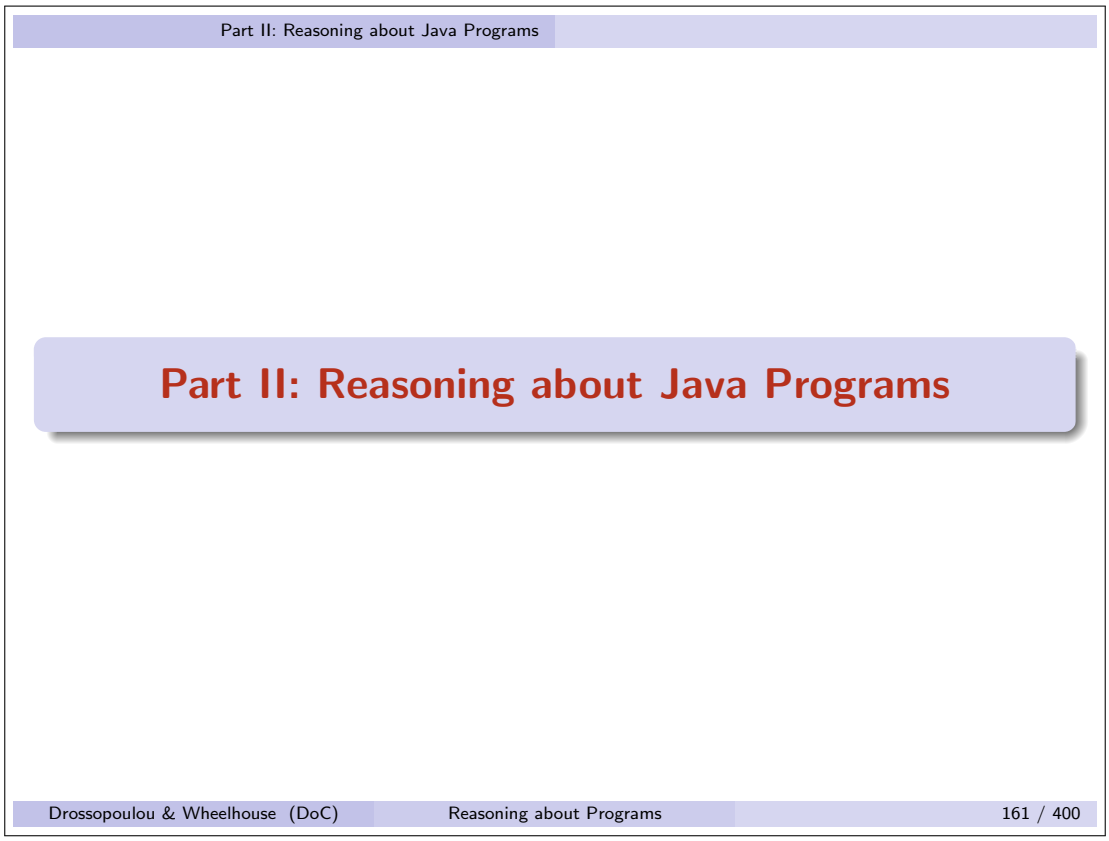
# 2 Part II: Reasoning about Java Programs

Slide 161



Slide 162



## Part II – Outline

2. Part II: Reasoning about Java Programs
   - Program Specifications
   - Conditional Branches
   - Method Calls
   - Recursion
   - Iteration
     - Loop Invariants and Loop Variants - Informally
     - Loop Invariant and Loop Variants - Examples
     - Loop Invariant and Loop Variants - Formally

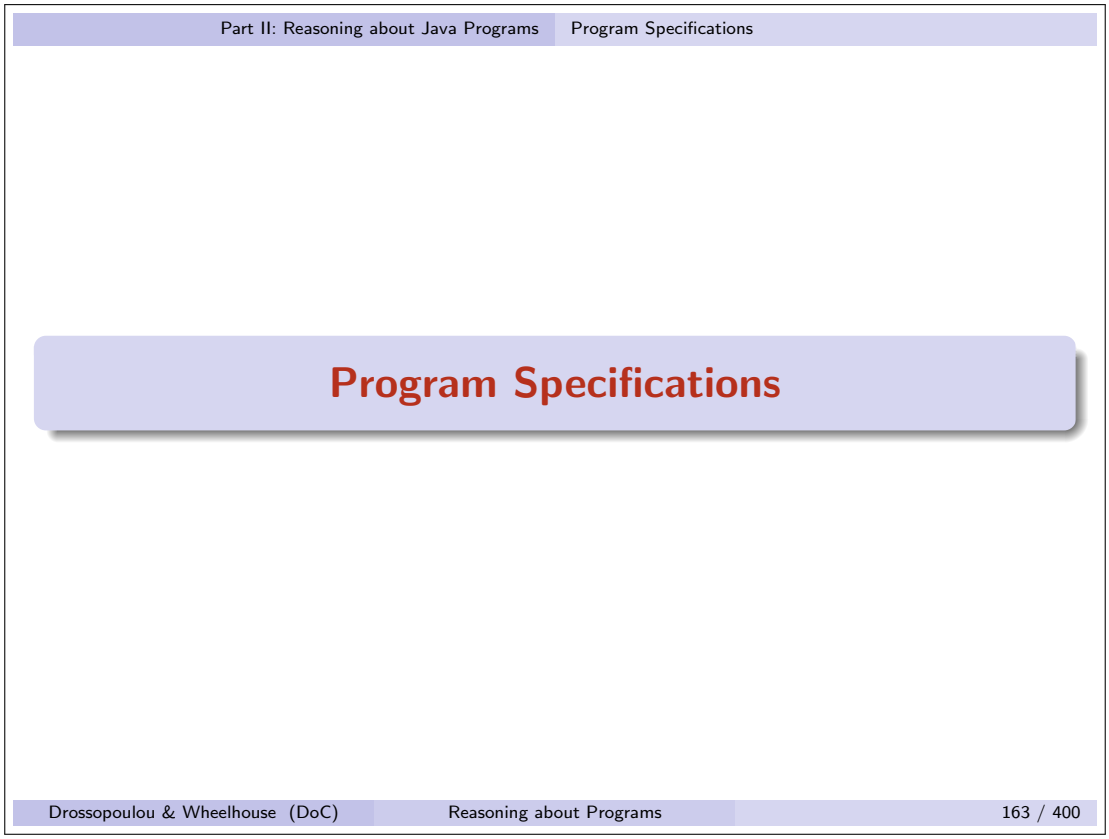
**Note**: We only consider a subset of Java, excluding OO features, aliasing and concurrency.



All of these more advanced features of Java are discussed in future courses available to you in later years of the degree.

## 2.1 Program Specifications

Slide 163




So far in this course we have focused on Haskell (functional) programs. Haskell functions are termed "pure" or "side-effect free", which makes them much simpler to reason about.

Slide 164



## Specifiying Java Programs

**Challenge:** The execution of some Java code can modify the program state in a way that depends on conditions *before* the execution.

We specify this in the following way:

```
// P
somecode
// Q
```

The above specification expresses that if the state satisfies $P$, then after the execution of `somecode`, the state will satisfy $Q$.

- We call $P$ the *precondition* of `somecode`
- We call $Q$ the *postcondition* of `somecode`



Slide 165



## Specifiying Java Programs

a) Does the following specification hold?

```
// a = 3 ∧ x = 7
int y = a + x;
// a = 3 ∧ x = 7 ∧ y = 10
```

b) Does the following specification hold?

```
// a = 5 ∧ x = 2
int y = a + x;
// x = 2 ∧ y = 7 ∧ a = 5
```

c) Does the following specification hold?

```
// a = 3 ∧ x = 7
int y = a + x;
// a = 6 ∧ x = 2 ∧ y = 8
```

```
// a = 3 ∧ x = 7
int y = a + x;
// a = 3 ∧ x = 7 ∧ y = 10
```

This specification does hold. The values stored in `a` and `x` are unmodifed by the code, so remain the same in the postcondition. Also $3 + 7 = 10$, so the value stored in `y` in the postcondition is also correct.

```
// a = 5 ∧ x = 2
int y = a + x;
// x = 2 ∧ y = 7 ∧ a = 5
```

This specificaiton also holds. The ordering of variables in the postcondition is not important (remember that $\wedge$ is both associative and commutative). Since $5 + 2 = 7$, the value stored in `y` in the postcondition is correct.

```
// a = 3 ∧ x = 7
int y = a + x;
// a = 6 ∧ x = 2 ∧ y = 8
```

This specification does **not** hold. Whilst $6 + 2 = 8$, so the postcondion might appear to be self-consistent, the code does not modify `a` or `x`, so the assertions `a` $= 6$ and `x` $= 2$ cannot hold in the postcondition from the given precondition.

If the precondition were instead to state `a` $= 6 \wedge$ `x` $= 2$, then the specification would then hold.

Slide 166



## Specification of Java Programs

We can generalise this behaviour over arbitrary integers $u$, and $v$.

```
// a = u ∧ x = v
int y = a + x;
// a = u ∧ x = v ∧ y = u + v
```

The specification above should be read as follows:

> For any integers $u$, and $v$, if we start with a program state where `a` has the value $u$ and `x` has the value $v$, and we execute the given code, then we will reach a program state where `a` still has the value $u$, `x` still has the value $v$ and `y` now has the value $u + v$.

**Important:** there is an implicit universal quantification of $u$ and $v$ over the whole specification.

It is important that this universal quantification is over the whole specification, otherwise we would change its meaning. For example, if the postcondition above were written as

$$\forall u, v.\, [\mathtt{a} = u \ \wedge \ \mathtt{x} = v \ \wedge \ \mathtt{y} = u + v]$$

then we would be saying that the variable `a` has all possible values after running this code, which is clearly nonsense!

Similarly, if the postcondition were written as

$$\exists u, v \ \ [\mathtt{a} = u \ \wedge \ \mathtt{x} = v \ \wedge \ \mathtt{y} = u + v]$$

then we would be saying that the variables `a` and `x` have *some* values, but we are not being terribly precise about *what* values. This specification is at least true, and does enforce that the value of `y` is the sum of the values of `a` and `x`, but we usually want to make stronger assertions about our program state.

Slide 167



## Straight Line Code

How do we reason about *sequences* of state-changing statements?
We reason with *Midconditions* – checkpoints in the code.
For example:

```
1  // PRE: P
2  somecode
3  // MID: P1
4  morecode
5  // MID: P2
6  evenmorecode
7  // POST: Q
```

- $P_1$ and $P_2$ are "stepping stones" in the reasoning.
- We can assume $P$ to show that $P_1$ holds after somecode.
- We can assume $P_1$ to show that $P_2$ holds after morecode.
- We can assume $P_2$ to show that $Q$ holds after evenmorecode.



If you were developing your code, you would hopefully write some comments to guide you. In a similar way, we use midconditions to guide us through the proof of the code.

Slide 168



## Proof Obligations - Hoare Triples

The reasoning steps in the previous slide are of the form:

> *"Assuming that some property $P$ holds, show that after execution of* `code`*, some other property $Q$ will hold".*

We can formally express this as a *Hoare Triple*:

$$\{\, P \,\} \quad \texttt{code} \quad \{\, Q \,\}$$

For example:

$$\{\ \mathit{true}\ \} \quad \texttt{x = 5;} \quad \{\ \texttt{x} > 0\ \}$$

$$\{\ 0 \leq \texttt{x} < 10\ \} \quad \texttt{x++;} \quad \{\ 0 \leq \texttt{x} \leq 10\ \}$$



Slide 169



## Proof Obligations - Hoare Triples

Proving the correctness of a Hoare triple:

$$\{\, P \,\} \quad \texttt{code} \quad \{\, Q \,\}$$

is essentially a proof that $P$ modified by the effects of `code` implies $Q$.

For example, to prove:

$$\{\ \mathit{true}\ \} \quad \texttt{x = 5;} \quad \{\ \texttt{x} > 0\ \}$$

we would need to show that:

$$\mathit{true} \ \wedge\ \texttt{x} = 5 \quad \longrightarrow \quad \texttt{x} > 0$$

Slide 170



## Notational Conventions - Variables in Proofs

Some proof obligations present a problem when it comes to the meaning of program variables. For example, in

$$\{\ 0 \leq \texttt{x} < 10\ \}\quad \texttt{x++;}\quad \{\ 0 \leq \texttt{x} \leq 10\ \}$$

the variable `x` appears in the precondition, the postcondition *and* it is modified by the code.

Naively, we might try to show that:

$$0 \leq \texttt{x} < 10 \ \wedge\ \texttt{x} = \texttt{x} + 1 \quad\longrightarrow\quad 0 \leq \texttt{x} \leq 10$$



The logical assertion

$$\texttt{x} = \texttt{x} + 1$$

is always false, regardless of the value of `x`. Thus, proving the above would be very easy, as `false` $\longrightarrow$ `anything` is always true.

However, if I could prove this specification, then I would also be able to prove

$$\{\ \texttt{true}\ \}\quad \texttt{x++;}\quad \{\ \texttt{gains}(\texttt{Mark}, \$1\texttt{million})\ \}$$

which is (unfortunately) not true...

The problem is that we have not correctly captured the temporal (or change of time) behaviour of the code. Our assertion needs to describe the values of `x` both before and after the assignment.

Slide 171



## Notational Conventions - Variables in Proofs

To make things clear, we use the following notational convention:

- $\mathtt{x}$ refers to the value of $\mathtt{x}$ *before* the code is executed.
- $\mathtt{x}'$ refers to the value of $\mathtt{x}$ *after* the code has been executed
- $\mathtt{x}_0$ refers to the *original* value of $\mathtt{x}$ as passed into the method.

So now to prove:

$$\{\ 0 \leq \mathtt{x} < 10\ \}\quad \mathtt{x{+}{+};}\quad \{\ 0 \leq \mathtt{x} \leq 10\ \}$$

we would need to show that:

$$0 \leq \mathtt{x} < 10\ \wedge\ \mathtt{x}' = \mathtt{x} + 1 \quad\longrightarrow\quad 0 \leq \mathtt{x}' \leq 10$$



Slide 172



## Hoare Logic

In 1969 Sir Tony Hoare developed a new logic which allows for formal reasoning *directly* with Hoare Triples (hence the name).
For example there is an axiom for dealing with assignment:

$$\frac{P \wedge \mathtt{x}' = u \quad\longrightarrow\quad Q[\mathtt{x} \mapsto \mathtt{x}']}{\{\ P\ \}\quad \mathtt{x = } u\mathtt{;}\quad \{\ Q\ \}}$$

This axiom states that after the assignment we can establish any property that is derivable from the precondition and the effect of the assignment.
For example:

$$\{\ \mathtt{x} = 5\ \}\quad \mathtt{x = x + 2;}\quad \{\ \mathtt{x} = 7\ \}$$

can be proven by showing that:

$$\mathtt{x} = 5\ \wedge\ \mathtt{x}' = \mathtt{x} + 2 \quad\longrightarrow\quad \mathtt{x}' = 7$$



Hopefully you will recall from the Logic course last term, that when we write rules of

inference of the form:

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

then this states that in order to prove the conclusion $Q$ holds we must first show that the premise $P$ holds.

For the rest of this course we will stick to a simplified presentation of such formal arguments. However, we will be relating our approach to the formal Hoare Logic rules that guide our style. In later years of the degree you will have the chance to study such formalisms in more detail.

The interested can find a primer on Hoare Logic on Wikipedia at:
`https://en.wikipedia.org/wiki/Hoare_logic`

Hoare's original paper that launched the program verification field "An Axiomatic Basis for Computer Programming" can be found at:
`https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf`

Hoare has also written a retrospective article on his career for the ACM's online magazine:
`http://cacm.acm.org/magazines/2009/10/42360-retrospective-an-axiomatic-basis-for-computer-programming/fulltext`

Slide 173



## Method Specifications

Method specifications are given in terms of pre- and post- conditions:

```
1   type someMethod(type x_1, ... , type x_n)
2   // PRE: P(x_1, ..., x_n)
3   // POST: Q(x_1, ..., x_n)
4   {
5       ...
6   }
```

The above specification promises that if $P(v_1, ..., v_n)$ holds before a call to `someMethod`$(v_1, ..., v_n)$, then $Q(v_1, ..., v_n)$ will hold upon return.



Note that $v_1, ..., v_n$ are values, while $x_1, ..., x_n$ are program variables.

Slide 174



## Method Bodies

How do we prove that a method satisfies its specification?

```
1   type someMethod(type x_1, ..., type x_n)
2   // PRE: P(x_1, ..., x_n)
3   // POST: Q(x_1, ..., x_n)
4   {
5       code
6   }
```

Using $P(x_1, ..., x_n)$ as an assumption we have to show that $Q(x_1, ..., x_n)$ holds after executing `code`.

i.e.

$$\{\ P(x_1, ..., x_n)\ \}\quad \texttt{code} \quad \{\ Q(x_1, ..., x_n)\ \}$$



Slide 175



## Method Bodies

If the body of the method consists of multiple lines of code?

```
1   type someMethod(type x_1, ..., type x_n)
2   // PRE: P(x_1, ..., x_n)
3   // POST: Q(x_1, ..., x_n)
4   {
5       code1
6       // MID: R(x_1, ..., x_n)
7       code2
8       // MID: S(x_1, ..., x_n)
9       code3
10  }
```

Then, as before, we introduce appropriate mid-conditions, i.e.

1. Assume $P(x_1, ..., x_n)$, show that $R(x_1, ..., x_n)$ holds after `code1`.
2. Assume $R(x_1, ..., x_n)$, show that $S(x_1, ..., x_n)$ holds after `code2`.
3. Assume $S(x_1, ..., x_n)$, show that $Q(x_1, ..., x_n)$ holds after `code3`.

Slide 176



## Pre-/Post-/Mid-conditions

- **Pre-condition**:
  - required to hold before some code is run
  - an assumption that the code can make
- **Post-condition**:
  - expected to hold after the code has been executed (assuming termination and precondition held before)
  - a guarantee that the code must make
- **Mid-condition**:
  - an assumption made at a specific point in the code
  - must be guaranteed by preceding code
  - can be assumed by subsequent code
  - a "stepping stone" in reasoning about correctness

**Important:** mid-conditions are actually post-conditions for the preceding lines of code and pre-conditions for the subsequent lines of code.



Note that if a midcondition evaluates to $false$, then this specifies that the program **cannot** reach that location in the code. This may sometimes be correct, but more often than not it implies that your midcondition is incorrect.

Slide 177



## Notational Conventions - Variables in Specifications

Just as in our proofs, it is important to be precise about the meaning of variables in our specifications.

- We use a `monospace` font when referring to program variables,
  e.g. `x`, `i` or `count`
- We use an *italic* font when refering to value variables,
  e.g. $u$ or $v$
- In pre-/post-/mid-conditions we use the subscript 0 to refer to the initial value of an input variable on entry to the method,
  e.g. $\mathtt{x}_0$ or $\mathtt{a}_0$
- In post-conditions we use a **bold** r to refer to the return value of the method (if there is one),
  e.g. $\mathbf{r}$

**Important:** all specifications refer to the **current** program state, we never use $\mathtt{x}'$ in our assertions (this would also be a bad choice of variable name).



Obviously, making such type-setting distinctions by hand is rather tricky. We encourage you to use different variables (as we have also done) to avoid any potential confusion.

Slide 178



## Notational Conventions - Shorthands

We also introduce the following notational shorthands:

- $\mathtt{a} \sim \mathtt{b}$ means that array `a` is a permutation of array `b`,
  i.e. $\forall v.[\ |\{\ k \mid \mathtt{a}[k] = v\ \}| = |\{\ k \mid \mathtt{b}[k] = v\ \}|\ ]$
- $\mathtt{a} \approx \mathtt{b}$ means that the arrays `a` and `b` are identical,
  i.e. $\mathtt{a.length} = \mathtt{b.length}$
  $\wedge\ \forall j.[\ 0 \leq j < \mathtt{a.length} \longrightarrow \mathtt{a[j]} = \mathtt{b[j]}\ ]$
- $\sum \mathtt{a}[x..y)$ is the sum of the elements of array `a` from index $x$ up to but not including index $y$,
  i.e. $\sum \mathtt{a}[x..y) = \sum_{k=x}^{y-1} \mathtt{a}[k]$
  $= \mathtt{a}[x] + \mathtt{a}[x+1] + ... + \mathtt{a}[y-1]$

where `a` and `b` stand for arbitrary arrays.



We can similarly define $\Pi\mathtt{a}[x..y)$, the product of the elements of array `a` from index $x$ up to but not including index $y$,
i.e.

$$\Pi\mathtt{a}[x..y) = \Pi_{k=x}^{y-1}\mathtt{a}[k] = \mathtt{a}[x] * \mathtt{a}[x+1] * ... * \mathtt{a}[y-1]$$

Slide 179



## Notational Conventions - Shorthands

We also introduce the following notational shorthands:

- *Sorted*(a) means that array a is ordered,
  i.e. $\forall j, k.[\ 0 \leq j \leq k < \texttt{a.length} \longrightarrow \texttt{a[j]} \leq \texttt{a[k]}\ ]$
- *min*(a) is the smallest element in the array a,
  i.e. $min(\texttt{a}) = min\{\ z \mid \exists j.\ 0 \leq j < \texttt{a.length} \wedge \texttt{a[j]} = z\}$
- *max*(a) is the biggest element in the array a,
  i.e. $max(\texttt{a}) = max\{\ z \mid \exists j.\ 0 \leq j < \texttt{a.length} \wedge \texttt{a[j]} = z\}$

where a stands for an arbitrary array.

**Important:** *Sorted*, *min* and *max* are only well-defined for an array whose contents are comparable (such as `Integers`).



Slide 180



## Proving Method Specifications - Example

```
1   int biggest(int x, int y, int z) {
2   // PRE: true
3   // POST: r = max{x,y,z}
4       if (x >= y){
5           res = x;
6       } else {
7           res = y;
8       }
9       // MID:  res = max{x,y}
10      if (z >= res){
11          res = z;
12      }
13      // MID:  res = max{z, max{x,y}}
14      return res;
15  }
```



When choosing the midcondition on line 13, you need to be careful with what you write.

One common mistake we have seen in the past is to write something like:

$$\texttt{res} = max\{\texttt{res}, \texttt{z}\}$$

Recall that this is a *logical assertion* and not code. The above does not describe an assignment to the variable `res`, but rather makes a claim about its value being equal to that of the larger of `res` or `z`. This can only be true if $\texttt{res} \geq \texttt{z}$ (i.e. we guarantee that $max\{\texttt{res}, \texttt{z}\}$ returns `res`) which is obviously not general enough for our proof as the method's precondition places no constraints on the input values of `x`, `y` or `z`.

Another common mistake is to write something like:

$$\texttt{res}' = max\{\texttt{res}, \texttt{z}\}$$

This describes the current value of some program variable `res'`, but there is no such variable in our code. Remember that we only use the $'$ annotations to distinguish between the values stored in a variable before/after some code has run. Such annotations have no place in our assertions.

Slide 181

Part II: Reasoning about Java Programs | Program Specifications

## Specification of Java Programs - Conclusions

- A piece of code may have more than one pre-condition/post-condition.
- The post-condition depends on the code as well as the pre-condition.
- We use mid-conditions as "stepping-stones" in our reasoning.

Food for thought:

- In general, given some code and a post-condition, there exists a *Weakest* pre-condition.
- In general, given some code and a pre-condition, there exists a *Strongest* post-condition.

Drossopoulou & Wheelhouse (DoC) | Reasoning about Programs | 181 / 400

### [Extra] Weakest Preconditions and Strongest Postconditions

Whilst we do not cover this area as part of the Reasoning About Programs course curriculum, the general idea is actually quite simple.

The *Weakest Precondition* for a program is the most general (or weakest) property which a program requires to function correctly (i.e. to satisfy its postcondition without faulting)

and which can be inferred from all correct preconditions.

For example, the weakest precondition of the tiny program `n++` with a desired postcondition of $\texttt{n} > 1$ would be $n > 0$. Then any concrete case, such as $\texttt{n} = 5$, would imply the weakest precondition.

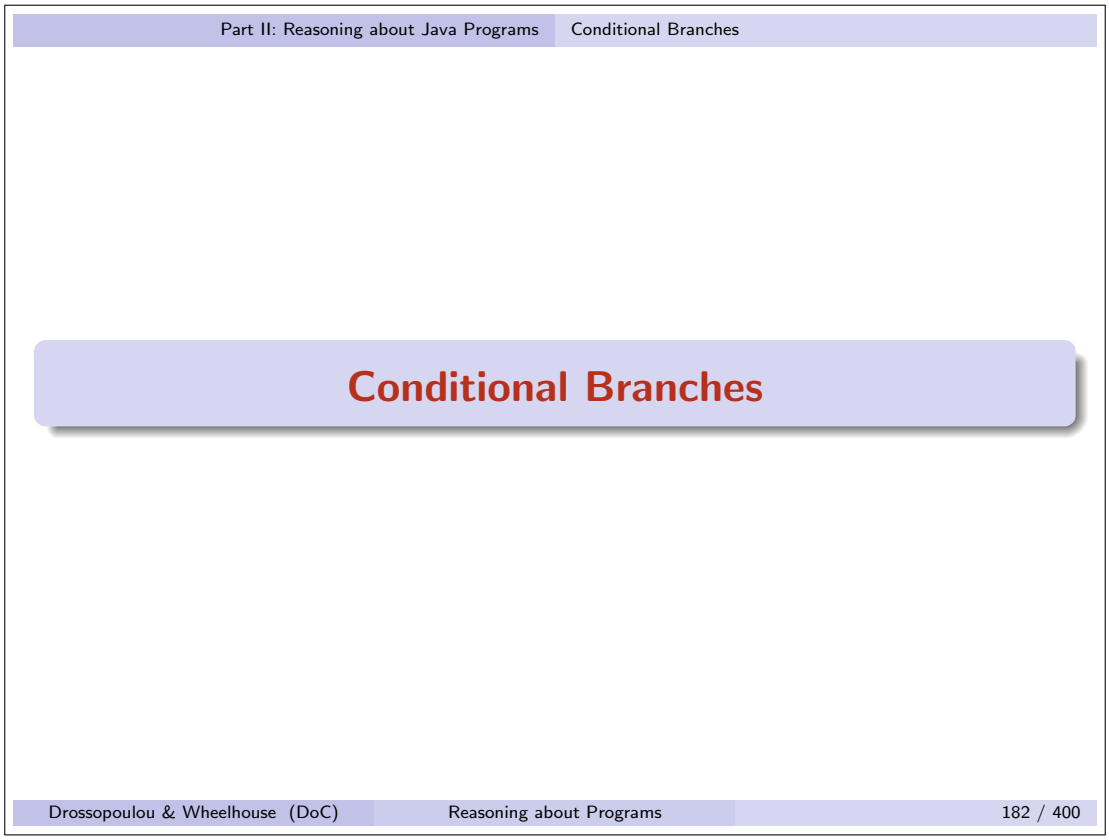
Conversely, the *Strongest Postcondition* for a program is the most specific (or strongest) property which holds after the program has run on a state satisfying its precondition, and from which any correct postcondition can be derived.

Again, taking the simple program `n++` as an example, now with a precondition of $\texttt{n} = 5$, the strongest postcondition would be $\texttt{n} = 6$. Any other more general case, such as $\texttt{n} > 1$, could then be derived from the strongest postcondition.

The weakest preconditions and strongest postconditions are primarily used in automated verification techniques. Typically you use only one, depending on if you are working through the program backwards or forwards.

## 2.2 Conditional Branches

Slide 182



Slide 183



## Reasoning about Conditional Branches

How do we prove that branching code satisfies its specification?

```
1  // PRE: P
2  if (cond){
3      code1;
4  } else {
5      code2;
6  }
7  // POST: Q
```

Using $P$ and the case of `cond` as assumptions we have to show that $Q$ holds after executing each branch `code1` and `code2`.

$$\frac{\{\ P \wedge \texttt{cond}\ \}\ \ \texttt{code1}\ \ \{\ Q\ \} \qquad \{\ P \wedge \neg\texttt{cond}\ \}\ \ \texttt{code2}\ \ \{\ Q\ \}}{\{\ P\ \}\ \ \texttt{if(cond)\{ code1 \} else \{ code2 \}}\ \ \{\ Q\ \}}$$



Notice above that there is a slightly unfortunate clash between the syntax of our programming language and that of our proof system, both making use of curly-brackets { }. In the code, these delimit the scope of our conditional and looping statements, whereas in the proof system there separate the assertions from the code. We have to take a little care not to get confused by this symbolic overloading.

Slide 184



## Reasoning about Conditional Branches

We can introduce appropriate mid-conditions to guide our proof:

```
1   // PRE: P
2   if (cond){
3       // MID: P ∧ cond
4       code1;
5       // MID: Q
6   } else {
7       // MID: P ∧ ¬cond
8       code2;
9       // MID: Q
10  }
11  // POST: Q
```

- `code1` and `code2` may "assume" $P$ and their respective case for free.
- It is the "responsibility" of both `code1` and `code2` to establish $Q$.



When reasoning about a conditional branch we get to assume that the pre-condition holds at the start of each branch. We also get to assume that the condition holds in the `then` branch and does not hold in the `else` branch. We then have to show that the post-condition holds on both branches of the code.

# Conditional Branches - Example

Looking at first part of our `biggest` method in more detail:

```
1   // PRE: true
2   if (x >= y){
3       // MID:  x ≥ y
4       res = x;
5       // MID:  res = x ∧ x ≥ y
6   } else {
7       // MID:  y > x
8       res = y;
9       // MID:  res = y ∧ y > x
10  }
11  // MID:  res = max{x, y}
```

## 2.3 Method Calls

Slide 186



# Method Calls



Slide 187



# Reasoning about Method Calls

How do we show that a method call satisfies its caller's specification?

```
1   // MID: P
2   z = someMethod(v1, ..., vn);
3   // MID: Q
```

We have to show that $P$ satisfies the method's precondition and that in turn that the method's postcondition satisfies $Q$.

$$\frac{\begin{array}{c} P \longrightarrow R[\mathtt{x1} \mapsto v_1, ..., \mathtt{xn} \mapsto v_n] \\ S[\mathtt{x1}_0 \mapsto v_1, ..., \mathtt{xn}_0 \mapsto v_n, \mathtt{x1} \mapsto v'_1, ..., \mathtt{xn} \mapsto v'_n, \mathbf{r} \mapsto \mathtt{z}'] \wedge P \\ \longrightarrow Q[v_1 \mapsto v'_1, ..., v_n \mapsto v'_n] \end{array}}{\{\ P\ \}\quad \mathtt{z} = \mathtt{someMethod}(v_1, ..., v_n);\quad \{\ Q\ \}}$$

where

```
type someMethod(type x1, ..., type xn)
// PRE: R
// POST: S
```

In the above we write $P[x \mapsto y]$ to represent the predicate $P$ with all free occurrences of $x$ replaced with $y$.

The substitutions of $[\texttt{x1} \mapsto v_1, ..., \texttt{xn} \mapsto v_n]$ in $R$ and $[\texttt{x1}_0 \mapsto v_1, ..., \texttt{xn}_0 \mapsto v_n]$ in $S$ above ensure that we replace the abstract variables in the method's specification with the concrete values used at its call point in our code.

The additional substitutions of $[\texttt{x1} \mapsto v'_1, ..., \texttt{xn} \mapsto v'_n, \mathbf{r} \mapsto \texttt{z}']$ in $S$ and $[v_1 \mapsto v'_1, ..., v_n \mapsto v'_n]$ in $Q$ account for the update in program state caused by the method call, allowing us to relate the initial values of variables to their final values.

Slide 188



## Reasoning about Method Calls

Let $R_{sub} = R[\texttt{x1} \mapsto v_1, ..., \texttt{xn} \mapsto v_n]$
Let $S_{sub} = S[\texttt{x1}_0 \mapsto v_1, ..., \texttt{xn}_0 \mapsto v_n, \texttt{x1} \mapsto v'_1, ..., \texttt{xn} \mapsto v'_n, \mathbf{r} \mapsto \texttt{z}']$

To reason about a method call, we have to show that the precondition $R_{sub}$ is established before the call. We can then assume that the postcondition $S_{sub}$ will hold after the call.

```
1  code1;
2  // MID: R_sub
3  z = someMethod(v_1, ..., v_n);
4  // MID: S_sub
5  code2;
```

- It is the "responsibility" of `code1` to establish $R_{sub}$.
- In return, `code2` may "assume" $S_{sub}$ for free.

Slide 189



## Method Call - Example

```
1   void sort(int[] b)
2   // PRE: b ≠ null                                          (P1)
3   // POST: b ∼ b0 ∧ Sorted(b)                               (Q1)
4   {
5       ...
6   }
7
8   int smallest(int[] a)
9   // PRE: a ≠ null ∧ a.length > 0                           (P2)
10  // POST: r = min(a0)                                      (Q2)
11  {
12      // MID: a ≈ a0 ∧ a ≠ null ∧ a.length > 0              (M1)
13      sort(a);
14      // MID: a ∼ a0 ∧ Sorted(a) ∧ a.length > 0             (M2)
15      int res = a[0];
16      // MID: a ∼ a0 ∧ res = min(a)                         (M3)
17      return res;
18  }
```



Slide 190



## Reasoning about `smallest` - Proof Obs. (Informal)

The correctness of the `smallest` method relies on the following argument:

**line 12:** The precondition of `smallest` must establish the midcondition $M_1$.
i.e. $P_2 \longrightarrow M_1$

**line 13:** The midcondition $M_1$ must establish the precondition of `sort`.
i.e. $M_1 \longrightarrow P_1$

**line 14:** The postcondition of `sort` must establish the midcondition $M_2$.
i.e. $Q_1 \wedge M_1 \longrightarrow M_2$

**line 16:** The midcondition $M_2$ and code must establish midcondition $M_3$.
i.e. $M_2 \wedge$ `res = a[0]` $\longrightarrow M_3$

**line 18:** The midcondition $M_3$ must establish the postcondition of `smallest`.
i.e. $M_3 \wedge$ `return res` $\longrightarrow Q_2$

**Important:** care must be taken to track the effects of the code.

Slide 191



## Reasoning about `smallest` - Care with Variables

A naive translation of the arguments leads to incorrect proof obligations.

We must carefully track which variable values we are referring to. Are they **initial** values, **current** values, or **final** values with respect to the code?

We achieve this by substituting the variables in method specifications (pre- and post-conditions) by the arguments that were *passed* to them.

**e.g.** on line 14, when we use the postcondition of `sort` to establish $M_2$, we are given:

$$Q_1[\, \mathtt{b}_0 \mapsto \mathtt{a},\ \mathtt{b} \mapsto \mathtt{a}' \,] \quad \equiv \quad \mathtt{a}' \sim \mathtt{a} \ \wedge \ \mathit{Sorted}(\mathtt{a}')$$



Slide 192



## Reasoning about `smallest` - Proof Obs. (Explicit)

The correctness of the `smallest` method relies on the following argument:

**line 12:** The precondition of `smallest` must establish the midcondition $M_1$.
$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0) \ \longrightarrow \ M_1$

**line 13:** The midcondition $M_1$ must establish the precondition of `sort`.
$M_1 \ \longrightarrow \ P_1[\mathtt{b} \mapsto \mathtt{a}]$

**line 14:** The postcondition of `sort` must establish the midcondition $M_2$.
$(Q_1[\mathtt{b}_0 \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}'] \wedge M_1) \ \longrightarrow \ M_2[\mathtt{a} \mapsto \mathtt{a}']$

**line 16:** The midcondition $M_2$ and code must establish midcondition $M_3$.
$(M_2 \wedge \mathtt{res}' = \mathtt{a}[0] \wedge \mathtt{a}' \approx \mathtt{a}) \ \longrightarrow \ M_3[\mathtt{a} \mapsto \mathtt{a}', \mathtt{res} \mapsto \mathtt{res}']$

**line 18:** The midcondition $M_3$ must establish the postcondition of `smallest`.
$(M_3 \wedge \mathbf{r} = \mathtt{res}) \ \longrightarrow \ Q_2$

**Important:** we only substitute non identity cases.



We discuss the development of each of these proof obligations over the following slides.

Slide 193



## Reasoning about `smallest` - Proof Obs. (Final)

**line 12:** The precondition of `smallest` must establish the midcondition $M_1$.

$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0) \longrightarrow M_1$

$$\mathtt{a}_0 \neq \mathtt{null} \ \wedge \ \mathtt{a}_0\mathtt{.length} > 0 \ \wedge \ \mathtt{a} \approx \mathtt{a}_0$$
$$\longrightarrow$$
$$\mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathtt{a} \neq \mathtt{null} \ \wedge \ \mathtt{a.length} > 0$$



In this first obligation we have to remember that the precondition $P_2$ only describes the variables that were passed into the method call. In order to use the precondition in our proof, we must substitute all of the variables for their initial ($\mathtt{x}_0$) versions.

Moreover, as no code has been executed so far in the method, we can gain the implicit information that $\mathtt{a} \approx \mathtt{a}_0$.

Slide 194



## Reasoning about `smallest` - Proof Obs. (Final)

**line 13:** The midcondition $M_1$ must establish the precondition of `sort`.

$M_1 \longrightarrow P_1[\mathtt{b} \mapsto \mathtt{a}]$

$$\mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathtt{a} \neq \mathtt{null} \ \wedge \ \mathtt{a.length} > 0$$
$$\longrightarrow$$
$$\mathtt{a} \neq \mathtt{null}$$



In this obligation no code is actually being executed, so we do not need to worry about the *before* (`x`) or *after* (`x'`) states of the variables.

However, we do need to substitute the call value of `a` into the precondition $P_1$ of the `sort` method.

Slide 195



## Reasoning about `smallest` - Proof Obs. (Final)

**line 14:** The postcondition of `sort` must establish the midcondition $M_2$.

$$(Q_1[\mathtt{b}_0 \mapsto \mathtt{a}, \mathtt{b} \mapsto \mathtt{a}'] \wedge M_1) \longrightarrow M_2[\mathtt{a} \mapsto \mathtt{a}']$$

$$\mathtt{a}' \sim \mathtt{a} \wedge \mathit{Sorted}(\mathtt{a}') \wedge \mathtt{a} \approx \mathtt{a}_0 \wedge \mathtt{a} \neq \mathtt{null} \wedge \mathtt{a.length} > 0$$
$$\longrightarrow$$
$$\mathtt{a}' \sim \mathtt{a}_0 \wedge \mathit{Sorted}(\mathtt{a}') \wedge \mathtt{a}'.\mathtt{length} > 0$$



In this obligation we are considering the effect of the call to the `sort` method. This means that we must track the state of the input array `a` both *before* and *after* the method call.

Thus, we replace the initial value $\mathtt{b}_0$ with the call value `a` and the final value `b` with $\mathtt{a}'$ in the postcondition of the `sort` method. We must also similarly update the variables in the midcondition $M_2$ to their *after* values.

Note that as well as knowing the postcondition of the `sort` method, we also still know what the state of the program was before the method call (i.e. $M_1$).

Slide 196



## Reasoning about `smallest` - Proof Obs. (Final)

**line 16:** The midcondition $M_2$ and code must establish midcondition $M_3$.
$(M_2 \wedge \texttt{res}' = \texttt{a}[0] \wedge \texttt{a}' \approx \texttt{a}) \longrightarrow M_3[\texttt{a} \mapsto \texttt{a}', \texttt{res} \mapsto \texttt{res}']$

$$\texttt{a} \sim \texttt{a}_0 \ \wedge \ \mathit{Sorted}(\texttt{a}) \ \wedge \ \texttt{a.length} > 0 \ \wedge \ \texttt{res}' = \texttt{a}[0] \ \wedge \ \texttt{a}' \approx \texttt{a}$$
$$\longrightarrow$$
$$\texttt{a}' \sim \texttt{a}_0 \ \wedge \ \texttt{res}' = \mathit{min}(\texttt{a}')$$



This obligation concerns the execution of the code `res = a[0]`. This means that we need to track the *before* and *after* values of all of the variables in the code.

The midcondition $M_2$ requires no modification, as it describes the starting state for this execution, but the behaviour of the code must describe the update to `res` and the midcondition $M_3$ needs to be modified to track the variables *after* the execution.

Slide 197



## Reasoning about `smallest` - Proof Obs. (Final)

**line 18:** The midcondition $M_3$ must establish the postcondition of `smallest`.

$(M_3 \wedge \mathbf{r} = \texttt{res}) \longrightarrow Q_2$

$$\texttt{a} \sim \texttt{a}_0 \ \wedge\ \texttt{res} = \mathit{min}(\texttt{a}) \ \wedge\ \mathbf{r} = \texttt{res} \longrightarrow \mathbf{r} = \mathit{min}(\texttt{a}_0)$$



This last obligation simply tracks the assignment of the return value for the method. As this assignment can have no other side effects, we choose not to explicitly track the *before* and *after* values of the variables, resulting in a fairly straightforward proof obligation.

If we decided to be more explicit in tracking these values, then we would have the following obligation:

$$M_3 \wedge \mathbf{r} = \texttt{res} \wedge \texttt{res}' = \texttt{res} \wedge \texttt{a}' \approx \texttt{a} \longrightarrow Q_2[\texttt{a} \mapsto \texttt{a}', \texttt{res} \mapsto \texttt{res}']$$

which we would fully unfold as follows:

$$\texttt{a} \sim \texttt{a}_0 \ \wedge\ \texttt{res} = \mathit{min}(\texttt{a}) \ \wedge\ \mathbf{r} = \texttt{res} \ \wedge\ \texttt{res}' = \texttt{res} \ \wedge\ \texttt{a}' \approx \texttt{a} \longrightarrow \mathbf{r} = \mathit{min}(\texttt{a}_0)$$

but does not take any further effort to prove (none of the *after* values appear on the right-hand side of the implication).

## 2.4 Recursion

Slide 198



# Recursion



Slide 199



We already know how to handle method calls,
recursion requires *no* special treatment!

(We just have to be careful with our book-keeping.)

We can deal with recursion using the techniques we have already seen for reasoning about method calls.

The "trick" is that when proving that a method's body satisfies its specification, we can assume that the method specifications hold for any methods that are called within the body, including any recursive calls.

This allows us to tackle our proofs in a modular way.

As a running example we shall use a recursive method that returns the sum of all elements in an array.

Slide 200



## Recursion - Example

```
1  int sum(int[] a)
2  // PRE: a ≠ null                                          (P1)
3  // POST: a ≈ a0 ∧ r = ∑ a[0..a.length)                   (Q1)
4  {
5      int res = sumAux(a,0);
6      // MID: a ≈ a0 ∧ res = ∑ a[0..a.length)               (M1)
7      return res;
8  }
```

where `sumAux` satisfies the following specification:

```
int sumAux(int[] a, int i)
// PRE: a ≠ null ∧ 0 ≤ i ≤ a.length                         (P2)
// POST: a ≈ a0 ∧ r = ∑ a[i..a.length)                      (Q2)
```



Note that we can derive the midcondition $M_1$ without any knowledge of the specification of `sumAux` by working backwards from the postcondition $Q_1$ of `sum`. The only way that the `return` statement can satisfy the postcondition is if `res` has been set up to be equal to $\sum$ a[0..a.length).

Slide 201



## Recursion - Example

```
1   int sumAux(int[] a, int i)
2   // PRE: a ≠ null ∧ 0 ≤ i ≤ a.length                    (P2)
3   // POST: a ≈ a0 ∧ r = ∑ a[i..a.length)                 (Q2)
4   {
5       if (i == a.length) {
6           // MID: a ≈ a0 ∧ i = a.length                  (M2)
7           return 0;
8       } else {
9           // MID: a ≈ a0 ∧ a ≠ null ∧ 0 ≤ i < a.length   (M3)
10          int val = a[i] + sumAux(a, i+1);
11          // MID: a ≈ a0 ∧ val = ∑ a[i..a.length)        (M4)
12          return val;
13      }
14  }
```



Slide 202



## Reasoning about `sum`

The correctness of `sum` and `sumAux` can be shown independently.

This is known as modular verification.

Slide 203



## Reasoning about sum - Proof Obligations (Informal)

The correctness of sum relies on the following arguments:

**line 5:** The precondition of sum must establish the precondition of sumAux.
i.e. $P_1 \longrightarrow P_2$

**line 6:** The postcondition of sumAux must establish the midcondition $M_1$.
i.e. $Q_2 \wedge P_1 \wedge \texttt{res = sumAux(a,0)} \longrightarrow M_1$

**line 7:** The midcondition $M_1$ must establish the postcondition of sum.
i.e. $M_1 \wedge \texttt{return res} \longrightarrow Q_1$

**Important:** in the above we use sumAux as a "black box".



Slide 204



## Reasoning about sum - Proof Obligations (Explicit)

The correctness of sum relies on the following arguments:

**line 5:** The precondition of sum must establish the precondition of sumAux.
$(P_1[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0) \longrightarrow P_2[\mathtt{i} \mapsto 0]$

**line 6:** The postcondition of sumAux must establish the midcondition $M_1$.
$(Q_2[\mathtt{a}_0 \mapsto \mathtt{a}, \mathtt{a} \mapsto \mathtt{a}', \mathtt{i} \mapsto 0] \wedge P_1[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0 \wedge \mathtt{res}' = \mathbf{r})$
$\longrightarrow M_1[\mathtt{a} \mapsto \mathtt{a}', \mathtt{res} \mapsto \mathtt{res}']$

**line 7:** The midcondition $M_1$ must establish the postcondition of sum.
$(M_1 \wedge \mathbf{r} = \mathtt{res}) \longrightarrow Q_1$

Slide 205



## Reasoning about sum - Proof Obligations (Final)

**line 5:** The precondition of sum must establish the precondition of sumAux.
$(P_1[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0) \longrightarrow P_2[\mathtt{i} \mapsto 0]$

$$\mathtt{a}_0 \neq \mathtt{null} \ \wedge \ \mathtt{a} \approx \mathtt{a}_0$$
$$\longrightarrow$$
$$\mathtt{a} \neq \mathtt{null} \ \wedge \ 0 \leq 0 \leq \mathtt{a.length}$$



Slide 206



## Reasoning about sum - Proof Obligations (Final)

**line 6:** The postcondition of sumAux must establish the midcondition $M_1$.
$(Q_2[\mathtt{a}_0 \mapsto \mathtt{a}, \mathtt{a} \mapsto \mathtt{a}', \mathtt{i} \mapsto 0] \wedge P_1[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a} \approx \mathtt{a}_0 \wedge \mathtt{res}' = \mathbf{r})$
$\longrightarrow M_1[\mathtt{a} \mapsto \mathtt{a}', \mathtt{res} \mapsto \mathtt{res}']$

$$\mathtt{a}' \approx \mathtt{a} \ \wedge \ \mathbf{r} = \textstyle\sum \mathtt{a}'[0..\mathtt{a}'.\mathtt{length}) \ \wedge \ \mathtt{a}_0 \neq \mathtt{null} \ \wedge \ \mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathtt{res}' = \mathbf{r}$$
$$\longrightarrow$$
$$\mathtt{a}' \approx \mathtt{a}_0 \ \wedge \ \mathtt{res}' = \textstyle\sum \mathtt{a}'[0..\mathtt{a}'.\mathtt{length})$$

Slide 207



## Reasoning about `sum` - Proof Obligations (Final)

**line 7:** The midcondition $M_1$ must establish the postcondition of `sum`.
$(M_1 \wedge \mathbf{r} = \texttt{res}) \longrightarrow Q_1$

$$\texttt{a} \approx \texttt{a}_0 \ \wedge \ \texttt{res} = \sum \texttt{a[0..a.length)} \ \wedge \ \mathbf{r} = \texttt{res}$$
$$\longrightarrow$$
$$\texttt{a} \approx \texttt{a}_0 \ \wedge \ \mathbf{r} = \sum \texttt{a[0..a.length)}$$



Slide 208



## Reasoning about `sumAux` - Proof Obs. (Informal)

The correctness of `sumAux` relies on the following arguments:

**line 6:** The precondition of `sumAux` must establish the midcondition $M_2$.
i.e. $P_2 \ \wedge$ `i == a.length` $\longrightarrow M_2$

**line 7:** The midcondition $M_2$ must establish the postcondition of `sumAux`.
i.e. $M_2 \ \wedge$ `return 0` $\longrightarrow Q_2$

**line 9:** The precondition of `sumAux` must establish the midcondition $M_3$.
i.e. $P_2 \ \wedge \ \neg$(`i == a.length`) $\longrightarrow M_3$

**line 10:** The midcondition $M_3$ must establish the precondition of `sumAux`.
i.e. $M_3 \longrightarrow P_2$

**line 11:** The postcondition of `sumAux` must establish the midcondition $M_4$.
i.e. $Q_2 \ \wedge \ M_3 \ \wedge$ `val = a[i] + sumAux(a, i+1)` $\longrightarrow M_4$

**line 12:** The midcondition $M_4$ must establish the postcondition of `sumAux`.
i.e. $M_4 \ \wedge$ `return val` $\longrightarrow Q_2$

**Important:** obs. for lines 10 and 11 reference the **recursive** `sumAux` call.

Slide 209



## Reasoning about sumAux - Proof Obs. (Explicit)

The correctness of sumAux relies on the following arguments:

**line 6:** The precondition of sumAux must establish the midcondition $M_2$.
$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{i} = \mathtt{a}_0.\mathtt{length}) \longrightarrow M_2[\mathtt{a} \mapsto \mathtt{a}']$

**line 7:** The midcondition $M_2$ must establish the postcondition of sumAux.
$(M_2 \wedge \mathbf{r} = 0) \longrightarrow Q_2$

**line 9:** The precondition of sumAux must establish the midcondition $M_3$.
$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{i} \neq \mathtt{a}_0.\mathtt{length}) \longrightarrow M_3[\mathtt{a} \mapsto \mathtt{a}']$

**line 10:** The midcondition $M_3$ must establish the precondition of sumAux.
$M_3 \longrightarrow P_2[\mathtt{i} \mapsto \mathtt{i+1}]$

**line 11:** The postcondition of sumAux must establish the midcondition $M_4$.
$(Q_2[\mathtt{a}_0 \mapsto \mathtt{a}, \mathtt{a} \mapsto \mathtt{a}', \mathtt{i} \mapsto \mathtt{i+1}] \wedge M_3 \wedge \mathtt{val}' = \mathtt{a[i]} + \mathbf{r}) \longrightarrow M_4[\mathtt{a} \mapsto \mathtt{a}', \mathtt{val} \mapsto \mathtt{val}']$

**line 12:** The midcondition $M_4$ must establish the postcondition of sumAux.
$(M_4 \wedge \mathbf{r} = \mathtt{val}) \longrightarrow Q_2$



Slide 210



## Reasoning about sumAux - Proof Obs. (Final)

**line 6:** The precondition of sumAux must establish the midcondition $M_2$.
$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{i} = \mathtt{a}_0.\mathtt{length}) \longrightarrow M_2[\mathtt{a} \mapsto \mathtt{a}']$

$$\mathtt{a}_0 \neq \mathtt{null} \ \wedge \ 0 \leq \mathtt{i} \leq \mathtt{a}_0.\mathtt{length} \ \wedge \ \mathtt{a}' \approx \mathtt{a}_0 \ \wedge \ \mathtt{i} = \mathtt{a}_0.\mathtt{length}$$
$$\longrightarrow$$
$$\mathtt{a}' \approx \mathtt{a}_0 \ \wedge \ \mathtt{i} = \mathtt{a}'.\mathtt{length}$$



Note: due to Java's call by value semantics the variable i cannot be modified in the body.

Slide 211



## Reasoning about `sumAux` - Proof Obs. (Final)

**line 7:** The midcondition $M_2$ must establish the postcondition of `sumAux`.
$(M_2 \wedge \mathbf{r} = 0) \longrightarrow Q_2$

$$\mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathtt{i} = \mathtt{a.length} \ \wedge \ \mathbf{r} = 0$$
$$\longrightarrow$$
$$\mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathbf{r} = \sum \mathtt{a}[\mathtt{i}..\mathtt{a.length})$$



There are a two interesting points to note about the proof obligations above.

Firstly, since `i` $=$ `a.length` we can infer from this that `a` $\neq$ `null`. This is because `a.length` returns a value which is comparable to `i`, so there cannot have been a null pointer dereference.

Secondly, proving that **r** has the desired return value relies on recalling from the definition of $\sum$ that:

$$\forall k.[\ \sum \mathtt{a}[\mathtt{k}..\mathtt{k}) = 0\ ]$$

That is, the sum of any empty range is always equal to 0 (the identity element of addition).

Slide 212



## Reasoning about sumAux - Proof Obs. (Final)

**line 9:** The precondition of sumAux must establish the midcondition $M_3$.
$(P_2[\mathtt{a} \mapsto \mathtt{a}_0] \wedge \mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{i} \neq \mathtt{a}_0.\mathtt{length}) \longrightarrow M_3[\mathtt{a} \mapsto \mathtt{a}']$

$$\mathtt{a}_0 \neq \mathtt{null} \wedge 0 \leq \mathtt{i} \leq \mathtt{a}_0.\mathtt{length} \wedge \mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{i} \neq \mathtt{a}_0.\mathtt{length}$$
$$\longrightarrow$$
$$\mathtt{a}' \approx \mathtt{a}_0 \wedge \mathtt{a}' \neq \mathtt{null} \wedge 0 \leq \mathtt{i} < \mathtt{a}'.\mathtt{length}$$



Slide 213



## Reasoning about sumAux - Proof Obs. (Final)

**line 10:** The midcondition $M_3$ must establish the precondition of sumAux.
$M_3 \longrightarrow P_2[\mathtt{i} \mapsto \mathtt{i+1}]$

$$\mathtt{a} \approx \mathtt{a}_0 \wedge \mathtt{a} \neq \mathtt{null} \wedge 0 \leq \mathtt{i} < \mathtt{a}.\mathtt{length}$$
$$\longrightarrow$$
$$\mathtt{a} \neq \mathtt{null} \wedge 0 \leq \mathtt{i+1} \leq \mathtt{a}.\mathtt{length}$$



Clearly we could also explicitly denote the substitution on $P_2$ of $\mathtt{a} \mapsto \mathtt{a}$.

However, recall that we choose to omit unnecessary identity substitutions to ease the notational burden of our proof obligations.

Slide 214



## Reasoning about sumAux - Proof Obs. (Final)

**line 11:** The postcondition of sumAux must establish the midcondition $M_4$.

$$(Q_2[\mathtt{a}_0 \mapsto \mathtt{a}, \mathtt{a} \mapsto \mathtt{a}', \mathtt{i} \mapsto \mathtt{i+1}] \wedge M_3 \wedge \mathtt{val}' = \mathtt{a[i]} + \mathbf{r}) \longrightarrow M_4[\mathtt{a} \mapsto \mathtt{a}', \mathtt{val} \mapsto \mathtt{val}']$$

$$\begin{array}{c} \mathtt{a}' \approx \mathtt{a} \ \wedge \ \mathbf{r} = \sum \mathtt{a}'[\mathtt{i+1..a}'.\mathtt{length}) \\ \wedge \ \mathtt{a} \approx \mathtt{a}_0 \ \wedge \ \mathtt{a} \neq \mathtt{null} \ \wedge \ 0 \leq \mathtt{i} < \mathtt{a.length} \ \wedge \ \mathtt{val}' = \mathtt{a[i]} + \mathbf{r} \\ \longrightarrow \\ \mathtt{a}' \approx \mathtt{a}_0 \ \wedge \ \mathtt{val}' = \sum \mathtt{a}'[\mathtt{i..a}'.\mathtt{length}) \end{array}$$



It is important that the premise of our proof obligation includes the fact that

$$0 \leq \mathtt{i} < \mathtt{a.length}$$

so that we know the array dereference of `a[i]` in the code is valid (does not result in an `ArrayOutOfBounds` error).

Also note that if line 10 of our program were modified to:

```
int val = sumAux(a, i+1) + a[i]
```

then it is important to note that the proof obligation above would have to be modified to track the new order of execution. In particular, the effect of the code would now be described as $\mathtt{val}' = \mathbf{r}$ + $\mathtt{a}'[\mathtt{i}]$

Note that when we add the $i^{th}$ element of the array to the sum, we do so from the array that was modified by the call to sumAux. Of course, the specification guarantees that the array is unmodified, so in this case the ordering makes no difference.

Slide 215



## Reasoning about `sumAux` - Proof Obs. (Final)

**line 12:** The midcondition $M_4$ must establish the postcondition of `sumAux`.
$(M_4 \wedge \mathbf{r} = \texttt{val}) \longrightarrow Q_2$

$$\texttt{a} \approx \texttt{a}_0 \ \wedge \ \texttt{val} = \sum \texttt{a[i..a.length)} \ \wedge \ \mathbf{r} = \texttt{val}$$
$$\longrightarrow$$
$$\texttt{a} \approx \texttt{a}_0 \ \wedge \ \mathbf{r} = \sum \texttt{a[i..a.length)}$$



A lot of what we have covered in this section may feel quite mechanical and it is correct to have this feeling. Everything we have done so far could be automated. In fact, there are a number of proof assistant tools that can do just this.

However, when we reason about iterative programs in the next part of the course, we will see that this requires more intuition, and so this is much harder to automate.

Slide 216



## Is There Anything We Have Overlooked?

Consider the following Java method `sillySum`:

```
1  int sillySum(int[] a)
2  // PRE: a ≠ null                                          (P)
3  // POST: a ≈ a0  ∧  r = ∑ a[0..a.length)                  (Q)
4  {
5    return sillySum(a);
6  }
```

...so what about Termination?

We need to prove that

$$\forall a \in \texttt{int[]}, i \in \mathbb{Z}.[\ 0 \leq i \leq a.\texttt{length} \longrightarrow \texttt{sumAux}(a,i) \text{ terminates }]$$

Which we could tackle this proof using mathematical induction considering the length of the remaining array to process i.e. `a.length` − `i`.



Note that the `sillySum` method would satisfy its postcondition *if* it were to terminate. We call this **partial correctness**. However, termination of `sillySum` is not guaranteed (or in this case even possible), so the method is not **totally correct**.

We will discuss these terms in more detail a little later in the course.

Slide 217



## Comparison with Induction

Recall the Haskell function `sum` and its tail-recursive version `sum_tr` from earlier in the course:

```
sum :: [Int] -> Int
sum [] = 0
sum i:is = i + sum is

sum_tr :: [Int] -> Int -> Int
sum_tr [] k = k
sum_tr (i:is) k = sum_tr is (i+k)
```

Using Structural Induction we were able to prove:

$$\forall \texttt{is:[Int]}.\ \texttt{sum is} = \texttt{sum\_tr is 0}$$

However, we have just proven a very similar property for our Java method `sumAux`, namely that is satisfies the specification:

$$\{\ \texttt{a} \neq \texttt{null}\}\quad \texttt{int res = sumAux(a,0);}\quad \{\ \texttt{res} = \sum \texttt{a}[0..\texttt{a.length})\}$$

but here we did not make use of any form of Induction ... or did we ... ?



Slide 218



## Comparison with Induction

The correctness of `sumAux` relies on the following arguments:

**line 6:** The precondition of `sumAux` must establish the midcondition $M_2$.
i.e. $P_2 \wedge$ `i == a.length` $\longrightarrow M_2$

**line 7:** The midcondition $M_2$ must establish the postcondition of `sumAux`.
i.e. $M_2 \wedge$ `return 0` $\longrightarrow Q_2$

**line 9:** The precondition of `sumAux` must establish the midcondition $M_3$.
i.e. $P_2 \wedge \neg$(`i == a.length`) $\longrightarrow M_3$

**line 10:** The midcondition $M_3$ must establish the precondition of `sumAux`.
i.e. $M_3 \longrightarrow P_2$

**line 11:** The postcondition of `sumAux` must establish the midcondition $M_4$.
i.e. $Q_2 \wedge M_3 \wedge$ `val = a[i] + sumAux(a, i+1)` $\longrightarrow M_4$

**line 12:** The midcondition $M_4$ must establish the postcondition of `sumAux`.
i.e. $M_4 \wedge$ `return val` $\longrightarrow Q_2$

- lines 6 - 7 corresponds to proving the Base Case
- lines 9 - 12 correspond to proving the Inductive Step
- line 10 corresponds to checking that we can apply the Inductive Hypothesis
- line 11 use of $Q_2$ corresponds to use of the Inductive Hypothesis

# Reasoning about Recursion - Conclusions

Reasoning about recursive functions in Java resembles reasoning about recursive functions in Haskell:

- Reasoning about the cases where a function terminates corresponds to proving the base cases.
- Reasoning about the cases where a function calls itself recursively corresponds to proving the inductive steps.
- The point where we establish that the precondition of the callee holds, corresponds to establishing that the inductive hypothesis holds.
- The point where we establish that the postcondition of the callee implies the midcondition of the caller, corresponds to the conclusion of the inductive step.