Principles of Software Construction (Design for change, class level)

Starting with Objects (dynamic dispatch, encapsulation, entry points)

Claire Le Goues Bogdan Vasilescu

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia (1 / 3): Homework 1 is released

Milestone due Monday; full assignment the next Monday.

Don’t panic, it’s a lot to figure out at first, and we know that.

- Setting up a new (to you) toolchain often involves roadbumps. This is why we released the HW before teaching you everything: for recitation!
- The milestone exists to ensure you iron out issues early.
- We also list additional language resources.

Miscellaneous:

- The rubric is descriptive and intended to be clear.
- Note the CI doesn’t test.
- The actual programming required is quite minimal!

Administrivia (2 / 3) : Office Hours

Office hours are on the calendar and mostly accurate.

- That said: please Google the error message or other symptoms you're seeing and try a few things before asking us in OH or Piazza. This is a CENTRAL skill.
- I will probably have OH on Monday next week, but probably not Mondays all semester, TBA.

Administrivia (3 / 3): The Waitlist

Non-update update: we are working on expanding capacity.

- If we succeed, it will be via the addition of a remote-only recitation.
- We expect to know for sure if this is feasible next week.

Even without it, there will be some movement in the WL.

If you're optimistic about joining from the WL, I encourage you to attend recitation and do/start the homework so you don't need to scramble to catch up if you officially join.

- We'll work with you about how/when to submit.

What did we talk about on Tuesday?

Tradeoffs?

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

This is Java code!

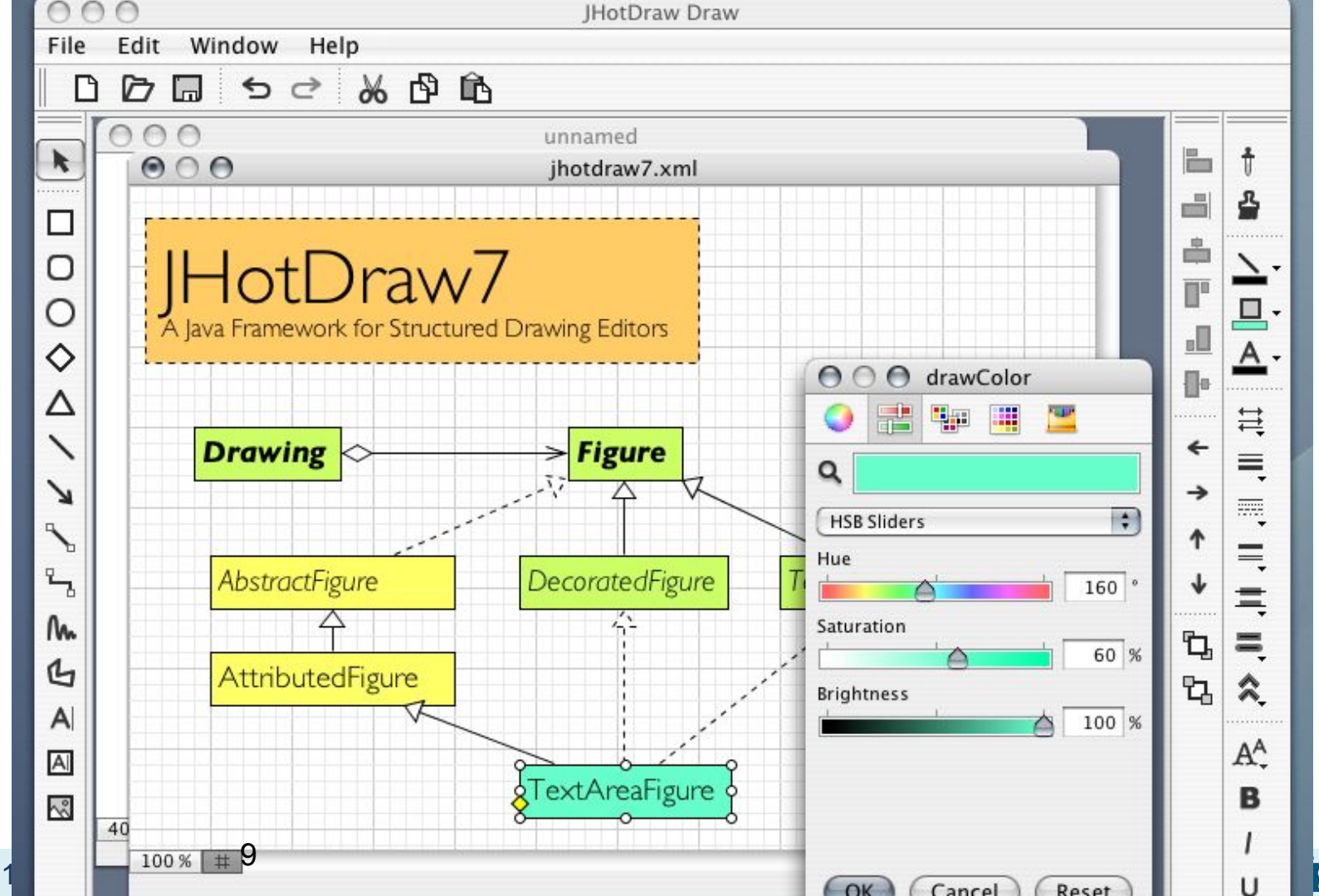
```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
```

Learning Goals

- Code:
  - Explain the need to design for change and design for division of labor
  - Understand subtype polymorphism and dynamic dispatch
  - Use encapsulation mechanisms
  - Distinguish object methods from global procedures
  - Start a program with entry code
- Tools:
  - Be able to submit the code to the checkpoint for HW1 using git.

Today: Key OOP Features that Support:

- **Design for Change** (flexibility, extensibility, modifiability)
- Design for Division of Labor
- Design for Understandability


JHotDraw Draw
File
Edit
Window
Help
unnamed
jhotdraw7.xml
JHotDraw7
A Java Framework for Structured Drawing Editors
Drawing
Figure
AbstractFigure
DecoratedFigure
AttributedFigure
TextAreaFigure
drawColor
HSB Sliders
Hue
160
Saturation
60
%
Brightness
100
%
OK
Cancel
Reset
100 %

Programming without Objects

This is C code!

Data structures and procedures

```
struct point {
   int    x;
   int    y;
 };
void movePoint(struct point p, int deltax, int deltay) { p.x = …; }


int main() {
    struct point p = { 1, 3 };
    int deltaX = 5;
    movePoint(p, 0, deltaX);
    ...
}
```

Data structures and procedures

Data is stored in memory in a certain format

All data the same memory layout, procedures expect that layout

Each procedure is compiled to an address

Procedure invocations jump to that address

Single address for procedure

(Function pointers provide more flexibility)

Objects

This is JavaScript code!

Object (JavaScript)

A program abstraction with internal state (data) and behavior (actions, methods)

Interact through messages (*invoking methods*)

- perform an action, update state (e.g., move)
- request some information (e.g., getSize)

```
const obj = {
    print: function() { console.log("foo"); }
}

obj.print()
// foo
```

Functions in an object are typically called *methods*

This is a *method invocation* (conceptually by sending a message to the object)

Objects can contain state

```
const obj = {
    v: 1,
    print: function() { console.log(this.v); },
    inc: function() { this.v++; }
}
obj.print()
// 1
obj.print()
// 1
obj.inc()
obj.print()
// 2
```

The object contains a variable *v*, called a *field*, to store state

Multiple methods in the object

This is JavaScript code!

Objects respond to messages, methods define *interface*

```
const obj = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
}
obj.get() + 2
// 3
obj.add(obj.get()+2)
// 4
obj.send()
// Uncaught TypeError: obj.send is not a function
```

Calling a method that does not exist results in an error

This is TypeScript code!

Typescript (and Java) allows us to explicitly define interface

```
interface Counter {
    v: number;
    inc(): void;
    get(): number;
    add(y: number): number;
}
const obj: Counter = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
};

obj.foo();
// Compile-time error: Property 'foo' does not exist
```

receiver

method name

v must be part of the interface in TypeScript. Ways to avoid this later.

The object assigned to *obj* must have all the same methods as the interface.

This is TypeScript code!

Typescript (and Java) allo

```
const obj = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
}
```

Ways to avoid this later.

```
interface Counter {
    v: number;
    inc(): void;
    get(): number;
    add(y: number): number;
}
const obj: Counter = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
};

obj.foo();
// Compile-time error: Property 'foo' does not exist
```

receiver

method name

The object assigned to *obj* must have all the same methods as the interface.

Yellow background is Java, Black is Typescript

Interfaces and Objects in Ja

```
interface Counter {
    int get();
    int add(int y);
    void inc();
}
Counter obj = new Counter() {
    int v = 1;
    public int get() { return this.v; }
    public int add(int y) { return this.v + y; }
    public void inc() { this.v++; }
};

System.out.println(obj.add(obj.get()));
// 2
```

```
interface Counter {
    v: number;
    inc(): void;
    get(): number;
    add(y: number): number;
}
const obj: Counter = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
}
```

object without a class. This isn't very common, it just looks a lot like the TS.

Object-oriented language feature enabling flexibility

SUBTYPE POLYMORPHISM /

DYNAMIC DISPATCH

Subtype Polymorphism / Dynamic Dispatch

- An interface describes the API/way to interact with an object. It does NOT provide the implementation.
- There may be multiple implementations of an interface!
- Multiple implementations can coexist in the same program
- *Every object has its own data and behavior, internals can be very different*

This is Java code!

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
```

class as template for objects with Point interface

This is Java code!

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    CartesianPoint(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
}
```

class as template for objects with Point interface

This is Java code!

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    CartesianPoint(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
}
Point p = new CartesianPoint(3, -10);
```

class as template for objects with Point interface

Constructor initializes the object

Calling *constructor* of class to create object

This is TypeScript code!

Note that Typescript lets us do this, too! Remember this?

```
interface Counter {
    v: number;
    inc(): void;
    get(): number;
    add(y: number): number;
}
const obj: Counter = {
    v: 1,
    inc: function() { this.v++; },
    get: function() { return this.v; },
    add: function(y) { return this.v + y; }
}
obj.foo();
// Compile-time error: Property 'foo' does not exist
```

This uses the *module pattern,* wrapping up variables and functions in a single scope, to instantiate a class conforming to the interface.

This is Typescript code!

TS/JS also allow classes explicitly, similar to Java

```
interface Counter {
    v: number;
    inc(): void;
    get(): number;
    add(y: number): number;
}
class C implements Counter = {
    v = 1;
    inc () { this.v++; }
    get () { return this.v; }
    add (y : number) { return this.v + y; }
}
const obj = new C();
// …
```

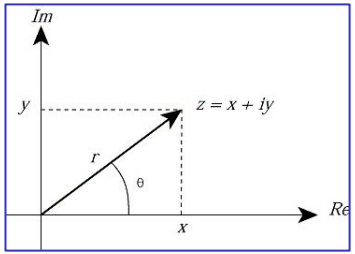
…but we can do things this way, too.

The module pattern/use of closures is *all over* JS/TS, so it's worth being comfortable with it.

This is Java code!

Multiple Implementations of Interface

```
interface Point {
    int getX();
    int getY();
}
class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
    double getAngle() {…}
}
Point p = new PolarPoint(5, .245);
```



This is Java code!

Multiple Implementations of Interface

```
interface Point {
    int getX();
    int getY();
}
class MiddlePoint implements Point {
    Point a, b;
    MiddlePoint(Point a, Point b) { this.a = a; this.b = b; }
    int getX() { return (this.a.getX() + this.b.getX()) / 2; }
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
Point p = new MiddlePoint(new PolarPoint(5, .245),
                          new CartesianPoint(3, 3));
```

Works with a imple-mentations of Point

This is Java code!

Clients work with all implementations of Interface

```
interface Point {
    int getX();
    int getY();
}
r = new Rectangle() {
    Point origin;
    int width, height;
    void draw() {
        this.drawLine(this.origin.getX(), this.origin.getY(),
            this.origin.getX()+this.width, this.origin.getY());
        … // more lines here
    }
};
```

Works with a
imple-
mentations
of Point

Subtype Polymorphism / Dynamic Dispatch

- An interface describes the API/way to interact with an object. It does NOT provide the implementation.
- There may be multiple implementations of an interface!
- Multiple implementations can coexist in the same program
- *Every object has its own data and behavior, internals can be very different*

This is Java code!

Points and Rectangles: Interface

```
interface Point {
    int getX();
    int getY();
}
interface Rectangle {
    Point getOrigin();
    int getWidth();
    int getHeight();
    void draw();
}
```

What are possible implementations of the Rectangle interface?

This is Java code!

Sets: Interface

```
interface IntSet {

    boolean contains(int element);

    boolean isSubsetOf(IntSet otherSet);

}
```

What are possible implementations of the IntSet interface?

This is Java code!

Programming against interfaces, not internals

```
interface Point {
    int getX();
    int getY();
    void moveUp(int y);
    Point copy();
}

Point p = …
int x = p.getX();
```

```
interface IntSet {
    boolean contains(
        int element);
    boolean isSubsetOf(
        IntSet otherSet);
}

IntSet a = …; IntSet b = …
boolean s = a.isSubsetOf(b);
```

This is Java code!

Java Twist: Classes implicitly have Interfaces

Classes can be used as types, like interfaces

All (public) methods can be called

No alternative implementations of class type

Prefer interfaces over class types!

```
class PolarPoint implements Point {
    double len, angle;
    ...
    int getX() {…}
    int getY() {…}
    double getAngle() {…}
}
PolarPoint pp = new PolarPoint(5, .245);
Point p = new PolarPoint(5, .245);
pp.getAngle(); // okay
p.getAngle(); // compilation error
```

This is Typescript code!

JavaScript and Classes

All methods of objects can be called

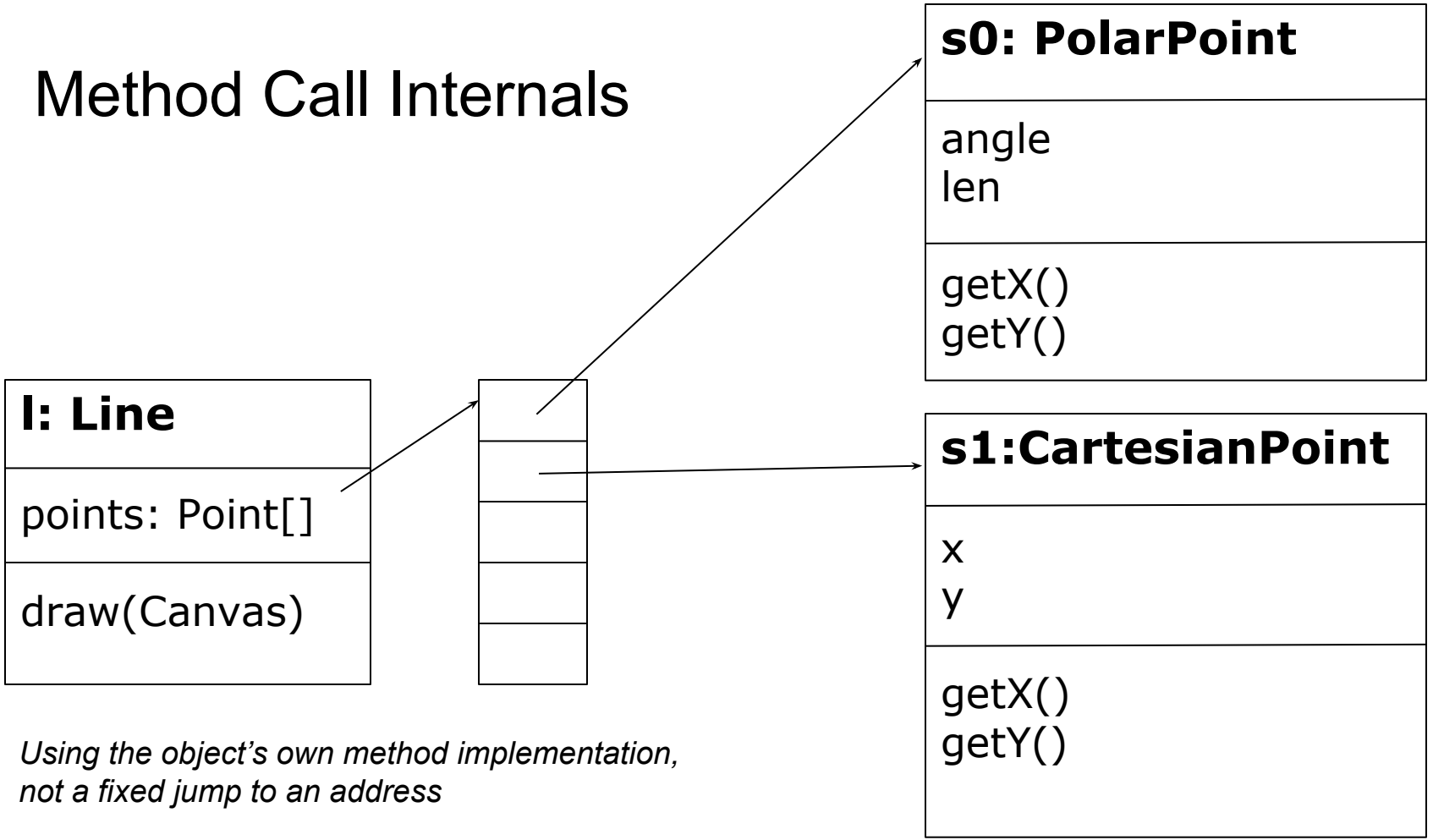
Objects with the same method can be called

No static checking by compiler; runtime error if method not exist

TypeScript added type system with interfaces

```
const pp = {
    len: 1, angle: 0,
    getX: function() {…}
    getAngle: function() {…}
}
const p = {
    x: 1, y: 0;
    getX: function() {…}
}
pp.getX(); p.getX(); // okay
pp.getAngle(); // okay
p.getAngle() // runtime error
```

Method Call Internals



Using the object's own method implementation, not a fixed jump to an address

Check your Understanding

This is Java code!

```
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); } }
class Cow implements Animal {
    public void makeSound() { moo(); }
    public void moo() {System.out.println("moo!"); } }
Animal x = new Animal() {
    public void makeSound() { System.out.println("chirp!"); }}
x.makeSound();

Animal d = new Dog();
d.makeSound();
Animal b = new Cow();
b.makeSound();
b.moo();
```

Check your Understanding

```
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); } }
class Cow implements Animal {
    public void makeSound() { moo(); }
    public void moo() {System.out.println("moo!"); } }
Animal x = new Animal() {
    public void makeSound() { System.out.println("chirp!"); }}
x.makeSound(); // “chirp”

Animal d = new Dog();
d.makeSound(); // “bark!”
Animal b = new Cow();
b.makeSound(); // “moo!”
b.moo(); // compile-time error
```

```
Animal a = new Animal();
a.makeSound(); // compile-time error
```

Dynamic Dispatch

Object Methods vs Global Functions/Procedures

This is Typescript code!

Flexibility of dynamic dispatch (JavaScript)

Each object decides implementation,
client does not care

Method is decided at runtime

Only single implementation of global function (and module)

```
// top-level function
function movePoint(p, x, y) { ... }

// create object, implementation unknown
const p = createPoint(...)

// call object's method
// object determines implementation
p.move(3, 5);

// single global implementation
// less flexibiliy
movePoint(p, 3, 5)
```

This is Java code!

Flexibility of dynamic dispatch (Java)

Each class decides implementation, client does not care

The "`main`" function is defined this way!

Static methods are *global functions*, only single copy exists; class provides only namespace

Java does *not* allow global functions outside of classes

```
interface Point {
    void move(int x, int y) { ... }
}
class Helper {
    static void movePoint(Point p,
                          int x, int y) {...}
}

Point p = createPoint(...);
// dynamic dispatch, object's method
p.move(4, 5);

// single global method, less flexible
Helper.movePoint(p, 4, 5);
```

Dynamic Dispatch

Benefits of Dynamic Dispatch

Discussion Dynamic Dispatch

- A user of an object does not need to know the object's implementation, only its interface
- All objects implementing the interface can be used interchangeably
- Allows flexible **change** (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

Why multiple implementations?

Different performance

- Choose implementation that works best for your use

Different behavior

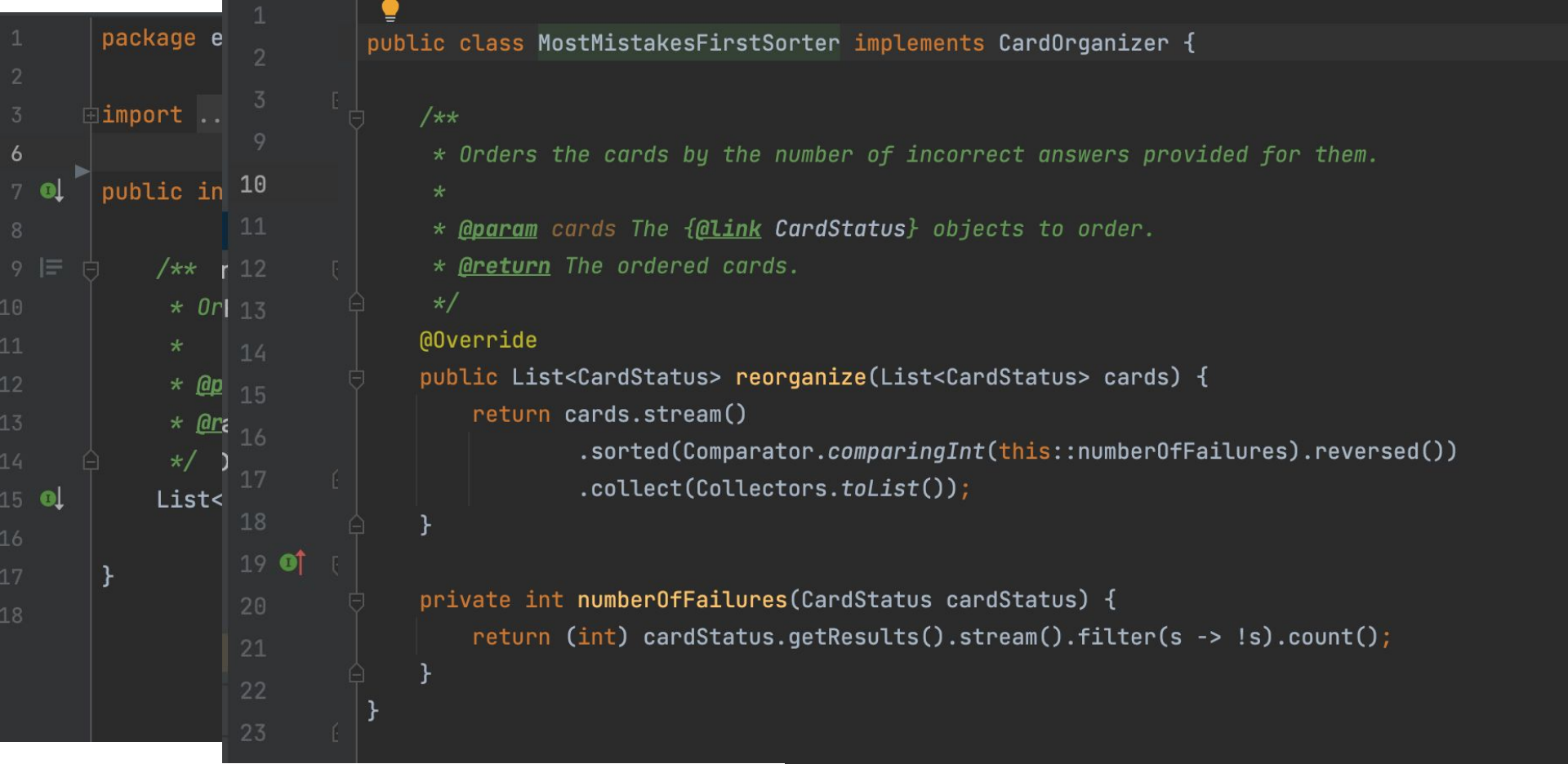
- Choose implementation that does what you want
- Behavior must comply with interface spec (“contract”)

Often performance and behavior both vary

- Provides a functionality – performance tradeoff
- Example: HashSet, TreeSet

This is Java code!

```
interface Order {
  boolean lessThan(int i, int j);
}

class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i < j; }
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i > j; }
}

static void sort(int[] list, Order order) {
  …
  boolean mustSwap =
    order.lessThan(list[j], list[i]);
  …
}
```



```
public class MostMistakesFirstSorter implements CardOrganizer {

    /**
     * Orders the cards by the number of incorrect answers provided for them.
     *
     * @param cards The {@link CardStatus} objects to order.
     * @return The ordered cards.
     */
    @Override
    public List<CardStatus> reorganize(List<CardStatus> cards) {
        return cards.stream()
                .sorted(Comparator.comparingInt(this::numberOfFailures).reversed())
                .collect(Collectors.toList());
    }

    private int numberOfFailures(CardStatus cardStatus) {
        return (int) cardStatus.getResults().stream().filter(s -> !s).count();
    }
}
```

This is Java code!

Historical note: simulation and the origins of OO programming

Simula 67 was the first object-oriented language

Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center

Developed to support discrete-event simulation

- Application: operations research, e.g. traffic analysis
- Extensibility was a key quality attribute for them
- Code reuse was another

Dahl and Nygaard at the time of Simula's development

Information Hiding

Encapsulation

Encapsulation / Information hiding

- Well designed objects project internals from others
  - both internal state and implementation details
- Well-designed code hides all implementation details
  - Cleanly separates interface from implementation
  - Modules communicate only through interfaces
  - They are oblivious to each others' inner workings
- Hidden details can be changed without changing client!
- Fundamental tenet of software design

Left is Java, right is Typescript

How to hide information?

```
class CartesianPoint {
    int x,y;
    Point(int x, int y) {
        this.x=x;
        this.y=y;
    }
    int getX() { return this.x; }
    int getY() { return this.y; }
    int helper_getAngle();
}
```

```
const point = {
    x: 1, y: 0,
    getX: function() {…}
    helper_getAngle:
        function() {…}
}
```

This is Java code!

Java: Access modifier to hide private details

```
public class PolarPoint implements Point {
    private double len, angle;
    private int xcache = -1;
    public PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle; computeX(); }
    public int getX() { return xcache; }
    public int getY() {...}
    private int computeX() {
        xcache = this.len * cos(this.angle);
    }
}
```

This is Java code!

Java: Access modifier to hide private details

```
public class PolarPoint implements Point {
    private double len, angle;
    private int xcache = -1;
    public PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle; computeX(); }
    public int getX() { return xcache; }
    public int getY() {...}
    private int computeX() {
        xcache = this.len * cos(this.angle);
    }
}
PolarPoint p = new PolarPoint(5, .245);
```

This is Java code!

Java: Access modifier to hide private details

```
public class PolarPoint implements Point {
    private double len, angle;
    private int xcache = -1;
    public PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle; computeX(); }
    public int getX() { return xcache; }
    public int getY() {...}
    private int computeX() {
        xcache = this.len * cos(this.angle);
    }
}
PolarPoint p = new PolarPoint(5, .245);
p.xcache // type error, trying to access private member
p.computeX(); // type error, private method
```

Benefits of information hiding

Decouples the objects that comprise a system: Allows them to be developed, tested, optimized, used, understood, and modified in isolation

Speeds up system development: Objects can be developed in parallel

Eases **maintenance burden**: Objects can be understood more quickly and debugged with little fear of harming other modules

Enables effective **performance tuning**: “Hot” classes can be optimized in isolation

Increases software **reuse**: Loosely-coupled classes often prove useful in other contexts

This is Java code!

Java: Information hiding with interfaces

```
public interface Point { … }
private class PolarPoint implements Point {
    private double len, angle;
    public void computeX() { … }
    public int getX() { return xcache; }
}
```

This is Java code!

Java: Information hiding with interfaces

```
public interface Point { … }
private class PolarPoint implements Point {
    private double len, angle;
    public void computeX() { … }
    public int getX() { return xcache; }
}
public class Factory {
    public Point createPoint(int x, int y) {
        return new PolarPoint(x, y);
    }
}
```

This is Java code!

Java: Information hiding with interfaces

```
public interface Point { … }
private class PolarPoint implements Point {
    private double len, angle;
    public void computeX() { … }
    public int getX() { return xcache; }
}
public class Factory {
    public Point createPoint(int x, int y) {
        return new PolarPoint(x, y);
    }
}
Point p = new Factory().createPoint((5, .245);
p.computeX(); // type error, method not in interface Point
```

Principles of Information hiding with interfaces (Java)

Declare variables using interface types, not class types

- Client can use only interface methods
- Fields and implementation-specific methods not accessible from client code

Use `private` for fields and internal methods to restrict access also in class types; accessible only from within same class

Interface methods must be `public`.

Other modifiers `protected` (for inheritance, more later) and package

JavaScript: Closures for Hiding

All methods and fields are public, no language constructs for access control

TypeScript added them, so it's quite similar to Java!

In JS: Encoding hiding with closures

```
function createPolarPoint(len, angle) {
    let xcache = -1;
    let internalLen=len;
    function computeX() {…}
    return {
        getX: function() {
            computeX(); return xcache; },
        getY: function() {
            return len * sin(angle); }
    };
}
const pp = createPolarPoint(1, 0);
pp.getX(); // works
pp.computeX(); // runtime error
pp.xcache // undefined
pp.len // undefined
```

This is Javascript code!

Closures

In nested functions/classes, inner functions/classes can access variables and arguments of outer functions

Frequently used in JavaScript

In Java: Closures for nested classes and lambda functions, but outer variables need to be final

```
function a(x) {
    const z = 3;
    function b(y) {
        x++;
        console.log(x+y+z);
    }
    b(5);
    console.log(x);
}
a(3);
// 12
// 4
```

This is Typescript code!

Type/JavaScript: Modules

Information hiding at the file level!

Decide what functions, variables, classes to keep private in a file

Historically, all code was in one file; later, multiple competing module systems
Standardized since ECMAScript 2015 (ES6)

In general, it is good practice to use files/modules to organize your code and do this kind of information hiding.

import interfaces / functions from other modules

```
import { f, b }
    from 'dir/file'
import fs from 'fs'

interface Point { ... }

function createP(a, b) {...}

function helper() { ...  }

export { Point, createP }
```

decide what functions / interfaces can be access from other modules

Java: Packages and classes

Each class is in a file with same name; classes grouped in packages (directories)

Fully qualified name = Package + Class name (e.g. `java.lang.String`)

All public classes from all packages can be used

Imports simplify names

```
import me.util.PolarPoint; PolarPoint p = new PolarPoint(...);
```

instead of

```
me.util.PolarPoint p = new me.util.PolarPoint(...);
```

Best practices for information hiding

- Carefully design your API
- Provide only functionality required by clients
  - All other members should be private / hidden through interfaces or closure
- *You can always make a private member public later (or export an additional method) without breaking clients, but not vice-versa!*

Objects do not do anything on their own; they wait for method calls…

Starting a Program

Starting a Program

This is Typescript code!

Typescript compiles to Javascript, by the way. There are several ways to run it.

Objects do not do anything on their own, they wait for method calls

Every program needs a starting point, or waits for events

```
// start with: node file.js
function createPrinter() {
    return {
        print: function() { console.log("hi"); }
    }
}
const printer = createPrinter();
printer.print()
// hi
```

Defining interfaces, functions, classes

Starting:
Creating objects and calling methods

This is Java code!

Starting Java Code

All Java code is in classes, so how to create an object and call a method?

Special syntax for *main* method in class (`java X` calls *main* in *X*)

```
// start with: java Printer
class Printer {
    void print() {
        System.out.println("hi");
    }
    public static void main(String[] args) {
        Printer obj = new Printer();
        obj.print();
    }
}
```

Main method to be executed, here used to create object and invoke method

Static methods belong to class not the object, generally avoid them

…so you can submit your homework…

Quick git basics!

… switch to terminal / browser on github.com and demo

Summary

Need to divide work, divide and conquer

Objects encapsulate state and behavior

Static/global functions: Only a single function provided, less flexibility

Dynamic dispatch: Each object's own method is executed, multiple implementations possible

Encapsulation: Hide object internals behind interface

Optional Survey: Confusions?

Link also in chat, will post to Piazza.

https://forms.gle/8hQDPgbvz4wnKdyQ6