Introduction to Invariants

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 7.1

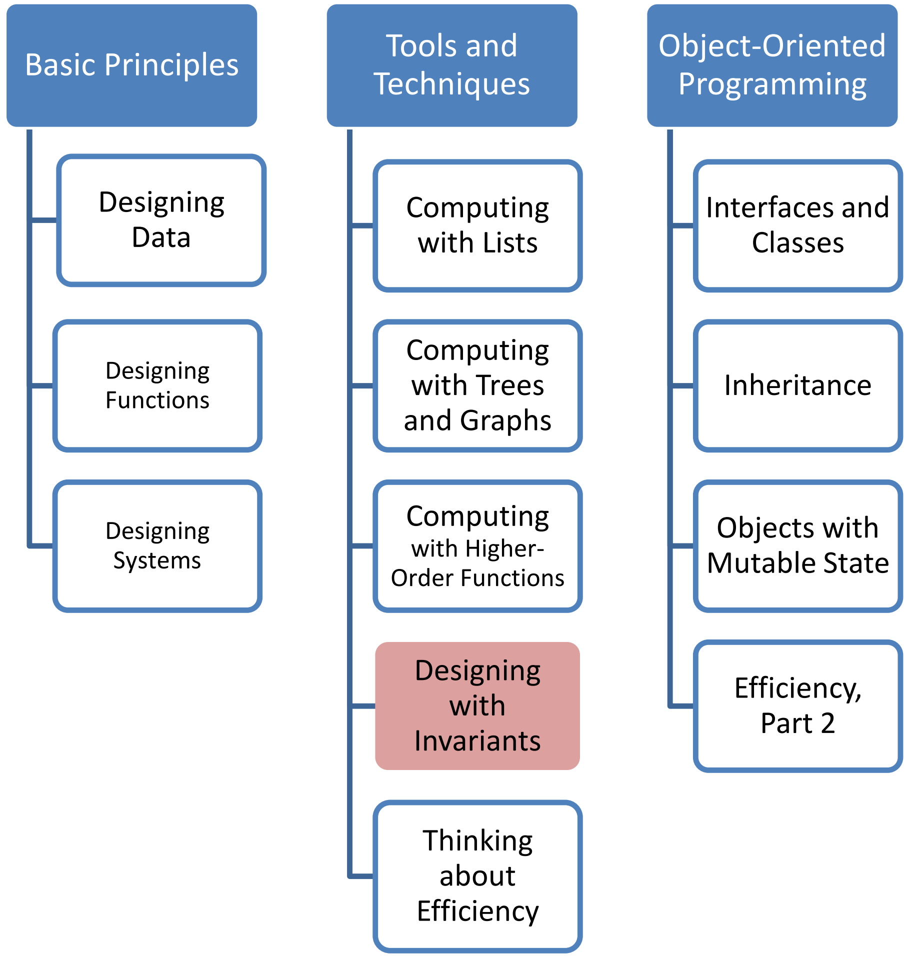
Module Introduction

- We introduce *invariants* as a way of recording the assumptions that a function makes about its arguments.
- Invariants divide the responsibility for guaranteeing the truth of these assumptions between the function and its callers.

Learning Outcomes for this Module

- At the end of this module, you should be able to
  - Determine from the purpose statement of a function whether an invariant is necessary.
  - write invariants to document the assumptions that a function makes about its arguments.
  - explain how invariants divide responsibility between a function and its callers.

Module 07


Basic Principles
Designing Data
Designing Functions
Designing Systems
Tools and Techniques
Computing with Lists
Computing with Trees and Graphs
Computing with Higher-Order Functions
Designing with Invariants
Thinking about Efficiency
Object-Oriented Programming
Interfaces and Classes
Inheritance
Objects with Mutable State
Efficiency, Part 2

Learning Outcomes for Lesson 7.1

- At the end of this lesson, you should be able to
  - explain how invariants divide responsibility between a function and its callers
  - Use invariants to document how a function fits into the call tree of your program

Review: What does a contract mean?

- A function always gets to assume that its arguments satisfy its contract.
- It’s up to the callers of the function to guarantee that the function’s contract is satisfied at every call.

Sometimes the contract isn’t enough

- Sometimes a function needs more information than is available in a contract.
- We need to write down the additional information that the function needs.

Remember: one of our goals is to get information out of your head and onto the page.

Example

Imagine a ball bouncing back and forth in a closed box in the x direction. We might write something like this:

```
;; ball-normal-motion : Ball -> Ball
;; GIVEN: a Ball
;; RETURNS: the state of the ball after a
;; tick.
(define (ball-normal-motion b)
  (make-ball
    (+ (ball-x-pos b) BALLSPEED)))
```

But if the ball is about to bounce off the wall, this code does NOT return the correct state of the ball after the next tick– it doesn't account for the bouncing.

This function only fulfills its purpose statement if it is given a ball that doesn't bounce on the next tick.

Here’s how to document the assumptions that this function makes

```
;; ball-normal-motion : Ball -> Ball
;; GIVEN: a Ball
;; WHERE: the Ball is not going to
;; collide with a wall on this tick
;; RETURNS: the state of the ball after a
;; tick.
(define (ball-normal-motion b)
  (make-ball
    (+ (ball-x-pos b) BALLSPEED)))
```

The WHERE-clause

- The WHERE-clause is called an *invariant* or *precondition*. We will use both words interchangeably.
- It is an additional restriction (beyond the contract) on the inputs to the function.
- Like the contract, it limits the responsibility of the function to only those inputs that satisfy both the contract and the precondition.

Dividing Responsibilities

- The invariant, along with the contract, sets down the assumptions that each function makes about the arguments that it processes
- It is up to each caller of the function to make sure that the invariant is true at every call.
- The function gets to assume that the invariant is true.
- The function does not need to check that the invariant is true– indeed, often that is impossible.

This isn't completely new:

Here are some examples of **WHERE** clauses that we've seen (or might have seen) before:

```
-- A Ring is a (make-ring Real Real)
   WHERE inner < outer

-- An TelephoneBook is a ListOfEntries
   WHERE the entries are sorted by name
```

More examples of **WHERE** clauses

```
unpaused-world-after-tick
  : World -> World
GIVEN: a World
WHERE: the world is not paused
RETURNS: the state of the world after the next tick

ball-bounce-motion
  : Ball -> Ball
GIVEN: a Ball
WHERE: we know the ball will hit the wall on the next
       tick
RETURNS: the state of the ball after the next tick.
```

In each case, it is the responsibility of the caller to make sure the invariant is satisfied before the function is called.

And conversely, the function gets to assume that the invariant is satisfied.

Invariants can help us keep track of our assumptions

Consider our ball again. We might write something like this:

```
;; ball-after-tick : Ball -> Ball
;; GIVEN: the state of a ball
;; RETURNS: the state of the ball
;; after the next tick
(define (ball-after-tick b)
  (if (ball-would-hit-wall? b)
      (ball-after-bounce b)
      (ball-normal-motion b)))
```

Purpose statements for our helper functions

```
;; ball-would-hit-wall? : Ball -> Boolean
;; GIVEN: the state of a Ball
;; RETURNS: true iff the ball, in its normal motion, would hit the
;; wall on the next tick

;; ball-after-bounce : Ball -> Ball
;; GIVEN: the state of a Ball
;; WHERE: the ball, in its normal motion, would hit the wall on the
;; next tick
;; RETURNS: the state of the ball after the next tick

;; ball-normal-motion : Ball -> Ball
;; GIVEN: the state of a Ball
;; WHERE: the ball, in its normal motion, would not hit the wall on the
;; next tick
;; RETURNS: the state of the ball after the next tick
```

The invariant documents the fact that **ball-after-bounce** is only called if **ball-would-hit-wall**? returns **true**.

Similarly, the invariant documents the fact that **ball-normal-motion** is only called if **ball-would-hit-wall**? returns **false.**

Lesson Summary

- You should now be able to
  - explain how invariants divide responsibility between a function and its callers
  - Use invariants to document how a function fits into the call tree of your program

Next Steps

- Study 07-1-1-bouncing-ball.rkt in the Examples folder.
- If you have questions about this lesson, ask them on Piazza.
- Go on to the next lesson.