

# Reasoning and Floyd-Hoare Logic

CSE 331: Software Design and Implementation, Autumn 2026

*Your job is to deliver code that you know works.*

# TODO

- [Reading 2](#): due Friday, October 16, 11:59pm
- Homework 2: out by Friday, due Friday, October 23, 11:59pm
- Section next Tuesday: Floyd-Hoare logic

# So far

- **Lectures 1 to 3:** a specification is a claim about every run
  - A method's contract with its client; a whole program's manual
- **Lectures 4 to 6:** a test checks one chosen run against it
  - Coverage says when to stop; errors are part of the contract
- **Today:** reasoning, from the code itself, about every run at once

# seconds, the specification

```
6  /**
7   * Converts a duration in hours, minutes, and seconds to seconds.
8   * The result could overflow an int, so it is returned as a long.
9   *
10  * @param h the hours
11  * @param m the minutes
12  * @param s the seconds
13  * @requires 0 <= h, 0 <= m, and 0 <= s
14  * @return 3600 * h + 60 * m + s
15  */
```

# seconds, the body

```
16  static long seconds(int h, int m, int s) {  
17      long t = h;  
18      t = t * 60 + m;  
19      t = t * 60 + s;  
20      return t;  
21  }
```

- Three assignments and a **return**: no **if**, no loop
- Lines 16 to 21 on the back of your program sheet

# Three tests pass. Is seconds right?

```
6  @Test
7  void allZero() {
8      assertEquals(0, Seconds.seconds(0, 0, 0));
9  }
10
11 @Test
12 void oneOfEach() {
13     assertEquals(3661, Seconds.seconds(1, 1, 1));
14 }
15
16 @Test
17 void ninetyMinutes() {
18     assertEquals(5400, Seconds.seconds(0, 90, 0));
19 }
```

# From three tests to every run

- No `if` and no loop, so every test runs every line
  - Lecture 5's list is complete: at least two tests, every statement
- The specification promises an answer for **every allowed input**
  - Every non-negative `h`, `m`, and `s`
  - Far more than a loop could try in our lifetimes
- The three tests tried three
- **Lecture 1**: testing refutes a claim about every run, never confirms

# Reasoning

- To *know* it is right: Lecture 4's third way, **reasoning**
  - Argue from the code itself that it satisfies its specification
- Lecture 1 argued about `Sum` in prose; today we make it precise
  - Write down what is true after each line
  - Name the rule that justifies each step, so anyone can check it
- On code with no loops today; Friday's lecture adds them

# What is true after each line

```
16  static long seconds(int h, int m, int s) {  
    {{ 0 <= h and 0 <= m and 0 <= s }}
```

```
17  long t = h;  
    {{ 0 <= h and 0 <= m and 0 <= s and t = h }}
```

nothing about h, m, s changed, and now t = h too

```
18  t = t * 60 + m;  
    {{ 0 <= h and 0 <= m and 0 <= s and t = 60 * h + m }}
```

t = h is no longer true: the new t is the old one, h, times 60, plus m

```
19  t = t * 60 + s;  
    {{ 0 <= h and 0 <= m and 0 <= s and  
      t = 3600 * h + 60 * m + s }}
```

the same with s: the old t was 60 \* h + m

```
20  return t;
```

# Every allowed input, without a run

- Every run that starts with the precondition true returns the total
  - That is what it means to satisfy the specification
- Four claims showed it for every allowed input, without a run
- From a claim before a line, through it, to a claim after
  - We call that **forward reasoning**
- Each step used a rule; the rest of the lecture gives the rules
  - A step that breaks a rule is a mistake in the argument

# Integers, and Java's int

- The claims treat `*` and `+` as the operations on the integers,  $\mathbb{Z}$
- Java's `int` wraps around at the top of its range (Lecture 1)
  - The seconds in 596524 hours already pass it
- So `seconds` returns a `long`, and its inputs are `int` values
  - Every value computed on the way stays far below a `long`'s top
- So no line wraps, and the claims may use integer arithmetic
- Exercise 5 returns an `int`, with the clause its specification needs

# A claim at a point

- **Assertion:** a true or false claim about the program's state
  - At a particular point in the program
- It **holds** if the claim is true each time a run reaches its point
  - If a run ever reaches the point with it false, it does not hold

# Points some runs never reach

```
16  static int abs(int n) {  
17      if (n < 0) {  
18          n = -n;  
19      }  
20      return n;  
21  }
```

Only the runs with  $n < 0$  reach line 18: a claim there is about those runs

Every run reaches line 20, once: a claim there is about every run

- **Inside a branch:** only the runs that take it reach the point
- **Inside a loop:** a point reached many times on one run (Friday)

# A triple

- **Hoare triple**  $\{\{ P \}\}$  `code`  $\{\{ Q \}\}$ : a claim, code, and a claim
  - $P$  is its **precondition**,  $Q$  its **postcondition**
- **Valid**: every terminating run of the code
  - from a state in which  $P$  is true
  - ends in a state in which  $Q$  is true
- A run that never terminates: the definition asks nothing of it
  - Whether code terminates is a separate question (Friday)

# A method's specification is a triple

- Read the body as a triple: precondition before, postcondition after
- **Satisfies:** that triple is valid, and every allowed call terminates
  - Lecture 2's definition, in the triple's words
- `seconds` showed its triple valid

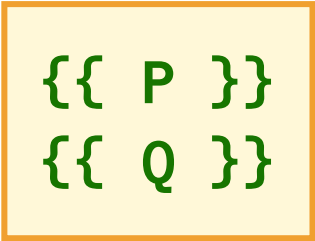
# Triples about one line

- $\{\{ \text{true} \}\} \ x = 5; \{\{ x = 5 \}\}$ 
  - Whatever the state was, the line makes  $x$  5: valid
- $\{\{ x = 3 \}\} \ y = x + 1; \{\{ y = 4 \}\}$ 
  - From any state where  $x$  is 3, the line makes  $y$  4: valid
- $\{\{ x = 3 \}\} \ y = x + 1; \{\{ 0 < y \}\}$ 
  - The line makes  $y$  4, and 4 is positive: valid too
  - A postcondition may say less than all that is true afterward

# More triples about one line

- $\{\{ \text{true} \}\} \ y = x + 1; \{\{ y = 4 \}\}$ 
  - $x$  may be 0, and then  $y$  is 1: a counterexample, not valid
- $\{\{ 0 < x \}\} \ x = x - 1; \{\{ 0 < x \}\}$ 
  - $x$  may be 1, and then  $x$  is 0: not valid
  - Valid needs  $Q$  true after a run from every state  $P$  allows
- $\{\{ \text{false} \}\} \ x = 0; \{\{ x = 1 \}\}$ 
  - No state makes  $\text{false}$  true, so nothing is asked: valid

# The empty triple



$\{\{ P \}\}$   
 $\{\{ Q \}\}$

- Running no code leaves the state as it was
- So the triple is valid exactly when Q is true wherever P is
- We say P **implies** Q
  - Lecture 2's words: P is *stronger* than Q, and Q *weaker*
- The last step of the **seconds** argument was an empty triple
  - The claim after line 19 contains the postcondition: it implies it

# A chain of implications

$$\begin{aligned} &\{\{ \emptyset \leq h \text{ and } \emptyset \leq m \text{ and } \emptyset \leq s \text{ and } t = h \}\} \\ &\{\{ \emptyset \leq h \text{ and } t = h \}\} \\ &\{\{ \emptyset \leq t \}\} \end{aligned}$$

- Claims written one under another at a point are empty triples
  - So they read as a chain: each claim implies the claim below it
- The second claim keeps two facts of the first
  - The third follows from the second
- A later rule, consequence, lets us write such a pair at any point

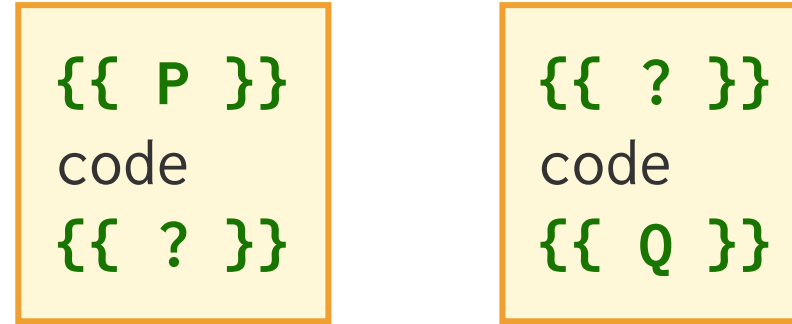
# Stronger and weaker

- Many claims hold at one point: after line 17,  $t = h$ ,  $0 \leq t$ ,  $true$
- Think of a claim as **the set of states in which it is true**
- **P implies Q**: P's set is inside Q's
  - P is the **stronger** claim, Q the **weaker**
- Stronger says more, and is true in fewer states
  - "Stronger" sounds bigger: easy to get backward
- The weakest claim of all is  $true$ ; the strongest is  $false$

# Which claim to write at a point

- Among the claims that **hold** there, one implies all the others
  - The strongest: exactly the states a run can be in at that point
- Suppose after line 18 we wrote  $0 \leq t$ : it holds, but says less
  - After line 19 we cannot say what  $t$  is in terms of  $h, m, s$
  - The argument fails, though the method satisfies its specification
- **Forward reasoning keeps the strongest claim at every step**
  - It weakens a claim only to check it against one someone wrote

# Two directions



- **Forward:** from the claim before, the strongest claim after
  - The **strongest postcondition** from P: it makes the triple valid
- **Backward:** from the claim after, the weakest claim before
  - The **weakest precondition** for Q
- Both keep every state the code could be in, no more and no fewer

# Either way, the last step is an implication

- Given  $\{\{ P \}\}$  code  $\{\{ Q \}\}$ :
- **Forward** from P to the strongest claim R after the code
  - $\{\{ P \}\}$  code  $\{\{ R \}\}$  is valid because we found R that way
  - So the triple is valid exactly when R implies Q
- **Backward** from Q to the weakest claim R before the code
  - $\{\{ R \}\}$  code  $\{\{ Q \}\}$  is valid because we found R that way
  - So the triple is valid exactly when P implies R

# The rules

- Each rule is a **schematic**: a triple laid out as in a program
  - With letters for the parts that change from one use to the next
- How a claim moves through an assignment, a sequence, and an **if**
- First the assignment: the only kind of line in **seconds** before **return**

# Rule: assignment, forward

```
{ { P } }  
x = e;  
{ { P[x ↦ x0] and x = e[x ↦ x0] } }
```

- $x$  is any variable;  $e$  any expression that changes no variable
- $P[x \mapsto x_0]$  is  $P$  with every  $x$  replaced by  $x_0$ 
  - A fresh name for the value  $x$  had before the line
  - A name that neither the program nor any claim uses
- The claim after the line is the strongest that makes it valid

# The rule on lines 18 and 19

```
{ { t = h } }  
t = t * 60 + m;  
{ { t0 = h and t = t0 * 60 + m } }  
t = t * 60 + s;  
{ { t0 = h and t1 = t0 * 60 + m and t = t1 * 60 + s } }
```

- Start from  $t = h$ , the one fact after line 17 that mentions  $t$
- At line 19 the rule renames  $t$  again, and  $t0$  is taken
  - So the fresh name is  $t1$
- Each use picks a fresh name, so no two old values share a name

# The facts chain back to h

```
{{ t = h }}
```

```
t = t * 60 + m;
```

```
{{ t0 = h and t = t0 * 60 + m }}
```

```
t = t * 60 + s;
```

```
{{ t0 = h and t1 = t0 * 60 + m and t = t1 * 60 + s }}
```

```
{{ t1 = h * 60 + m and t = t1 * 60 + s }}
```

```
{{ t = (h * 60 + m) * 60 + s }}
```

```
{{ t = 3600 * h + 60 * m + s }}
```

# Two habits keep the fresh names out

- If  $P$  and  $e$  never mention  $x$ , nothing has to be renamed
  - The claim after the line is  $P \text{ and } x = e$ : line 17
- If we can write the old value of  $x$  without  $x_0$ 
  - Put that expression for  $x_0$  in every other fact
  - The one fact still mentioning  $x_0$  says nothing of the state: drop it
  - Line 18, with  $h$  for  $t_0$ ; after  $x = x + 1$ , the old  $x$  is  $x - 1$
- A fact about  $w$  renames too:  $w = x + 1$  becomes  $w = x_0 + 1$

# Rule: assignment, backward

$$\begin{array}{l} \{\{ Q[x \mapsto e] \}\} \\ x = e; \\ \{\{ Q \}\} \end{array}$$

- $x$  is any variable;  $e$  any expression that changes no variable
- $Q[x \mapsto e]$  is  $Q$  with every  $x$  replaced by  $e$
- The claim before the line is the weakest that makes it valid
- Read it from the bottom up:  $e$  is what  $x$  is about to become
  - So what  $Q$  claims of  $x$  after had better be true of  $e$  before

# Backward through seconds

```

    {{ 0 <= h and 0 <= m and 0 <= s }}
    {{ true }}
    {{ 3600 * h + 60 * m + s = 3600 * h + 60 * m + s }}
    {{ (h * 60 + m) * 60 + s = 3600 * h + 60 * m + s }}
17  long t = h;
    {{ (t * 60 + m) * 60 + s = 3600 * h + 60 * m + s }}
18  t = t * 60 + m;
    {{ t * 60 + s = 3600 * h + 60 * m + s }}
19  t = t * 60 + s;
    {{ t = 3600 * h + 60 * m + s }}
```

- Three substitutions, from the postcondition up
- Then the chain up to the precondition: the verdict

# Meeting in the middle

```
17      long t = h;  
        {{ 0 <= h and 0 <= m and 0 <= s and t = h }}  
        {{ (t * 60 + m) * 60 + s = 3600 * h + 60 * m + s }}  
18      t = t * 60 + m;
```

- Forward from the precondition through line 17: the upper claim
- Backward from the postcondition through lines 19 and 18: the lower
- Valid exactly when the upper claim implies the lower: does it?

# The check, as a calculation

$$\begin{aligned}(t * 60 + m) * 60 + s &= (h * 60 + m) * 60 + s && \text{since } t = h \\ &= 3600 * h + 60 * m + s\end{aligned}$$

- One step to a line, with the fact a step uses beside it
- The backward claim's left side comes out as its right side
  - So the claim is true wherever  $t = h$

# Half sheet, question 2 (3 minutes)

- 2. Fill in the weakest claim:  $\{\{ \text{ ? } \}\} \ x = x + 1; \{\{ x = 4 \}\}$ 
  - Then: why is backward reasoning sometimes simpler than forward?
- Hand it in at the door as you leave

# Rule: sequence

$\{\{ P \}\}$

first

$\{\{ R \}\}$

second

$\{\{ Q \}\}$

- Valid exactly when some claim R between makes both halves valid
- Forward reasoning: R is the strongest claim after the first piece
- Backward reasoning: R is the weakest claim before the second
- Each argument about **seconds** used this rule twice

# Where the rules stop

- Each assignment rule gives one claim
  - The strongest after the line, or the weakest before it
- **The claim we have to reach is often another claim**
  - A specification may promise less than the strongest
  - Or require more than the weakest
- Forward through **seconds**: the rules gave the claim after line 19
  - And we had to reach the postcondition

# Rule: consequence

$\{\{ P \}\}$

$\{\{ A \}\}$

code

$\{\{ B \}\}$

$\{\{ Q \}\}$

- Valid if  $P$  implies  $A$ ,  $\{\{ A \}\}$  code  $\{\{ B \}\}$  is valid, and  $B$  implies  $Q$
- Two claims at one point: the empty triple, upper implies lower
- Follow a run down:  $P$  true, so  $A$  true; then  $B$  true, so  $Q$  true
- The code may assume less and promise more than its triple asks

# The rule on seconds, forward

```
{ { 0 <= h and 0 <= m and 0 <= s and  
    t = 3600 * h + 60 * m + s } }  
{ { t = 3600 * h + 60 * m + s } }  
return t;
```

- Upper: the rules' claim after line 19. Lower: the postcondition
- The lower is one fact of the upper, so the upper implies it
  - Weakening by dropping facts, as the chain of implications did
- Where the directions met, the same rule, through  $t = h$

# Where an argument can go wrong

- The two implications are **not mechanical**: arithmetic and logic
- They are where an argument goes wrong: a false one we believe
- The checks that closed both `seconds` arguments were this rule
  - So was the implication where the directions met
- So was writing `0 <= t` after line 18: it broke no rule
  - It dropped a fact the argument needed to reach the postcondition

# Stronger for whom

- **Precondition:** the client's promise; the implementer may assume it
  - Stronger: the client makes more true, the implementer handles less
- **Postcondition:** the implementer's promise; the client may rely on it
  - Stronger: the implementer delivers more, the client relies on more
- Each party wants its own promise weak and the other party's strong
- Lecture 2's stronger specification: weaker before, stronger after
  - The client smiles twice; the right strength is a design choice

# Across a call

- **Lecture 2** reasoned across a call with the specification
  - The precondition before the call, the call, the postcondition
- When the method satisfies its specification, that triple is valid
  - The argument never reads a line of the method's body
- To use it: the claim before the call must imply the precondition
  - The precondition is the client's promise; this checks it is kept
- After the call, the postcondition is part of what we know

# abs

```
9    /**
10   * Gives the absolute value of a number.
11   *
12   * @param n the number
13   * @requires -2147483647 <= n
14   * @return n if 0 <= n, and -n if n < 0
15   */
```

- The precondition keeps out one `int`, -2147483648; why at the end

# What is true after the if?

```
16  static int abs(int n) {  
17      if (n < 0) {  
18          n = -n;  
19      }  
20      return n;  
21  }
```

- In the true branch,  $n < 0$ , and  $n = -n$  makes  $n$  positive
- The false branch is empty, and in it,  $0 \leq n$
- After the `if`:  $0 < n$  or  $0 \leq n$ , which is  $0 \leq n$

# Rule: conditional

```
{{ P }}  
if (b) {  
    {{ P and b }}  
    first  
    {{ A }}  
} else {  
    {{ P and not b }}  
    second  
    {{ B }}  
}  
{{ A or B }}  
{{ Q }}
```

- $b$  is the `if`'s condition; A and B the strongest claims after each branch

# The rule, read

- A run entering the true branch found  $b$  true; the false,  $b$  false
  - So those facts hold at the top of their branches
- After the  $if$ , a run has taken one branch or the other
  - It ends with  $A$  true, or with  $B$  true: the strongest claim is  $A$  or  $B$
- $Q$  has to be true whichever branch ran: check  $Q$  against both
- An  $if$  with no  $else$  still has a false branch (Lecture 5)
  - It is empty, so  $B$  is  $P$  and not  $b$

# Is $0 \leq n$ enough?

- No: the specification promises more than a non-negative number
  - The number returned is the number given, or its negation
- Line 18 overwrites  $n$ : the claims need a name for the number given
  - Let  $n_0$  be the value of  $n$  at the call
  - When the specification says  $n$ , it means  $n_0$
- The precondition keeps  $n = -n$  from wrapping: no claim needs more
  - So the top claim is  $n = n_0$ : one value for the whole argument

# abs, the whole argument

```
16  static int abs(int n) {  
    {{ n = n0 }}  
17  if (n < 0) {  
    {{ n = n0 and n < 0 }}  
18    n = -n;  
    {{ n0 < 0 and n = -n0 }}  
19  }  
    {{ (n0 < 0 and n = -n0) or (0 <= n0 and n = n0) }}  
20  return n;  
21 }
```

# abs, the two cases

- First case:  $n0 < 0$  and  $n = -n0$ ; the specification promises  $-n0$
- Second case:  $0 \leq n0$  and  $n = n0$ ; it promises  $n0$
- Both implications are true, so **abs** satisfies its specification

# compare

```
6  /**
7   * Compares two numbers.
8   *
9   * @param a one number
10  * @param b the other number
11  * @return -1 if a < b, 0 if a == b, and 1 if b < a
12  */
```

- Two `ifs`, and the body of each is a `return`

# A return inside an if

- A `return` ends the run: no run comes out of line 15 to after the `if`
- Only the false branch leads there: the claim is `P and not b`
- No run reaches the end of the true branch
  - So every claim holds there, `false` among them: no run reaches it
- And if the claim at a point implies `false`, no run reaches it
  - Reasoning has found code that never runs (exercise 3)
- At each `return`, check the postcondition's case

# compare, the argument

```
13  static int compare(int a, int b) {  
    {{ true }}  
14  if (a < b) {  
    {{ a < b }}  
15      return -1;  
16  }  
    {{ b <= a }}  
17  if (a == b) {  
    {{ b <= a and a = b }}  
18      return 0;  
19  }  
    {{ b <= a and a != b }}  
20  return 1;  
21 }
```

# compare, the three checks

- Line 15:  $a < b$ , and the specification promises -1
- Line 18:  $a = b$ , and it promises 0
- Line 20:  $b \leq a$  and  $a \neq b$  say  $b < a$ , and it promises 1
- All three implications are true: `compare` satisfies its specification

# Reasoning finds the mistake: min

```
14  static int min(int a, int b) {  
15      int r = a;  
16      if (a <= b) {  
17          r = a;  
18      }  
19      return r;  
20  }
```

```
6  @Test  
7  void firstIsSmaller() {  
8      assertEquals(1, Min.min(1, 2));  
9  }
```

# min, forward from true

```
14  static int min(int a, int b) {  
    {{ true }}  
15    int r = a;  
    {{ r = a }}  
16    if (a <= b) {  
        {{ r = a and a <= b }}  
17        r = a;  
        {{ r = a and a <= b }}  
18    }  
    {{ (r = a and a <= b) or (r = a and b < a) }}  
    {{ r = a and (a <= b or b < a) }}  
    {{ r = a }}  
19    return r;  
20 }
```

# min, the check

- $r = a$  is in both cases, and the rest is true of any two numbers
  - So the claim after the `if` collapses to  $r = a$
- The method returns  $a$  on every run; check that against the promise
- The postcondition promises  $a$  when  $a \leq b$ , and  $b$  when  $b < a$ 
  - When  $b < a$ , the promise was  $b$ : false exactly on those states
- A run from any such state is a counterexample; `min(2, 1)` is one
- The argument says on which inputs the method is wrong

# quadrant

```
10  *           y
11  *         2   |   1
12  *  -----+----- x
13  *         3   |   4
14  *
15  * @param x the point's x coordinate
16  * @param y the point's y coordinate
17  * @requires x != 0 and y != 0
18  * @return 1 if x > 0 and y > 0,
19  *         2 if x < 0 and y > 0,
20  *         3 if x < 0 and y < 0, and
21  *         4 if x > 0 and y < 0
```

- Lecture 5's two tests complete the list, and both pass

# quadrant, the body

```
23  static int quadrant(int x, int y) {  
24      int q = 1;  
25      if (x < 0) {  
26          q = 2;  
27      }  
28      if (y < 0) {  
29          q = q + 1;  
30      }  
31      return q;  
32  }
```

- Two `ifs` in a row: after each, a claim with a case per branch taken

# quadrant, forward

```
int q = 1;
if (x < 0) {
    q = 2;
}
if (y < 0) {
    q = q + 1;
}
return q;
```

```
23 static int quadrant(int x, int y) {
    {{ x != 0 and y != 0 }}
24     int q = 1;
    {{ x != 0 and y != 0 and q = 1 }}
25     if (x < 0) {
        {{ x < 0 and y != 0 and q = 1 }}
26         q = 2;
        {{ x < 0 and y != 0 and q = 2 }}
27     }
    {{ (x < 0 and y != 0 and q = 2) or
        (0 < x and y != 0 and q = 1) }}
```

```
    {{ (x < 0 and y != 0 and q = 2) or
        (0 < x and y != 0 and q = 1) }}
28     if (y < 0) {
        {{ (x < 0 and y < 0 and q = 2) or
            (0 < x and y < 0 and q = 1) }}
29         q = q + 1;
        {{ (x < 0 and y < 0 and q = 3) or
            (0 < x and y < 0 and q = 2) }}
30     }
    {{ (x < 0 and y < 0 and q = 3) or
        (0 < x and y < 0 and q = 2) or
        (x < 0 and 0 < y and q = 2) or
        (0 < x and 0 < y and q = 1) }}
    quadrant 4 is promised, and q = 2: this implication is false
31     return q;
32 }
```

# Four cases, four lines of @return

- $0 < x$  and  $0 < y$ : quadrant 1 is promised, and  $q = 1$
- $x < 0$  and  $0 < y$ : quadrant 2 is promised, and  $q = 2$
- $x < 0$  and  $y < 0$ : quadrant 3 is promised, and  $q = 3$
- $0 < x$  and  $y < 0$ : quadrant 4 is promised, and  $q = 2$
- **The fourth implication is false**, exactly on the lower right
  - `quadrant(1, -1)` is there, and the method returns 2 on it
  - Lecture 5's list never asked for that test

# Paths and cases

- The four cases are the four paths through **quadrant**
- A test follows a path on one input
- A case covers every input that takes the path

# Rule: conditional, backward

```
{{ (b and A) or (not b and B) }}  
if (b) {  
    {{ A }}  
    first  
    {{ Q }}  
} else {  
    {{ B }}  
    second  
    {{ Q }}  
}  
{{ Q }}
```

- A and B: the weakest claims before each branch for Q

# quadrant, backward

```
int q = 1;
if (x < 0) {
    q = 2;
}
if (y < 0) {
    q = q + 1;
}
return q;
```

```

    {{ (y < 0 and ((x < 0 and q = 2) or (0 < x and q = 3))) or
      (0 < y and ((0 < x and q = 1) or (x < 0 and q = 2))) }}
28   if (y < 0) {
    {{ (x < 0 and q = 2) or (0 < x and q = 3) }}
29       q = q + 1;
    {{ (x < 0 and q = 3) or (0 < x and q = 4) }}
30   }
    {{ (0 < x and 0 < y and q = 1) or (x < 0 and 0 < y and q = 2) or
      (x < 0 and y < 0 and q = 3) or (0 < x and y < 0 and q = 4) }}
31   return q;
32 }
```

```
23 static int quadrant(int x, int y) {
    {{ x != 0 and y != 0 }}
    does not imply the claim below: 0 < x, y < 0 is a counterexample
    {{ (x < 0 and y != 0) or (0 < x and 0 < y) }}
24   int q = 1;
    {{ (x < 0 and y != 0) or
      (0 < x and ((y < 0 and q = 3) or (0 < y and q = 1))) }}
25   if (x < 0) {
    {{ y != 0 }}
26       q = 2;
    {{ (y < 0 and ((x < 0 and q = 2) or (0 < x and q = 3))) or
      (0 < y and ((0 < x and q = 1) or (x < 0 and q = 2))) }}
27   }
    {{ (y < 0 and ((x < 0 and q = 2) or (0 < x and q = 3))) or
      (0 < y and ((0 < x and q = 1) or (x < 0 and q = 2))) }}
```

# The inputs that work

- The claim at the top: the weakest precondition of the body
  - Exactly the inputs on which `quadrant` returns what it promises
- The precondition allows more:  $0 < x$  and  $y < 0$ , the lower right
  - There the precondition is true and the claim false
- So the precondition does not imply the claim: not valid
  - On exactly the inputs forward reasoning found
- Forward found the case that fails; backward, the inputs that work

# A proof

- An argument from what is given to what is claimed, in steps
  - Each step justified by a rule, so anyone can check it: a **proof**
- The arguments for `seconds`, `abs`, and `compare` are proofs
  - The given is the precondition, the claim the postcondition
  - The steps use the rules
- A proof that a method satisfies its specification: every run at once

# An argument says where it fails

- Unlike Lecture 1's prose argument about `Sum`
  - An argument built from the rules says where it fails
- When every claim is the strongest, the false implication says
  - On which inputs the method is wrong, as for `min` and `quadrant`

# A way of thinking

- Proving a whole program correct this way: very hard, rarely done
- What Floyd and Hoare taught us: **a program is a mathematical object**
  - So a programmer can ask what it is supposed to do
  - And whether they could prove that it does that
- In this lecture's terms, the question asked at every line above
  - What is true here, and what is true after the next line?

Leslie Lamport, "The Future of Computing: Logic or Biology" (2003), one of Reading 2's articles.

# What reasoning tells us

- **Lecture 5 closed:** tests that complete the list and pass are enough
  - That is still less than knowing the method is correct
- A proof that the specification's triple is valid is that knowledge
- A test confirms a claim about its own run
  - That the triple is valid is a claim about every run
- **Lecture 1:** reasoning can confirm it, if the argument is right

# What can still go wrong

- **A false implication:** a slip in a rule shows on the page
  - A false implication we believe does not
- Lecture 1's catch: a refuting test refutes the argument too
- So we still test

# Arithmetic that is not Java's

- The claims are about the integers,  $\mathbb{Z}$ , and `int` wraps
- `abs` without its precondition: `n = -n` leaves -2147483648 as it is
  - `abs(-2147483648)` returns -2147483648!
- Each method today has such a precondition, as `abs` does
  - Or none that could wrap: `seconds` computes in a `long`

# And a loop

- Every method today ran each line at most once
- The rules do not carry a claim through a loop
- Friday's lecture gives the rule for a loop
  - It needs one claim from the programmer
  - A claim the rule itself does not produce

*Your job is to deliver code that you know works.*

# Coming up

- **Friday:** Lecture 8, loop invariants
- **Reading 2:** due Friday, October 16, 11:59pm
- **Monday:** the Reading 2 discussion
- **Tuesday:** section, Floyd-Hoare logic
- Homework 2: out by Friday, due Friday, October 23, 11:59pm
- Hand in your half sheet at the door