Contracts

Contract Details

- Contracts in the APIs for OSU CSE components include these important features:
  - Parameter modes
  - Two stipulations:
    - Parameter names in requires and ensures clauses always stand for the ***object values***, never the reference values, of the corresponding arguments to a method call
    - Reference-type arguments are always ***non-null***

Contra

These are local decisions that apply to OSU CSE components' contracts; there are no industry standards (yet) that govern how to write contracts.

- Contracts in the A
  components inclu
  features:
  - Parameter modes
  - Two stipulations:
    - Parameter names in requires and ensures clauses always stand for the ***object values***, never the reference values, of the corresponding arguments to a method call
    - Reference-type arguments are always ***non-null***

Parameter Modes

- There are four ***parameter modes***, each of which indicates a possible way that a method might change the value of the corresponding argument
- Parameter modes help us in three ways:
  - They concisely summarize which arguments might have their values modified by a call
  - They make requires/ensures clauses shorter
  - They allow us to perform “sanity checks” of contracts against certain simple errors

Parameter Modes

- There are four ***parameter modes***, each of which indicates a possible way that a method might change the value of the corresponding argument

...us in three ways:

...ze which arguments ...modified by a call

...ures clauses shorter

...n "sanity checks" of contracts against certain simple errors

Modes are listed for the formal parameters, including **`this`**, but actually apply to their corresponding arguments for a call, including the receiver.

Restores Mode

- Upon return from a method call, a ***restores-mode*** parameter once again has its incoming value
  - Equivalent to adding, e.g., `... and x = #x` to the ensures clause
  - An old restores-mode parameter, e.g., `#x`, should not appear in the ensures clause
  - This is the ***default parameter mode***, so if a parameter is not listed with some other mode then its mode is ***restores***

Clears Mode

- Upon return from a method call, a ***clears-mode*** parameter has an ***initial value*** for its type, i.e., a value that an assignment of the *no-argument constructor* could give it
  - Equivalent to adding, e.g., `... and x = [an initial value for its type]` to the ensures clause
  - A clears-mode parameter, e.g., `x`, should not appear in the ensures clause except as `#x`

Cle

It's possible there isn't a no-argument constructor; see the contract for the `clear` method in interface `Standard` for technical details.

- Upon return fro
  mode parameter
  its type, i.e., a value that an assignment of the *no-argument constructor* could give it
  - Equivalent to adding, e.g., `... and x = [an initial value for its type]` to the ensures clause
  - A clears-mode parameter, e.g., `x`, should not appear in the ensures clause except as `#x`

Example

```
void transferFrom(NaturalNumber n)
```

- Sets **`this`** to the incoming value of `n`, and resets `n` to an initial value.
- Replaces: **`this`**
- Clears: `n`
- Ensures:

  `this` = *#n*

Replaces Mode

- Upon return from a method call, a ***replaces-mode*** parameter has a value that might be changed from its incoming value, but the method's behavior *does not depend* on its incoming value
  - A replaces-mode parameter, e.g., `x`, should not appear in the requires clause, and `#x` should not appear in the ensures clause

Example

`void` `copyFrom(NaturalNumber n)`

- Copies `n` to **`this`**.
- Replaces: **`this`**
- Ensures:

  `this` = *n*

Updates Mode

- Upon return from a method call, an ***updates-mode*** parameter has a value that might be changed from its incoming value, and the method's behavior *does depend* on its incoming value

Updates Mode

- Upon return from a method call, an ***updates-mode*** parameter has a value that might be changed from its incoming value, and the method's behavior *does depend* on its incoming value

In other words, both ***replaces*** and ***updates*** modes indicate that the parameter can change value. The difference is that for the former, the behavior of the method is independent of the incoming value, while for the latter it isn't.

Example

```
void add(NaturalNumber n)
```

- Adds `n` to **`this`**.
- Updates: **`this`**
- Ensures:

  `this` `=` `#`***`this`*** `+` `n`

Parameters Stand for Object Values

- When a parameter name is used in a requires or ensures clause, with or without the # to indicate the incoming value, it stands for the ***object value*** of the corresponding argument

Example

`void` `copyFrom(NaturalNumber n)`

- Copies `n` to **`this`**.
- Replaces: **`this`**
- Ensures:

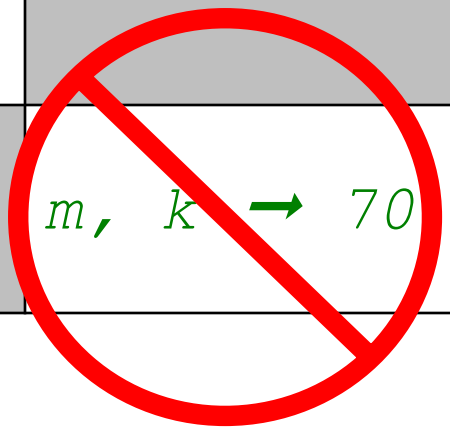
`this` $= n$

Which Means It Does This...

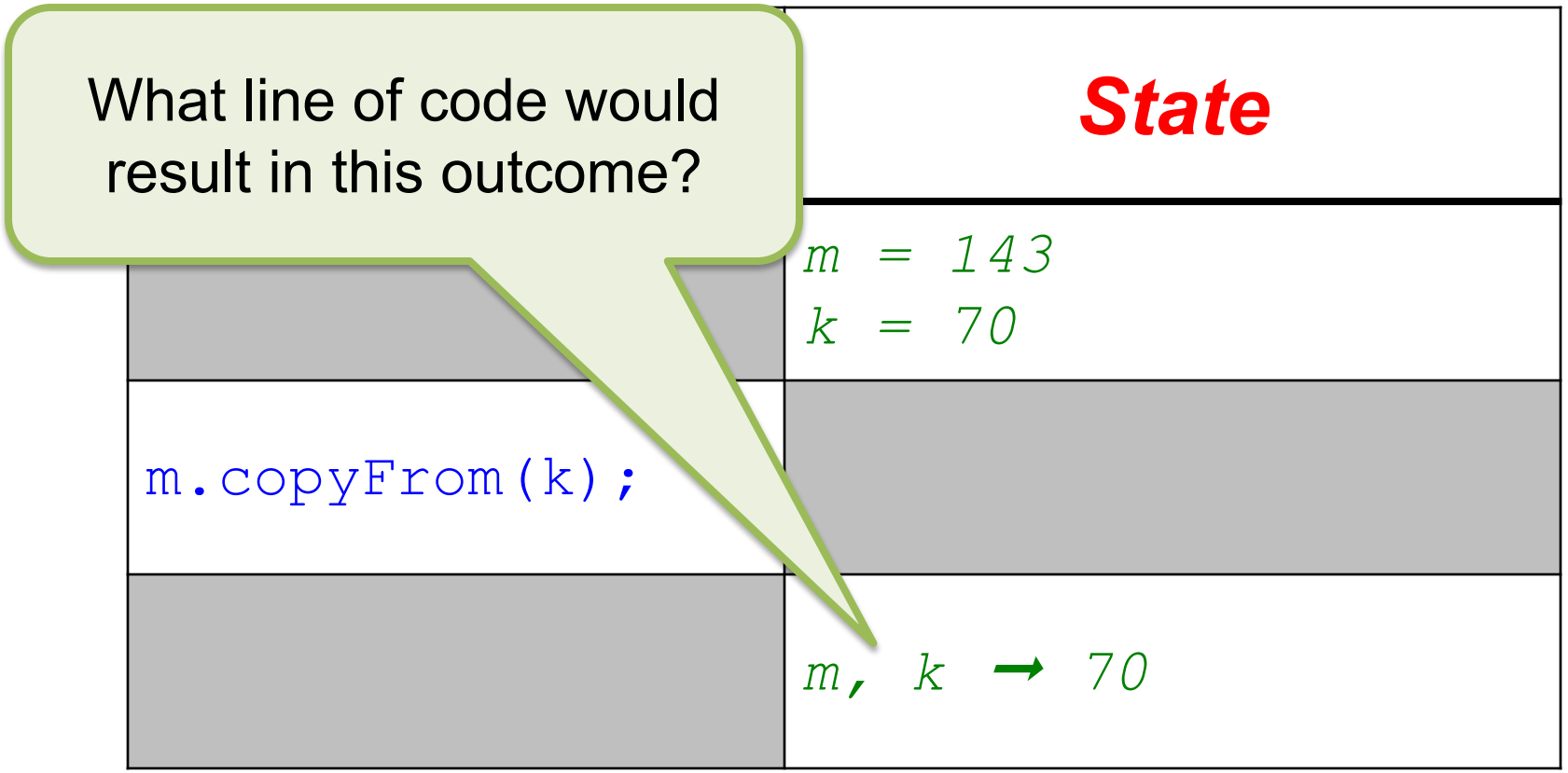
Code	State
	m = 143 *k = 70*
`m.copyFrom(k);`	
	m = 70 *k = 70*

... Not This!

Code	State
	m = 143 *k = 70*
`m.copyFrom(k);`	
	m, k → 70



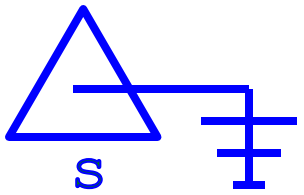
... Not This!


What line of code would result in this outcome?
State
m = 143
k = 70
m.copyFrom(k);
m, k → 70

Null References

- In Java, any reference variable may be given the special value **`null`**, meaning that it *does not refer to any object at all*:

```
String s = null;
```

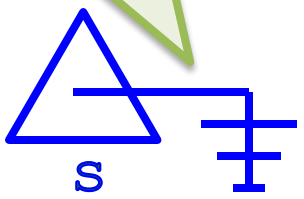


Null References

- In Java, any ref
  given the speci
  that it *does not*

```
String s = null;
```

This is special notation to replace the arrow when a reference is ***null***.



Best Practices for Null References

- It is not unusual to find such ***null references*** in Java code, even though it is often easy to avoid using them, and it is now considered a good idea to try to avoid making references null
- The most common cause of crashes in Java is `NullPointerException`, which means the code attempted to follow a null reference to the (non-existent) object to which it refers

Best Practices for Null References

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*
>
> — Sir C.A.R. Hoare, 2009

- Pretty much says it all...

Non-Null References Required

- OSU CSE components' contracts stipulate that no argument to any method may have a null reference value
  - Hence, there can be no question about what a reference-type parameter stands for in a requires or ensures clause: the reference always points to an object, and the parameter stands for that ***object value***

Resources

- *Null References: The Billion Dollar Mistake*
  - http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare