Introduction to Computational Thinking

Lecture #4:

Data Conversions; Assignment Operators; for Loops

Qiao Xiang, Qingyu Song

https://sngroup.org.cn/courses/ct-xmuf25/index.shtml

10/22/2025

This deck of slides are heavily based on cs112 at Yale University and cs101 at UCAS, respectively, by courtesy of Dr. Y. Richard Yang and Dr. Zhiwei Xu.

Recap: Data Types

- Why data types?
  - Define (1) data representation/storage, (2) allowed operations, and (3) semantics of operations (e.g., 1 / 2 vs 1.0 / 2.0)
- Java is a strong typed language: every literal, every variable, every operation has a type

```
int nA;
nA = 4;
int nB = 1;
int total = nA * 4 + nB * 3;
System.out.println( total / (nA + nB) );
double GPA = 3.0 + 0.8;
char lastNameInitial = 'Y';
```

Variable Assignments

- A variable can't be used until it is assigned a value.
  - ```
    int x;
    System.out.println(x);    // ERROR: x has no value
    ```
- You may not declare the same variable twice.
  - ```
    int x;
    int x;              // ERROR: x already exists
    ```
  - ```
    int x = 3;          // OK: declare and initialize
    int x = 5;          // ERROR: x already exists
    x = 5;              // this is OK
    ```
    ```

Operator Precedence Rules

- Generally operators evaluate left-to-right.
  `1 - 2 - 3` **is** `(1 - 2) - 3` **which is** `-4`

- But `* / %` have a higher level of precedence than `+ -`
  `1 - 3 * 4` **is** `-11`

- Parentheses can force a certain order of evaluation:
  `(1 + 3) * 4` **is** `16`

- Spacing does not affect order of evaluation
  `1+3 * 4-2` **is** `11`

Precedence: Examples

- What is the order of evaluation in the following expressions?

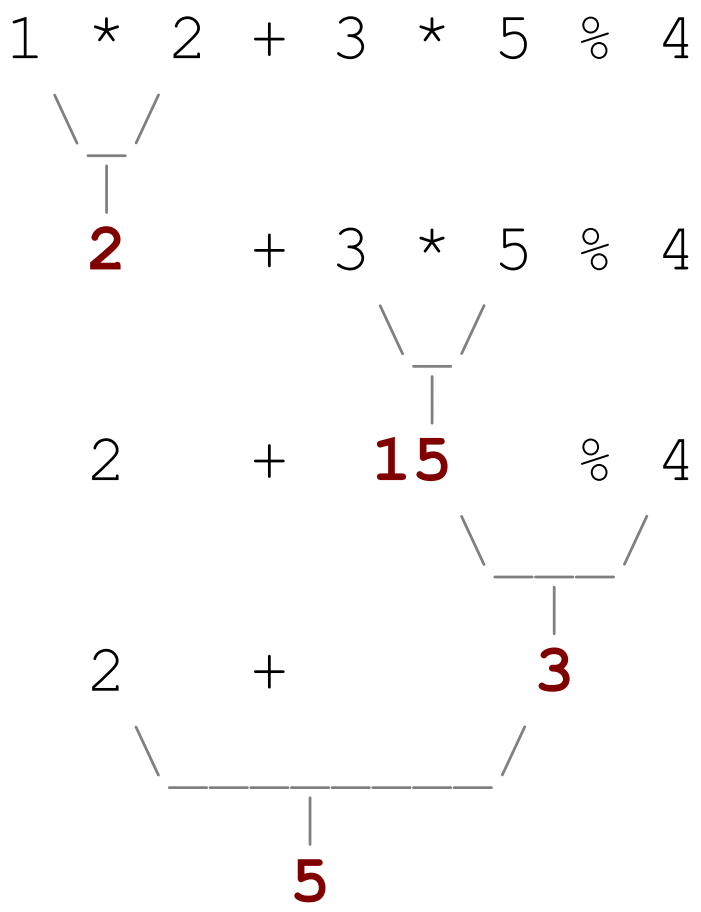
```
a + b + c + d + e
  1   2   3   4
```

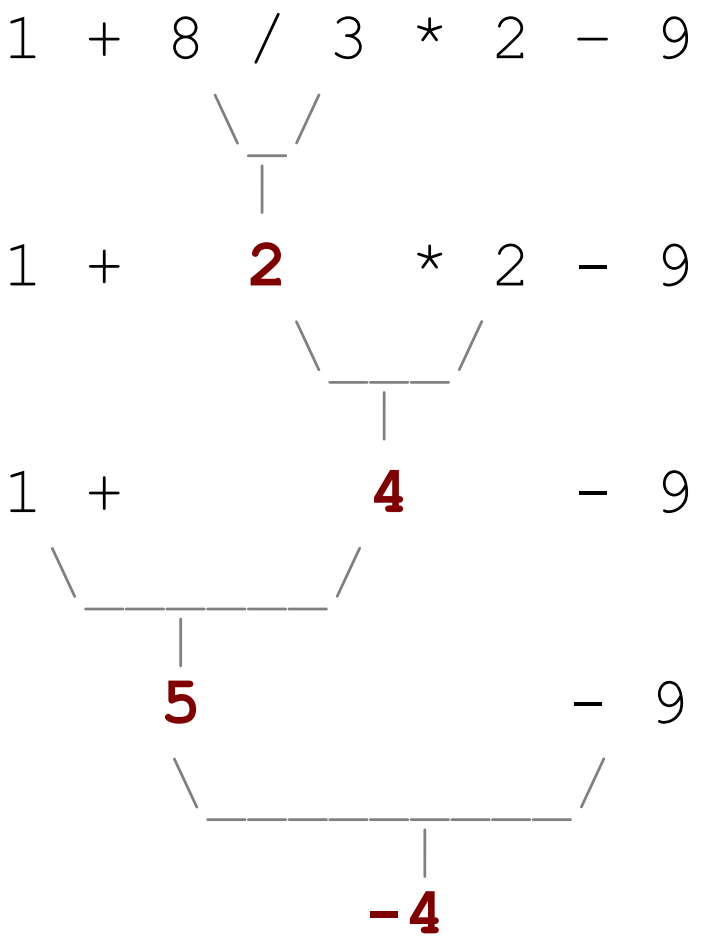
```
a + b * c - d / e
  3   1   4   2
```

```
a / (b + c) - d % e
  2    1     4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Precedence: Examples (Offline)

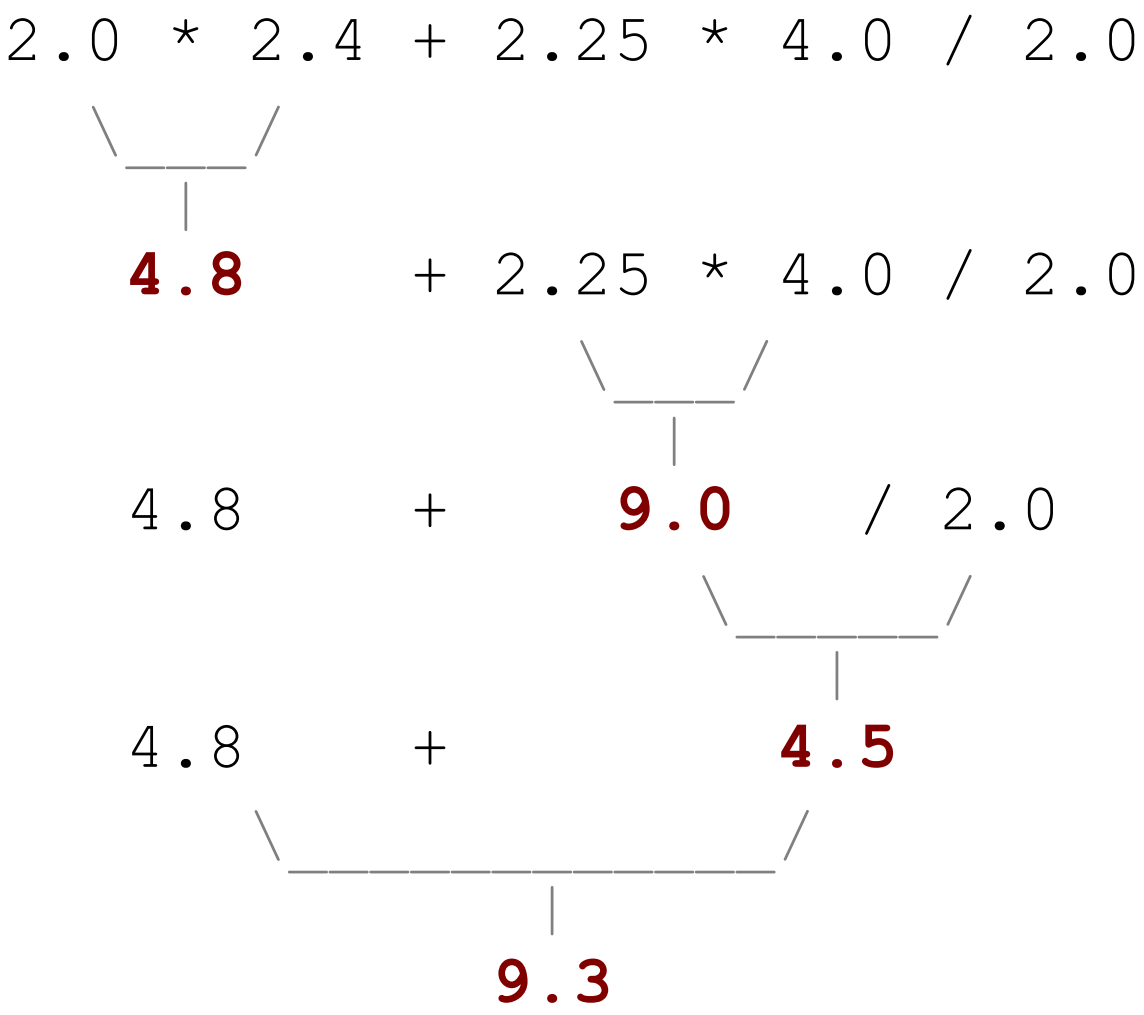




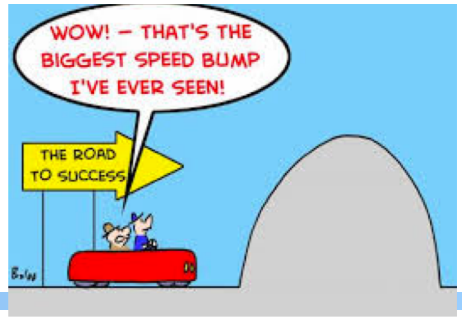
Precedence Questions (Offline)

- What values result from the following expressions?
  - `9 / 5`
  - `695 % 20`
  - `7 + 6 * 5`
  - `7 * 6 + 5`
  - `248 % 100 / 5`
  - `6 * 3 - 9 / 4`
  - `(5 - 7) * 4`
  - `6 + (18 % (17 - 12))`

Real Number Example (Offline)



Recap: Type System Issue




```
int nA = 4;
int nB = 1;
int total = nA * 4 + nB * 3;
System.out.println( total / (nA + nB) );
```

Integer result, not what we want

Not intuitive type; 4 is int?

Which * operation?

```
float nA = 4;
float nB = 1;
System.out.println( (nA * 4 + nB * 3)
                    /
                   (nA + nB) );
```

Outline

- Admin and recap
- Primitive data types
  - why data types?
  - storage and representation
  - operations
  - expressions
  - *data conversions*

Issue

- Sometimes it is more efficient and natural to represent data as one type, but during a computation, we may want to get *desired* result in a different type
  - e.g., raw grade points and # of grades as integers, but GPA as double

- Sometimes we just write mixed-type expressions
  - 4.0 / 8

Data Conversion

- Data conversion is the conversion of data from one type to another, e.g.,
  - an `int` -> a `double`,
  - a `double` -> an `int`,
  - an `int` -> a `string`

- Java data conversion is per-operator, occurring when the operator is evaluated according to the precedence rule
- Java has two types of data conversion
  - Implicit (automatic/predefined) data conversion
  - Explicit (cast) data conversion

(Implicit) Predefined Data Conversion

- Seeing a mixed operation, Java tries a set of predefined data conversion rules
  - If successful, you get the results
  - If not, you get a compiler error
- Discussion: How may you define a conversion rule?
  - E.g., 4.0 / 8

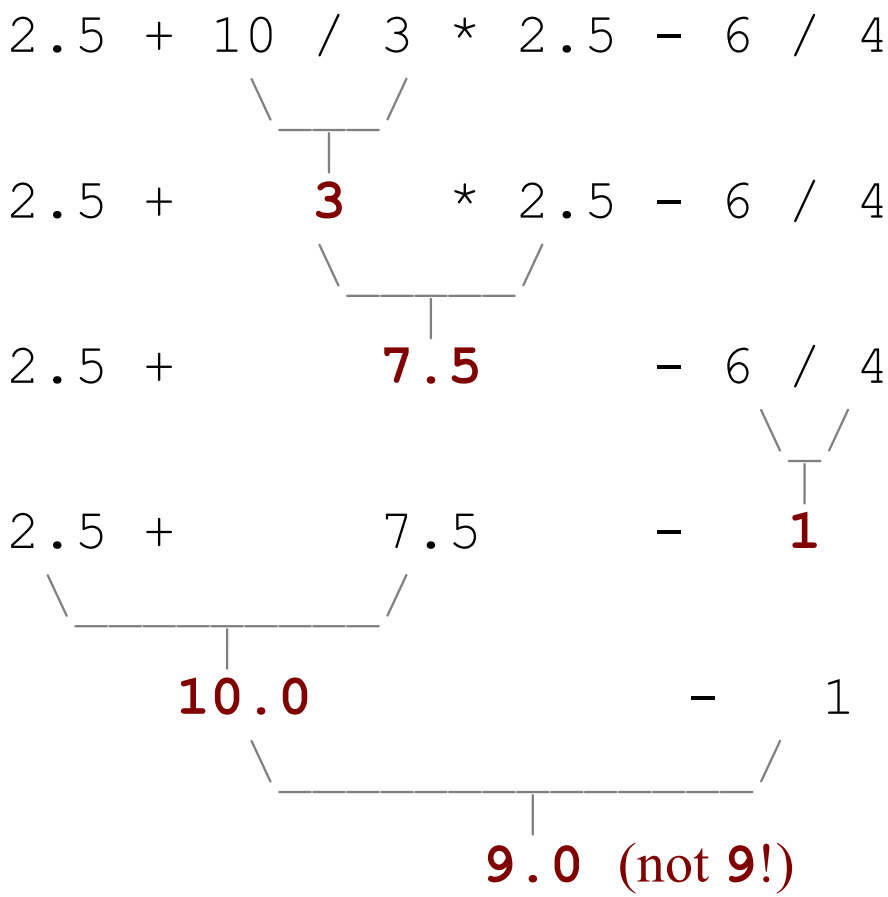
Predefined Data Conversion Rule: Arithmetic (numeric) Promotion

- Occurs *automatically* when the operands of a binary arithmetic operator are of different types
  - if either operand is `double`, the other is converted to `double`
  - otherwise, if either operand is `float`, the other is converted to `float`
  - otherwise, if either operand is `long`, the other is converted to `long`
  - otherwise, **both operands are converted to `int`**

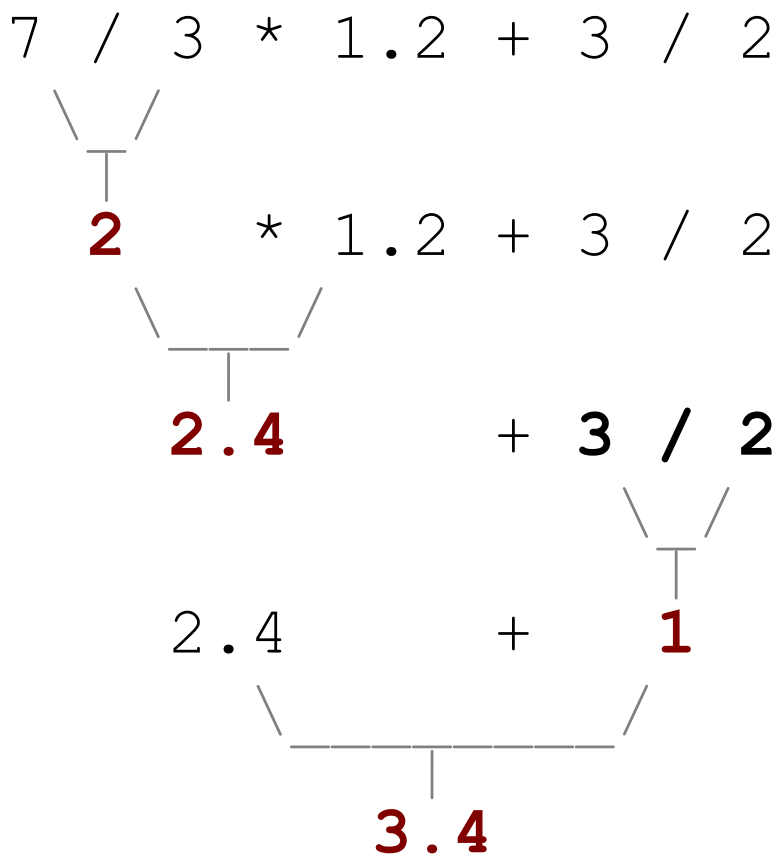
Examples:

- 4.0 / 8 (which / is it: /double, /float, /int)
- 4 / 8.0 (which / is it: /double, /float, /int)
- 4 / 8 (which / is it: /double, /float, /int)

Example: Mixed Arithmetic Expression



Practice: Mixed Arithmetic Expression (Offline)



Example: Mixed Assignment

- Automatic mixed type assignment allowed only if allowed by automatic numeric promotion
  - ```
    int x;
    x = 2.5;                    // ERROR: incompatible types
    ```

```
double myGPA = 4;
```

myGPA	4.0

```
double avg = 11 / 2;
```

avg	5.0
```

Predefined Data Conversion Rule: Numeric to Java String

- Occurs *automatically* when one operand is a number and the other a string in the "+" operator
- The conversion is per-operator, affecting only its operands.
- This produces the convenient string concatenation operation.

Java String Concatenation Conversion: Examples

```
1 + "abc" + 2       is   "1abc2"
"abc" + 1 + 2       is   "abc12"
1 + 2 + "abc"       is   "3abc"
"abc" + 9 * 3 +1    is   "abc271"
4 - 1 + "abc"       is   "3abc"
```

Examples

- See IntOps.java
- Fix the GPA.java program

User Forced (Explicit) Conversion: Type Casting

- **type cast**: An explicit, FORCED conversion from one type to another.
- Syntax:

  (type) expression

- Type casting has high precedence and casts only the item immediately next to it.

- You can cast either up (promotion) or down (truncate)

Type Casting Examples

```
double result = (double) 19 /5;    // 3.8

int result2 = (int) result;        // 3

double x = (double) 1 + 1 / 2;     // 1.0

double y = 1 + (double) 1 / 2;     // 1.5
```

Outline

- Admin and recap
- Primitive data types
  - why data types?
  - storage and representation
  - operations
  - expressions
  - data conversions
  - *assignment as operator*

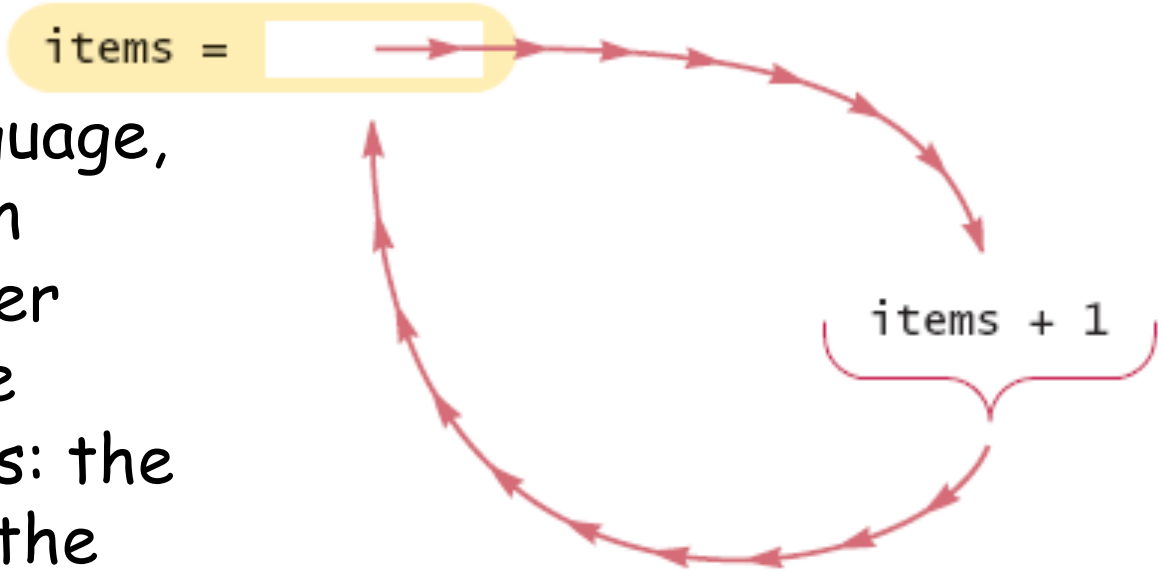
Update vs. Algebra

- What happens here?

```
int items = 3;
items = items + 1;   // ???
```

items	4

```
items =  items + 1;
```



In programming language, assignment is just an operator, with a lower precedence than the arithmetic operators: the right is assigned to the left var.

Example: Ruler

```
public class Ruler {
   public static void main(String[] args) {
      String ruler = "1";
      ruler = ruler + " 2 " + ruler;
      ruler = ruler + " 3 " + ruler;
      ruler = ruler + " 4 " + ruler;
      System.out.println(ruler);
   }
}
```

```
      "1"
   "1 2 1"
"1 2 1 3 1 2 1"
```

```
% java Ruler
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

Update Shorthand

- Since increment updates are common, Java introduces shorthand:

```
count =  count + increment;
```

```
count += increment;
```

when increment is 1

```
count ++;
```

These expressions have the same effect

```
count  =  count + 1;
count  += 1;
count  ++;
```

Modify-and-assign

shortcuts to modify a variable's value

Shorthand	Equivalent longer version
variable ++;	**variable = variable + 1;**
variable --;	**variable = variable - 1;**
variable += value;	**variable = variable + (value);**
variable -= value;	**variable = variable - (value);**
variable *= value;	**variable = variable * (value);**
variable /= value;	**variable = variable / (value);**
variable %= value;	**variable = variable % (value);**

```
int x = 2;
double gpa = 3.8;

x += 3;                  // x = x + (3) -> 5;
gpa --;                  // gpa = gpa - 1.0 -> 2.8;

x *= 2;                  // x = x * 2 -> 10;
x *= 2 + 1;              // x = x * (2+1) -> 30;
```

General: Assignment/Modify-and-Assign as Operators

- Assignment operators in a complex expression

First the expression on the right hand side of the += operator is evaluated

```
answer  +=  sum / 4 + MAX * lowest;
```

4 1 3 2

Then the result is used to calculate in the variable on the left hand side.

(Offline) Practice

- Compile the list of operators that we covered and their precedence levels

Example: SavingsSuccess

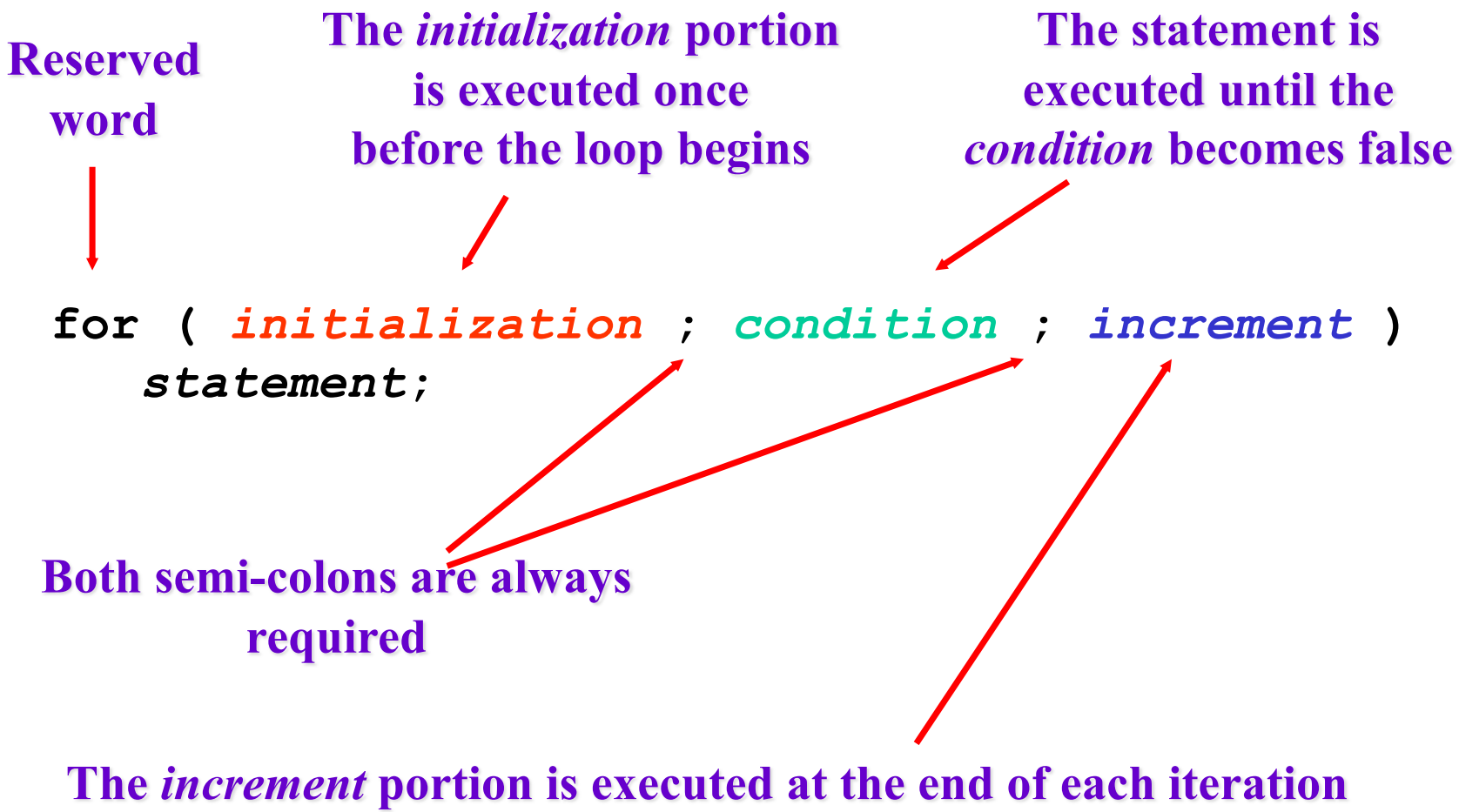
- What is the result of adding $1000 on Jan. 1 of each year to a fund that returns 6% per year, for 30 years?

http://www.1stock1.com/1stock1_141.htm

Outline

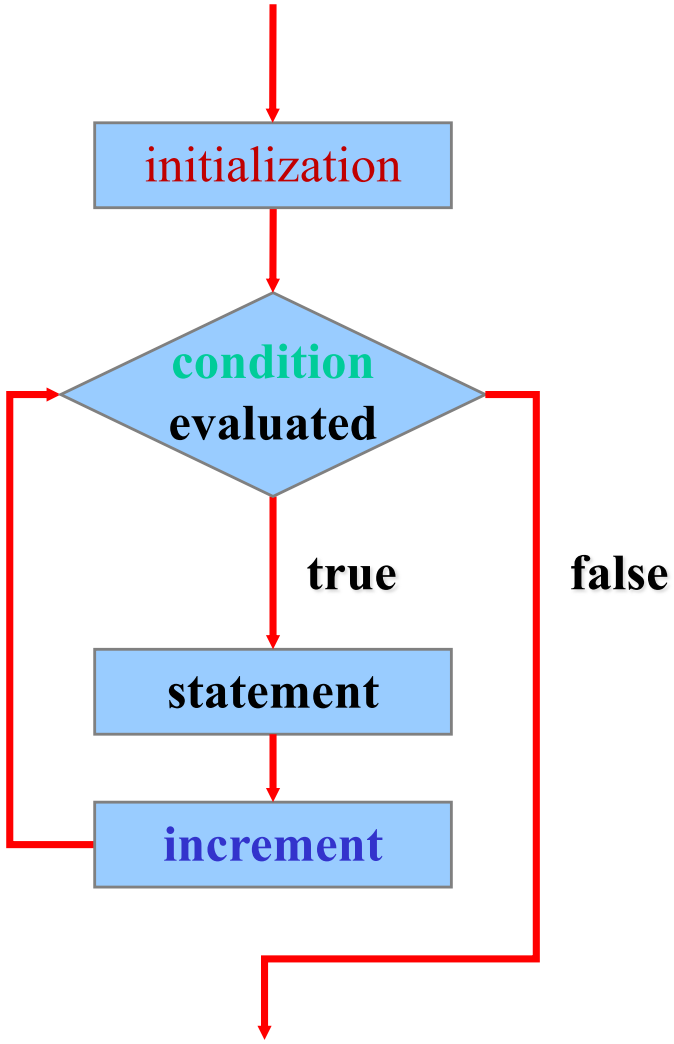
- Admin and recap
- Primitive data types
- *for Loops*

The `for` Statement: Syntax



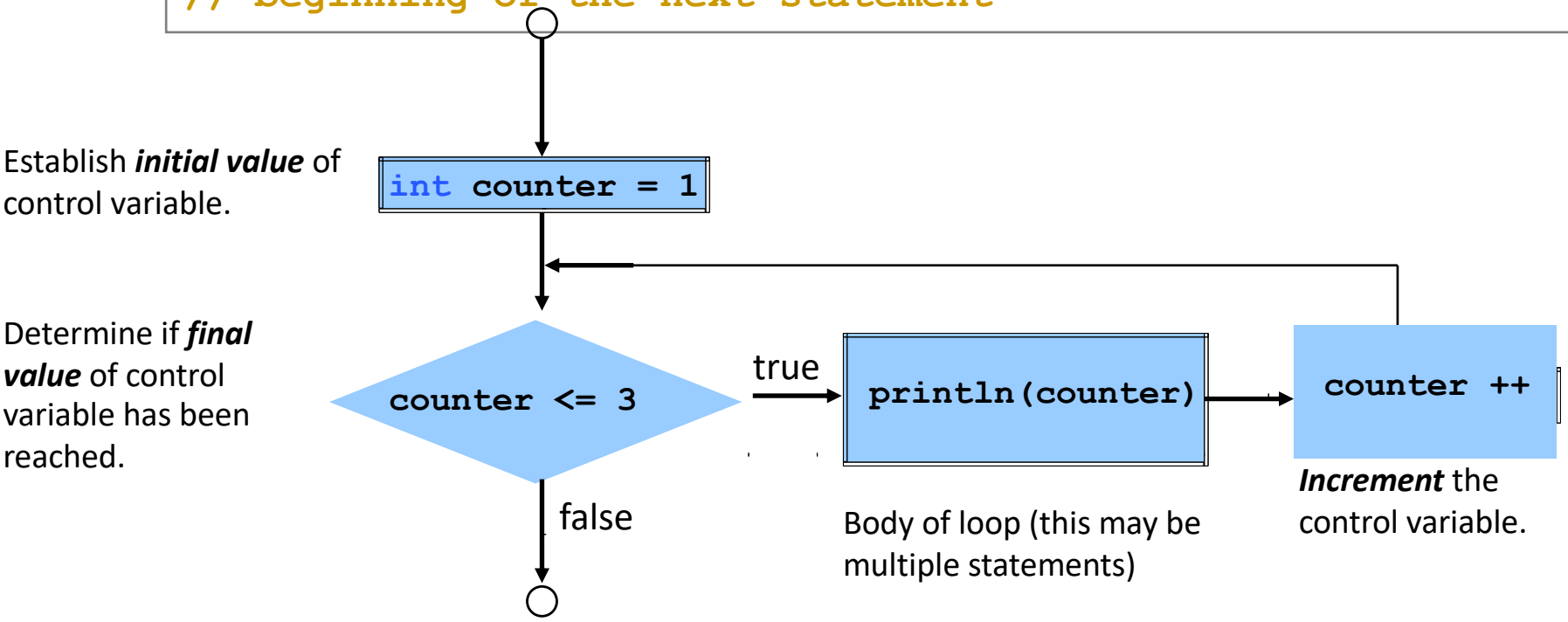
Flowchart of a `for` loop

```
for ( initialization ; condition ; increment )
    statement;
```



The `for` Statement: Example

```
for (int counter = 1; counter <= 3; counter ++)
{
  System.out.println ( counter );
}
// beginning of the next statement
```



Flexibility of for Loop with Counter

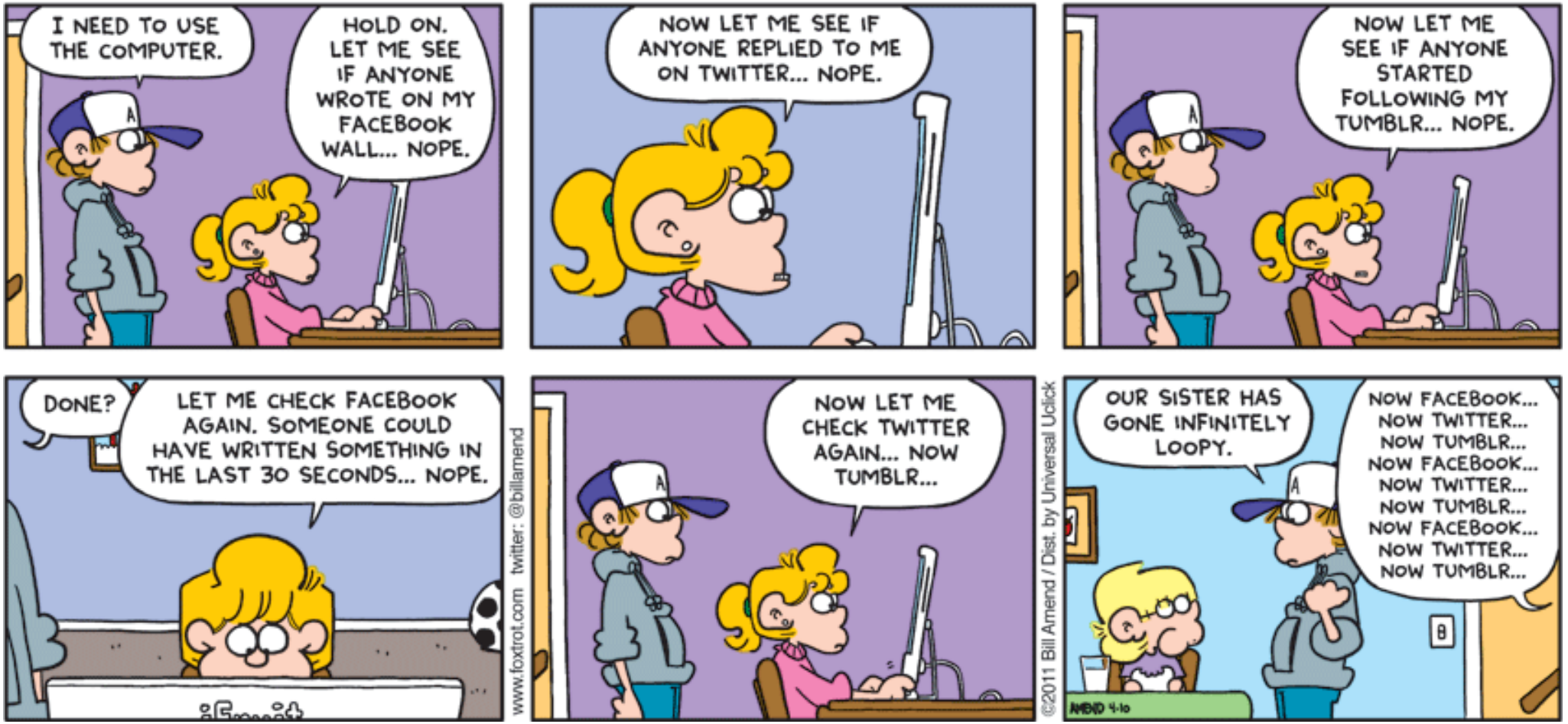
Loop counter:
- can use any name, not just i
- can start at any value, not just 1
- only valid in the loop

Compare loop counter with target:
- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to

Can increment, decrement, times, …

```
for (int i = 1; i <= 6; i ++)
{
    System.out.println("I am so smart");
}
```

Using `for` Loops?

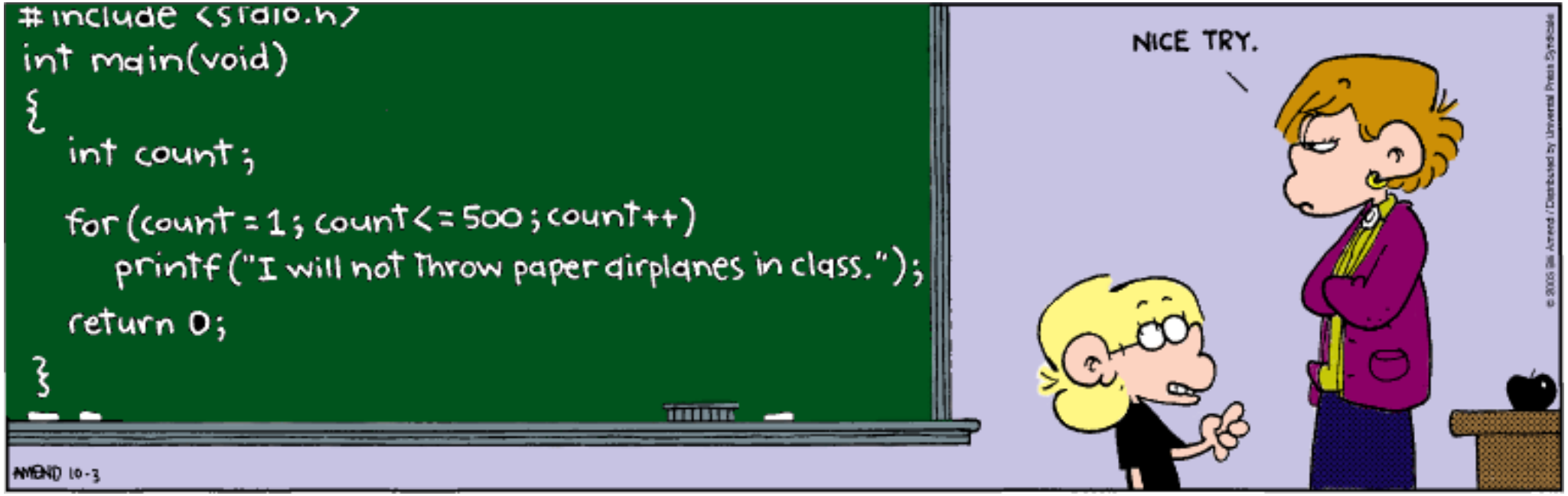


Using `for` Loops

```
for (int i = 1; i <= 3; i++)
{
   System.out.println("Now Facebook");
   System.out.println("Now Twitter");
   System.out.println("Now Tumblr");
}
```

Using `for` Loops

- Java's **`for` loop** statement performs a task many times.



Using `for` Loop: Print Result from 2017-2030

- What is the result of adding $1000 on Jan. 1 of each year to a fund that returns 6% per year?

http://www.1stock1.com/1stock1_141.htm

Using for Loop: Counting Down

- Write a program generating output

```
T-minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!
```

- Requirement: loop counter starts with 10 and counts down

Counting Down v1

- The **update** uses `--` to count down.

```
System.out.print("T-minus ");
for (int i = 10; i >= 1; i--) {
      System.out.print(i + ", ");
}
System.out.println("blastoff!");
```

Counting Down v2

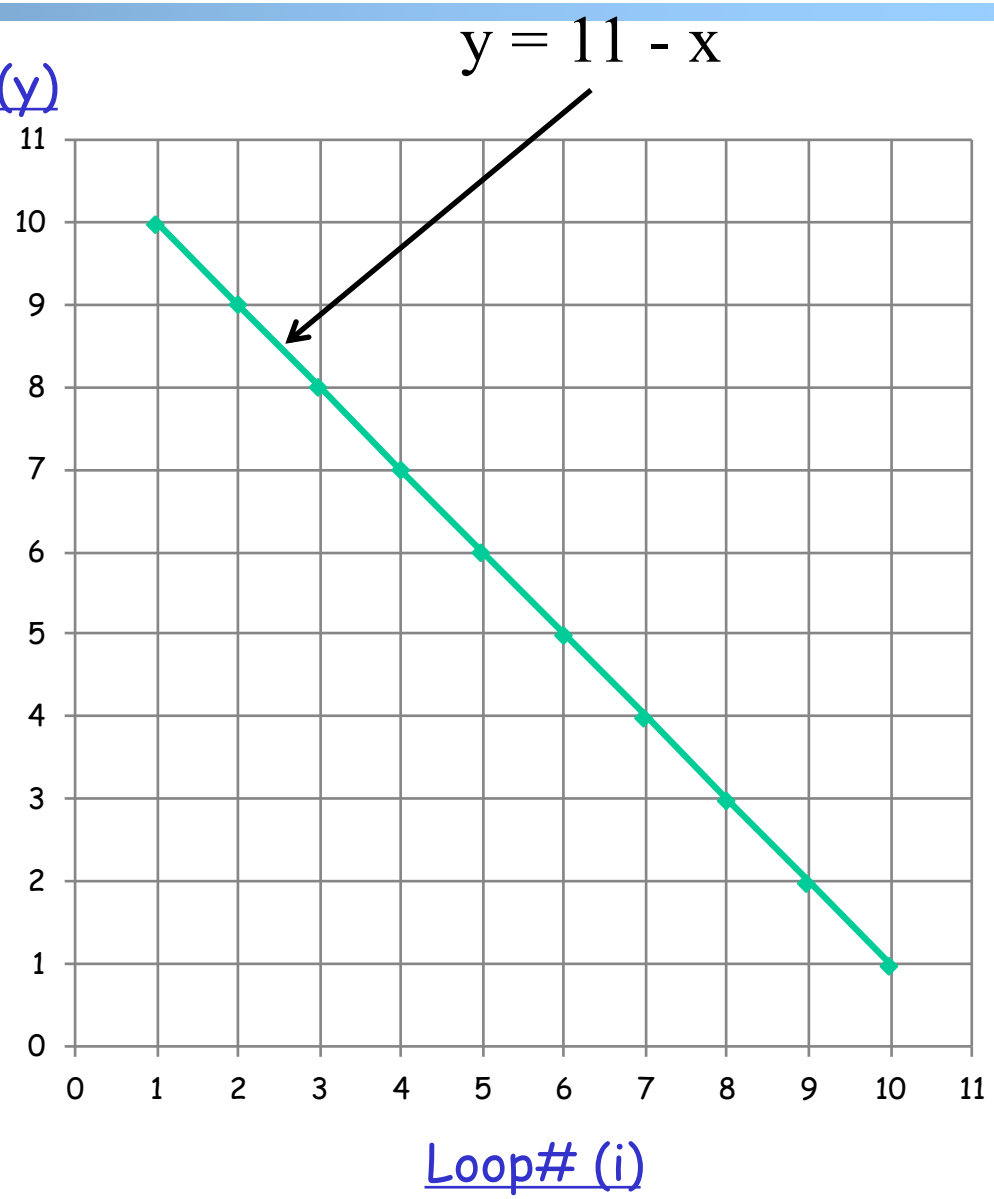
- Requirement: loop counter starts with 1 and counts up:

```
T-minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!
```

```
System.out.print("T-minus ");
for (int i = 1; i <= 10; i++) {
      // ???
}
System.out.println("blastoff!");
```

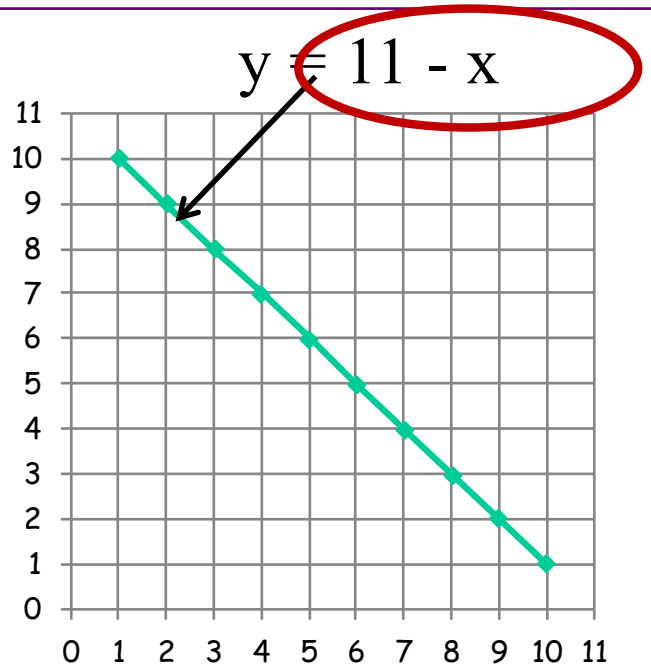
Mapping Loop# to Target Pattern

i	number to print
1	10
2	9
3	8
4	7
5	6
…	…

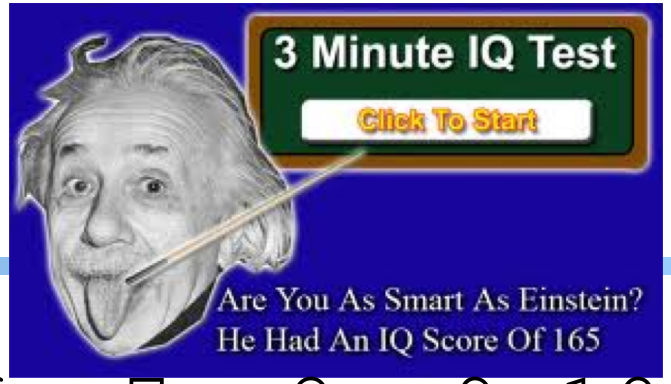


Counting Down

```
System.out.print("T-minus ");
for (int i = 1; i <= 10; i++) {
        System.out.println(11 - i + ", ");
}
System.out.println("blastoff!");
```



An "IQ Test" Format

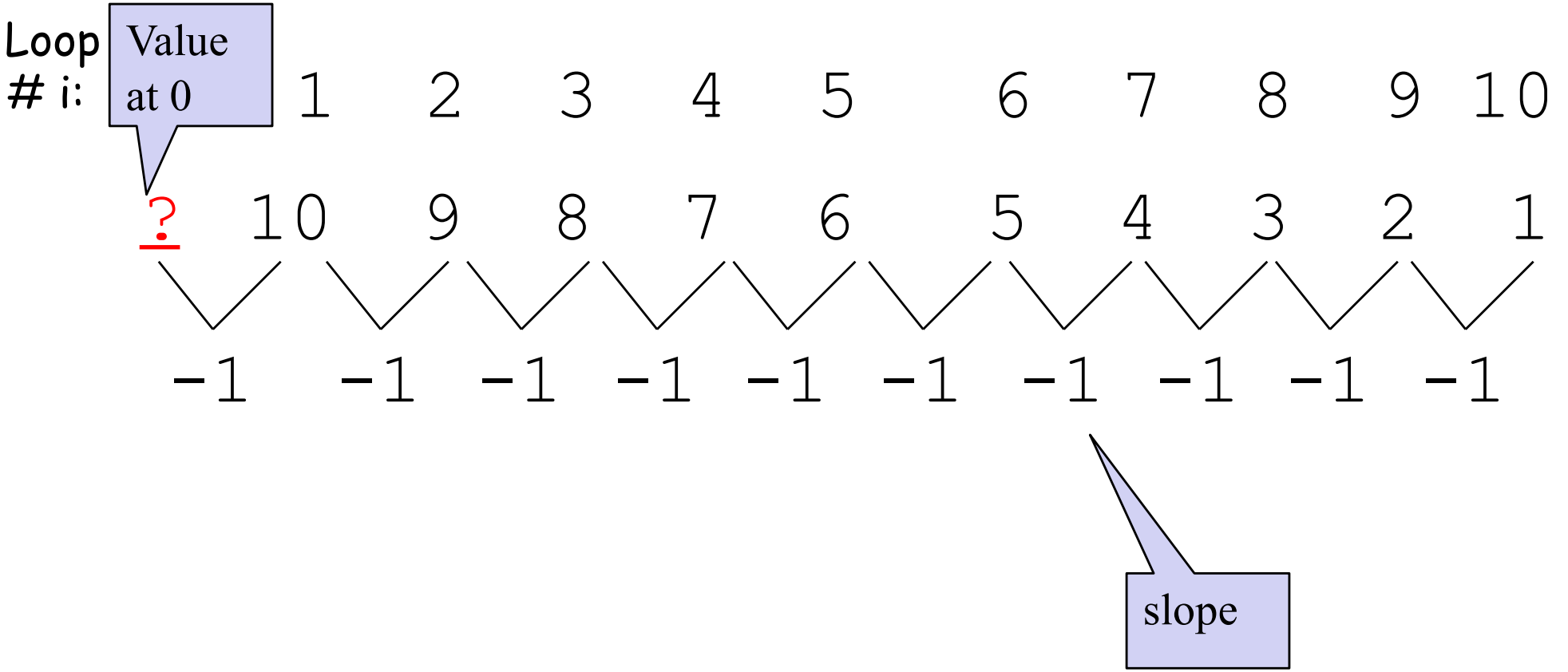


Loop
i:

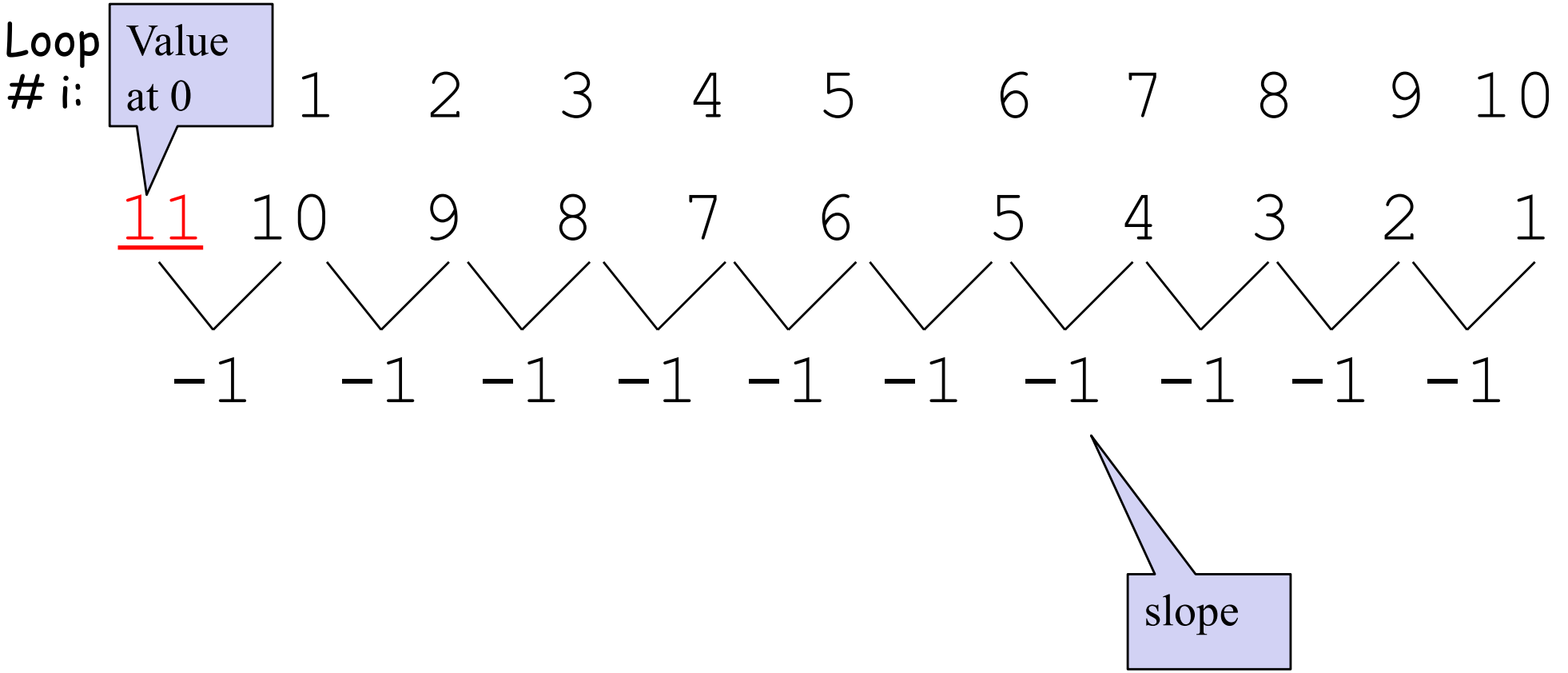
1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1
-1	-1	-1	-1	-1	-1	-1	-1	-1	

slope

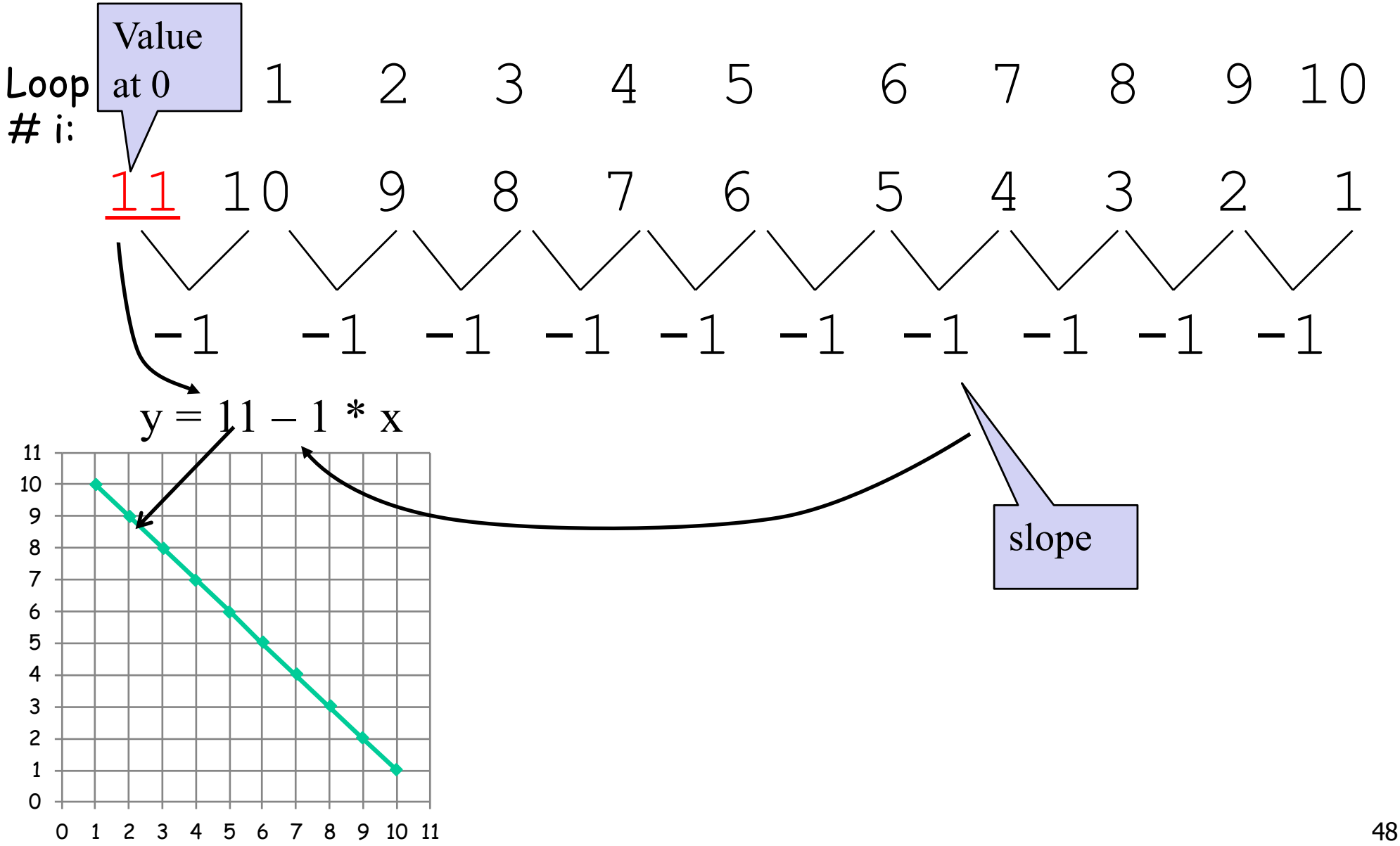
An "IQ Test" Format



An "IQ Test" Format



An "IQ Test" Format



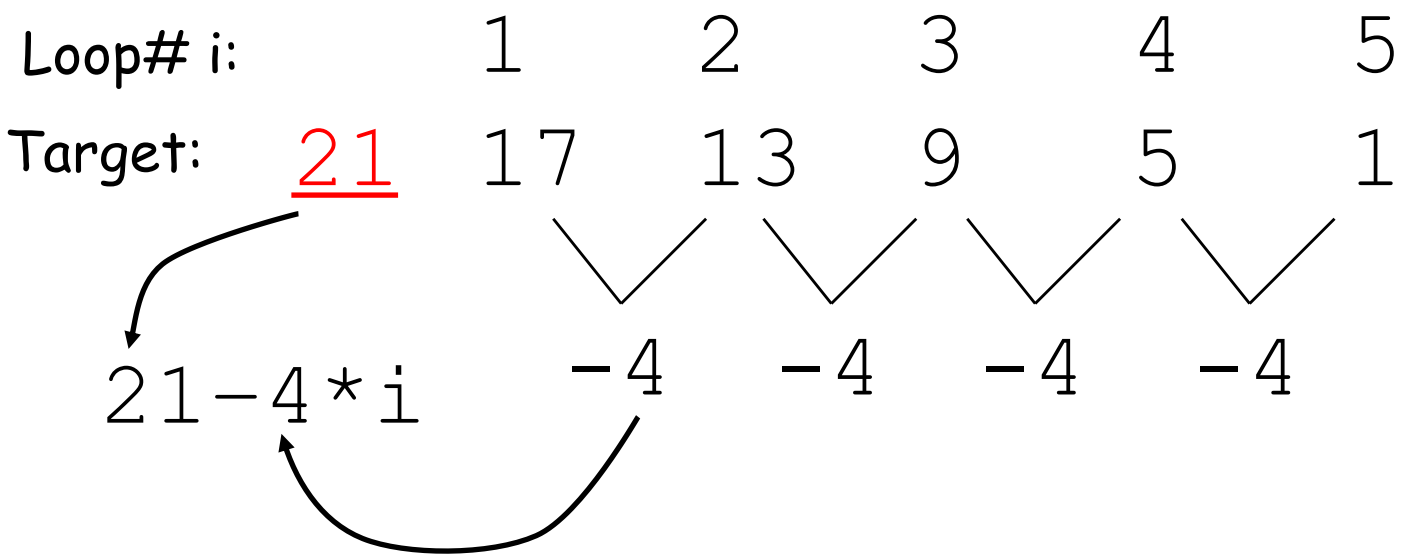
Practice: Mapping loop# to numbers

```
for (int count = 1; count <= 5; count++) {
    System.out.print( ... );
}
```

- What statement in the body would cause the loop to print:

  `17 13 9 5 1`

Mapping loop# to numbers




```
for (int count = 1; count <= 5; count++) {
    System.out.print(-4 * count + 21 + " ");
}
```

(Offline) Practice: Mapping loop# to numbers

```
for (int count = 1; count <= 5; count++) {
    System.out.print( ... );
}
```

- What statement in the body would cause the loop to print:

  4 7 10 13 16

```
for (int count = 1; count <= 5; count++) {
    System.out.print(3 * count + 1 + " ");
}
```

(Offline) Practice: Mapping loop# to numbers

```
for (int count = 1; count <= 5; count++) {
    System.out.print( ... );
}
```

- What statement in the body would cause the loop to print:

  2 7 12 17 22

```
for (int count = 1; count <= 5; count++) {
    System.out.print(5 * count - 3 + " ");
}
```

Counting Down v2a

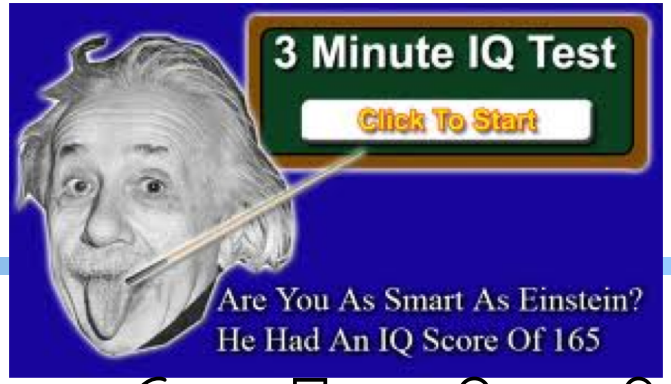
- Requirement: loop counter starts with 0 and counts up:

```
T-minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!
```

```
System.out.print("T-minus ");
for (int i = 0; i __ 10; i++) {
   System.out.println(_____ + ", ");

}
System.out.println("blastoff!");
```

An "IQ Test" Format



Loop # i:

0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1
	-1	-1	-1	-1	-1	-1	-1	-1	-1

Already knows value at 0

Summary: Code Style

Counting down or up, starting w/ 0 or 1 is mostly a personal style.

```
System.out.print("T-minus ");
for (int i = 10; i >= 1; i--) {
   System.out.print(i + “, “);
}
System.out.println("blastoff!");
```

```
System.out.print("T-minus ");
for (int i = 1; i <= 10; i++) {
   System.out.print(11-i + “, “);
}
System.out.println("blastoff!");
```

```
System.out.print("T-minus ");
for (int i = 0; i < 10; i++) {
   System.out.print(10-i + “, “);
}
System.out.println("blastoff!");
```

Counting Down: v3

- If I want to count down from 12, what should we change?
  - T-minus 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!


```
System.out.print("T-minus ");
for (int i = 1; i <= 10; i++) {
    System.out.print(11-i + ", ");
}
System.out.println("blastoff!");
```

Counting Down: v3

- Problem: The code has two "magic" numbers 11 and 10, but they are not independent

```
System.out.print("T-minus ");
for (int i = 1; i <= 10; i++) {
    System.out.print(11-i + ", ");
}
System.out.println("blastoff!");
```

Good Code Style

Minimize # of magic numbers (make change easier).

```
int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; i++) {
   System.out.print(N+1-i + ", ");
}
System.out.println("blastoff!");
```

Counting Down: v4

Does the following program give the correct countdown?:

```
int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; N++) {
    System.out.print(N+1-i + ", ");
}
System.out.println("blastoff!");
```

Counting Down: v4

Does the following program give the correct countdown?:

```
int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; N++) {
   System.out.print(N+1-i + ", ");
}
System.out.println("blastoff!");
```

Answer: No. There is a typo (N for i)

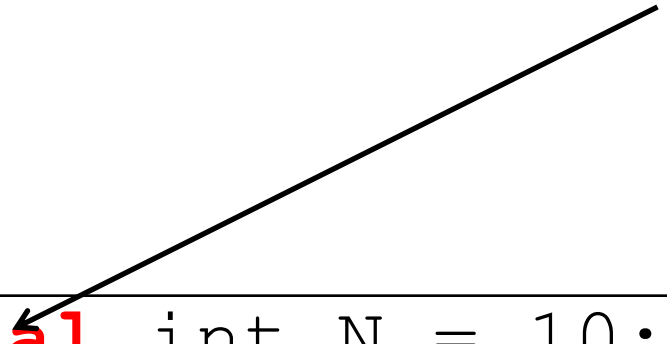
Q: can the computer help me to find it (read my mind?)

Constant

- Use keywords to tell computer your intension
- If there is a `final` before a variable declaration, it is your promise to the computer that you will not modify it after declaration
- If you break your promise, the compiler will catch you

Good Program Style

Convey your intention to the computer to help you.



```
final int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; i++) {
   System.out.print(N+1-i + “, “);
}
System.out.println("blastoff!");
```

Outline

- Admin and recap
- for loops
  - basic syntax
  - CountDown as as loop and coding style example
  - variable scoping of loop variables

Counting Down: Code Puzzle

What if we want to print out the values of N after the loop:

```
final int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; i++) {
   System.out.print(N+1-i + ", ");
}
System.out.println("blastoff!");

System.out.println("N = " + N); //?
```

Counting Down: Code Puzzle

What if we want to print out the values of i after the loop:

```
final int N = 10;
System.out.print("T-minus ");
for (int i = 1; i <= N; i++) {
   System.out.print(N+1-i + ", ");
}
System.out.println("blastoff!");

System.out.println("Final i =" + i); //?
```

Counting Down: Code Puzzle

```
% javac CountDownValue.java
CountDownValue.java:25: cannot find symbol
symbol  : variable i
location: class CountDownValue
      System.out.println( "Final i = " + i );
                                         ^
1 error
```

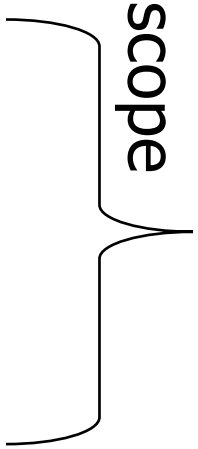
Variable Scope

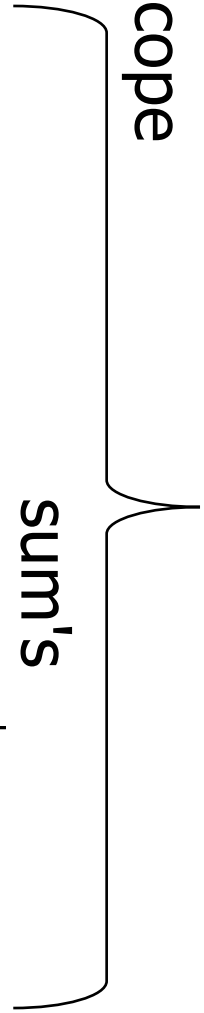
- **Scope**: The part of a program where a variable exists.
- Basic rule: from its declaration to the end of the enclosing `{ }` braces
- Examples
  - A variable declared in a `for` loop exists only in that loop.
  - A variable declared in a specific method exists only in that method.
  - A variable declared not inside any method but in a class is said to have class scope.

Variable Scope

```
public class CountSum {
   static int N = 10;
   public static void main(String[] args) {
      countSum();
   }

   public static void countSum() {
      System.out.print("T-minus ");
      int sum = 0;
      for (int i = 1; i <= N; i++) {
          System.out.println( i );
          sum += i;
      }
      System.out.println("N: " + N);
      System.out.println("Sum: " + sum);
   } // end of countSum
} // end of class
```





Why Scope?

- Encapsulation
  - e.g., different methods can use the same variable name without the need for coordination
  - many analogies: folders allow same file name so long in different folders

```
public static void aMethod()
{
     int x = 1;
      …
}
```

```
public static void bMethod()
{
     int x = 2;
      …
}
```

Loop Example

- Does the following code work?

```
public static void main() {
    final int N = 10;
    for (int i = 1; i <= N; i++) {
        System.out.print( N+1-i + " " );
    }
    System.out.println();

    for (int i = 1; i <= N; i++) {
        System.out.print( N+1-i + " ");
    }
}
```

Output:

```
10 9 8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
```

Loop Example

- Does the following code work?

```
public static void main() {
    final int N = 10;
    int i;
    for (i = 1; i <= N; i++) {
        System.out.print( N+1-i + " " );
    }
    System.out.println();

    for (int i = 1; i <= N; i++) {
        System.out.print( N+1-i + " ");
    }
}
```

Loop Example

- Does the following code work?

```
public static void main() {
    final int N = 10;
     int i;
    for (i = 1; i <= N; i++) {
        System.out.print( N+1-i + " " );
    }
    System.out.println();

    for (i = 1; i <= N; i++) {
        System.out.print( N+1-i + " ");
    }
}
```

Code Style

```
public static void main() {
   final int N = 10;
   int i;
   for (i = 1; i <= N; i++) {
      System.out.print( N+1-i + " " );
   }
   System.out.println();
   for (i = 1; i <= N; i++) {
      System.out.print( N+1-i + " ");
   }
}
```

Personally, I prefer this one as it declares variables with minimal scope.

```
public static void main() {
   final int N = 10;
   for (int i = 1; i <= N; i++) {
      System.out.print( N+1-i + " " );
   }
   System.out.println();
   for (int i = 1; i <= N; i++) {
      System.out.print( N+1-i + " ");
   }
}
```

Loop Example

- Does the following code work?

```
for (int set = 1; set <= 5; set++) {
    for (int rps = 1; rps <= set; rps++) {
        System.out.print("*");
    }
    System.out.println();
}
```

Output:

```
*
**
***
****
*****
```

Outline

- Admin and recap
- for loops
  - basic syntax
  - CountDown as as loop and coding style example
  - variable scoping of loop variables
  - nested loops

Nested Loop

```
for (int set = 1; set <= 5; set++) {
    for (int rps = 1; rps <= set; rps++) {
        System.out.print("*");
    }
    System.out.println();
}
```

- There can be another loop inside one loop to form a nested loop
- The #loop times of the inner loop can depend on the outer loop variable
- Nested loops are common, important programming patterns.

Practice: Nested `for` loop example

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print(i);
    }
    System.out.println();
}
```

- Output:

```
1
22
333
4444
55555
```

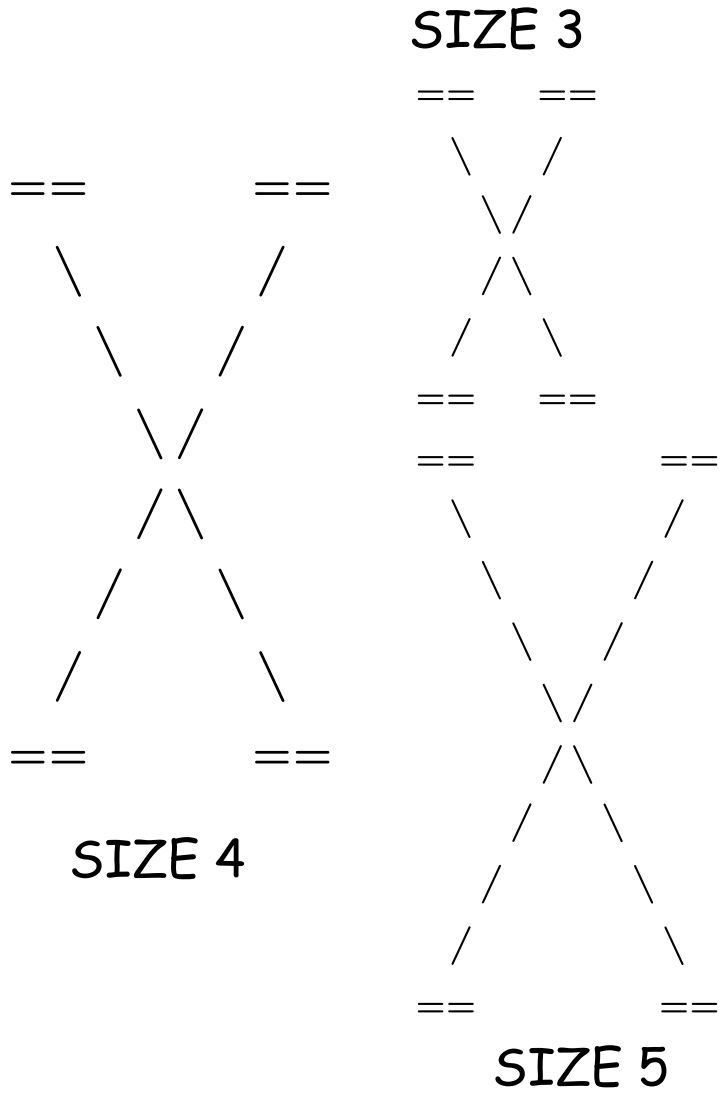
Practice: Nested `for` loop example

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 9; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print(i*j + "\t");
    }
    System.out.println();
}
```

Nested Loop Design Example: Drawing A Complex Pattern

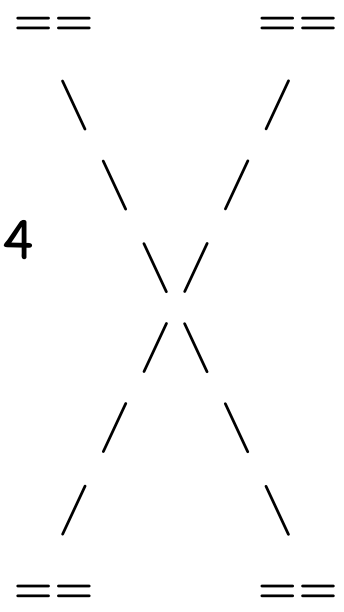
- Use nested `for` loops to draw ASCII X
- A size SIZE ASCII X has 2 * SIZE rows
- Why draw ASCII art?
  - Real graphics will require some finesse (shortly)
  - ASCII art has complex patterns
  - Can focus on the algorithms



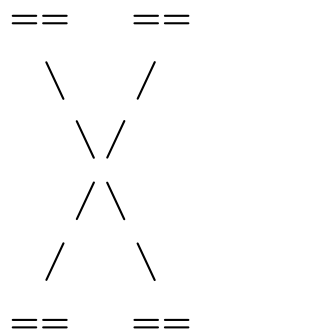
Drawing ASCII X

A size SIZE (S) ASCII X has 2 * SIZE rows, 2*SIZE columns

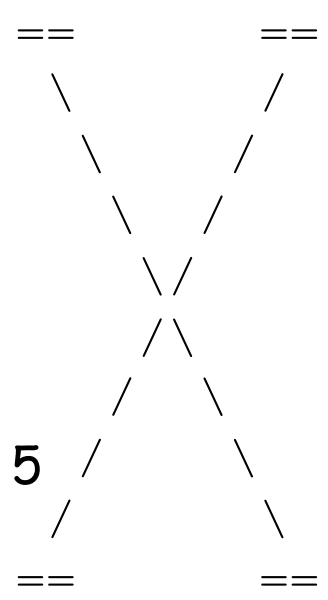
SIZE 4



SIZE 3

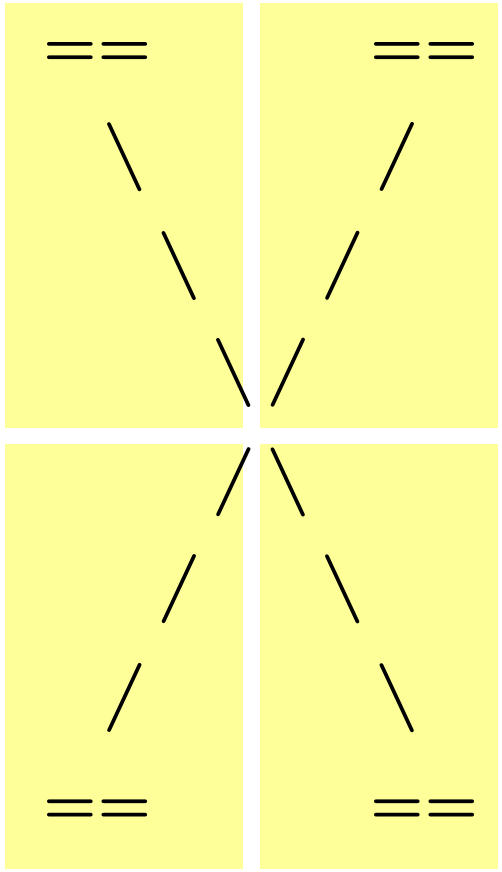


SIZE 5



Top-Down Decomposition

A size SIZE (S) ASCII X has 2 * SIZE rows, 2*SIZE columns



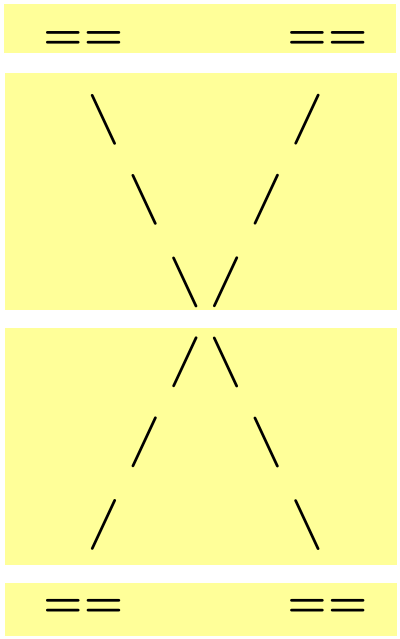
This decomposition is hard to implement because ASCII drawing cursor movement constraint: left -> right, top -> bottom

Better decompose along movement line

Drawing ASCII X

A size SIZE (S) ASCII X has 2 * SIZE rows, 2*SIZE columns

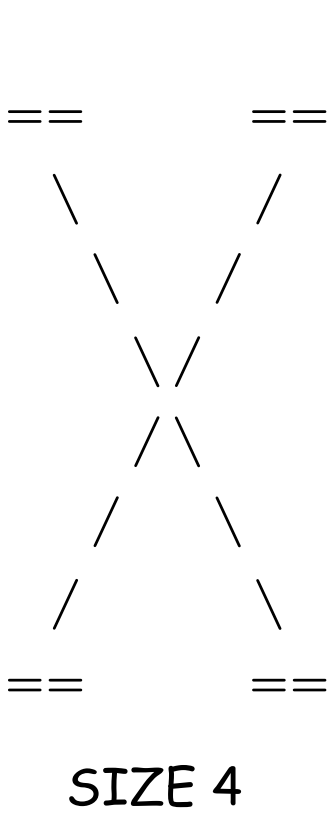
1. Bound
2. Top half (drawV)
3. Bottom half (top half upside-down, drawUV)
4. Bound



SIZE 4

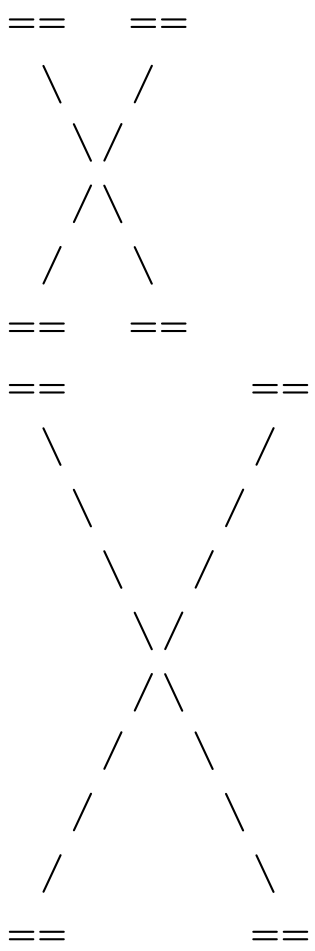
Top-Down Decomposition

A size SIZE (S) ASCII X has 2 * SIZE rows, 2*SIZE columns

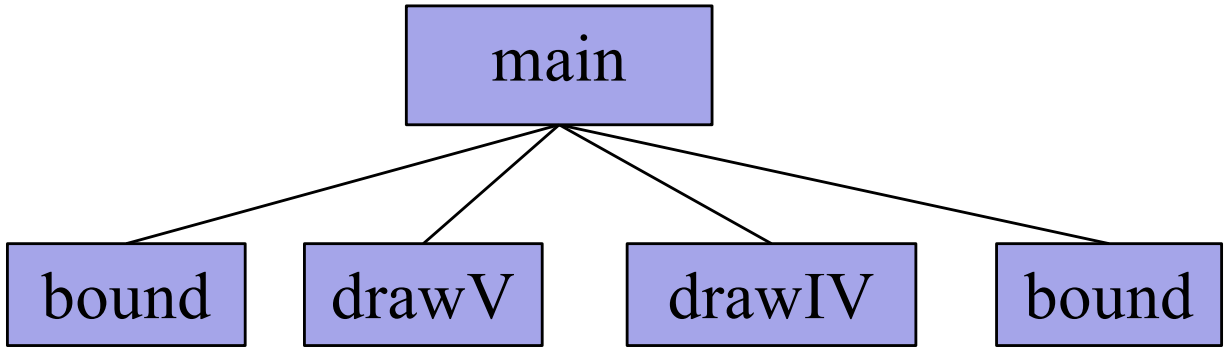


SIZE 4

SIZE 3

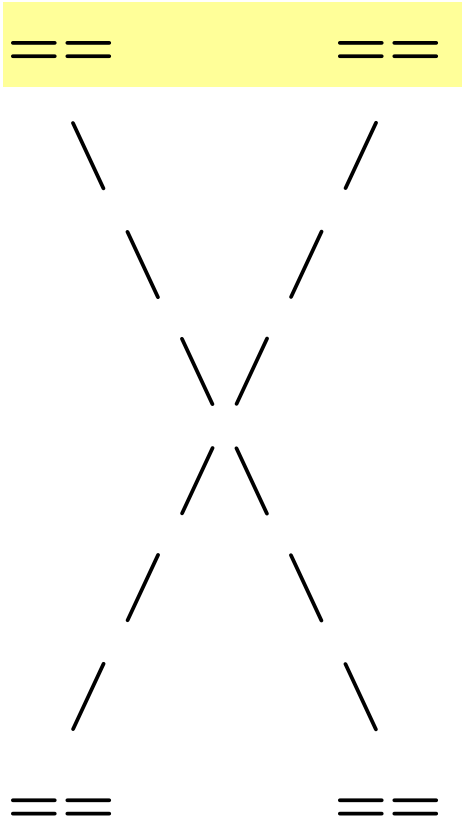


SIZE 5



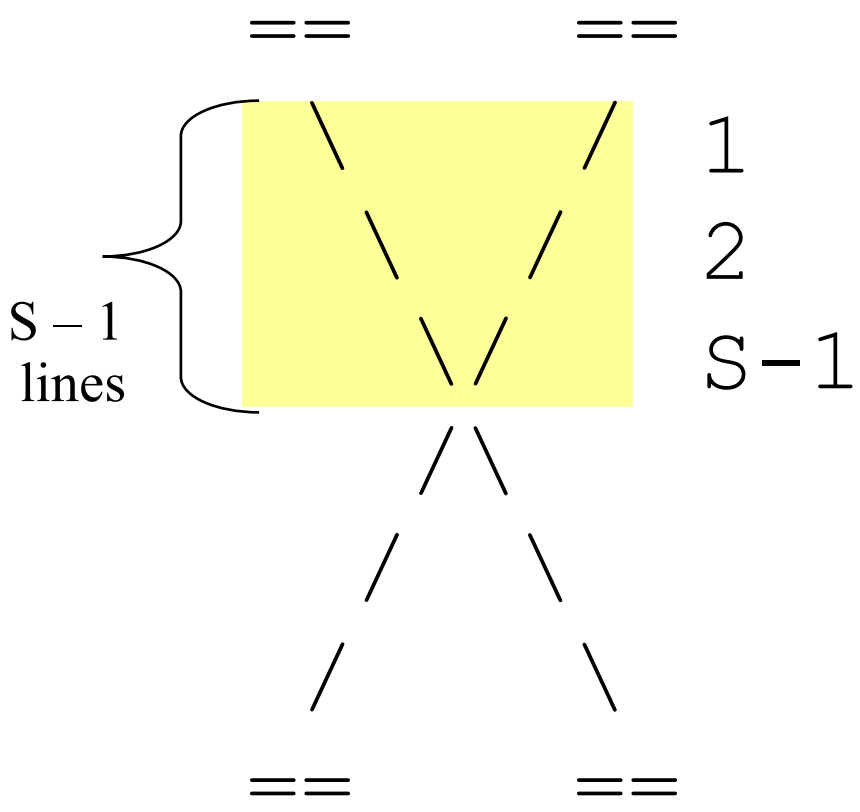
Bound

A size SIZE (S) ASCII X has 2 * SIZE rows, 2*SIZE columns



== , 2 SIZE - 2 *2 spaces, ==

Top Half (V)



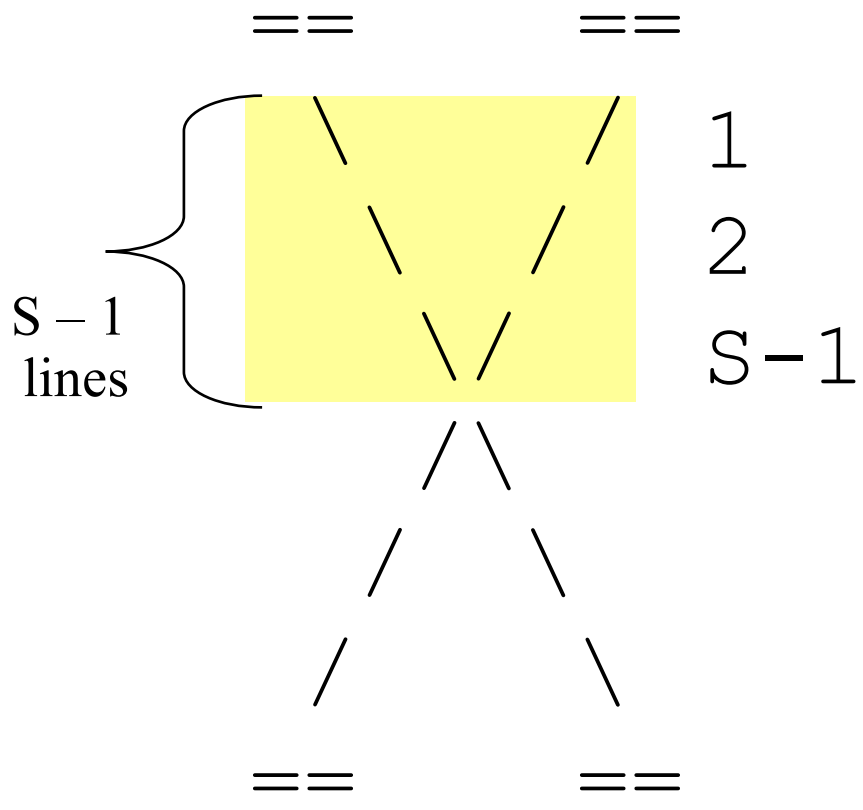
Two dimensional
=> nested loop
S - 1 lines

```
for (int line = 1;
     line <= S-1; line++)
{
    …
}
```

Structure of each line:

- some white space
- \
- some white space
- /

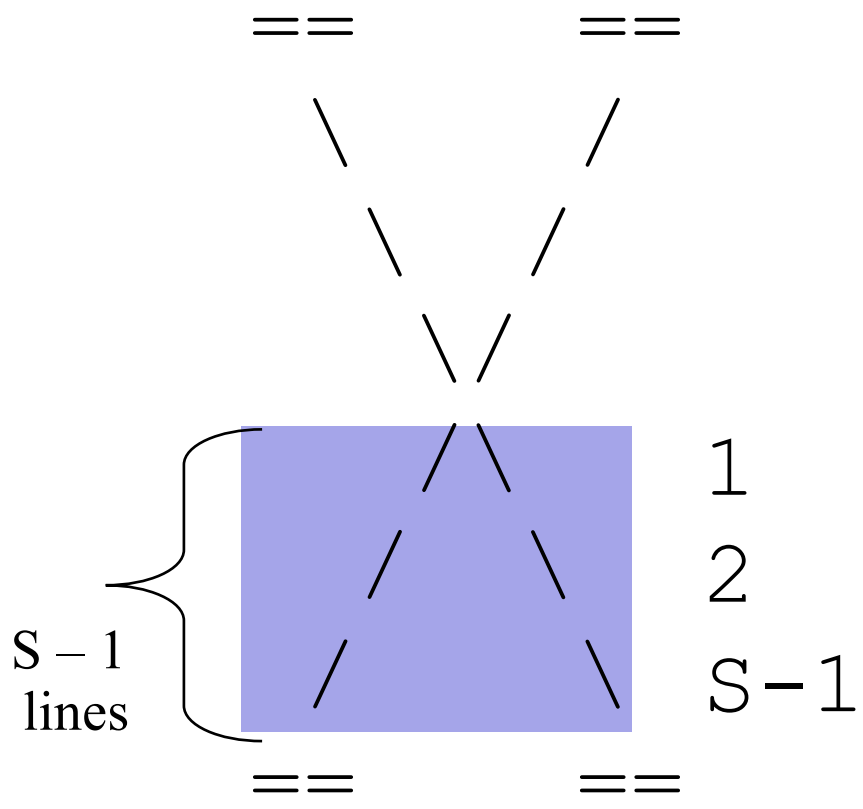
Top Half (V)



Structure at line:

- some white space
  - line # white spaces
- \
- some white space
  - slope : -2
  - line=1: 2S-4 spaces
  - V0 = 2S-4 + 2
  - 2S - 2 - 2 line spaces
- /

Bottom Half (IV)



Structure at line:

- some white space
  - slope : -1
  - line=1: S-1 spaces
  - V0 = S
  - S - line spaces
- /
- some white space
  - slope : 2
  - line=1: 0 spaces
  - V0 = -2
  - -2 + 2 line spaces
- \

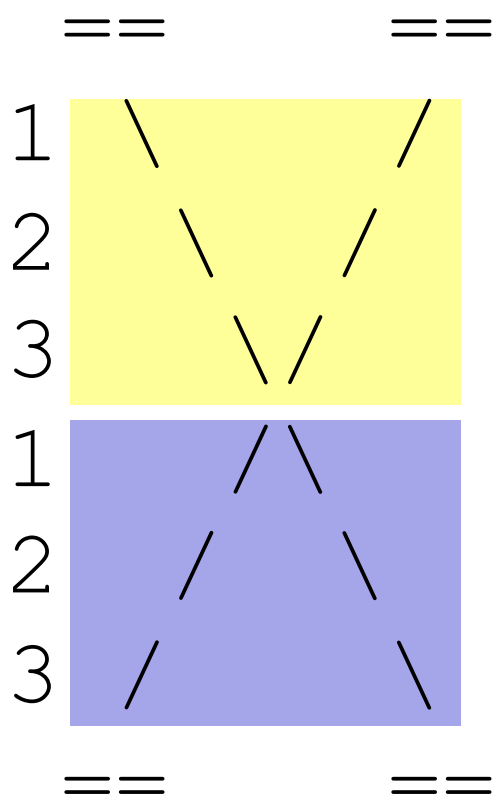
Complete Pseudo Code

1. Bound
   - == , 2S – 4 spaces, ==

2. for line = 1 to S -1
   line spaces
   \
   2 S – 2 – 2 * line spaces
   /

3. for line = 1 to S – 1
   S – line spaces
   /
   (line – 1) * 2 spaces
   \

4. Bound



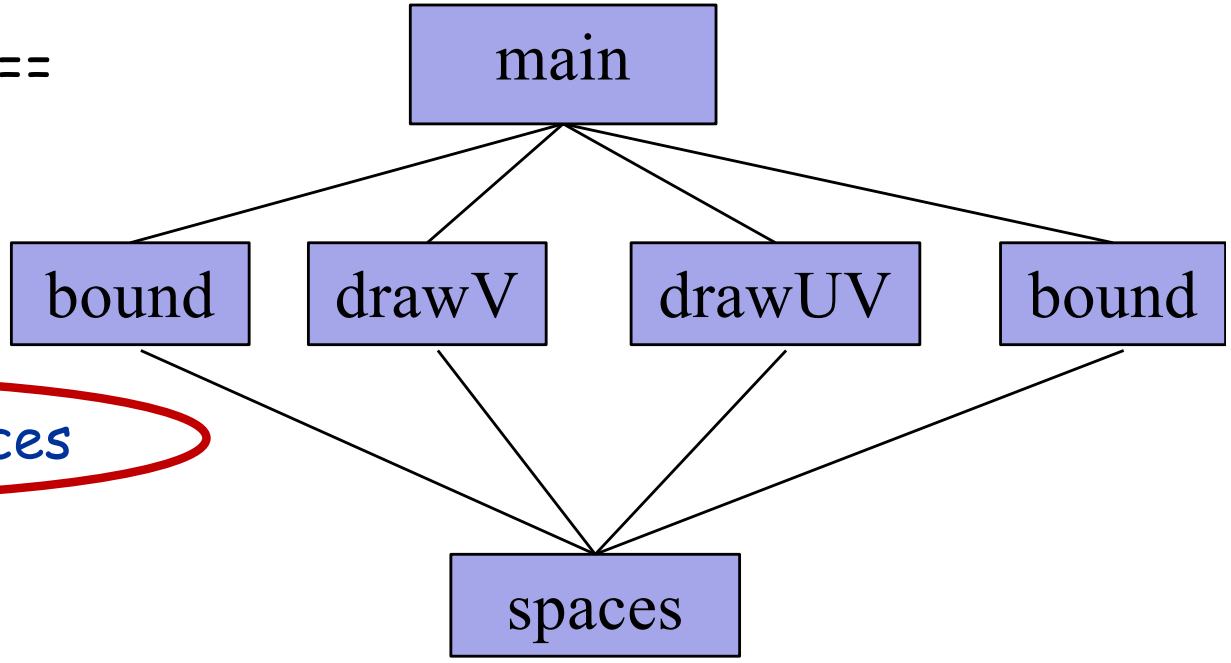
Identify Subtask (Method)?

1. Bound
   - == 2S - 4 spaces ==

2. for line = 1 to S -1
   - line spaces
   - 2 S - 2 - 2 * line spaces

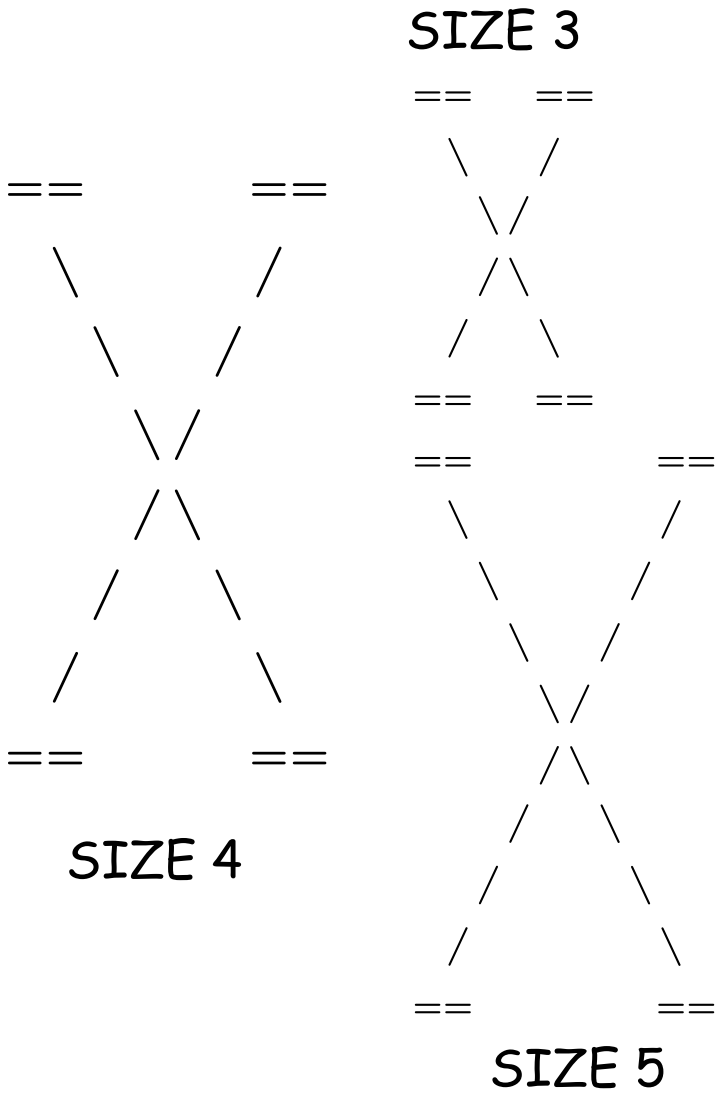
3. for line = 1 to S - 1
   - S - line spaces
   - (line - 1) * 2 spaces

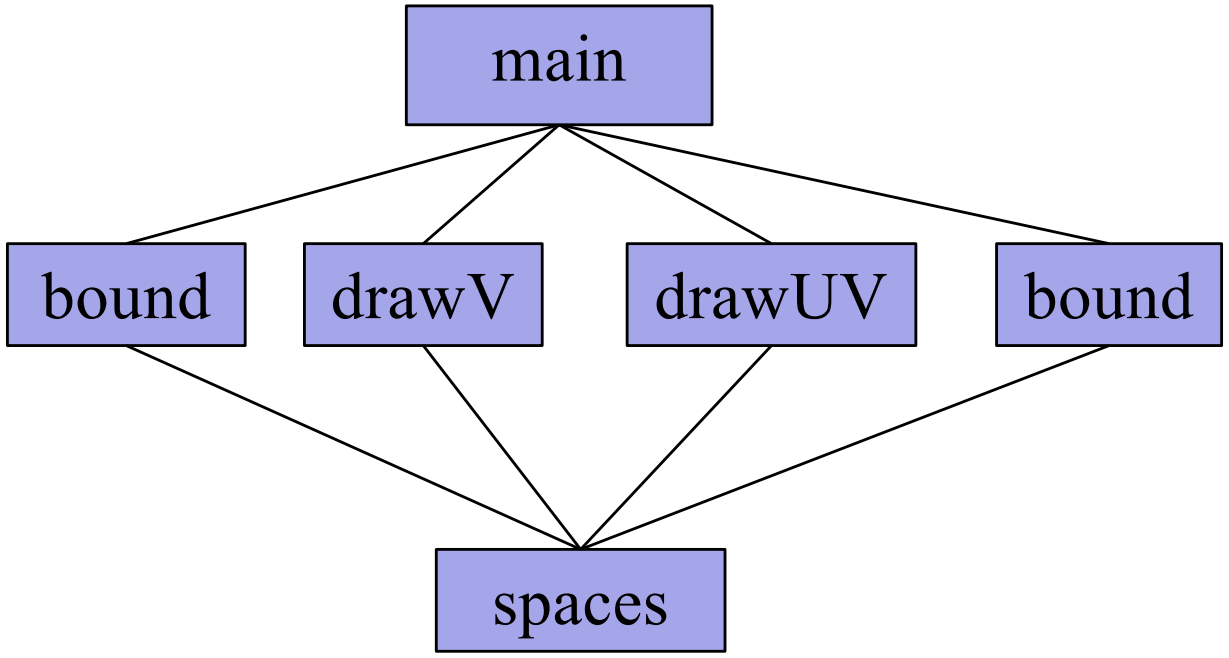
4. Bound



Drawing spaces is a reusable function.

Top-Down Decomposition





Issue: Drawing spaces need to draw different numbers of spaces.

Method Parameterization

- Specify a parameter to control the behavior of a method
  - Methods with parameters solve an entire class of similar problems
- Redundancy removal/abstraction through generalization
  - The more general a building block, the easier to reuse it
  - We will learn more techniques on generalization/abstraction

Parameterization

- **parameter**: A value passed to a method by its caller, e.g.,
  - When *declaring* a method, we will state that it requires a parameter for the number of spaces.
  - When *calling* the method, we will specify the number.

Syntax: Declaring a Method with a Parameter

```
public static void <method_name> (<type> <param_name>) {
    <statement>(s);
}
```

- The parameter is called the formal argument

- Example:

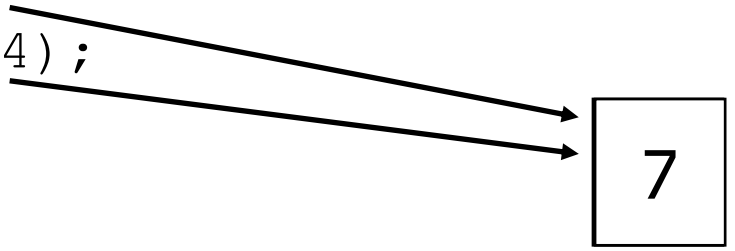
```
public static void chant(int times) {
    for (int i = 1; i <= times; i++) {
        System.out.println("Just a salad...");
    }
}
```

How Parameters are Passed

- When a method with a formal argument is called:
  - A value is passed to the formal argument
  - The passed value is called the **actual argument**
  - The method's code executes using that value.

```
public static void main(String[] args) {
    chant(3);
    chant(3+4);
}

public static void chant(int times) {
    for (int i = 1; i <= times; i++) {
        System.out.println("Just a salad...");
    }
}
```



Common Errors

- If a method accepts a parameter, it is illegal to call it without passing any value for that parameter.

```
chant();        // ERROR: parameter value required
```

- The value passed to a method must be of the correct type.

```
chant(3.7);     // ERROR: must be of type int
```

Method Exercise

```
for (int set = 1; set <= 5; set++) {
    for (int rps = 1; rps <= set; rps++) {
        System.out.print("*");
    }
    System.out.println();
}
```

- Exercise: Turn the inner loop into a method to be invoked by the outer loop.

Method Exercise

- Exercise: Design and implement the `DrawX` program.

Out-Class Read: Another Way to Motivate Methods With Parameters

Example: Draw Matrix

- Print two copies of inverse diagonal matrix

```
..1
.2.
3..
....1
...2.
..3..
.4...
5....
```

```
int N = 5;
for (int line = 1; line <= N; line++) {
    // initial dots
    for (int j = 1; j <= (-1 * line + N); j++) {
        System.out.print(".");
    }

    // the number
    System.out.print(line);

    // the second set of dots
    for (int j = 1; j <= (line - 1); j++) {
        System.out.print(".");
    }
    System.out.println();
}
```

Solution 1

- This code is redundant.

```
public class DrawMatrix1 {
   public static void main(String[] args) {
      drawMatrix3();
      drawMatrix5();
   }

   public static void drawMatrix3() {
      int N = 3;
      for (int line = 1; line <= N; line++) {
         for (int j = 1; j <= (-1 * line + N); j++) { // initial dots
            System.out.print(".");
         }
         System.out.print(line);                     // the number
         for (int j = 1; j <= (line - 1); j++) { // the second set of dots
            System.out.print(".");
         }
         System.out.println();
      }
   } // end of drawMatrix3
   public static void drawMatrix5() {
      int N = 5;
      for (int line = 1; line <= N; line++) {
         for (int j = 1; j <= (-1 * line + N); j++) { // initial dots
            System.out.print(".");
         }
         System.out.print(line);                     // the number
         for (int j = 1; j <= (line - 1); j++) { // the second set of dots
            System.out.print(".");
         }
         System.out.println();
      }
   } // end of drawMatrix5
}
```

Remove Redundancy

```
public class DrawMatrix2 {

   public static void main(String[] args) {
      int N = 3;
      drawMatrix(); // should print 3

      N = 5;
      drawMatrix(); // should print 5
   } // end of main

   public static void drawMatrix() {
      for (int line = 1; line <= N; line++) {
         for (int j = 1; j <= (-1 * line + N); j++) { // initial dots
            System.out.print(".");
         }
         System.out.print(line);                     // the number

         for (int j = 1; j <= (line - 1); j++) { // the second set of dots
            System.out.print(".");
         }
         System.out.println();
      } // end of for line
   } // end of drawMatrix
}
```

ERROR:
N is out of scope

Solution 2

```
public class DrawMatrix2 {
   static int N; // introduce a class scope variable

   public static void main(String[] args) {
      N = 3;
      drawMatrix(); // should print 3

      N = 5;
      drawMatrix(); // should print 5
   } // end of main

   public static void drawMatrix() {
      for (int line = 1; line <= N; line++) {
         for (int j = 1; j <= (-1 * line + N); j++) { // initial dots
            System.out.print(".");
         }
         System.out.print(line);                     // the number

         for (int j = 1; j <= (line - 1); j++) { // the second set of dots
            System.out.print(".");
         }
         System.out.println();
      } // end of for line
   } // end of drawMatrix
}
```

Problem of Class Variable Indirection

```
public class DrawMatrix2 {
   static int N; // introduce a class scope variable

   public static void main(String[] args) {
      drawMatrix();
      drawMatrix();
   } // end of main

   public static void drawMatrix() {
      for (int line = 1; line <= N; line++) {
         for (int j = 1; j <= (-1 * line + N); j++) { // initial dots
            System.out.print(".");
         }
         System.out.print(line);                     // the number

         for (int j = 1; j <= (line - 1); j++) { // the second set of dots
            System.out.print(".");
         }
         System.out.println();
      } // end of for line
   } // end of drawMatrix
}
```

Not self clear;
Implicit, depends on context;

Solution 3: Parameterization

- Specify a parameter to control the behavior of a method
  - Methods with parameters solve an entire class of similar problems

- Redundancy removal/abstraction through generalization
  - The more general a building block, the easier to reuse it
  - We will learn more techniques on generalization/abstraction

End of Out-Class Read

Backup Slides

Type Conversions in Java

- Identity conversion (i.e., no conversion)
- Conversions related to primitive data types
  - √ widening primitive conversions
  - √ narrowing primitive conversions
- Conversions related to classes
  - widening reference conversions
  - narrowing reference conversions
  - we will cover these two cases later in the course; they are powerful tools to allow polymorphism
- Conversions related to Strings
  - √ string conversions: i.e., convert a numerical data to a string, e.g., the number 17 to the string "17"

Widening Primitive Conversions

- Widening primitive conversions are those that do not lose information about the overall magnitude of a numeric value
- Java defines 19 primitive conversions as widening primitive conversions

```
byte   → short, int, long, float, double
short  → int, long, float, double
char   → int, long, float, double
int    → long, float, double
long   → float, double
float  → double
```

- They are generally safe because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
  - can potential problems happen in some of the cases?

Narrowing Primitive Conversions

- Java defines 23 primitive conversions as narrowing primitive conversions

```
byte   → char
short  → byte, char
char   → byte, short
int    → byte, short, char
long   → byte, short, char, int
float  → byte, short, char, int, long
double→ byte, short, char, int, long, float
```

- Narrowing primitive conversions may lose either overall magnitude of a numeric value and/or precision

Assignment during Declaration

- You can assign a value to a variable when declaring it.
  - This is called initialization

- Syntax:

  <type> <name> = ***<expression>***;

  - `int x = (11 % 3) + 12;`

x	14

  - `double myGPA = 3.95;`

myGPA	3.95

Example: Receipt

- Once given a value, a variable can be used in expressions:

```
public class Receipt {
  public static void main(String[] args) {
    // Calculate total owed
    // assuming 6% tax / 15% tip
    int subtotal = 38 + 40 + 30;
    double tax = subtotal * .06;
    double tip = subtotal * .15;
    double total = subtotal + tax + tip;

    System.out.println("Subtotal: " + subtotal);
    System.out.println("Tax: " + tax);
    System.out.println("Tip: " + tip);
    System.out.println("Total: " + total);
  }
}
```