Recall from last time...

Is this solution **correct**?

```java
int indexOfMaximum(int[] arr, int n) {
    int maxValue = arr[0];
    int maxIndex = 0;
    for (int i = 1; i < n; i++) {
        if (arr[i] > maxValue) {
            maxValue = arr[i];
            maxIndex = i;
        }
    }
    return maxIndex;
}
```
Reasoning about code

**Idea:** determine what *facts* are true at each line of the program

- We would like to know:
  - at the end, `maxIndex` is index of the maximum element
  - at the end, negatives before zeros before positives in `arr`

- Get there by understanding what is true at each line until end
  - then check that those facts that are true at the end include all the things we require
Why do this?

• Essential for building **high quality** programs
  – allows us to inspect code to check correctness
  – need all three: tools, *inspection*, & testing
  – inspection is even the most effective of the three

• Essential for building **high complexity** programs
  – allows us to build modular programs
    • each module has assumptions about how it will be used
  – misunderstandings btw module writers will cause bugs
  – assumptions must be clearly stated (and inspected)
Approaches

- We will discuss two approaches
  - forward reasoning: start at the top and work down
  - backward reasoning: start at the end and work up

- Plan:
  1. intuitive version (by example)
  2. formal definitions & rules
Example of Forward Reasoning

Suppose we initially know (or assume) \( w \geq 1 \)

\[
x = 2 \times w;
\]

\[
y = x + 2;
\]

\[
z = y / 2;
\]

What can we say at the end about \( z \)?
Example of Forward Reasoning

Suppose we initially know (or assume) $w \geq 1$

\[
x = 2 \times w;
\]
\[
// \ x \geq 2 \times 1 = 2
\]
\[
y = x + 2;
\]
\[
z = y / 2;
\]

What can we say at the end about $z$?
Example of Forward Reasoning

Suppose we initially know (or assume) $w \geq 1$

\[
x = 2 \times w; \\
// \ x \geq 2 \times 1 = 2
\]

\[
y = x + 2; \\
// \ y \geq 2 + 2 = 4
\]

\[
z = y / 2;
\]

What can we say at the end about $z$?
Example of Forward Reasoning

Suppose we initially know (or assume) \( w \geq 1 \)

\[
x = 2 \times w;
\]
\[
// \ x \geq 2 \times 1 = 2
\]
\[
y = x + 2;
\]
\[
// \ y \geq 2 + 2 = 4
\]
\[
z = y / 2;
\]
\[
// \ z \geq 4 / 2 = 2
\]

What can we say at the end about \( z \)? \( z \geq 2 \)
Forward Reasoning

• Forward reasoning:
  – informally, simulates the code (for all inputs at once)
  – formally, determine what follows from initial assumptions

• This is the way most programmers *inspect* their code

• Advantages and disadvantages:
  – intuitive
  – introduces (many) irrelevant facts
Example of Backward Reasoning

Suppose we want to show that $z \geq 1$ (at the end)
What needs to be true about $w$?

$$x = 2 \times w;$$

$$y = x + 2;$$

$$z = y / 2;$$

// $z \geq 1$
Example of Backward Reasoning

Suppose we want to show that $z \geq 1$ (at the end)
What needs to be true about $w$?

```
x = 2 * w;
y = x + 2;
// y / 2 \geq 1 or equivalently y \geq 2
z = y / 2;
// z \geq 1
```
Example of Backward Reasoning

Suppose we want to show that $z \geq 1$ (at the end)
What needs to be true about $w$?

\[
\begin{align*}
x &= 2 \times w; \\
&\quad // \ x + 2 \geq 2 \ or \ equivalently \ x \geq 0 \\
y &= x + 2; \\
&\quad // \ y / 2 \geq 1 \ or \ equivalently \ y \geq 2 \\
z &= y / 2; \\
&\quad // \ z \geq 1
\end{align*}
\]
Example of Backward Reasoning

Suppose we want to show that $z \geq 1$ (at the end)
What needs to be true about $w$?

```plaintext
// 2 * w >= 0 or equivalently w >= 0
x = 2 * w;
// x + 2 >= 2 or equivalently x >= 0
y = x + 2;
// y / 2 >= 1 or equivalently y >= 2
z = y / 2;
// z >= 1
```
Backward Reasoning

• Backward reasoning:
  – determines sufficient conditions for an end result
    • e.g., assumptions needed for correctness

• Advantages and disadvantages:
  – less intuitive
  – determines exactly what is necessary to achieve the goal
  – gives you another (powerful) way to reason about code
Our approach

• We will take a **methodical** approach to reasoning about code
  – spell everything out in detail to avoid any misunderstanding
  – (you can move more quickly as you get practice)

• Hoare Logic
  – named after its inventor, Sir Anthony Hoare (inventor of quicksort)
  – considers just assignments, if-statements, and while-loops
    • everything else can be built out of these
  – we will consider just integer-valued variables
    • for Java, we will need floats, strings, objects, etc.

• This lecture: assignments & if-statements; Next lecture: loops
Terminology

• The *program state* is the values of all the (relevant) variables

• An *assertion* is a logical formula referring to the program state (e.g., contents of variables) at a given point

• An assertion *holds* for a program state if the formula is true when those values are substituted for the variables

• An assertion before the code is a *precondition*  
  – these represent assumptions about when that code is used

• An assertion after the code is a *postcondition*  
  – these represent what we want the code to accomplish
Notation

• Instead of writing assertions as comments, Hoare logic uses {..}
  – since Java code also has {..}, I will use {{...}}
  – e.g., {{ w >= 1 }} x = 2 * w; {{ x >= 2 }}

• Assertions are math not Java
  – you can use the usual math notation
    • (e.g., = instead of == for equals)
  – purpose is communication with other humans (not computers)
  – we will need and, or, not as well
    • can also write use ∧ (and) ∨ (or) etc.

• The Java language also has assertions (assert statements)
  – throws an exception if the condition does not evaluate true
  – we will discuss these more later in the course
Hoare Logic

• A Hoare triple is two assertions and one piece of code:
  \[ \{ \{ P \} \} \ S \ \{ \{ Q \} \} \]
  – \( P \) the precondition
  – \( S \) the code
  – \( Q \) the postcondition

• A Hoare triple \( \{ \{ P \} \} \ S \ \{ \{ Q \} \} \) is called valid if:
  – in any state where \( P \) holds, executing \( S \) produces a state where \( Q \) holds
  – i.e., if \( P \) is true before \( S \), then \( Q \) must be true after it
  – otherwise the triple is called invalid
Do programmers really do this?

“Warren [Buffet] often talks about these discounted cash flows, but I’ve never seen him do one.”
-- Charlie Munger

• Programmers rarely spell it out in this much detail
  – like Buffet, they usually just do it in their heads

• But there are some key exceptions
  – extremely tricky code
  – loops (next lecture)
  – preconditions for methods
Example 1

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

\[
\{ x \neq 0 \} \quad y = x \times x; \quad \{ y > 0 \}
\]
Example 1

Is the following Hoare triple valid or invalid?

– assume all variables are integers and there is no overflow

`{{ x != 0 }} y = x*x; {{ y > 0 }}`

Valid

• `y` could only be zero if `x` were zero (which it isn’t)
Example 2

Is the following Hoare triple valid or invalid?
  – assume all variables are integers and there is no overflow

\[\{\{ z \neq 1 \}\} \ y = z \times z; \ \{\{ y \neq z \}\}\]
Example 2

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

\[
\{ z \neq 1 \} \quad y = z*z; \quad \{ y \neq z \}
\]

Invalid

- counterexample: \( z = 0 \)
Example 3

Is the following Hoare triple valid or invalid?
- assume all variables are integers and there is no overflow

\[
\{\{ x \geq 0 \}\} \; y = 2*x; \; \{\{ y > x \}\}
\]
Example 3

Is the following Hoare triple valid or invalid?
- assume all variables are integers and there is no overflow

```
{{ x >= 0 }} y = 2*x; {{ y > x }}
```

Invalid
- counterexample: \( x = 0 \)
Example 4

Is the following Hoare triple valid or invalid?

```
{{
  if (x > 7) {
    y = 4;
  } else {
    y = 3;
  }
}{{y < 5}}
```
Example 4

Is the following Hoare triple valid or invalid?

{{ }}
if (x > 7) {
    y = 4;
} else {
    y = 3;
}
{{ y < 5 }}

Valid
• y is either 3 or 4; in either case, it is less than 5
Example 5

Is the following Hoare triple valid or invalid?

\[
\begin{align*}
\{ & \} \\
x & = y \\
z & = x \\
\{ & y = z \}
\end{align*}
\]
Example 5

Is the following Hoare triple valid or invalid?

\[
\{\{ \}
  \quad x = y;
  \quad z = x;
\}
\{\{ y = z \}\}
\]

Valid
Example 6

Is the following Hoare triple valid or invalid?

\[
\{\{ x = 7 \ \text{and} \ y = 5 \}\}
// \text{swap x and y}
tmp = x;
x = tmp;
y = x;
\{\{ x = 5 \ \text{and} \ y = 7 \}\}
\]
Example 6

Is the following Hoare triple valid or invalid?

\[
\{\{ x = 7 \text{ and } y = 5 \}\} \quad // \text{ swap } x \text{ and } y
\]
\[
tmp = x;
\]
\[
x = tmp;
\]
\[
y = x;
\]
\[
\{\{ x = 5 \text{ and } y = 7 \}\}\]

Invalid

• first two lines leave \( x \) unchanged, so we get \( x = y = 7 \)
The general rules

- Some of these require some thought
  - it would be preferable to do this without (much) thought
  - fortunately, there is a “turn the crank” way of doing these

- For each kind of construct, there is a general rule
  - assignment statements
  - two statements in sequence
  - conditionals
  - loops (next lecture)
Assignment Rule

\[
\{\{ P \} \} \ x = e; \ \{\{ Q \} \}
\]

- Let \( Q[x=e] \) be like \( Q \) except replace every \( x \) with \( e \)
  - after "\( x = e; \)”, \( Q \) and \( Q[x=e] \) are equivalent
  - but \( Q[x=e] \) does not involve \( x \) so it holds after "\( x = e; \)" if and only if it holds before
  - so we can consider \( P \) and \( Q[x=e] \) w/out the assignment

- This triple is valid iff: whenever \( P \) holds, \( Q[x=e] \) also holds
  - in logic, we’d say it is valid if \( P \) implies \( Q[x=e] \)
Assignment Rule Example

\[
\{\{ z > 34 \} \} \ y = z + 1; \ \{\{ y > 1 \} \}
\]

- \( Q[y=z+1] \) is \( z + 1 > 1 \)
  - this is equivalent to \( z > 0 \)
  - whenever \( z > 34 \), we also have \( z > 0 \)
  - this is valid
Sequence Rule

\[
{\{ P \}} \ S_1; S_2 \ {\{ Q \}}
\]

• Triple is valid iff: there is an assertion \( R \) such that
  - \( {\{ P \}} \ S_1 \ {\{ R \}} \) is valid and
  - \( {\{ R \}} \ S_2 \ {\{ Q \}} \) is valid

• For now, we will need to guess \( R \)
  - we will see shortly that we can find an \( R \) without guessing
Sequence Rule Example

\[
\{\{ z \geq 1 \}\} \ y = z+1; \ w = y*y; \ \{\{ w > y \}\}
\]

- Choose \( R \) to be \( y > 1 \)
- Show \( \{\{ z \geq 1 \}\} \ y=z+1; \ \{\{ y > 1 \}\} \)
  - use assignment rule: \( z \geq 1 \) implies \( z+1 > 1 \)?
  - equivalently, \( z \geq 1 \) implies \( z > 0 \)? Valid.
- Show \( \{\{ y > 1 \}\} \ w=y*y; \ \{\{ w > y \}\} \)
  - use assignment rule: \( y > 1 \) implies \( y*y > y \)
  - requires some thought, but valid
- Both of these are triples valid, so the triple at the top is valid
Conditional Rule

\[
\{\{ P \}\} \text{ if (b) } \{S1\} \text{ else } \{S2\} \{\{ Q \}\}
\]

• When S1 executes, we know \( P \) and \( b \)
• When S2 executes, we know \( P \) and not \( b \)

• Triple is valid iff: there are assertions \( Q_1 \) and \( Q_2 \) such that
  – \( \{\{ P \text{ and } b \}\} \) S1 \( \{\{ Q_1 \}\} \) is valid and
  – \( \{\{ P \text{ and not } b \}\} \) S2 \( \{\{ Q_2 \}\} \) is valid and
  – \( Q_1 \) or \( Q_2 \) implies \( Q \)
    • we only know that one holds (which depends on \( b \))
Conditional Rule

```c
{{ }} if (x > 7) {y=x;} else {y=20;} {{ y > 5 }}
```

- Let \( Q_1 \) be \( y > 7 \) (other choices work too)
  - use assignment rule to show \( {{ x > 7 }} y=x; {{ y > 7 }} \)
- Let \( Q_2 \) be \( y = 20 \) (other choices work too)
  - use assignment rule to show \( {{ x <= 7 }} y=20; {{ y = 20 }} \)
- Check that \( y > 7 \) or \( y = 20 \) implies \( y > 5 \)
Weaker vs Stronger

If “whenever $P_1$ holds, $P_2$ also holds”, then:
- $P_1$ is called **stronger** than $P_2$
- $P_2$ is called **weaker** than $P_1$

- It is more (or at least as) “difficult” to satisfy $P_1$
  - the program states where $P_1$ holds are a subset of the states where $P_2$ holds
- $P_1$ puts more constraints on program states
- $P_1$ is a stronger set of requirements

- We do not always have $P_1$ stronger than $P_2$ or vice versa!
  - most assertions are incomparable
Examples

• \( x = 17 \) is stronger than \( x > 0 \)

• \( x \) is prime is neither stronger nor weaker than \( x \) is odd
  – these two statements are incomparable

• \( x \) is prime and \( x > 2 \) is stronger than
  \( x \) is odd and \( x > 2 \)

• Many other examples...
Applications to Method Design

• When writing a method, you decide the preconditions
  – e.g., a parameter may be assumed positive
  – e.g., an array may be assumed to be non-empty

• There are advantages and disadvantages to weaker vs stronger
  – stronger preconditions make the code easier to change
    • there are more allowed implementations
  – weaker preconditions allow more uses
    • there are more allowed calls
  – stronger preconditions may make the code easier to write
  – weaker preconditions are necessary for libraries

• We will discuss this more later on…
Applications to Hoare Logic

• Suppose:
  – \( \{ P \} S \{ Q \} \) is valid and
  – some \( P_1 \) is stronger than \( P \) and
  – some \( Q_1 \) is weaker than \( Q \)

• Then these are all valid too:
  – \( \{ P_1 \} S \{ Q \} \)
    • a state where \( P_1 \) holds is one where \( P \) also holds
  – \( \{ P \} S \{ Q_1 \} \)
    • a state where \( Q \) holds is one where \( Q_1 \) also holds
  – \( \{ P_1 \} S \{ Q_1 \} \)
Example Applications to Hoare Logic

\{\{ x \geq 0 \}\} y = x + 1; \{\{ y > 0 \}\}

- We know this is valid by the assignment rule

- Let \( P_1 \) be \( x > 0 \)
  - stronger since \( x \geq 0 \) implies \( x > 0 \)

- Let \( Q_1 \) be \( y \geq 0 \)
  - weaker since \( y \geq 0 \) implies \( y > 0 \)

- Thus, the following is also valid:

\{\{ x > 0 \}\} y = x + 1; \{\{ y \geq 0 \}\}
Weakest preconditions

- Suppose we know $Q$ and $S$
- There are potentially many $P$ such that ${\{P\}} S {\{Q\}}$ is valid
- Would be ideal if there were a unique weakest precondition $P$
  - most general assumptions under which $S$ makes $Q$ hold
  - get a valid triple for $P_1$ if and only if $P_1$ implies $P$
- Amazingly, without loops, for any $S$ and $Q$, this exists!
  - we denote this by $wp(S,Q)$
  - can be found by general rules
- Allows you to reason backward without any guessing
  - just as you do with forward reasoning
Rules for weakest preconditions

• \( \text{wp}(x = e, Q) \) is \( Q[x=e] \)
  – Example: \( \text{wp}(x = 2*y, x > 4) = 2*y > 4 \), i.e., \( y > 2 \)

• \( \text{wp}(S1;S2, Q) \) is \( \text{wp}(S1, \text{wp}(S2, Q)) \)
  – i.e., let \( R \) be \( \text{wp}(S2, Q) \) and overall \( \text{wp} \) is \( \text{wp}(S1, R) \)
  – Example: \( \text{wp}(y = x+1, \text{wp}(z = y+1, z > 2)) = \text{wp}(y = x+1, y+1 > 2) = (x+1)+1 > 2 \) or equivalently \( x > 0 \)

• \( \text{wp}(\text{if } b \ S1 \ \text{else } S2, Q) \) is this logic formula:
  \[
  (b \ \text{and} \ \text{wp}(S1,Q)) \ \text{or} \ (\neg b \ \text{and} \ \text{wp}(S2,Q))
  \]
  – you need \( \text{wp}(S1,Q) \) if \( S1 \) is executed and \( \text{wp}(S2,Q) \) if \( S2 \) is
  – you can often simplify the result considerably
More Examples

- If $S$ is $x = y \times y$ and $Q$ is $x > 4$, then $wp(S, Q)$ is $y \times y > 4$, i.e., $|y| > 2$

- If $S$ is $y = x + 1; \ z = y - 3$; and $Q$ is $z = 10$, then $wp(S, Q) ...$
  \[ = wp(y = x + 1; \ z = y - 3, z = 10) \]
  \[ = wp(y = x + 1, \ wp(z = y - 3, z = 10)) \]
  \[ = wp(y = x + 1, \ y - 3 = 10) \]
  \[ = wp(y = x + 1, \ y = 13) \]
  \[ = x + 1 = 13 \]
  \[ = x = 12 \]
Bigger Example

\[
S \text{ is if } (y < 5) \{ x = y*y; \} \text{ else } \{ x = y+1; \}
\]

\[
\text{wp}(S, x \geq 9) = (y < 5 \text{ and wp}(x = y*y, x \geq 9))
\]
\[
\quad \text{or } (y \geq 5 \text{ and wp}(x = y+1, x \geq 9))
\]
\[
= (y < 5 \text{ and } y*y \geq 9)
\]
\[
\quad \text{or } (y \geq 5 \text{ and } y+1 \geq 9)
\]
\[
= (y \leq -3) \text{ or } (y \geq 3 \text{ and } y < 5)
\]
\[
\quad \text{or } (y \geq 8)
\]
If-statements review

Forward reasoning

\[
\begin{align*}
\{ P \} \\
\text{if } B \\
\quad \{ P \text{ and } B \} \\
\quad S1 \\
\quad \{ Q1 \} \\
\text{else} \\
\quad \{ P \text{ and } \text{not } B \} \\
\quad S2 \\
\quad \{ Q2 \} \\
\{ Q1 \text{ or } Q2 \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ (B \text{ and } \text{wp}(S1, Q)) \text{ or } \\
\quad (\text{not } B \text{ and } \text{wp}(S2, Q)) \} \\
\text{if } B \\
\quad \{ \text{wp}(S1, Q) \} \\
\quad S1 \\
\quad \{ Q \} \\
\text{else} \\
\quad \{ \text{wp}(S2, Q) \} \\
\quad S2 \\
\quad \{ Q \} \\
\{ Q \}
\end{align*}
\]
One caveat

• With forward reasoning, there is a problem with assignment:
  – changing a variable can affect other assumptions

```plaintext
{{ }}
w = x + y;
{{ w = x + y }}
x = 4;
{{ w = x + y and x = 4 }}
y = 3;
{{ w = x + y and x = 4 and y = 3 }}
```

• But clearly we do not know $w = 7$!
• The assertion $w = x + y$ means the original values of $x$ and $y$
One Fix

- Use different names for the values at different points
  - common to use subscripts to distinguish these
  - on every assignment, rename references to the old values

```c
{{
  w = x + y;
  {{ w = x + y }}
  x = 4;
  {{ w = x_1 + y and x = 4 }}
  y = 3;
  {{ w = x_1 + y_1 and x = 4 and y = 3 }}
```
Useful example: swap

- Consider code for a swapping $x$ and $y$

```c
{{ }}
    tmp = x;
    {{ tmp = x }}
    x = y;
    {{ tmp = x \_1 and x = y }}
    y = tmp;
    {{ tmp = x \_1 and x = y \_1 and y = tmp }}
```

- Post condition implies $x = y \_1$ and $y = x \_1$
- I.e., their final values are equal to the original values swapped
Announcements

• Link to notes from last quarter are also on the web

• HW1 posted
  – practice applying these ideas
  – builds up to verifying correctness of short, non-loop code
  – due on Friday by 11pm