CS 228: Introduction to Data Structures
Lecture 5

An Extended Example (Continued)

Please refer again to the `Insect` class on Blackboard. As before, we assume that each part is executed separately.

5. Static type checking.

```
Locomotion l = new Katydid(3,“Brown”);// OK
// static type: Locomotion
// dynamic type: Katydid
~~l.attack()~~;  // ERROR: attack() undefined
                 // for Locomotion (static
                 // type)
```

6. Dynamic binding.

```
Locomotion l = new Locust(2, "Black");// OK
// static type of l: Locomotion (interface)
// dynamic type of l: Locust ->  Bee
```

```
l.move();  // "hop" - use the version for
           // Grasshopper (closest ancestor
           // of dynamic type)

l = new Bee(1, "Gold", “Hill"); // OK
// dynamic type of l: Bee

l.move();  // "fly" - use the version for
           // Bee (dynamic type)
```

7. **Inheritance / Dynamic binding / method overriding.**

```
Insect i = new Katydid(2, “Green");
// i’s static type: Insect (abstract class)
// i’s dynamic type: Katydid

i.attack();  // "bite"

i = new Mantis (4, "Green");
// i’s dynamic type: Mantis

i.attack();  // "strike"
```

8. Static type checking / dynamic binding.

```
Insect k = new Katydid(3, "Green");
// k’s static type: Insect
// k’s dynamic type: Katydid

k.attack();  // OK: ”bite"

k.antennae(); // ERROR: antennae() is
              // undefined for Insect
```

9. Dynamic binding.

```
Grasshopper g = new Katydid(3, “Golden");
// g’s static type: Grasshopper
// g’s dynamic type: Katydid

System.out.println(g.feedOn());
                              // "variety"
```

10. Static type checking.

```
Pollination p = new Bee(1, "Golden-Black",
"Lake");
p.getSwarm(); // compile error:
              // getSwarm() undefined
              // method for Pollination
              // (static type)
```

The error can corrected with a downcast:

```
((Bee) p).getSwarm();  // OK: ”Lake"
```

11. Static type checking / dynamic binding.

```
Mantis m = new Mantis(5, "Green");
m.move();  // "crawl"

Insect i = m.preyOn();
i.move();  // compile error: move()
           // undefined method for Insect
```

The error can corrected with a downcast:

```
((Grasshopper) i).move();  // "hop"
```

12. Casts change the static type.

```
Insect i = new Mantis(4, "Yellow");
((Mantis) i).move();    // "crawl"
((Mantis) i).preyOn().attack();
// preyOn() returns a Locust; "bite"

i = new Bee(1, "Golden-Black", "Hill");
((Bee) i).makeHoney();  // "Orange Blossom"
```

13. The compiler always downcast to a subclass and upcast to a superclass.

```
Mantis m = new Mantis(4, “Green");
((Insect) m).attack();  // OK: ”bite"
```

This works because `Mantis` is a subtype of `Insect`. It does not accomplish much, however, since the `attack()` method used will anyway be that of a `Mantis` (by dynamic binding).

On the other hand, the following compiles, but fails at run time.

```
((Bee) ((Insect) m)).makeHoney(); // ERROR:
// ClassCastException: Mantis cannot be
cast to
// Bee
```

The statement passes static type checking, because `Insect` is a super-type of `Mantis` and `Bee` is a subtype of `Insect`. It fails because the dynamic type of m is `Mantis`, which is not a subtype of `Bee`.

In general, upcasting a class variable to a superclass is not useful other than to fool the compiler. There are two reasons:

(i) The class already inherits every method that the superclass is willing to share.

(ii) By dynamic binding, when a method that is implemented by both a class and a superclass is called, the version for the class is always invoked.

Access Modifiers

There are four access modifiers that may be applied to fields and methods.

- **`private`:** accessible only within the class, used for implementation details
- (none) aka **"package-private":** accessible from any class within the package
- **`protected`**: accessible from subclasses (or from any class within the package)
- **`public`:** accessible from any class

We sometimes use `protected` access when designing a class to be extended: if you need subclasses to have access to a variable or method that is not part of the public API, make it `protected`.

Note that if a variable is non-private, you should never ***shadow*** it (re-declare a variable with the same name) in the subclass, even though this is allowed by the compiler.

You can override methods, but there's no such thing as overriding variables!

Package-private access is rarely used. It may be useful for short programs where multiple classes and interfaces are in the same file. (In contrast, a public class or interface is required to be in its own file.)

Package-private (default) access is ***almost*** synonymous with `protected` access. The `protected` modifier specifies that the member can only be accessed within its own package (as with package-private) and, ***in addition***, by a subclass of its class in ***another*** package.

The Root of the Java Class Hierarchy

At the root of the Java class hierarchy is the class `java.lang.Object`. Every class in Java is a subclass of this class.

The `Object` class has several predefined methods. One of them is **`toString()`**, which returns a `String` representation of an object. When Java is asked to print an object `x` — for instance when we invoke `System.out.println(x)` — what is actually printed is `x.toString()`.

The default implementation of `toString()` returns a string consisting of the class name, the ‘`@`' character, and the unsigned hexadecimal representation of the “hash code” of the object (we’ll see hashing and hash codes towards the end of the semester).

```
Insect  i = new Katydid(2, “Green");

System.out.println(toString());
        // insect.Katydid@7852e922
```

Since this string is typically meaningless to us, we will usually override `toString()` to provide a more useful description. A simple example of how to do this for the `Complex` class — which represents complex numbers — is shown below. (See also Blackboard.)

```java
public class Complex {
     private double re, im; // real and imaginary parts

     public Complex(double re, double im)
     {
          this.re = re;
          this.im = im;
     }

     // common format is "re + i im"
     @Override
     public String toString()
     {
          if (im >= 0)
               return String.format(re + " + i " + im);
          else
               return String.format(re + " - i " + -1 * im);
     }
}
```

Another predefined method in the `Object` class is **`getClass()`**, which returns the runtime class of an object. This class is represented by a special, unique, object of type `Class`.

```java
System.out.println(i.getClass());
              // class insect.Katydid
```

You will find a complete list of the methods in the `Object` class online[1]. In the next few lectures, we will focus on two of these methods: `equals()` and `clone()`.

Primitive Types and Object Types

As a Java program is executed, the variables it uses are recorded in a data structure known as the ***call stack***. The way a variable is stored depends on whether or not it belongs to a ***primitive type***.

A ***primitive type*** variable is literally a memory location in the call stack that stores the value of that variable. Java has 8 primitive types:

1. `byte`: 8-bit signed two's-complement integers
2. `short`: 16-bit signed two's-complement integers
3. `int`: 32-bit signed two's-complement integers
4. `long`: 64-bit signed two's-complement integers
5. `char`: 16-bit unsigned integers representing Unicode characters.
6. `float`: 32-bit IEEE 754 floating-point numbers
7. `double`: 64-bit IEEE 754 floating-point numbers.

---

[1] http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

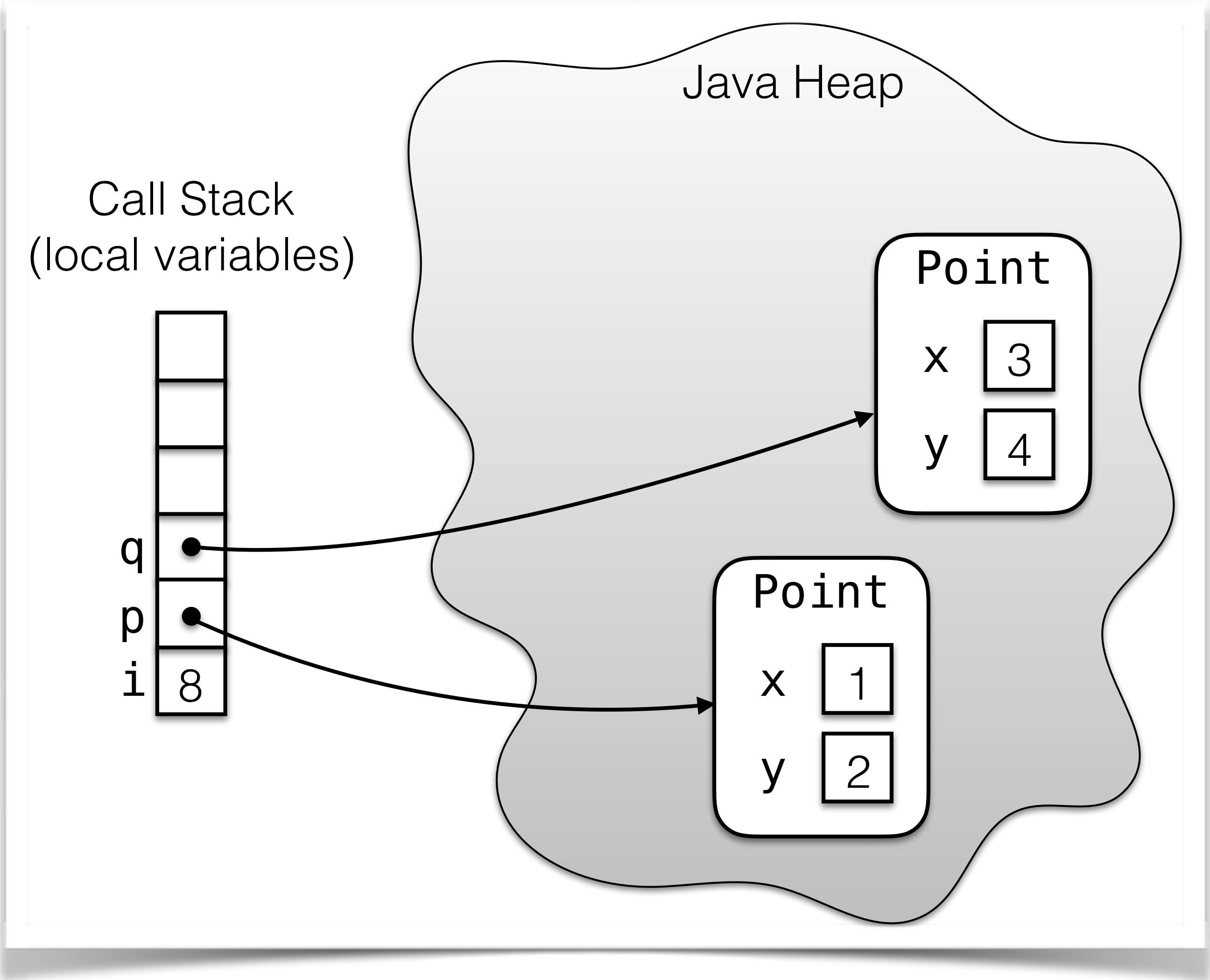
8. `boolean`: has exactly two values: true and false.

Every non-primitive variable is an ***object reference***: the Java stack does not store the actual value or values that make up the object, it only stores a ***reference*** to that object in the ***Java heap***.

Example. To illustrate the difference between the two kinds of variables, consider the `Point` class from a few lectures ago. Suppose we execute the following statements.

```
int i = 8;
Point p = new Point(1, 2);
Point q = new Point(3, 4);
```

Then, `i` is a variable of a primitive type, while p and q are references to two different `Point` objects. See the next figure.



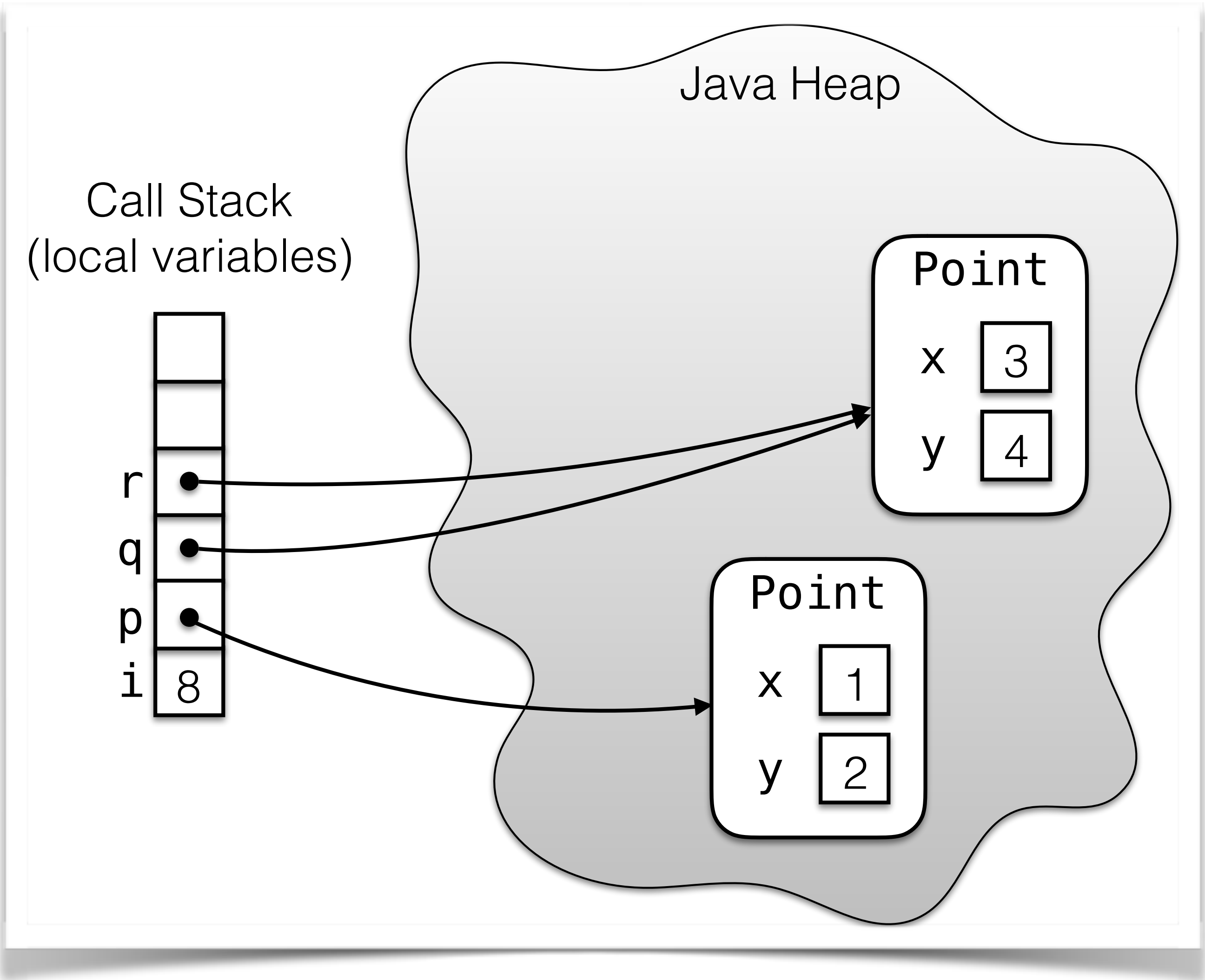
Now suppose we write:

```
Point r = q;
```

By definition of non-primitive variables:

*The object is **not** copied, only the reference!*

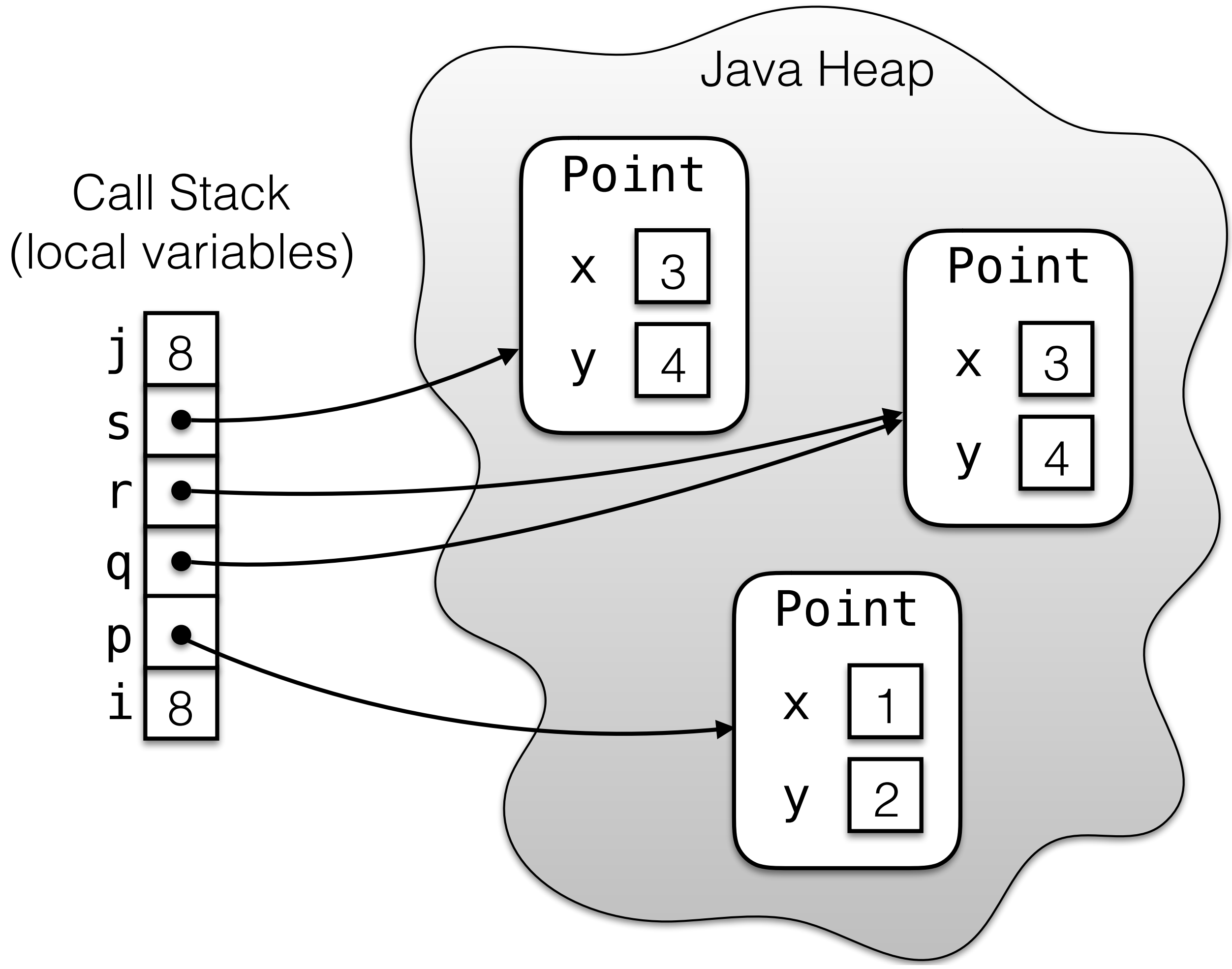
That is, we have two variables referring to the same object. The picture looks like this:



Now, suppose we do the following.

```
Point s = new Point(3, 4);
int j = 8;
```

The result is illustrated next.



Let us examine how the behavior of the == operator for the different kinds of variables.

When we do `x == y`, we are testing whether the contents of the memory locations on the call stack for `x` and `y` have the same value. Therefore,

- `i == j` is `true`, since the contents of i and j are both 8,

- `q == r` is `true`, since `q` and `r` reference the same object,
- `p == q` is false, since `p` and `q` reference different objects, and
- `q == s` is false, since `q` and `s` reference different objects.

The last statement is somewhat counterintuitive, because q and s are, in a real sense, “equal”: they represent two points with exactly the same coordinates. However, == does not capture this notion. Next time, we will see how to override the predefined `equals()` method from the `Object` class so that we can test whether two point objects have the same coordinates.