingRule implements ToppingRule {
    private final Topping topping;
    private final double modifier;

    public OneToppingRule(Topping topping, double modifier) {
        this.topping = topping;
        this.modifier = modifier;
    }

    public double applyRule(Collection<Topping> toppings, double value) {
        if (toppings.contains(topping)) value *= modifier;
        return value;
    }
}
```

c. (+1) appropriate fields (all three private, correct types, sensible names)
(+1) a constructor that is public and sets the three fields
(+1) a correct implementation of `applyRule` (only half points if they apply the rule multiple times if the topping occurs multiple times)

```
public class TwoToppingsRule implements ToppingRule {
    private Topping topping1, topping2;
    private double modifier;

    public TwoToppingsRule(Topping topping1, Topping topping2,
                           double modifier) {
        this.topping1 = topping1;
        this.topping2 = topping2;
        this.modifier = modifier;
    }

    public double applyRule(Collection<Topping> toppings, double value) {
        if (toppings.contains(topping1) &&
            toppings.contains(topping2)) value *= modifier;
        return value;
    }
}
```

d. (+1) a single **`private`** field for the rules, appropriate type (List, Set, something like that)
(+1) appropriate initialization of the field
(+1) addRule is **`public`** and simply adds the given rule to the field

```
public class ToppingRules {
    private List<ToppingRule> rules = new ArrayList<>();

    public void addRule(ToppingRule rule) {
        rules.add(rule);
    }
}
```

In the original exam, instead of `ToppingRule` it said `ValueModifier` as the parameter to `addRule`. This was corrected via a message on the global whiteboard of the BBB chat (and this was clarified to any students who asked about it.)

e. (+1) a loop that iterates over each rule
(+2) appropriately uses the `applyRule` method of each rule, starting with value 10 (no points for only starting with value 10 and not using `applyRule`)

```
public double evaluate(Collection<Topping> toppings) {
    double value = 10.0;
    for (ToppingRule r : rules) {
        value = r.applyRule(toppings, value);
    }
    return value;
}
```

f. (+1) Correctly create a `ToppingRules` object and initialize it with the rules
(+2) A nested loop or something similar to iterate through all combinations of toppings
(+1) A call to `evaluate` to evaluate each topping
(+2) Some method to extract (one of) the best combination of toppings and print it to standard output. (no points for just printing something to standard output)

```
public static void main(String[] args) {
    ToppingRules rules = new ToppingRules();
    rules.addRule(new TwoToppingsRule(Topping.onions, Topping.salami, 1.25));
    rules.addRule(new OneToppingRule(Topping.mushrooms, 1.2));
    rules.addRule(new OneToppingRule(Topping.pineapple, 0.2));
    rules.addRule(new OneToppingRule(Topping.mozzarella, 1.25));

    double best = 0;
    Collection<Topping> bestToppings = null;

    for (Topping first : Topping.values()) {
        for (Topping second : Topping.values()) {
            for (Topping third : Topping.values()) {
                List<Topping> l = new ArrayList<>();
                l.add(first);
                l.add(second);
                l.add(third);
                if (bestToppings == null || best < rules.evaluate(l)) {
                    bestToppings = l;
                    best = rules.evaluate(l);
                }
            }
        }
    }

    System.out.println("Best␣toppings:␣" + bestToppings.toString());
}
```

Question 8 (30 points)

You are asked to make a management system for managing the inventory of pizza toppings.

a. *(7 pts)* Implement a class `Inventory` that maintains the amount of available toppings. Initially, there are 10 units of each topping available. Implement a method **`void`** `refill(Topping topping,` **`int`** `amount)` that adds the given amount of units of the given topping in the inventory. Implement a method **`boolean`** `take(Topping topping)` which tries to take one unit of the specified topping, if possible. It must return **`true`** if successful, and **`false`** otherwise.

b. *(5 pts)* Define appropriate preconditions and postconditions for `refill`. Pay attention that your postcondition should describe the change to the state of the `Inventory` object after a call to `refill`.

c. *(5 pts)* After some soul searching, you decide that you actually want to throw an exception if the topping is not available (all are taken), instead of returning **`false`**. Define a new exception `NoToppingsLeftException` for this purpose and modify `take` to return nothing and instead throw the exception if the topping is not available. Give both the newly defined exception and the modified `take` method as your answer.

d. *(7 pts)* You suddenly remember that the inventory object will be manipulated by many different threads. Modify your implementation to ensure that no race conditions occur, i.e., `refill` and `take` always run as expected. Explain why your implementation is thread-safe and describe what can happen without your modifications. Give the modified `take` and `refill` methods as your answer.

e. *(6 pts)* Modify `take` to only return when a topping has been taken. That means that if there is no topping available, your implementation should wait until the topping has been refilled by another thread. Use an appropriate method to do this. Give the modified `take` and `refill` methods as your answer.

Answer to question 8

a. (+1) for correct private field of type `Map<Topping, Integer>`. Only 0.5 for a `List<Topping>` (where toppings can appear multiple times).
(+2) for correct initialization (correct instance of field; every topping 10 each) - the initialization can be directly and/or in a constructor. (although "initializer block" could work, it's not actually advised, so no points for that) give only 0.5 points for correct instance of field and no other initialization
(+2) `refill` correctly **adds** the given amount (not replaces old amount by the parameter amount)
(+1) `take` returns false if no toppings left
(+1) `take` removes exactly 1 topping and returns true if any toppings left

```
public class Inventory {
    private Map<Topping, Integer> amounts = new HashMap<>();

    public Inventory() {
        for (Topping topping : Topping.values()) {
            amounts.put(topping, 10);
        }
    }

    public void refill(Topping topping, int amount) {
        amounts.put(topping, amounts.get(topping) + amount);
    }

    public boolean take(Topping topping) {
        int current = amounts.get(topping);
        if (current > 0) {
            amounts.put(topping, current-1);
            return true;
        } else {
            return false;
        }
    }
}
```

b. (+1) for a precondition `@requires topping != null`
(+1) for a precondition `@requires amount >= 0`
(+2) for a postcondition that the given topping is changed:

```
@ensures amounts.get(topping) == \old(amounts.get(topping)) + amount
```

(but only +1 if the answer is that it changes without quantifying the amount)
(+1) for an additional postcondition that the rest stays the same:

```
@ensures (\forall Topping t; t != topping ==>
    amounts.get(topping) == \old(amounts.get(topping)))
```

Informal but precise descriptions get full points; the less precise, the fewer points (check that each given postcondition is really ensured by every method call etc)

c. (+2) Appropriate exception that extends the `Extension` class (constructor is optional but nice to have; perhaps required in future exams)
(+1) `take` is now **`void`** and **`throws`** `NoToppingsLeftException` (0.5 if only one of these)
(+2) `take` correctly throws the exception

```
public class NoToppingsLeftException extends Exception {
    public NoToppingsLeftException() {
        super("There␣are␣no␣toppings␣left!");
    }
}

public void take(Topping topping) throws NoToppingsLeftException {
    int current = amounts.get(topping);
    if (current > 0) {
        amounts.put(topping, current-1);
    } else {
        throw new NoToppingsLeftException();
    }
}
```

d. (+3) an implementation that uses **`synchronized`** on the method call is most appropriate. Only 2 points for solutions that explicitly create extra objects or locks. 1 point for a solution that uses explicit locks and does not unlock when throwing an exception.
(+2) an explanation that explains that only one thread at a time can be inside the critical section (the synchronized method, the synchronized block, between lock and unlock) and therefore there are no race conditions
(+2) a description of what can happen when not protecting the `amounts` field, e.g., a `take` or `refill` call does not actually have effect because another thread was simultaneously modifying the map.

Standard implementation:

```
public synchronized void refill(Topping topping, int amount) {
    amounts.put(topping, amounts.get(topping) + amount);
}

public synchronized void take(Topping topping)
            throws NoToppingsLeftException {
    int current = amounts.get(topping);
    if (current > 0) {
        amounts.put(topping, current-1);
    } else {
```

```
            throw new NoToppingsLeftException();
        }
    }
```

Also correct:

```
public class Inventory {
    private Map<Topping, AtomicInteger> amounts = new HashMap<>();

    public Inventory() {
        for (Topping topping : Topping.values()) {
            amounts.put(topping, new AtomicInteger(10));
        }
    }

    public void refill(Topping topping, int amount) {
        amounts.get(topping).addAndGet(amount);
    }

    public void take(Topping topping) throws NoToppingsLeftException {
        if (amounts.get(topping).decrementAndGet() < 0) {
            amounts.get(topping).incrementAndGet();
            throw new NoToppingsLeftException();
        }
    }
}
```

This is correct but not expected. Award full points only if the usage of `AtomicInteger` is completely correct (but they can use slightly different methods since an `AtomicInteger` offers a lot of ways to atomically modify the integer). Award no points for the mixing of **synchronized** and `AtomicInteger`. And for the next part, it turns out using `AtomicInteger` was actually not ideal.

e. (+2) Using `notifyAll` inside **`synchronized`** in `refill`. Only 1 point for `notify`, because that just wakes up a single thread that might be waiting for a different topping.
(+2) Using `wait` inside **`synchronized`** inside a **`while`** loop as in the example answer. After each `wait`, check again if now there are now toppings available.
(+1) Actually return or break from the **`while`** loop if the topping was taken *and* `take` must no longer throw an exception and no longer have the **`throws`** keyword.
(+1) Correctly deal with the `InterruptedException` ensuring that the method ends if it is thrown inside `wait` (throwing it etc is fine, as long as it results in the method ending)
Similarly correct is using a `ReentrantLock`, creating a `Condition`, then use `signalAll` instead of `notifyAll` and `await` instead of `wait`.

```
public synchronized void refill(Topping topping, int amount) {
    amounts.put(topping, amounts.get(topping) + amount);
    notifyAll();
}

public synchronized void take(Topping topping) {
    try {
        while (true) {
            int current = amounts.get(topping);
            if (current > 0) {
                amounts.put(topping, current - 1);
                break;
```

```
                } else {
                    wait();
                }
            }
        } catch (InterruptedException ignored) {
            // we were interrupted; just end this now
        }
    }
```

Also possible:

```
public synchronized void refill(Topping topping, int amount) {
    amounts.put(topping, amounts.get(topping) + amount);
    notifyAll();
}

public synchronized void take(Topping topping) throws InterruptedException {
    while (amounts.get(topping) <= 0) {
        wait();
    }

    amounts.put(topping, amounts.get(topping) - 1);
}
```