Introduction to Programming

Week 6

Magnus Madsen

Week 6: Functions and Libraries

Monday

- Mathematical functions
- Functions in Java

Thursday

- Pass by value vs. pass by reference
- Overloading
- Live programming

TA classes for ITE

The TA classes for ITE (IT Product Development) have been moved to **10.15 to 12.00** in **Nygaard (5335-184)** on Wednesdays 😊

Prologue

Quote of the Week

"Programs must be written for people to read, and only incidentally for machines to execute."

— Abelson & Sussman

Epigram of the Week

"There will always be things we wish to say in our programs that in all known languages can only be said poorly."

— Alan Perlis

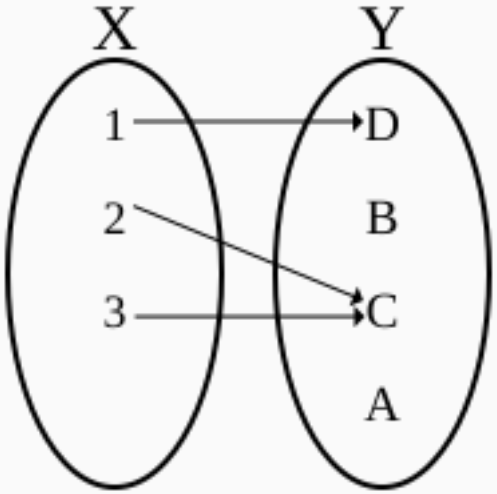
Mathematical Functions

Functions are a super power!

What is a function?

In mathematics, a **function** is a *relation* in which each element of the *domain* is associated with exactly one element of the *codomain*.

In programming, we more broadly think of functions as mappings from input to output.



i In programming we have to distinguish between **pure** mathematical functions and **impure** functions that may have side-effects.

A **pure function** is one that always returns the same output when given the same inputs.

A **pure function** is **referentially transparent**.

That is, the program on the left should be equivalent to the program on the right:

```
var x = f(42);
var y = x;
```

```
var x = f(42);
var y = f(42);
```

Example: An impure function

The function `System.out.println` always returns the same value (i.e. `void`).

But is it **referentially transparent**? That is, are these programs the same?

```
var x = System.out.println("A");
var y = x;
```

```
var x = System.out.println("A");
var y = System.out.println("A");
```

(**Note**: The above is not technically valid Java – but ignore that for now.)

Examples

We have already seen several pure functions:

- `Math.abs` – returns the absolute value of a number.
- `Math.max` – returns the larger of two values.
- `Math.min` – returns the smaller of two values.
- `String.length` – returns the number of characters in a string.
- `Character.isDigit` – returns true if the character is a digit.

Examples

We have already seen several pure functions:

- `Math.abs` – returns the absolute value of a number.
- `Math.max` – returns the larger of two values.
- `Math.min` – returns the smaller of two values.
- `String.length` – returns the number of characters in a string.
- `Character.isDigit` – returns true if the character is a digit.

We have also seen several impure functions:

- `System.out.println` – prints a string to the terminal.
- `StdIn.readString` – reads input from the user.
- `StdOut.print` – prints text without a newline.
- `StdDraw.line` – draws a line on the canvas.

Q:

Are these functions (a) pure or (b) impure?

- `StdIn.readDouble`
- `StdDraw.circle`
- `Math.sqrt`
- `StdOut.println`
- `Math.random`
- `Integer.parseInt`

Functions in Java

What is a Java function?

In Java a function is called a *static method*. (We will get into the details of "why" later.)

The mathematical function $f(x, y) = x + y$ is written as:

```
static int f(int x, int y) {
    return x + y;
}
```

What is a Java function?

In Java a function is called a *static method*. (We will get into the details of "why" later.)

The mathematical function $f(x, y) = x + y$ is written as:

```
static int f(int x, int y) {
    return x + y;
}
```

The mathematical function $f(x) = |x^2|$ is written as:

```
static int f(int x) {
    return Math.abs(x * x);
}
```

We can write any computable function in Java.

Functions: Terminology

The Java function:

```
static int add(int x, int y) {
    return x + y;
}
```

consists of the following parts:

- **Function signature**: A broad name for the following components:
  - **Function name**: The identifier used to call the function (`add` in this example).
  - **Formal parameter list**: The complete list of parameters `(int x, int y)`.
    - **Formal parameter**: A variable declared in the function definition (`x` and `y`).
  - **Return type**: The type of value the function returns (`int` in this example).
- **Function body**: The code between the braces `{ return x + y; }`.

Programmers love to argue about names:

- **Function**: What mathematicians call them.
- **Method**: What Java programmers call them.
- **Procedure**: What Pascal programmers called them.
- **Subroutine**: What FORTRAN programmers called them in the 1950s.
- **Subprogram**: What Ada programmers call them.

They're all basically the *same thing*: reusable pieces of code! 😊

Q: What is the difference between an *identifier* and a *keyword*?

Why Java functions?

Q: Why do we need methods? Aren't variables, loops, and arrays enough?

Why Java functions?

Q: Why do we need methods? Aren't variables, loops, and arrays enough?

- **Software is too big:** we have to break it into **manageable pieces**.
  - We can use functions to decompose and structure our code.

Why Java functions?

Q: Why do we need methods? Aren't variables, loops, and arrays enough?

- **Software is too big:** we have to break it into **manageable pieces**.
  - We can use functions to decompose and structure our code.
- **Software takes too long to write:** functions allow us to **reuse code**.
  - We don't have to scratch from scratch.
  - We can stand on the shoulders of giants.

Reusability

If we have to use a formula or algorithm frequently, it's convenient to make a function for it.

Here, we use the distance formula several times: $d = \texttt{sqrt}\left\{(x_2 - x_1)^2 + (y_2 - y_1)^2\right\}$

```
public static void main(String[] args) {
    double dist1 = Math.sqrt((3 - 0) * (3 - 0) + (4 - 0) * (4 - 0));
    StdOut.println("(0, 0) to (3, 4): " + dist1);

    double dist2 = Math.sqrt((7 - 1) * (7 - 1) + (10 - 2) * (10 - 2));
    StdOut.println("(1, 2) to (7, 10): " + dist2);

    double dist3 = Math.sqrt((8 - 5) * (8 - 5) + (9 - 5) * (9 - 5));
    StdOut.println("(5, 5) to (8, 9): " + dist3);
}
```

Reusability

We can avoid repeating the formula by writing a function:

```
static double distance(double x1, double y1, double x2, double y2) {
    return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

public static void main(String[] args) {
    StdOut.println("(0, 0) to (3, 4): "  + distance(0, 0, 3, 4));
    StdOut.println("(1, 2) to (7, 10): " + distance(1, 2, 7, 10));
    StdOut.println("(5, 5) to (8, 9): "  + distance(5, 5, 8, 9));
}
```

A well-structured book is divided into **chapters** which are divided into **paragraphs** which are divided into **sentences**.

A well-structured program: Many small functions

A well-structured book is divided into **chapters** which are divided into **paragraphs** which are divided into **sentences**.

A well-structured program is divided into **modules** which are divided into **functions** which are divided into **statements** (the function bodies).

Separation of Concerns

Functions enable **separation of concerns**:

Separation of concerns is the "process of separating a computer program into distinct features that overlap functionally as little as possible." *via Wiktionary*

The key is that each function can focus on doing *one thing well*.

- Then we can combine functions to make the full program.

Separation of Concerns

Functions enable **separation of concerns**:

Separation of concerns is the "process of separating a computer program into distinct features that overlap functionally as little as possible." *via Wiktionary*

The key is that each function can focus on doing *one thing well*.

- Then we can combine functions to make the full program.

For example, if we write a student grade calculator, we might have:

- one function to read student scores from a file,
- one function to calculate the average grade, and
- one function to display the results in a formatted report

Example 1: A mathematical function

We can write a function to calculate the compound interest for a given principal amount, annual interest rate, and number of years:

```
static double compoundInterest(double principal, double rate, int years) {
    return principal * Math.pow(1 + rate, years);
}
```

Example 1: A mathematical function

We can write a function to calculate the compound interest for a given principal amount, annual interest rate, and number of years:

```
static double compoundInterest(double principal, double rate, int years) {
    return principal * Math.pow(1 + rate, years);
}
```

Upshot: We don't have to remember the complicated formula.

Example 1: A mathematical function

We can write a function to calculate the compound interest for a given principal amount, annual interest rate, and number of years:

```
static double compoundInterest(double principal, double rate, int years) {
    return principal * Math.pow(1 + rate, years);
}
```

Upshot: We don't have to remember the complicated formula.

Upshot: If the formula is wrong, we only have to change it in one place.

Example 1: A mathematical function

We can write a function to calculate the compound interest for a given principal amount, annual interest rate, and number of years:

```
static double compoundInterest(double principal, double rate, int years) {
    return principal * Math.pow(1 + rate, years);
}
```

Upshot: We don't have to remember the complicated formula.

Upshot: If the formula is wrong, we only have to change it in one place.

Upshot: If someone discovers a *better* formula they can change the function and have the change reflected *everywhere*.

Example 2: Draw a triangle

We can write a function to draw a triangle with the bottom-left corner at `(x, y)`:

```
static void drawTriangle(double x, double y, double size) {
    double[] xCoords = new double[]{x, x + size, x + size / 2};
    double[] yCoords = new double[]{y, y, y + size * Math.sqrt(3) / 2};
    StdDraw.polygon(xCoords, yCoords);
}
```

Example 2: Draw a triangle

We can write a function to draw a triangle with the bottom-left corner at `(x, y)`:

```
static void drawTriangle(double x, double y, double size) {
    double[] xCoords = new double[]{x, x + size, x + size / 2};
    double[] yCoords = new double[]{y, y, y + size * Math.sqrt(3) / 2};
    StdDraw.polygon(xCoords, yCoords);
}
```

Upshot: We can work at a higher level of abstraction.

Example 3: Draw a rectangle

We can write a function to draw a rectangle with the bottom-left corner at `(x, y)` using more intuitive parameters than the standard `StdDraw.rectangle` method:

```
static void drawRectangle(double x, double y, double width, double height) {
    StdDraw.rectangle(x + width / 2, y + height / 2, width / 2, height / 2);
}
```

Upshot: We can design more ergonomic APIs.

Example: A Picture Says a Thousand Words

A class that uses several methods to draw a beautiful image:

```
public class Picture {
    static void drawHouse(double x, double y) {
        // ...
    }

    static void drawTree(double x, double y) {
        // ...
    }

    static void drawSun(double x, double y) {
        // ...
    }

    public static void main(String[] args) {
        // ...
    }
}
```

Example: A Picture Says a Thousand Words

The complete implementation using various StdDraw methods:

```
public class Picture {
    static void drawHouse(double x, double y) {
        StdDraw.setPenColor(216, 131, 148);  // Dusty rose
        StdDraw.filledRectangle(x, y, 0.15, 0.1);
        StdDraw.setPenColor(119, 158, 203);  // Muted blue
        StdDraw.filledPolygon(new double[]{x - 0.2, x, x + 0.2},
                              new double[]{y + 0.1, y + 0.25, y + 0.1});
    }

    static void drawTree(double x, double y) {
        StdDraw.setPenColor(160, 130, 95);  // Muted brown
        StdDraw.filledRectangle(x, y, 0.02, 0.05);
        StdDraw.setPenColor(119, 168, 119);  // Sage green
        StdDraw.filledCircle(x, y + 0.08, 0.06);
    }

    ...
```

Example: A Picture Says a Thousand Words

```
    static void drawSun(double x, double y) {
        StdDraw.setPenColor(240, 220, 130);  // Soft gold
        StdDraw.filledCircle(x, y, 0.08);
        StdDraw.setPenColor(220, 190, 120);  // Warm sand
        StdDraw.setPenRadius(0.005);
        for (int i = 0; i < 8; i++) {
            double angle = i * Math.PI / 4;
            StdDraw.line(x, y, x + 0.12 * Math.cos(angle), y + 0.12 * Math.sin(angle));
        }
    }

    public static void main(String[] args) {
        drawHouse(0.5, 0.3);
        drawTree(0.2, 0.2);
        drawTree(0.8, 0.25);
        drawTree(0.35, 0.18);
        drawSun(0.85, 0.8);
        drawSun(0.15, 0.75);
    }
}
```

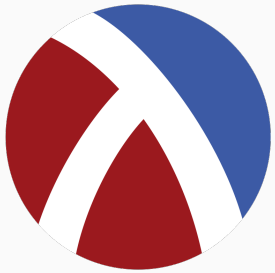
Q:

- Is `drawSun` pure?
- Is `drawSun` idempotent?

Functional programming

Functions are such a powerful concept that entire programming languages are built around them.

But not in this course 😔



Haskell *Racket*

OCaml *Flix*

Calls and Returns

Reminder: Control Flow

Control flow is "the order in which individual statements, instructions, or function calls of an imperative program are executed or evaluated." *via Wiktionary*

We have already seen `if` and `while` statements as a form of control flow.

Reminder: Control Flow

✏️ **Control flow** is "the order in which individual statements, instructions, or function calls of an imperative program are executed or evaluated." *via Wiktionary*

We have already seen `if` and `while` statements as a form of control flow.

Functions are a new and powerful form of control flow.

ℹ️ Functions allow you to package code into reusable units that can be called from anywhere in your program.

Calling a function

Given the two methods `compoundInterest` and `main`:

```
static double compoundInterest(double principal, double rate, int years) {
    return principal * Math.pow(1 + rate, years);
}

public static void main(String[] args) {
    double p = Double.parseDouble(args[0]);
    double amount = compoundInterest(p, 0.05, 10);
    System.out.println("$" + p + " becomes $" + amount);
}
```

At the method **call-site** `compoundInterest(p, 0.05, 10)`, the **arguments** are passed into the **formal parameters** and **control** is transfered to the method.

When the method finishes computation, i.e. **returns**, the **return value** is passed back to the call-site and the computation resumes from there.

Return statement

The **return statement** is how a function sends a value back to its caller.

- Every non-void function **must** have at least one return statement.
- The type of the returned value must match the function's declared return type.
- When a return statement is executed, the function immediately exits.

Example: Multiple Return Statements

A function can have **multiple return statements** with **complex control flow**:

```
static String grade(int score) {
    if (score >= 90) {
        return "Excellent";
    } else if (score >= 80) {
        return "Good";
    } else if (score >= 70) {
        return "Satisfactory";
    } else if (score >= 60) {
        return "Pass";
    } else {
        return "Fail";
    }
    // We never get here.
}
```

Example: Returns from within loops

We can use **return statements** inside loops for **early exit**:

```
static boolean contains(int element, int[] array) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == element) {
            return true;  // Found it! Exit immediately
        }
    }

    return false;  // Not found after checking all elements.
}
```

Q: Can a function call an impure function and be pure itself?

Void Functions

Void functions

Java requires that every method has a a **return type**.

- But not every method needs to return a value.
- Functions that don't return any value have the return type `void`.

Example: Void function

Here is a function that prints `text` three times:

```
static void printThreeTimes(String text) {
    for (int i = 0; i < 3; i++) {
        StdOut.println(text);
    }
}
```

Note: A `void` method does not need a `return` statement. All other methods do.

Example: Early return in void function

A `void` function can use **return** to **exit early** without returning a value:

```
static void printGreeting(String name) {
    if (name.isEmpty()) {
        System.out.println("Hello, stranger!");
        return;  // Exit early
    }

    System.out.println("Hello, " + name + "!");
    System.out.println("Welcome to the program!");
}
```

Q: Does it make sense to have a pure function with return type `void`?

Argument passing

Primitive and reference types

In Java, we distinguish between **primitive types** and **reference types**.

A **primitive type** is a "data type provided by a programming language as a basic building block, such as integer, character or Boolean."

via Wiktionary

A **reference type** is a "data type that represents a reference to a value, as opposed to the value itself."

via Wiktionary

In general:

- Every type starting with a lowercase letter is a **primitive type**. (`int`, `boolean`, `double`, ...)
- Every type starting with an uppercase letter is a **reference type**. (`String`, ...)
- Arrays are always **reference type**s.

Reference types are often *mutable*, meaning their contents can be changed.

Reminder: mutating values vs. reassigning variables

❗ Mutating a **value** is different from reassigning to a **variable**.

Here, we reassign to the **variable** `a1`. We change it to point to a new array. Variable `a2` points to the old array.

Output: `[4, 5, 6]`, `[1, 2, 3]`

```
int[] a1 = new int[]{1, 2, 3};
int[] a2 = a1;
a1 = new int[]{ 4, 5, 6 };
System.out.println(Arrays.toString(a1));
System.out.println(Arrays.toString(a2));
```

Here, we mutate the **value** in `a1`. We change the contents of the array. Variable `a2` points to the same array.

Output: `[4, 2, 3]`, `[4, 2, 3]`

```
int[] a1 = new int[]{1, 2, 3};
int[] a2 = a1;
a1[0] = 4;
System.out.println(Arrays.toString(a1));
System.out.println(Arrays.toString(a2));
```

Q: What is an example of a reference type that is immutable?

Argument passing

i When we pass **reference type** arguments to a function, we create new variables that point to the *same values*.

! Hence, if we mutate a function argument those changes are visible *outside* the function as well.

Example: Mutating a reference type

```
static void modifyArray(int[] arr) {
    arr[0] = 999;
    StdOut.println("Inside function: " + arr[0]);
}

public static void main(String[] args) {
    int[] numbers = new int[]{1, 2, 3};
    StdOut.println("Before: " + numbers[0]);
    modifyArray(numbers);
    StdOut.println("After: " + numbers[0]);
}
```

Output: `Before: 1`, `Inside function: 999`, `After: 999`

The original array is modified because `arr` and `args` point to the same array.

Example: Reassigning to a reference type

```
static void modifyArray(int[] arr) {
    arr = new int[]{4, 5, 6};
    StdOut.println("Inside function: " + arr[0]);
}

public static void main(String[] args) {
    int[] numbers = new int[]{1, 2, 3};
    StdOut.println("Before: " + numbers[0]);
    modifyArray(numbers);
    StdOut.println("After: " + numbers[0]);
}
```

Output: `Before: 1`, `Inside function: 4`, `After: 1`

The original array is **not** modified because we *reassigned* to the variable `arr`.

Defensive copy

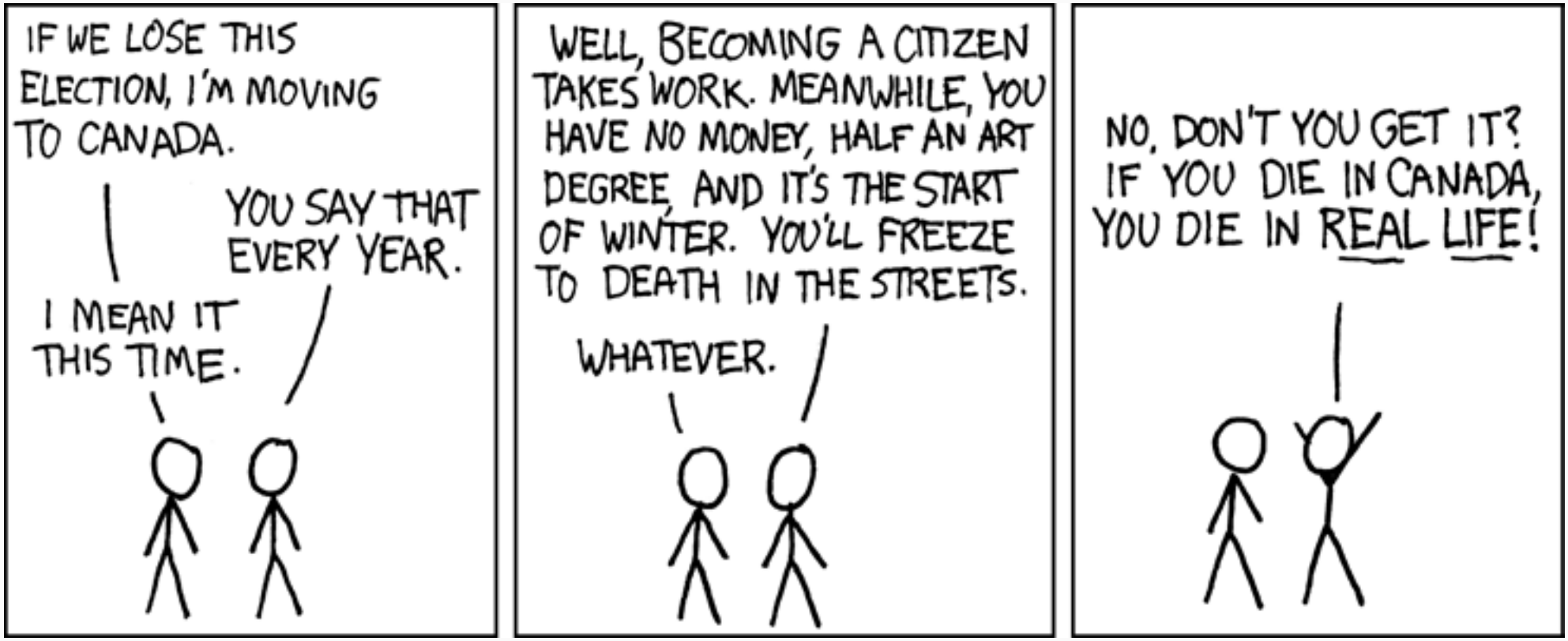
To avoid unintended modifications, we can create a **defensive copy**:

```
static void safeModifyArray(int[] arr) {
    int[] copy = arr.clone();  // Create a copy
    copy[0] = 999;
    StdOut.println("Modified copy: " + copy[0]);
}

public static void main(String[] args) {
    int[] numbers = new int[]{1, 2, 3};
    StdOut.println("Before: " + numbers[0]);
    safeModifyArray(numbers);
    StdOut.println("After: " + numbers[0]);
}
```

Output: `Before: 1`, `Modified copy: 999`, `After: 1`

Life lesson

If you mutate in a function, you mutate in real life.



Q:

What is wrong with this Java program?

```
static int sum(int x, int y, int z) {
    int result = x + y + z;
}
```

Q: Can a method with return type `void` contain a `return` statement?

The Call Stack

What is the call stack?

The **call stack** is a data structure that tracks which methods are currently executing and stores information about function calls, including parameters and local variables.

When a function is called:

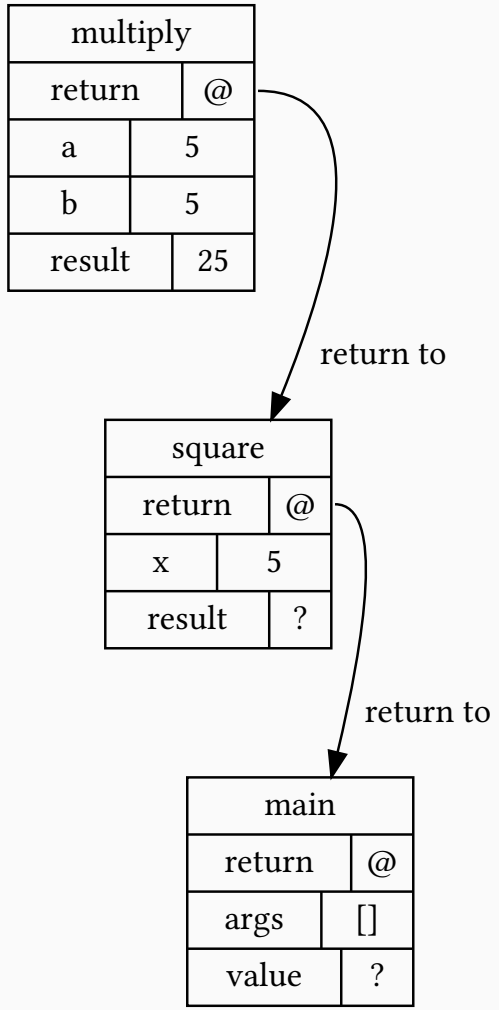
1. A new **stack frame** is created and pushed onto the call stack
2. The frame contains the function's parameters and local variables
3. When the methods returns, its frame is popped from the stack

Example: Understanding the call stack

This stack has 3 frames for its 3 calls.

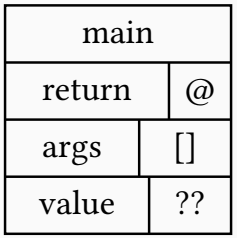
Each frame tracks where it must return.

```
public static void main(String[] args) {
    int value = square(5);
    StdOut.println("5 squared is: " + value);
}

static int square(int x) {
    return multiply(x, x);
}

static int multiply(int a, int b) {
    return a * b;
}
```



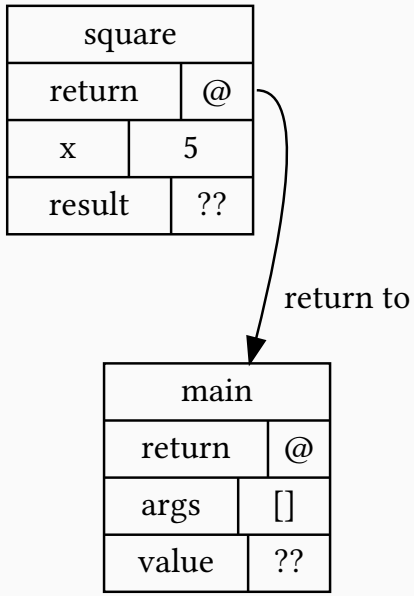
Example: Call stack step-by-step

1. A `main` frame is **allocated**
   - `args` contains the empty array
   - `value` is allocated but not yet known

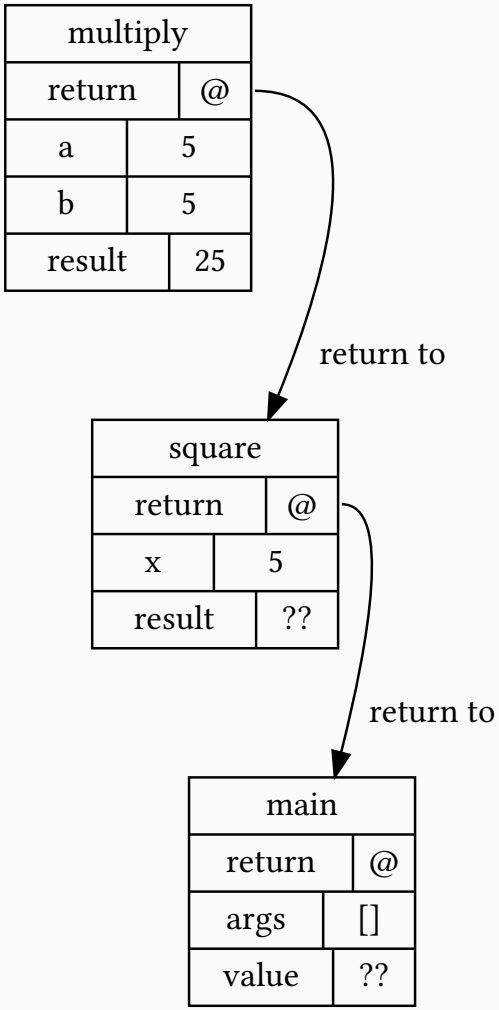


Example: Call stack step-by-step

1. A `main` frame is **allocated**
   - `args` contains the empty array
   - `value` is allocated but not yet known
2. A `square` frame is **allocated**
   - the return address points to `main`
   - `x` contains the argument `5`
   - the result is allocated but not yet known

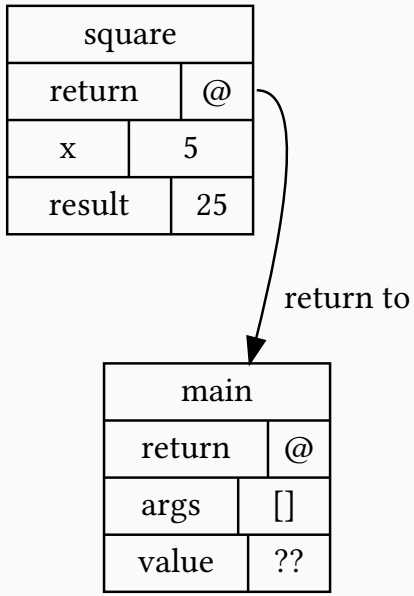


Example: Call stack step-by-step



1. A `main` frame is **allocated**
   - `args` contains the empty array
   - `value` is allocated but not yet known
2. A `square` frame is **allocated**
   - the return address points to `main`
   - `x` contains the argument `5`
   - the result is allocated but not yet known
3. A `multiply` frame is **allocated**
   - the return address points to `square`
   - `a` contains the argument `5`
   - `b` contains the argument `5`
   - the result is calculated as `25`

Example: Call stack step-by-step



1. A `main` frame is **allocated**
   - `args` contains the empty array
   - `value` is allocated but not yet known
2. A `square` frame is **allocated**
   - the return address points to `main`
   - `x` contains the argument `5`
   - the result is allocated but not yet known
3. A `multiply` frame is **allocated**
   - the return address points to `square`
   - `a` contains the argument `5`
   - `b` contains the argument `5`
   - the result is calculated as `25`
4. The `multiply` frame is **deallocated**
   - the result is updated with `25`

Example: Call stack step-by-step

1. A `main` frame is **allocated**
   - `args` contains the empty array
   - `value` is allocated but not yet known
2. A `square` frame is **allocated**
   - the return address points to `main`
   - `x` contains the argument `5`
   - the result is allocated but not yet known
3. A `multiply` frame is **allocated**
   - the return address points to `square`
   - `a` contains the argument `5`
   - `b` contains the argument `5`
   - the result is calculated as `25`
4. The `multiply` frame is **deallocated**
   - the result is updated with `25`
5. The `square` frame is **deallocated**
   - `value` is updated with `25`

main	
return	@
args	[]
value	25

Overloading

What is overloading?

✏️ In Java, **overloading** allows us to use the *same name* for *different* functions.

- We can overload based on **the number of formal parameters** (arity).
- We can overload based on **the type of the formal parameters**.
- We *cannot* overload based on the return type.

Example 1: Argument arity overloading

We can overload based on the number of formal parameters:

```
static int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

static int max(int a, int b, int c) {
    return max(max(a, b), c);
}
```

Example 2: Argument type overloading

We can overload based on the type of the formal parameters:

```
static void printValue(int x) {
    StdOut.println("Integer: " + x);
}

static void printValue(double x) {
    StdOut.println("Double: " + x);
}
```

Example 3: No overloading for return types

We *cannot* overload based on the return type 😔

```
static int getValue() {
    return 42;
}

static double getValue() {
    return 3.14;
}
```

```
java: method getValue() is already defined in class...
```

Ambiguity

We cannot overload methods such that method resolution becomes **ambiguous**.

```
static int f(String s) {
    System.out.println("Hello!");
}

static int f(String s) {
    System.out.println("Goodbye!");
}
```

What would be the meaning of `f("Magnus")`? Hence, Java disallows this.

Q:

True or false?

- A method does not need to have a **return type**.
- A method does not need to have a **formal parameter**.
- A method does not need to have a **return statement**.

Q:

Here is a Java method:

```
static int multiply(int a, int b) {
    return a * b;
}
```

Identify the:

- Method signature
- Method name
- Formal parameter list
- Return type

Libraries

What is a library?

A **library** is "a collection of software routines that provide functionality to be incorporated into or used by a computer program."

via Wiktionary

At this point, a library is a group of functions in their own file.

Putting code with different functionality in different libraries helps us organize code.

The **key idea** is that someone can write a library, and someone else can use it.

Making our own library

We can make our own library to organize and reuse code. Suppose we want an array library.

1. We create a new class `ArrayUtils.java` and put our array functions in it.
2. We refer to all of these functions as `ArrayUtils.xyz` in any other file.

```
public class ArrayUtils {
    public static int count(String[] a, String x) { ... }
}
```

```
public class CountForks {
    public static void main(String[] args) {
        int n = ArrayUtils.count(args, "fork");
        System.out.print("Forks found: " + n);
    }
}
```

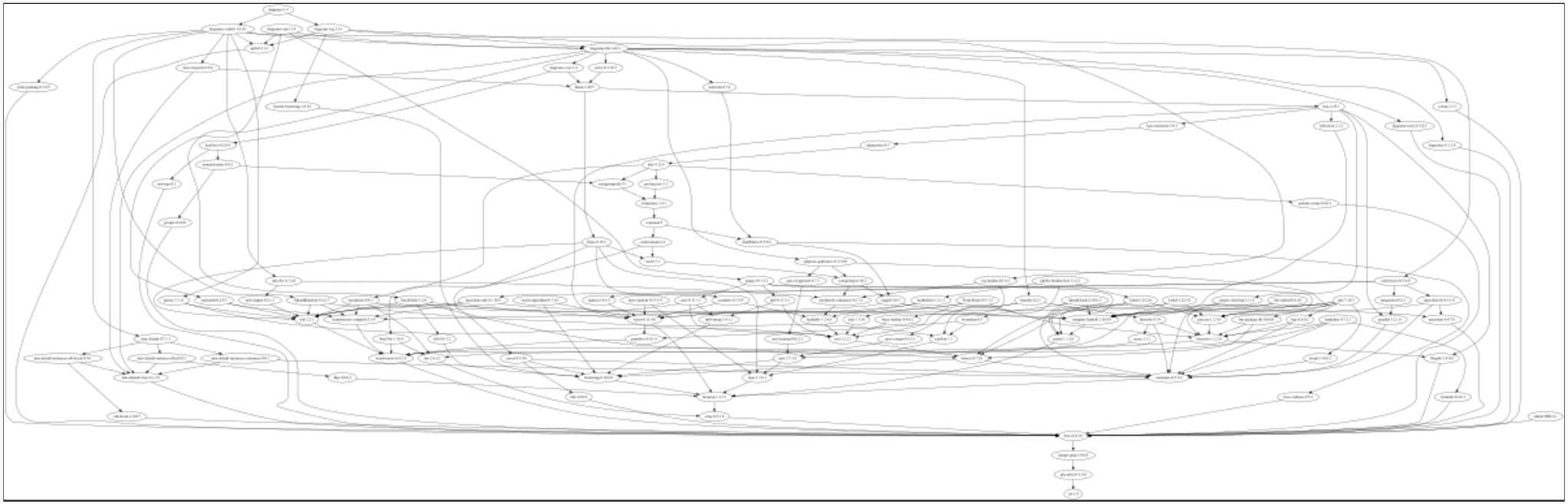
Libraries can build on libraries

The `StdStats` library **depends on** `StdDraw` to create visualizations.

```
public class StdStats {

    public static void plotPoints(double[] a) {
        int n = a.length;
        StdDraw.setXscale(-1, n);
        StdDraw.setPenRadius(0.01);
        for (int i = 0; i < n; i++) {
            StdDraw.point(i, a[i]);
        }
    }
}
```

Libraries can build on libraries

❗ Ideally reuse **simplifies** code, but too much reuse can lead to **complexity**.



Haskell dependency tree

Where do we find libraries

The Java standard library comes bundled with over 5000 classes.

The typical way to find libraries is via a **package manager**, such as *Maven*, *npm*, or *NuGet*, which host millions of libraries.

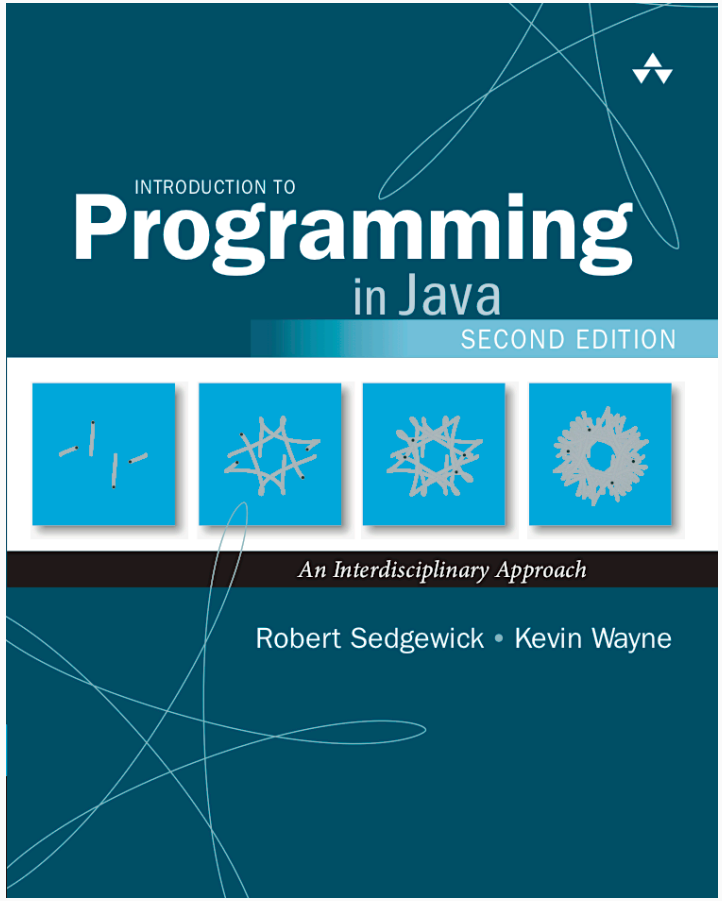
> **i** **Maven** is the main library repository for Java.
>
> *(We will not explore it in this course.)*

Java standard library vs. textbook library

! The **(1)** Java Standard Library and the **(2)** Textbook library are *not* the same thing.

(1) is built into the Java language, available wherever Java is used.

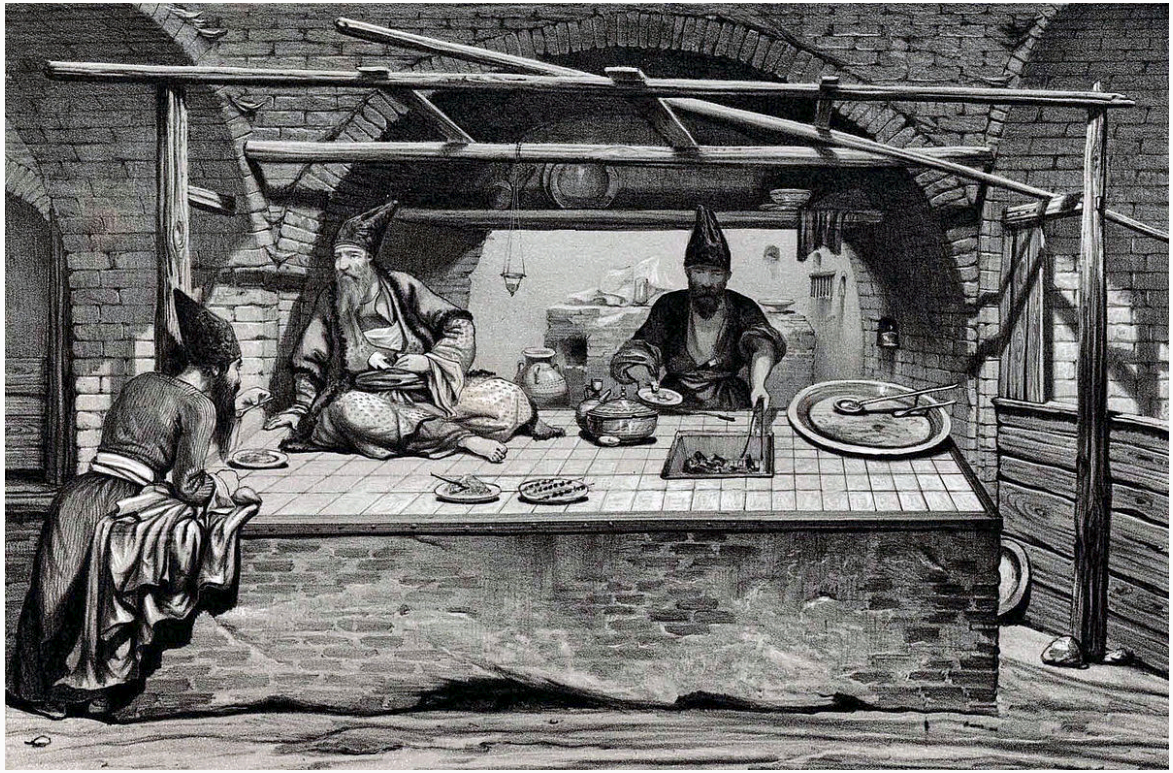
(2) is specific to the book. You must include it separately if you want to use it.



Terminology

- A **library** is a collection of programs designed to be used by other programs.
- A **client** is a program that uses a library.
- An **API** (application programming interface) is a collection of *protocols* for using a library or other programming resource. These are requirements and promises about what the library does and the kind of data that are expected and returned.
- An **implementation** is the code behind the API. It is the actual code behind fulfills the promises of the API.

Terminology



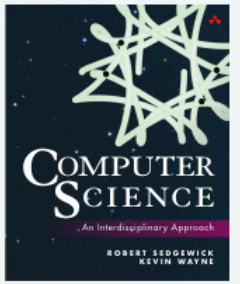
If a **restaurant** is a **library**,
the **customer** is the **client**,
the **menu** is the **API**, and
the **kitchen** is the **implementation**.

The `StdRandom` Library

Example: StdRandom library

Developed for this course, but broadly useful

- Implement methods for generating random numbers of various types.
- Available for download at booksite (and included in introcs software).




```
int getRandomNumber()
{
   return 4;  // chosen by fair dice roll.
              // guaranteed to be random.
}
```

API

public class StdRandom	
int uniform(int N)	*integer between* 0 *and* N-1
double uniform(double lo, double hi)	*real between* lo *and* hi
booean bernoulli(double p)	true *with probability* p
double gaussian()	*normal with mean* 0, *stddev* 1
double gaussian(double m, double s)	*normal with mean* m, *stddev* s
int discrete(double[] a)	i *with probability* a[i]
void shuffle(double[] a)	*randomly shuffle the array* a[]

First step in developing a library: Articulate the API!

Example: Shuffle

We can use `StdRandom.shuffle` to shuffle a deck of cards.

```
public static void main(String[] args) {
    String[] deck = {"A♠", "K♠", "Q♠", "J♠", "A♥", "K♥", "Q♥", "J♥",
            "A♣", "K♣", "Q♣", "J♣", "A♦", "K♦", "Q♦", "J♦"};

    StdRandom.shuffle(deck);

    for (int i = 0; i < deck.length; i++) {
        StdOut.print(deck[i] + " ");
    }
}
```

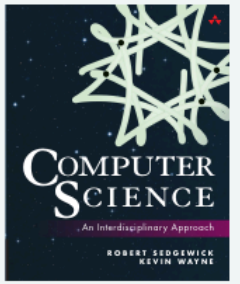
The `StdStats` Library

Example: StdStats library

Developed for this course, but broadly useful

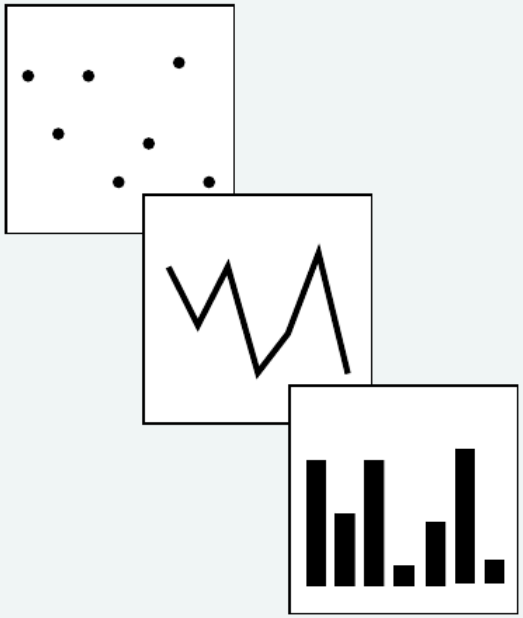
- Implement methods for computing statistics on arrays of real numbers.
- Available for download at booksite (and included in introcs software).

and plotting on StdDraw



API

public class StdStats	
double max(double[] a)	*largest value*
double min(double[] a)	*smallest value*
double mean(double[] a)	*average*
double var(double[] a)	*sample variance*
double stddev(double[] a)	*sample standard deviation*
double median(double[] a)	*plot points at* (i, a[i])
void plotPoints(double[] a)	*plot points at* (i, a[i])
void plotLines(double[] a)	*plot lines connecting points at* (i, a[i])
void plotBars(double[] a)	*plot bars to points at* (i, a[i])



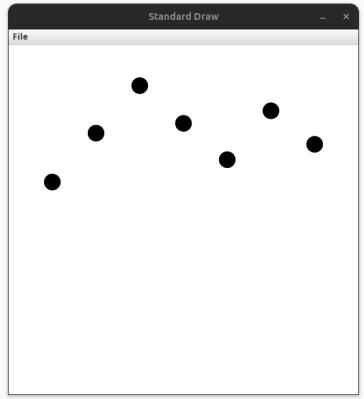
Easy to implement, but easier to use!

← one reason to develop a library: clarify client code

Example: Plot Points

We can use `StdStats.plotPoints` to make a simple graph of the daily temperature.

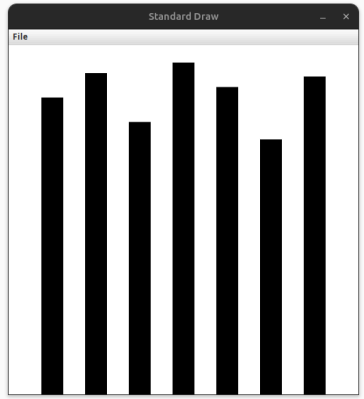
```
public static void main(String[] args) {
    // Daily temperature in Copenhagen (°C) for a week
    double[] temperatures = new double[]{
        15.2, 18.7, 22.1, 19.4, 16.8, 20.3, 17.9
    };

    StdDraw.setYscale(0, 25);
    StdStats.plotPoints(temperatures);
}
```



Example: Plot Bars

We can use `StdStats.plotBars` to make a simple bar chart of student scores.

```
public static void main(String[] args) {
    // Student exam scores (out of 100)
    double[] scores = new double[]{
        85, 92, 78, 95, 88, 73, 91
    };

    StdDraw.setYscale(0, 100);
    StdStats.plotBars(scores);
}
```



Advantages of libraries

Reusability	Write once, use many times across different programs.
Time saving	No need to reinvent the wheel for common functionality.
Quality	Well-tested libraries have fewer bugs than custom implementations.
Expertise	Libraries are often written by experts in specific domains.
Maintainability	Updates and bug fixes benefit all programs using the library.

Disadvantages of libraries

Dependencies	Your program depends on external code you don't control.
Complexity	Adding libraries can make your project more complex to manage.
Security risks	Libraries may contain **vulnerabilities** or malicious code.

Q: What is the difference between an API and an implementation?

Compiler error of the week (1/3)

What is wrong with this code?

```
static int getAge() {
    StdOut.println("What is your age?");
    int age = StdIn.readInt();
    StdOut.println("Your age is " + age);
}
```

Compiler error of the week (1/3)

What is wrong with this code?

```
static int getAge() {
    StdOut.println("What is your age?");
    int age = StdIn.readInt();
    StdOut.println("Your age is " + age);
}
```

```
java: missing return statement
```

Compiler error of the week (2/3)

```
static double multiply(double a, double b) {
    return a * b;
}

static double multiply(double a, double b, double c) {
    return a * b * c;
}

static int computeAreaOrVolume(boolean b, int[] arr) {
    if (b) {
        return multiply(arr[0], arr[1]);
    } else {
        return multiply(arr[0], arr[1], arr[2]);
    }
}
```

```
java: incompatible types: possible lossy conversion from double to int
```

Compiler error of the week (3/3)

What is wrong with this code?

```
static char getCharOrString() {
    return StdIn.readChar();
}

static String getCharOrString() {
    return StdIn.readString();
}
```

Compiler error of the week (3/3)

What is wrong with this code?

```
static char getCharOrString() {
    return StdIn.readChar();
}

static String getCharOrString() {
    return StdIn.readString();
}
```

```
java: method getCharOrString() is already defined in class...
```

Live Programming

Live Programming

- The Call Stack (explain using `DrawPicture`)
  - Understanding calls and returns.
  - Including early return.
  - Inspecting the stack in the debugger. (Using step-into and step-over).
- Overloading (e.g. `max`)
- Understanding pass-by-value and pass-by reference.

Sources for images and slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://commons.wikimedia.org/wiki/File:Injection_keine_Injektion_2a.svg
- https://xkcd.com/180/
- https://commons.wikimedia.org/wiki/File:Bookshelf_(cropped).jpg
- https://commons.wikimedia.org/wiki/File:A_graph_containing_all_installed_Haskell_packages.svg
- https://commons.wikimedia.org/wiki/File:Apache_Maven_logo.svg
- https://commons.wikimedia.org/wiki/File:Npm-logo.svg
- https://commons.wikimedia.org/wiki/File:NuGet_project_logo.svg