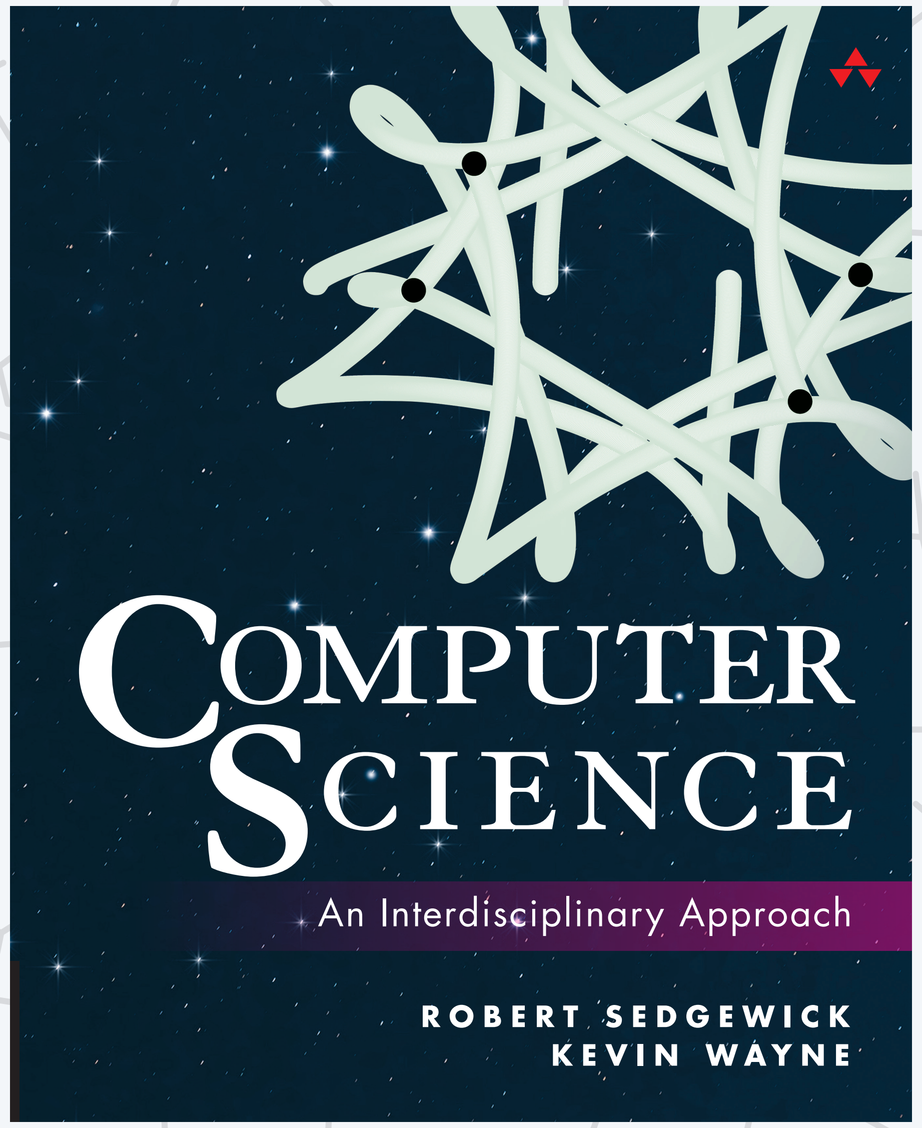
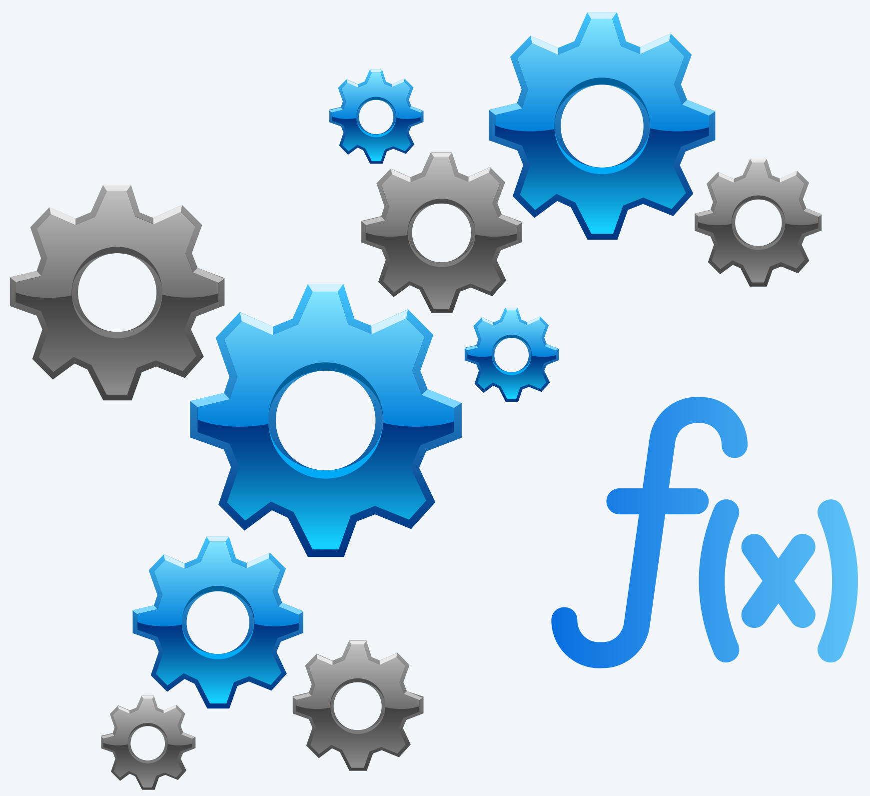


https://introcs.cs.princeton.edu

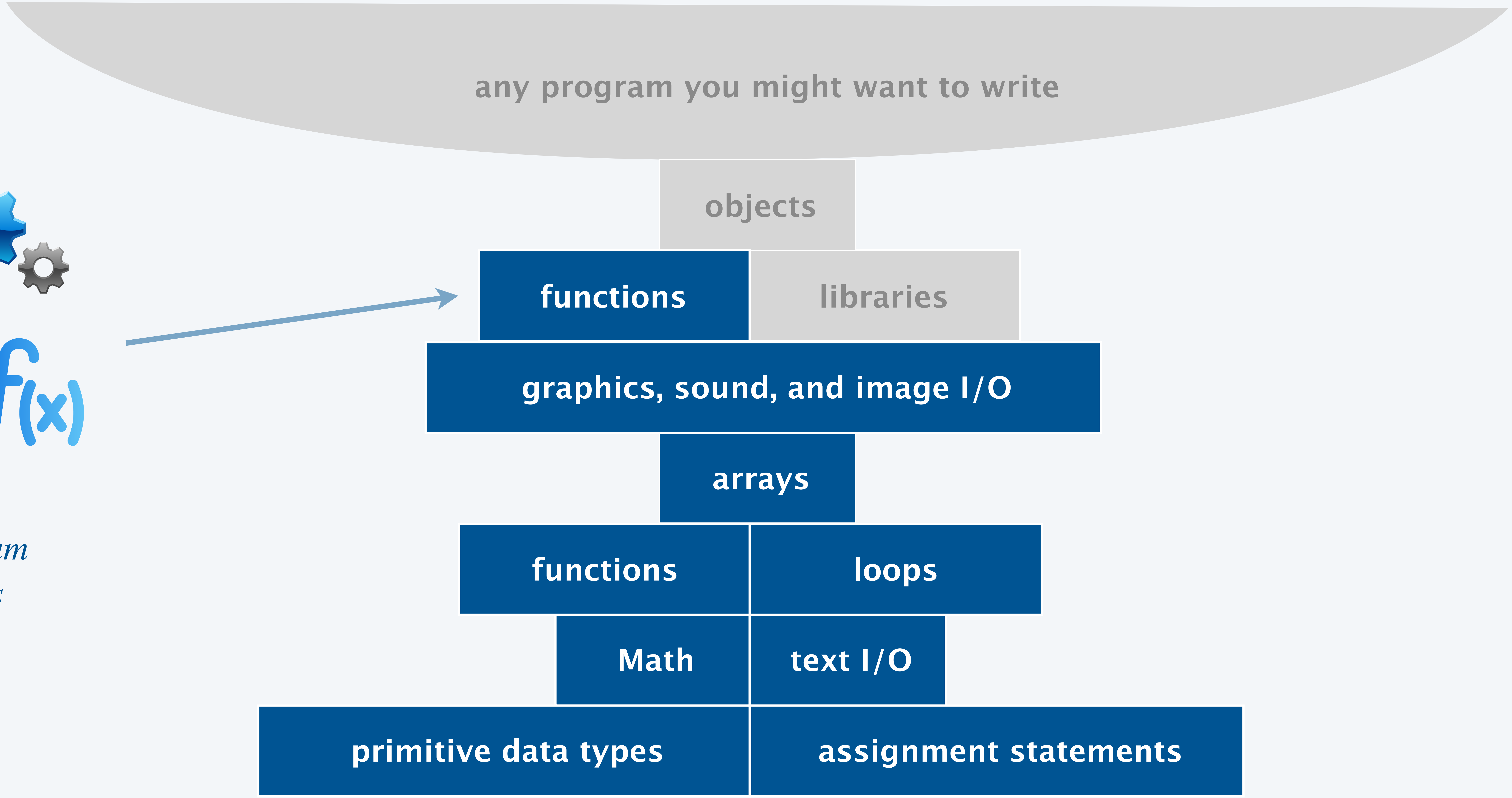
2.1 Functions

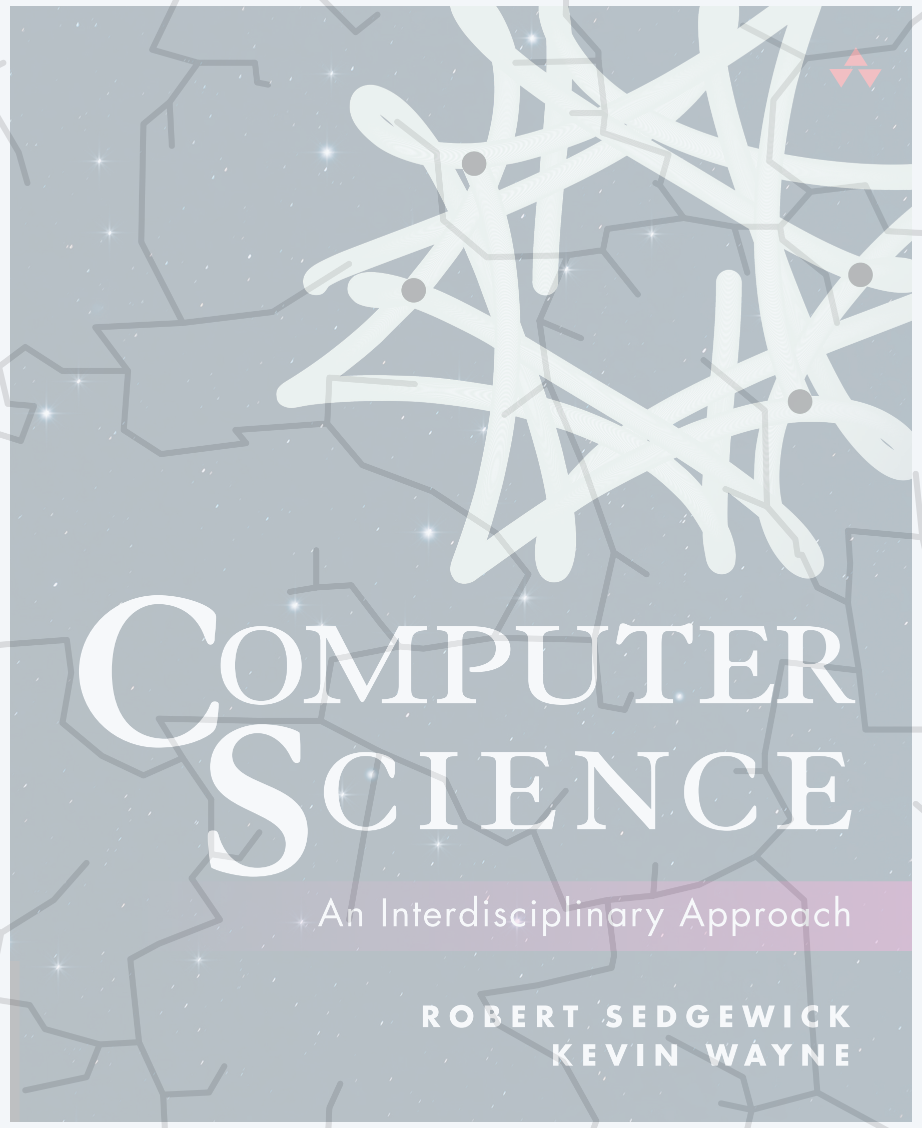
- *flow-of-control*
- *properties*
- *call by value*
- *number-to-speech*

Basic building blocks for programming



divide a program
into functions





https://introcs.cs.princeton.edu

2.1 Functions

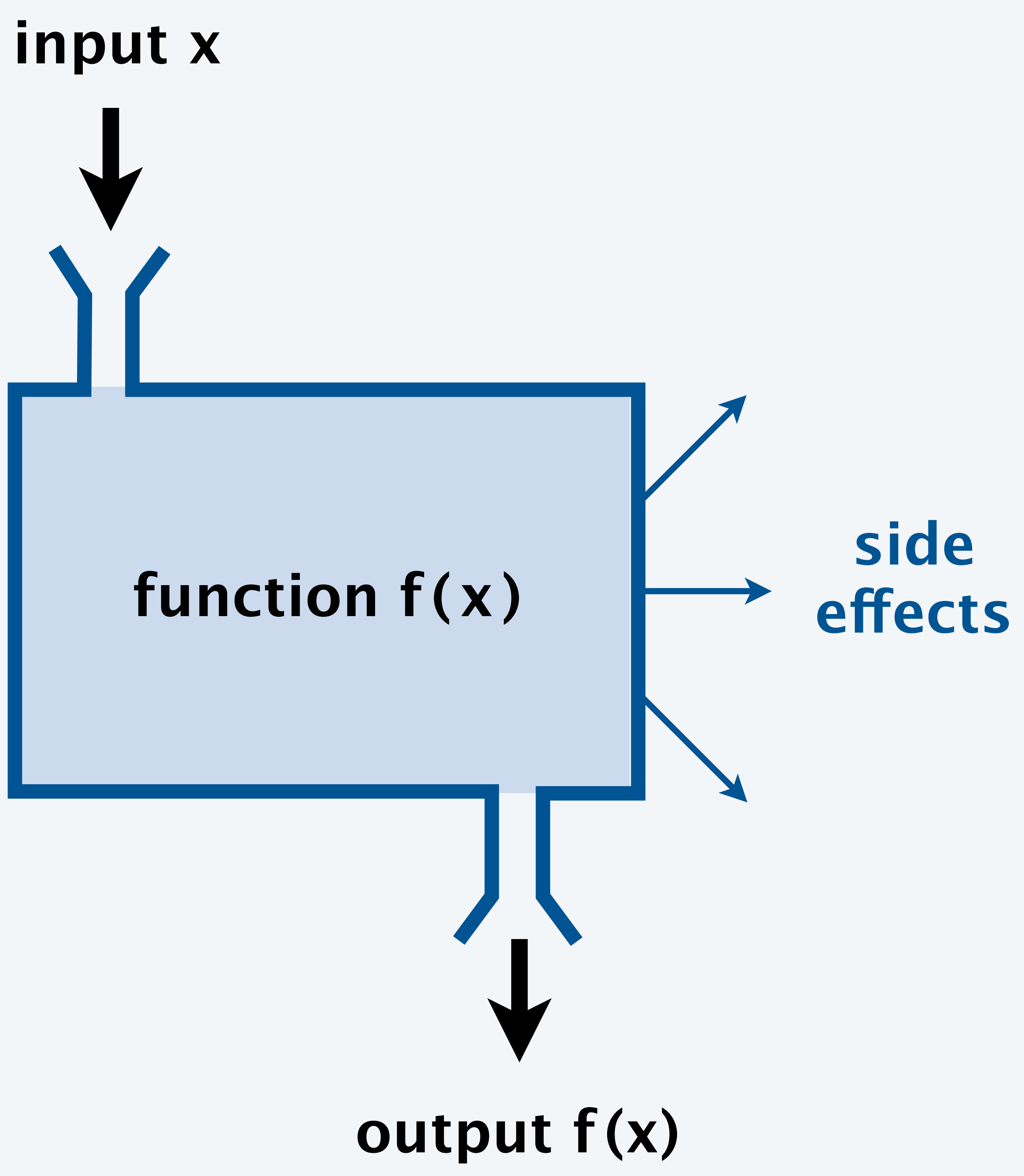
- *flow-of-control*
- *properties*
- *call by value*
- *number-to-speech*

Functions

Java function (static method).

- Takes zero or more input arguments.
- Returns zero or one output value.
- May cause side effects.

more general than mathematical functions



Benefits. Makes code easier to read, test, debug, reuse, and extend.

Familiar examples.

- Built-in functions: `Math.random()`, `Math.abs()`, `Integer.parseInt()`.
- Our I/O libraries: `StdIn.readInt()`, `StdDraw.line()`, `StdAudio.play()`.
- User-defined functions: `main()`.

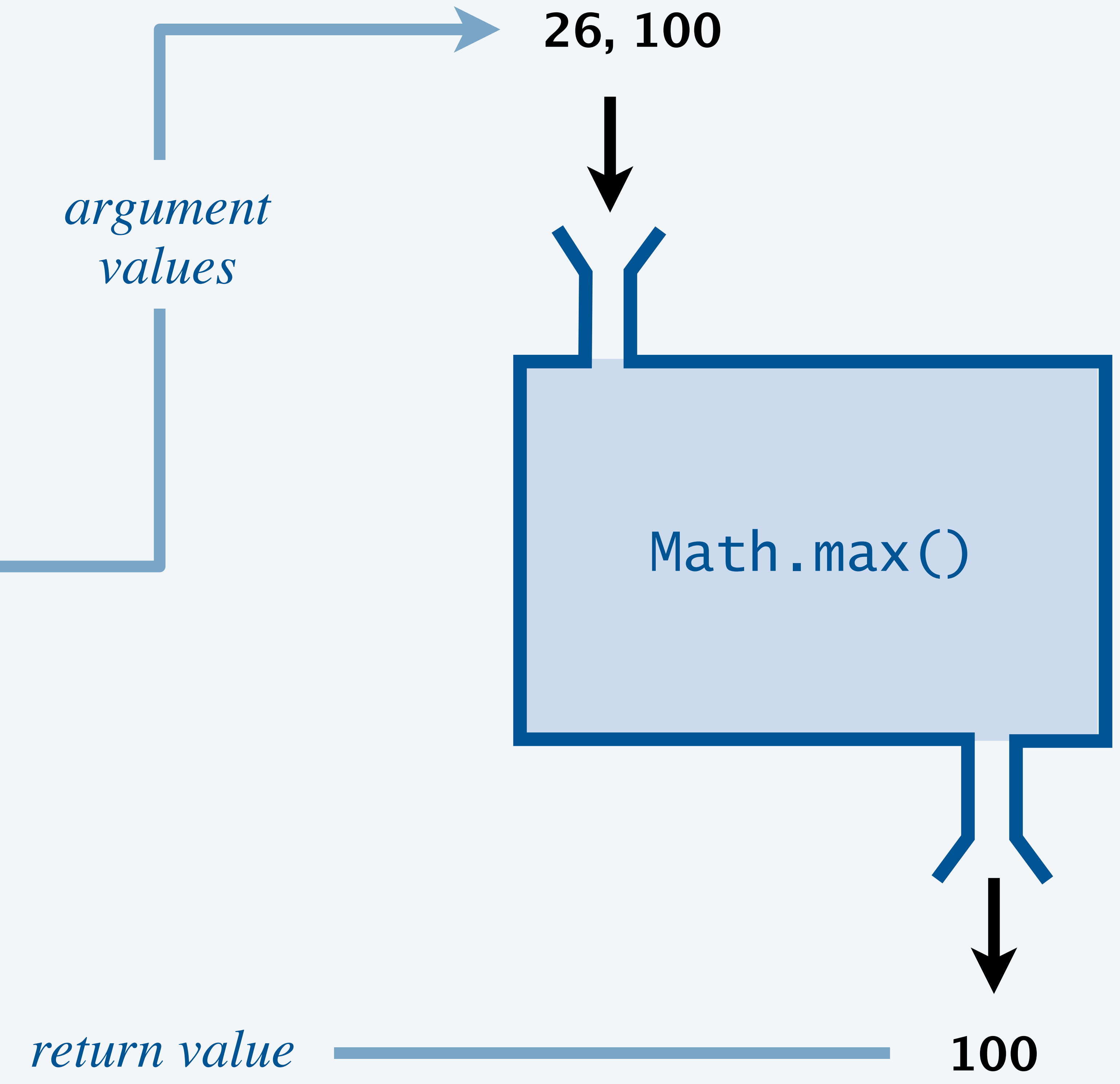
Flow of control

Mechanics of a function call.

- Control transfers from calling code to function code, passing argument values.
- Function code executes, producing a return value.
- Control transfers back to calling code. ← *function-call expression evaluates to return value*

```
public class Max {
   public static void main(String[] args) {
      int a = 26;
      int b = 100;
      int max = Math.max(a, b);
      StdOut.println(max);
   }
}
```

function call



Bottom line. Functions provide a useful *way* to control the flow of execution.

Anatomy of a Java function (static method)

To implement a Java function:

- Choose a *method name*.
- Declare type and name of each *parameter variable*.
- Specify type for *return value*.
- Include *modifiers*. ← *for now, always* `public` *and* `static`
- Implement *method body*, including a *return statement*.

(first four items ← *method header*)

Ex. Harmonic sum: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.

modifiers · *return type* · *method name* · *parameter type* · *parameter name*

method header →

```java
public static double sum(int n) {
   double result = 0.0;
   for (int j = 1; j <= n; i++)
      result += 1.0 / j;
   return result;
}
```

← *method body*

return statement →

Function call trace (i = 0)

```
public class Harmonic {

   public static double sum(int n) {
      double result = 0.0;
      for (int j = 1; j <= n; j++)
         result += 1.0 / j;
      return result;
   }

   public static void main(String[] args) {
     for (int i = 0; i < args.length; i++) {
         int arg = Integer.parseInt(args[i]);
         double value = sum(arg);
         StdOut.println(value);
      }
   }
}
```

i	arg	value
0	1	1.0

variable trace in *main*()

j	n	result
	1	0.0
1	1	1.0

variable trace in *sum*()

```
~/cos126/functions> java-introcs Harmonic 1 2 5
1.0
```

Function call trace (i = 1)

```
public class Harmonic {

   public static double sum(int n) {
      double result = 0.0;
      for (int j = 1; j <= n; j++)
         result += 1.0 / j;
      return result;
   }

   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         int arg = Integer.parseInt(args[i]);
         double value = sum(arg);
         StdOut.println(value);
      }
   }
}
```

i	arg	value
0	1	1.0
1	2	1.5

variable trace in *main*()

j	n	result
	2	0.0
1	2	1.0
2	2	1.5

variable trace in *sum*()

```
~/cos126/functions> java-introcs Harmonic 1 2 5
1.0
1.5
```

Function call trace (i = 2)

```
public class Harmonic {

   public static double sum(int n) {
      double result = 0.0;
      for (int j = 1; j <= n; j++)
         result += 1.0 / j;
      return result;
   }

   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         int arg = Integer.parseInt(args[i]);
         double value = sum(arg);
         StdOut.println(value);
      }
   }
}
```

i	arg	value
0	1	1.0
1	2	1.5
2	5	2.2833

variable trace in *main*()

j	n	result
	5	0.0
1	5	1.0
2	5	1.5
3	5	1.8333
4	5	2.0833
5	5	2.2833

variable trace in *sum*()

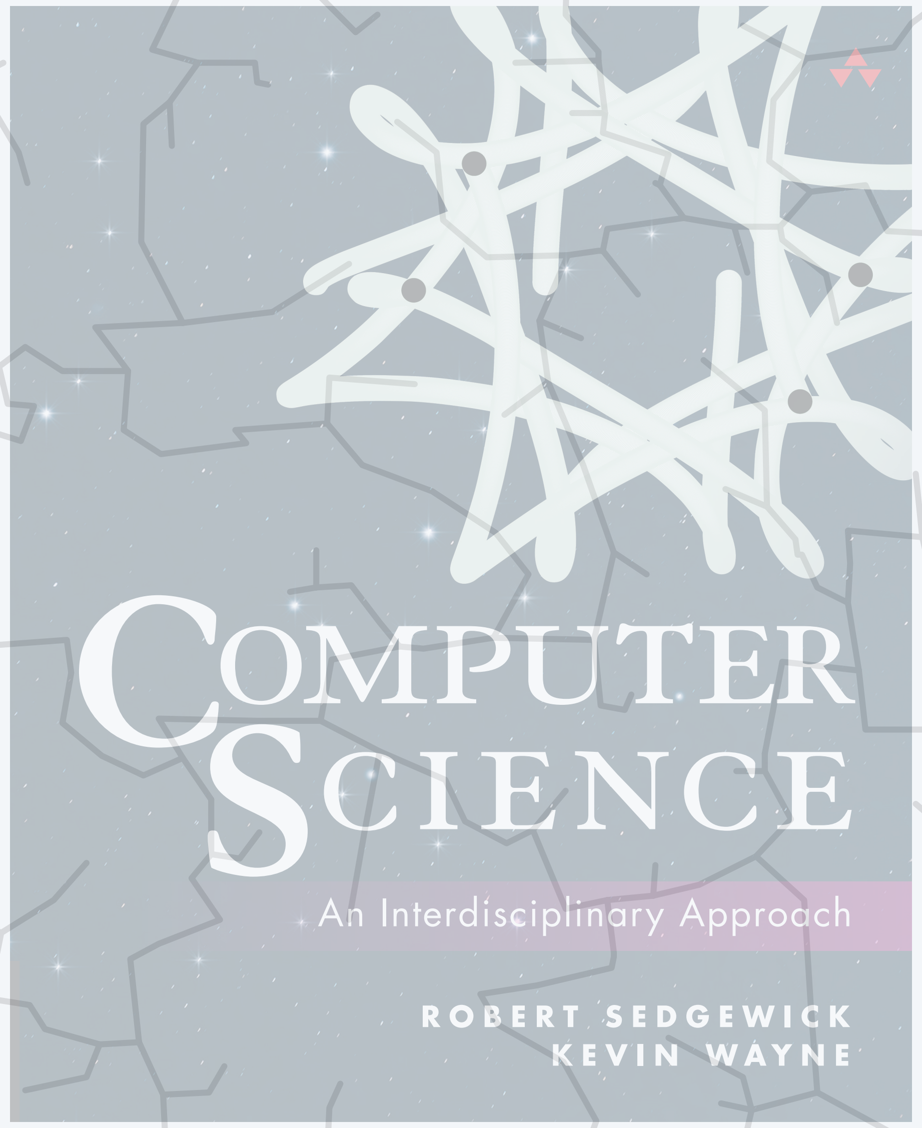
```
~/cos126/functions> java-introcs Harmonic 1 2 5
1.0
1.5
2.283333333333333
```

What is the result of executing this program with the given command-line argument?

- **A.** 126.0
- **B.** 378.0
- **C.** Compile-time error.
- **D.** Run-time error.

```
public class Mystery {

   public static double triple(double x) {
      return 3*x;
   }

   public static void main(String[] args) {
      double x = Double.parseDouble(args[0]);
      triple(x);
      StdOut.println(x);
   }
}
```

```
~/cos126/functions> java-introcs Mystery 126.0
```



https://introcs.cs.princeton.edu

2.1 FUNCTIONS

- *flow-of-control*
- *properties*
- *call by value*
- *number-to-speech*

Single *return* value

When a function reaches a return statement, it transfer control back to code that invoked it.

- The type of the return value must be compatible with the function's return type.
- Java returns a single return value to the calling code.

that value can be of any type (`double`, `String`, `int[]`, …)

return type

```java
public static double sum(int n) {
  double result = 0.0;
  for (int i = 1; i <= n; i++)
     result += 1.0 / i;
  return result;
}
```

return statement

Multiple *return* statements

Control is transferred back to calling code upon reaching the first *return* statement.

```
public static double signum(double x) {
   if      (x <  0.0) return -1.0;
   else if (x >  0.0) return +1.0;
   else if (x == 0.0) return  0.0;
   else return Double.NaN;
}
```

multiple `return` *statements*

sign (signum) function

```
public static double signum(double x) {
   if (x <  0.0) return -1.0;
   if (x >  0.0) return +1.0;
   if (x == 0.0) return  0.0;
   return Double.NaN;
}
```

equivalent function

$$signum(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

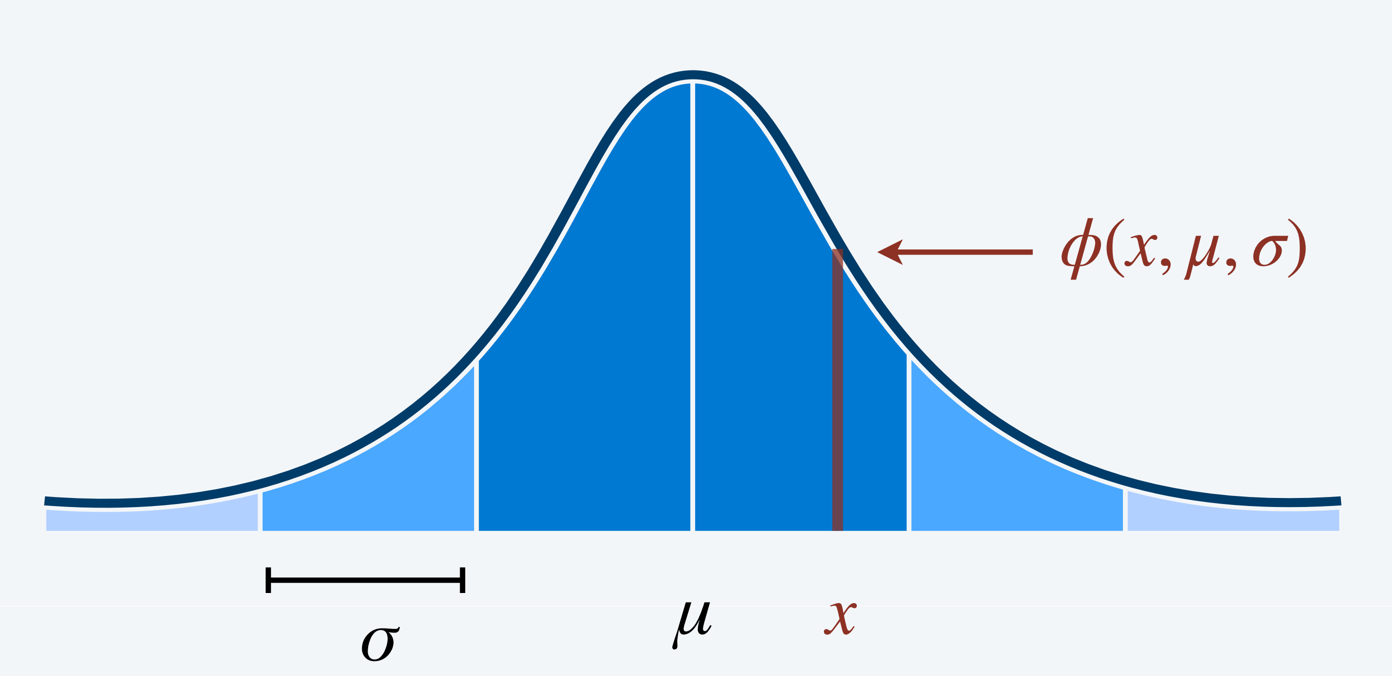
Note. This function appears in Java's *Math* library.

Multiple arguments

A function can take multiple arguments.

- Each parameter variable has a type and a name.
- The argument values are assigned to the corresponding parameter variables.

Ex. Gaussian (normal) probability distribution function: $\phi(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$.




```
pdf(88.0, 90.0, 10.0)
```

```
public class Gaussian {

   public static double pdf(double x, double mu, double sigma) {
      double z = (x - mu) / sigma;
      return Math.exp(-z*z / 2) / (sigma * Math.sqrt(2 * Math.PI));
   }

}
```

function takes three `double` *arguments*

Multiple functions

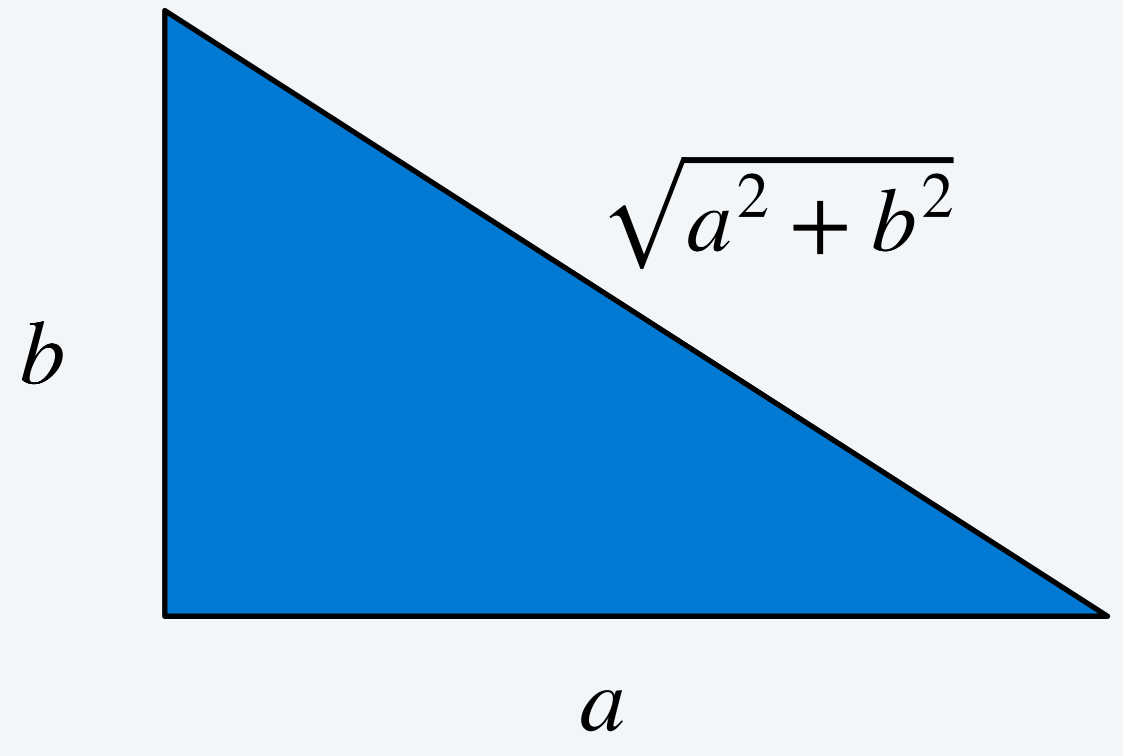
You can define many functions in a class.

- One function can call another function.
- The order in which the functions are defined in the file is unimportant.

```
public class RightTriangle {

   public static double square(double x) {
      return x*x;
   }

   public static double hypotenuse(double a, double b) {
      return Math.sqrt(square(a) + square(b));
   }

}
```

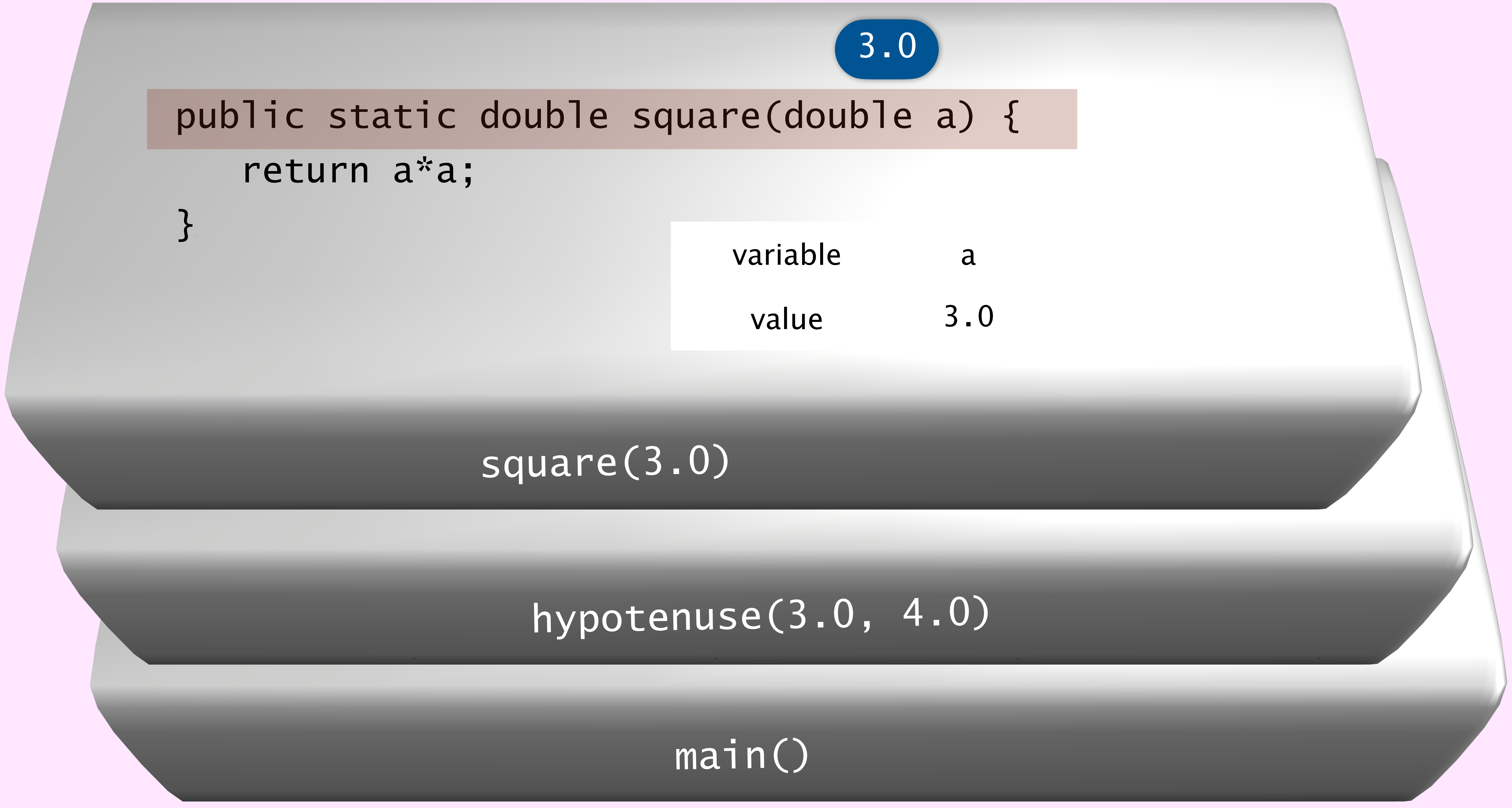
function calls a function defined in a different class

function calls a function defined in the same class



$\sqrt{a^2 + b^2}$

Mechanics of a function call



function-call stack

Overloaded functions

Overloading. Two functions with the same name (but different ordered list of parameter types).

```
public class Math {

   public static int abs(int x) {
      if (x < 0) return -x;
      else       return  x;
   }

   public static double abs(double x) {
      if (x < 0) return -x;
      else       return  x;
   }

}
```

`abs(-126)` *calls this function (and evaluates to* `126`*)*

`abs(-126.0)` *calls this function (and evaluates to* `126.0`*)*

Note. These two overloaded functions appear in Java's *Math* library.

Overloaded functions

Overloading. Two functions with the same name (but different ordered list of parameter types).

```
public class Gaussian {

   public static double pdf(double x) {
      return pdf(x, 0.0, 1.0);
   }

   public static double pdf(double x, double mu, double sigma) {
      double z = (x - mu) / sigma;
      return Math.exp(-z*z / 2) / (sigma * Math.sqrt(2 * Math.PI));
   }

}
```

pdf(3.0) *calls this function*

pdf(3.0, 0.0, 1.0) *calls this function*

pdf(3.0, 0.0, "126") *is incompatible (compile-time error)*

Bottom line. Java determines which function to call based on list of arguments.

Which value does *triple*(126) return?

A. 378

B. 378.0

C. "126126126"

D. Compile-time error.

E. Run-time error.

```
public class Mystery {

   public static double triple(double x) {
      return 3*x;
   }

   public static String triple(String x) {
      return x + x + x;
   }

}
```

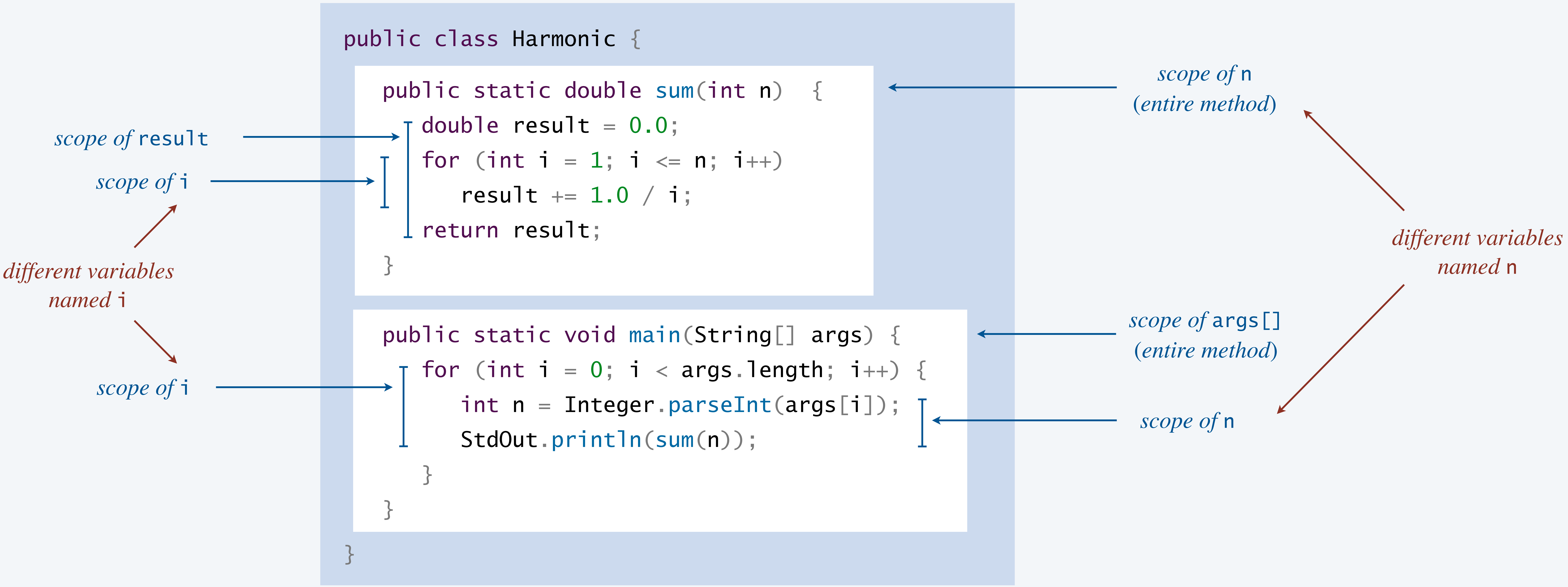
Scope of a variable

Def. The scope of a variable is the code that can refer to it by name. ← *code following its declaration, in the same block*

Significance. Can develop functions independently. ← *variables defined in one function do not interfere with variables defined in another*

Best practice. Declare variables so as to limit their scope.

```
public class Harmonic {
   public static double sum(int n)  {
      double result = 0.0;
      for (int i = 1; i <= n; i++)
         result += 1.0 / i;
      return result;
   }

   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         int n = Integer.parseInt(args[i]);
         StdOut.println(sum(n));
      }
   }
}
```



How many different variables named *n* are created when *Harmonic* is execute with 10 command-line arguments?

A. 1

B. 2

C. 10

D. 11

E. 20

```
public class Harmonic {
   public static double sum(int n)  {
      double result = 0.0;
      for (int i = 1; i <= n; i++)
         result += 1.0 / i;
      return result;
   }

   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         int n = Integer.parseInt(args[i]);
         StdOut.println(sum(n));
      }
   }
}
```

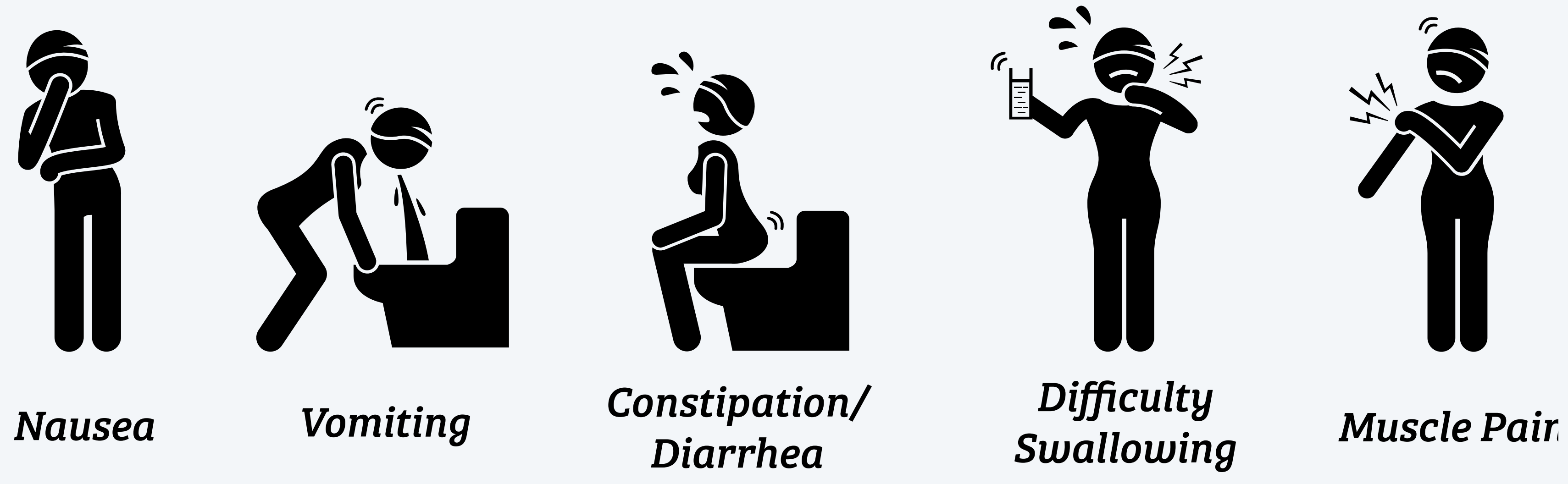
Side effects

Def. A side effect of a method is anything it does besides computing and returning a value.

- Print to standard output.
- Draw a circle.
- Play an audio file.
- Display a picture.
- Launch a missile.
- Consume input.
- Mutate an array.
- …

← *produce output* (Print to standard output, Draw a circle, Play an audio file, Display a picture, Launch a missile)

← *stay tuned* (Mutate an array)



Note. The primary purpose of some methods is to produce side effects, not return values.

↑ *differs from medicine*

Void functions

A method need not return a value.

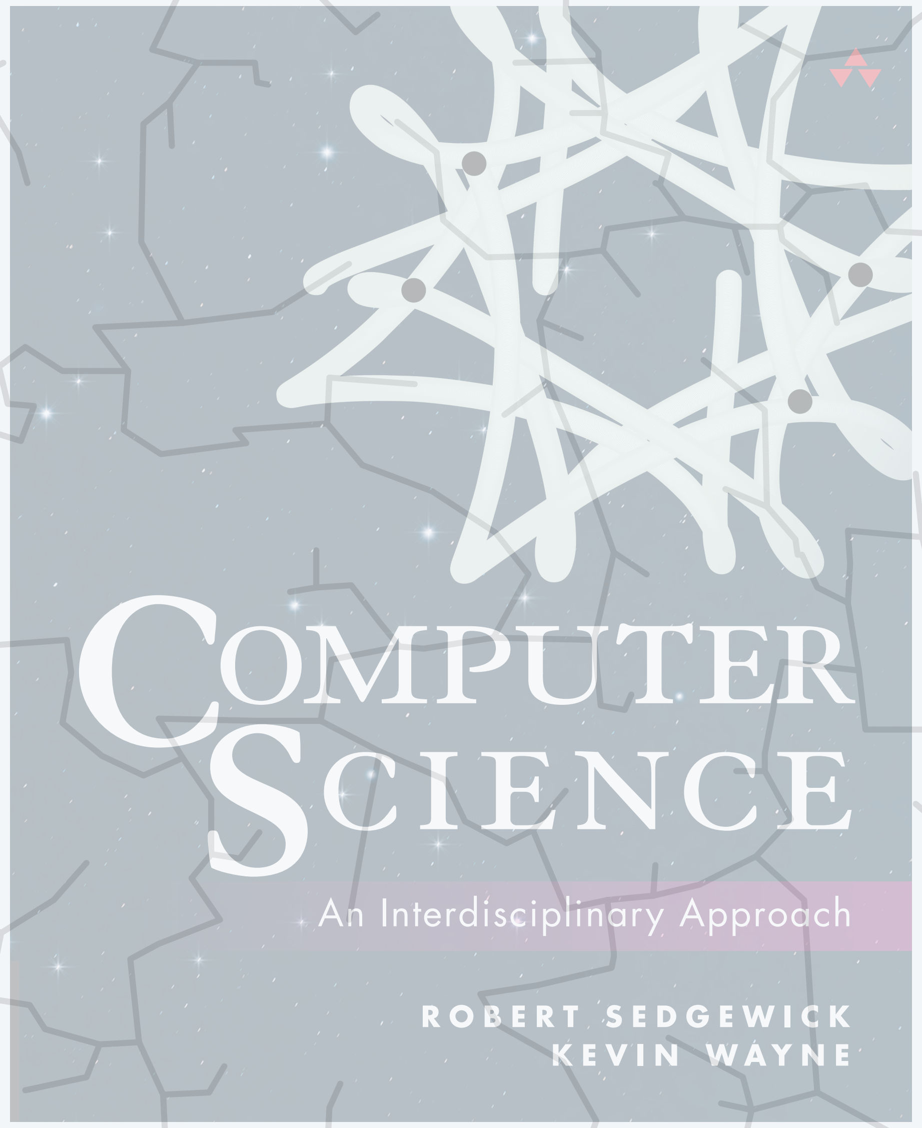
- Its purpose is to produce side effects.
- Use keyword *void* as return type.
- No explicit *return* statement needed. ← *upon reaching the end of method, control returns to calling code*

```
public static void loop(String filename, int n) {
   for (int i = 0; i < n; i++) {
      StdAudio.play(filename);
   }
}
```

loop an audio file n times

```
public static void main(String[] args) {
   int n = Integer.parseInt(args[0]);
   if (n <= 0) {
      StdOut.println("n must be positive");
      return;
   }
   ...
}
```

abort if the wrong number of command-line arguments



https://introcs.cs.princeton.edu

2.1 FUNCTIONS

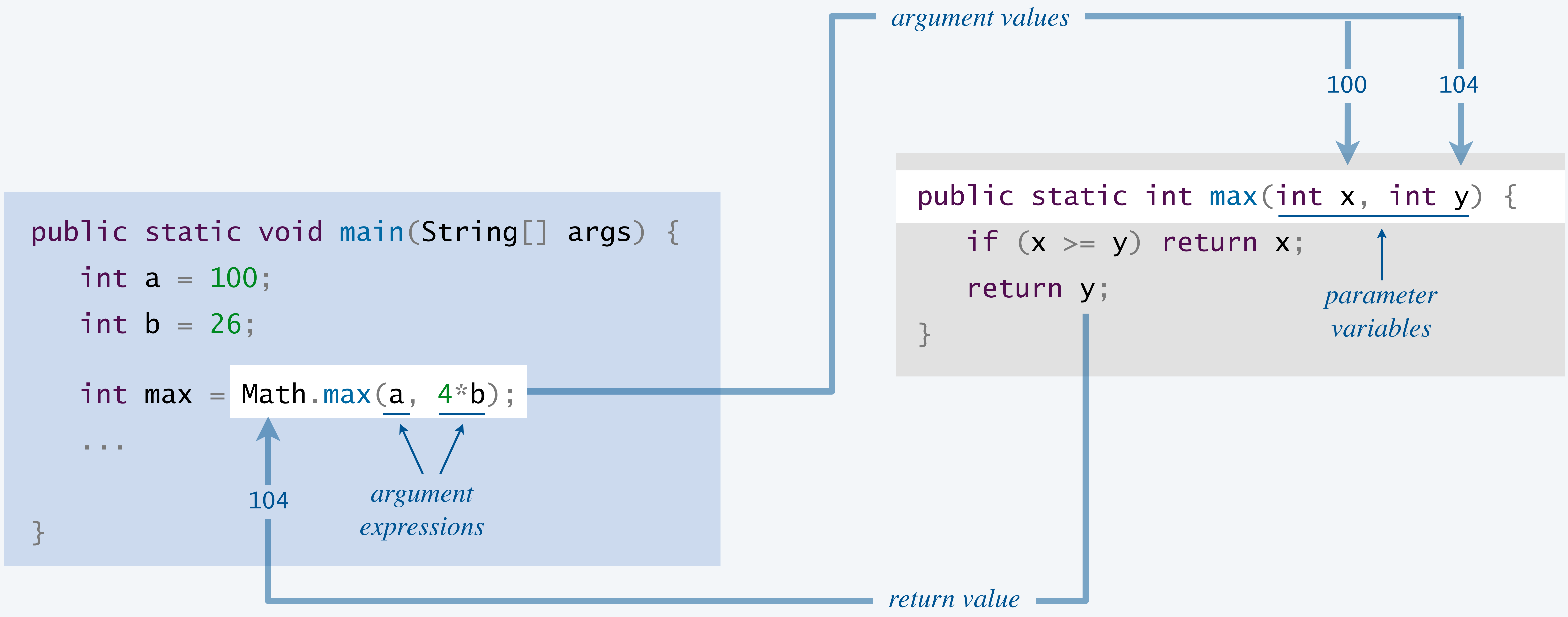
- *flow-of-control*
- *properties*
- *call by value*
- *number-to-speech*

Call by value

Java uses call by value to pass arguments to methods.

- Java evaluates each argument expression to produce a value.
- Java assigns each value to the corresponding parameter variable.

for primitive types, the value is the data-type value;
for arrays (and other non-primitive types),
the value is an "object reference"



What does the following program print?

A. -126

B. 126

C. Compile-time error.

D. Run-time error.

```
public class Mystery1 {

   public static void negate(int a) {
      a = -a;
   }

   public static void main(String[] args) {
      int a = 126;
      negate(a);
      StdOut.println(a);
   }
}
```

negate() cannot change the value stored in the variable a in main()

primitive variable in *main()*

primitive variable in *negative()*

What does the following program print?

A. 12 6

B. -12 -6

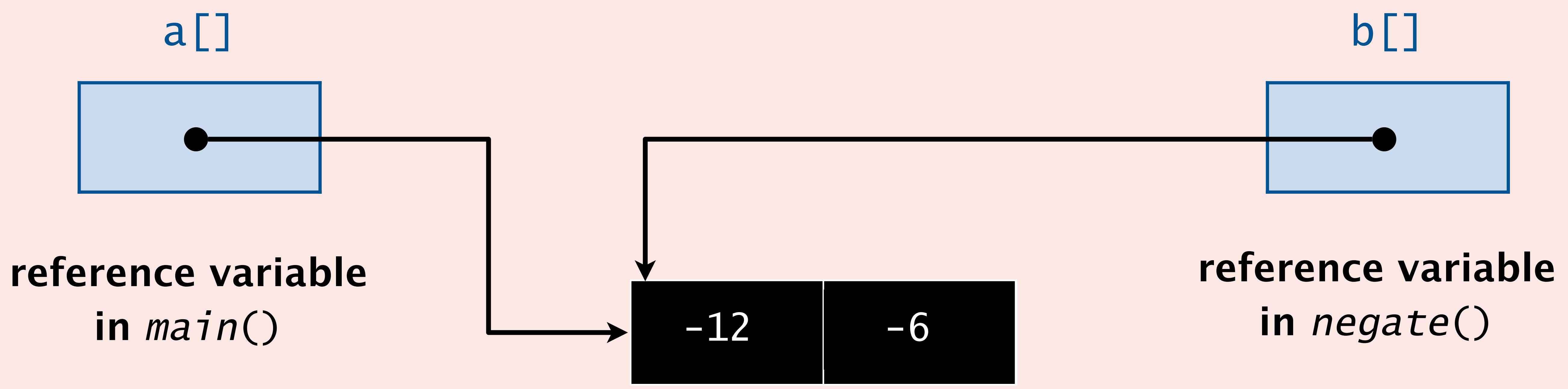
C. Compile-time error.

D. Run-time error.

```
public class Mystery2 {

   public static void negate(int[] b) {
      for (int i = 0; i < b.length; i++)
         b[i] = -b[i];
   }

   public static void main(String[] args) {
      int[] a = { 12, 6 };
      negate(a);
      StdOut.println(a[0] + " " + a[1]);
   }
}
```

negate() cannot change the value stored in the variable a[] in main() (e.g., length or type of a[])

but negate() can change the array elements that a[] references



Side effects with arrays

Functions and arrays.

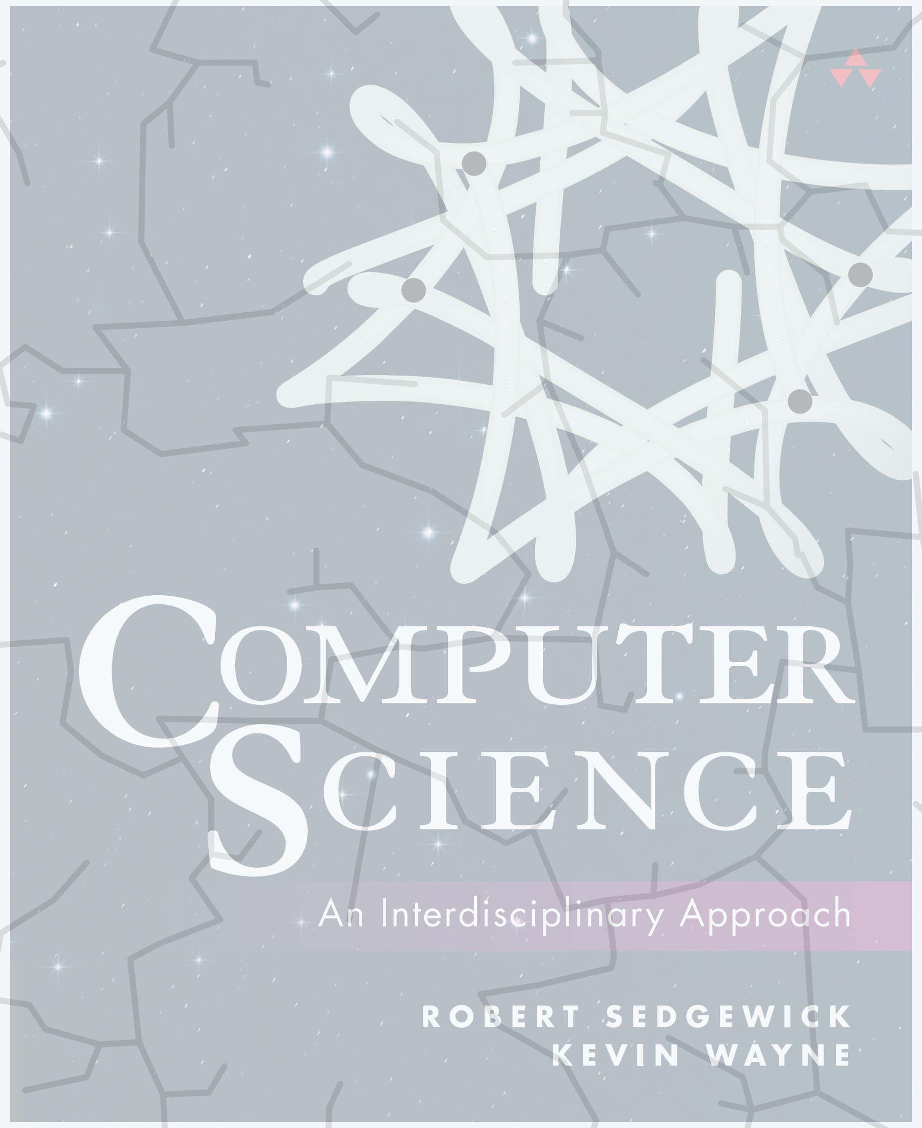
- A function can have the side effect of changing the elements in an argument array. ← *shuffle, reverse, sort, shift, ...*
- But the function cannot change the argument array itself. ← *to refer to a different array (e.g., of a different length or type)*

```
public class Mutate {

   public static void shuffle(String[] a) {
      int n = a.length;
      for (int i = 0; i < n; i++) {
         int r = (int) (Math.random() * (i+1));
         String temp = a[r];
         a[r] = a[i];
         a[i] = temp;
      }
   }

   public static void main(String[] args) {
      shuffle(args);
      for (int i = 0; i < args.length; i++)
         StdOut.println(args[i]);
   }
}
```

a[] and args[] refer to the same array

swaps a[r] and a[i]

```
~/cos126/functions> java-introcs Mutate A B C D
C
A
B
D

~/cos126/functions> java-introcs Mutate A B C D
B
A
C
D

~/cos126/functions> java-introcs Mutate COS 126
126
COS
```



https://introcs.cs.princeton.edu

2.1 Functions

- *flow-of-control*
- *properties*
- *call by value*
- *number-to-speech*

Number-to-speech

Goal. Write a program to say/print a positive integer. ← *use U.S. conventions*

Place Value											
Billions			Millions			Thousands			Ones		
2	8	6	4	0	1	0	0	0	4	5	6

Algorithm.

- Split into 3-digit groups, from right-to-left.
- For each group, from left-to-right:
  - say 3-digit integer ← *see algorithm on next slide*
  - if 3-digit integer is not 0, say group name (*billion*, *million*, *thousand*)

↑ *but not for ones group*

number	spoken
126	*one hundred twenty six*
2,024	*two thousand twenty four*
401,000,011	*four hundred one million eleven*

Number-to-speech: procedural decomposition

Small-integer rule. If number is 1–19, say number; if 0, say nothing.

Two-digit rule.

- If number is 0–19, say number. ← *small-integer rule*
- Otherwise, break up into tens and ones digits.
  - say tens digit as *twenty*, *thirty*, …, *ninety*
  - say ones digit ← *small-integer rule*

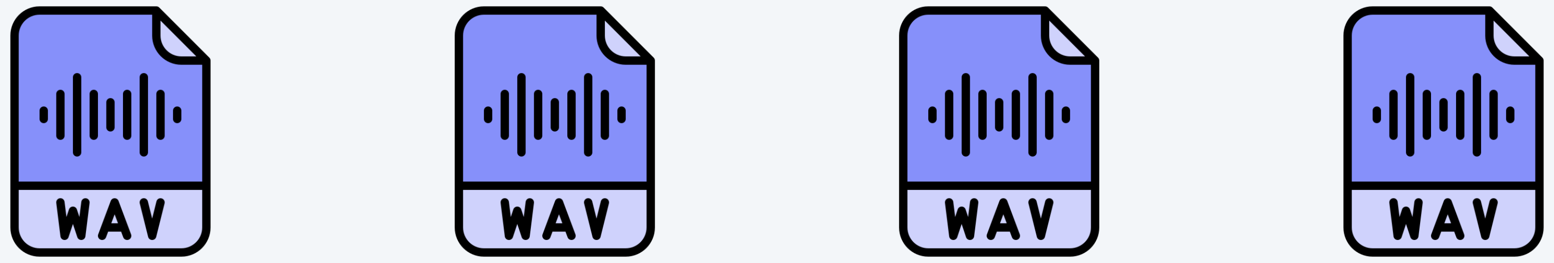
Three-digit rule. Break up into hundreds digit and 2-digit remainder.

- If hundreds digit is not 0, say digit, followed by *hundred*. ← *small-integer rule*
- Say 2-digit remainder. ← *two-digit rule*

number	spoken
6	*six*
0	[*nothing*]
26	*twenty six*
126	*one hundred twenty-six*

Text-to-speech approach

Domain-specific synthesis. Concatenate pre-recorded words to form desired output.



1.wav hundred.wav 20.wav 6.wav

speaking the number 126

word	audio file
1, 2, 3, …, 19	`1.wav`, `2.wav`, `3.wav`, …
20, 30, 40, …, 90	`20.wav`, `30.wav`, `40.wav`, …
hundred	`hundred.wav`
thousand	`thousand.wav`
million	`million.wav`
billion	`billion.wav`

vocabulary

Live coding

```java
public class SayNumber {

   // play audio file corresponding to word
   public static void sayWord(String word) {
      StdOut.print(word + " ");
      StdAudio.play(word + ".wav");
   }

   // say integer n for 1-19, nothing for 0
   public static void saySmallInteger(int n) {
      if (n > 0) sayWord("" + n);
   }

   // say integer n for 1-99, nothing for 0
   public static void sayTwoDigitInteger(int n) {
      if (n < 20) saySmallInteger(n);
      else {
         int tensDigit = n / 10;
         int onesDigit = n % 10;
         sayWord("" + (10 * tensDigit));
         saySmallestInteger(onesDigit);
      }
   }

   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]);
      sayPositiveInteger(n);
   }

   // say integer n for 1-999, nothing for 0
   public static void sayThreeDigitInteger(int n) {
      int hundredsDigit = n / 100;
      int twoDigits = n % 100;
      if (hundredsDigit > 0) {
         saySmallInteger(hundredsDigit);
         sayWord("hundred");
      }
      sayTwoDigitInteger(twoDigits);
   }

   // say integer n > 0
   public static void sayPositiveInteger(int n) {
      String[] PLACES = { "", "thousand", "million", "billion" };
      int[] groups = new int[PLACES.length];

      for (int i = 0; i < groups.length; i++) {
         groups[i] = n % 1000;
         n = n / 1000;
      }

      for (int i = groups.length - 1; i >= 0; i--) {
         sayThreeDigitInteger(groups[i]);
         if (i > 0 && groups[i] > 0) sayWord(PLACES[i]);
      }
   }
}
```

extract the 3-digit groups (right-to-left)

process 3-digit groups (left-to-right)

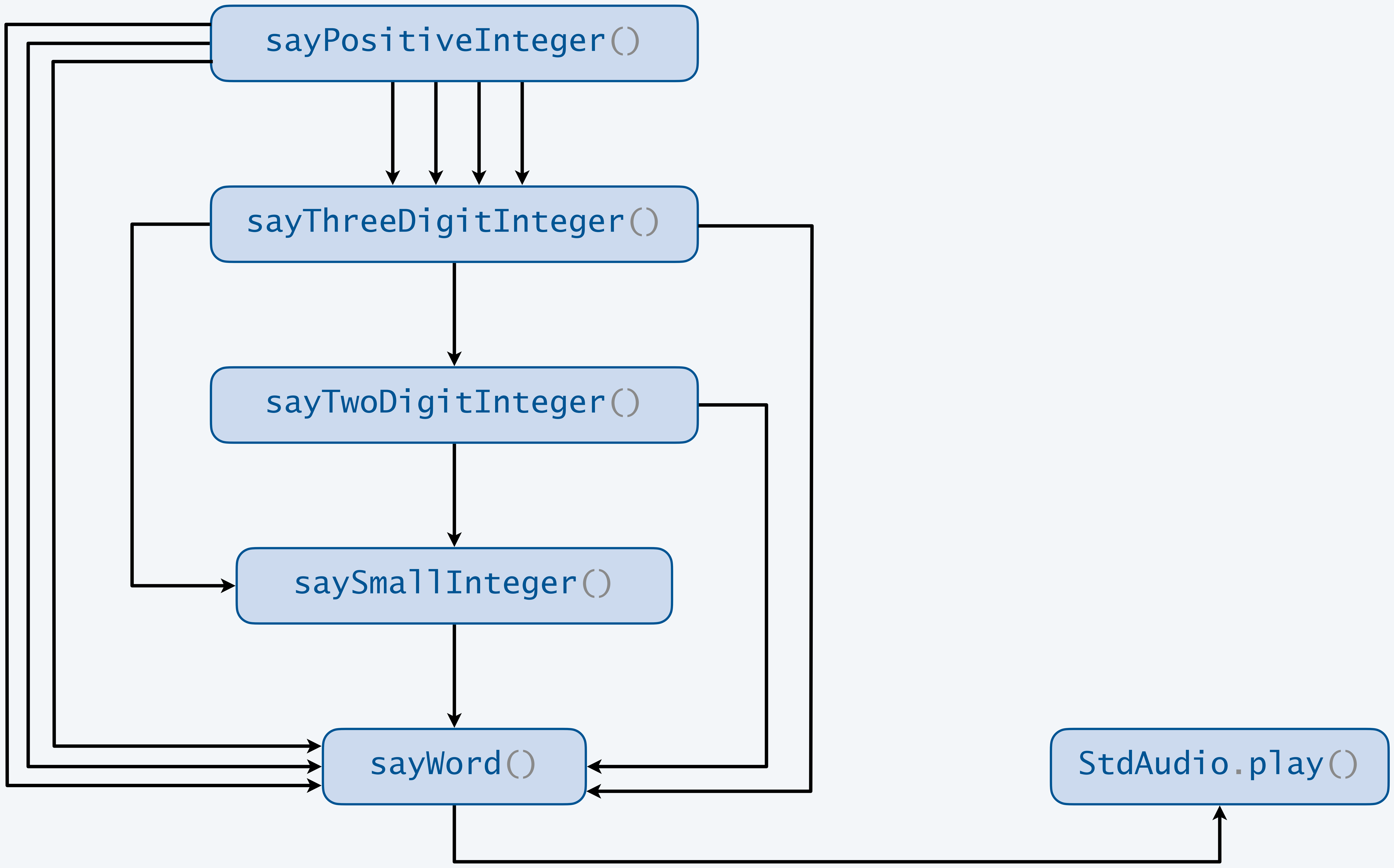
Testing

Principle. Supply inputs that activate all possible execution paths through program. ← *so that all code gets tested*

```
~/cos126/functions> java-introcs SayNumber 6          ← one-digit number
[speaks "six"]

~/cos126/functions> java-introcs SayNumber 26         ← two-digit number
[speaks "twenty six"]

~/cos126/functions> java-introcs SayNumber 126        ← three-digit number
[speaks "one hundred twenty six"]

~/cos126/functions> java-introcs SayNumber 2024       ← typical case
[speaks "two thousand twenty four]


~/cos126/functions> java-introcs SayNumber 401000011  ← no thousands unit
[speaks "four hundred one million eleven"]
```

Function call graph

Function call graph. Graphical representation of different function calls within a program.



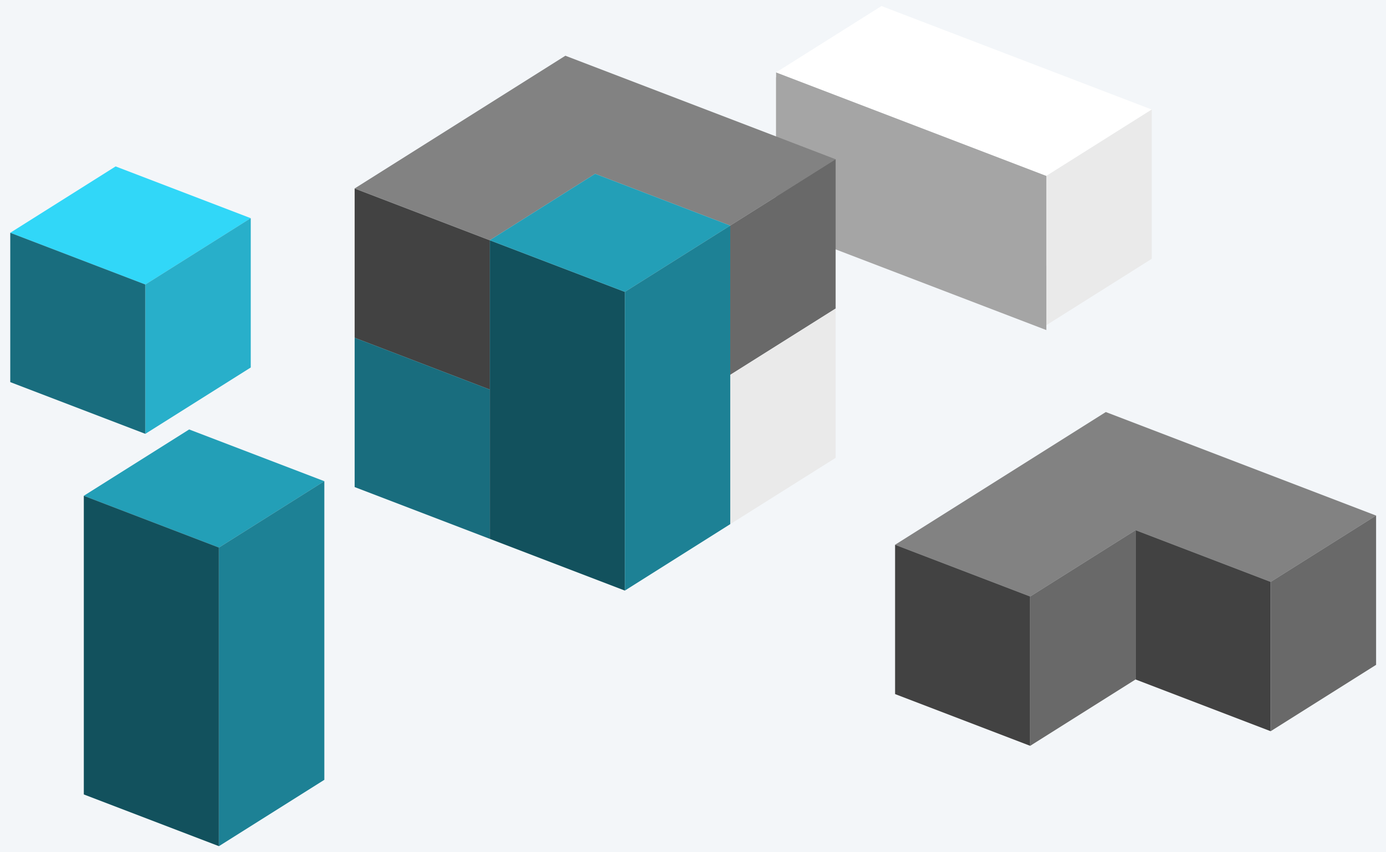
Procedural decomposition

Decomposition. Break up a complex programming problem into smaller functional parts.

Procedural decomposition. Implement each part as a separate function.

Ex. Say a positive integer.

- Play an audio file corresponding to a word.
- Say a small integer.
- Say a two-digit integer.
- Say a three-digit integer.



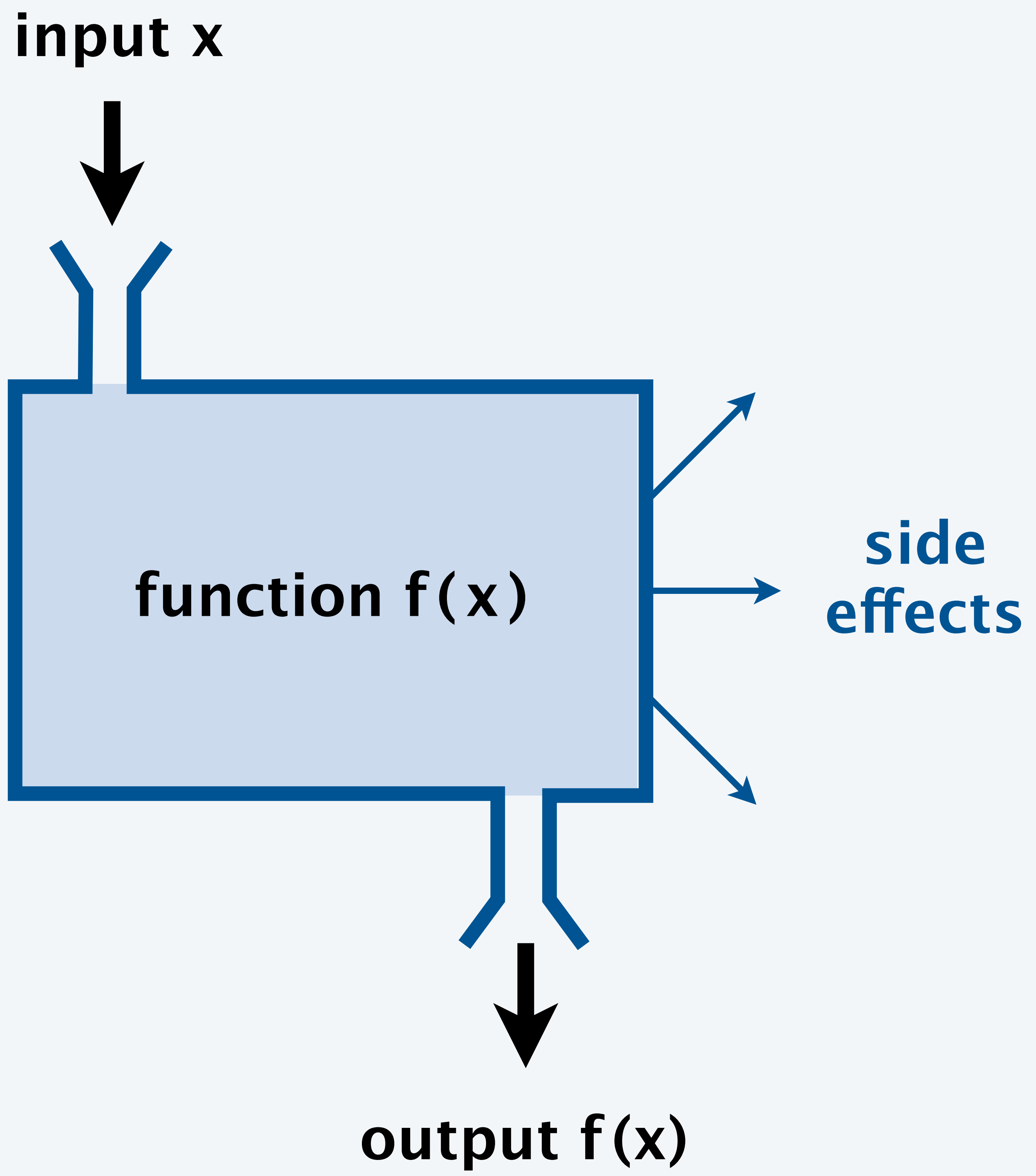
Benefits. Supports the 3 Rs:

- Readability: understand and reason about code.
- Reliability: test, debug, and maintain code.
- Reusability: reuse and share code.

Summary

Functions. Provide a fundamental way to change flow of control of program.

- Java evaluates the arguments and passes by value to function.
- Function initializes parameter variables with corresponding argument values.
- Function computes a single return value and returns it to caller.



Applications.

- Scientists use mathematical functions to calculate formulas.
- Programmers use functions to build modular programs.
- You use functions for both.

This lecture. Write your own functions.

Next lecture. Build reusable libraries of functions.

Credits

media	source	license
Gears	Adobe Stock	education license
Function Gradient	Adobe Stock	education license
Function Machine	Wvbailey	public domain
Normal Distribution	Adobe Stock	education license
Chemotherapy Side Effects	Adobe Stock	education license
Decimal Place Value	Adobe Stock	education license
Code Testing	SAMDesigning	education license
SpongeBob SquarePants Multitasking	Giphy	
Modular Design	Modular Management	