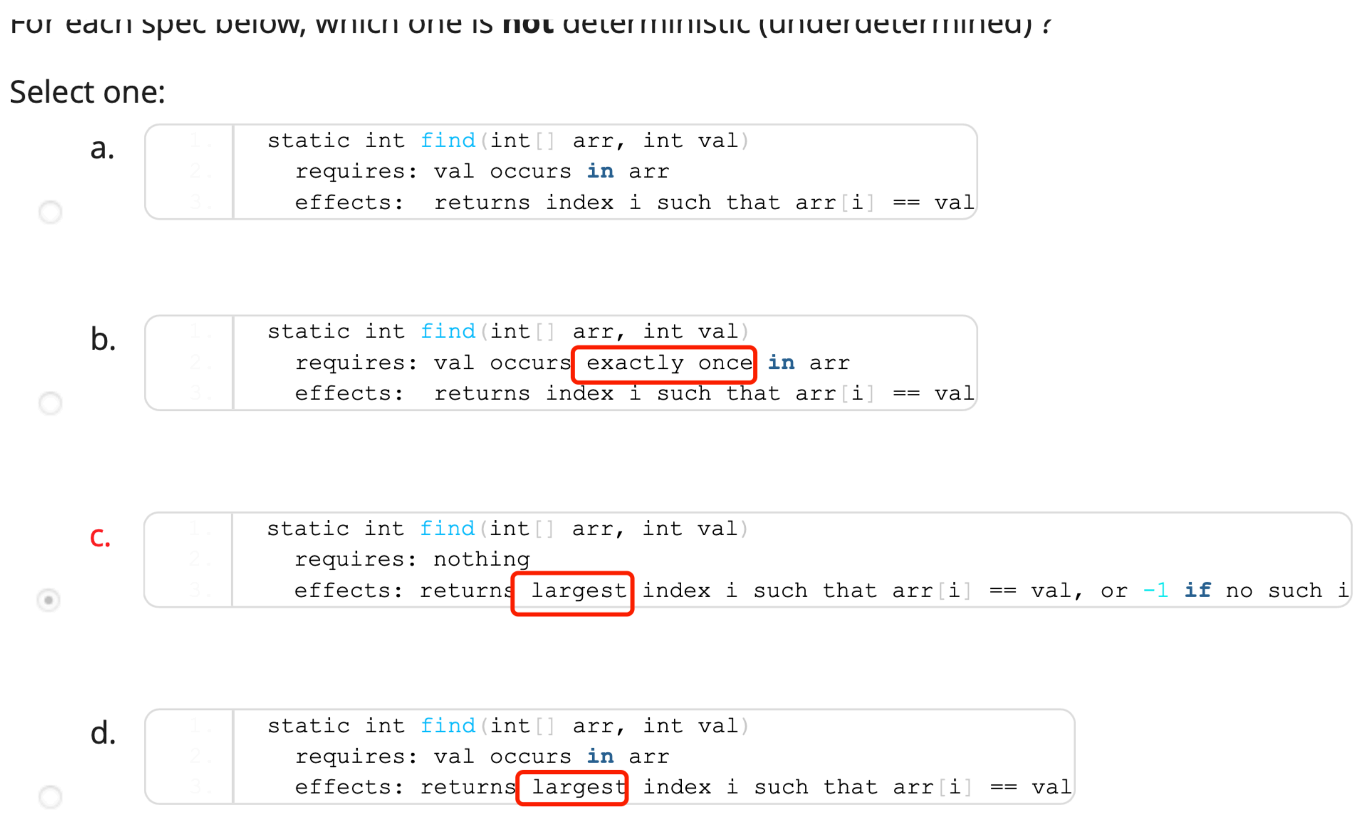
L8

Deterministic vs Underdetermined Specs



For each spec below, which one is **not** deterministic (underdetermined) ?

Select one:

a.
```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] == val
```

b.
```
static int find(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] == val
```

[✓ c.]
```
static int find(int[] arr, int val)
  requires: nothing
  effects: returns largest index i such that arr[i] == val, or -1 if no such i
```

d.
```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects: returns largest index i such that arr[i] == val
```

想红框中有明确说明的就是deterministic 没有就是underdetermined

Declarative vs Operational Specs

Operational specifications 给了方法执行的细节，而declarative没有给中间的执行细节

Stronger vs Weaker Specs

specification S2 is stronger than or equal to a specification S1 if

S2 的先决条件比s1的弱或者相等 **并且** s2的后置条件比s1的强或等于s1，并且状态满足先决条件

nothing 很弱

Incomparable

如果两个的specifications的没有比另一个强，那么他俩就是Incomparable的

Diagraming Specification

strengthening 意思就是说它更不自由

一个强spec或许有一个弱先决条件 并且/或 强后置条件

In both cases, the implementor must be more careful; but clients with more varied inputs or more specific needs might now be able to make use of the stronger spec.

Designing good specifications

it should obviously be succinct, clear, and well-structured, so that it’s easy to read.

The content of the specification, however, is harder to prescribe. There are no infallible rules, but there are some useful guidelines.

- ### The specification should be coherent
- ### The results of a call should be informative
- ### The specification should be strong enough
- ### The specification should also be weak enough
- ### The specification should use *abstract types* where possible

@SuppressWarnings("unchecked")

About access control

We have been using *public* for almost all of our methods, without really thinking about it. The decision to make a method public or private is actually a decision about the contract of the class. Public methods are freely accessible to other parts of the program. Making a method public advertises it as a service that your class is willing to provide. If you make all your methods public — including helper methods that are really meant only for local use within the class — then other parts of the program may come to depend on them, which will make it harder for you to change the internal implementation of the class in the future. Your code won’t be as **ready for change**.

Making internal helper methods public will also add clutter to the visible interface your class offers. Keeping internal things *private* makes your class’s public interface smaller and more coherent (meaning that it does one thing and does it well). Your code will be **easier to understand**.

We will see even stronger reasons to use *private* in the next few classes, when we start to write classes with persistent internal state. Protecting this state will help keep the program **safe from bugs**.

About static vs. instance methods

Summary

A specification acts as a crucial firewall between implementor and client — both between people (or the same person at different times) and between code. As we saw last time, it makes separate development possible: the client is free to write code that uses a module without seeing its source code, and the implementor is free to write the implementation code without knowing how it will be used.

规范充当实施者和客户端之间的关键防火墙-无论是人（或同一个人在不同的时间）和代码之间。正如我们上次看到的，它使单独的开发成为可能：客户端可以自由编写使用模块的代码，而无需查看其源代码，实施者可以自由编写实施代码，而不知道如何使用该代码。

Declarative specifications are the most useful in practice. Preconditions (which weaken the specification) make life harder for the client, but applied judiciously they are a vital tool in the software designer’s repertoire, allowing the implementor to make necessary assumptions.

声明性规范在实践中最为有用。先决条件（削弱规范）使客户的生活更加艰难，但明智地应用它们是软件设计者剧目中的重要工具，允许实施者做出必要的假设。

As always, our goal is to design specifications that make our software:

- **Safe from bugs**. Without specifications, even the tiniest change to any part of our program could be the tipped domino that knocks the whole thing over. Well-structured, coherent specifications minimize misunderstandings and maximize our ability to write correct code with the help of static checking, careful reasoning, testing, and code review.
- **Easy to understand**. A well-written declarative specification means the client doesn’t have to read or understand the code. You’ve probably never read the code for, say, Python `dict.update`, and doing so isn’t nearly as useful to the Python programmer as reading the declarative spec.
- **Ready for change**. An appropriately weak specification gives freedom to the implementor, and an appropriately strong specification gives freedom to the client. We can even change the specs themselves, without having to revisit every place they’re used, as long as we’re only strengthening them: weakening preconditions and strengthening postconditions.

免于错误。如果没有规格，即使是我们计划任何部分最微小的变化，也可能是将整个事情都打翻的多米诺骨牌。结构合理、连贯一致的规范最大限度地减少了误解，并最大限度地提高了我们通过静态检查、仔细推理、测试和代码审查编写正确代码的能力。

易于理解。写得很好的声明规范意味着客户不必阅读或理解代码。你可能从来没有读过代码， 比如说，Pythondict. update， 这样做对 Python 程序员来说并不像阅读声明性规范那么有用。

准备好改变。适当的弱规范为实施者提供了自由，而适当强大的规范为客户提供了自由。我们甚至可以自行更改规格，而不必重新审视它们使用的每一个地方，只要我们只加强它们：削弱先决条件和加强后条件。