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

Starting with Objects (dynamic dispatch, encapsulation)

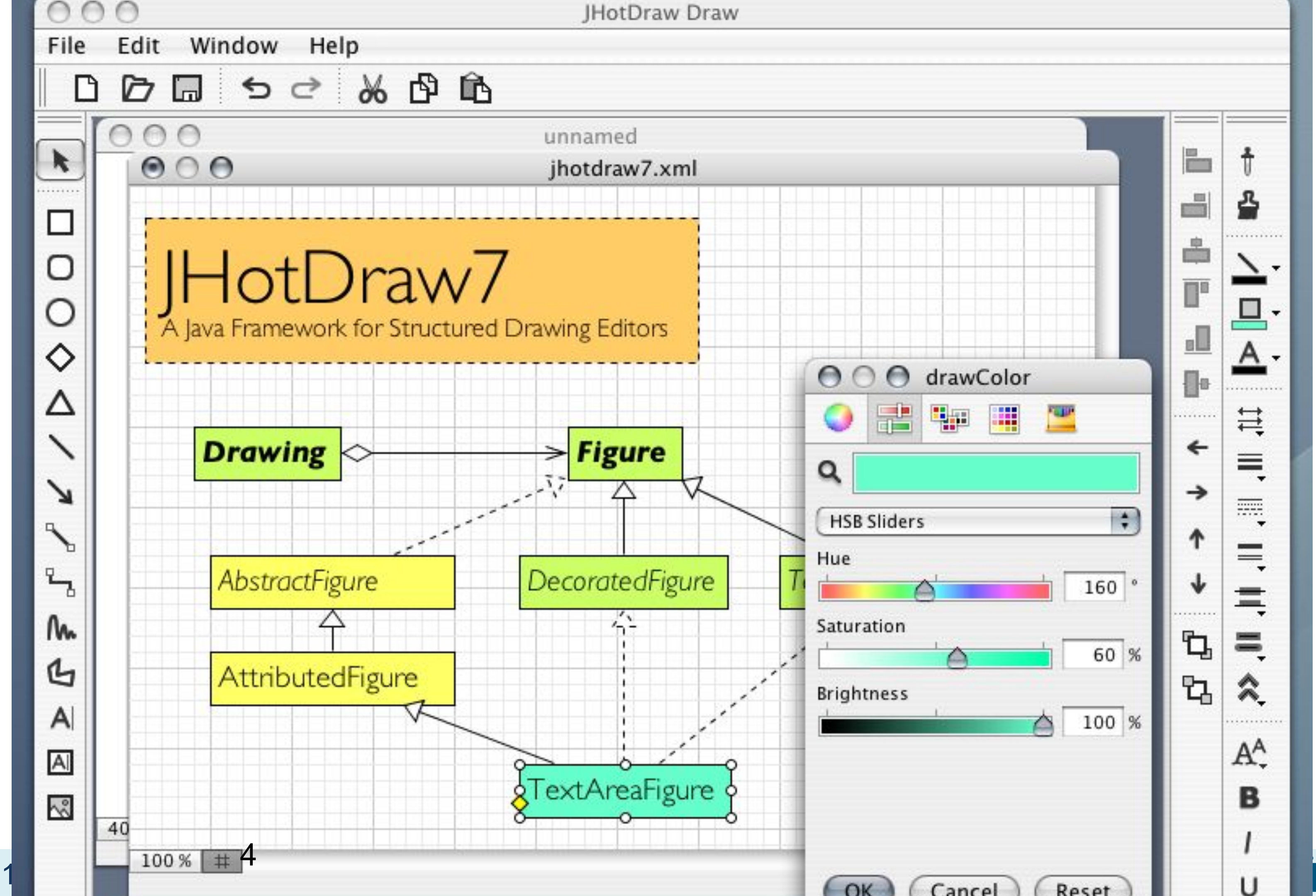
Christian Kästner Vincent Hellendoorn

Tradeoffs?

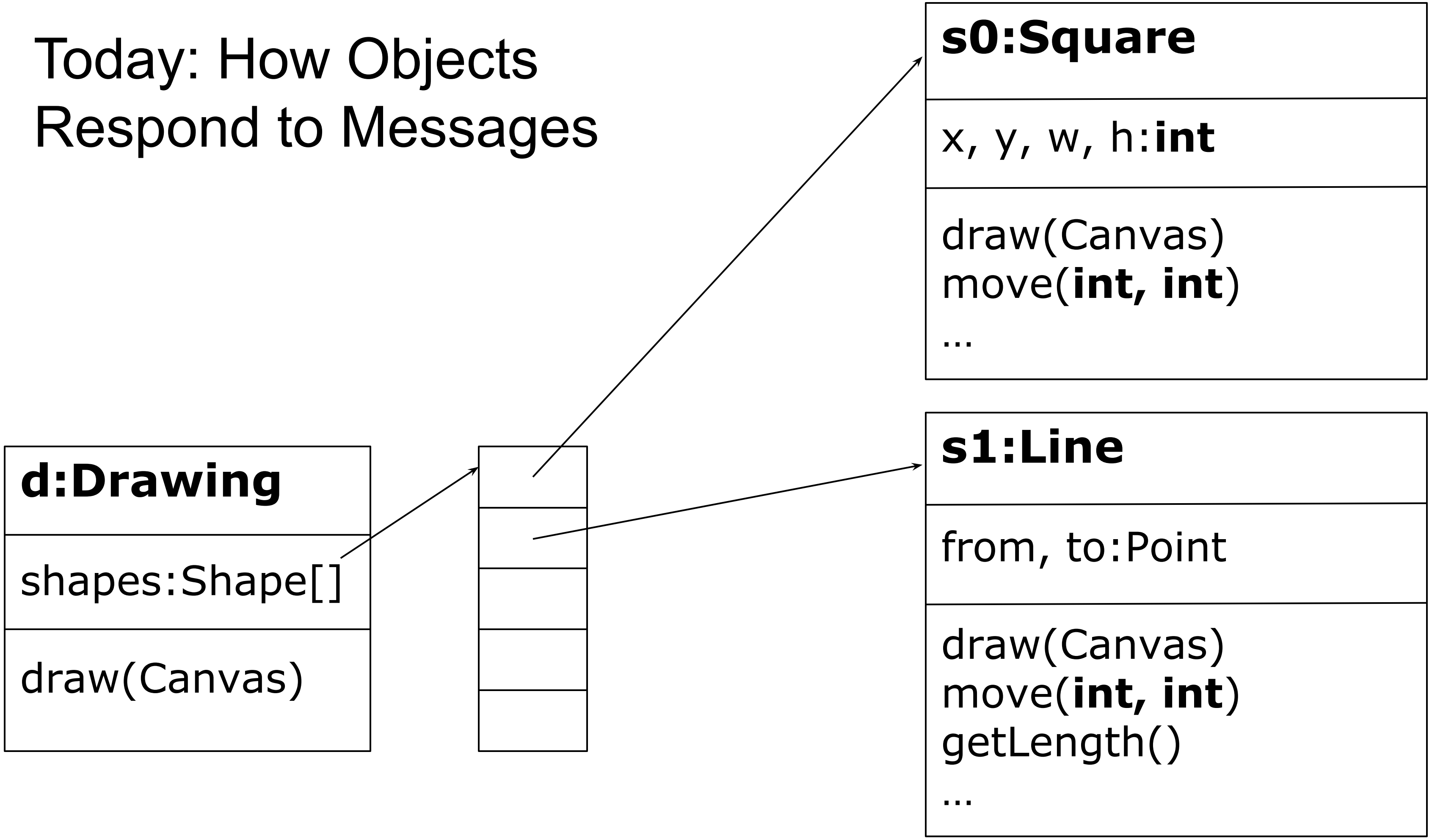
```java
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

```java
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 {
```


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
100 %
OK
Cancel
Reset

Today: How Objects Respond to Messages



Learning Goals

- 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

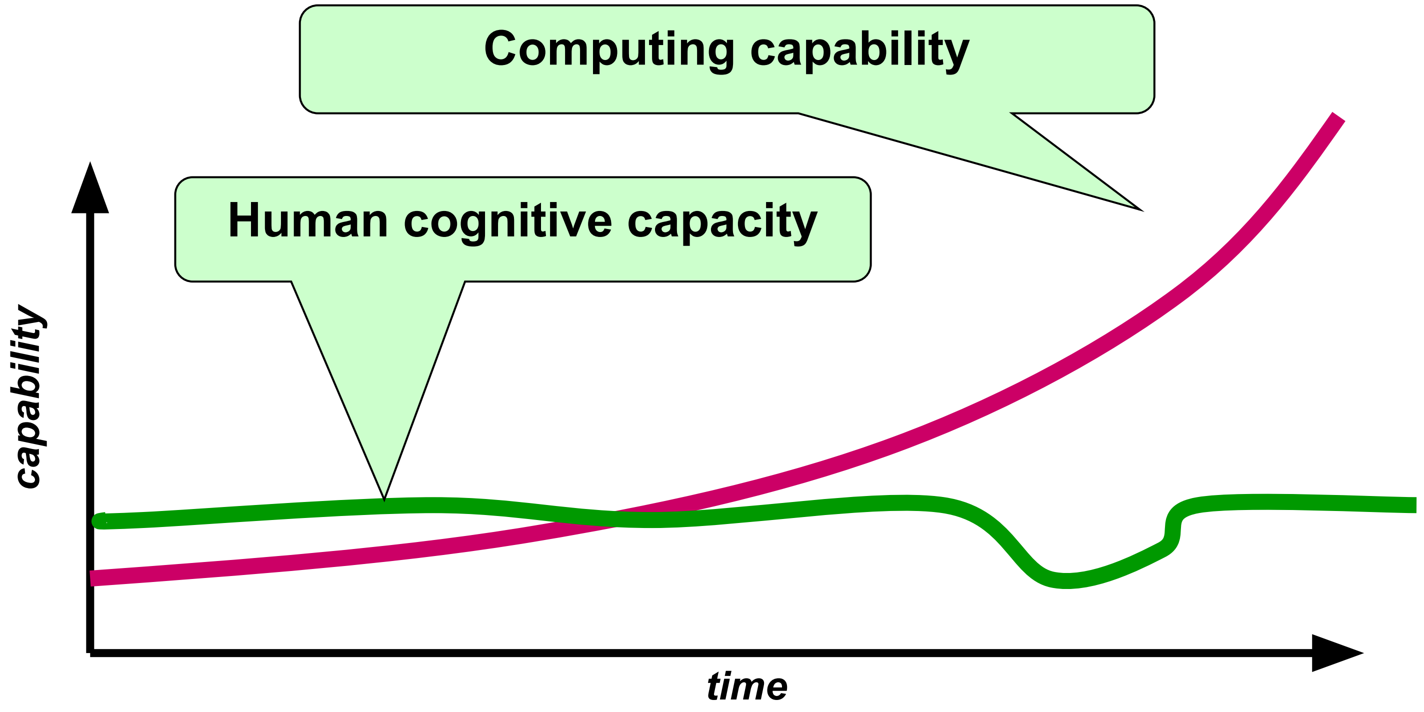
Design Goals, Principles, and Patterns

- Design Goals
  - Design for Change
  - Design for Division of Labor
- Design Principles
  - Explicit Interfaces (clear boundaries)
  - Information Hiding (hide likely changes)
- Design Patterns
  - Strategy Design Pattern
  - Composite Design Pattern
- Supporting Language Features
  - Subtype Polymorphism
  - Encapuslation

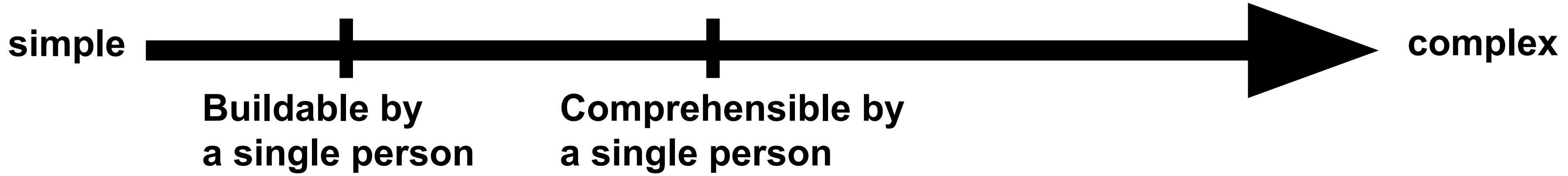
Software Change

- …accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974
- Software that does not change becomes useless over time.
  —Belady and Lehman
- For successful software projects, most of the cost is spent evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles of software design

The limits of exponentials



Building Complex Systems


simple
complex
Buildable by a single person
Comprehensible by a single person

- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations

Today: Key OOP Features that Support:

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

Programming without Objects

Data structures and procedures

```c
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

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

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

Interface declared explicitly with TypeScript

```
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
```

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.

Interfaces and Objects in Java

```
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
```

This uses anonymous classes to create an object without a class. More later.

Object-oriented language feature enabling flexibility

SUBTYPE POLYMORPHISM / DYNAMIC DISPATCH

Subtype Polymorphism / Dynamic Dispatch

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior, internals can be very different

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);
```

Creating Objects

```
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}
IntSet emptySet = new IntSet() {
    boolean contains(int element) { return false; }
    boolean isSubsetOf(IntSet otherSet) { return true; }
}
```

Creating Objects

```
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(IntSet otherSet);
}
IntSet threeSet = new IntSet() {
    boolean contains(int element) {
        return element == 3;
    }
    boolean isSubsetOf(IntSet otherSet) {
        return otherSet.contains(3);
    }
}
```

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

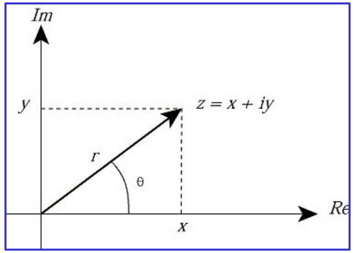
Calling *constructor* of class to create object

Multiple Implementations of Interface

```
interface Point {
    int getX();
    int getY();
}
class SkewedPoint implements Point {
    int x, y;
    SkewedPoint(int x, int y) {
            this.x = x + 10; this.y = y * 2; }
    int getX() { return this.x - 10; }
    int getY() { return this.y / 2; }
}
Point p = new SkewedPoint(3, -10);
p.getX()
```

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);
```



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

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

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable
- Every object has its own data and behavior, internals can be very different

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?

Sets: Interface

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

What are possible implementations of the IntSet interface?

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
```

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);
```

JavaScript Twist: No Interfaces!

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 adds 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
```

JavaScript and Classes

JavaScript traditionally had no classes; easy to create objects directly

Classes introduced later in ECMAScript 2015 (ES6)

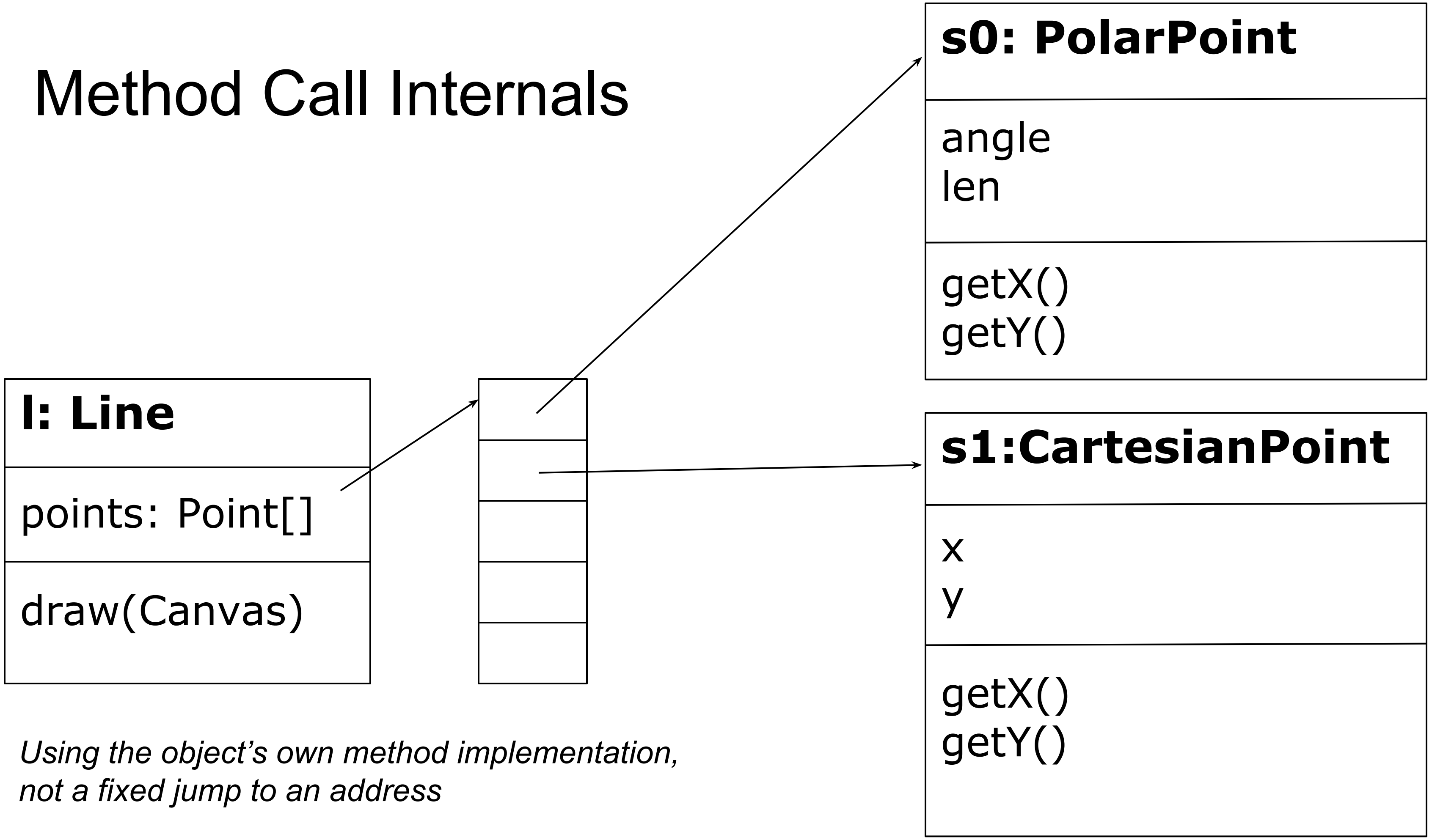
TypeScript supports classes and interfaces

Use somewhat controversial

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


const p = new Point(4, 5);
p.getX();
```

Method Call Internals



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

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() { mew(); }
    public void mew() {System.out.println("Mew!"); } }
Animal x = new Animal() {
    public void makeSound() { System.out.println("chirp!"); }}
x.makeSound();
Animal a = new Animal();
a.makeSound();
Animal d = new Dog();
d.makeSound();
Animal b = new Cow();
b.makeSound();
b.mew();
```

Dynamic Dispatch

Object Methods vs Global Functions/Procedures

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)
```

Flexibility of dynamic dispatch (Java)

Each class decides implementation,
client does not care

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);
```

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

Design for Change!

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

```
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]);
  …
}
```

Other Examples of Multiple Implementations

Change the sorting criteria in a list

Change the aggregation method for computations over a list (e.g., fold)

Compute the tax on a sale

Compute a discount on a sale

Change the layout of a form

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

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() {…}
}
```

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

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
```

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 (only recent TypeScript)

Encoding 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
```

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
```

JavaScript: Modules

Information hiding at the file level!

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

Traditionally, all code in one file; later multiple competing module systems

Standardized since ECMAScript 2015 (ES6)

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 in 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(...);
```

Java 9: Modules

Advanced feature, discussed in later lecture

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!

Starting a Program

Starting a Program

Objects do not do anything on their own, 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

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

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

Quick Survey: The Muddiest Point

Link also on Piazza under “Resources”

https://forms.gle/W2YoKCvHE9ayqZb27