toad

Spring 2013

Principles of Software Construction: Objects, Design, and Concurrency

Objects

Charlie Garrod **Christian Kästner**

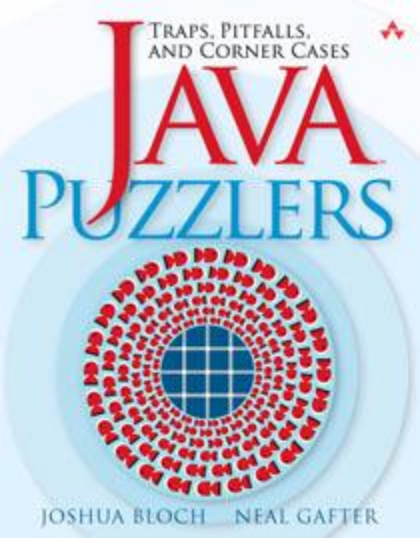
School of Computer Science

institute for SOFTWARE RESEARCH

Recap Tuesday

- 214: managing complexity, from programs to systems
  - **T**hreads and concurrency
  - **O**bject-oriented programming
  - **A**nalysis and modeling
  - **D**esign
- Object-oriented programming organizes code around **concepts**
  - Objects contain state and behavior
  - Methods capture behavior, fields capture state
  - Classes as template for objects
  - As we will see, this organization allows
    - Greater reuse of concepts
    - Better support for change when concepts vary

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



Object-Oriented Programming Languages

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

Agenda

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

Objects and References

(heap representation)

Example: Points and Rectangles

```
class Point {
    int x, y;
    int getX() { return this.x; } // a method; getY() is similar
    Point(int px, int py) {this.x = px; this.y = py; }  // constructor for creating the object
}
class 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
    }
    Rectangle(Point o, int w, int h) {
            this.origin = o; this.width = w; this.height = h;
    }
}
```

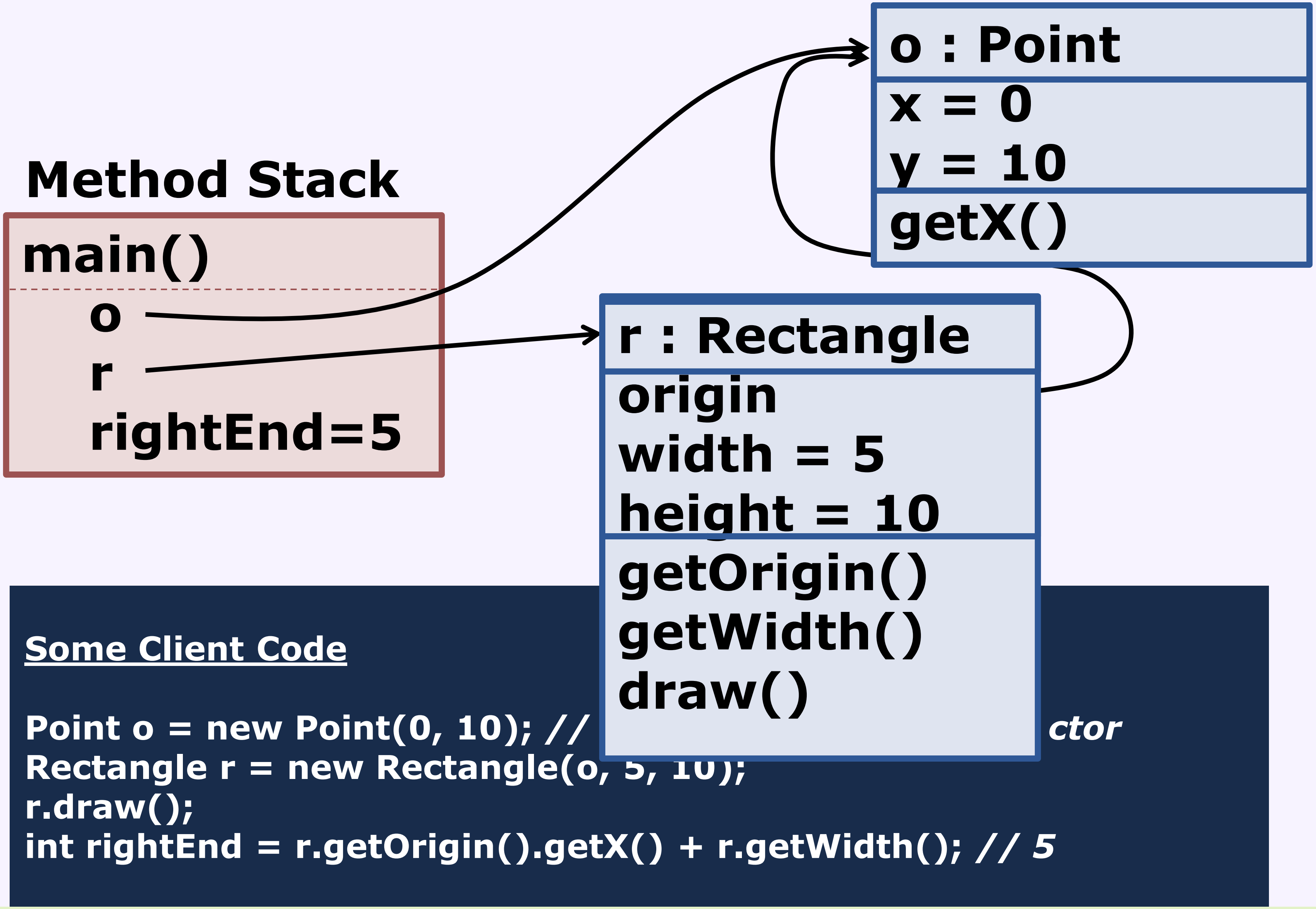
Example: Points and Rectangles

```
class Point {
    int x, y;
    int getX() { return this.x; } // a method; getY() is similar
    Point(int px, int py) {this.x = px; this.y = py; }  // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {



    }
    R


    }
}
```

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
```

What’s really going on?



Method Stack

main()

o

r

rightEnd=5

o : Point

x = 0

y = 10

getX()

r : Rectangle

origin

width = 5

height = 10

getOrigin()

getWidth()

draw()

Some Client Code

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

Encapuslation

(Visibility)

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 i
    public P
}
class Recta
    private
    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 Point {
  int x,y;
}

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

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

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

Polymorphism

Points and Rectangles: Interface

```
interface IPoint {
  int getX();
  int getY();
}
interface IRectangle {
  IPoint getOrigin();
  int getWidth();
  int getHeight();
  void draw();
}
```

Cartesian Points

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

Strange Points

```
interface IPoint {
   int getX();
   int getY();
}
class SkewedPoint implements IPoint {
   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; }
}
IPoint p = new SkewedPoint(3, -10);
```

Polar Points

```
interface IPoint {
   int getX();
   int getY();
}
class PolarPoint implements IPoint {
   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() {…}
}
IPoint p = new PolarPoint(5, .245);
```

Polar Points

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

Example: Points and Rectangles

Polymorphism

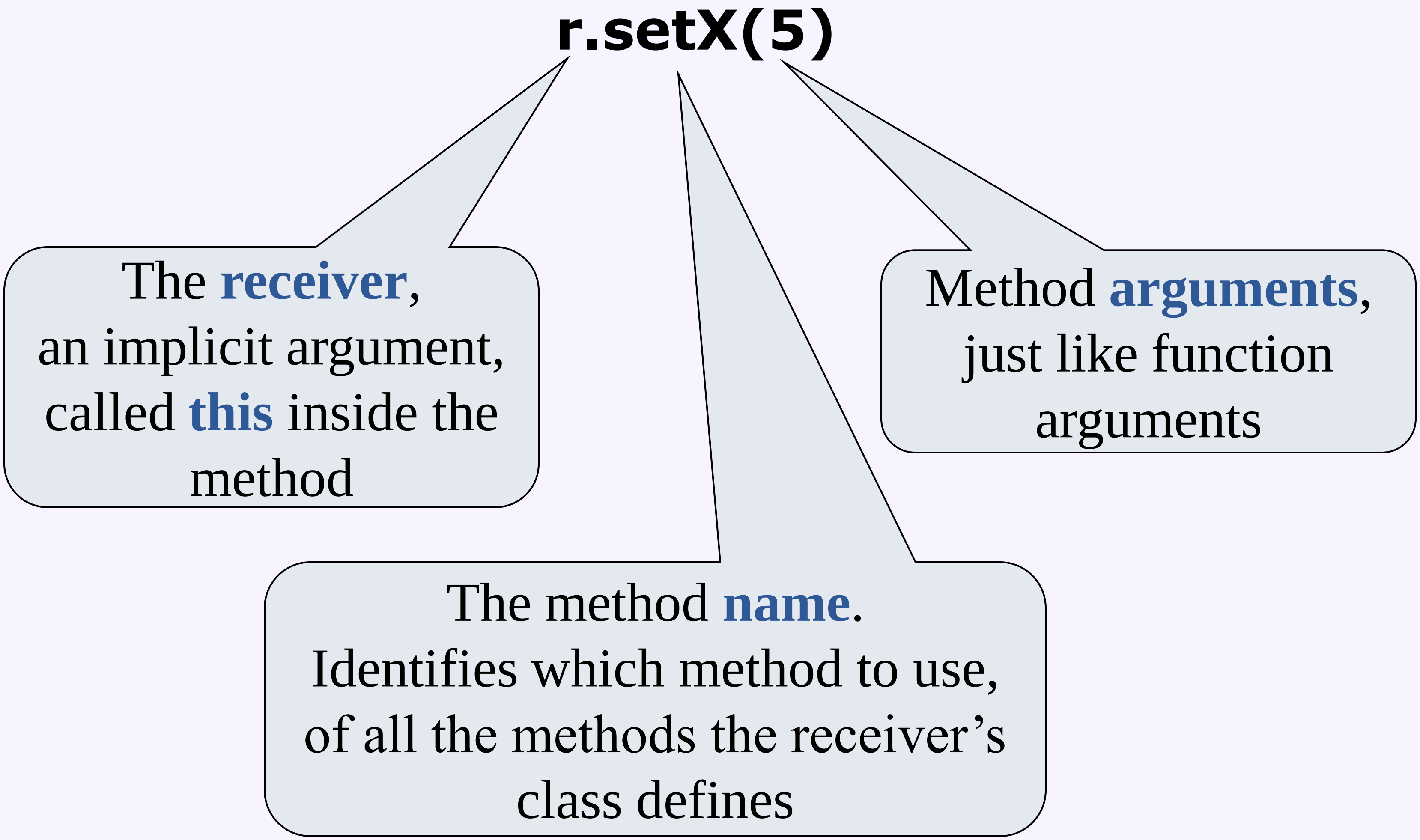
```
interface IPoint {
    int getX();
    int getY();
}
class Rectangle {
    IPoint origin;
    int width, height;
    IPoint getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {
            this.drawLine(this.origin.getX(), this.origin.getY(),           //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

Points and Rectangles: Interface

```
interface IPoint {
  int getX();
  int getY();
}
interface IRectangle {
  IPoint getOrigin();
  int getWidth();
  int getHeight();
  void draw();
}
```

What are possible implementations of the IRectangle interface?

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

The keyword **this** refers to the "receiver"

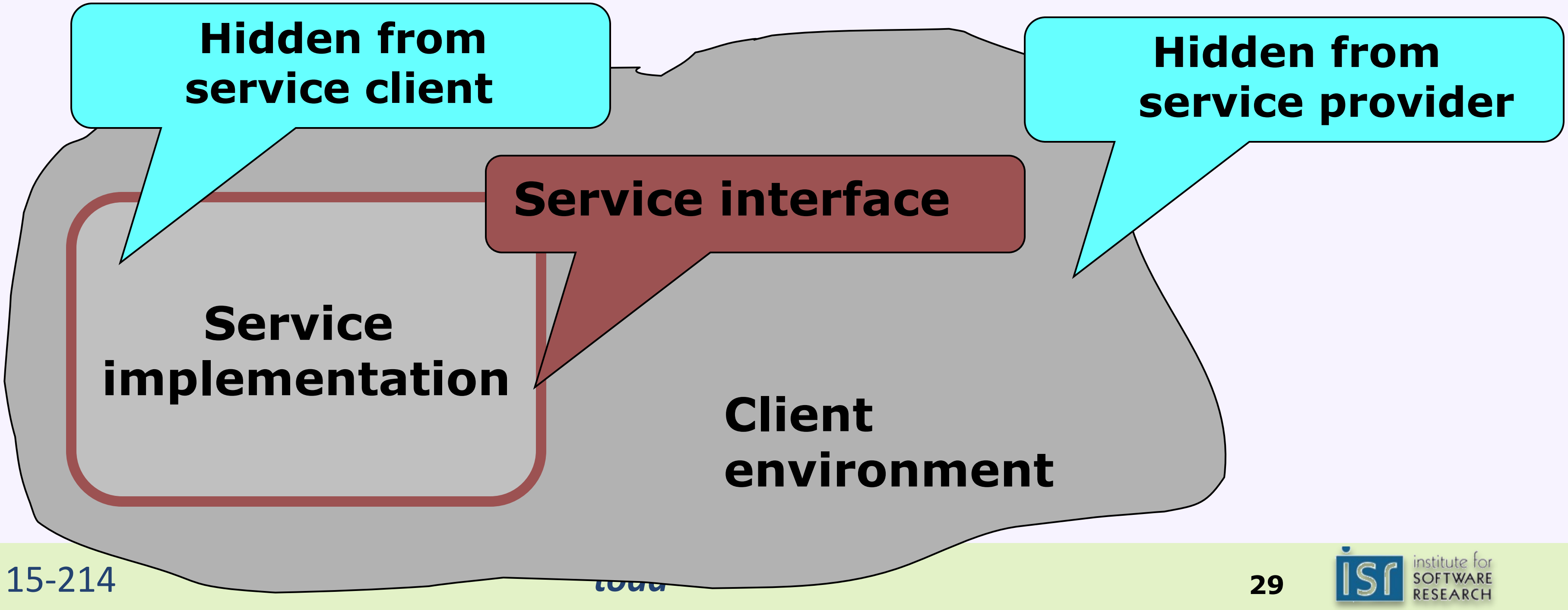
```
class Point {
  int x, y;
  int getX() { return this.x; }
  Point(int x, int y) { this.x = x; this.y = y; }
}
```

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; }
}
```

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**”



Hidden from service client

Hidden from service provider

Service interface

Service implementation

Client environment

Interfaces state Expectations

```
interface IPoint {
  int getX();
  int getY();
}
interface IRectangle {
  IPoint getOrigin();
  int getWidth();
  int getHeight();
  void draw();
}
```

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)
3. Explicitly represent the taxonomy of object types
    - This is the “inheritance hierarchy”
        - A **square** is a **shape**

Interfaces and Classes (Review)

```
interface IPoint {
  int getX();
  int getY();
}
class PolarPoint implements IPoint {
  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); }
}
IPoint p = new PolarPoint(5, .245);
PolarPoint p = new PolarPoint(5, .245);
```

Implementation of interfaces

- Classes can *implement* one or more interfaces.

```
public class PolarPoint implements IPoint, Cloneable {…}
```

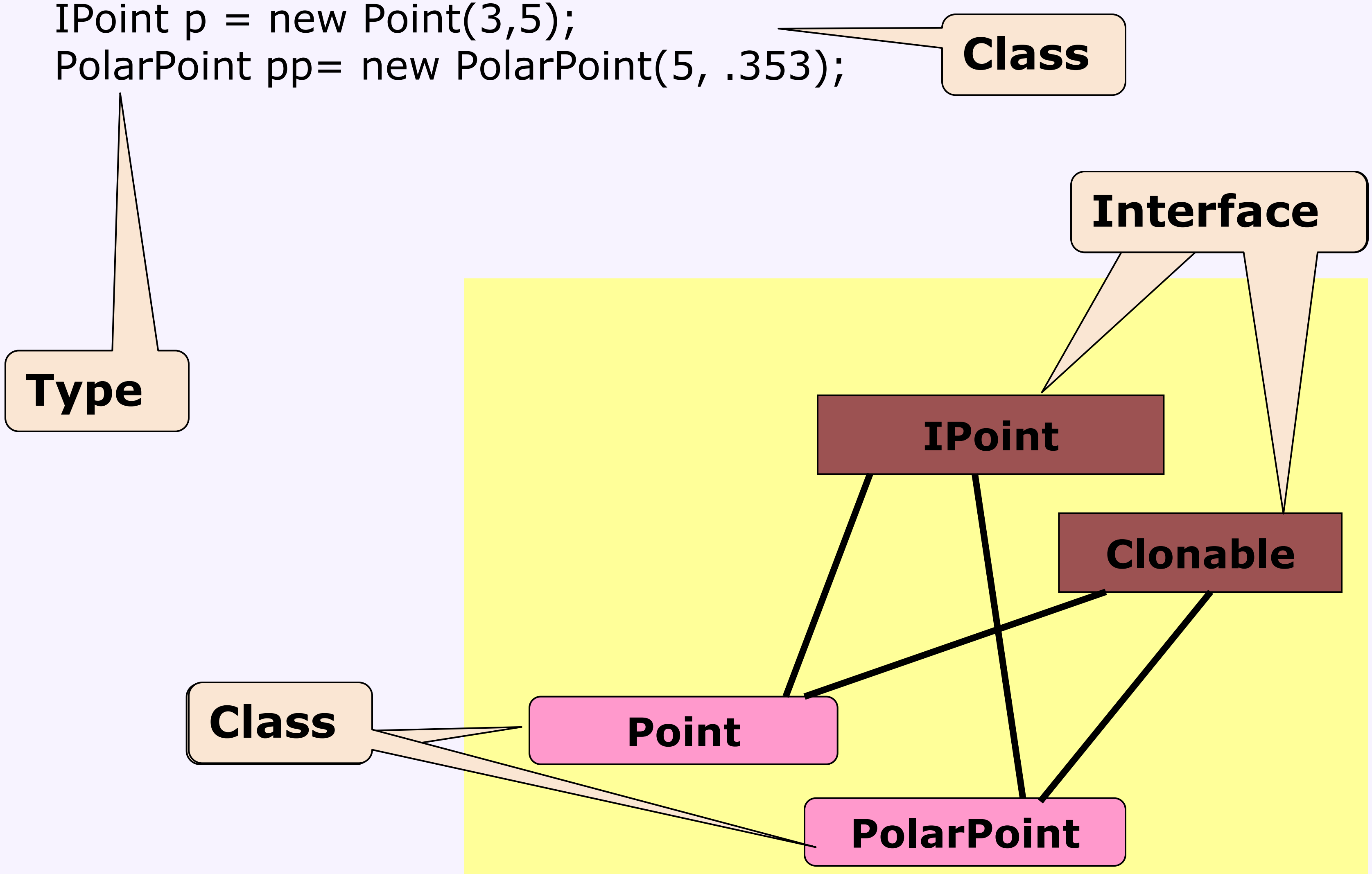
- **Semantics**
  - **Must provide code** for all methods in the interface(s)
- **Best practices**
  - Define an interface whenever there may be **multiple implementations** of a concept
  - Variables should have **interface type**, not class type

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

Interfaces, Types, Classes

- Two ways to put a new empty list into a variable

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



Interfaces and Classes (Review)

```
class PolarPoint implements IPoint {
   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; }
}
IPoint p = new PolarPoint(5, .245);
p.getX();
p.getAngle();
PolarPoint pp = new PolarPoint(5, .245);
pp.getX();
pp.getAngle();
```

Method dispatch (simplified)

Example:

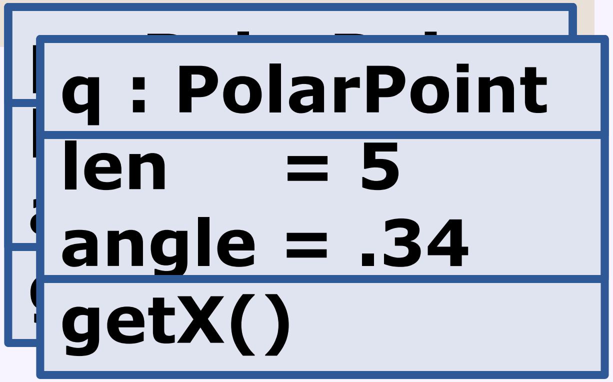
```
IPoint p = new PolarPoint(4, .34);
p.getX();
p.getAngle();
```

- Step 1 (compile time): determine what type to look in
    - Look at the static type (IPoint) 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)



q : PolarPoint

len = 5

angle = .34

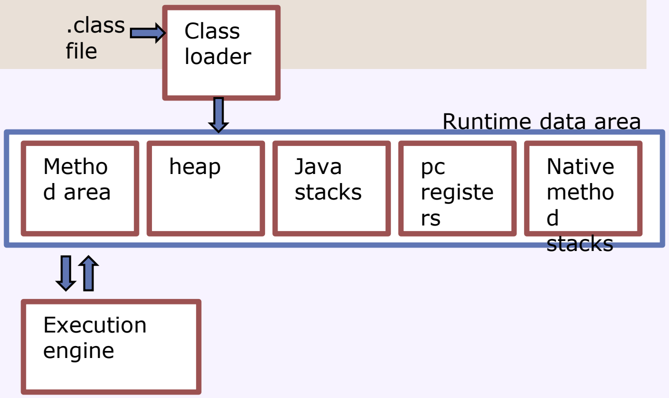
getX()

Example:

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

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

Method dispatch (actual; simplified)



.class file

Class loader

Runtime data area

Metho d area

heap

Java stacks

pc registe rs

Native metho d stacks

Execution engine

Example:

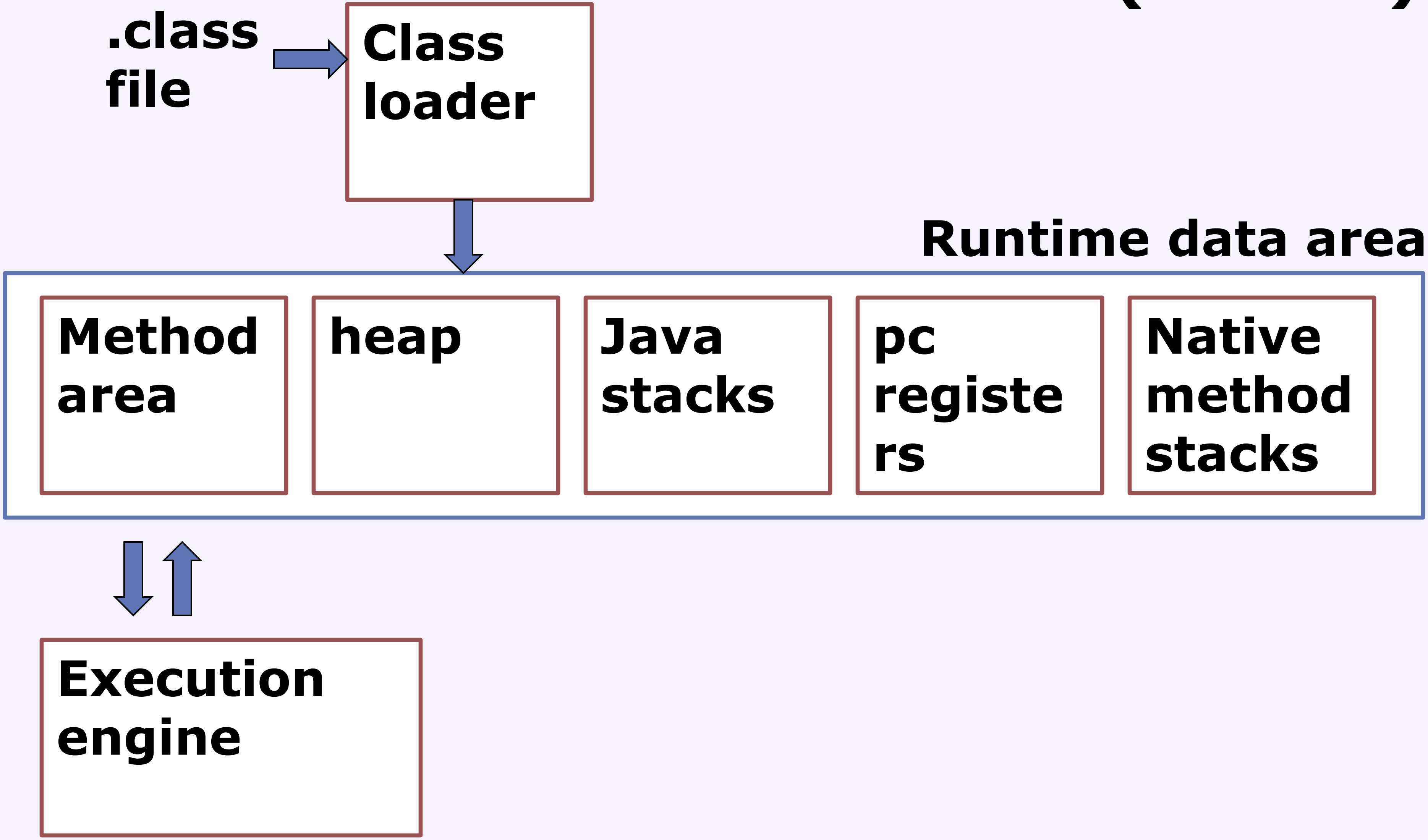
```
IPoint p = new PolarPoint(4, .34);
p.getX();
```

- 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)



.class file

Class loader

Runtime data area

Method area

heap

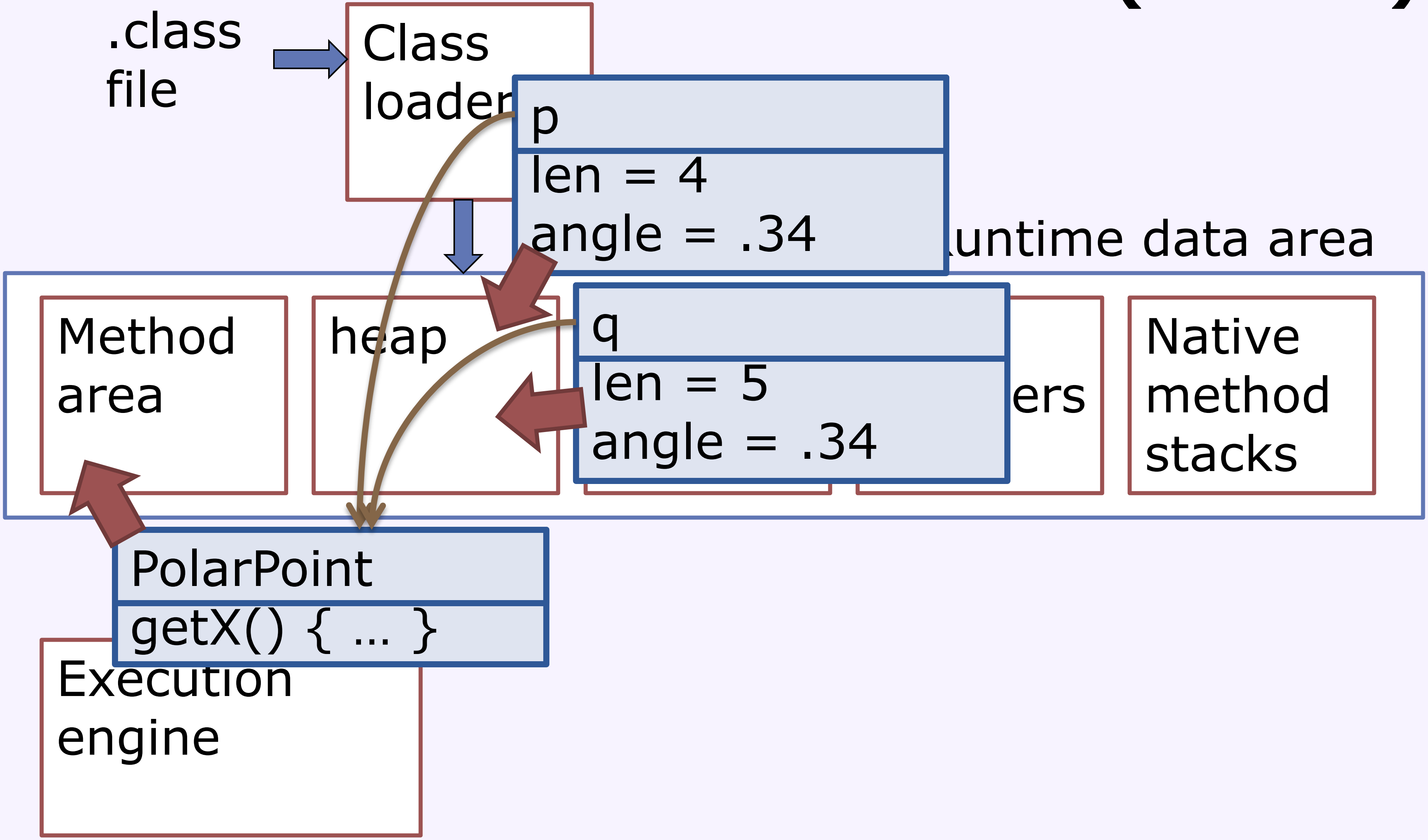
Java stacks

pc registers

Native method stacks

Execution engine

The Java Virtual Machine (sketch)



.class file

Class loader

p

len = 4

angle = .34

untime data area

Method area

heap

q

len = 5

angle = .34

ers

Native method stacks

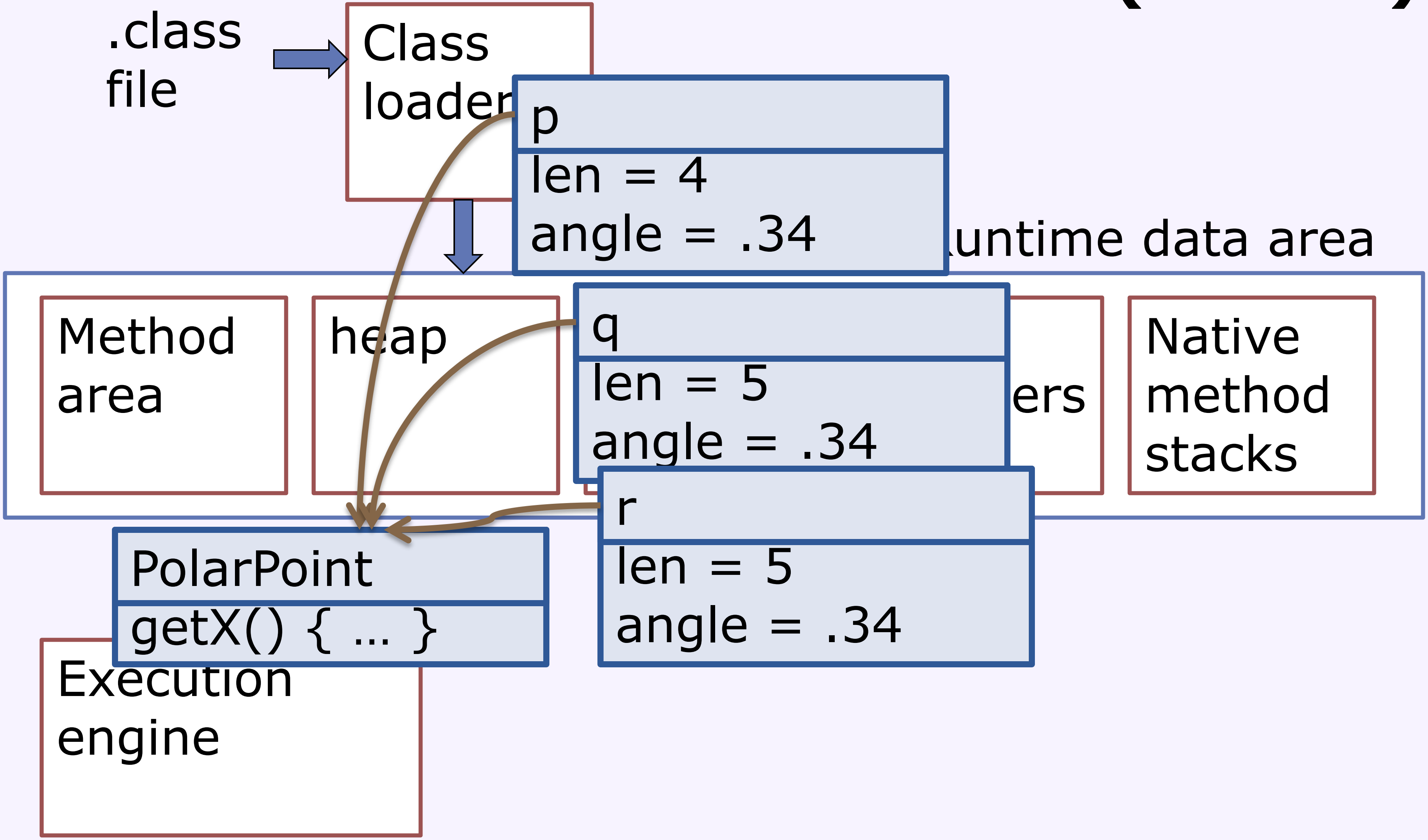
PolarPoint

getX() { … }

Execution engine

Object Identity & Object Equality

The Java Virtual Machine (sketch)



.class file

Class loader

p
len = 4
angle = .34

untime data area

Method area

heap

q
len = 5
angle = .34

ers

Native method stacks

r
len = 5
angle = .34

PolarPoint
getX() { … }

Execution engine

Object identity vs. equality

- There are two notions of equality in OO
  - 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.

Encore: Polymorphism Example 2

Functional Lists of Integers

- Some operations we **expect** to see:
  - **create** a new list
    - empty, or by adding an integer to an existing list
  - return the **size** of the list
  - **get** the *i*th integer in the list
  - **concatenate** two lists into a new list
- Key questions
  - How to **implement** the lists?
    - Many options
      - Arrays, linked lists, etc
    - How to hide the details of this choice from client code?
      - Why do this?
  - How to state **expectations**?
    - A variable **v** can reference a list of integers

Interfaces – stating **expectations**

- The IntList interface

```
public interface IntList {
    int size();
    int get(int n);
    IntList concatenate(IntList otherList);
    String toString();
}
```

- The declaration for **v** ensures that any object referenced by v will have implementations of the methods **size**, **get**, **concatenate**, and **toString**

```
Intlist v = …
```

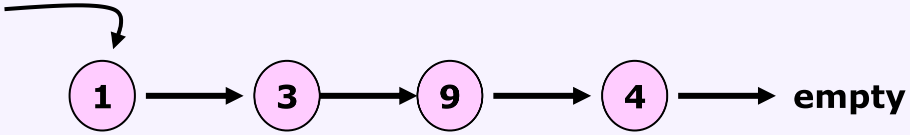
```
int len = v.size();
int third = v.get(2);
System.out.println (v.toString());
```

Implementing lists

- Two options (among many):
  - Arrays

1	3	7	5	11	13	6	42

  - Linked lists



• Operations:	Array	List
▪ **create** a new **empty** list	const	const
▪ return the **size** of the list	const	linear
▪ return the *ith* integer in the list	?	?
▪ **create** a list by adding to the **front**	?	?
▪ concatenate two lists into a new list	?	?

An inductive definition

- *The size of a list L is*
  - *0* *if L is the empty list*
  - *1 + size of the tail of L* *otherwise*

Implementing Size

```
public class EmptyIntList implements IntList {
      public int size() {
            return 0; }
   . . .
}
```

Base case

```
public class IntListCell implements IntList {
      public int size() {
            return 1 + next.size(); }
   . . .
}
```

Inductive case

List Representation (BROKEN!)

```
public class EmptyIntList implements IntList {
   public int size() {
      return 0;
   }
   . . .
}
```

Base case

```
public class IntListCell implements IntList {
   private int value;
   private IntListCell next;

   public int size() {
      return 1 + next.size();
   }
   . . .
}
```

Type is wrong! May be a cell or an empty list!

Inductive case

List Representation (FIXED!)

```
public class EmptyIntList implements IntList {
   public int size() {
      return 0;
   }
   . . .
}
```

Base case

```
public class IntListCell implements IntList {
   private int value;
   private IntList next;

   public int size() {
      return 1 + next.size();
   }
   . . .
}
```

Interface type provides needed flexibility.

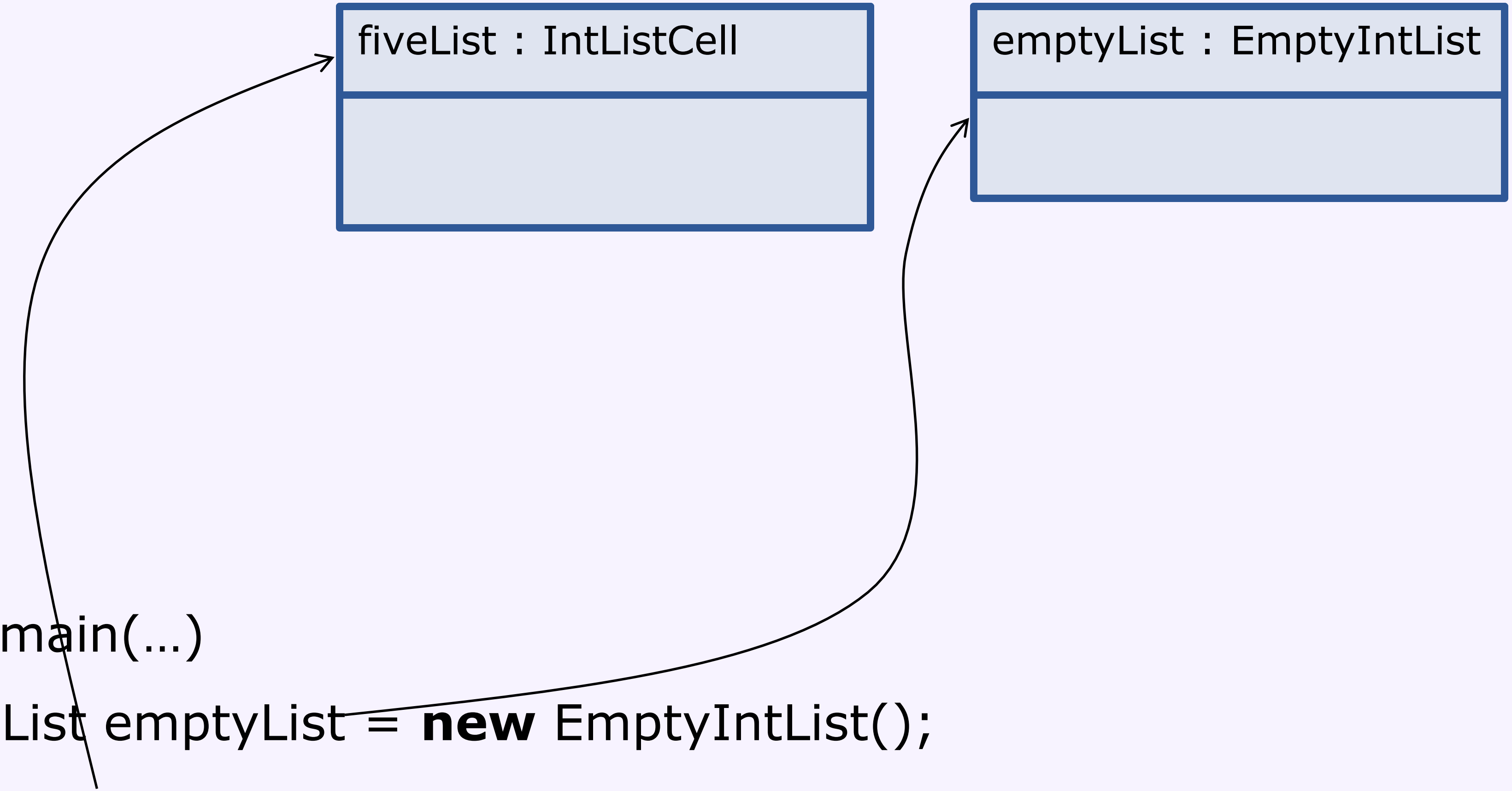
Inductive case

List Constructors

```
public class EmptyIntList implements IntList {
   public EmptyIntList() {
      // nothing to initialize
   }
   . . .
}

public class IntListCell implements IntList {
   public IntListCell(int val, IntList next) {
      this.value = val;
      this.next = next;
   }

   private int value;
   private IntList next;
   . . .
}
```

Java gives us this **default constructor** for free if we don't define any constructors.

Some Client Code

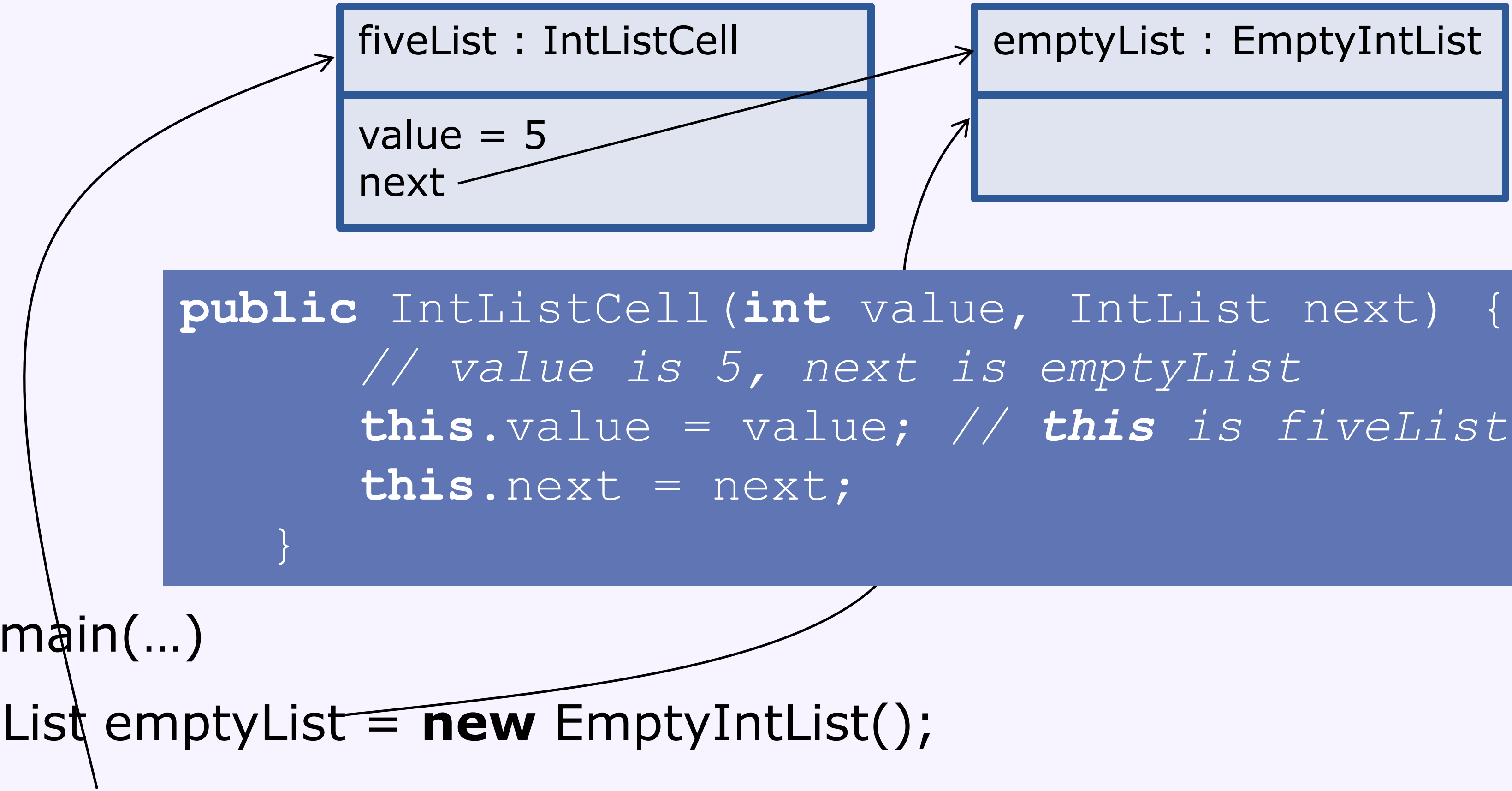

fiveList : IntListCell

emptyList : EmptyIntList

In main(…)

```
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
```

Some Client Code



fiveList : IntListCell

value = 5
next

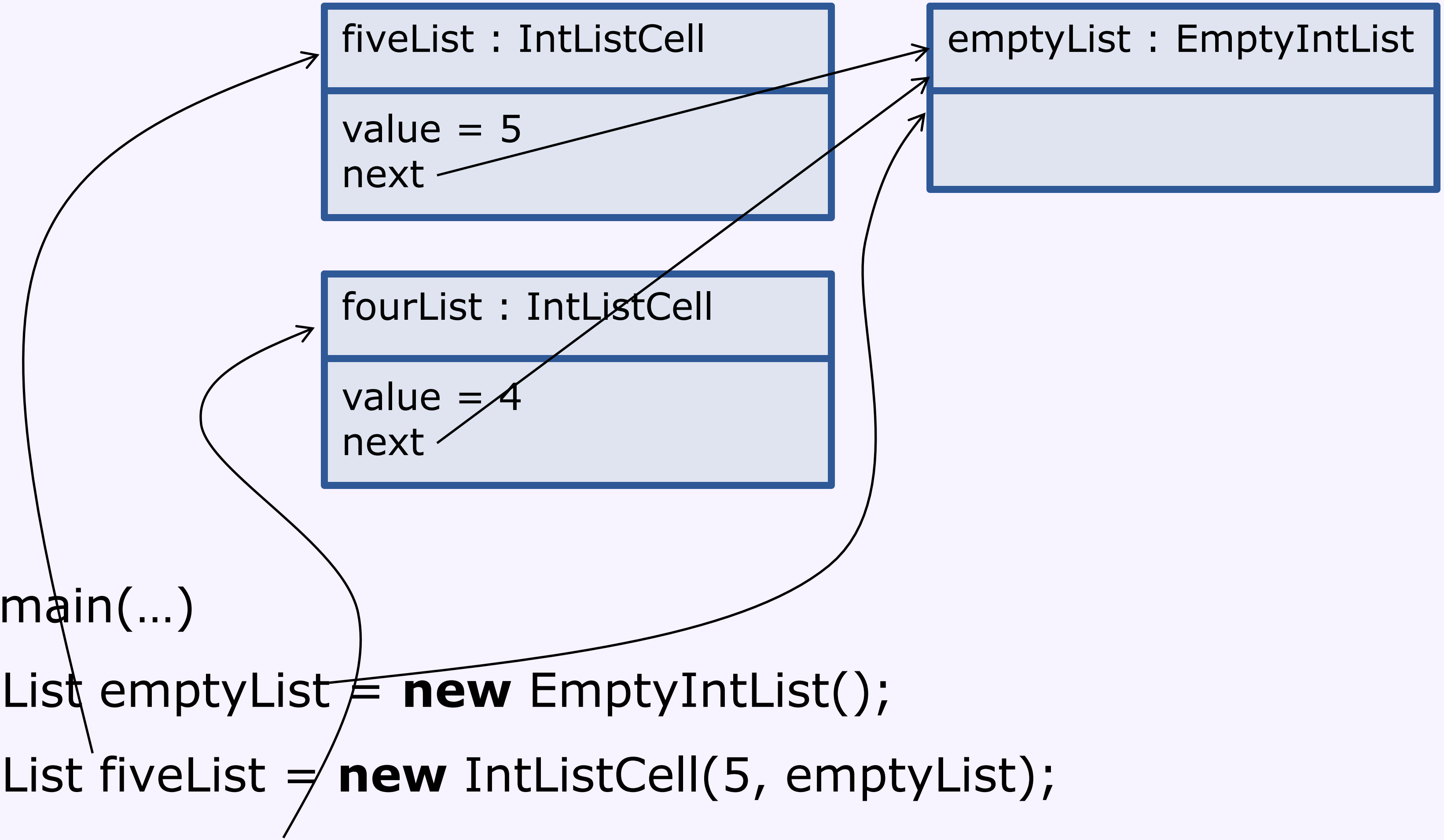
emptyList : EmptyIntList

```
public IntListCell(int value, IntList next) {
        // value is 5, next is emptyList
        this.value = value; // this is fiveList
        this.next = next;
    }
```

In main(…)

```
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
```

Some Client Code



In main(…)

```
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
```

Implementing Concatenate

```
public class EmptyIntList implements IntList {
      public IntList concatenate(IntList other) {
            return other; }
   . . .
}
```

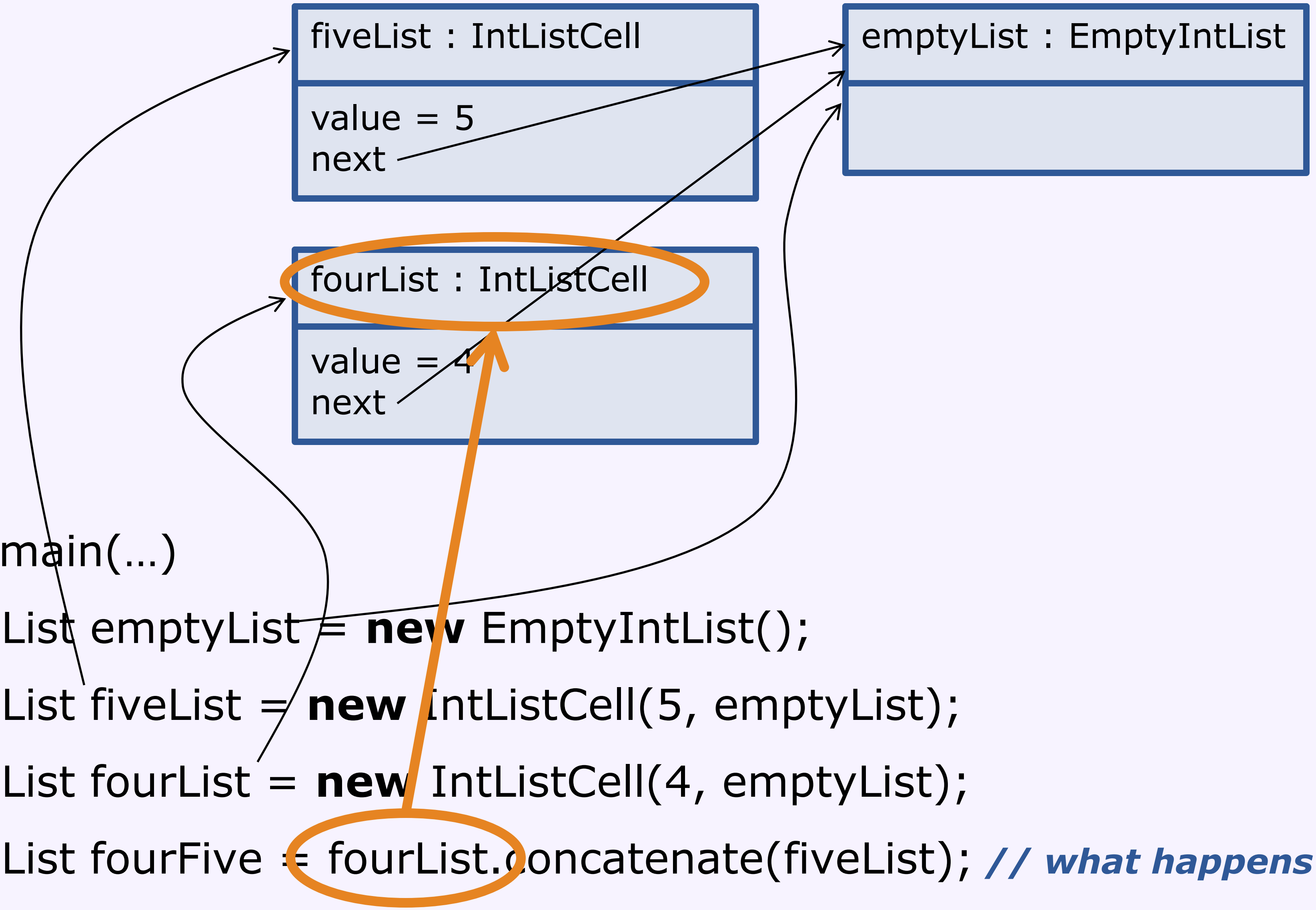
Base case

```
public class IntListCell implements IntList {
      public IntList concatenate(IntList other) {
            IntList newNext = next.concatenate(other);
            return new IntListCell(value, newNext); }
   . . .
}
```

Inductive case

Two concatenate methods – which do we use?

Some Client Code



```
In main(…)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
```

Method dispatch (simplified)

Example:

IntList fourList = **new** IntListCell(4, emptyList);

IntList fourFive = fourList.concatenate(fiveList);

- Step 1 (compile time): determine what type to look in
  - Look at the static type (IntList) of the receiver (fourList)
- Step 2 (compile time): find the method in the class with the right name
  - Find the method in the class with the right name
    - Later: there may be more than one such method

    IntList concatenate(IntList otherList);

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

Method dispatch (simplified)

Example:

List fourList = **new** IntListCell(4, emptyList);

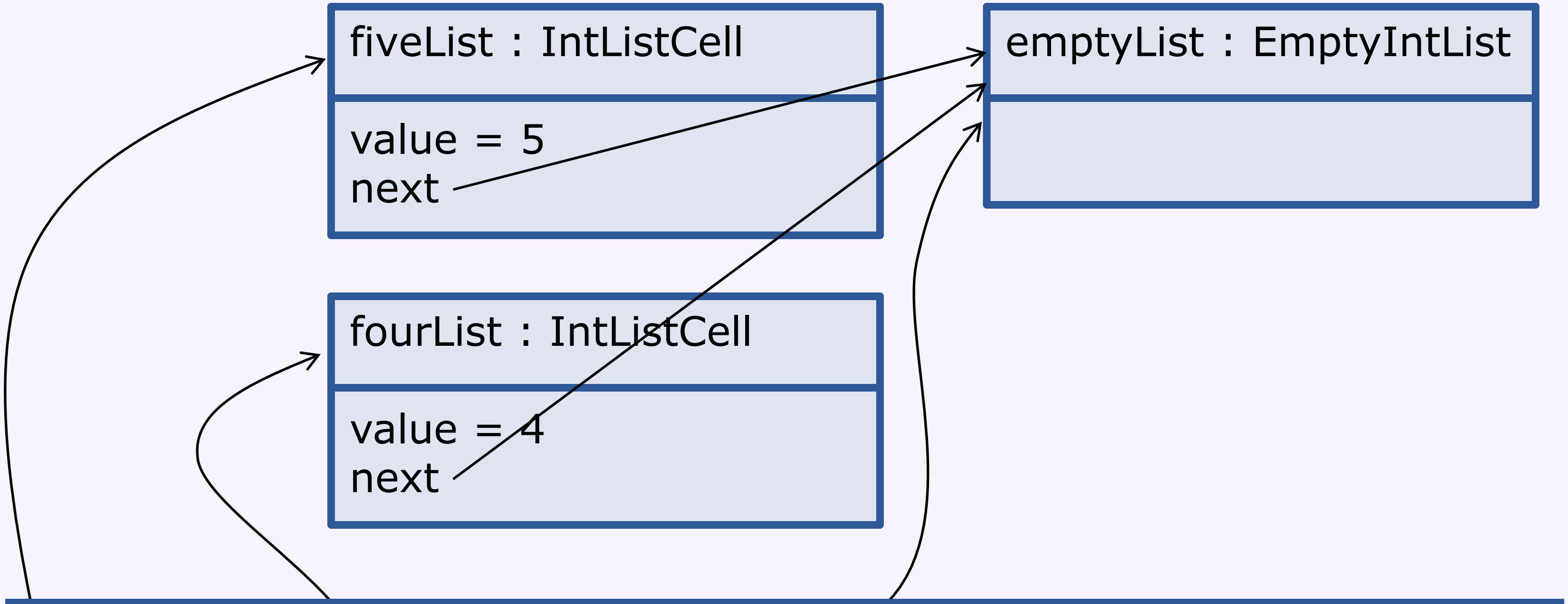
List fourFive = fourList.concatenate(fiveList);

- 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)

```
public IntList concatenate(IntList other) {
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext); }
```

- Invoke the method

Some Client Code



fiveList : IntListCell

value = 5
next

emptyList : EmptyIntList

fourList : IntListCell

value = 4
next

```
class IntListCell {
        public IntList concatenate(IntList other) {
                // this is fourList, other is fiveList
                IntList newNext = next.concatenate(other);
                return new IntListCell(value, newNext);
        }
```

I

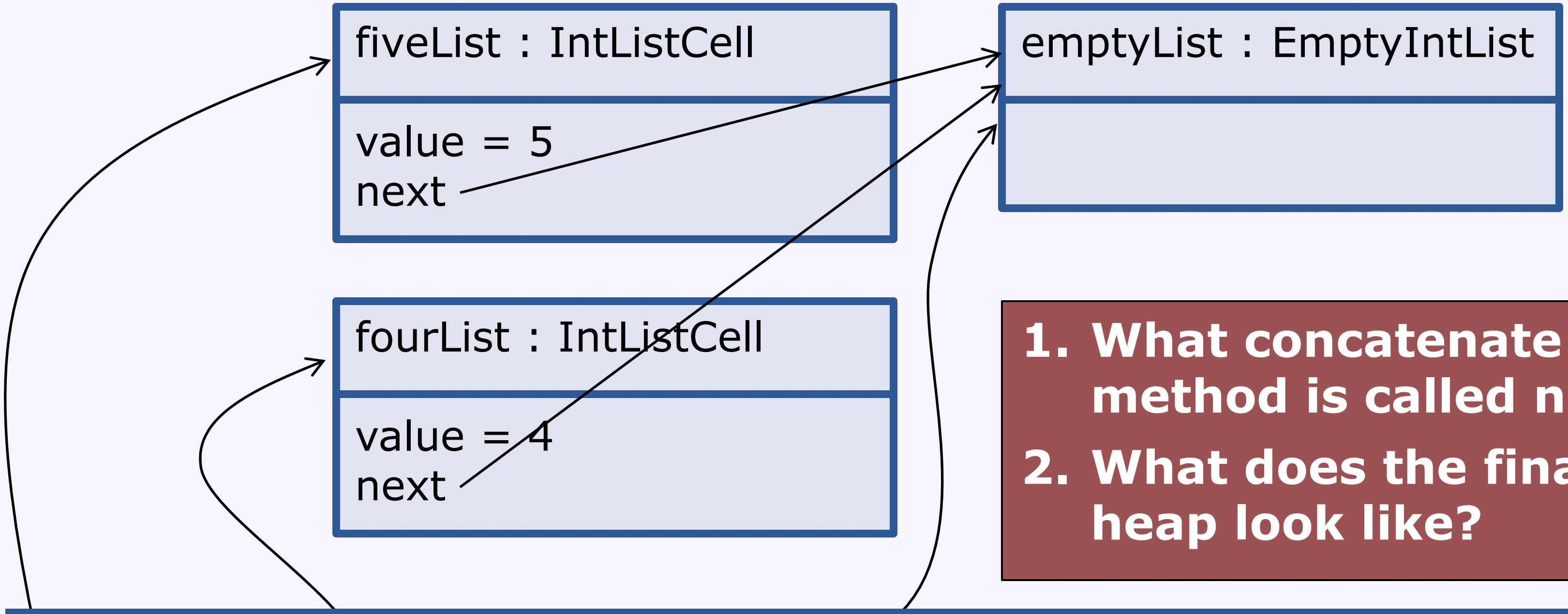
L

L

List fourList = **new** IntListCell(4, emptyList);

List fourFive = fourList.concatenate(fiveList); ***// what happens?***

A Question for You!


fiveList : IntListCell

value = 5
next

emptyList : EmptyIntList

fourList : IntListCell

value = 4
next

1. **What concatenate method is called next?**
2. **What does the final heap look like?**

I

L

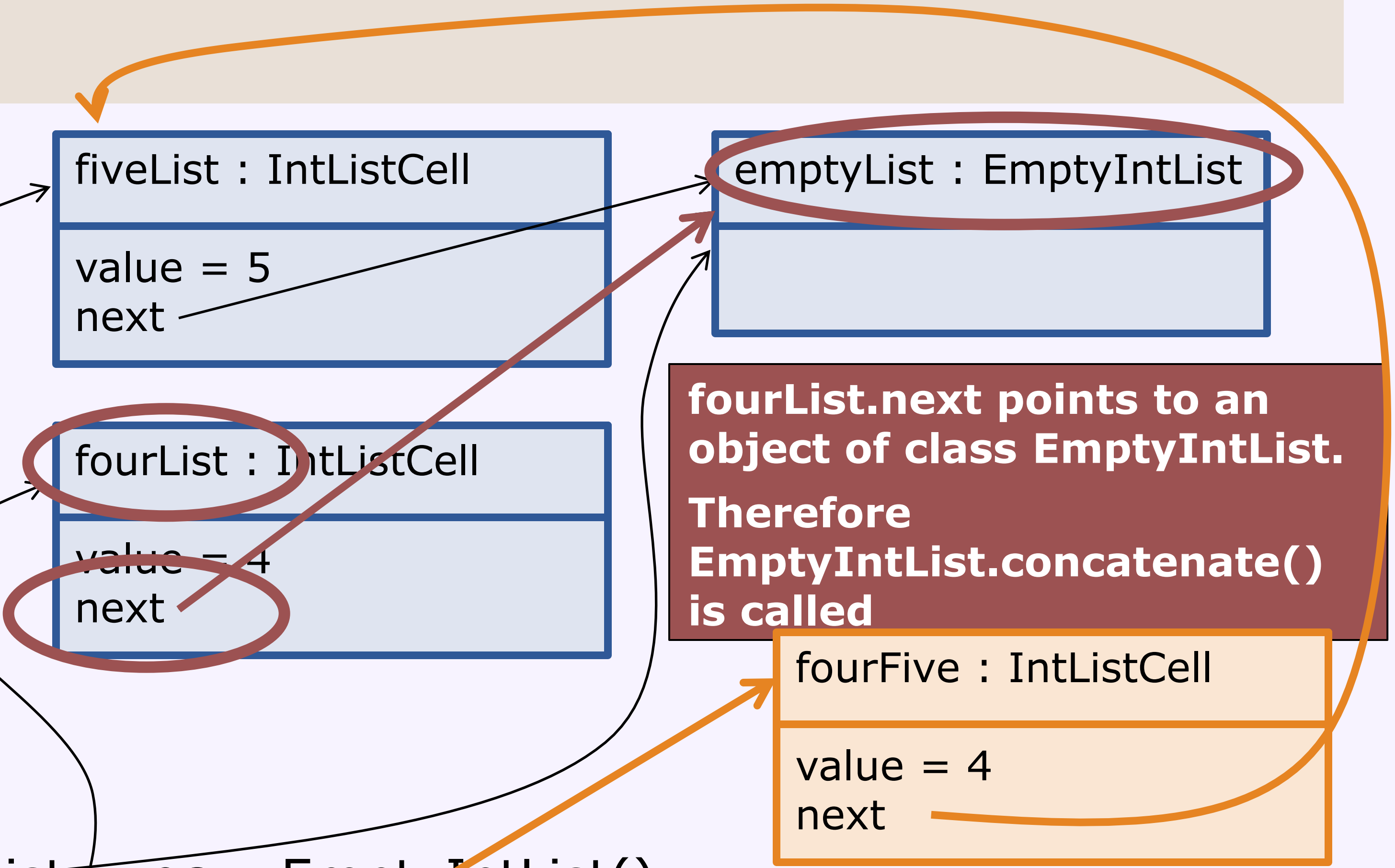
L

```
class IntListCell {
        public IntList concatenate(IntList other) {
                // this is fourList, other is fiveList
                IntList newNext = next.concatenate(other);
                return new IntListCell(value, newNext);
        }
```

List fourList = **new** IntListCell(4, emptyList);

List fourFive = fourList.concatenate(fiveList); ***// what happens?***

Answers



fiveList : IntListCell

value = 5
next

emptyList : EmptyIntList

fourList : IntListCell

value = 4
next

fourList.next points to an object of class EmptyIntList.

Therefore EmptyIntList.concatenate() is called

fourFive : IntListCell

value = 4
next

In main(…)

```
List emptyList = new EmptyIntList();
List fiveList = new IntListCell(5, emptyList);
List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?
```

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

- OOP – code is organized code around *kinds of things*
    - **Objects** correspond to things/concepts of interest
    - 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 family of similar objects
    - An **interface** states expectations for classes and their objects
    - Polymorphism and Encapsulation as key concepts
        - Allow different implementations behind a common interface
- Objects reside in the **heap**
    - They are accessed by **reference**, which gives the objects **identity**
    - **Dispatch** is used to choose a method implementation based on the **class** of the **receiver**
    - Equivalence (**equals**) does not mean the same object (==)