Programming and Algorithms II

Lecture 4: Inheritance and Polymorphism

Nicolas Wu

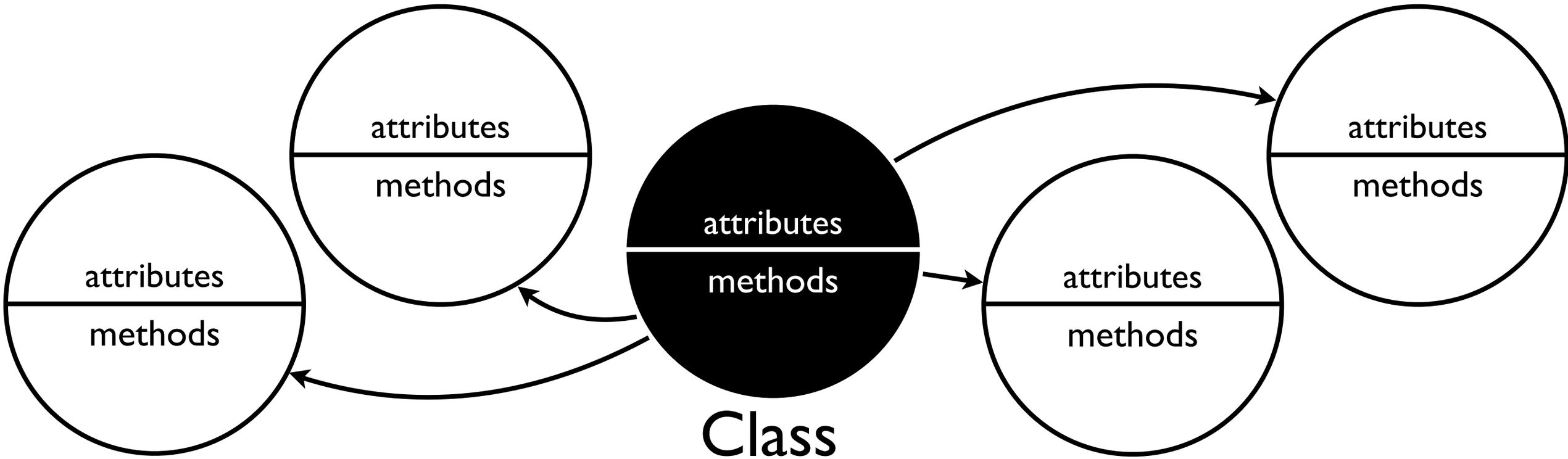
nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Object Scope

Object Scope

- Objects share nothing implicitly, other than their class and *static* features:
  - Objects store their own attributes
  - Objects execute their own copies of methods

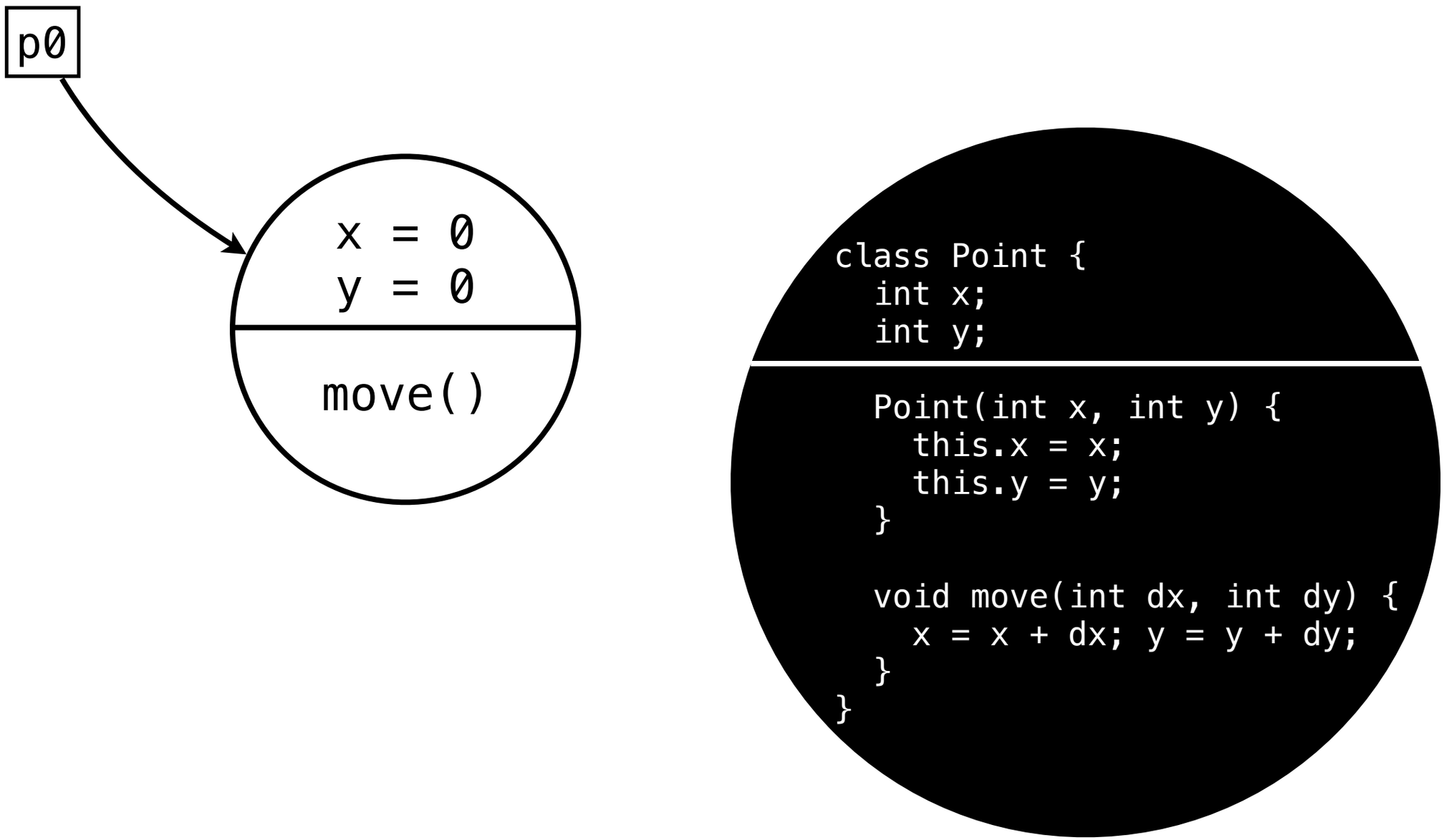


Object Scope

```
class Point {
  int x;
  int y;

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

  void move(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
}
```

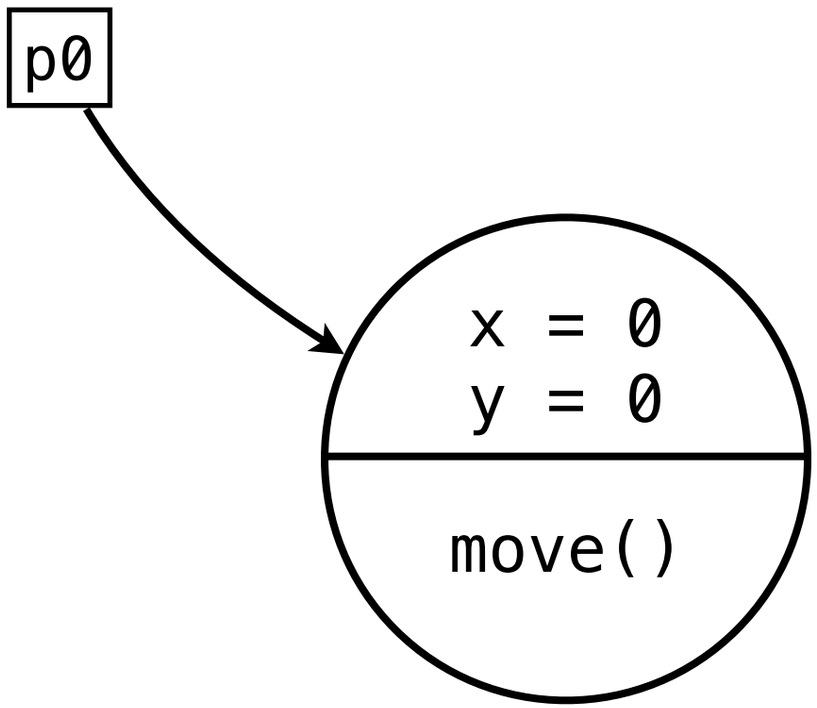
Object Scope

```
Point p0 = new Point(0,0);
```



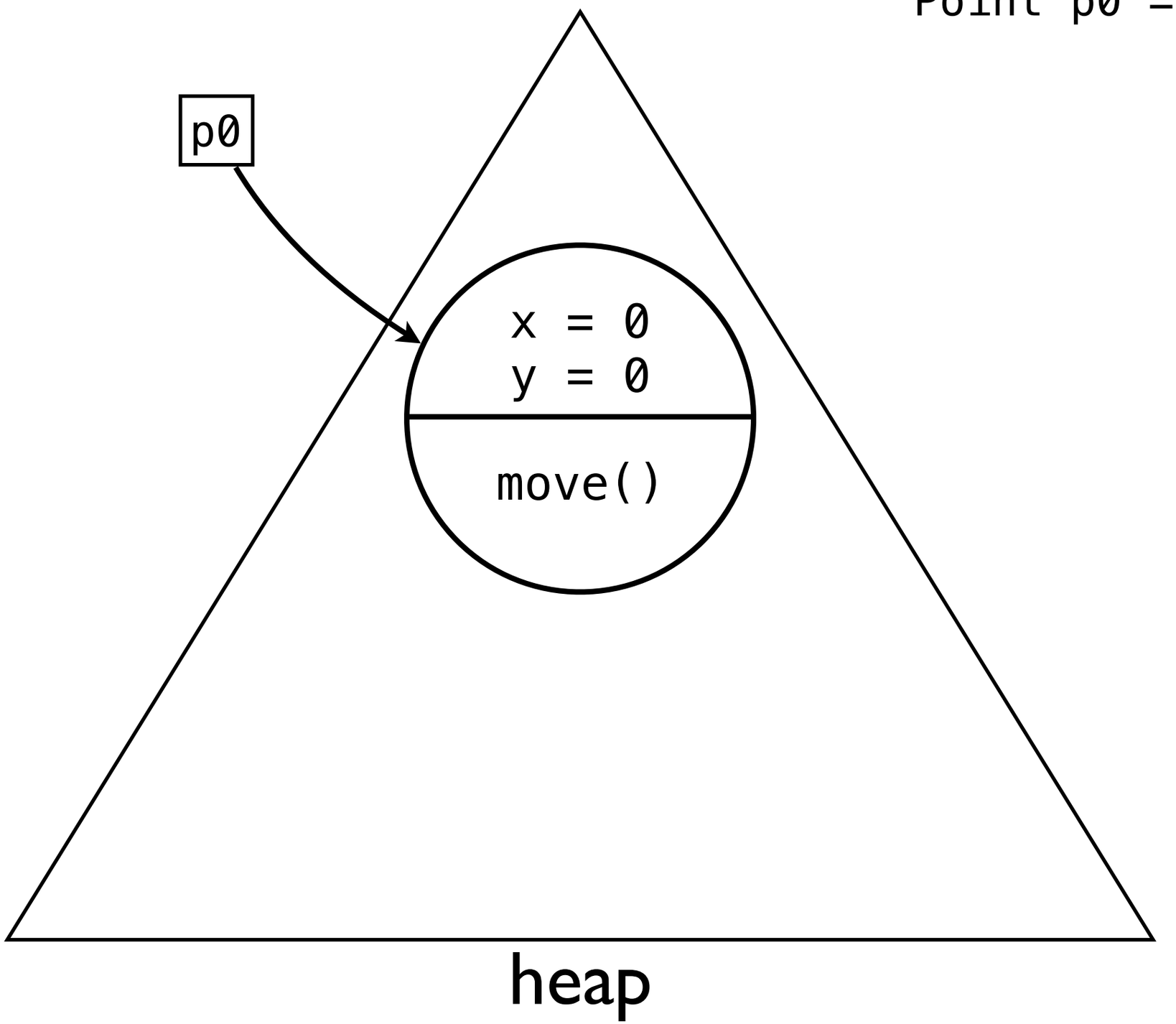
Object Scope

```
Point p0 = new Point(0,0);
```



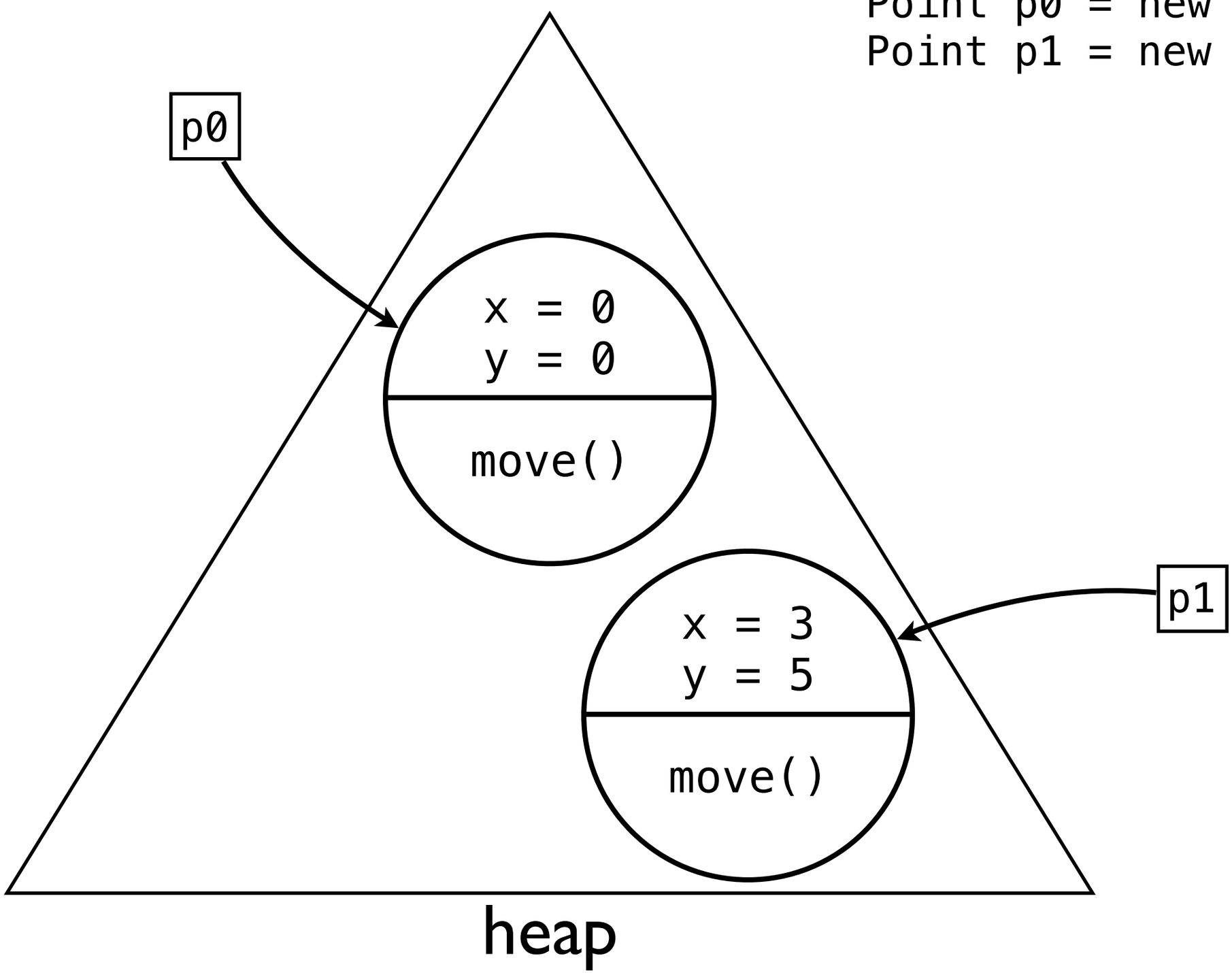
Object Scope

```
Point p0 = new Point(0,0);
```



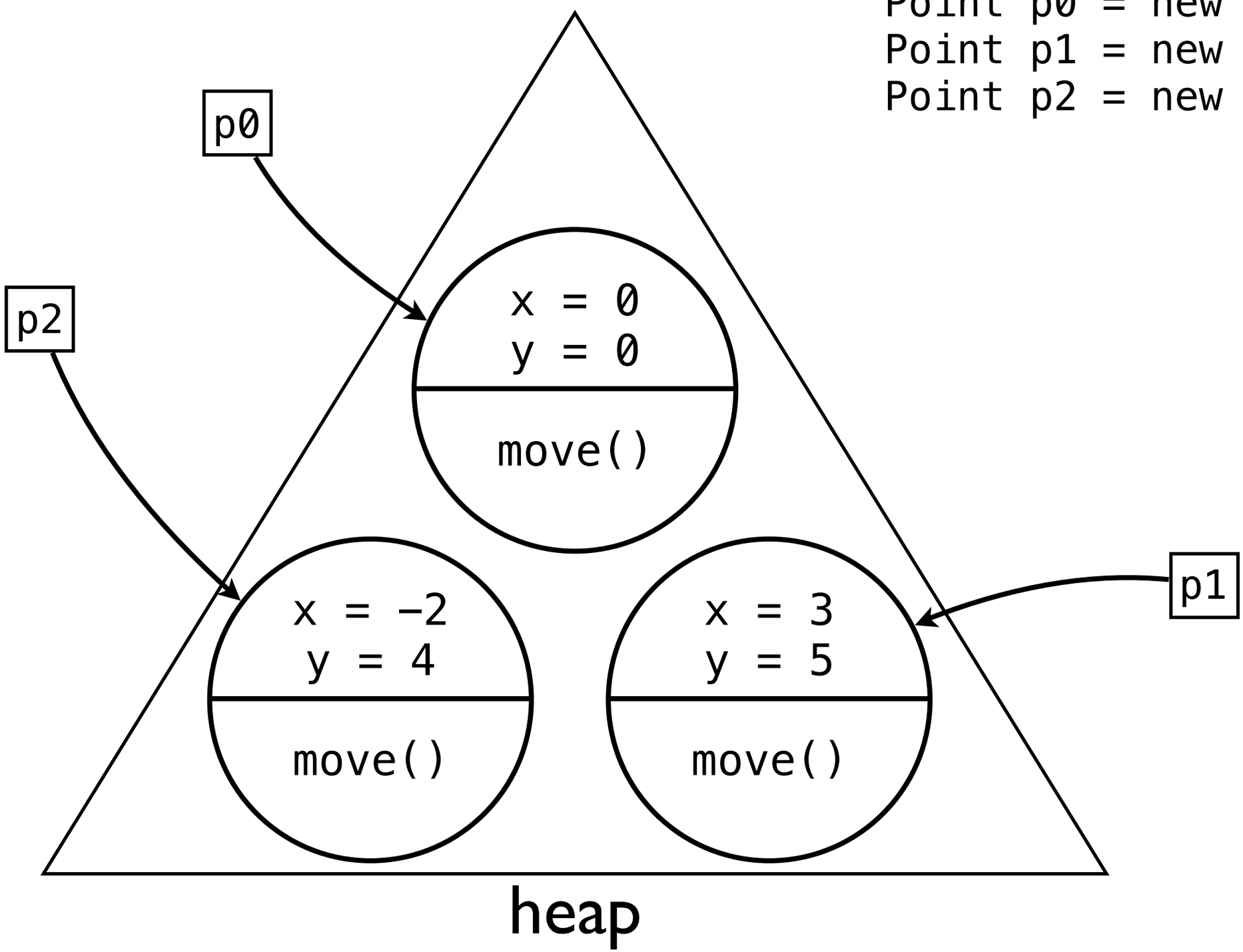
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
```



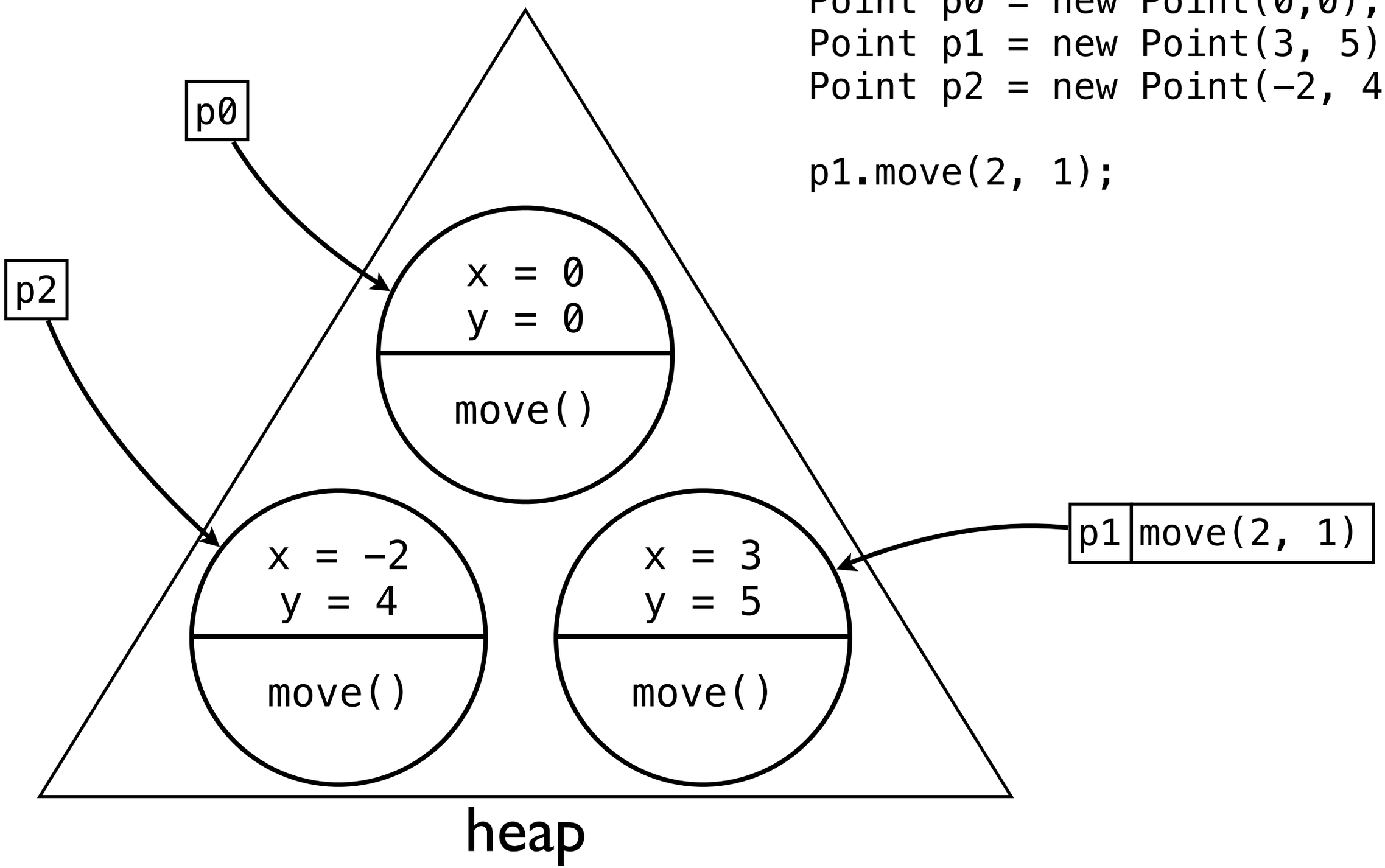
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);
```



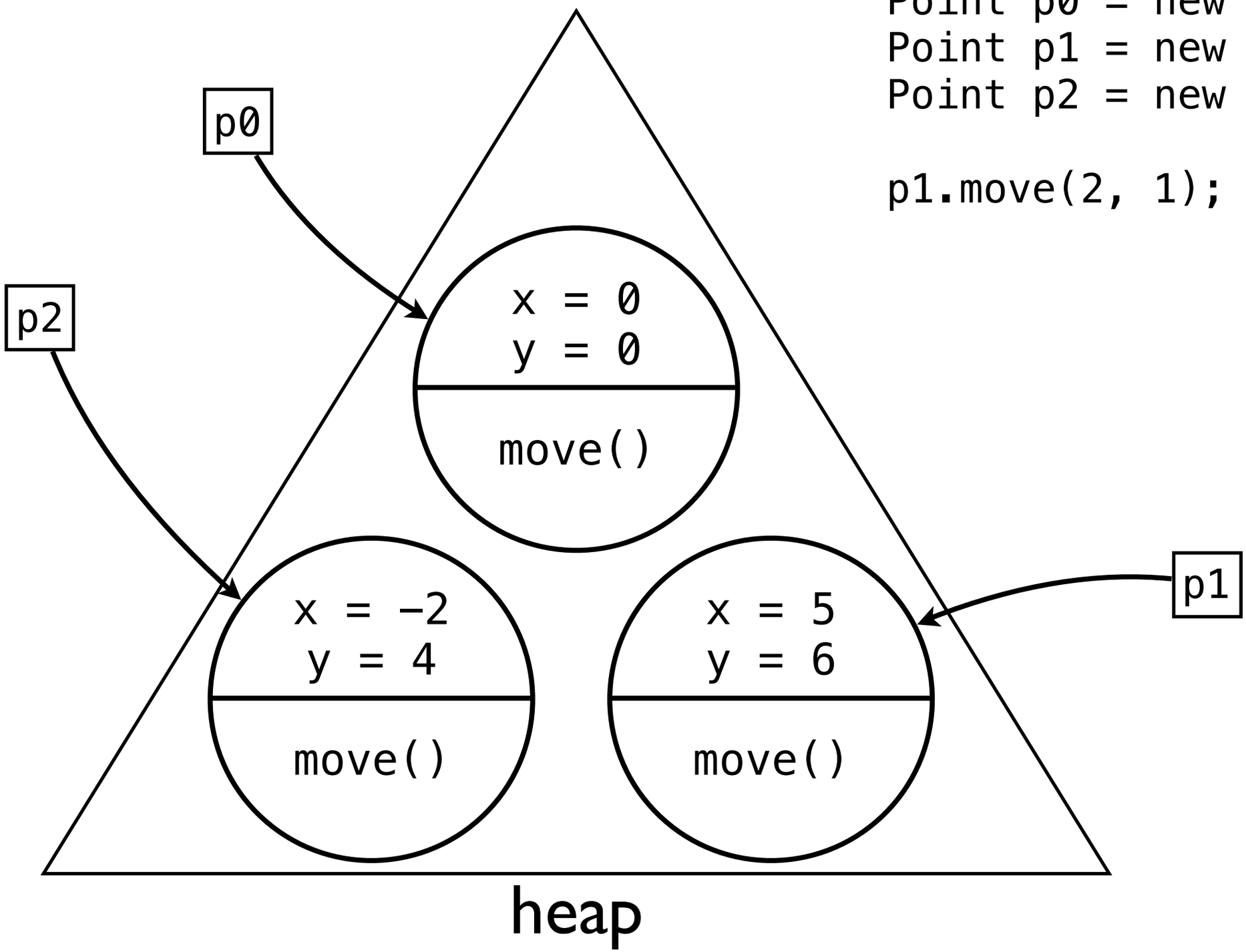
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
```



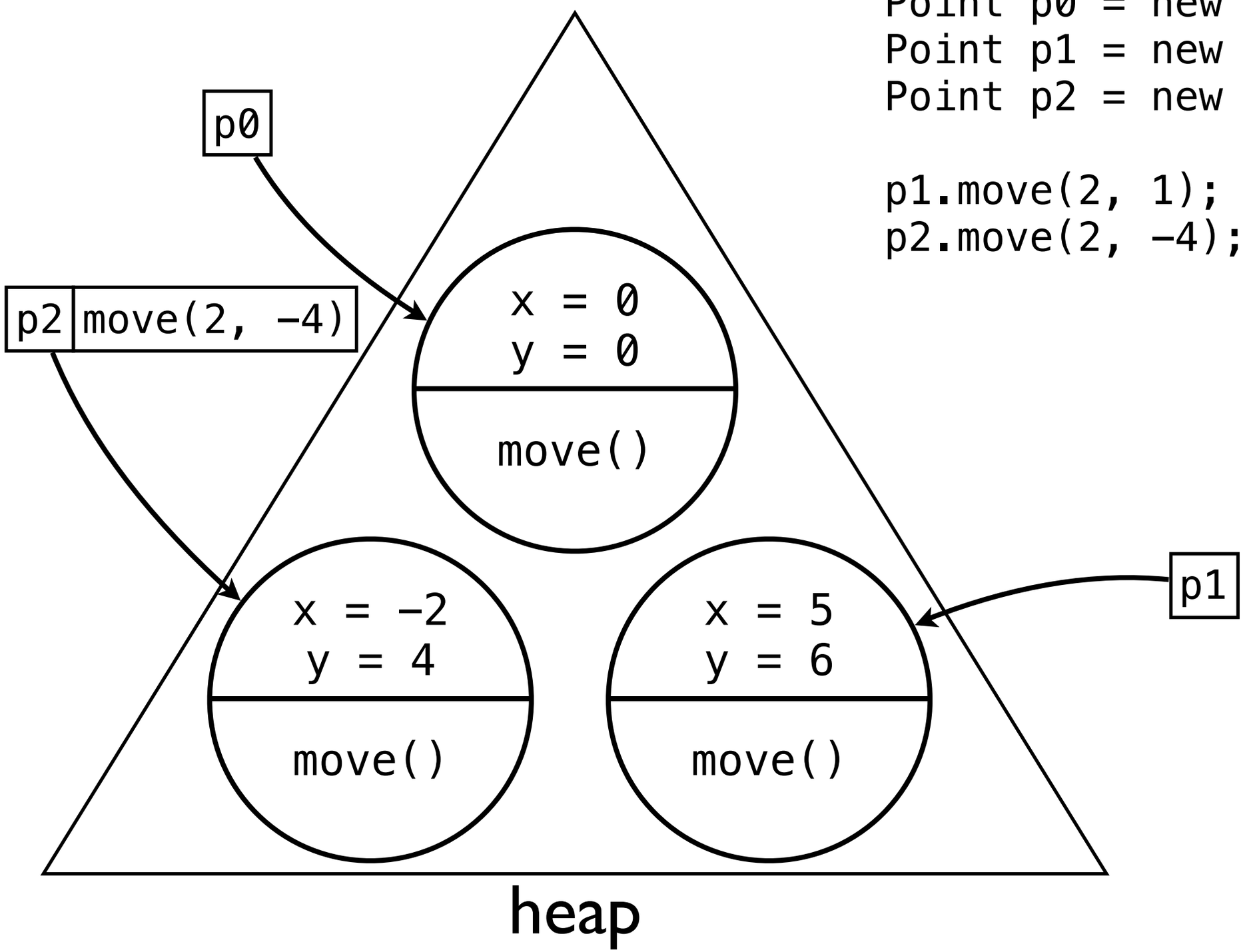
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
```



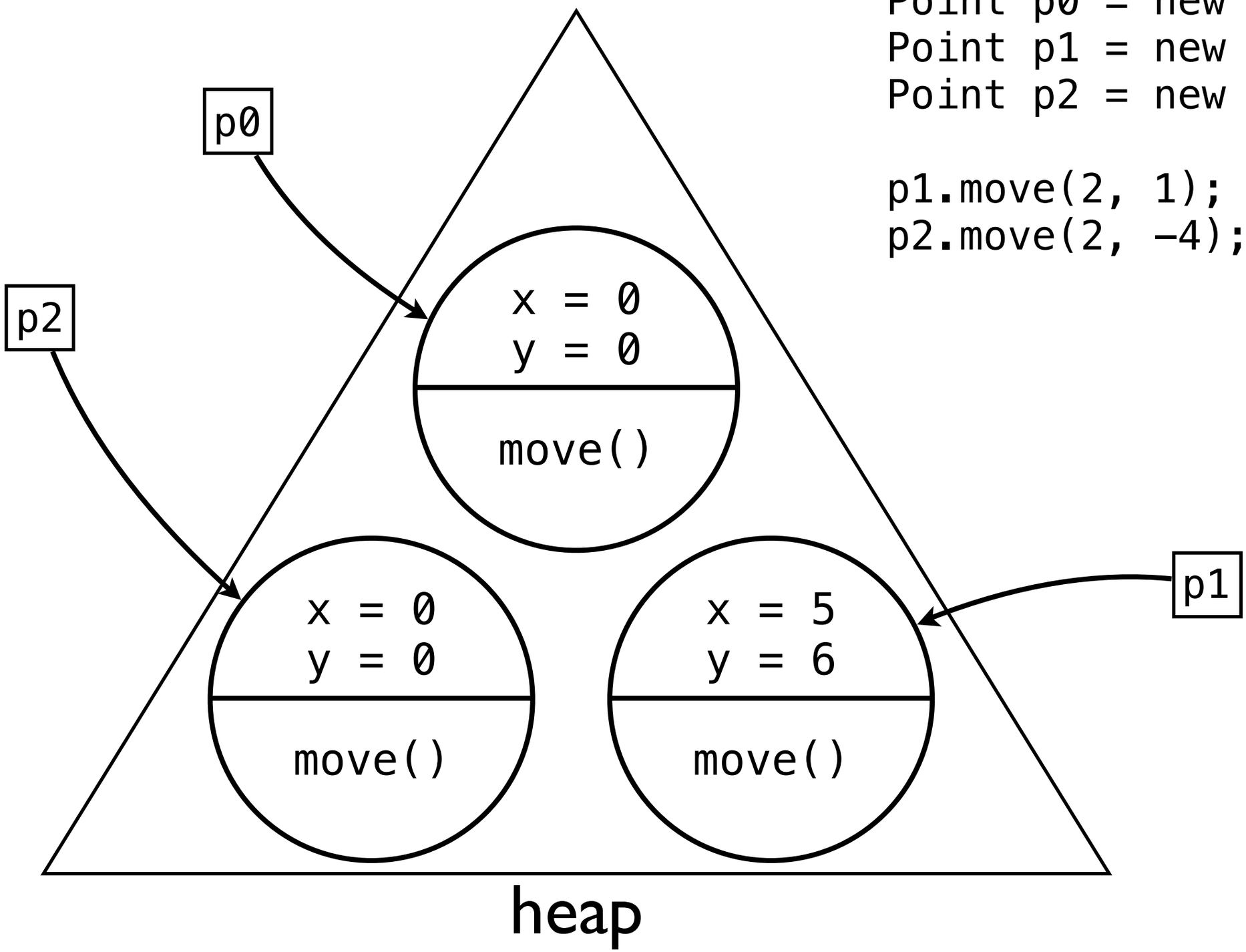
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
```



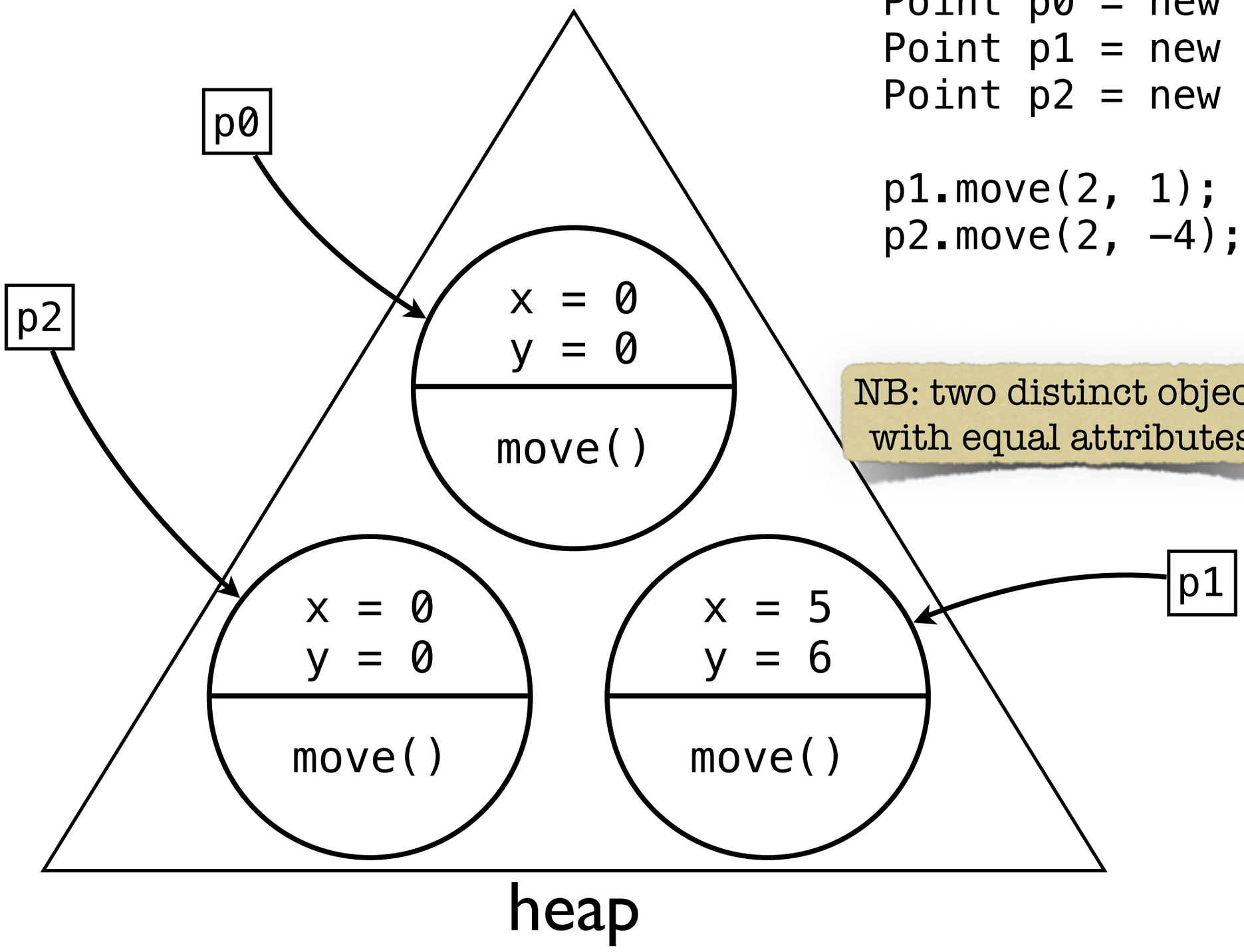
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
```



Object Scope

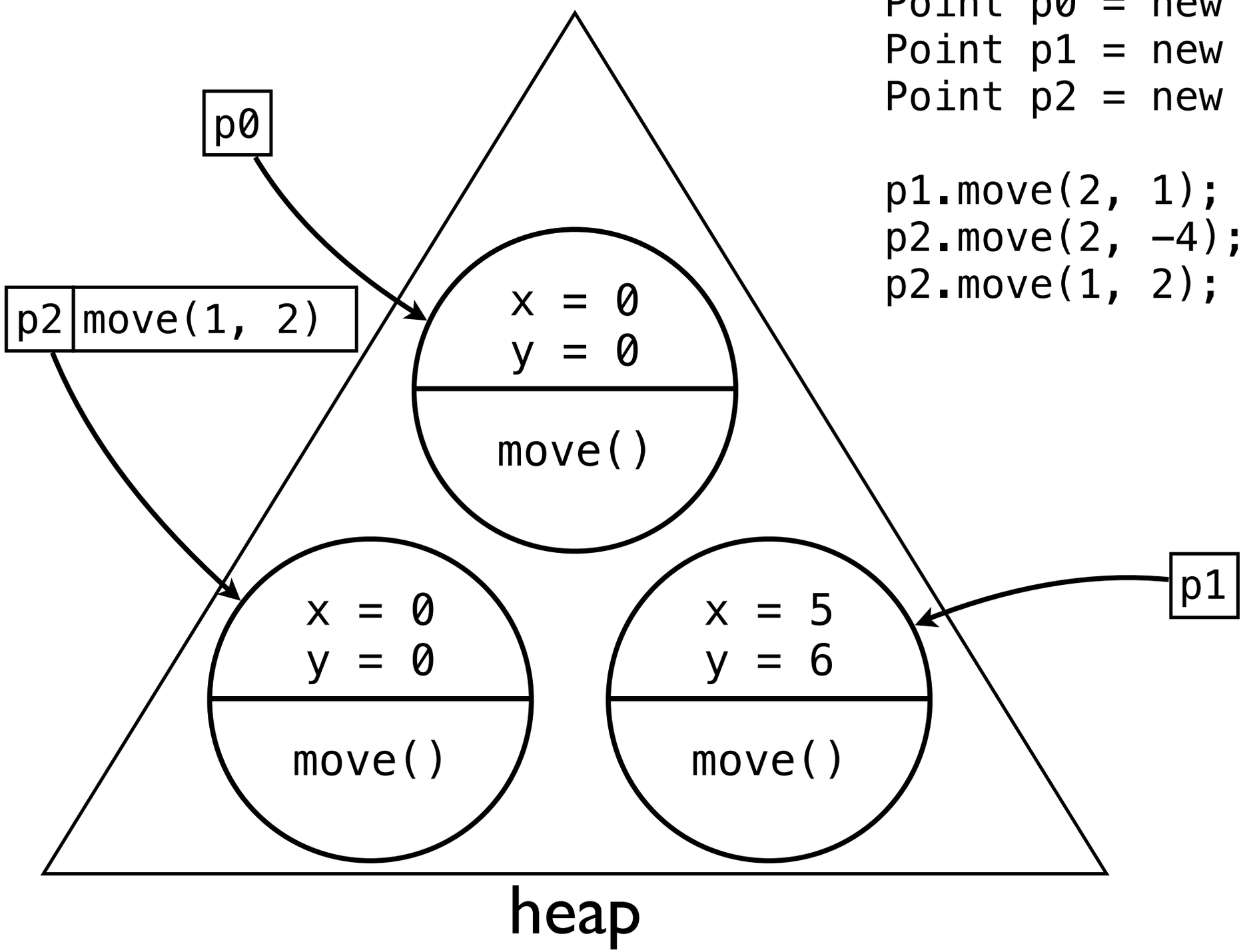
```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
```



NB: two distinct objects with equal attributes!

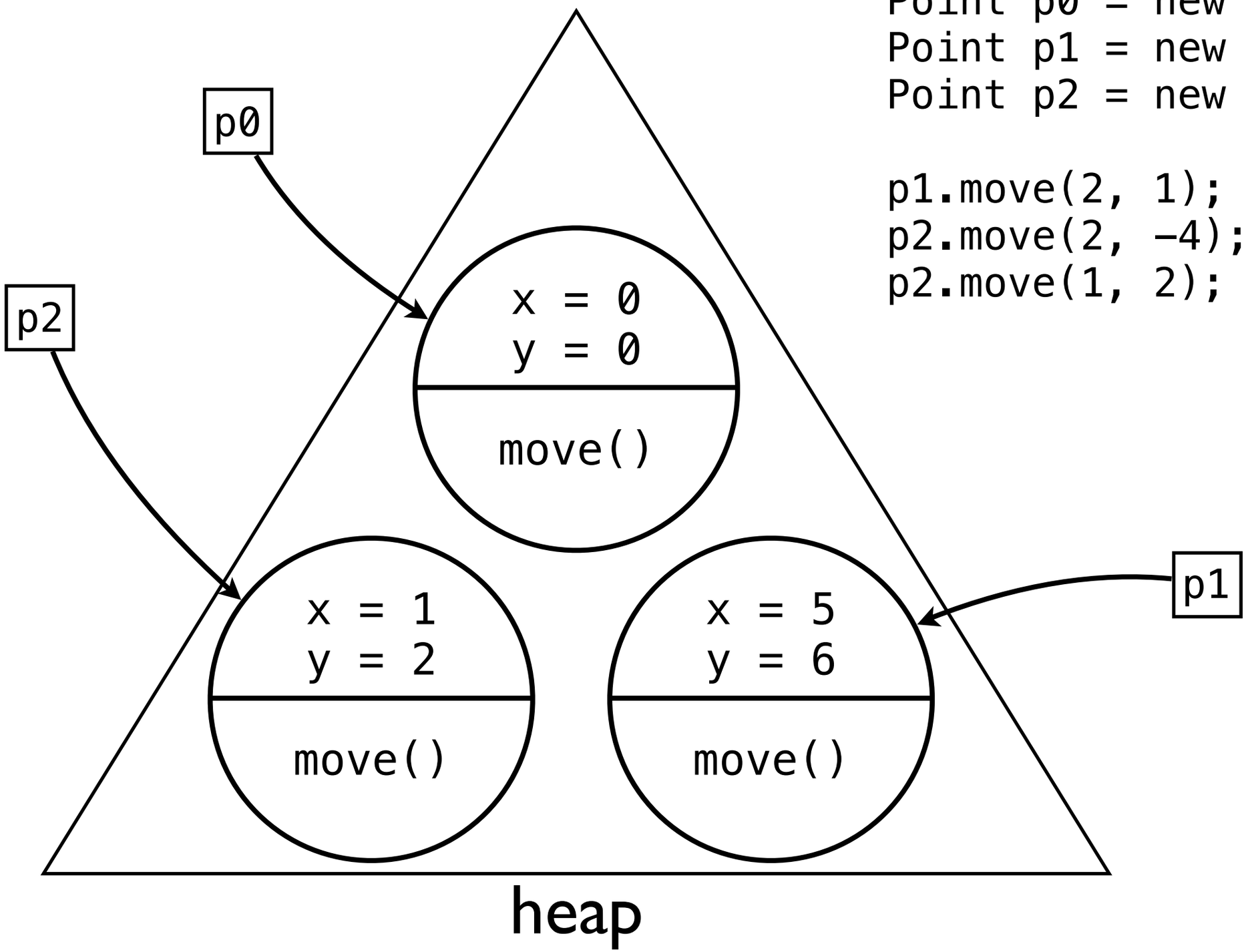
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
```



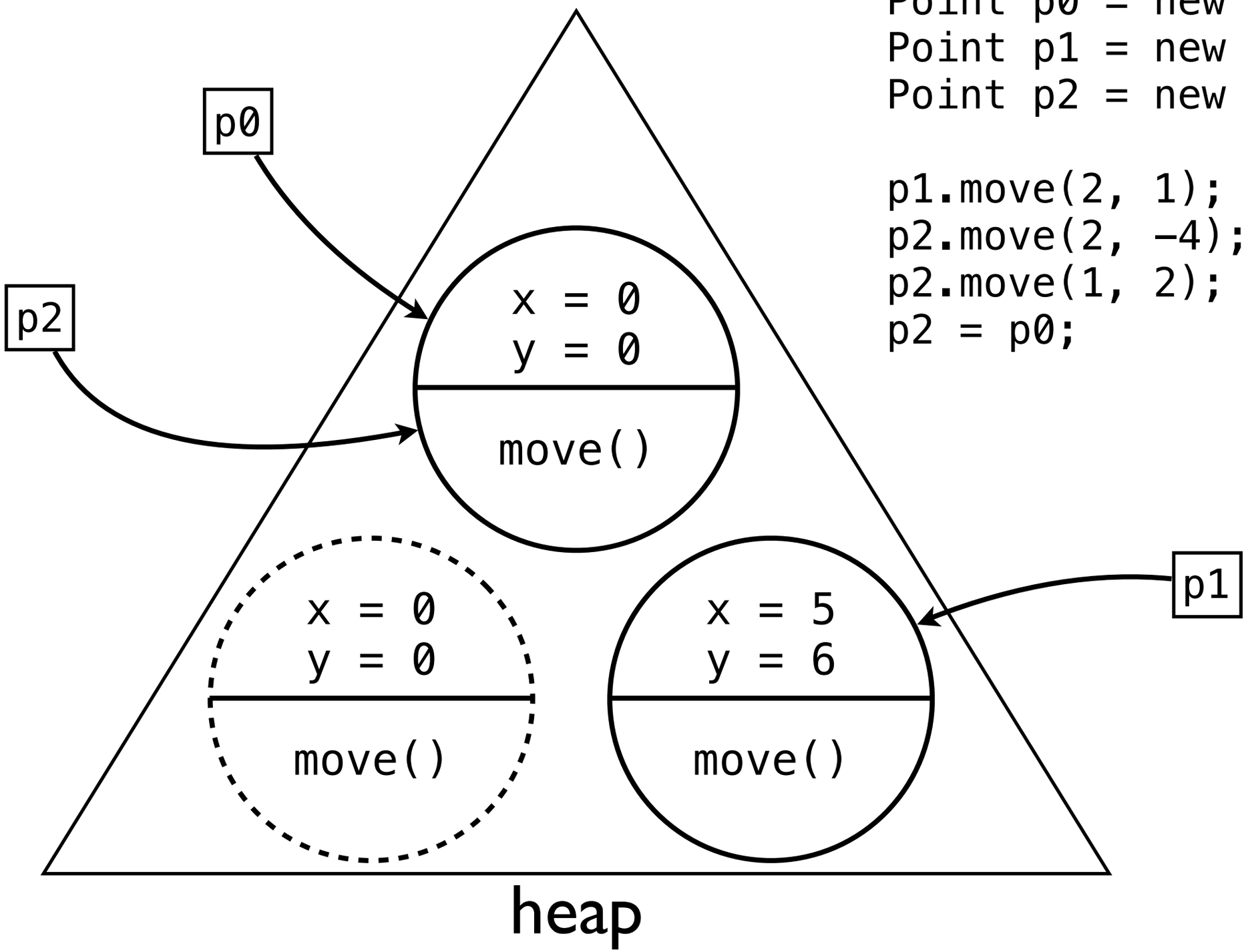
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
```



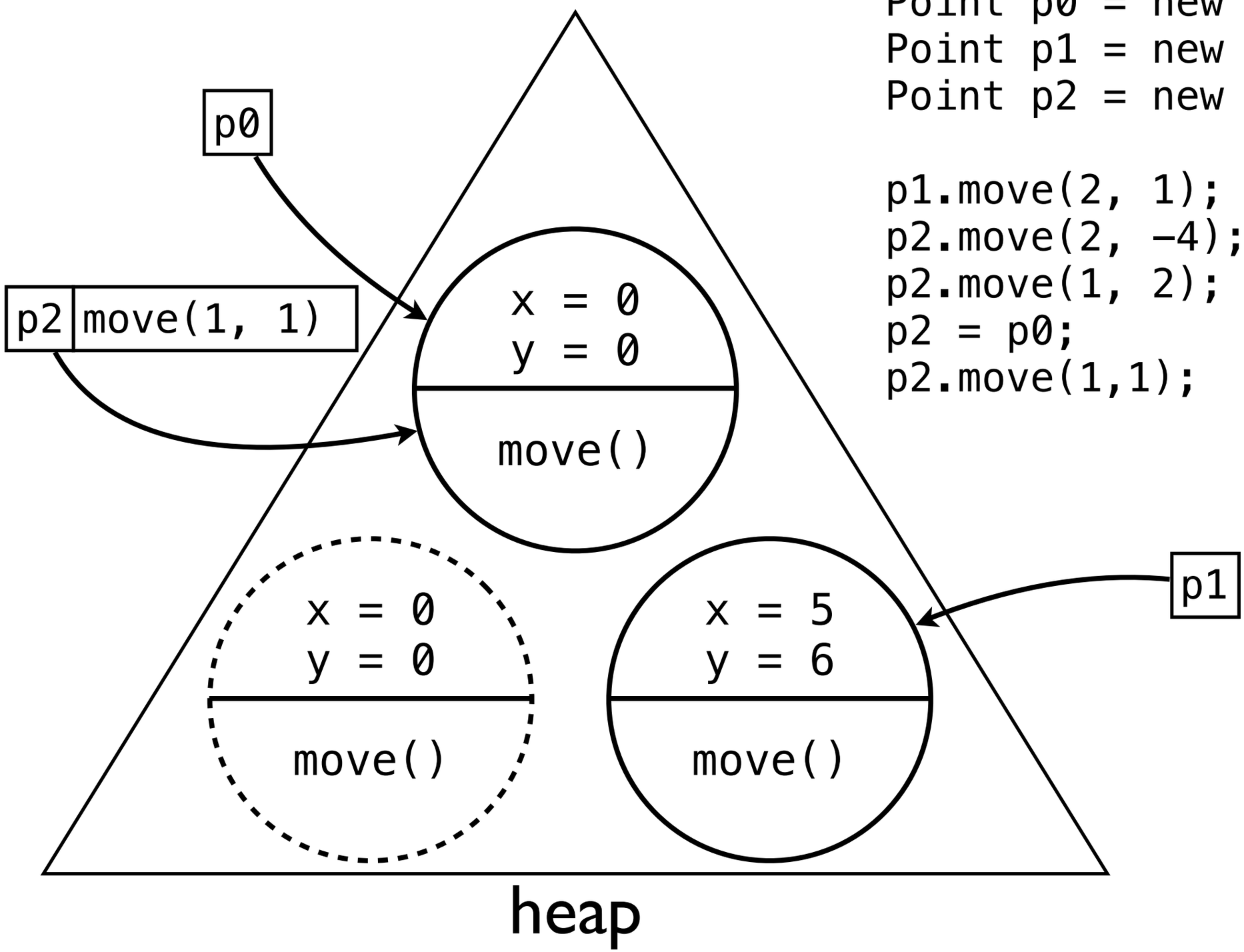
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
p2 = p0;
```



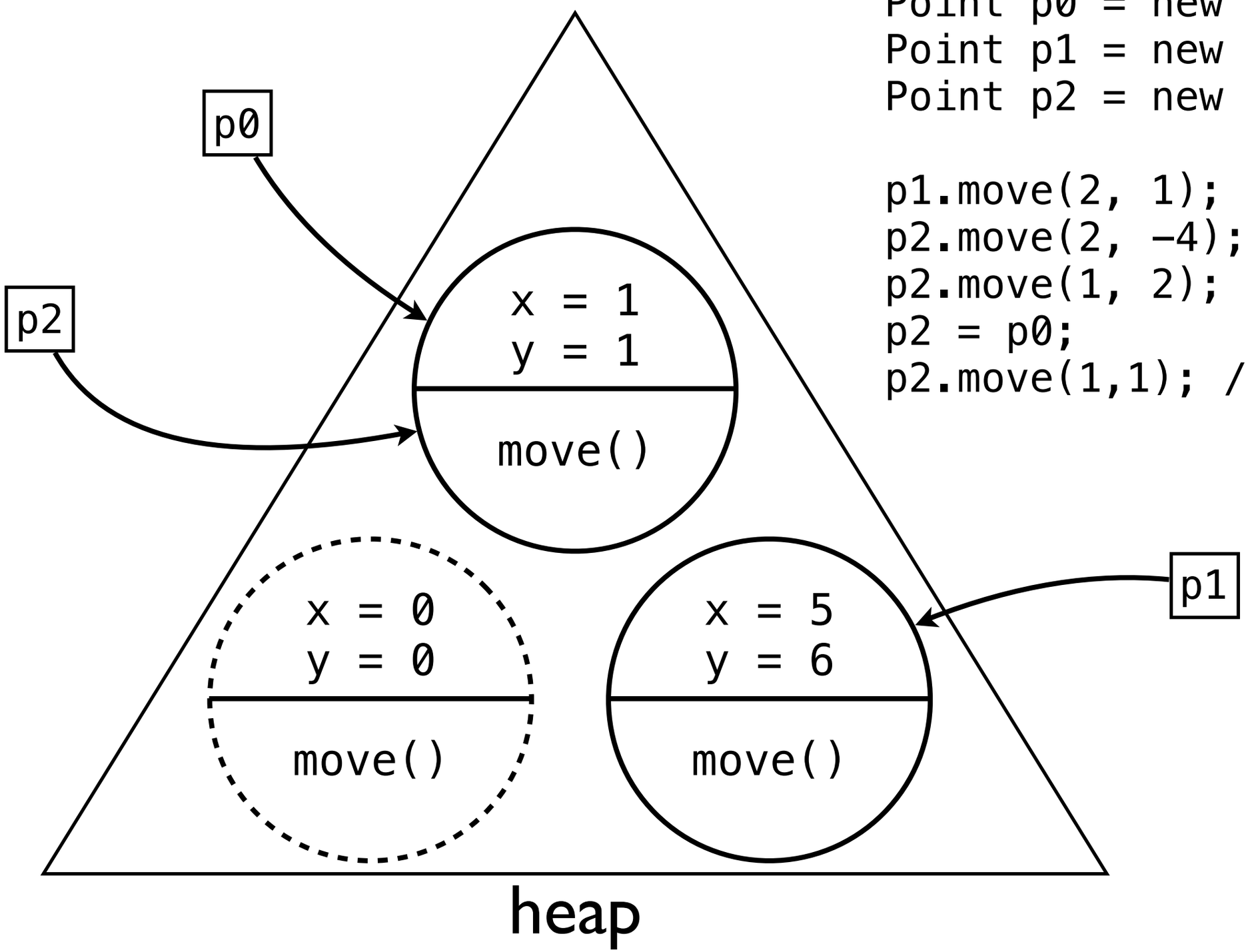
Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
p2 = p0;
p2.move(1,1);
```

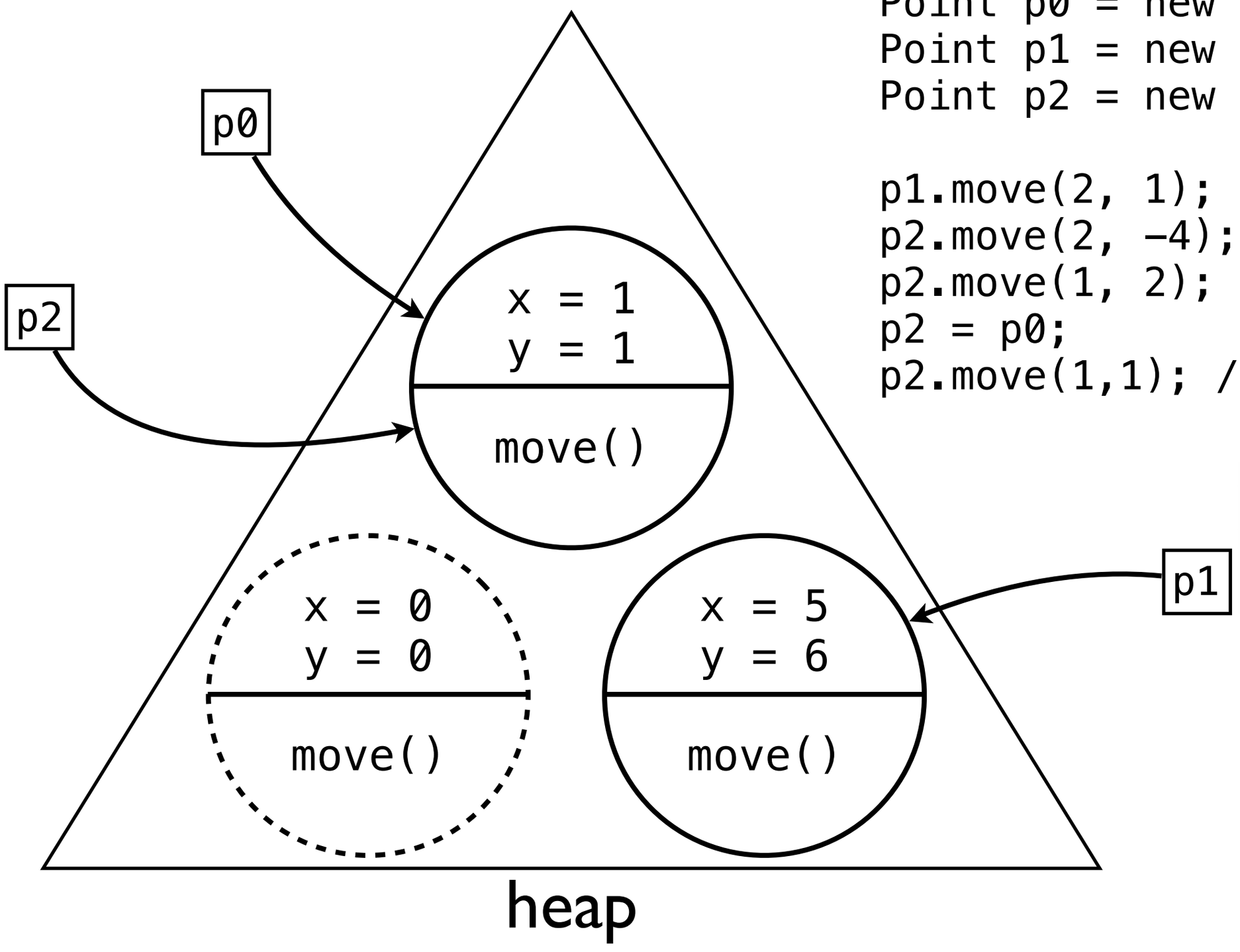


Object Scope

```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
p2 = p0;
p2.move(1,1); // “p0” changed!
```



Object Scope




```
Point p0 = new Point(0,0);
Point p1 = new Point(3, 5);
Point p2 = new Point(-2, 4);

p1.move(2, 1);
p2.move(2, -4);
p2.move(1, 2);
p2 = p0;
p2.move(1,1); // “p0” changed!
```

NB: p0 still points to the same object, but that object changed!

Reference Equality

Reference Equality

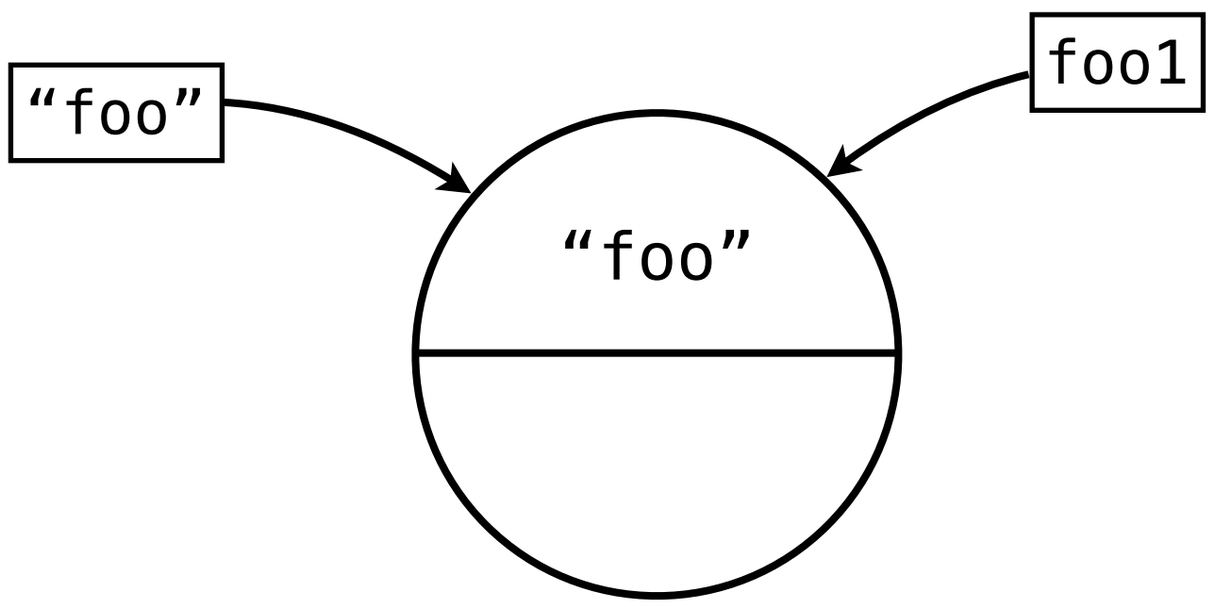
- On PODs, == checks for equality just as you might expect

```
6 == (4 + 2)
```

- With references, == means reference equality, not object equality
- To compare objects that are referenced, use equals()

Reference Equality

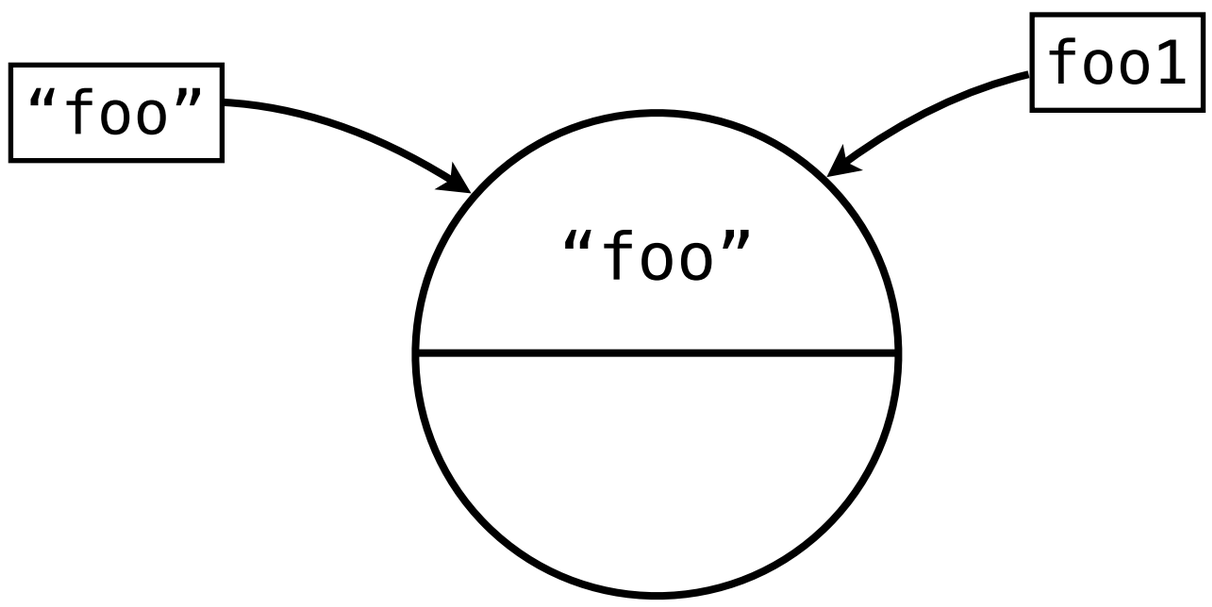
```
String foo1 = “foo”;
```



Reference Equality

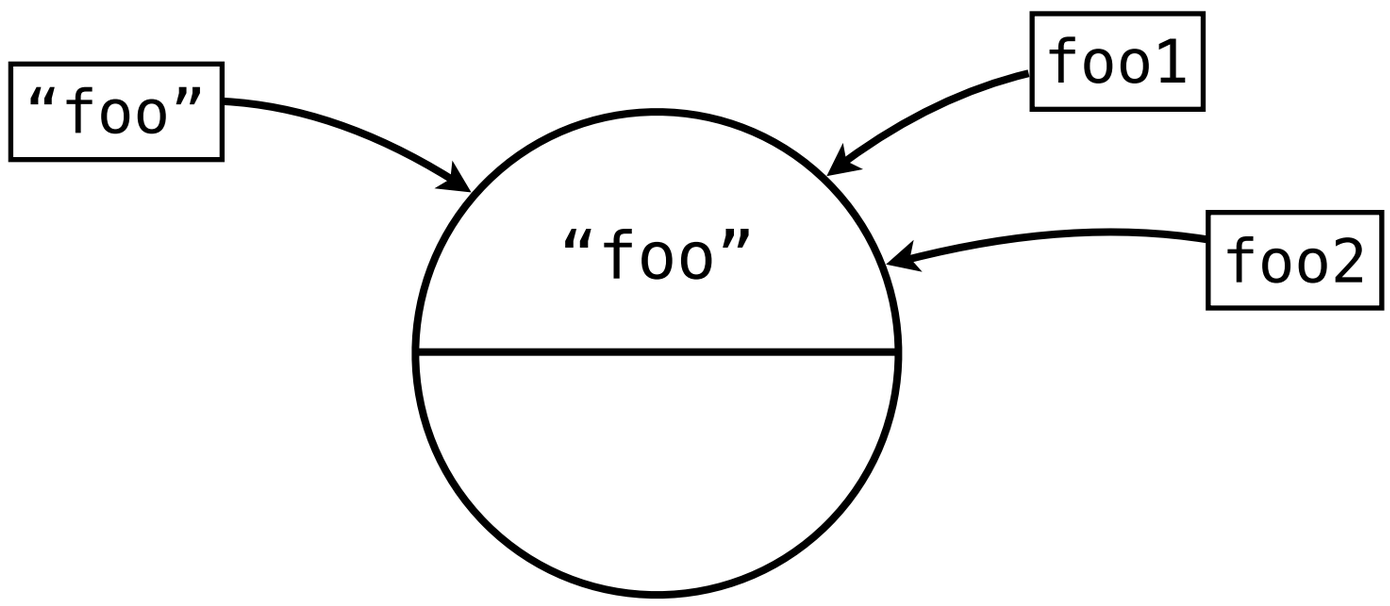
```
String foo1 = “foo”;
```

“foo” is a reference to a particular static String “foo”



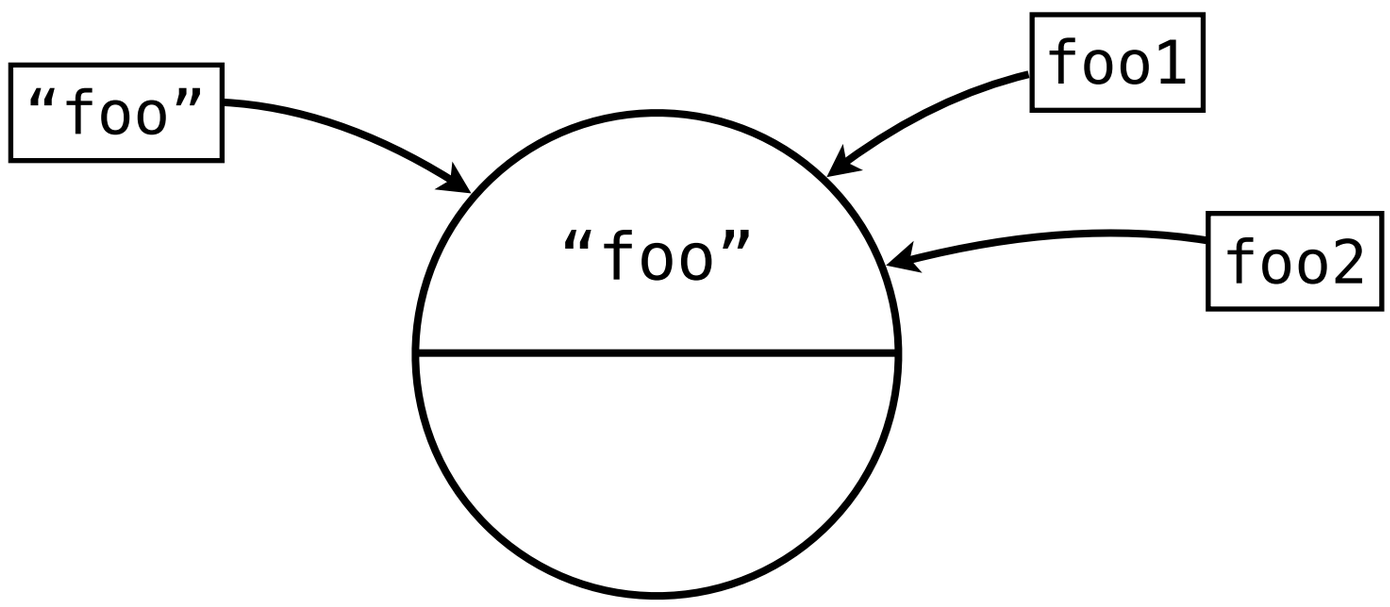
Reference Equality

```
String foo1 = “foo”;
String foo2 = “foo”;
```



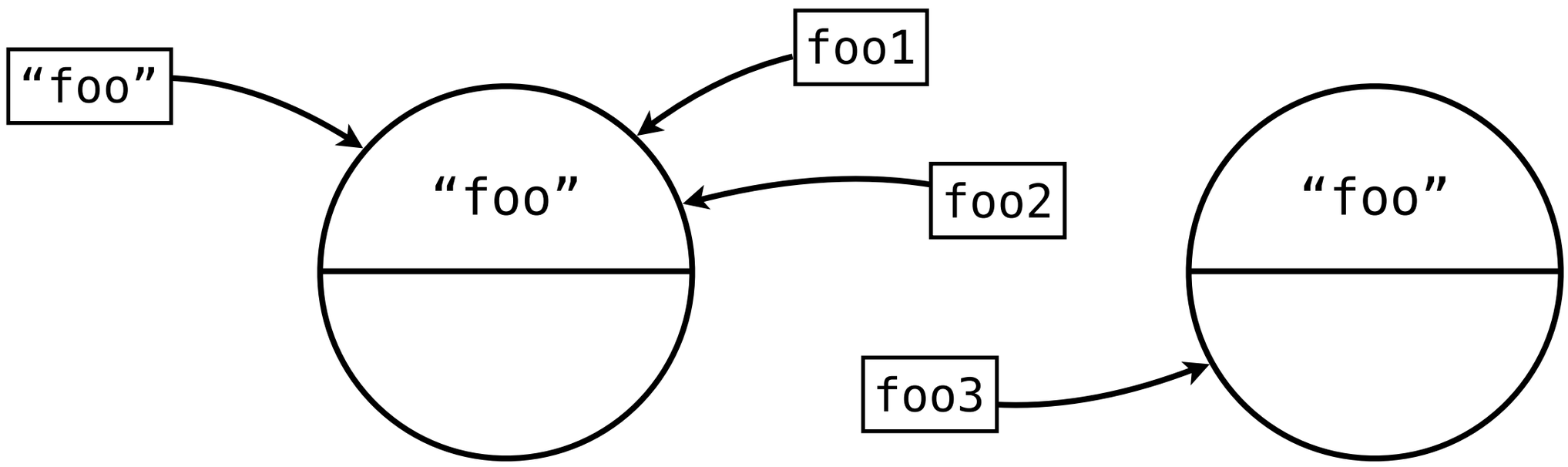
Reference Equality

```
String foo1 = “foo”;
String foo2 = “foo”;
System.out.println(foo1 == foo2);        // true
```



Reference Equality

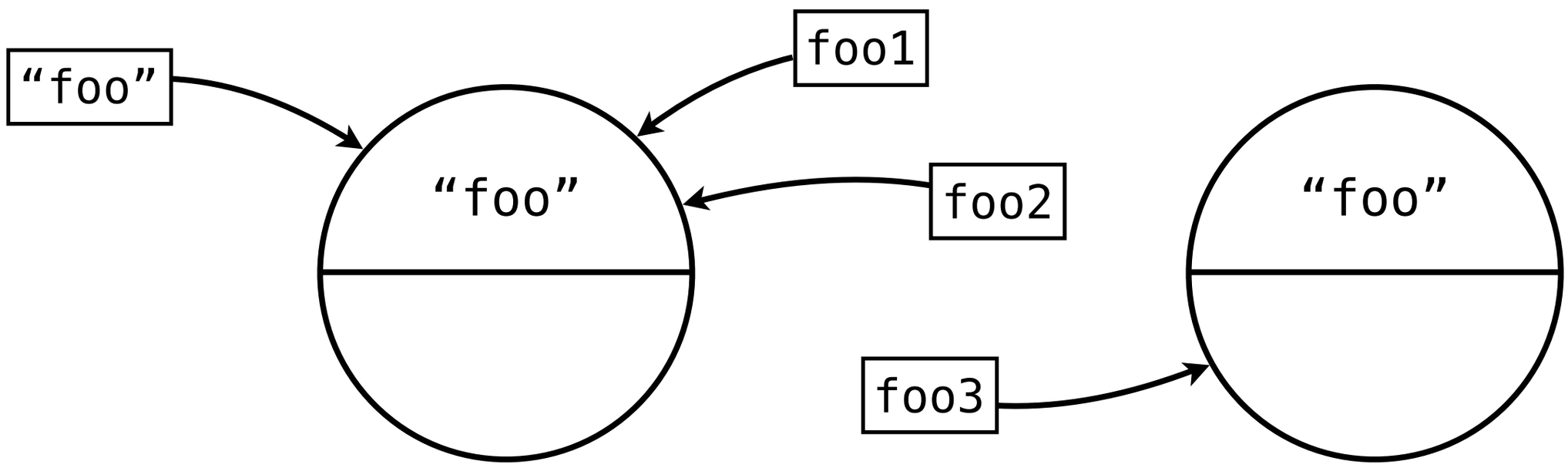
```
String foo1 = “foo”;
String foo2 = “foo”;
System.out.println(foo1 == foo2);        // true

String foo3 = new String(“foo”);
```



Reference Equality

"==" compares the value, which is a reference

```
String foo1 = “foo”;
String foo2 = “foo”;
System.out.println(foo1 == foo2);        // true

String foo3 = new String(“foo”);
System.out.println(foo1 == foo3);        // false
```

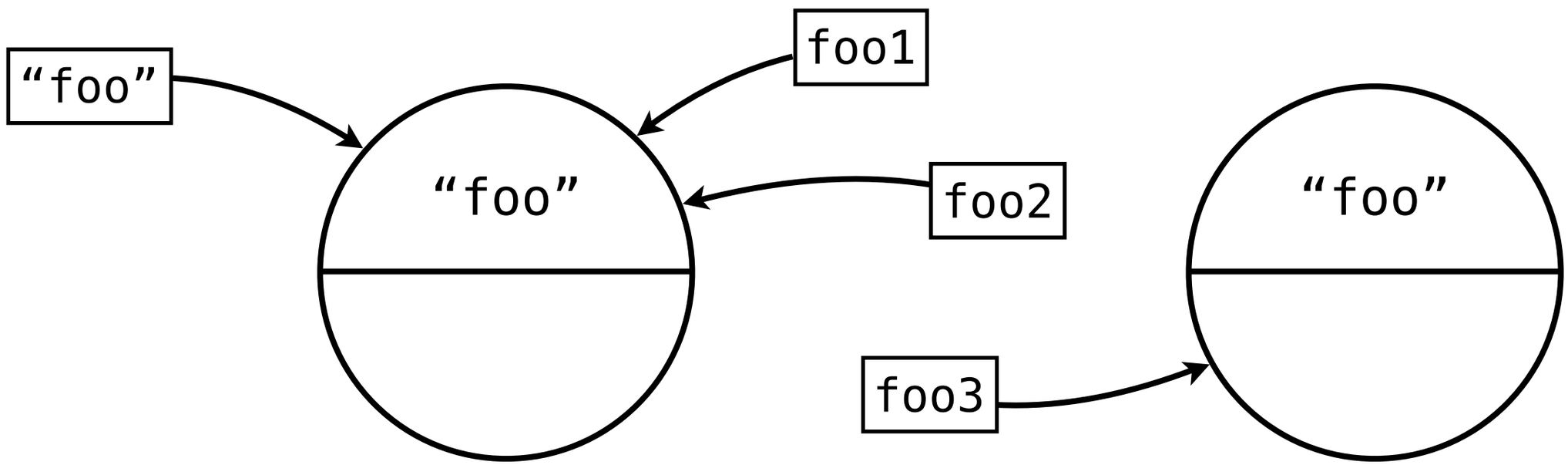


Reference Equality

"==" compares the value, which is a reference

```
String foo1 = "foo";
String foo2 = "foo";
System.out.println(foo1 == foo2);          // true

String foo3 = new String("foo");
System.out.println(foo1 == foo3);          // false
System.out.println(foo1.equals(foo3));     // true
```

"equals" compares the attributes in the object



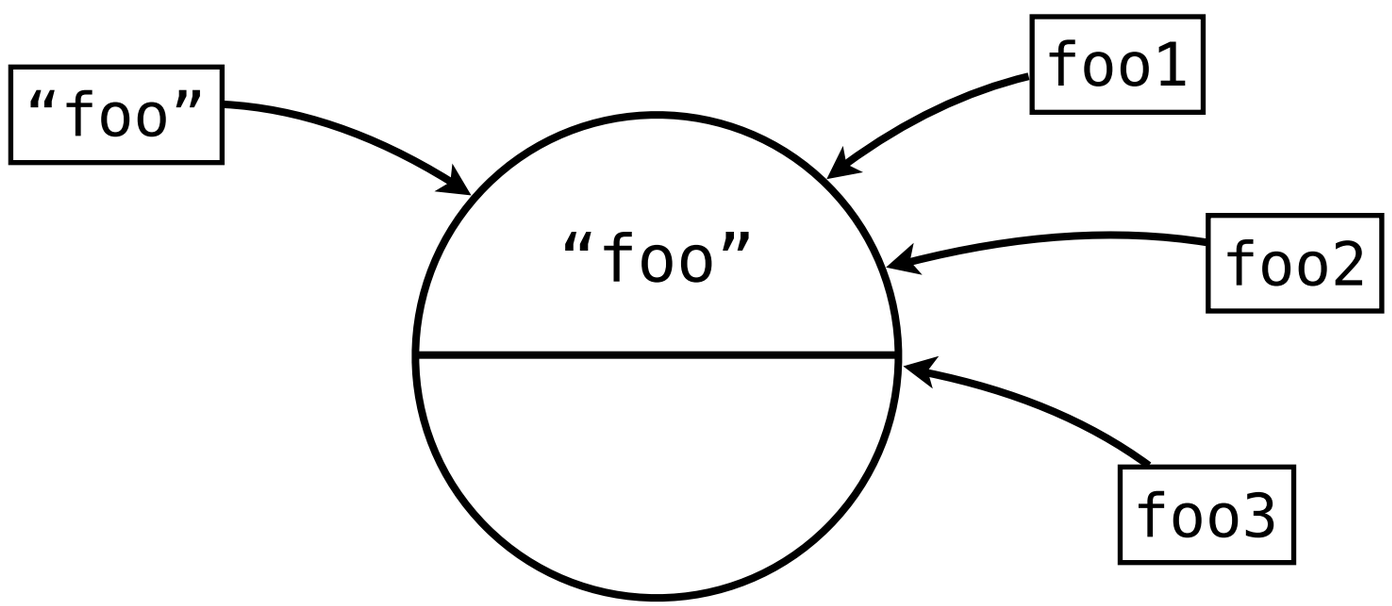
Reference Equality

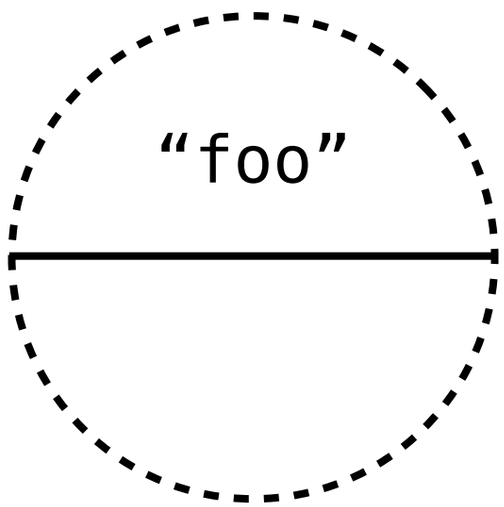
"==" compares the value, which is a reference

```
String foo1 = "foo";
String foo2 = "foo";
System.out.println(foo1 == foo2);          // true

String foo3 = new String("foo");
System.out.println(foo1 == foo3);          // false
System.out.println(foo1.equals(foo3));     // true
```

"equals" compares the attributes in the object

```
foo3 = foo1;
```





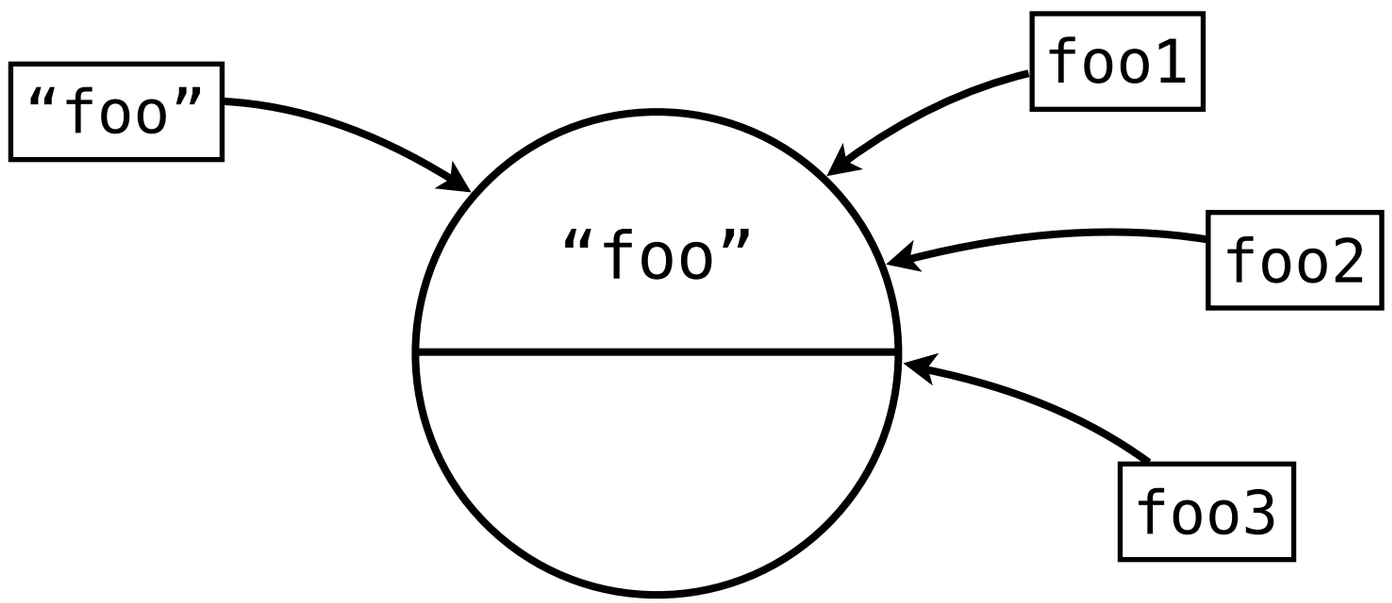
Reference Equality

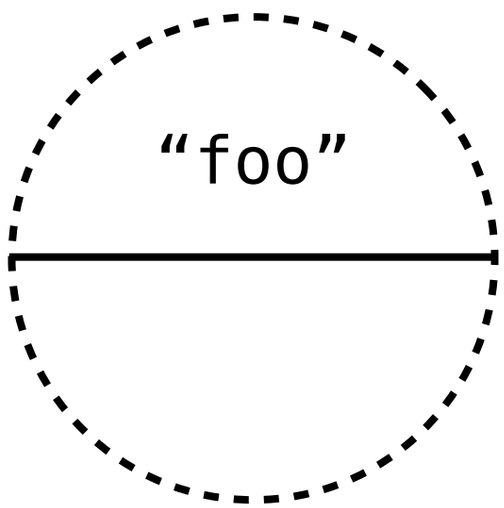
"==" compares the value, which is a reference

```
String foo1 = "foo";
String foo2 = "foo";
System.out.println(foo1 == foo2);          // true

String foo3 = new String("foo");
System.out.println(foo1 == foo3);          // false
System.out.println(foo1.equals(foo3));     // true
```

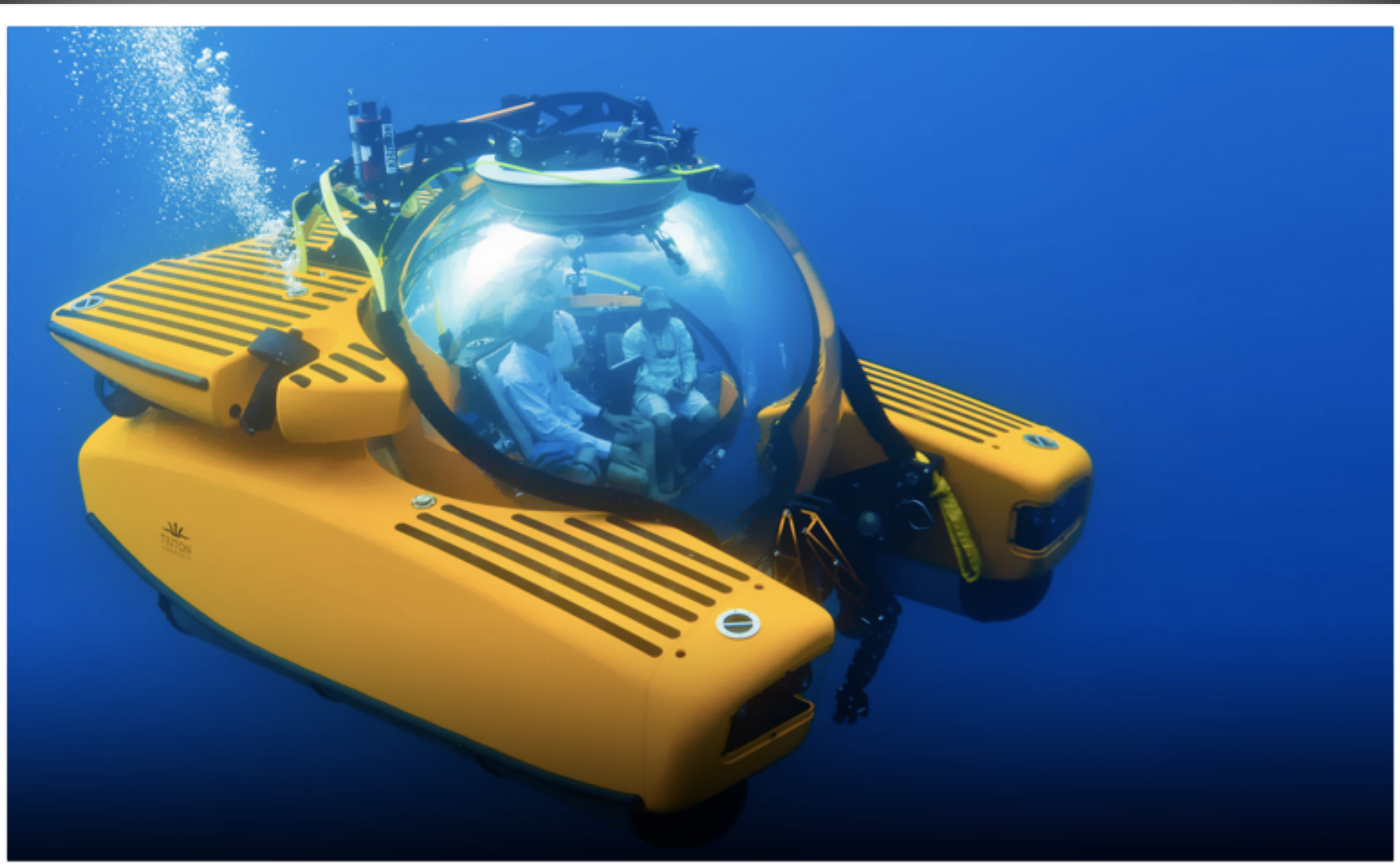
"equals" compares the attributes in the object

```
foo3 = foo1;
System.out.println(foo1 == foo3);          // true
```





Subclasses

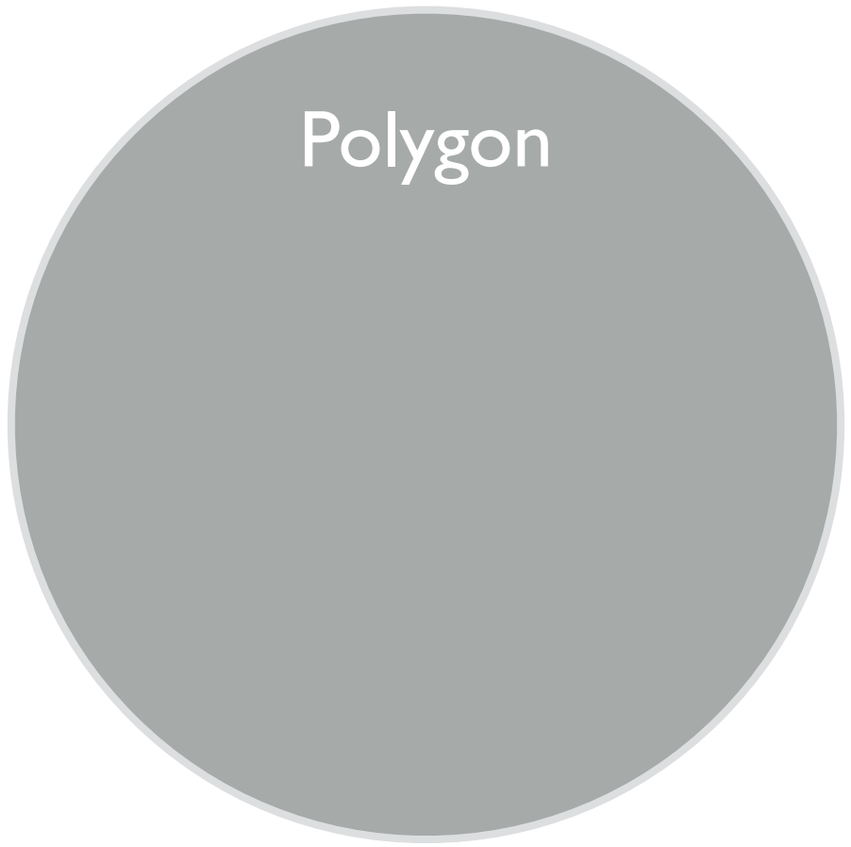


Subclasses

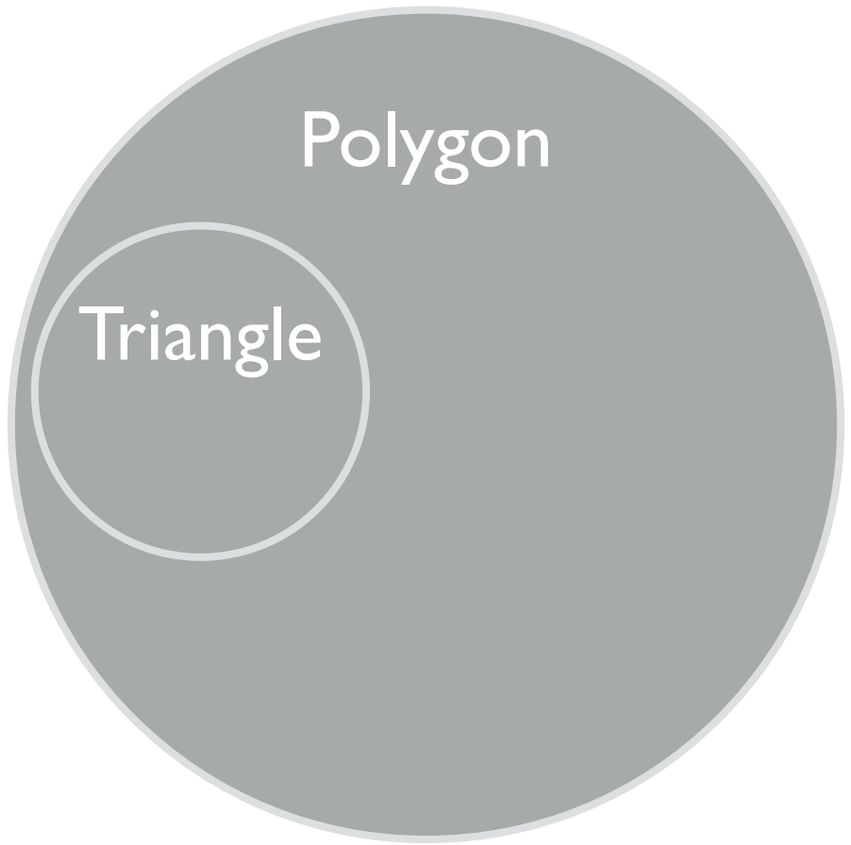
- Every object belongs to a class
- Classes are like types
- A value of a type can be used in place of a value of a subtype via *polymorphism*
- A subclass is a subtype with support for polymorphism and inheritance

subclass = subtype with polymorphism + inheritance

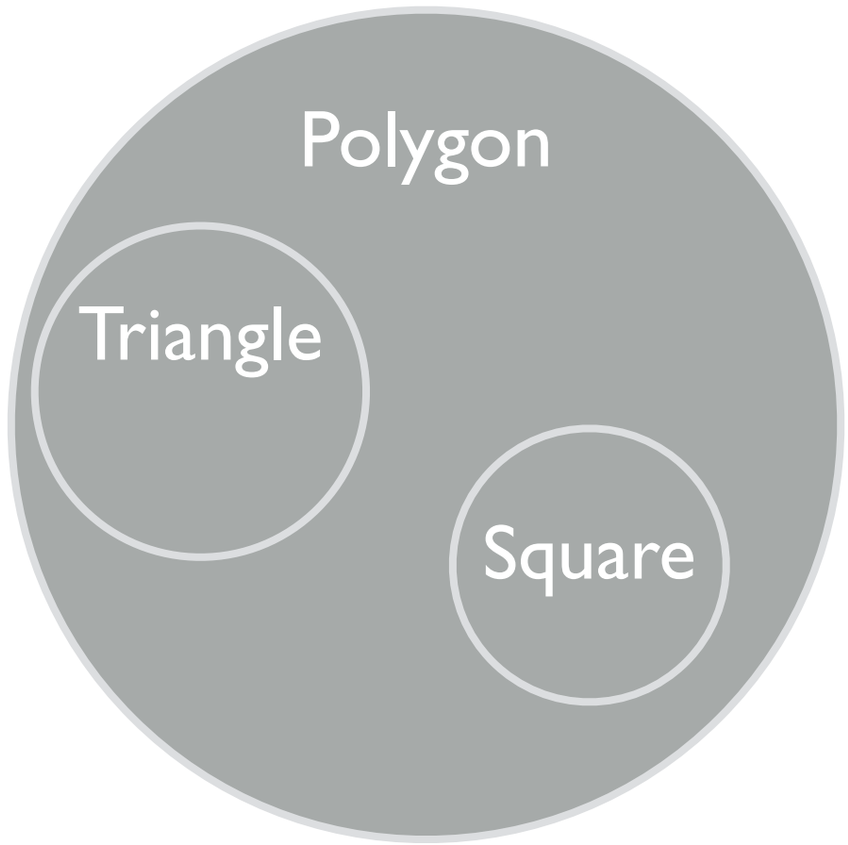
Subtypes



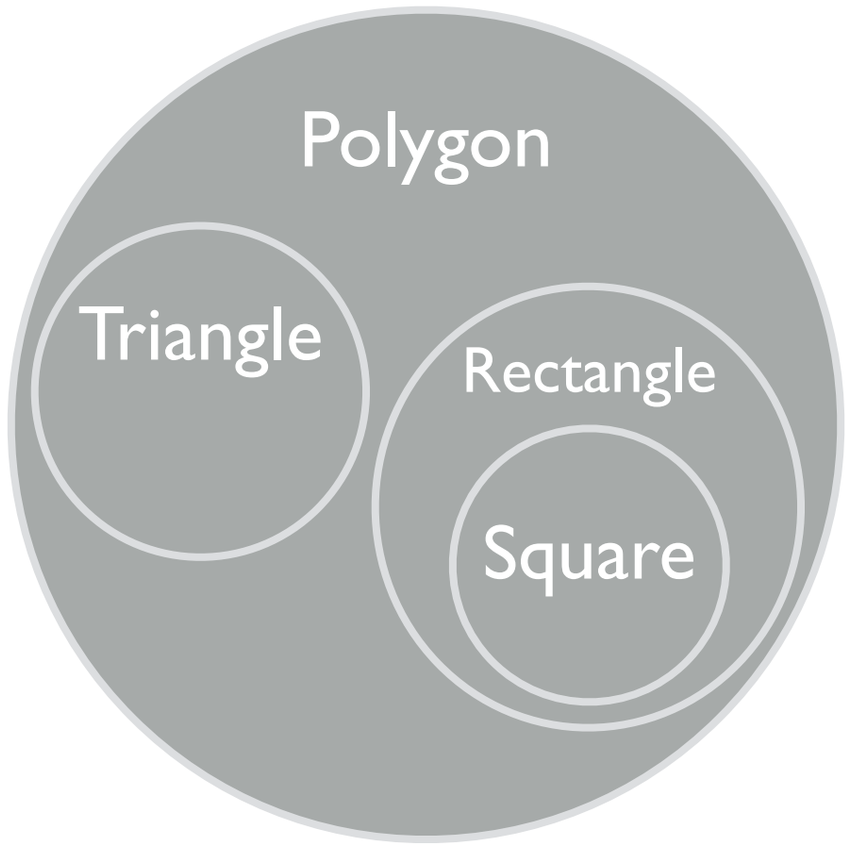
Subtypes



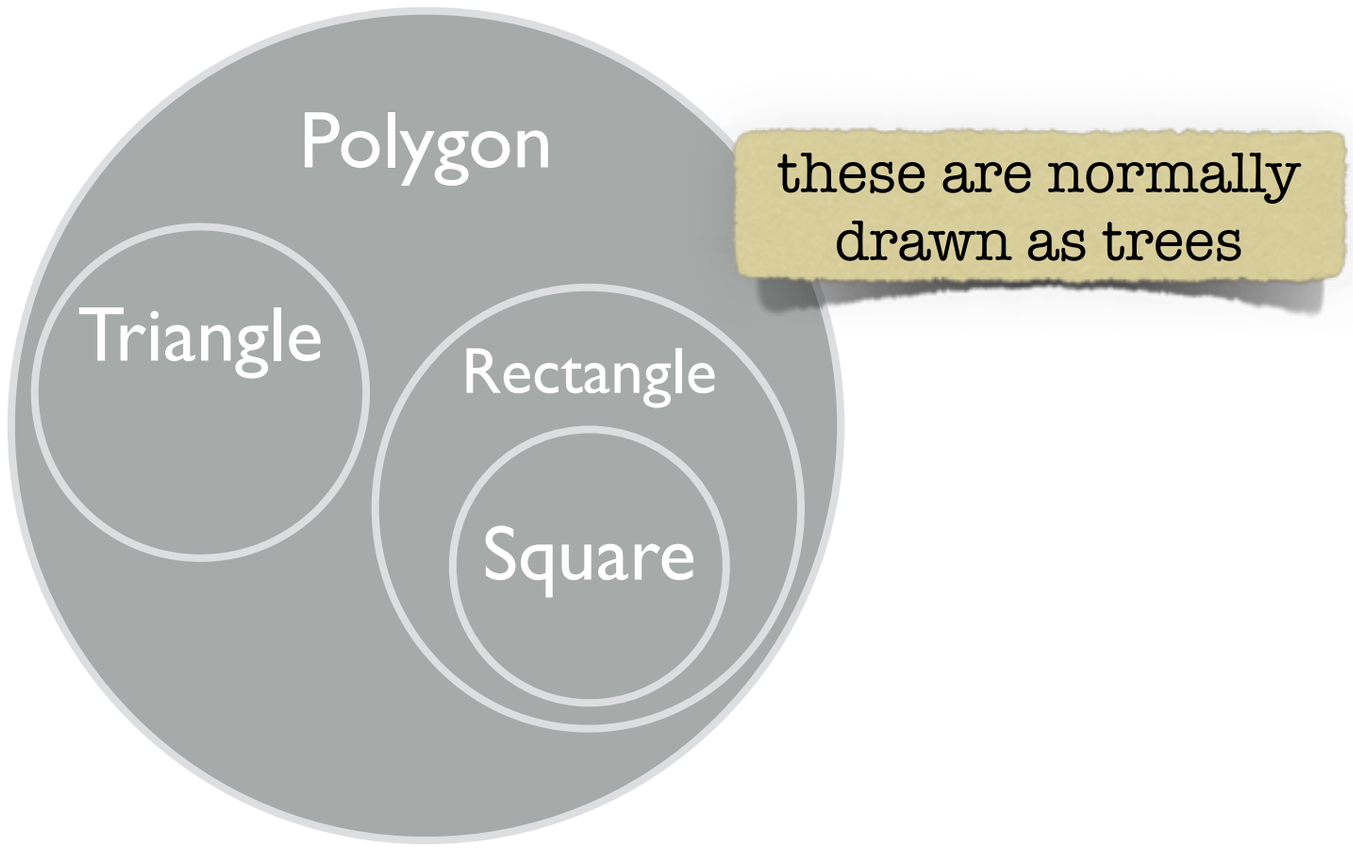
Subtypes



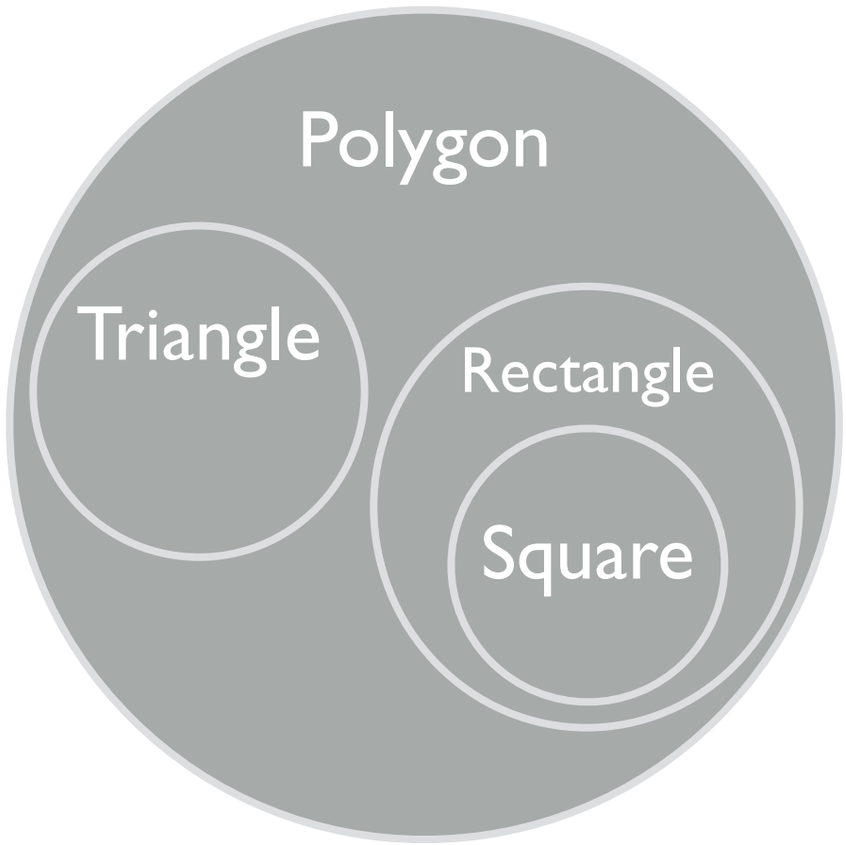
Subtypes

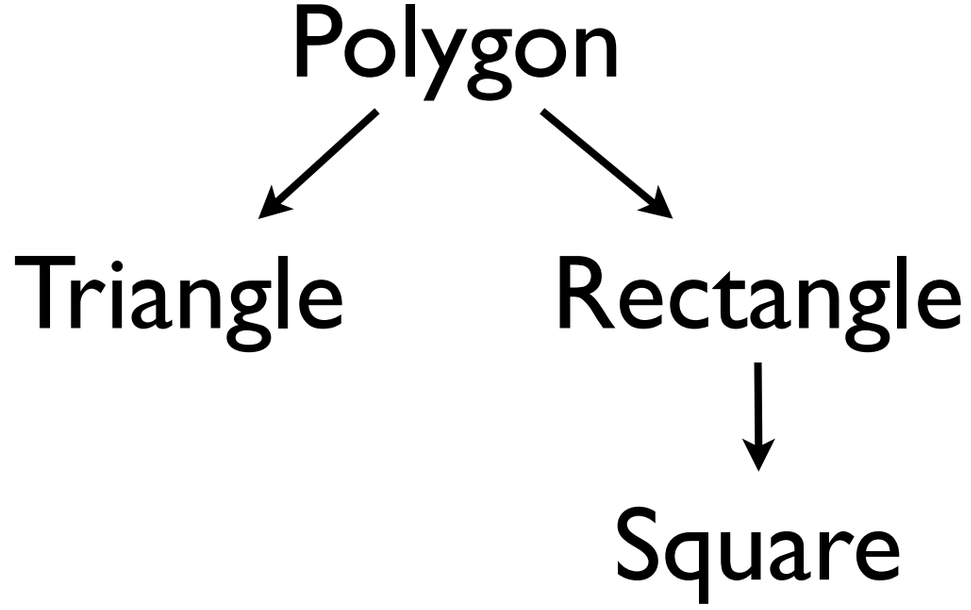


Subtypes


Polygon
Triangle
Rectangle
Square
these are normally drawn as trees

Subtypes

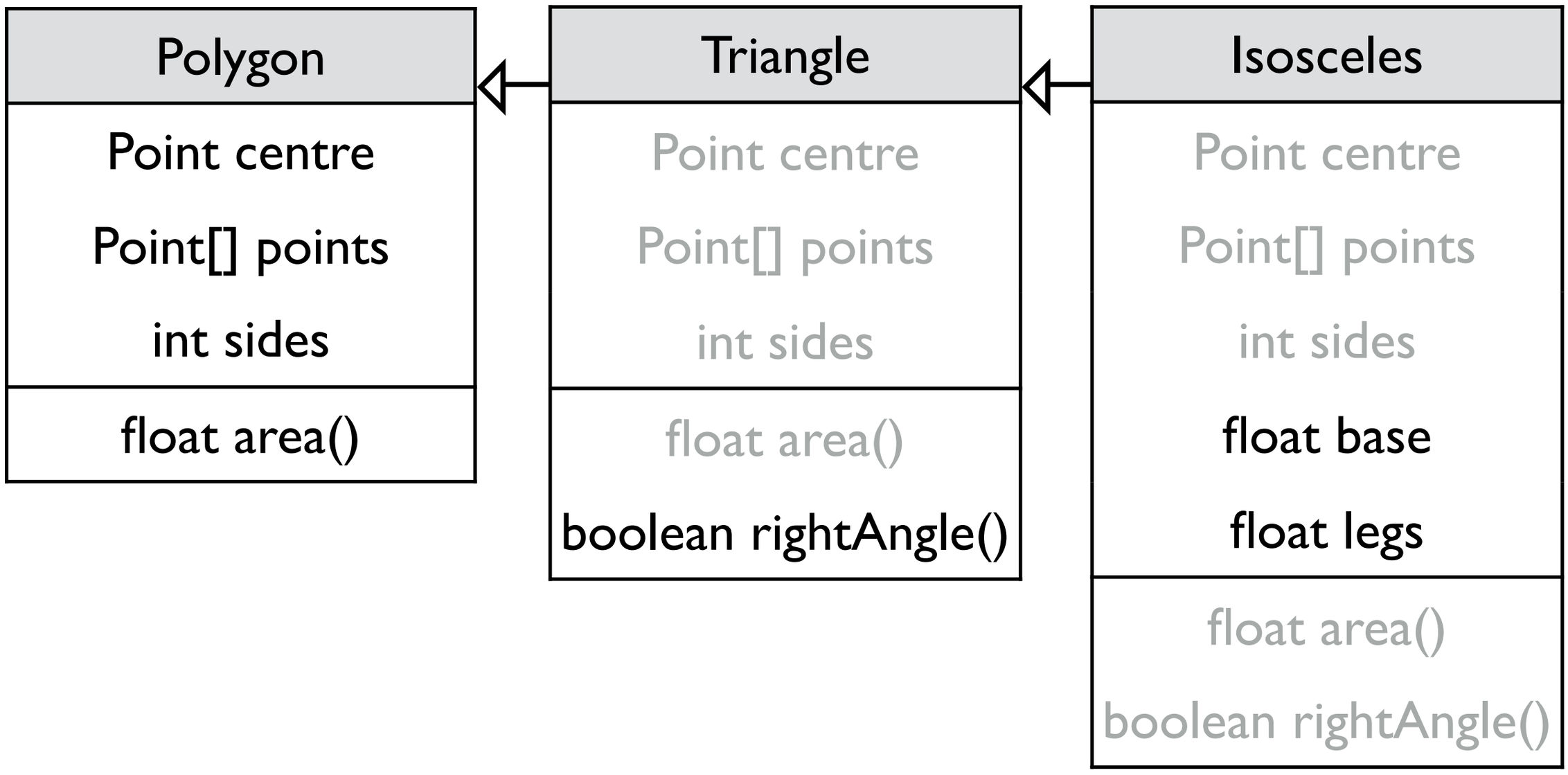

Polygon
Triangle
Rectangle
Square


Polygon
Triangle
Rectangle
Square

Inheritance

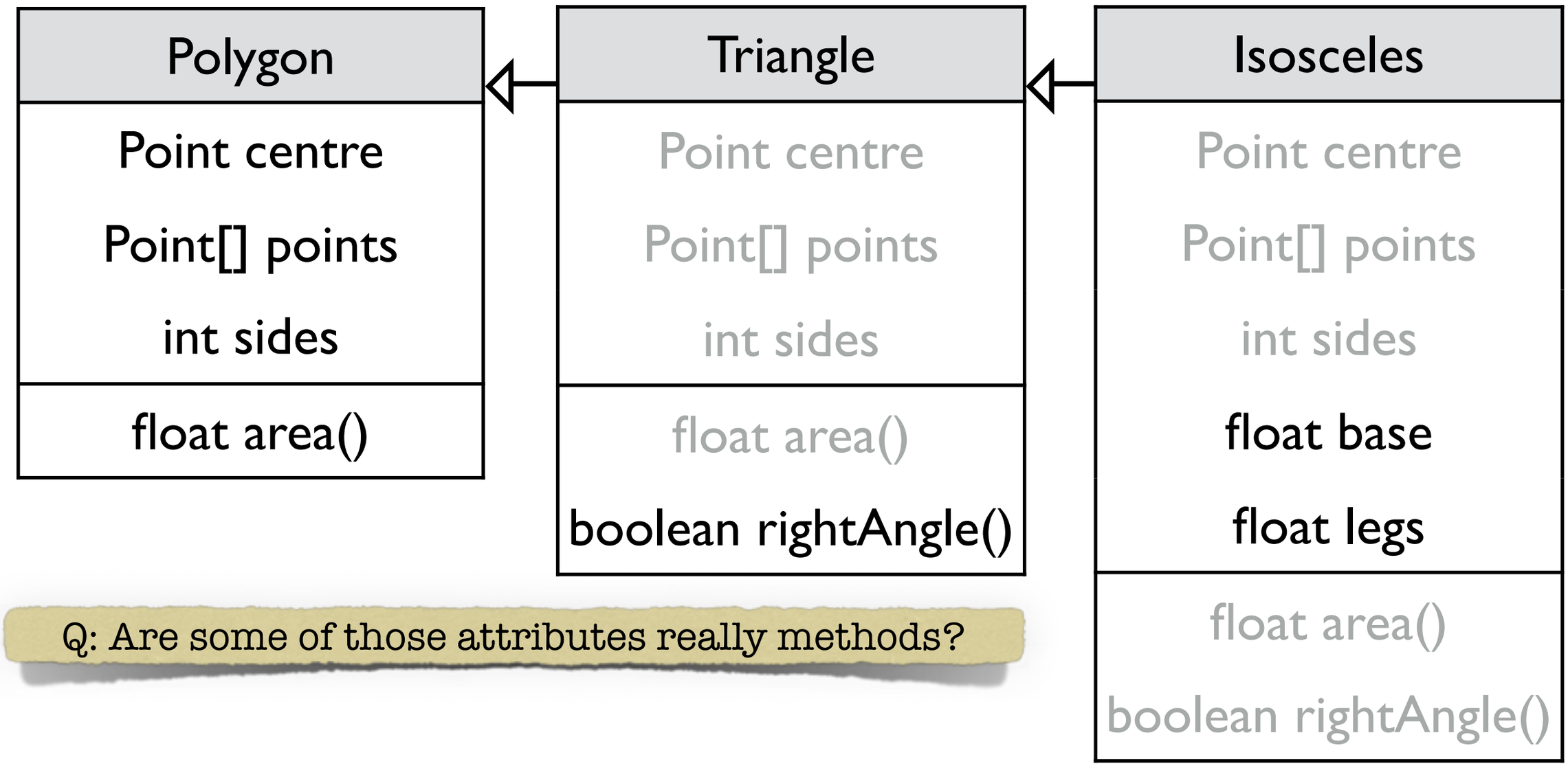
Inheritance

- Subclasses *inherit* the features of their parents
- Subclasses can have *additional* features



Inheritance

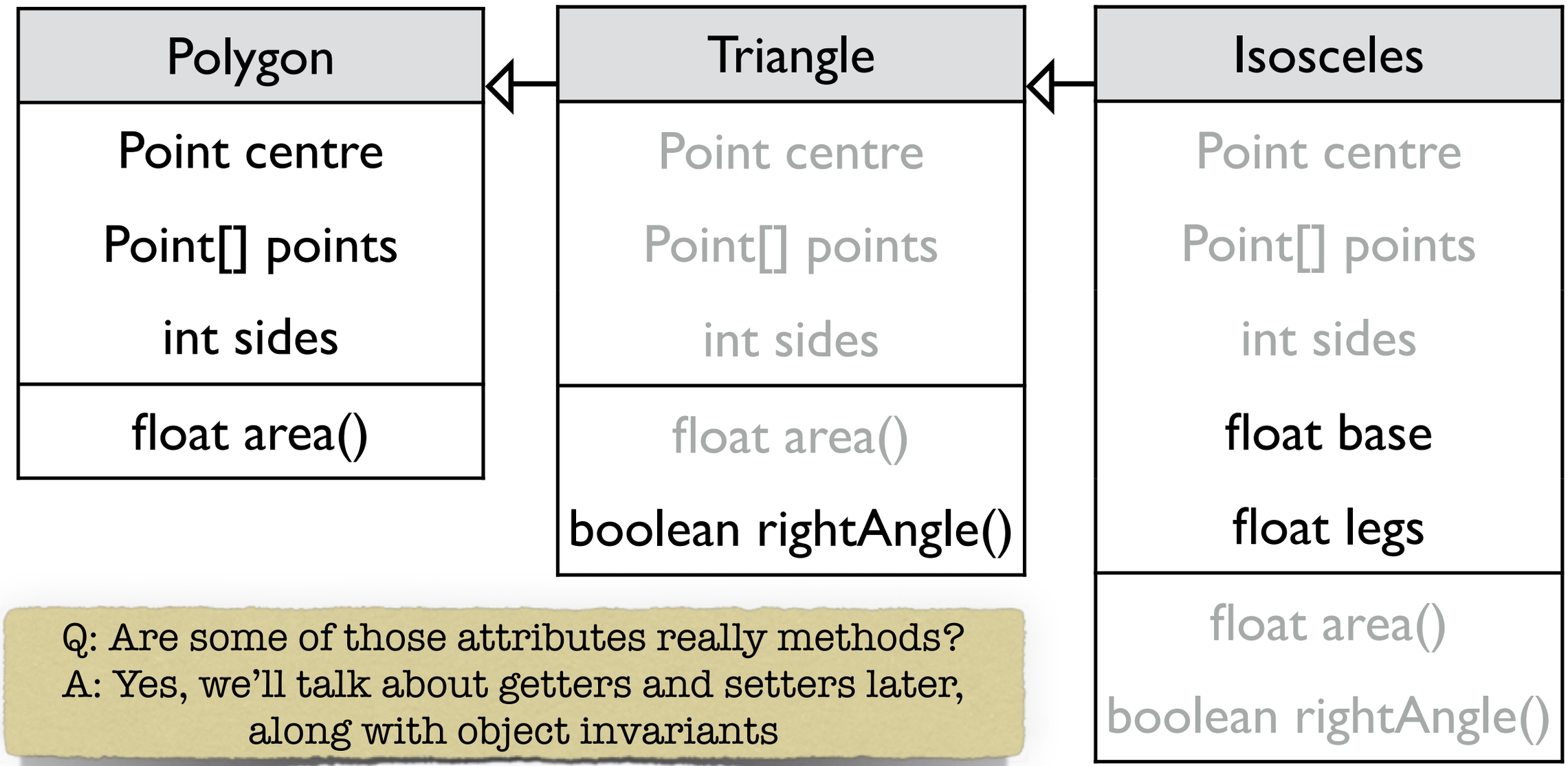
- Subclasses *inherit* the features of their parents
- Subclasses can have *additional* features



Q: Are some of those attributes really methods?

Inheritance

- Subclasses *inherit* the features of their parents
- Subclasses can have *additional* features



Q: Are some of those attributes really methods?
A: Yes, we'll talk about getters and setters later, along with object invariants

Inheritance

- Subclasses are implemented with *extends*:

```
class Polygon {
  Point[] points;
  int sides() { return points.length; };
}

class Triangle extends Polygon {
  boolean rightAngle() { … };
}

class Isosceles extends Triangle {
  double base;
  double legs;
}
```

Inheritance means we don't have to repeat the definition of features

Inheritance

- Inheritance by itself is primarily a way of achieving a reduction in code
- But it comes with a *big* warning: it's very easy to abuse!
- *Only* use inheritance when there is an *is-a* relationship, not a *has-a* relationship
- For example, a Shape might inherit from point, since every shape has a centre
  This would be a terrible mistake!

Inheritance

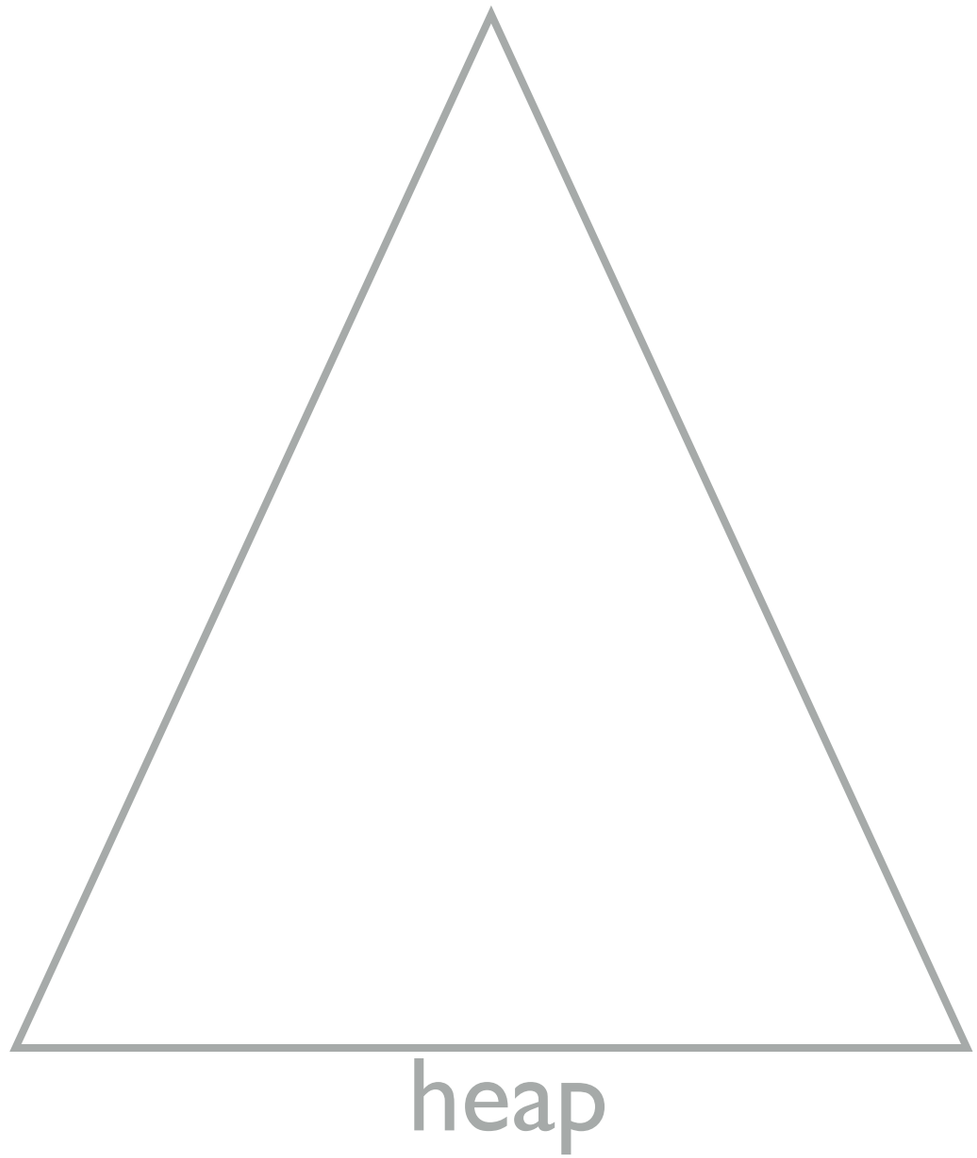
- If you're not sure about inheritance, then think about the *Liskov Substitution Principle:*
  - Liskov substitution principle: *properties* of a program don't change under subtyping (though *behaviour* might!)
- Should Square extend Rectangle, or Rectangle extend Square?
- What are the properties you'd expect to be preserved?

Polymorphism

Polymorphism

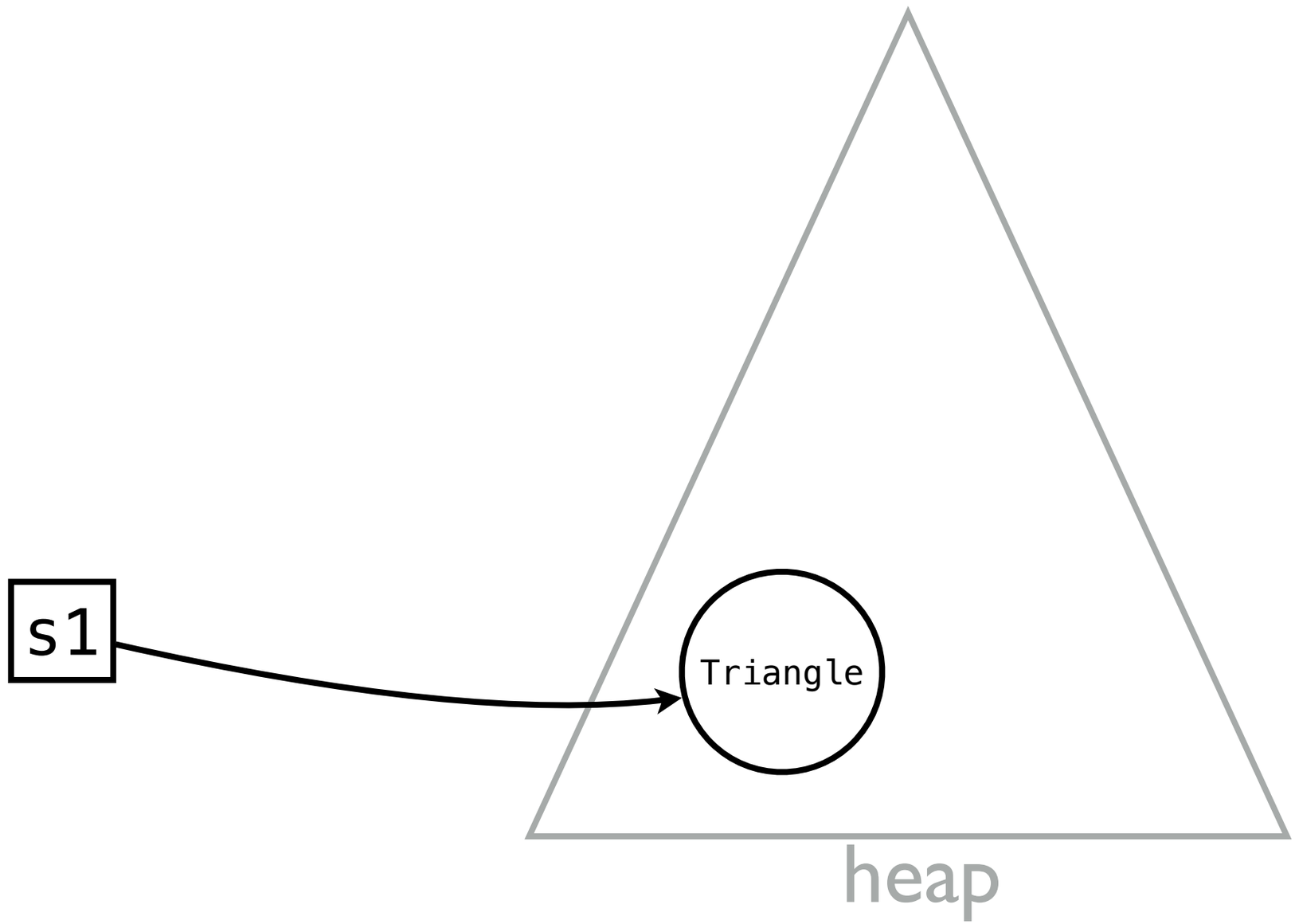
- Polymorphism lets you use an object of a subclass as if it was an object in its parent class
  - Every reference belongs to a class
  - A reference can be *to* any object in a subclass of the reference's class
  - This does *not* change the reference's class

Polymorphism



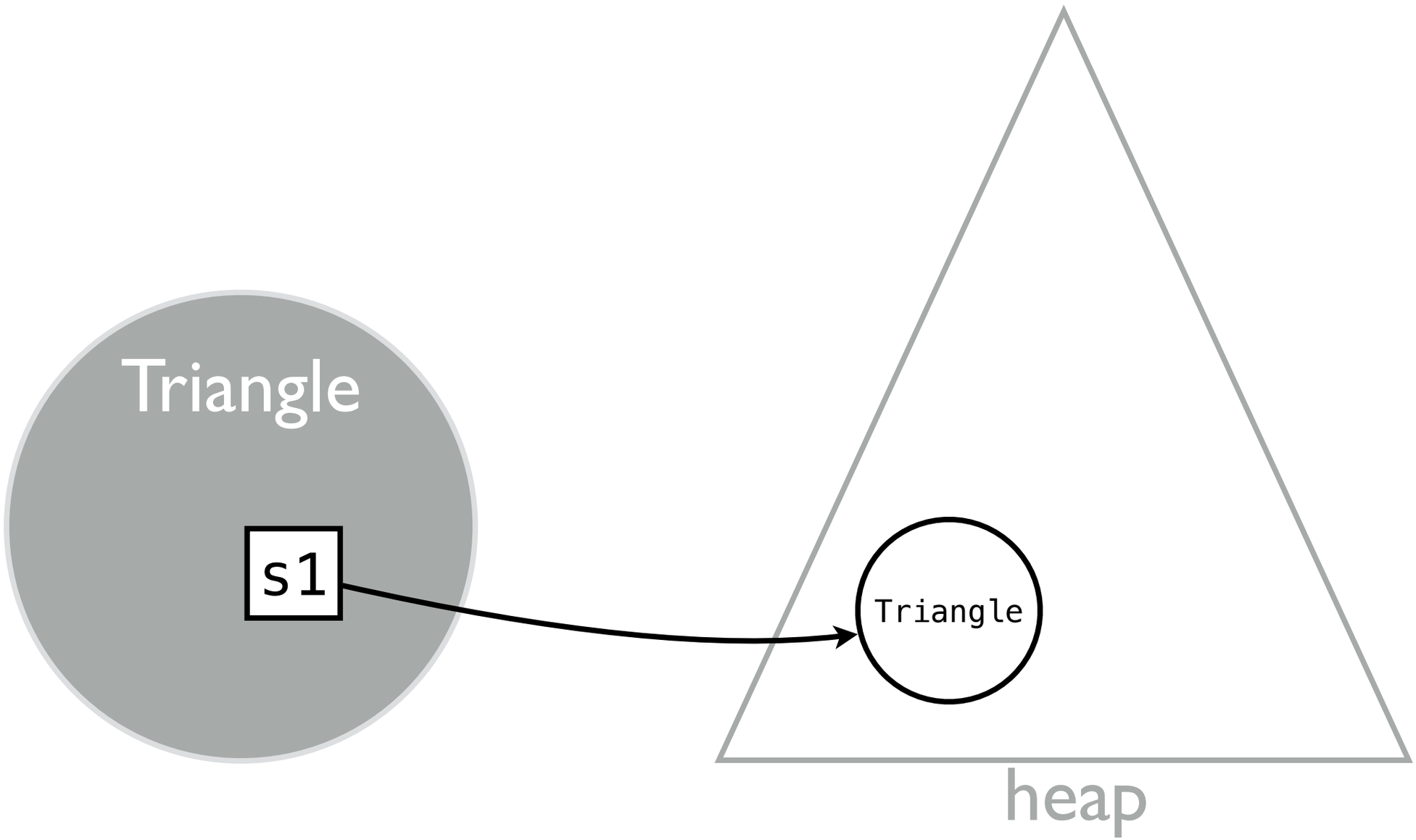
Polymorphism

```
Triangle s1 = new Triangle();
```



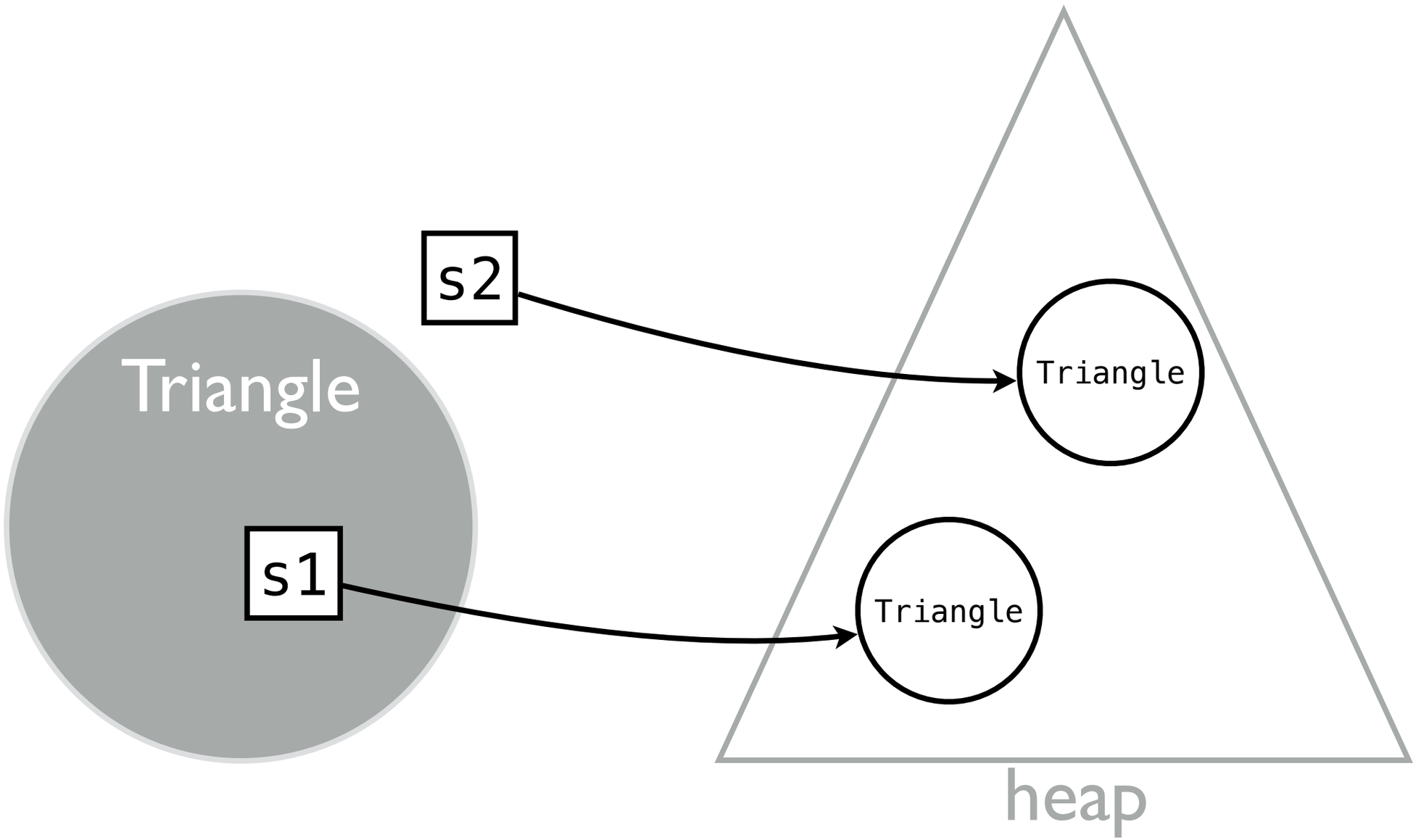
Polymorphism

```
Triangle s1 = new Triangle();
```



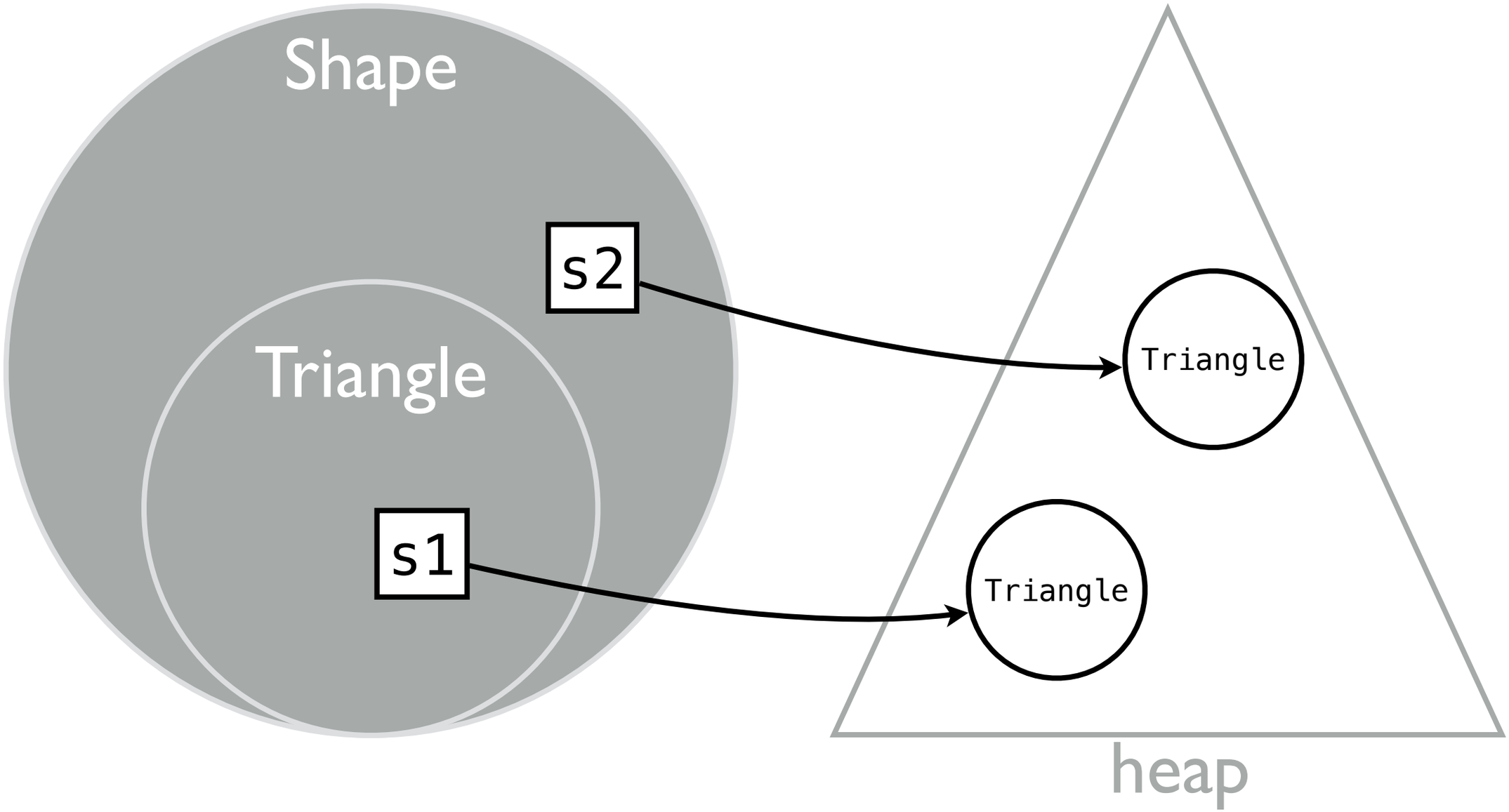
Polymorphism

```
Triangle s1 = new Triangle();
Shape    s2 = new Triangle();
```



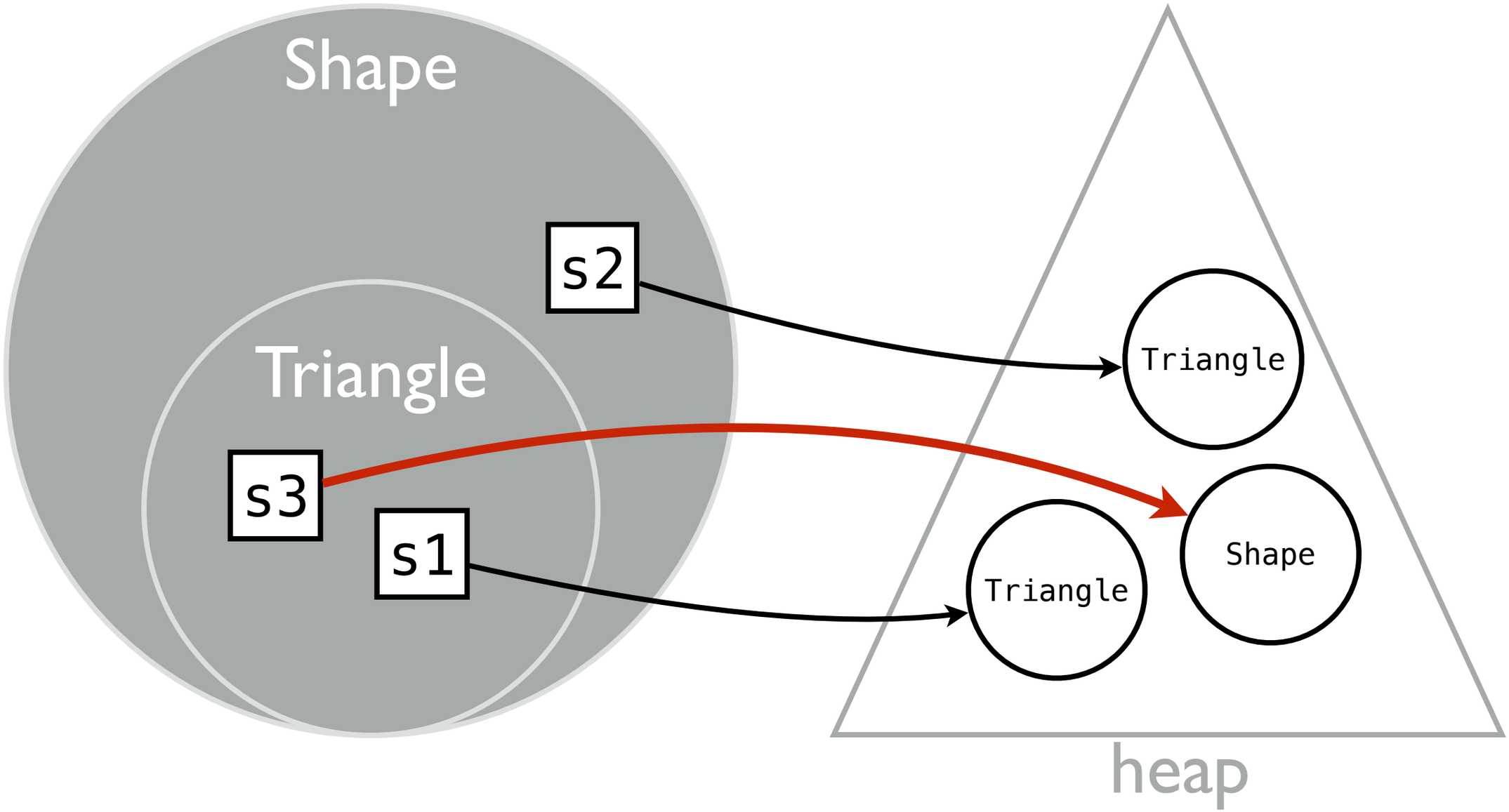
Polymorphism

```
Triangle s1 = new Triangle();
Shape    s2 = new Triangle();
```



Polymorphism

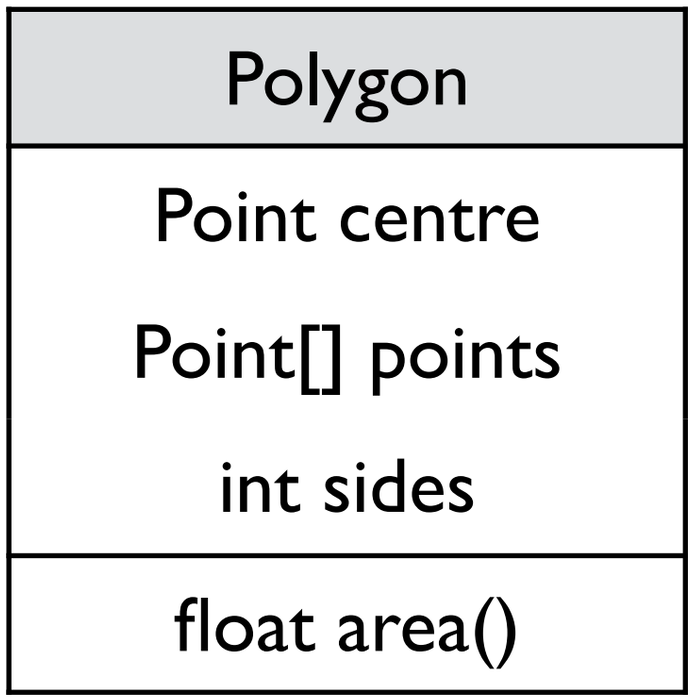
```
Triangle s1 = new Triangle();
Shape    s2 = new Triangle();
Triangle s3 = new Shape(); // BOGUS!
```



Overriding

Overriding

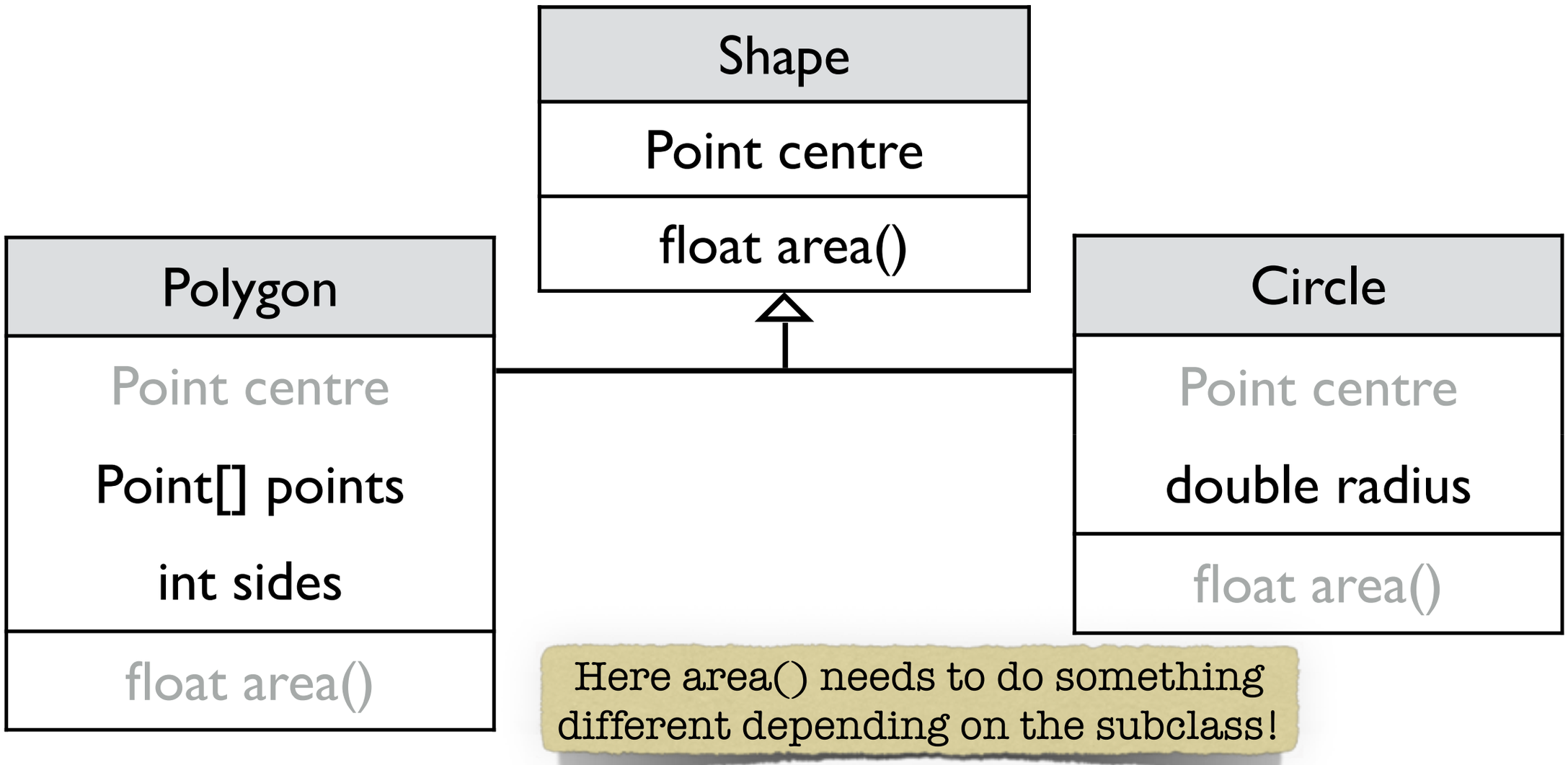
- Subclasses *inherit* the features of their parents
- But sometimes behaviour needs *overriding*



Here we got lucky, since area() was the same for all subclasses

Overriding

- How could we implement area() when the children are fundamentally different?



Overriding

- Subclasses *inherit* the features of their parents
- But sometimes behaviour needs *overriding*
- Overriding lets you provide specific behaviour to be used for subclasses

```
class Shape {
  Point centre;
  abstract double area();
}
```

```
class Polygon extends Shape {
  Point[] points;
  int sides;

  @Override
  double area() {
    … // implement algorithm
  }
  …
}
```

```
class Circle extends Shape {
  double radius;

  @Override
  double area() {
    return (Math.PI * radius * radius);
  }
  …
}
```

Overriding

- Overriding happens in many places, such as the equals() and clone() methods
- Some people add @Override: it helps the compiler to catch your typos

```
class Polygon extends Shape {
  Point[] points;
  int sides;


  double arae() {
    … // implement algorithm
  }
  …
}
```

without @Override, everything compiles, but your program is mysteriously buggy

Overriding

- Overriding happens in many places, such as the equals() and clone() methods
- Some people add @Override: it helps the compiler to catch your typos

```
class Polygon extends Shape {
  Point[] points;
  int sides;

  @Override
  double arae() {
    … // implement algorithm
  }
  …
}
```

With @Override, the compiler says "huh, what's arae?"

@Override helps the compiler help you!

Overriding

- Overriding provides a new implementation to be used instead of the inherited one
- But what if we also want to reuse the inherited method?
- We can use the **super** keyword to access things in the parent class

Overriding

- For instance, suppose our Triangle class wants to reuse the area() method from Polygon, but also wants to print out a message
- We could copy-paste the code, but this is evil
  - Instead, we invoke our parent's method

```
class Triangle {
  …
  double area() {
    System.out.println("I'm a triangle area!");
    super.area();
  }
}
```

Overloading

Overloading

- Sometimes there might be different ways of asking an object to do the same thing
- For example, we might want to construct a triangle either by giving 3 points, or by providing an array
- Rather than have different method names, one for each case, we can *overload*

Overloading

- Sometimes there might be different ways of asking an object to do the same thing
- For example, we might want to construct a triangle either by giving 3 points, or by providing an array
- Rather than have different method names, one for each case, we can *overload*

Do not confuse this with overriding; overriding is about subclasses, overloading is about accepting different parameters

Overloading

- For instance, here's how we would overload the constructor for Triangle

```
class Triangle {
  Triangle(Points[] points) {
    super(points);
  }

  Triangle(Point p0, Point p1, Point p2) {
    super(new Point[] {p1, p2, p3})
  }
  …
}
```

Note that we reuse the parent constructor by calling super

Dispatch

Dispatch

- Messages are dispatched according to the *dynamic* receiver, and *static* parameter

```
Animal scooby = new Dog();
Food   snack  = new Chocolate();

scooby.eat(snack);
```

dynamic: Dog

static: Food

Dispatch

- Dispatch can be thought of as a kind of (primitive) pattern matching from your Haskell days

```
data Tree = Leaf Int | Fork Tree Tree
```

How can we encode this in Java?

Tree-Total

- Show how to implement trees using classes
- Find the total of values in leaves

Tree-Total

- We start with an abstract tree class
- It promises that it can find the total()

```
abstract class Tree {
   abstract int total();
}
```

I'll talk more about abstract later ... it roughly means you can't make an object of this class directly

Tree-Total

- We start with an abstract tree class
- It promises that it can find the total()

```
abstract class Tree {
  abstract int total();
}
```

- What is the relationship between a Tree, a Leaf, and a Fork?

```
data Tree = Leaf Int | Fork Tree Tree
```

Tree-Total

- Leaf and Fork are both Trees!
- A Leaf must simply return its value

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  int total () {
    return value;
  }
}
```

Tree-Total

- A Fork has to return the total of its children

```
class Fork extends Tree {
  Tree l;
  Tree r;

  Fork(Tree l, Tree r) {
    this.l = l;
    this.r = r;
  }

  int total() {
    return (l.total() + r.total());
  }
}
```

Tree-Total

- We can test this with a little main function

```
public static void main (String[] args) {
  Tree l1   = new Leaf(1);
  Tree l2   = new Leaf(2);
  Tree l3   = new Leaf(3);
  Tree n12  = new Fork(l1, l2);
  Tree n123 = new Fork(n12, l3);

  System.out.println("total: " + n123.total());
}
```

Multiple Dispatch

Multiple Dispatch

- Is it possible to make a dispatch dynamic in *both* parameters?

Multiple Dispatch

- Is it possible to make a dispatch dynamic in *both* parameters?
- Not directly, but we can be cunning!

(we'll talk about double dispatch, but this generalises)

Multiple Dispatch

- Our task will be to make the operation performed on the tree be more generic
- For instance we might want to:
  - sum all the values in the tree
  - multiply all the values in the tree
  - count the number of elements in the tree
  - print the structure of the tree

Multiple Dispatch

- In other words, we want to visit the tree with different operations:

```
public static void main (String[] args) {
  Tree l1   = new Leaf(1);
  Tree l2   = new Leaf(2);
  Tree l3   = new Leaf(3);
  Tree n12  = new Fork(l1, l2);
  Tree n123 = new Fork(n12, l3);

  SumVisitor visitor = new SumVisitor();
  n123.accept(visitor);
  System.out.println("total: " + visitor.total);
}
```

Multiple Dispatch

- A Tree only promises to accept visitors

```
abstract class Tree {
  abstract void accept(Visitor v);
}
```

- A Visitor knows how to visit the tree

```
abstract class Visitor {
  abstract void visit(Fork t);
  abstract void visit(Leaf t);
}
```

Multiple Dispatch

- A Fork simply accepts and passes on

```
class Fork extends Tree {
  Tree l;
  Tree r;

  Fork(Tree l, Tree r) {
    this.l = l;
    this.r = r;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

Multiple Dispatch

- A Leaf also just accepts and passes on

```
class Leaf extends Tree {
  int value;

  Leaf(int value) {
    this.value = value;
  }

  @Override
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

Multiple Dispatch

- The visitor needs to know how to visit the different nodes

```
class SumVisitor extends Visitor {
  int total;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  }

  @Override
  void visit(Leaf leaf) {
    total += leaf.value;
  }
}
```

Multiple Dispatch

- Now different visitors are quite easy to implement

```
class MaxVisitor extends Visitor {
  int maximum;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  }

  @Override
  void visit(Leaf leaf) {
    maximum = Math.max(maximum, leaf.value);
  }
}
```

Multiple Dispatch

- Now different visitors are quite easy to implement

```
class MaxVisitor extends Visitor {
  int maximum;

  @Override
  void visit(Fork fork) {
    fork.l.accept(this);
    fork.r.accept(this);
  }

  @Override
  void visit(Leaf leaf) {
    maximum = Math.max(maximum, leaf.value);
  }
}
```

there are variations of this where we return different values ... more later