Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Introduction

You can learn a programming language but still not know how to program.

Introduction

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You've seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

I aim to provide you with that methodology.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

I aim to provide you with that methodology.

Since I don’t want to teach a language, I’ll stick to a tiny, universal one.

You can learn a programming language but still not know how to program.

You learned the constructs, but have little idea how to use them.

You’ve seen finished programs, but struggle to produce one yourself.

You will grope in the dark until someone shows you a methodology.

I aim to provide you with that methodology.

Since I don’t want to teach a language, I’ll stick to a tiny, universal one.

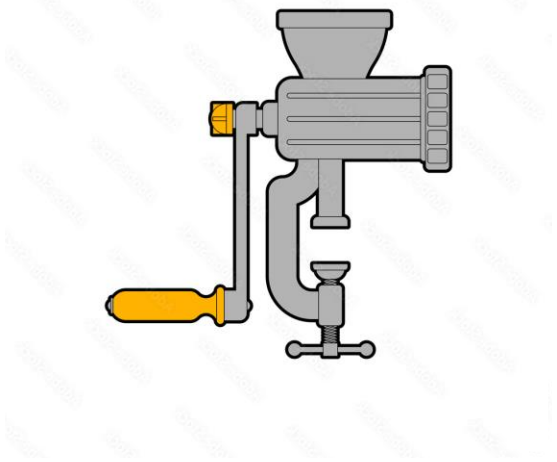
It’s a subset of Java, Python, C/C++, JavaScript, …, (any imperative language).

Motivation

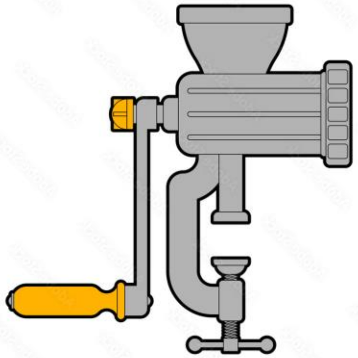
Can programming be mechanized?

Can programming be mechanized?

Requirements

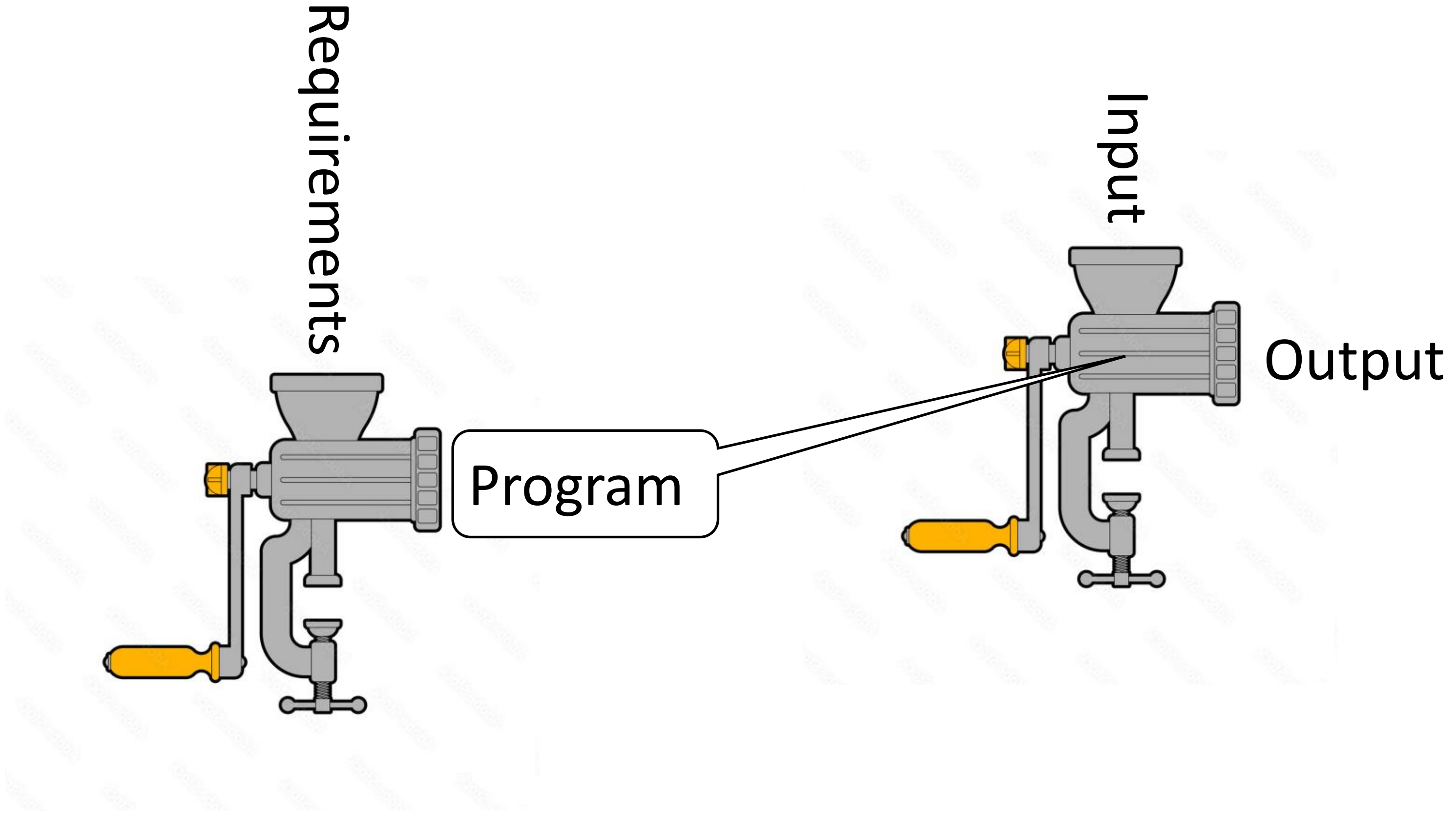


Can programming be mechanized?

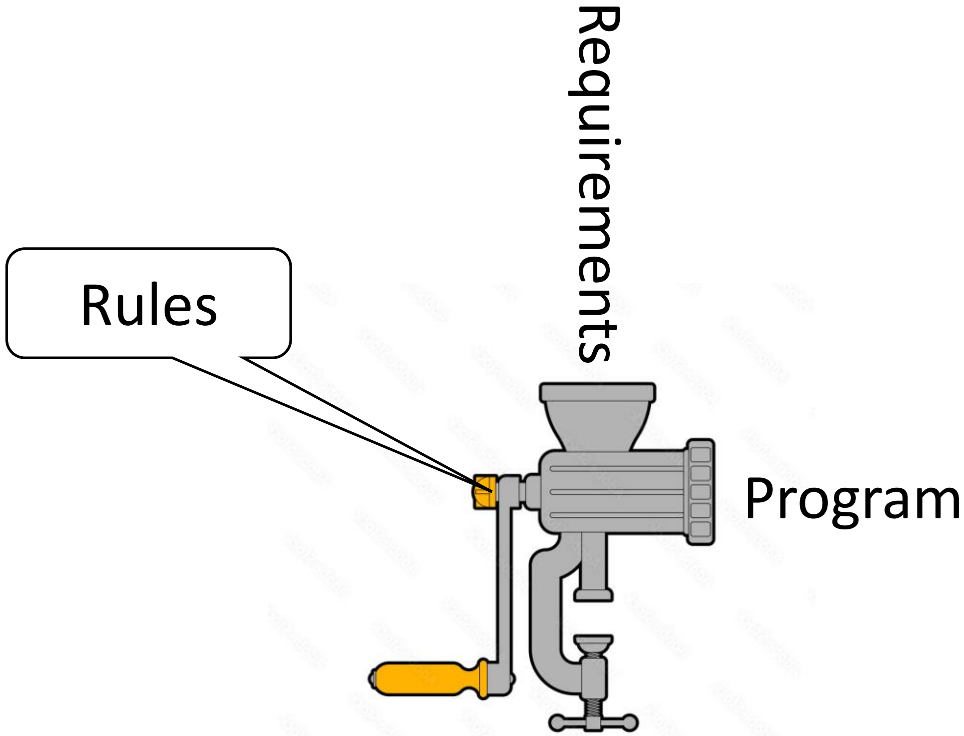


Motivation

Can programming be mechanized?



Can programming be mechanized?

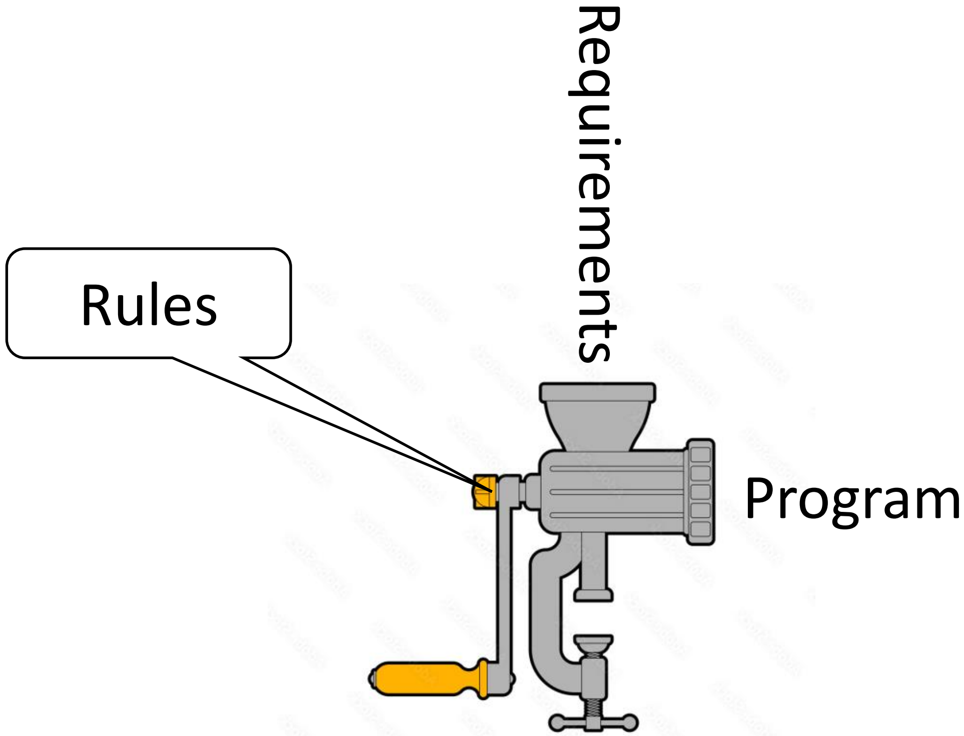


Fully-automatic programming would need rules that are:

- Effective
- Produce good code
- Efficient
- Complete

Motivation

Can programming be mechanized?



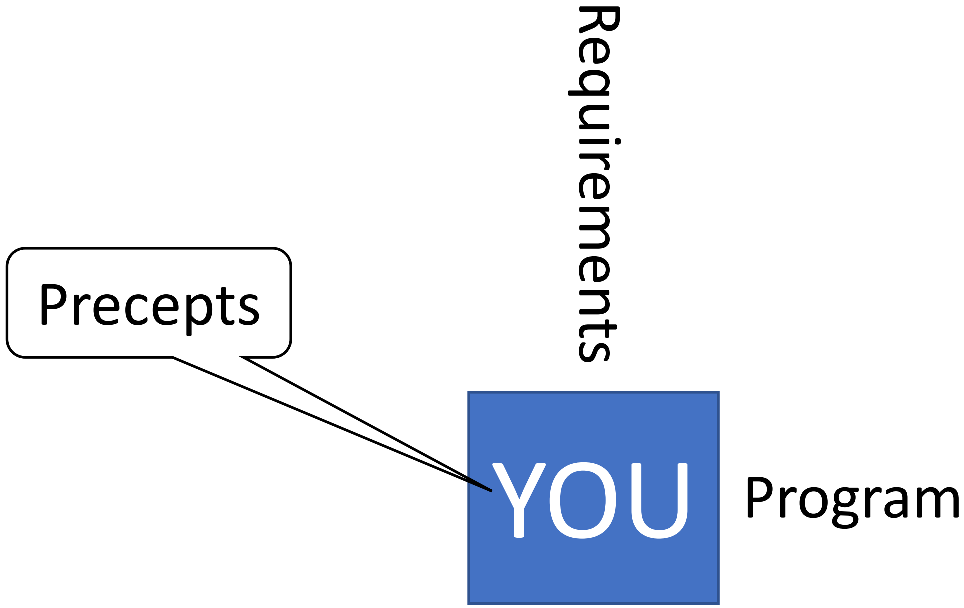
Fall back

- Rules for people
- Make programming
  - Easy
  - Accurate

Precepts

pre·cept

A command or principle intended especially as a general rule of action



Fall back

- Rules for people
- Make programming
    - Easy
    - Accurate

pre·cept

A command or principle intended especially as a general rule of action

First Precept

pre·cept

A command or principle intended especially as a general rule of action

First Precept

☞ **Follow programming precepts.**

pre·cept

A command or principle intended especially as a general rule of action

Second Precept

pre·cept

A command or principle intended especially as a general rule of action

Second Precept

☞ **Ignore precepts, when appropriate.**

pre·cept

A command or principle intended especially as a general rule of action

INCONSISTENCIES

☞ **Code with deliberation. Be mindful.**

Sample precept

Sample precept

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Sample precept, with an application:

```
amount = price * quantity;
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Same precept, with a different application:

```
piece = board[row+deltaRow[direction]][column+deltaColumn[direction]];
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

Same precept, with a different application:

```
piece = board[row+deltaRow[direction]][column+deltaColumn[direction]];
```

```
piece = B[r+deltaR[d]][c+deltaC[d]];
```

☞ **Aspire to making code self-documenting by choosing descriptive names.**

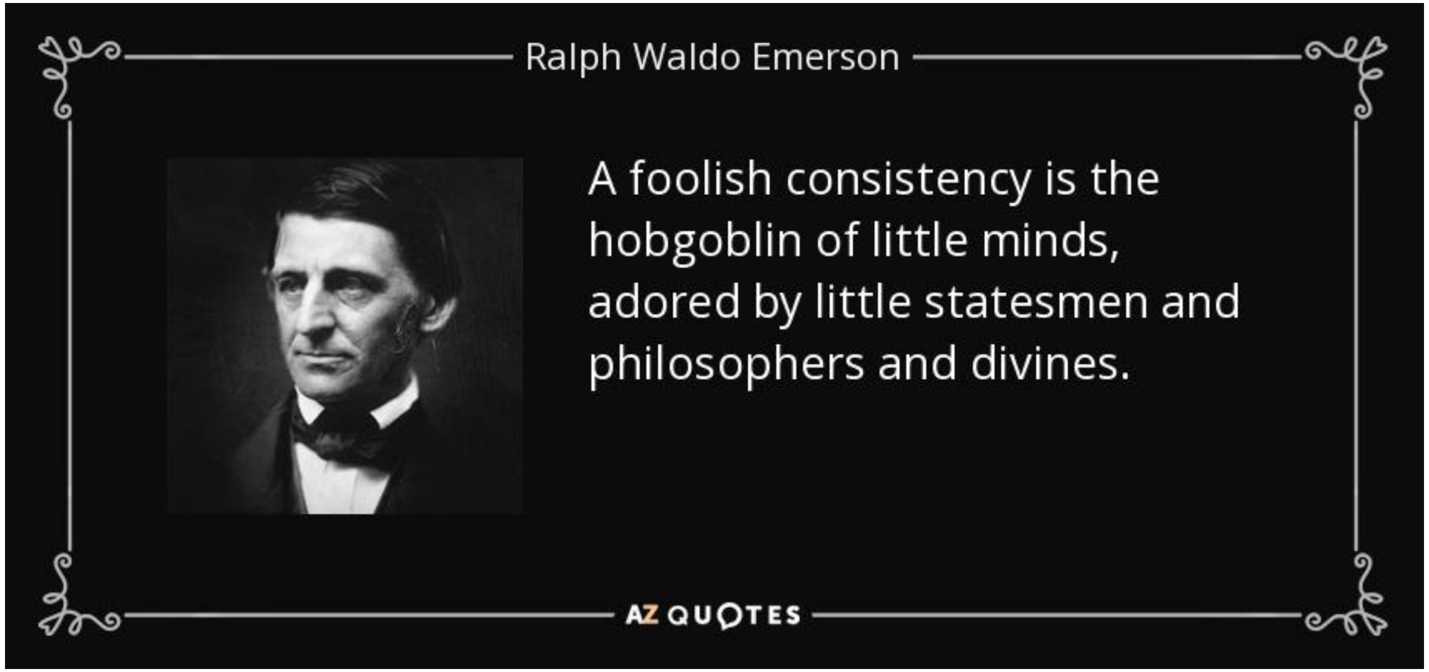
Alternative precept

```
piece = board[row+deltaRow[direction]][column+deltaColumn[direction]];

piece = B[r+deltaR[d]][c+deltaC[d]];
```

☞ **Use single-letter variable names when it makes code more understandable.**

Precepts



☞ **Resolve contradictory precepts with care.**

Exercise judgement

Make tradeoffs

Don’t make decisions casually

Indulge in personal preference

☞ **Resolve contradictory precepts with care.**

☞ **Be humble. Programming is hard and error prone. Respect it.**

Despite your humility, aim for perfection

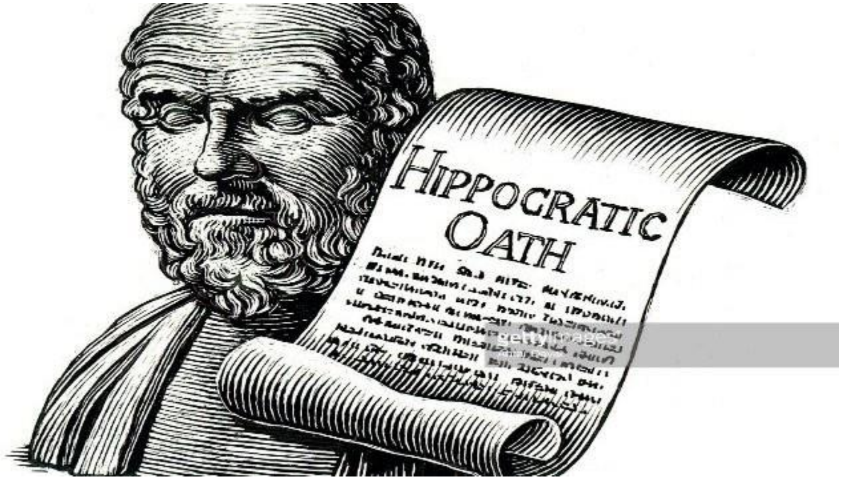
- The quality of the code you write
- The quality of the process you use to write it

☞ **Be humble. Programming is hard and error prone. Respect it.**

Process quality

☞ **Aspire to code it right the first time.**

Process quality: Hippocratic Coding



☞ **Aspire to code it right the first time. Do no harm. Avoid writing code that must be redone.**

An approach to Hippocratic Coding: Patterns

☞ **Master stylized code patterns, and use them.**

Sample programming problem

☞ **Master stylized code patterns, and use them.**

Pattern: compute-use

```
/* Compute. */
/* Use. */
```

☞ **Master stylized code patterns, and use them.**

Application of compute-use

```
/* Compute k. */
/* Use k. */
```

☞ **Master stylized code patterns, and use them.**

Application of compute-use

```
/* Compute k. */
   k = thus-and-such;
/* Use k. */
   if ( /* k has some desired property */ ) /* Do this and that. */
```

☞ **Master stylized code patterns, and use them.**

Application of compute-use

```
/* Compute k. */
```

Smaller hairy mess

```
   k = thus-and-such;
/* Use k. */
   if ( /* k has some desired property */ ) /* Do this and that. */
```

☞ **Master stylized code patterns, and use them.**

Another sample programming problem

☞ **Master stylized code patterns, and use them.**

Another pattern: indeterminate iteration

```
/* Enumerate from start. */
   int k = start;
   while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

Another pattern: indeterminate iteration

```
/* Enumerate from start. */
   int k = start;
   while ( condition ) k++;
```

Effect

Initialize k to *start*

Repeatedly add 1 to k

provided *condition* is **true**

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

```
int k = start;
while ( condition ) k++;
```

Indeterminate iteration is an instance of the general interative-computation pattern.

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

```
int k = start;
while ( condition ) k++;
```

Indeterminate iteration is an instance of the general interative-computation pattern.

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

```
int k = start;
while ( condition ) k++;
```

Indeterminate iteration is an instance of the general interative-computation pattern.

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

```
int k = start;
while ( condition ) k++;
```

Indeterminate iteration is an instance of the general interative-computation pattern.

☞ **Master stylized code patterns, and use them.**

Yet another pattern: general iterative computation

```
/* Initialize. */
while ( /* not finished */ ) {

   /* Go on to next. */
   }
```

```
int k = start;
while ( condition ) k++;
```

Indeterminate iteration is an instance of the general interative-computation pattern.

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Shorthand: general iterative computation

```
for (initialize; condition; go-on-to-next) compute
```

```
/* Initialize. */
while ( /* not finished */ ) {
   /* Compute. */
   /* Go on to next. */
   }
```

☞ **Master stylized code patterns, and use them.**

Convention: We could use a **for**-statement to express indeterminate iteration

```
for (initialize; condition; go-on-to-next) compute
```

```
int k = start;
while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

Convention: We could use a **for**-statement to express indeterminate iteration

```
for (int k=start; condition; go-on-to-next) compute
```

```
int k = start;
while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

Convention: We could use a **for**-statement to express indeterminate iteration

```
for (int k=start; condition; k++) compute
```

```
int k = start;
while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

Convention: We could use a **for**-statement to express indeterminate iteration

```
for (int k=start; condition; k++);
```

```
int k = start;
while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

But, by convention: Never use a **for**-statements for indeterminate iteration

```
for (int k=start; condition; k++);
```

```
int k = start;
while ( condition ) k++;
```

☞ **Master stylized code patterns, and use them.**

Convention: Reserve **for**-statements for determinate iteration

```
for (int k=start; k<limit; k++) compute
for (int k=start; k<=limit; k++) compute
```

k goes up

```
for (int k=start; k>limit; k--) compute
for (int k=start; k>=limit; k--) compute
```

k goes down

☞ **Master stylized code patterns, and use them.**

Another approach to Hippocratic Coding: Analysis

☞ **Aspire to code it right the first time. Do no harm. Avoid writing code that must be redone.**

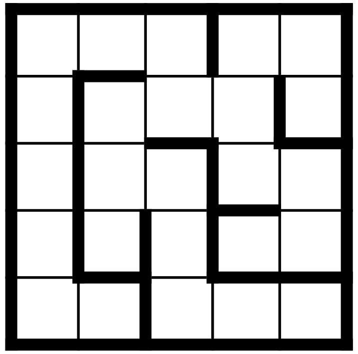
Another approach to Hippocratic Coding: Analysis

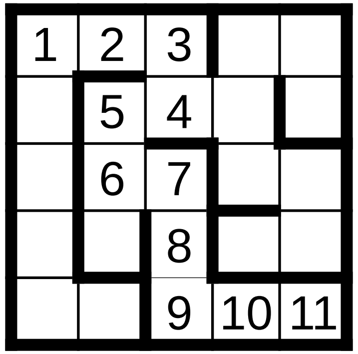
☞ **Analyze first.**

Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.

Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs "Unreachable" otherwise. A path is direct if it never visits any cell more than once.





Analysis

- Problem
- Architecture
- Data
- Components

☞ **Analyze first.**

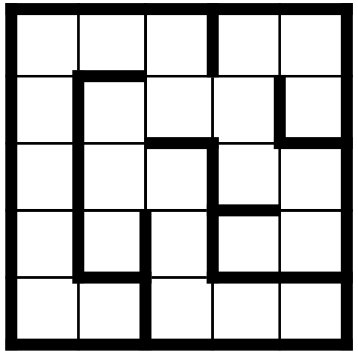
Analysis

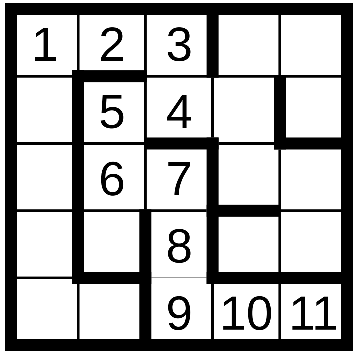
Problem

☞ **Make sure you understand the problem.**

Example: Running a Maze

Background. Define a maze to be a square two-dimensional grid of cells separated (or not) from adjacent cells by walls. One can move between adjacent cells if and only if no wall divides them. A solid wall surrounds the entire grid of cells, so there is no escape from the maze.
Problem Statement. Write a program that inputs a maze, and outputs a direct path from the upper-left cell to the lower-right cell if such a path exists, or outputs “Unreachable” otherwise. A path is direct if it never visits any cell more than once.

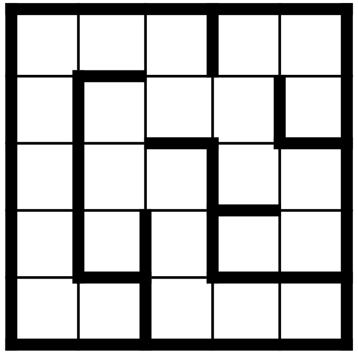


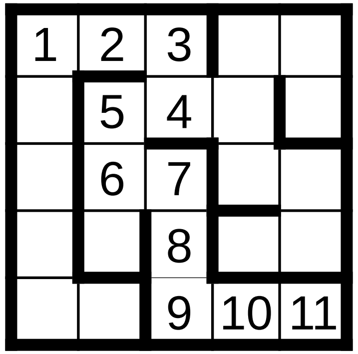


☞ **Make sure you understand the problem.**

Example: Running a Maze

- Do I understand each noun: *maze*, *grid*, *cell*, *wall*, *path*, and *direct path*?
- Do I understand the verbs: Specifically, how does one *move* between cells?
- How is a maze represented in the input?
- Is there any upper limit on the size of a maze? Is there a lower limit?
- What is the expected program behavior if the input is not well-formed?
- Is a direct path the same as a shortest path?
- What if there is more than one direct path?
- How is a path to be displayed in the output?





☞ **Make sure you understand the problem.**

Architecture: What sort of computation will it be?

Architecture: What sort of computation will it be?

- **Online**. Read a sequence of inputs, and process them on the fly.
- **Offline**. Read all inputs, perform a computation, output result.
- **Other**.

Architecture: Offline computation pattern

```
/* Input. */
/* Compute. */
/* Output. */
```

Architecture: Restate the problem on the architecture

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

Programs: Instructions for manipulating values

Instructions: code

Values: data

Patterns and Architecture: Code-centered perspective

☞ **Dovetail thinking about code and data.**

Code

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

☞ **Dovetail thinking about code and data.**

Data

maze

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

path

☞ **Dovetail thinking about code and data.**

External Data

external data (maze)

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

internal data (path)

☞ **Dovetail thinking about code and data.**

Variables

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
```

maze

```
/* Compute a direct path through the maze, if one exists. */
```

path

```
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

☞ **Specify how individual program steps will cooperate with one another.**

Internal Data

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
```

internal data (maze)

```
/* Compute a direct path through the maze, if one exists. */
```

internal data (path)

```
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

☞ **A program’s internal data representation is central to the code; consider it early.**

External Data

external data (maze)

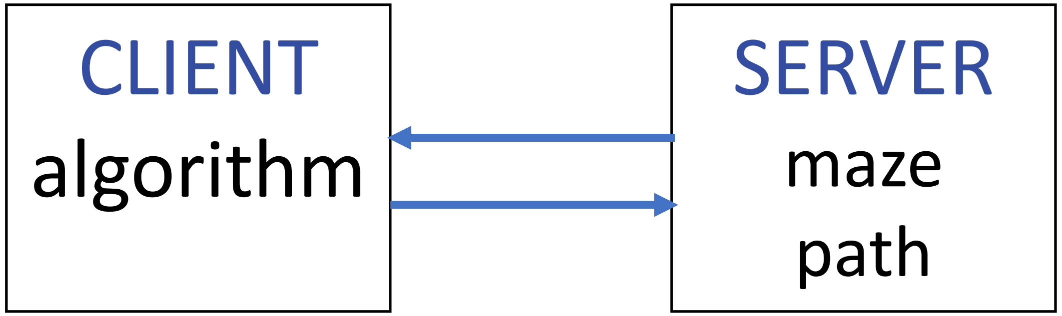
```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

internal data (path)

☞ **Consider a program’s external data representation late.**

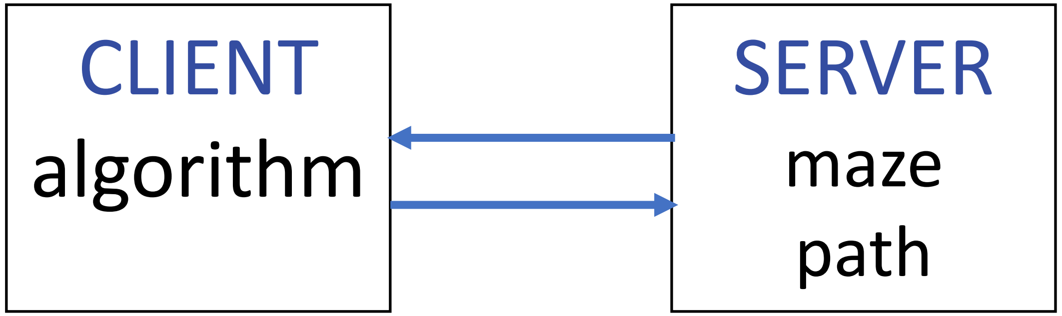
Components

- A program can be organized into components.
- Distinguish between the maze-running algorithm (a client of data) and the data itself (housed in a server).
- What operations are needed by the client?
- What operations can be provided by the server?
- Resolve differences by negotiation.



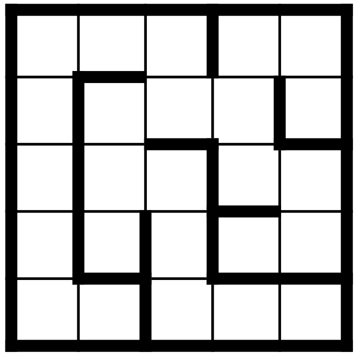
Components

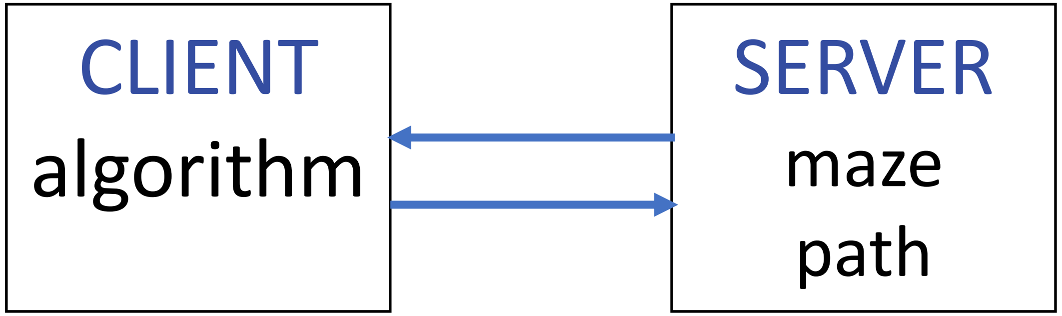
- Some aspects of data are static, i.e., don't change (maze)
- The client learns of static data by queries.
- Other aspects of data are dynamic, i.e., change (path)
- The client is an actor that effects changes by actions:
  - extend path (if possible); retract path (if necessary)
  - The cumulative effect of actions is recorded in state.



Components

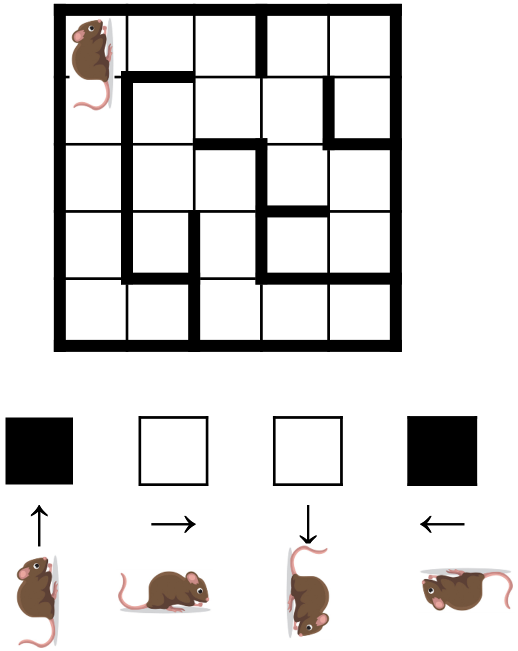
- A client may have/want global perspective
  - algorithm is aware of the full maze

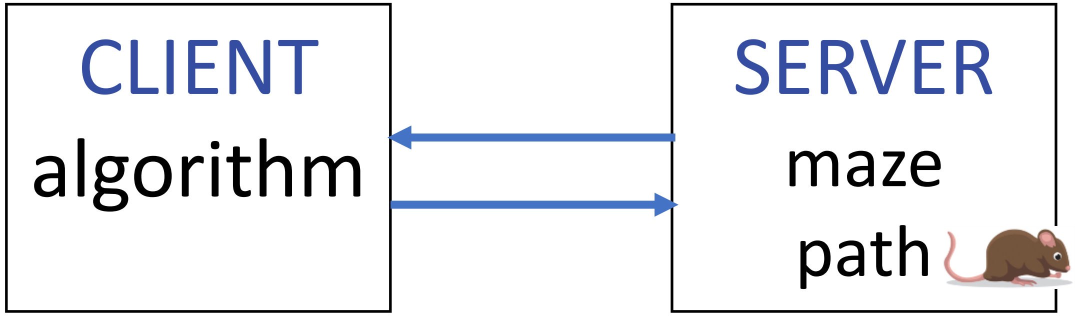




Components

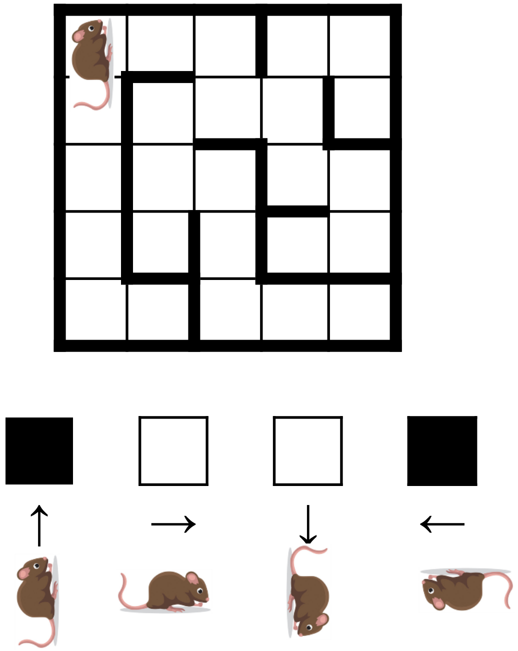
- A client may have/want global perspective
  - algorithm is aware of the full maze
- Other clients have/want only local perspective
  - rat is unaware of full maze

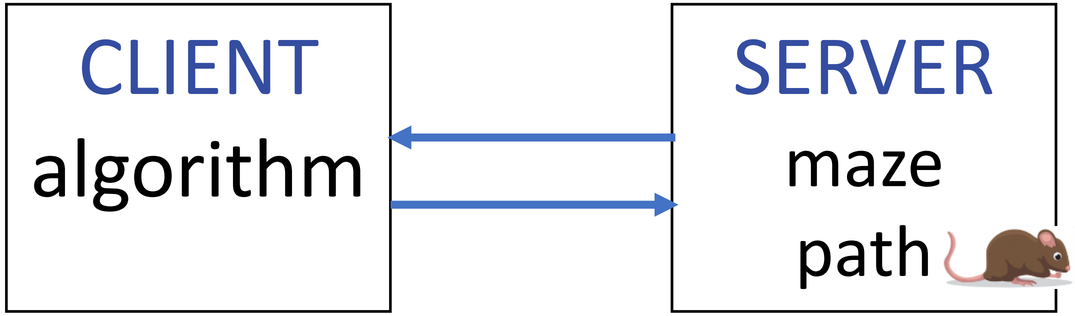




Components

- A client may have/want global perspective
  - algorithm is aware of the full maze
- Other clients have/want only local perspective
  - rat is unaware of full maze





Analysis

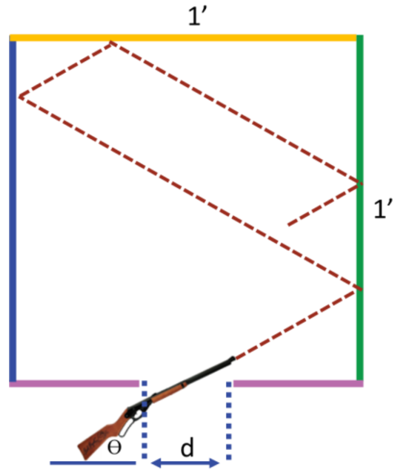
Another programming problem

☞ **Analyze first.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

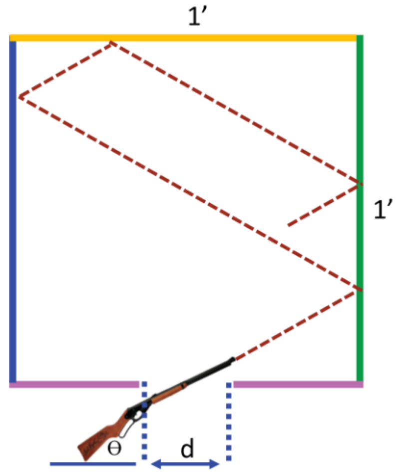
Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



☞ **Analyze first.**

Example: Output the sum of the integers between 1 and n

Example: Output the sum of the integers between 1 and n

```
/* Output the sum of 1 through n. */
```

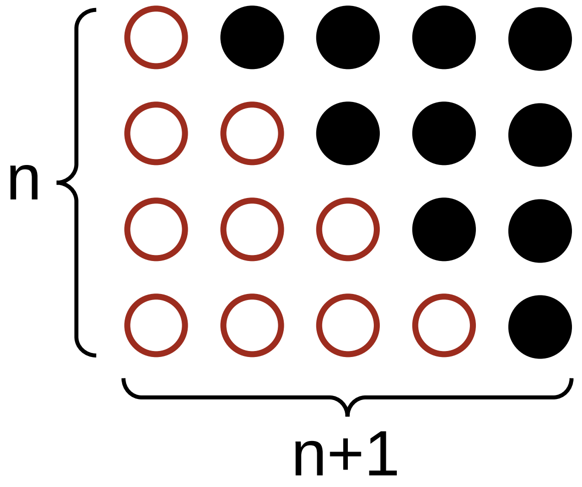
Example: Output the sum of the integers between 1 and n

```
/* Output the sum of 1 through n. */
   int sum = 0;
   for (int k=1; k<=n; k++) sum = sum + k;
   System.out.println( sum );
```

knee-jerk, brute force

Example: Output the sum of the integers between 1 and n

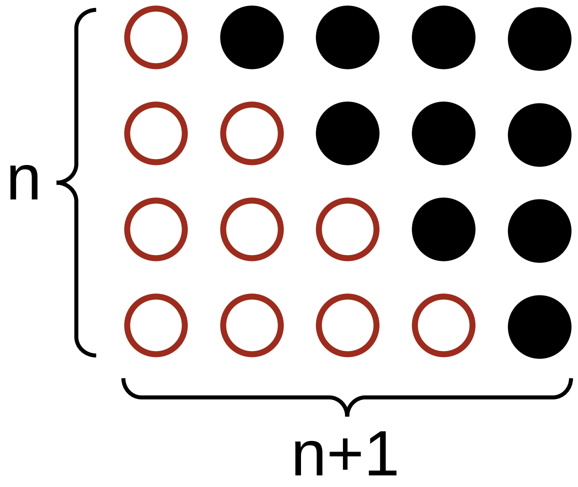
```
/* Output the sum of 1 through n. */
   int sum = 0;
   for (int k=1; k<=n; k++) sum = sum + k;
   System.out.println( sum );
```



☞ **Analyze first.**

Example: Output the sum of the integers between 1 and n

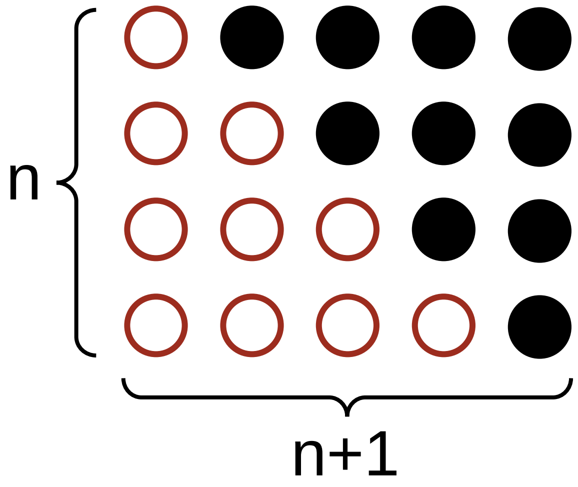
```
/* Output the sum of 1 through n. */
   System.out.println( n*(n+1)/2 );
```



☞ **Analyze first.**

Example: Output the sum of the integers between 1 and n

```
/* Output the sum of 1 through n. */
   System.out.println( n*(n+1)/2 );
```

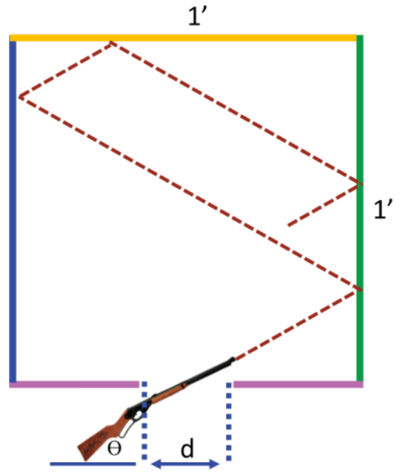


☞ **Sometimes iteration is unnecessary because a closed-form solution is available.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

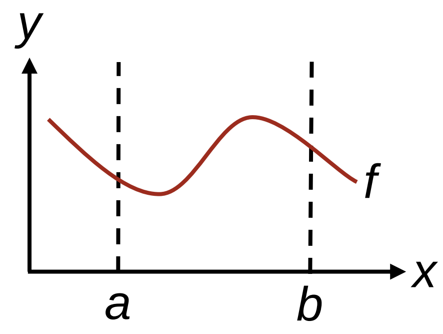
Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



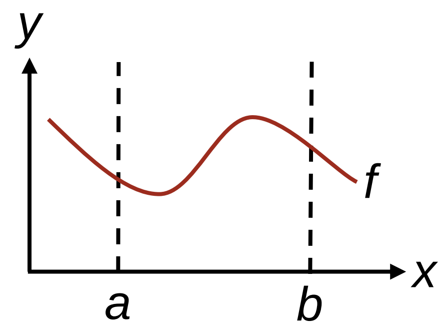
☞ **Sometimes iteration is unnecessary because a closed-form solution is available.**

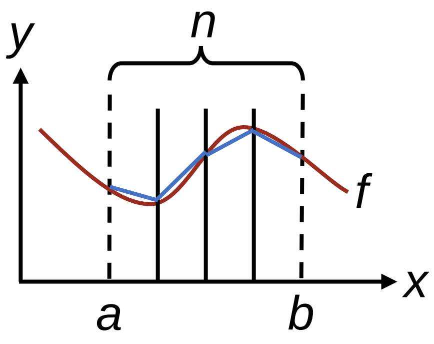
Analogy: A possible source of inspiration

Analogy: Computing arc length s of a curve y=*f(x),* between a and b.

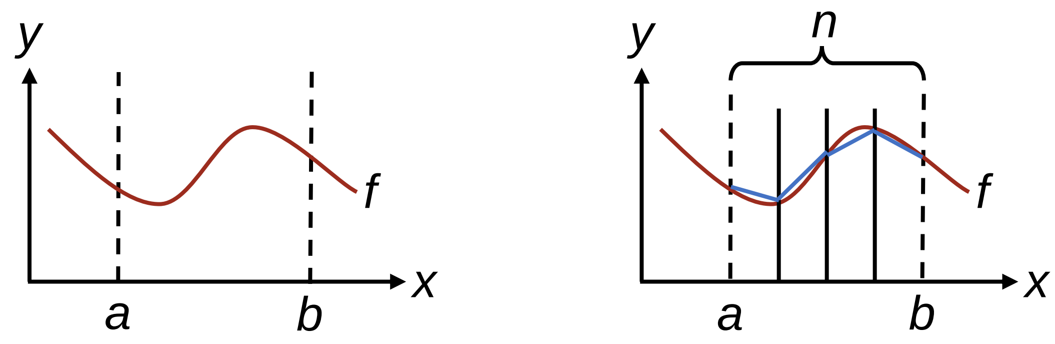


Analogy: Computing arc length s of a curve y=*f(x),* between a and b.



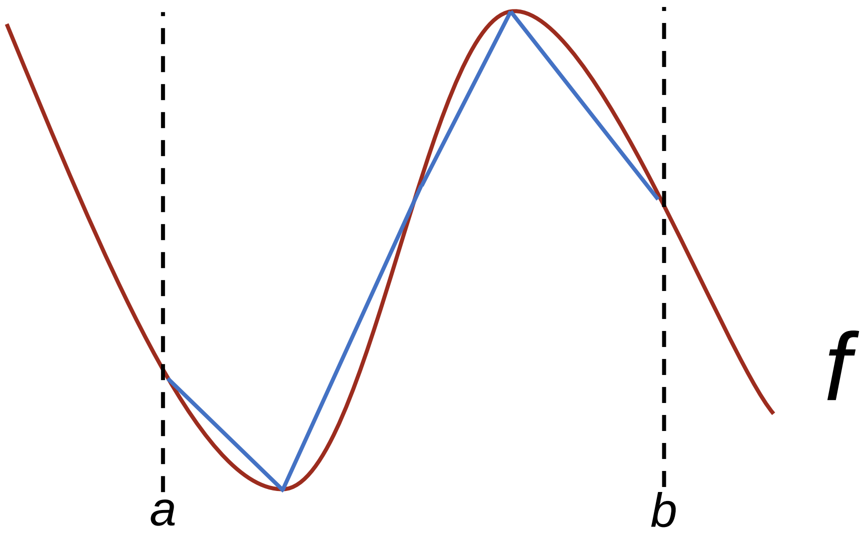


Analogy: Computing arc length s of a curve y=*f(x),* between a and b.

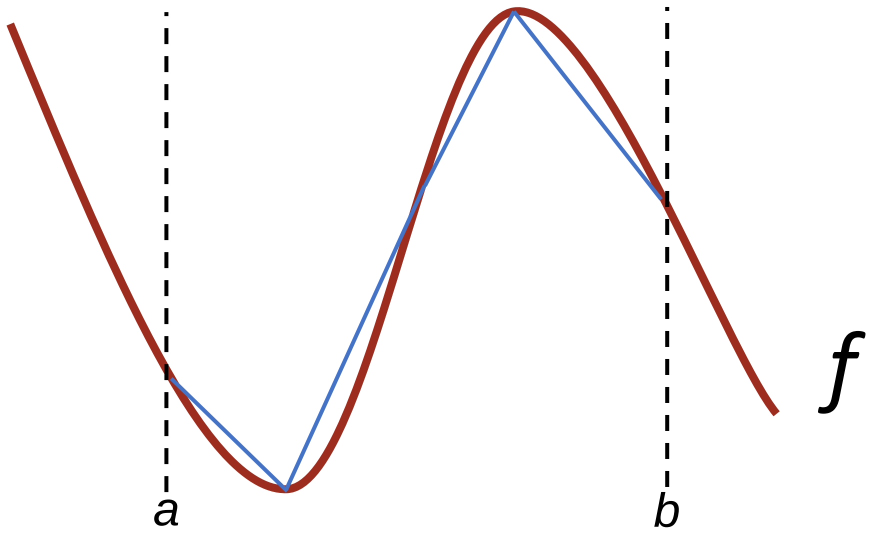


$$s = \int_a^b \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\; dx$$

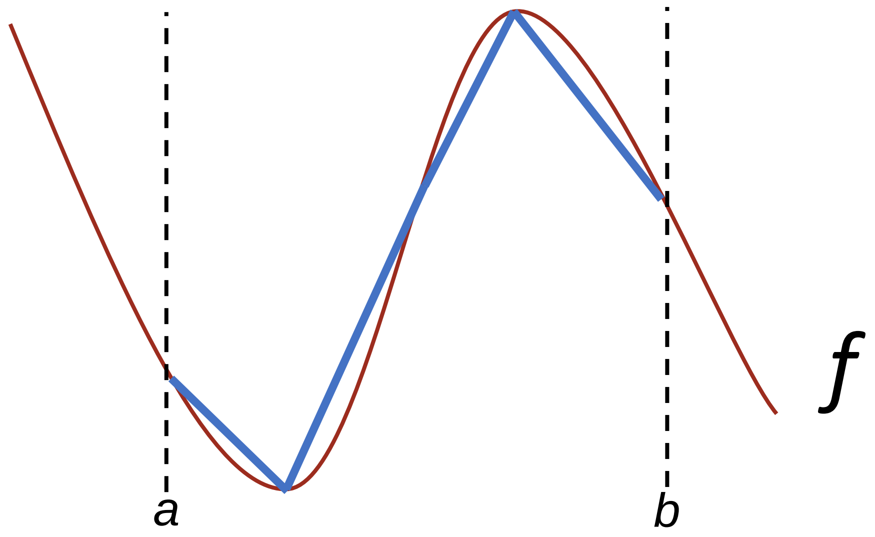
Analogy: Where does the analogy faulter?



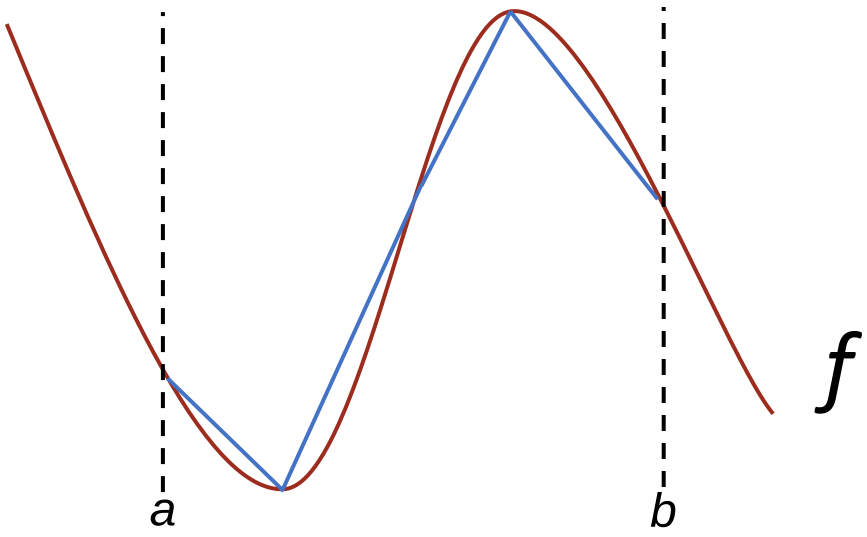
Analogy: In the calculus problem, we seek the length of f



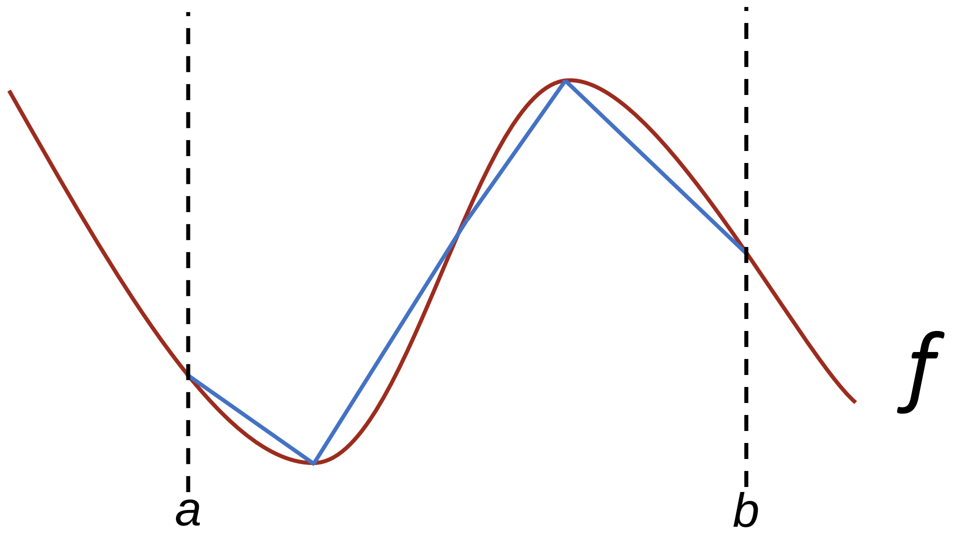
Analogy: In the bee-bee problem, we seek the total lengths of the pieces



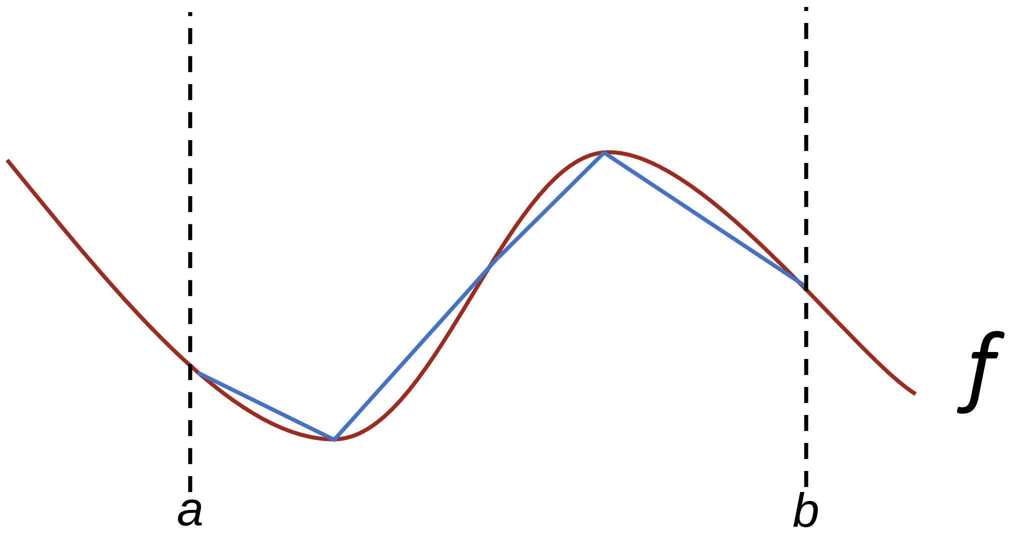
Analogy: How can we unify the two points of view?



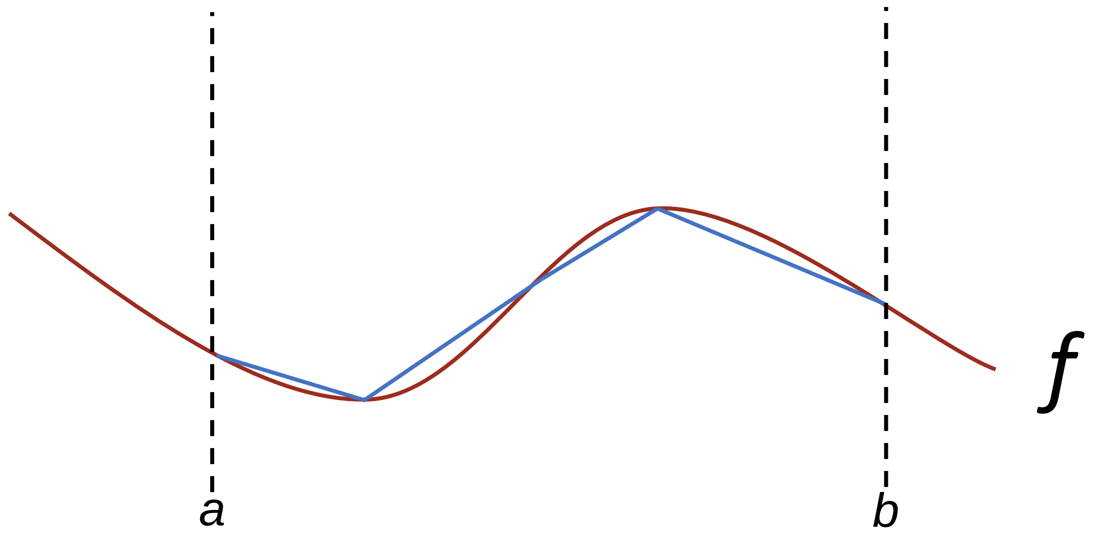
Analogy: By finding a related problem where they are the same.



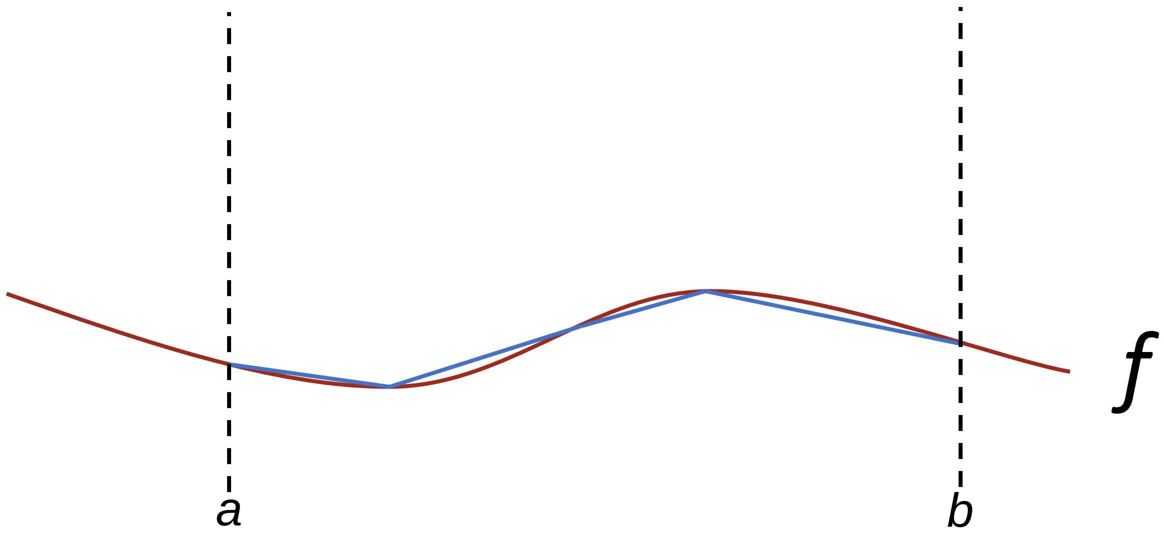
Analogy: By finding a related problem where they are the same



Analogy: By finding a related problem where they are the same



Analogy: By finding a related problem where they are the same

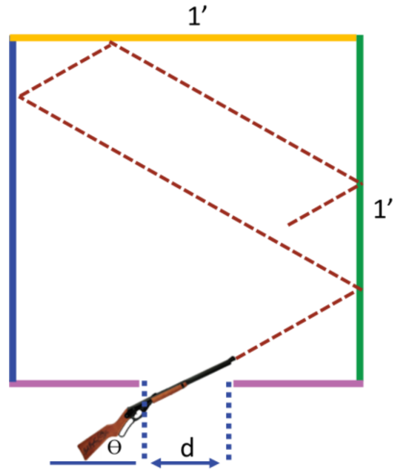


Analogy: By finding a related problem where they are the same

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.

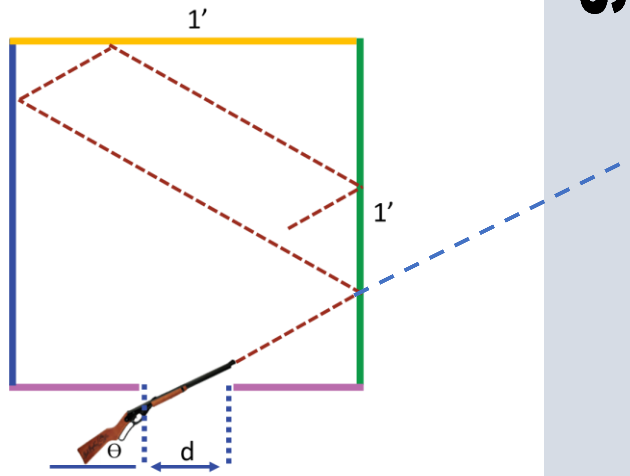


☞ **Solve a different problem, and use that solution to solve the original problem.**

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



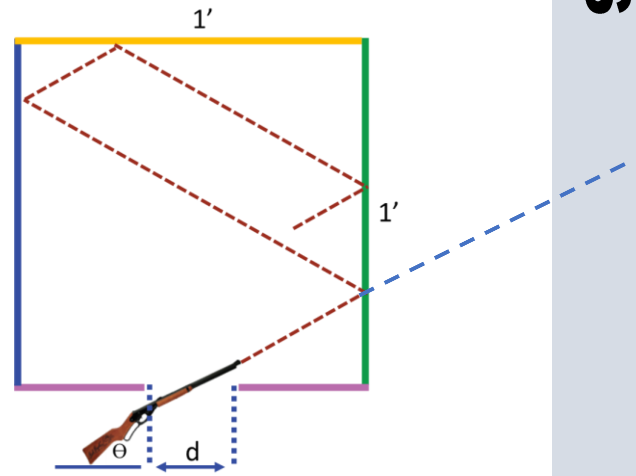
☞ **Solve a different problem, and use that solution to solve the original problem.**

Analysis

Example: Ricocheting Bee-Bee

Background. A square tin box measuring one foot on each side has a slit of size d centered on one side. Insert a bee-bee gun at the center of the slit at angle Θ, and shoot. The bee-bee ricochets off sides, one after another. On each ricochet, the angle of reflection is equal to the angle of incidence.

Problem Statement. Write a program that inputs d and Θ, and outputs the total distance the bee-bee travels before it exits.



☞ **Solve a different problem, and use that solution to solve the original problem.**

Hippocratic Coding:-

- ☞ Aspire to code it right the first time.

Patterns:

- ☞ Master stylized code patterns, and use them.

Analysis:

- ☞ Analyze first.

Hippocratic Coding:-

☞ **Aspire to code it right the first time.**

Patterns:

☞ **Master stylized code patterns, and use them.**

Analysis:

☞ **Analyze first.**

Process:

☞ **Mitigate errors.**

Process: Don’t make mistakes

- Hope for the best, but
- Plan for the worst.

☞ **Avoid debugging like the plague.**

Process: Find mistakes as soon as possible

☞ **Test programs incrementally.**

Process: Stay in control

- Define relevant subproblems that can be tested.
- Preserve end-to-end correctness

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: Subproblem end-to-end correctness

```
/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */
/* Compute a direct path through the maze, if one exists. */
/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */
```

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: Subproblem end-to-end correctness

```
~~/* Input a maze of arbitrary size, or output “malformed input” and stop if the
   input is improper. Input format: TBD. */~~
/* Compute a direct path through the maze, if one exists. */
~~/* Output the direct path found, or “unreachable” if there is none. Output
   format: TBD. */~~
```

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Process: Subproblem end-to-end correctness

jury rig a specific maze

```
/* Compute a direct path through the maze, if one exists. */
```

provide simple diagnostic output

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Process: Subproblem end-to-end correctness

jury rig a specific maze

```
/* Compute a direct path through the maze, if one exists. */
```

provide simple diagnostic output

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Process: Undo if necessary

☞ **Don’t be wedded to code. Revise and rewrite when you discover a better way.**

Process: Stepwise refinement

☞ **Program top-down, outside-in.**

Example: Print the integer part of the square root of an integer `n≥0`.

Example: Print the integer part of the square root of an integer `n≥0`.

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

```
/* Given n≥0, output the Integer Square Root of n. */
```

Example: Print the integer part of the square root of an integer `n≥0`.

☞ **Write comments as an integral part of the coding process, not as afterthoughts.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
```

Example

```
/* Given n≥0, output the Integer Square Root of n. */          1
```

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
```
1

Where did n come from?

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */    1
```

Where did n come from?

- It is a program variable

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */     1
```

Where did n come from?

- It is a program variable
- It is assumed to already contain a value ≥0

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */    1
```

Where did n come from?

- It is a program variable
- It is assumed to already contain a value ≥0
- We are asked to write a program segment

☞ **Make sure you understand the problem.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */    1
```

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
```
1

Can’t we just do this using a few library routines?

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
   System.out.println( Math.floor( Math.sqrt(n) ) );
```

2

Can't we just do this using a few library routines?

- Yes.

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
   System.out.println( Math.floor( Math.sqrt(n) ) );
```
2

Can't we just do this using a few library routines?

- Yes.
- But that would deprive us of a good example.
- So, we amend our problem statement.

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
   System.out.println( Math.floor( Math.sqrt(n) ) );
```

2

Example: Print the integer part of the square root of an integer n≥0 without using built-in functions.

Can't we just do this using a few library routines?

- Yes.
- But that would deprive us of a good example.
- So, we amend our problem statement.

☞ **Make sure you understand the problem.**

```
/* Given n≥0, output the Integer Square Root of n. */
```

Example: Print the integer part of the square root of an integer $n \geq 0$ without using built-in functions.

Example

```
/* Given n≥0, output the Integer Square Root of n. */
```

☞ **Master stylized code patterns, and use them.**

```
/* Given n≥0, output the Integer Square Root of n. */
```
1

☞ **Master stylized code patterns, and use them.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
```

1

```
/* Compute. */
/* Use. */
```

☞ **Master stylized code patterns, and use them.**

```
/* Given n≥0, output the Integer Square Root of n. */
```

1

```
/* Compute r. */
/* Use r. */
```

☞ **Specify how individual program steps will cooperate with one another.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

```
/* Compute r. */
/* Use r. */
```

☞ **Specify how individual program steps will cooperate with one another.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

☞ **Master stylized code patterns, and use them.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

☞ **If you "smell a loop", write it down.**

Example

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

☞ **Decide first whether an iteration is indeterminate (use while) or determinate (use for).**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

```
/* Indeterminate iteration */
   int k = start;
   while ( condition ) k++;
```

Which?

```
/* Determinate iteration */
   for (int k=start; k<=limit; k++) compute
```

☞ **Decide first whether an iteration is indeterminate (use while) or determinate (use for).**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

```
/* Indeterminate iteration */
   int k = start;
   while ( condition ) k++;
```

Which?

```
/* Determinate iteration */
   for (int k=start; k<=limit; k++) compute
```

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
   System.out.println( r );
```

2

```
/* Indeterminate iteration */
   int k = start;
   while ( condition ) k++;
```

```
/* Determinate iteration */
   for (int k=start; k<=limit; k++) compute
```

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

```
/* Indeterminate iteration */
   int k = start;
   while ( condition ) k++;
```

☞ **Beware of for-loop abuse; if in doubt, err in favor of while.**

Example

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

☞ **There is no shame in reasoning with concrete examples.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

r	r*r	n
0	0	0

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

☞ **Elaborate the expected input/output mapping explicitly.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

3

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

4

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```


r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?
- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?
- 9, 10, 11, …

What is special about 9?
- It is the square of 3.

But what is special about 3?
- It is one more than 2, the value of r.

☞ **Elaborate the expected input/output mapping explicitly.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( condition ) r++;
   System.out.println( r );
```

4

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

But what is special about 3?

- It is one more than 2, the value of r.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */                    4
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

r	r*r	n
0	0	0
1	1	1, 2, 3
2	4	4, 5, 6, 7, 8
3	9	9, 10, 11, 12, 13, 14, 15

When r=2, for which n do we **stop**?

- 4, 5, 6, 7, or 8.

When r=2, for which n do we **continue**?

- 9, 10, 11, …

What is special about 9?

- It is the square of 3.

But what is special about 3?

- It is one more than 2, the value of r.

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

4

Example

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

4

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

4

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```
4

Elaborate and eliminate choices

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

4

Elaborate and eliminate choices for the relation

==, != No. Given r, must be true for many n, and false for many n.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```
4

Elaborate and eliminate choices for the relation

==, != No. Given r, must be true for many n, and false for many n.

>, >= No. Must keep going for little r and big n.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */                    4
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.
<	No. Must keep going for "equal n" case.

☞ **Alternate between concrete reasoning and abstract reasoning.**

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) ____ n ) r++;
   System.out.println( r );
```

4

Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.
<	No. Must keep going for "equal n" case.
<=	Yes.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) <= n ) r++;
   System.out.println( r );
```

5

Elaborate and eliminate choices for the relation

==, !=	No. Given r, must be true for many n, and false for many n.
>, >=	No. Must keep going for little r and big n.
<	No. Must keep going for "equal n" case.
<=	Yes.

☞ **Alternate between concrete reasoning and abstract reasoning.**

Example

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) <= n ) r++;
   System.out.println( r );
```

5

```
/* Given n≥0, output the Integer Square Root of n. */
   /* Let r be the integer part of the square root of n≥0. */
      int r = 0;
      while ( (r+1)*(r+1) <= n ) r++;
   System.out.println( r );
```

5

Pragmatics

- For a full program, we need
  - Get a value for n
  - A receptacle for our code (`boilerplate`)
  - A mechanism for input (`Scanner`)
  - A named operation to invoke (`main`)

```
/* Output the Integer Square Root of an integer input. */
   /* Obtain an integer n≥0 from the user. */
   /* Given n≥0, output the Integer Square Root of n. */
      /* Let r be the integer part of the square root of n≥0. */
         int r = 0;
         while ( (r+1)*(r+1) <= n ) r++;
      System.out.println( r );
```

6

Pragmatics

- For a full program, we need
  - Get a value for n
  - A receptacle for our code (`boilerplate`)
  - A mechanism for input (`Scanner`)
  - A named operation to invoke (`main`)

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

Pragmatics

- For a full program, we need
  - Get a value for n
  - A receptacle for our code (boilerplate and main)
  - A mechanism for input (Scanner)
  - A named operation to invoke (main)

Pragmatics

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

7

Pragmatics

- For a full program, we need
  - Get a value for n
  - A receptacle for our code (boilerplate and main)
  - A mechanism for input (Scanner)
  - A named operation to invoke (main)

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

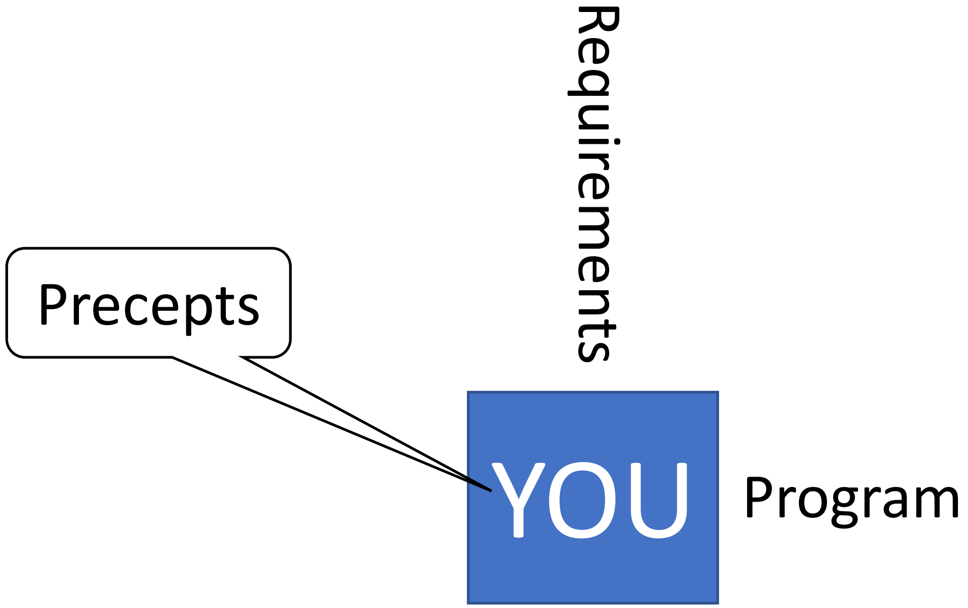

Pragmatics

- For a full program, we need
  - Get a value for n
  - A receptacle for our code (`boilerplate` and `main`)
  - A mechanism for input (`Scanner`)
  - A named operation to invoke (`main`)

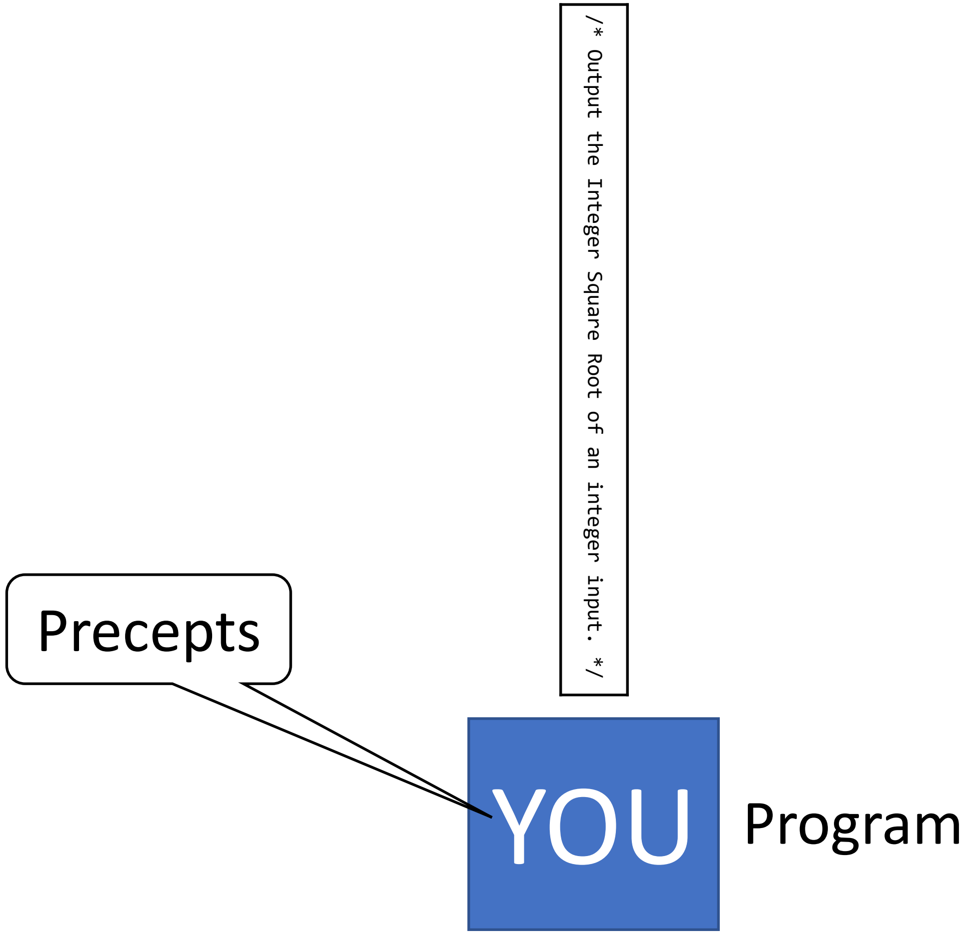
```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8

Pragmatics



Pragmatics


/* Output the Integer Square Root of an integer input. */
Precepts
YOU
Program

Pragmatics

/* Output the Integer Square Root of an integer input. */

Precepts

YOU

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8

Pragmatics

```
/* Output the Integer Square Root of an integer input. */
```

Precepts

YOU

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8

Pragmatics

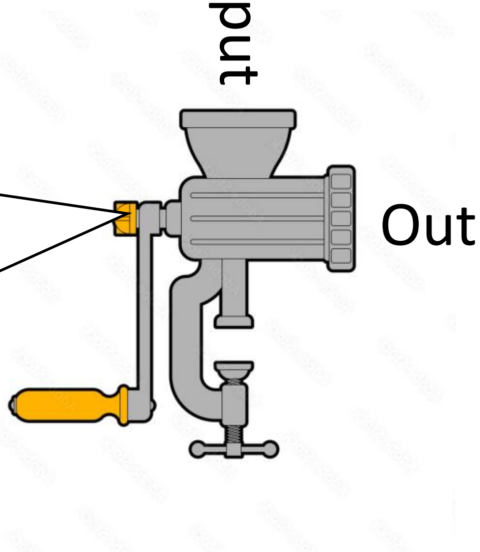
Precepts

YOU

```
/* Output the Integer Square Root of an integer input. */
```

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8



Pragmatics

Precepts

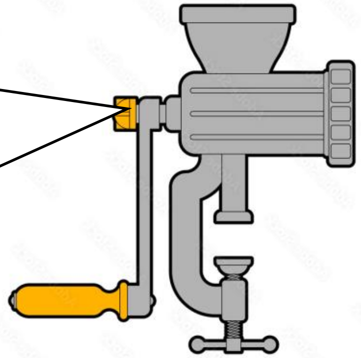
YOU

```
/* Output the Integer Square Root of an integer input. */
```

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8

7



Output

Pragmatics

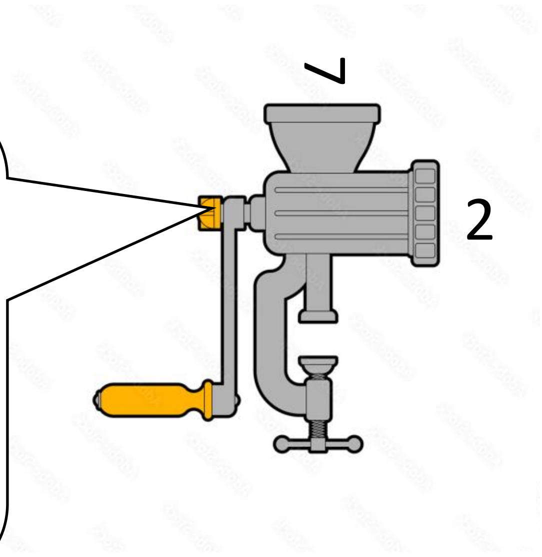
Precepts

YOU

/* Output the Integer Square Root of an integer input. */

```
import java.util.Scanner;
class boilerplate {
   static Scanner in = new Scanner(System.in);
   static void main() {
      /* Output the Integer Square Root of an integer input. */
         /* Obtain an integer n≥0 from the user. */
            int n = in.nextInt();
         /* Given n≥0, output the Integer Square Root of n. */
            /* Let r be the integer part of the square root of n≥0. */
               int r = 0;
               while ( (r+1)*(r+1) <= n ) r++;
            System.out.println( r );
      } /* main */
   } /* boilerplate */
```

8



Summary

Goals

Elements of methodology

- Precepts, Patterns, Analysis, Process

Core programming-language constructs

- (almost all that we will need)

Illustrated the approach with a complete example