Principles of Software Construction: Objects, Design, and Concurrency (Part 1: Designing Classes)

Design for Change (class level)

Christian Kästner Bogdan Vasilescu

School of Computer Science

Polar Points

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

```
interface PolarPoint {
  double getAngle() ;
  double getLength();
}
```

```
class PolarPointImpl implements Point, PolarPoint {
    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() {…}
    double getLength() {… }
}
PolarPoint p = new PolarPointImpl(5, .245);
Point q = new PolarPointImpl(5, .245);
```

Middle Points

```
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));
```

Design Goals for Today

- **Design for Change** (flexibility, extensibility, modifiability)

also

- Design for Division of Labor
- Design for Understandability

See “UML and Patterns” Ch. 26.7

STRATEGY DESIGN PATTERN (EXPLOITING POLYMORPHISM FOR FLEXIBILITY)

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];
  }
  …
}
```

```
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 {
  boolean compare(int I, int j) { return i>j; }}
```

```
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 {
  boolean compare(int I, int j) {
     return i>j;
  }}
```

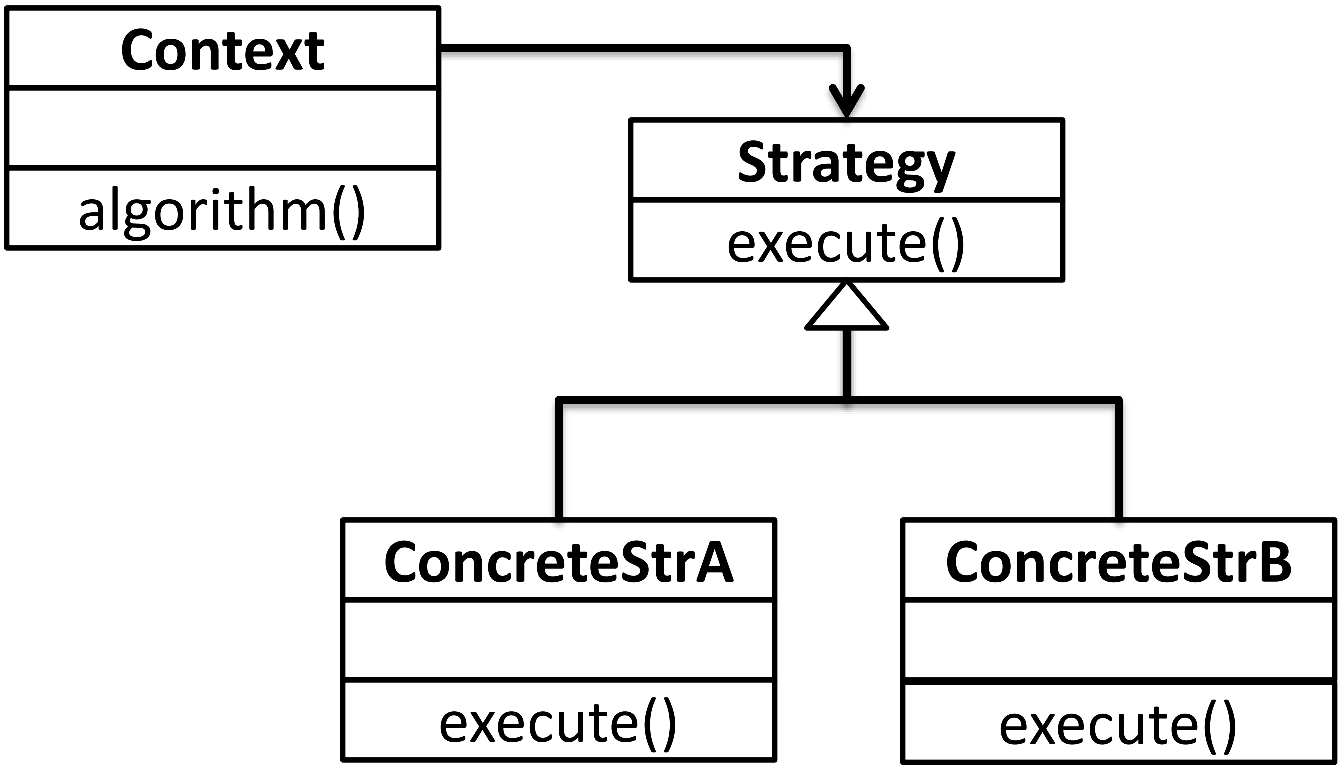
One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

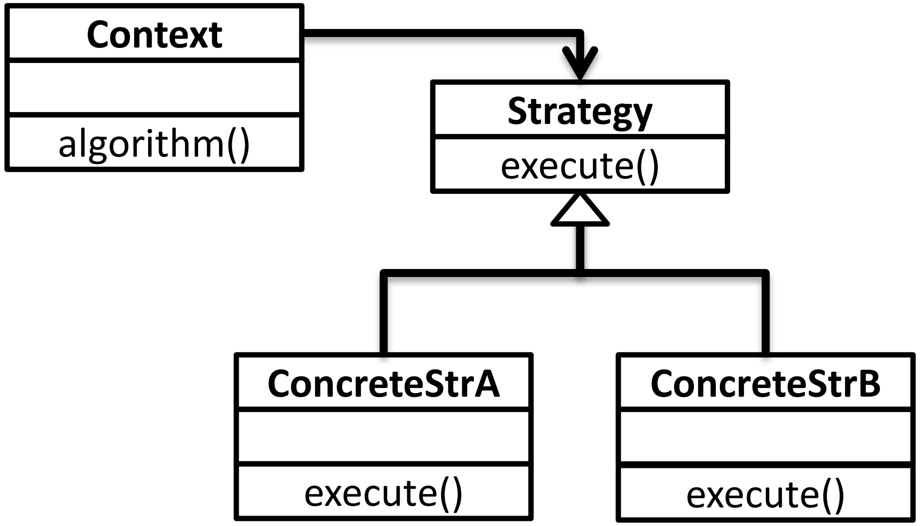
Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

Behavioral: Strategy



Behavioral: Strategy



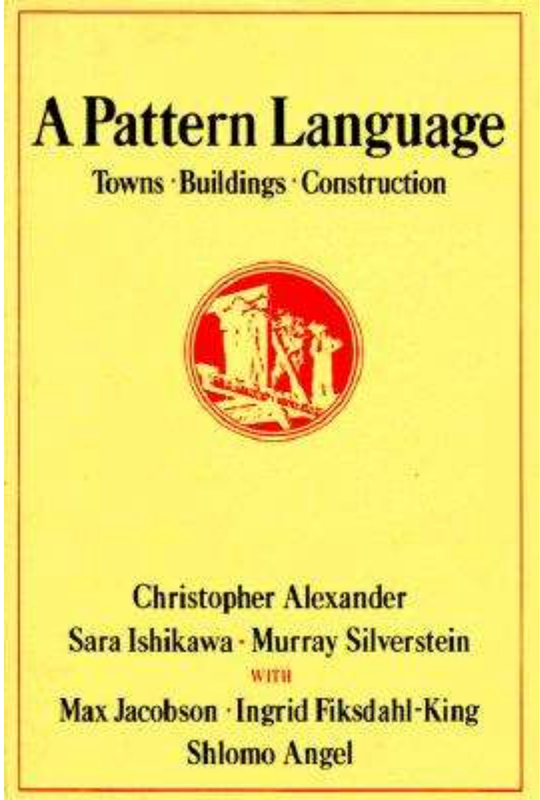
- Applicability
  - Many classes differ in only their behavior
  - Client needs different variants of an algorithm
- Consequences
  - Code is more extensible with new strategies
    - compare to conditionals
  - Separates algorithm from context
    - each can vary independently
    - design for change and reuse; reduce coupling
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead
- Design for change
  - Find what varies and encapsulate it
  - Allows changing/adding alternative variations later
  - Class *Context* closed for modification, but open for extension
- Equivalent in functional progr. languages: Higher-order functions

More Design Scenarios

- 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

Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
  – Christopher Alexander



Benefits of Patterns

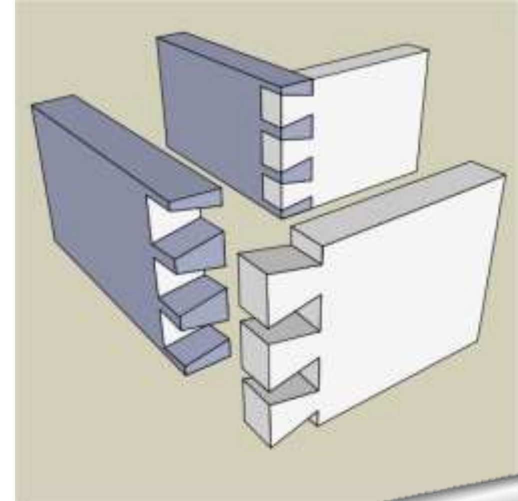
- Shared language of design
  - Increases communication bandwidth
  - Decreases misunderstandings
- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?

How not to discuss design

(from Shalloway and Trott)

- Carpenter 1: How do you think we should build these drawers?
- Carpenter 2: Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…
- SE example: “I wrote this if statement to handle … followed by a while loop … with a break statement so that…”

Discussion with design patterns



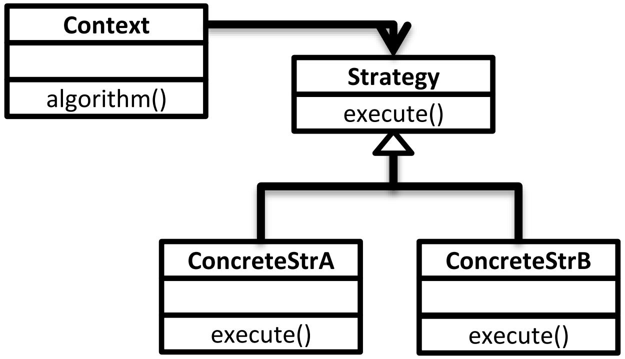
- Carpenter 1: Should we use a dovetail joint or a miter joint?

- Subtext:
  - miter joint: cheap, invisible, breaks easily
  - dovetail joint: expensive, beautiful, durable
- Shared terminology and knowledge of consequences raises level of abstraction
  - CS: Should we use a Strategy?
  - Subtext: Is there a varying part in a stable context? Might there be advantages in limiting the number of possible implementations?

Elements of a Pattern

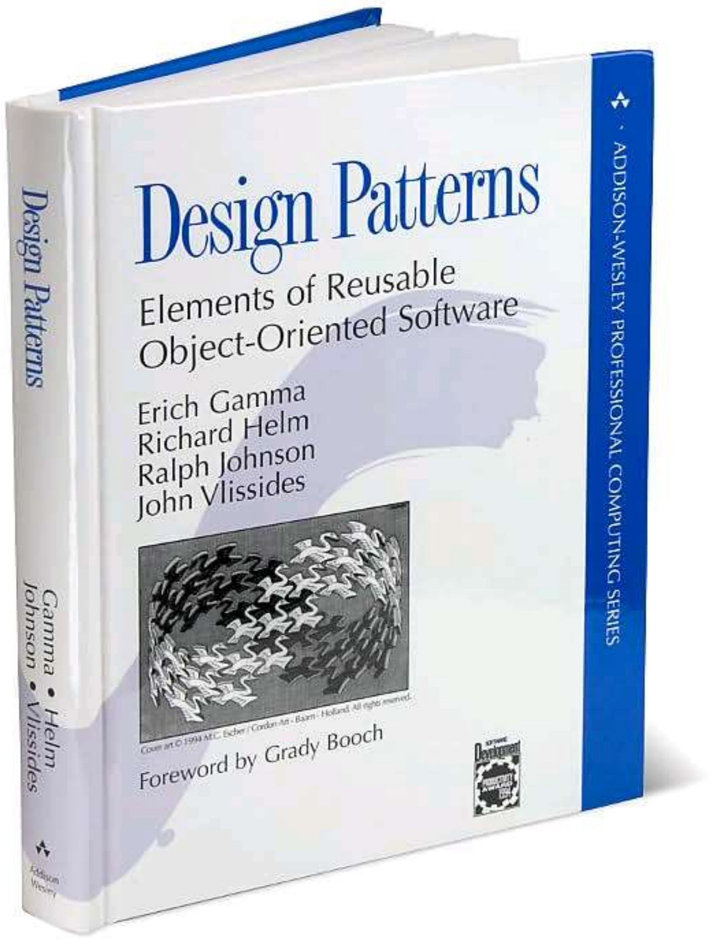
- Name
  - Important because it becomes part of a design vocabulary
  - Raises level of communication
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs as well as benefits
    - Issues include flexibility, extensibility, etc.
    - There may be variations in the pattern with different consequences

Strategy pattern



- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes: Code can be harder to understand; Lots of overhead if the strategies are simple

History: Design Patterns Book

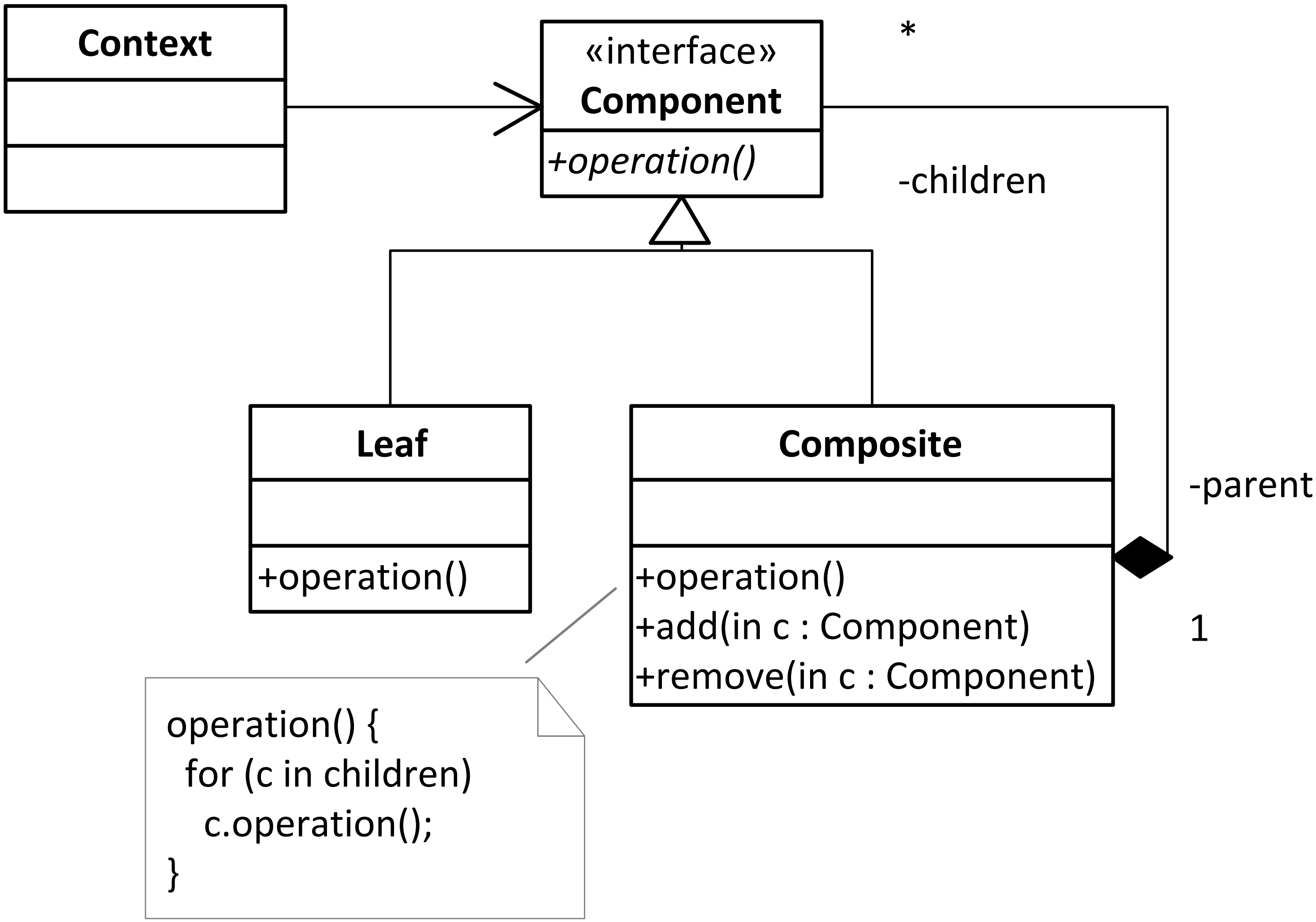


- Brought Design Patterns into the mainstream
- Authors known as the Gang of Four (GoF)
- Focuses on *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*
- Great as a reference text
- Uses C++, Smalltalk

Design Exercise (on paper)

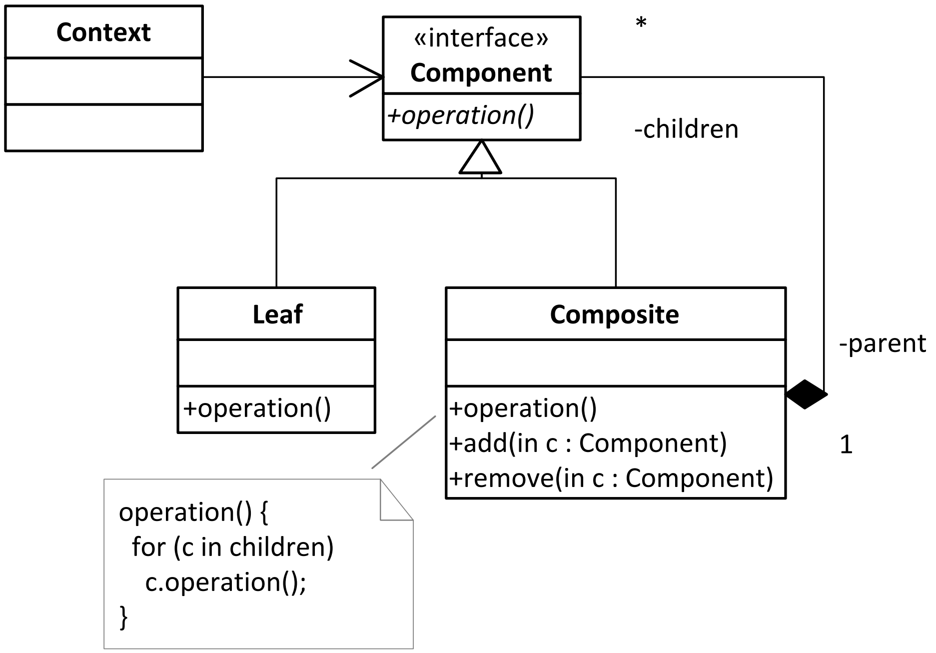
- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

The Composite Design Pattern



The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



We have seen this before

```
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; }
}
```

ENCAPSULATION (LANGUAGE FEATURE TO CONTROL VISIBILITY)

Controlling Access – Best practices

- Define an interface
- Client may only use the messages in the interface
- Fields not accessible from client code
- Methods only accessible if exposed in interface

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    Point(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
    String getText() { return this.x + " x " + this.y; }
}
Point p = new CartesianPoint(3, -10);
p.getX();
p.getText(); // not accessible
p.x; // not accessible
```

Interface Type

Java: Classes as Types

- Classes usable as type
  - (Public) methods in classes usable like methods in interfaces
  - (Public) fields directly accessible from other classes
  - Language constructs (public, private, protected) to control access
- Prefer programming to interfaces (variables should have **interface type**, not class type)
  - Esp. whenever there are multiple implementations of a concept
  - Allows to provide different implementations later
  - Prevents dependence on implementation details

```
int add(CartesianPoint p) { …    // preferably no
int add(Point p) { …             // yes!
```

Interfaces and Classes (Review)

```
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() { return angle; }
}
```

```
Point p = new PolarPoint(5, .245);
p.getX();
p.getAngle(); // not accessible
p.len // not accessible
```

```
PolarPoint pp = ...
pp.getX();
pp.getAngle();
pp.len
```

Java: Visibility Modifiers

```
class Point {
    private int x, y;
    public int getX() { return this.x; } // a method; getY() is similar
    public Point(int px, int py) { this.x = px; this.y = py; }// constructor
}
class Rectangle {
    private Point origin;
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(this.origin.getX(), this.origin.getY(),     // first line
                this.origin.getX()+this.width, origin.getY());
        … // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        this.origin = o; this.width = w; this.height = h;
    }
}
```

Hiding interior state

```
class Point {
    private
    public
    public
}
class Recta
    private
    private int width, height;
    public Point getOrigin() { return origin; }
    public
    public
        (
        .
    }
    public Rectangle(Point o, int w, int h) {
        this.origin = o; this.width = w; this.height = h;
    }
}
```

<u>Some Client Code</u>

```
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

<u>Client Code that will *not* work in this version</u>

```
Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.origin.x + r.width; //  trying to "look inside"
```

Hiding interior state

```
class Point {
    private
    public
    public
}
class Recta
    private
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(this.origin.getX(), this.origin.getY(),      // first line
                 this.origin.getX()+this.width, origin.getY());
        … // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        this.origin = o; this.width = w; this.height = h;
    }
}
```

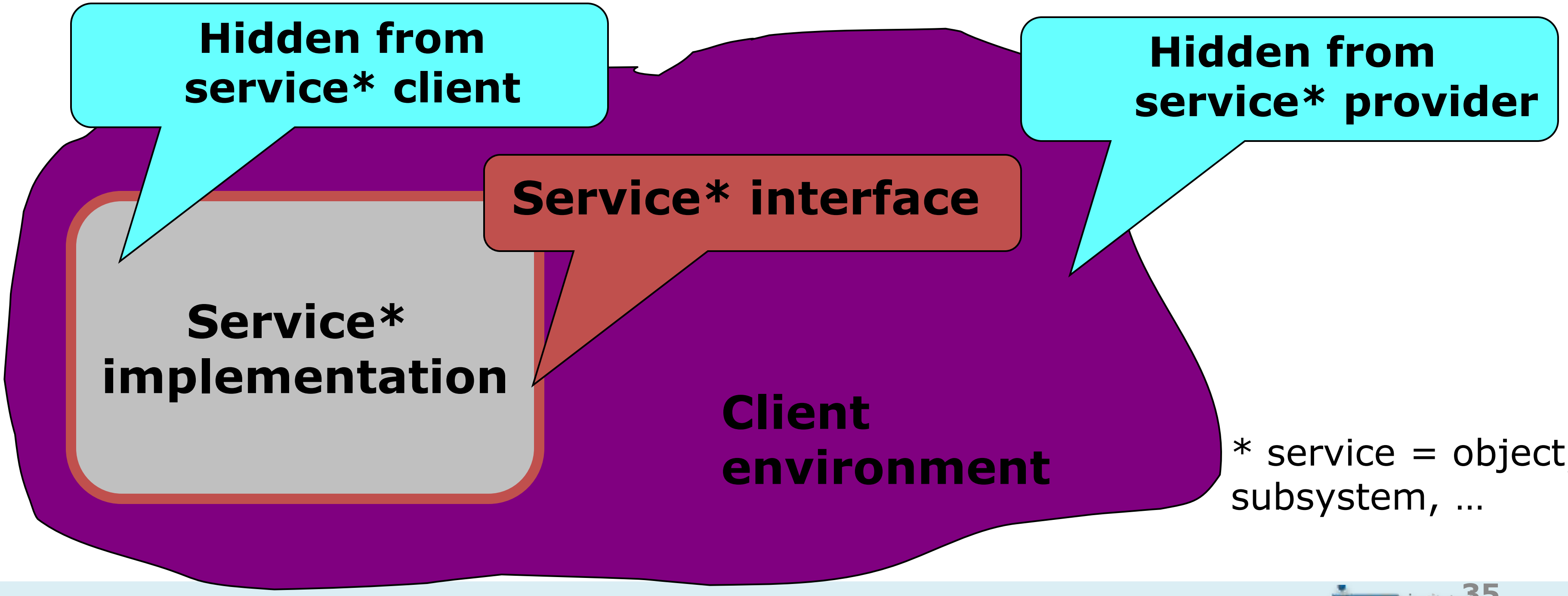
Discussion:

- ***What are the benefits of* private *fields?***
- ***Methods can also be private – why is this useful?***

DESIGN PRINCIPLE: INFORMATION HIDING

Fundamental Design Principle for Change: Information Hiding

- Expose as little implementation detail as necessary
- Allows to change hidden details later



* service = object, subsystem, …

Information Hiding

- Interfaces (contracts) remain stable
- Hidden implementation can be changed easily
- => Identify what is likely to change, and hide it
- => Requires anticipation of change (judgment)

- Points example: Minimal stable interface, allows alternative implementations and flexible composition

- (Not all change can be anticipated, causing maintenance work or reducing flexibility)

Information Hiding promotes Reuse

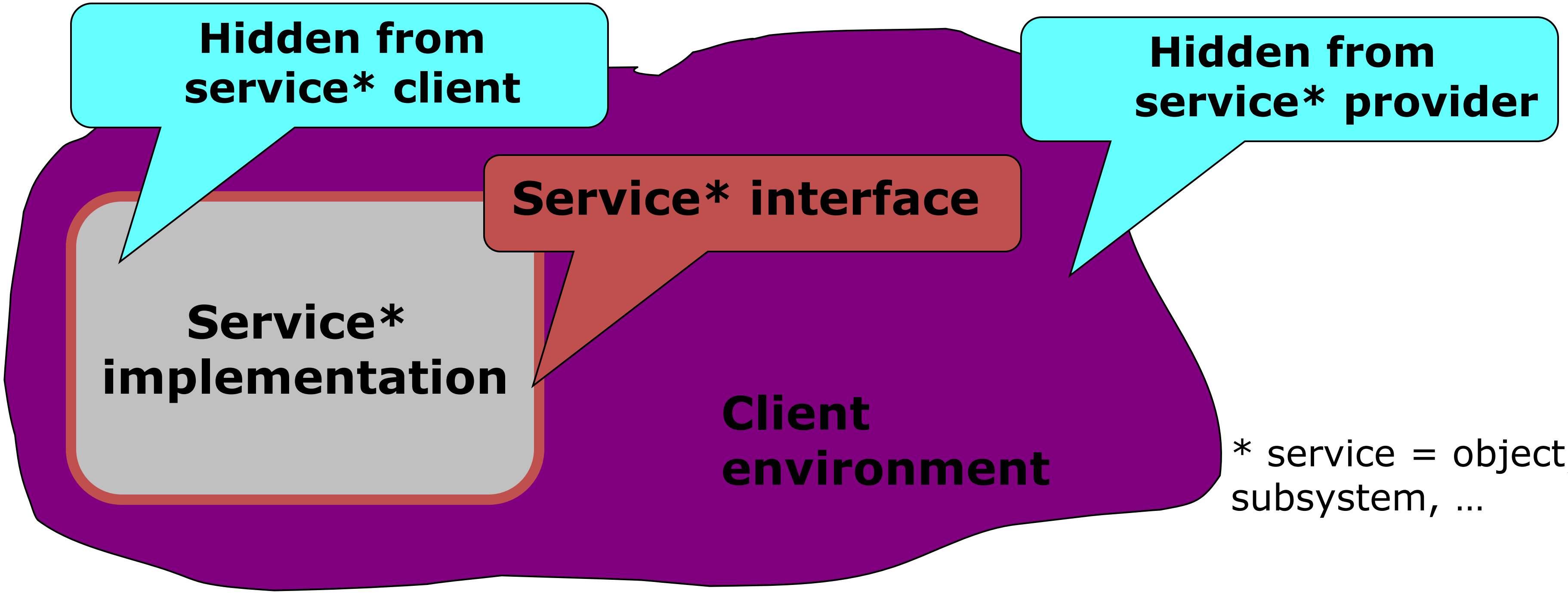
- Think in terms of abstractions not implementations
  - e.g., Point vs CartesianPoint
- Abstractions can often be reused
- Different implementations of the same interface possible,
  - e.g., reuse Rectangle but provide different Point implementation
- Decoupling implementations
- Hiding internals of implementations

More on reuse next week

INFORMATION HIDING CASE STUDY

CONTRACTS
(BEYOND TYPE SIGNATURES)

Contracts and Clients



* service = object, subsystem, …

Contracts

- Agreement between provider and users of an object
- Includes
  – Interface specification (types)
  – Functionality and correctness expectations
  – Performance expectations
- What the method does, not how it does it
  – Interface (API), not implementation

Who’s to blame?

```
Algorithms.shortestDistance(g,
   “Tom”, “Anne”);

> ArrayOutOfBoundsException
```

Who’s to blame?

```
Algorithms.shortestDistance(g,
    “Tom”, “Anne”);

> -1
```

Who's to blame?

```
Algorithms.shortestDistance(g,
    “Tom”, “Anne”);

> 0
```

Who’s to blame?

```
class Algorithms {
    /**
     * This method finds the
     * shortest distance between to
     * verticies. It returns -1 if
     * the two nodes are not
     * connected. */
    int shortestDistance(…) {…}
}
```

Who’s to blame?

```
Math.sqrt(-5);
```

```
> 0
```

Who’s to blame?

```
/**
  * Returns the correctly rounded positive square root of a
  * {@code double} value.
  * Special cases:
  * <ul><li>If the argument is NaN or less than zero, then the
  * result is NaN.
  * <li>If the argument is positive infinity, then the result
  * is positive infinity.
  * <li>If the argument is positive zero or negative zero, then
  * the result is the same as the argument.</ul>
  * Otherwise, the result is the {@code double} value closest to
  * the true mathematical square root of the argument value.
  *
  * @param a a value.
  * @return the positive square root of {@code a}.
  * If the argument is NaN or less than zero, the result is NaN.
  */

public static double sqrt(double a) { …}
```

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data f
  attempt is made to read as ma
  The number of bytes actually r
  until input data is available, er
- If len is zero, then no bytes ar
  attempt to read at least one by
  end of file, the value -1 is retu
  into b.
- The first byte read is stored in
  on. The number of bytes read
  bytes actually read; these byte
  1], leaving elements b[off+*k*]
- In every case, elements b[0] through b[off] and
  elements b[off+len] through b[b.length-1] are unaffected.

- **Specification of return**
- **Timing behavior (blocks)**
- **Case-by-case spec**
  - **len=0 → return 0**
  - **len>0 && eof → return -1**
  - **len>0 && !eof →return >0**
- **Exactly where the data is stored**
- **What parts of the array are not affected**

- Throws:
  - IOException - If the first byte c
    or if the input stream has been
  - NullPointerException - If b is n
  - IndexOutOfBoundsException -
    than b.length - off

- **Multiple error cases, each with a precondition**
- **Includes "runtime exceptions" not in throws clause**

Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes (performance, scalability, security, …)
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

Functional Specification

- States method's and caller's responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule…
  - I will build a with the following detailed specification
  - Some contracts have remedies for nonperformance
- Method contract structure
  - **Preconditions**: what method requires for correct operation
  - **Postconditions**: what method establishes on completion
  - **Exceptional behavior**: what it does if precondition violated
- Defines what it means for impl to be correct

Functional Specificati

What does the implementation have to fulfill if the client violates the precondition?

- State
- Analo
  - If yo
  - I wil … ing detailed specification
  - Som … contracts have remedies for nonperformance
- Method contract structure
  - **Preconditions**: what method requires for correct operation
  - **Postconditions**: what method establishes on completion
  - **Exceptional behavior**: what it does if precondition violated
- Defines what it means for impl to be correct

Formal Specifications

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @              (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```

Advantage of formal specifications:

* runtime checks (almost) for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)

Disadvantages?

Runtime Checking of Specifications with Assertions

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @           (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
    assert len >= 0;
    assert array.length == len;
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    assert sum …;
    return sum;
}
```

java -ea Main

Runtime Checking with Exceptions

```
/*@ requires len >= 0 && array.length == len
  @ ensures \result ==
  @            (\sum int j;  0 <= j && j < len;  array[j])
  @*/
float sum(int array[], int len) {
     if (len < 0 || array.length != len)
          throw IllegalArgumentException(…);
     float sum = 0.0;
     int i = 0;
     while (i < len) {
          sum = sum + array[i]; i = i + 1;
     }
     return sum;
     assert …;
}
```

Check arguments even when assertions are disabled.
Good for robust libraries!

Specifications in the real world

Javadoc

```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant time.
 * In some implementations, it may run in time proportional to the
 * element position.
 *
 * @param index position of element to return; must be non-negative and
 *              less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

postcondition

precondition

Write a Specification

- Write
  - a type signature,
  - a textual specification, and
  - a formal specification

for a function **slice(list, from, until)** that returns all values of a list between positions <from> and <until> as a new list

Contacts and Interfaces

- All objects implementing an interface must adhere to the interface's contracts
  - Objects may provide different implementations for the same specification
  - Subtype polymorphism: Client only cares about interface, not about the implementation

p.getX() **s.read()**

=> **Design for Change**

Specifications in Practice

- Describe expectations beyond the type signature
- Ideally formal pre- and post-conditions
- Textual specifications in practice
  - Best effort approach
- If any specification at all
- Specification especially necessary when reusing code and integrating code

- Writing specifications is good practice
- Writing fully formal specifications is often unrealistic

ASIDE:
SPECIFICATION OF CLASS INVARIANTS

62

Data Structure Invariants (cf. 122)

```
struct list {
    elem data;
    struct list* next;
};
struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```

Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ ... }
```

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

Class Invariants

- Properties about the fields of an object
- Established by the constructor

```
public class SimpleSet {

    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { … }
}
```

```
public class SimpleSet {

    int contents[];
    int size;

    //@invariant sorted(contents);

    SimpleSet(int capacity) { … }

    boolean add(int i) { … }

    boolean contains(int i) { … }
}
```

Java: Constructors

- Special “Methods” to create objects
  - Same name as class, no return type
- May initialize object during creation
- Implicit constructor without parameters if none provided

```java
class APoint {
    int x,y;
}
APoint p = new APoint();
p.x=3;
p.y=-10;
```

```java
class BPoint {
    int x,y;
    BPoint(int x, int y)
            {this.x=x; this.y=y;}
}
BPoint p = new BPoint(3, -10);
```

EXCURSION: TECHNICAL REALIZATION OF SUBTYPE POLYMORPHISM

Reminder: Subtype Polymorphism

- A type (e.g. Point) can have many forms (e.g., CartesianPoint, PolarPoint, …)

- All implementations of an interface can be used interchangeably

- When invoking a method p.x() the specific implementation of x() from object p is executed
  – The executed method depends on the actual object p, i.e., on the runtime type
  – It does not depend on the static type, i.e., how p is declared

Objects and References (example)

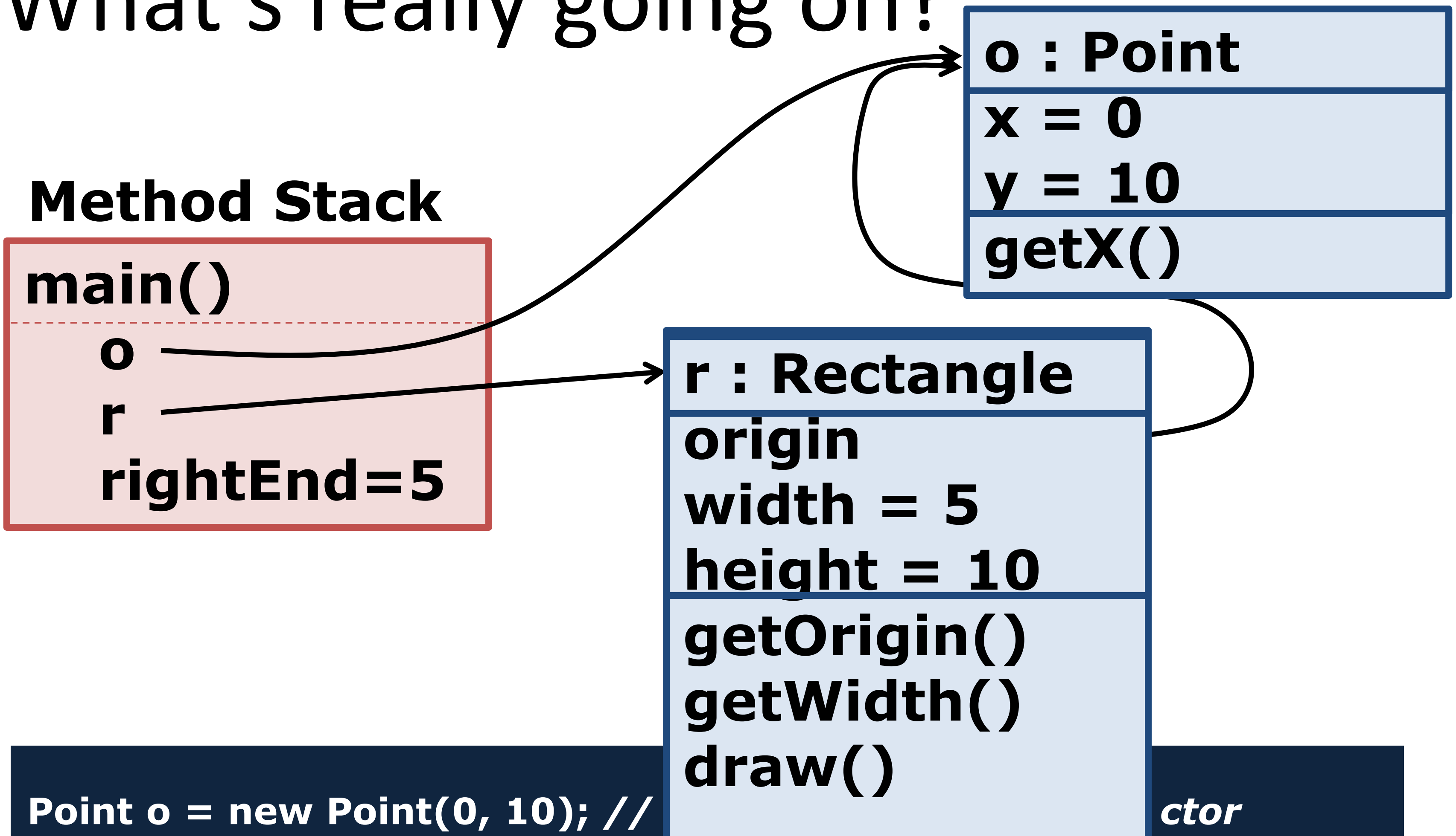
```
// allocates memory, calls constructor
Point o = new PolarPoint(0, 10);

Rectangle r = new MyRectangle(o, 5, 10);

r.draw();

int rightEnd = r.getOrigin().getX() +
    r.getWidth(); // 5
```

What’s really going on?




```
Point o = new Point(0, 10); //                                    ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
```

Anatomy of a Method Call

r.setX(5)

The **receiver**, an implicit argument, called **this** inside the method

The method **name**. Identifies which method to use, of all the methods the receiver's class defines

Method **arguments**, just like function arguments

Static types vs dynamic types

- Static type: how is a variable declared
- Dynamic type: what type has the object in memory when executing the program (we may not know until we execute the program)

```
Point createZeroPoint() {
        if (new Math.Random().nextBoolean())
                return new CartesianPoint(0, 0);
        else    return new PolarPoint(0,0);
}
Point p = createZeroPoint();
p.getX();
p.getAngle();
```

Method dispatch (conceptually)

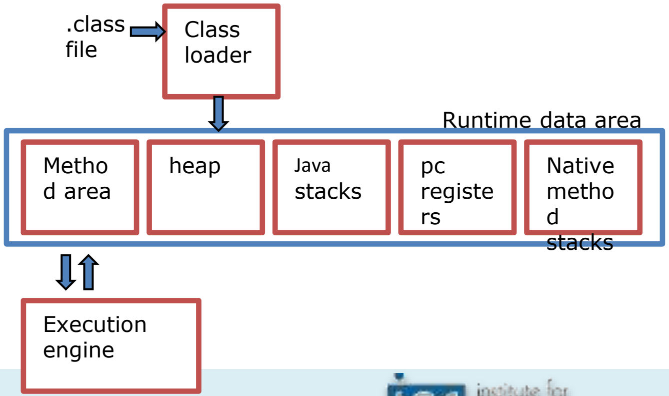
- Step 1 (compile time): determine what type to look in
  - Look at the **static type** (Point) of the receiver (p)
- Step 2 (compile time): find the method in that type
  - Find the method in the **interface/class** with the right name

    `int getX();`
  - Error if there is no such method
  - Error if the method is not accessible (e.g., private)
- Step 3 (run time): Execute the method stored in the **object**

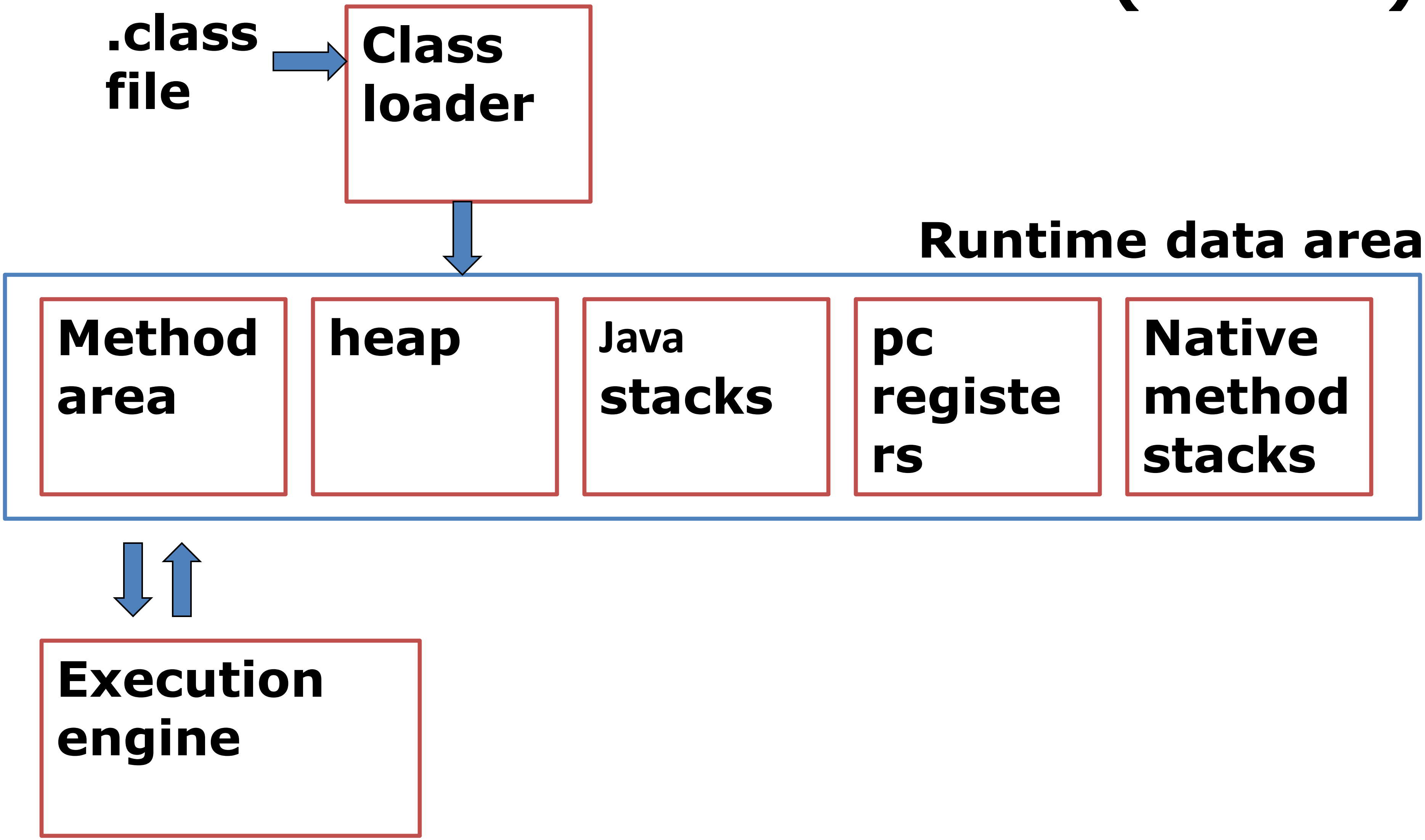
q : PolarPoint
len = 5
angle = .34
getX()

Method dispatch (actual; simplified)

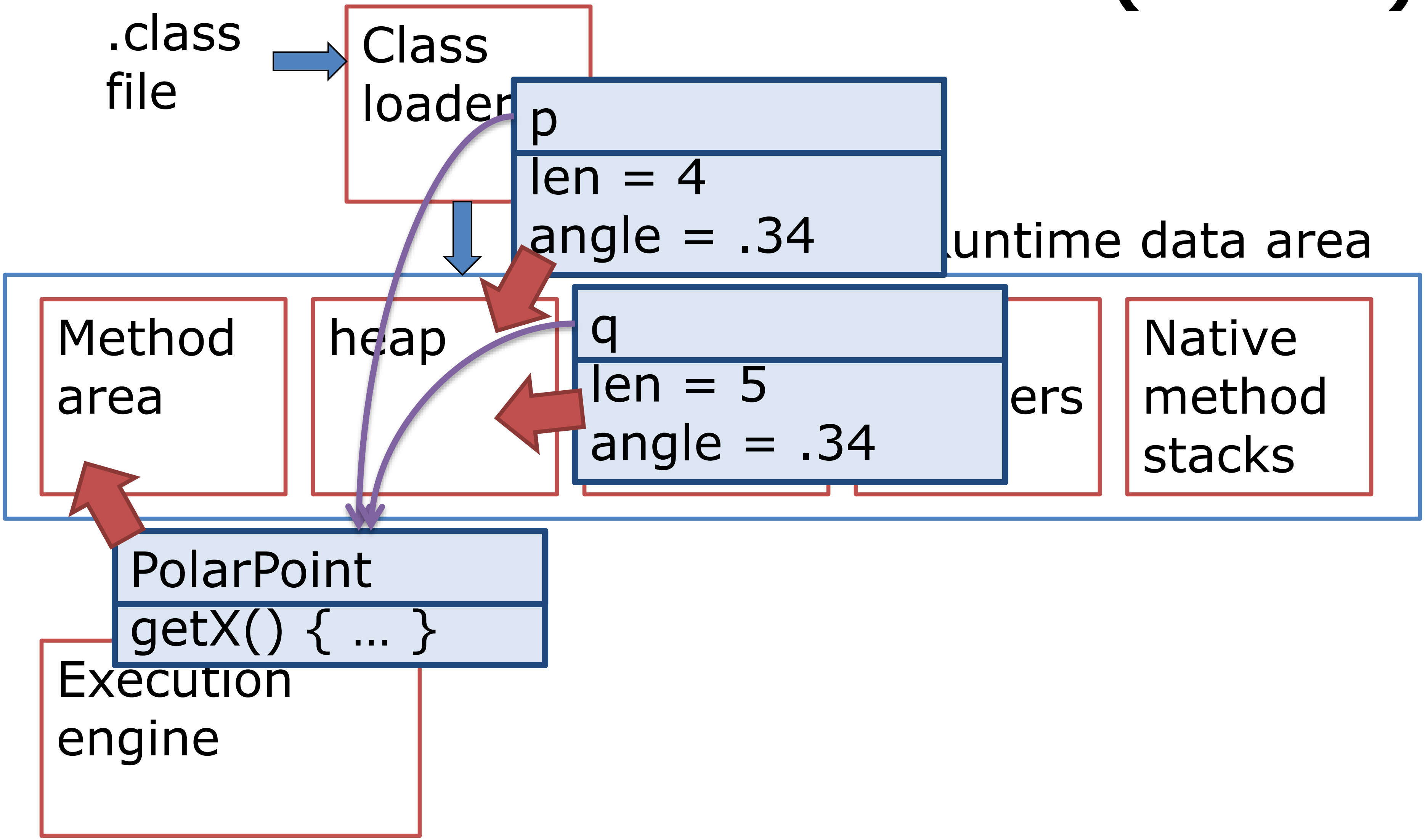
- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the **heap** and get its **class**
- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method
  - Invoke that implementation



The Java Virtual Machine (sketch)



The Java Virtual Machine (sketch)



SUMMARY: DESIGN FOR CHANGE/ DIVISION OF LABOR

Design Goals

- Design for Change such that
  - Classes are *open for extension* and modification without invasive changes
  - Subtype polymorphism enables changes behind interface
  - Classes encapsulate details likely to change behind (small) stable interfaces
- Design for Division of Labor such that
  - Internal parts can be *developed* independently
  - Internal details of other classes do not need to be *understood*, contract is sufficient
  - Test classes and their contracts separately (unit testing)