Objects Analysis Threads Design

15-214

toad

Spring 2014

Principles of Software Construction: Objects, Design, and Concurrency

An Introduction to Object-Oriented Programming

Charlie Garrod **Christian Kästner**

School of Computer Science

isr institute for SOFTWARE RESEARCH

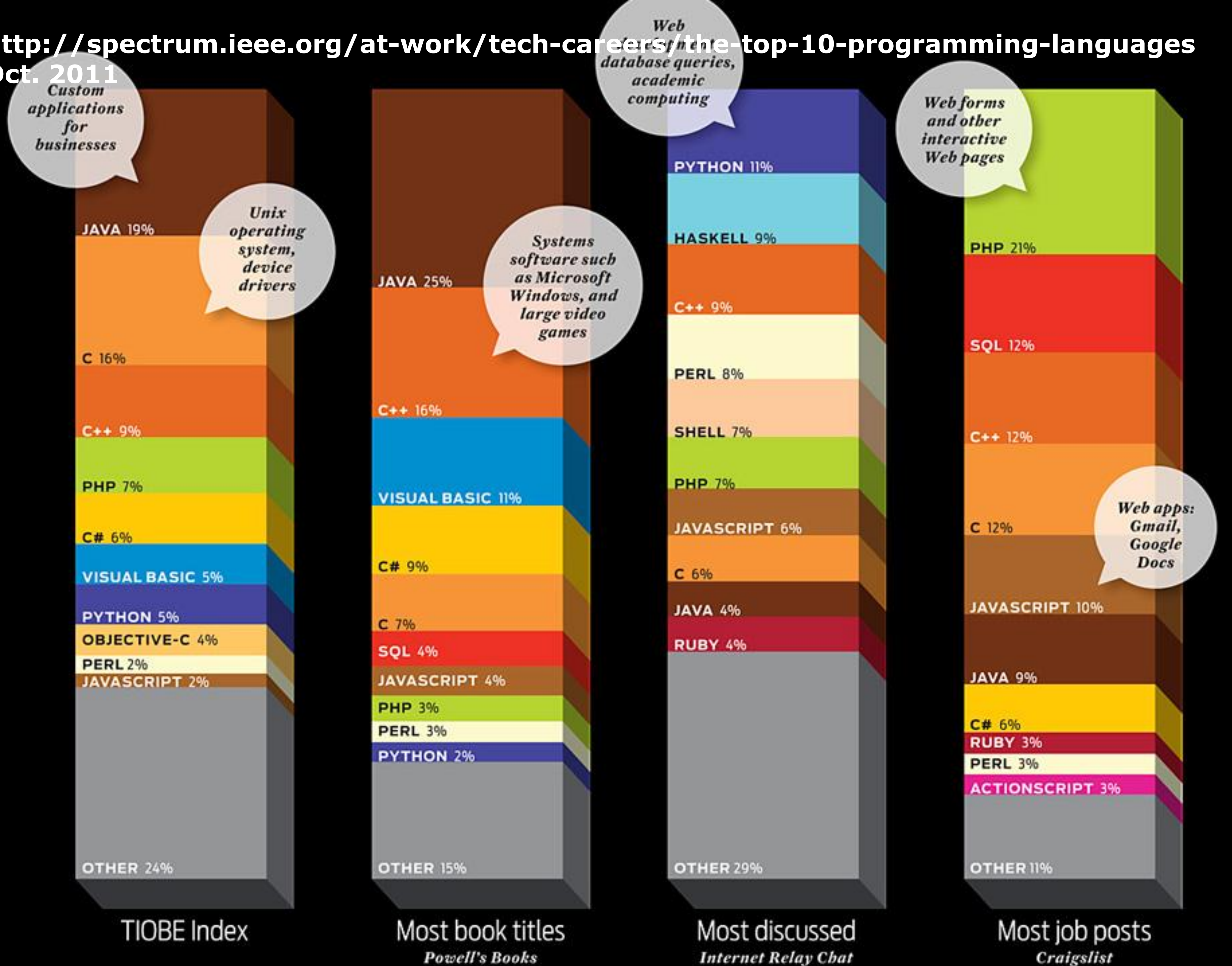
Learning Goals

- Understanding key object-oriented concepts
- Understand the purpose of interfaces and how interfaces can be implemented
- Distinguish the concepts interface, class, type
- Explain concepts to encapsulate data and behavior inside objects
- Explain method dispatch to objects and the differences to non-OOP languages as C
- Understand the difference between object identity and object equality

Object-Oriented Programming Languages

- C++
- Java
- C#
- Smalltalk
- Scala
- Objective-C
- JavaScript
- Ruby
- PHP5
- Object Pascal/Delphi
- OCaml
- …

http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages
Oct. 2011

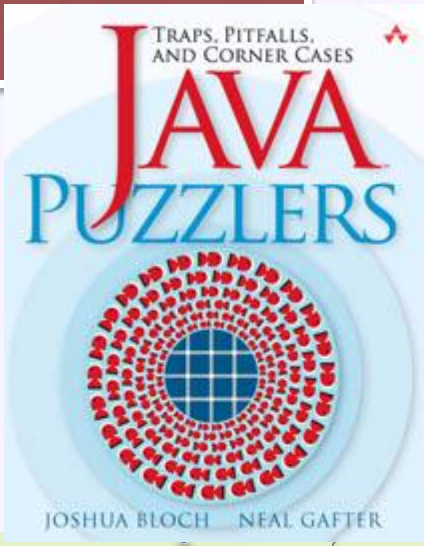


This is not a Java course

but you will be writing a lot of Java code

```
int a = 010 + 3;
System.out.println("A" + a);
```

```
int a = 010 + 3;
System.out.println("A" + a);
```



Learning Java

- Books
  - Head First Java (CMU libraries)
  - Introduction to Java Programming
  - Introduction to Programming Using Java (free online textbook)
  - Blue Pelican Java (free online textbook)
  - Effective Java
- Lots of resources online…
- Java API Documentation
- Ask on Piazza for tips

Concepts of Object-Oriented Languages: Overview

- Sending messages
- Objects and References
- Encapsulation (Visibility)
- Polymorphism
  - Interfaces
  - Method Dispatch
- Object Equality

Sending Messages

Objects

- A package of state (data) and behavior (actions)

- Can interact with objects by sending messages
  - perform an action (e.g., move)
  - request some information (e.g., getSize)

```
Point p = …
int x = p.getX();
```

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

- Possible messages described through an interface

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

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

Implementing Objects

(subtype polymorphism)

Subtype Polymorphism

- 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

Creating Objects

```
interface Point {
  int getX();
  int getY();
}
Point p = new Point() {
  int getX() { return 3; }
  int getY() { return -10; }
}
```

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 Point implements 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; }
}
Point p = new CartesianPoint(3, -10);
```

More Classes

```
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);
```

Polar Points

```
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);
```

Implementation of interfaces

- Classes can *implement* one or more interfaces.

```
public class PolarPoint implements Point, IPolarPoint {…}
```

- **Semantics**
  - **Must provide code** for all methods in the interface(s)

Polar Points

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

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

```
class PolarPoint implements Point, IPolarPoint {
   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() {… }
}
IPolarPoint p = new PolarPoint(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));
```

Example: Points and Rectangles

Subtype Polymorphism

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

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 IRectangle interface?

Java **interfaces** and **classes**

Object-orientation

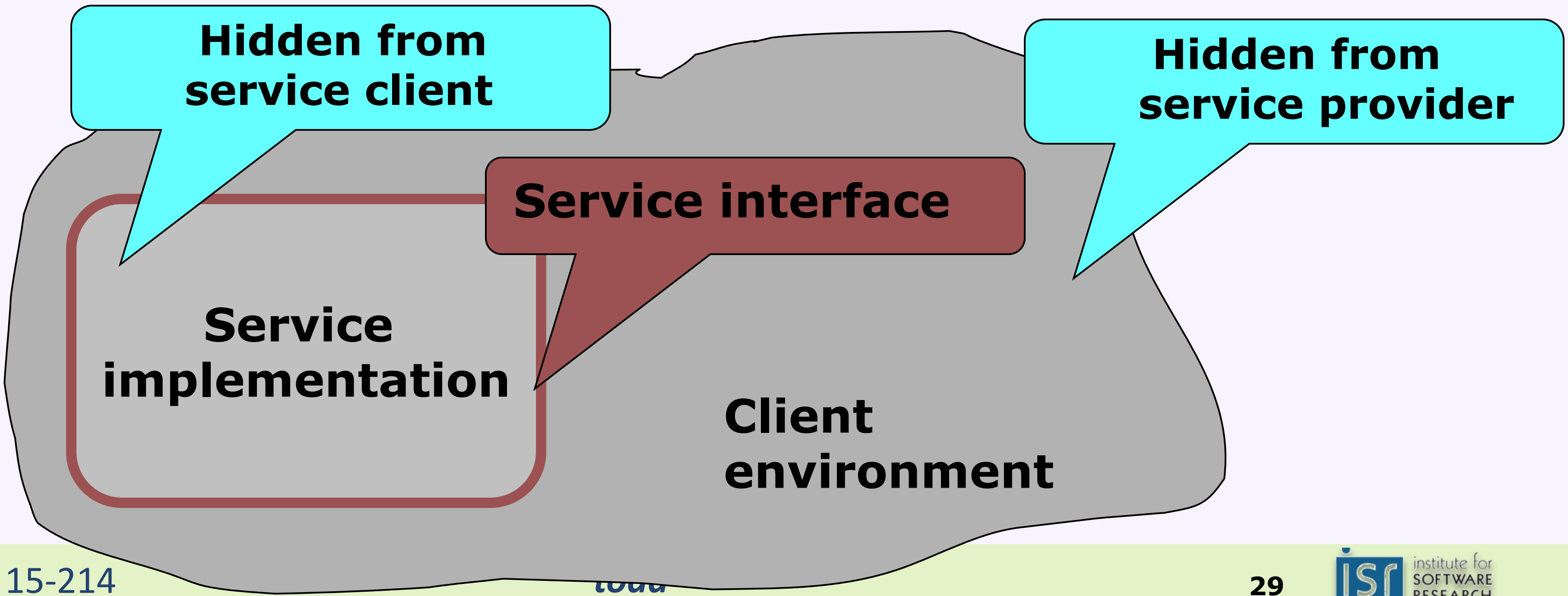
1. Organize program functionality around kinds of abstract “objects”
   - For each object kind, offer a specific set of operations on the objects
   - Objects are otherwise opaque
     - Details of representation are hidden
   - “Messages to the receiving object”
2. Distinguish *interface* from *class*
   - **Interface**: expectations
   - **Class**: delivery on expectations (the implementation)
   - **Anonymous class**: special Java construct to create objects without explicit classes
     Point x = new Point() { / implementation */ };*
3. Explicitly represent the taxonomy of object types
   - This is the type hierarchy *(!= inheritance, more on that later)*
     - A **PolarPoint** is a **Point**

Encapuslation

(Visibility)

Contracts and Clients

- Contract of service provider and client
  - Interface specification
  - Functionality and correctness expectations
  - Performance expectations
  - Hiding of respective implementation details
  - "Focus on **concepts** rather than **operations**"



Controlling Access

- Best practice:
  - 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
- Classes usable as type
  - Methods in classes usable as methods in interfaces
  - Even fields directly accessable
  - Access to methods and fields in classes can be private or public
  - Private methods and fields only accessible within the class
- Prefer programming as an interface (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

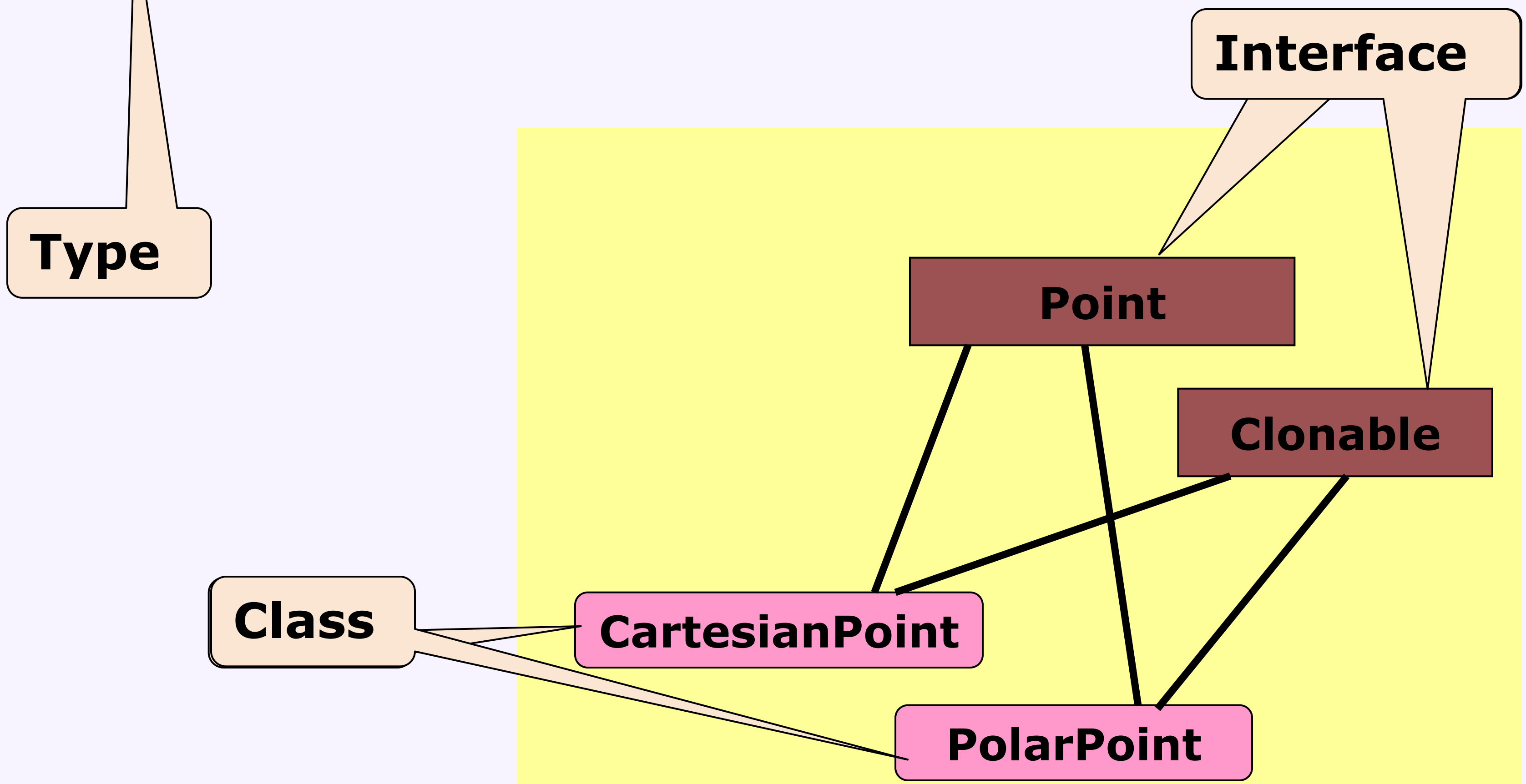
```
int add(PolarPoint list) { …      // preferably no
int add(Point list) { …           // yes!
```

Interfaces and Classes both usable as Types

- Two ways to put an object into a variable

```
Point p = new CartesianPoint(3,5);
PolarPoint pp= new PolarPoint(5, .353);
```

Class



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();
PolarPoint pp = new PolarPoint(5, .245);
pp.getX();
pp.getAngle();
```

Controlling access by client code

```
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 creating the object
}
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 i
    public in
    public Po
}
class Recta
    private 
    private int width, height;
    public Point getOrigin() { return origin; }
    public in
    public v
            c

            ..
    }
    public R
            c
    }
}
```

Some Client Code

```
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
```

Client Code that will *not* work in this version

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

Discussion:

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

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

```
class APoint {
  int x,y;
}

APoint p = new APoint();
p.x=3;
p.y=-10;
```

```
class BPoint {
  int x,y;
  BPoint(int x, int y)
        {this.x=x; this.y=y;}
}

BPoint p = new BPoint(3, -10);
```

Breaking encapsulation: instanceof and typecast

- Java allows to inspect an object's runtime type

```
Point p = …
if (p instanceof PolarPoint) {
    PolarPoint q = (PolarPoint) p;
    q.getAngle()
}
```

- Objects always assignable to variables of supertypes ("**upcast**")

(this effectively throws away parts of the interface)

```
PolarPoint q = …
Point p = q;
```

- Assignment to subtype requires **downcast** (may fail at runtime!)

```
Point p = …
PolarPoint q = (PolarPoint) p;
```

Avoid instanceof and downcasts

Instanceof breaks encapsulation

- Never ask for the type of an object
- Instead, ask the object to do something (call a method of the interface)
- If the interface does not provide the method, maybe there was a reason? Rethink design!

- Instanceof and downcasts are indicators of poor design
- They break abstractions and encapsulation
- There are only few exceptions where **instanceof** is needed
- Use polymorphism instead

- Pure object-oriented languages do not have an instanceof operation

Object-Oriented Programming 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

Excursion: Objects vs ADTs

```
interface Point {
        int getX();
        int getY();
}
class CartesianPoint
        implements Point { … }
class PolarPoint
        implements Point { … }

Point p = …
p.getX()
```

```
class CartesianPoint { … }
class PolarPoint { … }

Object p = …
if (p instanceof CartesianPoint)
   return ((CartesianP.)p).x;
if (p instanceof PolarPoint)
   return ((PolarPoint)p).r*…;
```

```
datatype point
   = CartesianP of int * int
   | PolarPoint of real * real

fun getX point =
  case shape
     of CartesianP (x, _) => x
     | PolarPoint (r, a) => r*…
```

Excursion: Objects vs ADTs

```
interface Point {
        int getX();
        int getY();
}
class CartesianPoint
        implements Point { … }
class PolarPoint
        implements Point { … }

Point p = …
p.getX()
```

- OOP solution with polymorphism
  - Easy to extend with new implementations of interface
  - Functions fixed; adding a function to the interface requires changes in all implementations

```
class CartesianPoint { … }
class PolarPoint { … }

Object p = …
if (p instanceof CartesianPoint)
   return ((CartesianP.)p).x;
if (p instanceof PolarPoint)
   return ((PolarPoint)p).r*…;
```

- ADT solution with case analysis/ pattern matching
  - ADTs fixed; cannot add new class without changing all functions
  - Easy to add new functions
  - No language/compiler support in Java

Dynamic Dispatch
(subtype polymorphism)

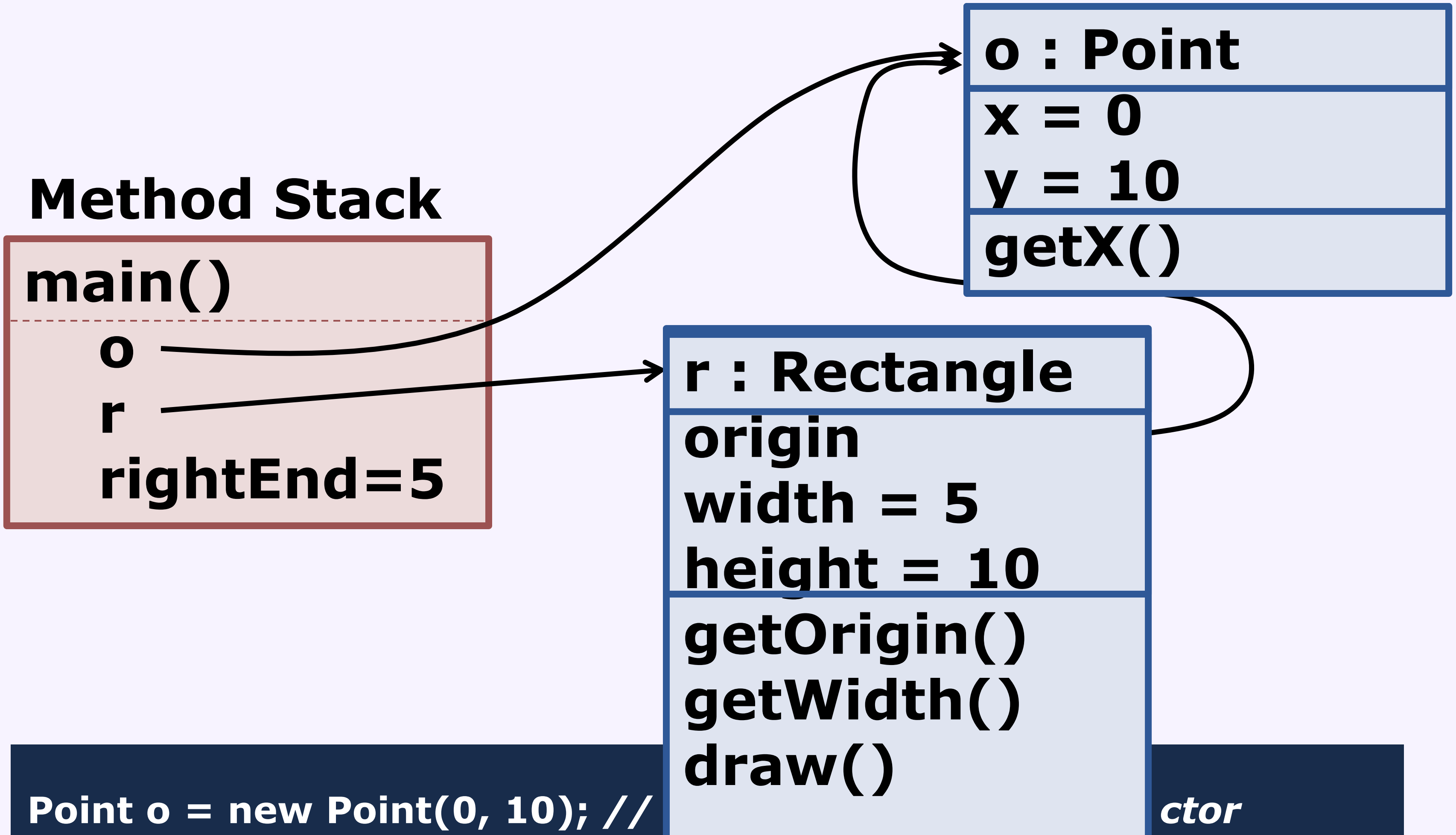
(Subtype) Polymorphism

- 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 ctor
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

Method **arguments**, just like function arguments

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

Java Specifics: The keyword **this** refers to the “receiver”

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

can also be written in this way:

```
class Point {
   int x, y;
   int getX() { return x; }
   Point(int px, int py) { x = px; y = py; }
}
```

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 p = createZeroPoint();
p.getX();
p.getAngle();

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

Method dispatch (simplified)

Example:

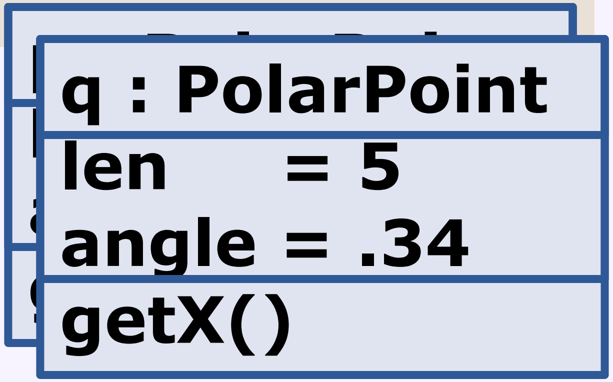
```
Point p = new PolarPoint(4, .34);
p.getX();
p.getAngle();
```

- 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
    - Later: there may be more than one such method

      int getX();

  - Keep the method only if it is *accessible*
    - e.g. remove private methods
  - Error if there is no such method

Method dispatch (conceptually)

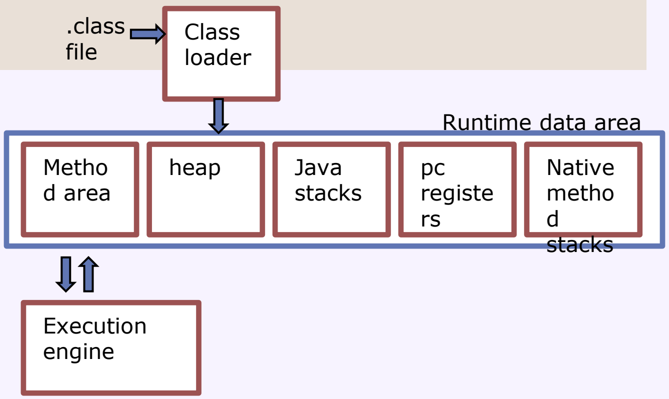


Example:

```
Point p = new PolarPoint(4, .34);
p.getX();
```

- Step 3 (run time): Execute the method stored in the object

Method dispatch (actual; simplified)



Example:

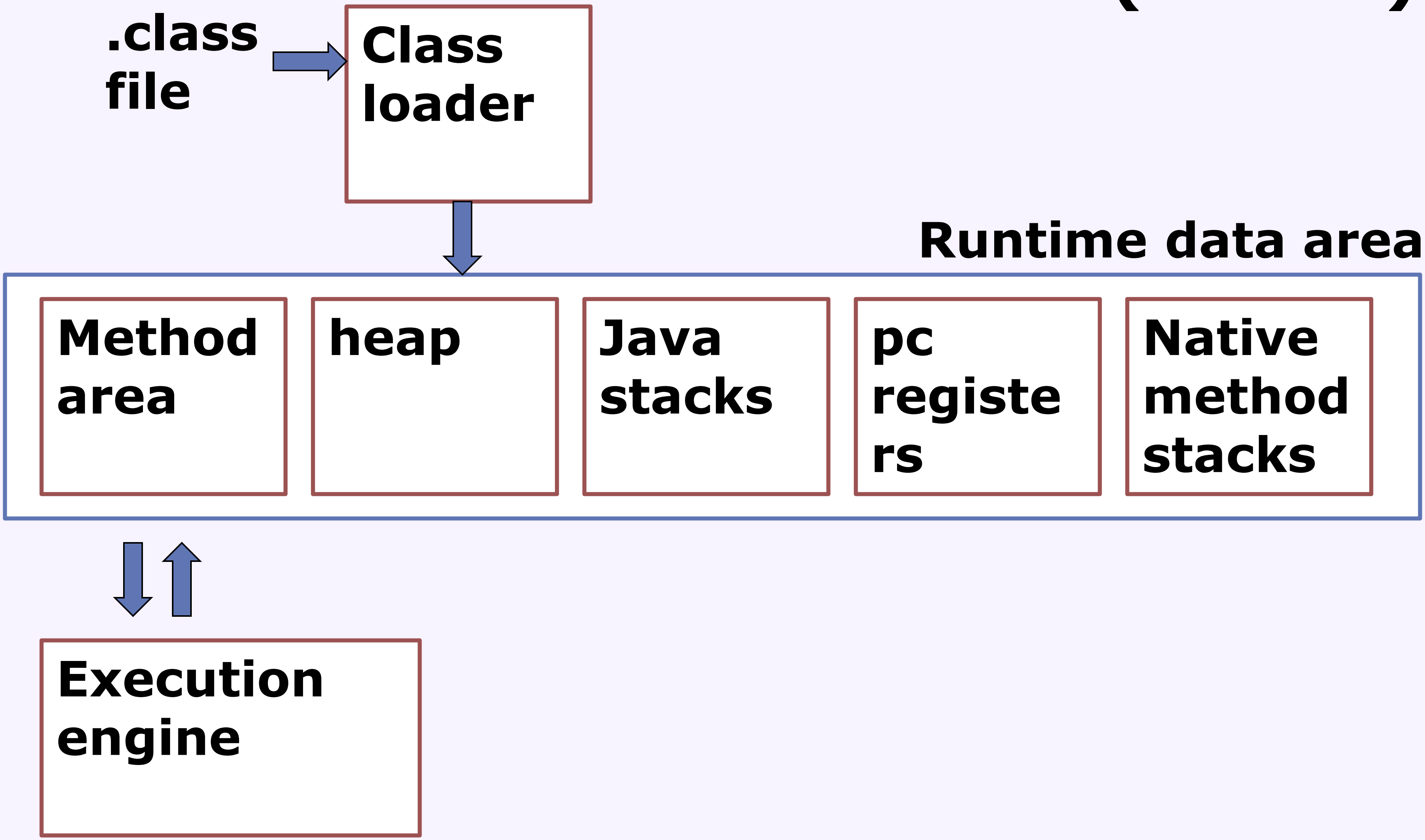
```
Point p = new PolarPoint(4, .34);
p.getX();
```

- 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 we found statically (step 2)

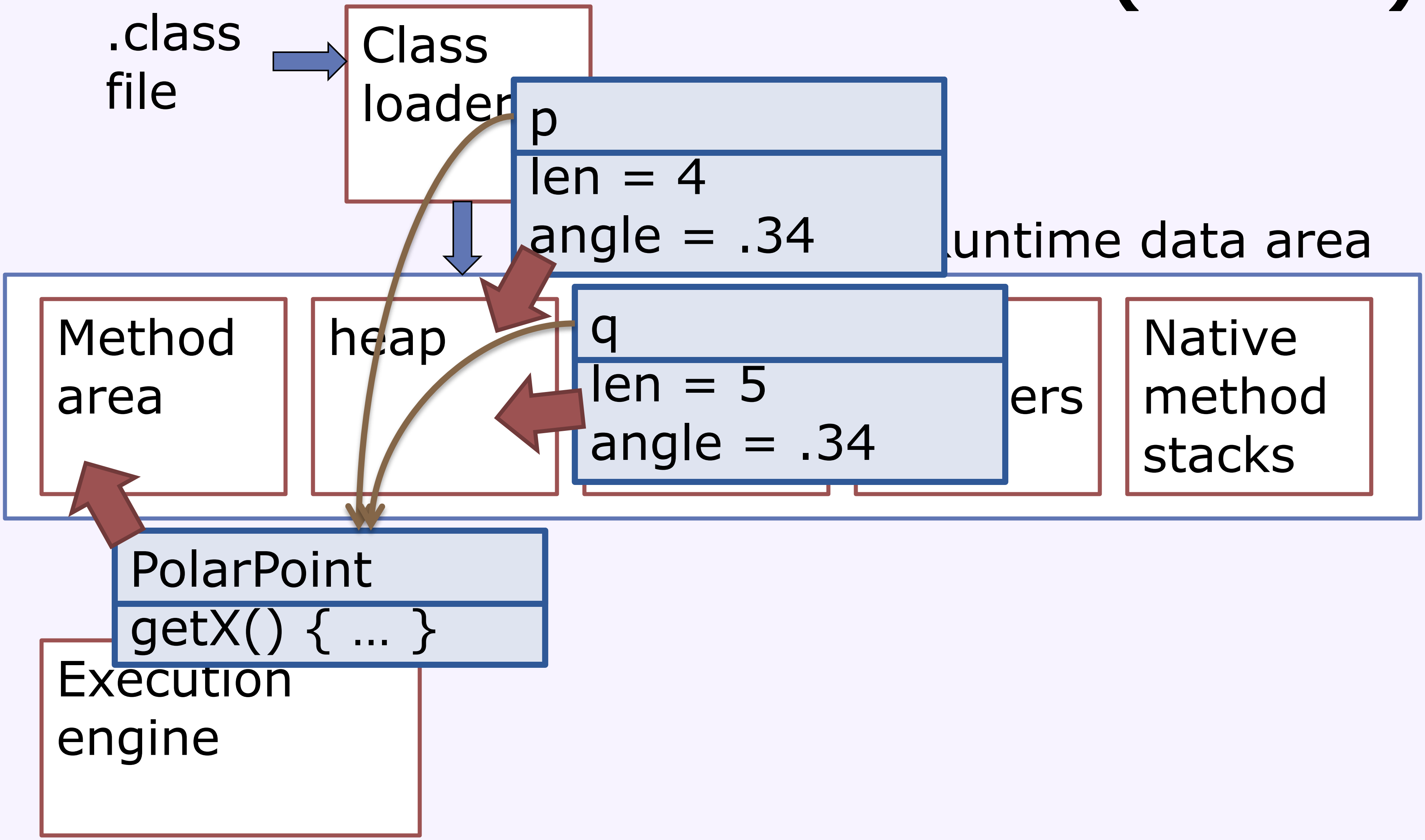
```
int getX() { return this.len * cos(this.angle);}
```

  - Invoke the method

The Java Virtual Machine (sketch)



The Java Virtual Machine (sketch)



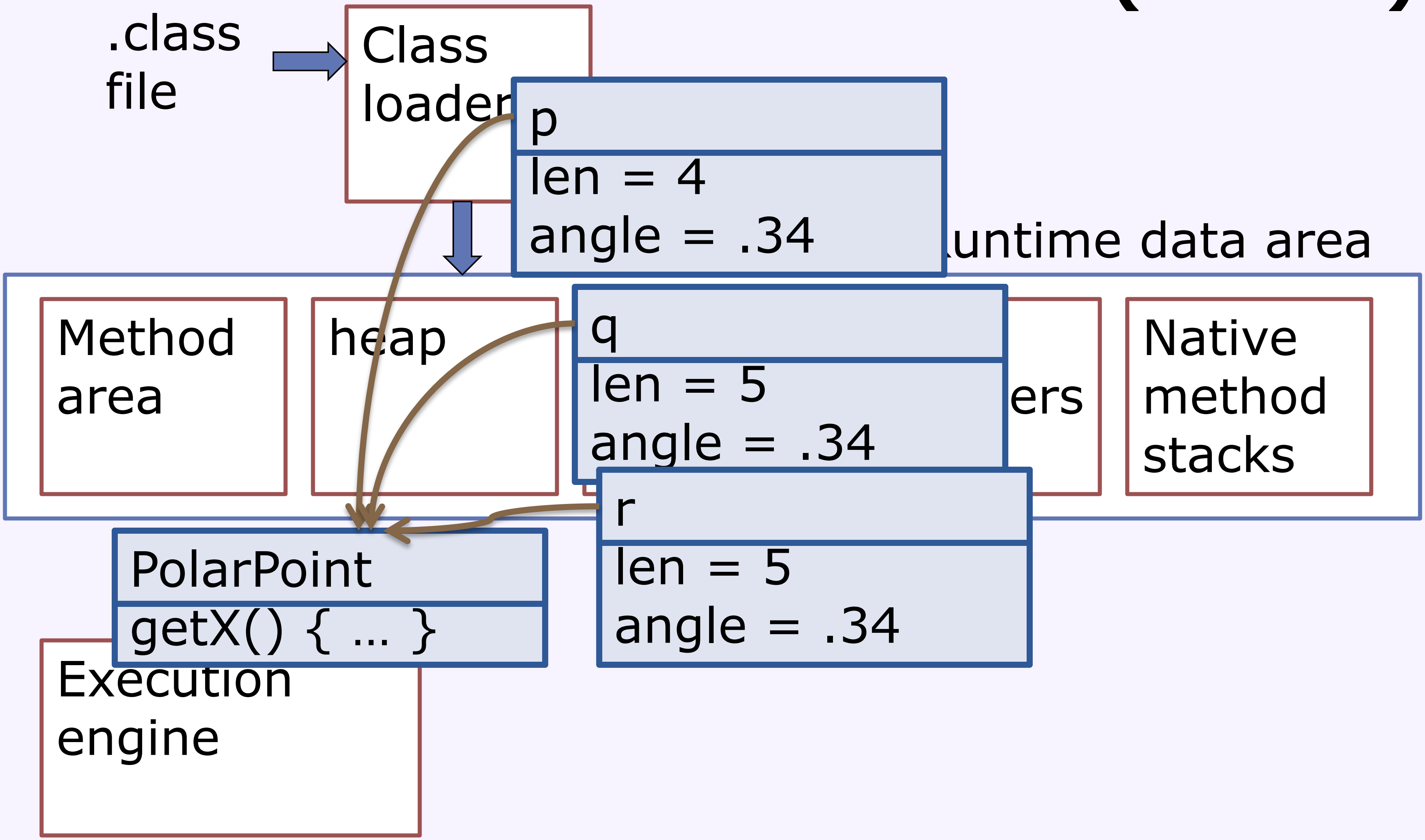
Check your Understanding

```java
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 a = new Animal();
a.makeSound();
Dog d = new Dog();
d.makeSound();
Animal b = new Cow();
b.makeSound();
b.mew();
Cow c = b;
c.mew();
```

- What does this program return?
- Are there compile-time problems?

Object Identity & Object Equality

The Java Virtual Machine (sketch)



Object Identity vs. Object Equality

- There are two notions of equality in most OO languages
- Every object is created with a unique identity

```
Point a = new PolarPoint(1,1); // new object
Point b = a; // same reference, same object
Point c = new PolarPoint(1,1); // new object
Point d = new PolarPoint(1,.9999999); // new object
```

  - Comparing object identity compares references
    in Java: a == b, a != c
- Object equality is domain specific
  - When are two points equal?

    a.equals(b), c.equals(a), d.equals(a)?

  - Developer needs to provide own equals functions
  - Java provides a contract for equal
  - Equals hard to implement correctly (more on this later)
- Object identity faster to decide (comparing references instead of calling functions)

Strings are weird!

- The *same object.* References are the same.
- Possibly different objects, but equivalent content
  - From the client perspective!! The actual internals might be different

```
String s1 = new String ("abc");
String s2 = new String ("abc");
```

- There are two string objects, s1 and s2.
  - The strings are are equivalent, but the references are different

```
if (s1 == s2) { same object } else { different objects }

if (s1.equals(s2)) { equivalent content } else { not}
```

Defined in the class String

- An interesting wrinkle: *literals*

```
String s3 = "abc";
String s4 = "abc";
```

- These are true: s3==s4. s3.equals(s2). s2 != s3.

How to implement equals?

- All Java objects have "boolean equals(Object o)" method
  - by default checks for object identity only

- Assumptions on "equals"
  - Defined as intended contract in the Java standard
  - Reflexive: $\forall$x x.equals(x)
  - Symmetric: $\forall$x,y x.equals(y) if and only if y.equals(x)
  - Transitive: $\forall$x,y,z x.equals(y) and y.equals(z) implies x.equals(z)
  - Consistent: Invoking x.equals(y) repeatedly returns the same value unless x or y is modified
  - x.equals(null) is always false
  - always terminating and side-effect free

- Hard to do correctly with subclassing, delegation, and objects of different classes

Equal points

- Java's equals method "boolean equals(Object o)"
- Typecast needed when comparing any specifics

```
class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    boolean equals(Object o) {
            if (!(o instanceof CartesianPoint)) return false;
            CartesianPoint that = (CartesianPoint) o;
            return (this.getX() == that.getX()) &&
                    (this.getY() == that.getY());
    }
}
```

Equal points

- Java's equals method "boolean equals(Object o)"
- Typecast needed when comparing any specifics

```
class CartesianPoint implements Point {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    boolean equals(Object o) {
            if (!(o instanceof Point)) return false;
            Point that = (Point) o;
            return (this.getX() == that.getX()) &&
                    (this.getY() == that.getY());
    }
}
```

Equals only symmetric if all objects implement equals like this

One more thing: Hashcode

- Fast search?
  - Sorting or hashing
- Many Java libraries use hashing
- Requires that equal (and identical) objects have the same hash
- Every Java object has "int hashCode()" function
  - Object provides hash, not external function
  - by default, hash based on object identity
- Java specification: Equal objects return the same hash!
  - `x.equals(y)` implies `x.hashCode() == y.hashCode()`
  - (same hash code does not imply equality though)
  - Consistency: repeatedly calling hashCode returns the same value
- Whenever providing an "equals" function **always** provide a corresponding "hashCode" function

Equal points

- Java's equals method "boolean equals(Object o)"
- Typecast needed when comparing any specifics

```
class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    boolean equals(Object o) {
            if (!(o instanceof CartesianPoint)) return false;
            CartesianPoint that = (CartesianPoint) o;
            return (this.getX() == that.getX()) &&
                   (this.getY() == that.getY());
    }
    int hashCode() { return x + y*7; }
}
```

Advice: If sets and maps are behaving funny, check whether you have implemented hashCode

Equal points

- Java's equals method "boolean equals(Object o)"
- Typecast needed when comparing any specifics

```
class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    boolean equals(Object o) {
        if (!(o instanceof CartesianPoint)) return false;
        CartesianPoint that = (CartesianPoint) o;
        return (this.getX() == that.getX()) &&
               (this.getY() == that.getY());
    }
    int hashCo
}
```

What happens if we use
int hashCode() { return 0; }
?

Check your understanding

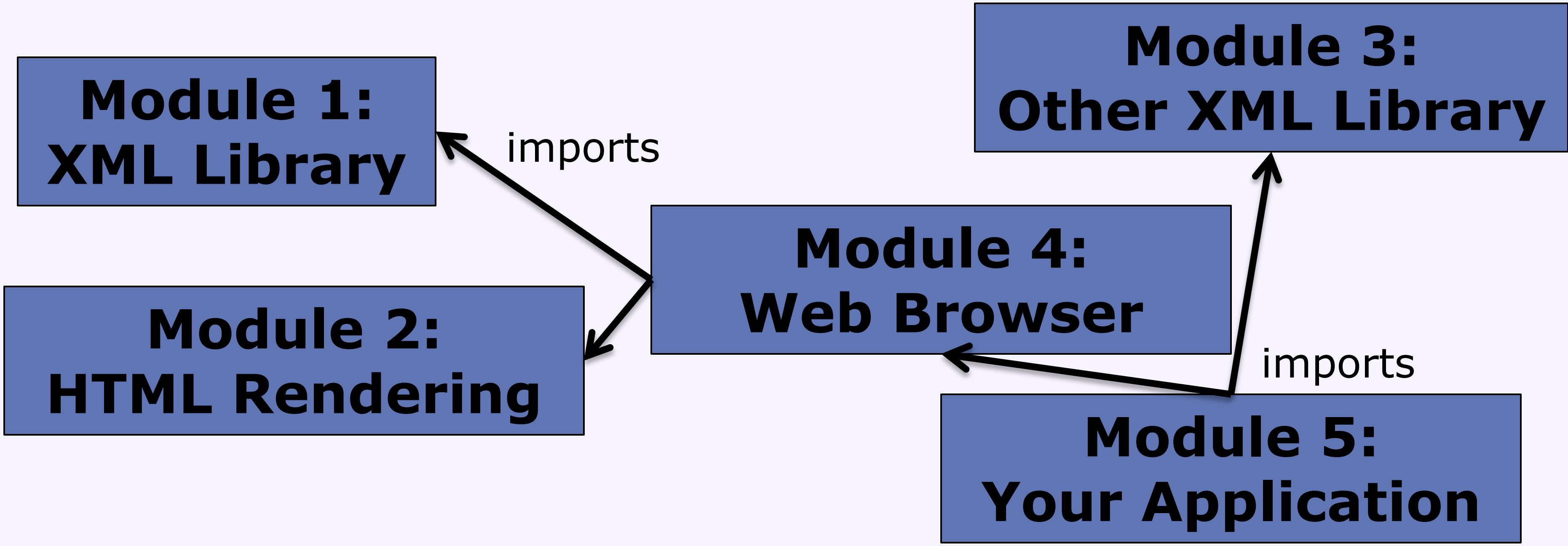
- Complete this class to support object equality checks

```
class Person {
    private String firstName, lastName;
    private int ssn;
    Person(String name, int ssn) {
            this.firstName = name.split(" ")[0];
            this.lastName = name.split(" ")[1];
            this.ssn = ssn;
    }
















}
```

Modules

Module Systems

- Many languages have a module system
  - Modules can be developed independently
  - Modules encapsulate functionality behind interfaces, own internal name space
  - Modules can be composed (importing or linking)
  - Type errors are checked within modules



Java's "module system"

- Classes/Objects encapsulate methods and fields
- No module imports
- Global name space (worldwide)
- Avoid name clashes by naming conventions
  - edu.cmu.cs.214.assignment1.Graph
  - Fully qualified names, typically including domain names

```
new edu.cmu.cs.214.assignment1.Graph(
            new java.util.List(...));
```

Java's Imports

- Imports as shorthand for not having to write fully qualified names

```
new edu.cmu.cs.214.assignment1.Graph(new java.util.List(…));
```

↓

```
import edu.cmu.cs.214.assignment1.Graph;
import java.util.*;

new Graph(new List(…));
```

- Fully qualified names may still be used, especially if multiple types with the same name are in scope
(Compiler will warn about ambiguous references)

```
import java.util.*;

new edu.cmu.mylist.List(new List(…));
```

Java's Packages

- Every substring in a fully qualified name corresponds to a package
- Package represented with folders (IDEs offer better abstractions)
- In practice: **Organize related classes in package**
- Packages have extra visibility mechanisms:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default (no modifier)	Y	Y	N	N
private	Y	N	N	[illegible]

With package visibility, everybody can still place a class in your package to gain access

Encapsulation design principles

- Restrict accessibility as much as possible
  - Make data and methods `private` unless you have a reason to make it more visible
  - Use interfaces to abstract from implementations

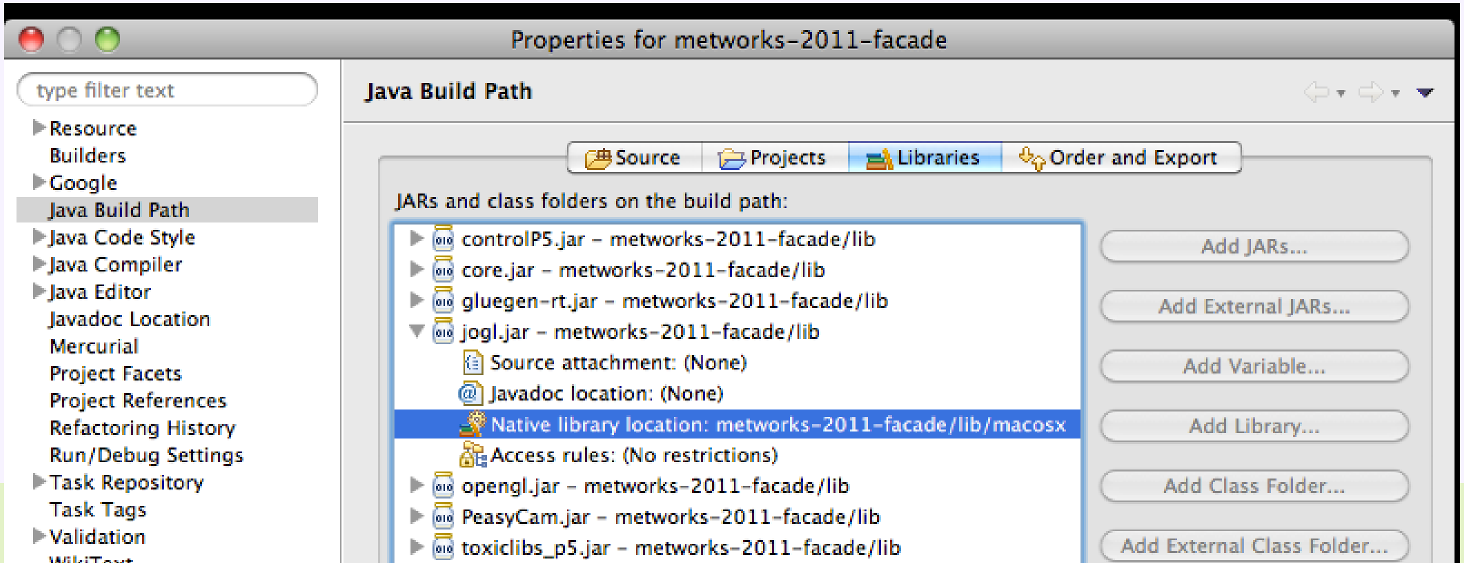
  - -> Easier to develop, test, understand, debug, use, and optimize code in isolation.

"The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details." -- Josh Bloch

Class Loading

- All classes in the class path are accessible through imports or fully qualified names (modulo visibility)
- .jar files contain bundled classes
  - essentially just a .zip file
- Adding classes to class path when starting the JVM

java –cp /home/xanadu:lib/parser.jar:. Main



Module Systems for Java

- Java provides deep hooks into how classes are loaded
- Separate module systems exist
  - OSGi commonly used, e.g., in Eclipse
- Ongoing discussions for Java 9 (JSR 277)
- External module systems have a separate module construct and separate access control
- Classes with the same (fully qualified) name can coexist (e.g. revisions)

Object orientation (OO)

- History
  - Simulation – Simula 67, first OO language
  - Interactive graphics – SmallTalk-76 (inspired by Simula)

- Object-oriented programming (OOP)
  - Organize code bottom-up rather than top-down
  - Focus on **concepts** rather than **operations**
  - Concepts include both **conventional data types** (e.g. List), and **other abstractions** (e.g. Window, Command, State)

- Some benefits, informally stated
  - Easier to reuse concepts in new programs
    - Concepts map to ideas in the target domain
  - Easier to extend the program with new concepts
    - E.g. variations on old concepts
  - Easier to modify the program if a concept changes
    - **Easier** means the changes can be **localized** in the code base

Toad's Take-Home Messages

- **Objects** correspond to things/concepts of interest
- An **interface** states expectations for an object
- Objects embody:
  - State – held in **fields**, which hold or reference data
  - Actions – represented by **methods**, which describe operations on state
  - **Constructors** – how objects are created
- A **class** is a template for creating objects
- Subtype polymorphism allows different implementations of the same interface; method selected at runtime
- Encapsulation hides implementation internals from users
- Object equality is different from object identity, equality and hashcode
- Fully qualified names, packages, and imports to structure the name space