Objects and Classes

Ye Yang
Stevens Institute of Technology

Acknowledgements

Materials are taken from Jonathan Shewchuk's *Data Structure* course:

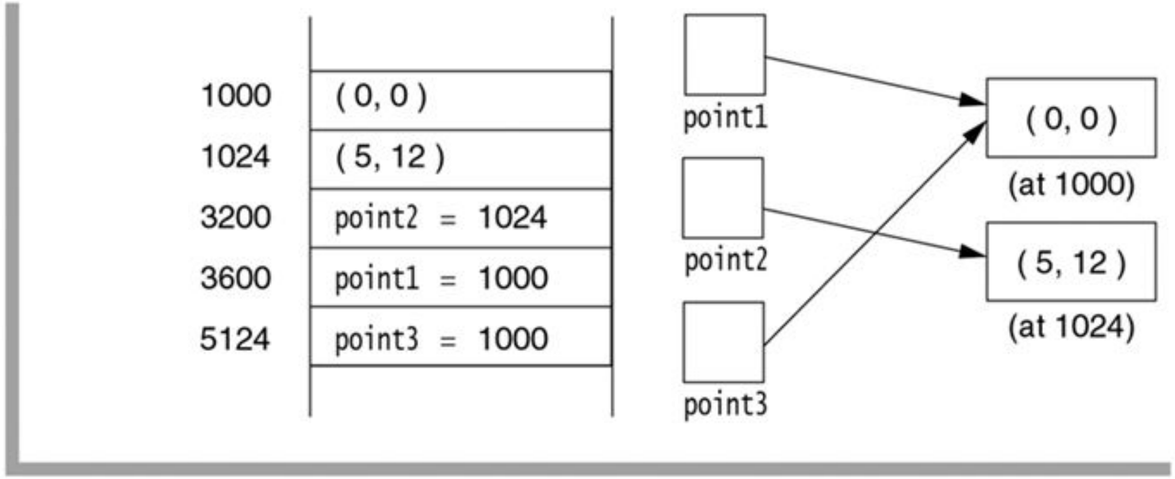
https://people.eecs.berkeley.edu/~jrs/61b/

What is a Reference?

- A reference variable in Java is a variable that somehow stores the memory address where an object resides.

figure 2.1

An illustration of a reference. The Point object stored at memory location 1000 is referenced by both point1 and point3. The Point object stored at memory location 1024 is referenced by point2. The memory locations where the variables are stored are arbitrary.



1000	(0, 0)
1024	(5, 12)
3200	point2 = 1024
3600	point1 = 1000
5124	point3 = 1000

Operations on Reference Variables

- On reference value
  - Examine or manipulate the reference value
    - Change the stored value (i.e. assignment via =)
  - Compare two variables
    - Determine whether referencing the same object (i.e. == and !=)
- On the objects being referenced
  - Examine or change internal state of objects
    - dot operator
- Invalid operations
  - Arithmetic operations on pointer variable is meaningless

OBJECTS AND CONSTRUCTORS

- String s; // Step 1: declare a String variable.
- s = new String(); // Steps 2, 3: construct new empty String; assign it to s.
- **At this point, s is a variable that references an "empty" String, i.e. a String containing zero characters.**

- String s = new String(); // Steps 1, 2, 3 combined.
- s = "Yow!"; // Construct a new String; make s a reference to it.
- String s2 = s; // Copy the reference stored in s into s2.
- **Now s and s2 reference the same object.**

- s2 = new String(s); // Construct a copy of object; store reference in s2.
- **Now they refer to two different, but identical, objects.**

What Java Does When Executing “s2 = new String(s)”?

- It does the following things, in the following order:
  - Java looks inside the variable s to see where it's pointing.
  - Java follows the pointer to the String object.
  - Java reads the characters stored in that String object.
  - Java creates a new String object that stores a copy of those characters.
  - Java stores a reference to the new String object in s2.

Three String Constructors

- new String()
  - constructs an empty string--it's a string, but it contains zero characters.
- "Yow!"
  - constructs a string containing the characters Yow!.
- new String(s)
  - takes a parameter s. Then it makes a copy of the object that s references.

Methods

- s2 = s.toUppercase(); // Create a string like s, but in all upper case.
- String s3 = s2.concat("!!"); // Also written: s3 = s2 + "!!";
- String s4 = "*".concat(s2).concat("*"); // Also written: s4 = "*" + s + "*";

- An important fact:
  - when Java executed the line s2 = s.toUppercase(), the String object "Yow!" did _not_ change.
  - Instead, s2 itself changed to reference a new object.
  - Java wrote a new "pointer" into the variable s2, so now s2 points to a different object than it did before.

Strings are _immutable_

- In Java Strings are _immutable_
  - Once they've been constructed, their contents never change.
  - If you want to change a String object, you've got to create a brand new String object that reflects the changes you want.
  - This is not true of all objects; most Java objects let you change their contents.
- You might find it confusing that methods like "toUppercase" and "concat" return newly created String objects, though they are not constructors.
- The trick is that those methods calls constructors internally, and return the newly constructed Strings.

I/O Classes and Objects in Java

- Here are some objects in the System class for interacting with a user:
  - System.out is a PrintStream object that outputs to the screen.
  - System.in is an InputStream object that reads from the keyboard.
  - Reminder: this is shorthand for "System.in is a variable that references an InputStream object."

Example

- Here's a complete Java program that reads a line from the keyboard and prints it on the screen.

```
import java.io.*;

class SimpleIO {
        public static void main(String[] arg) throws Exception {
                BufferedReader keybd =
                        new BufferedReader(new InputStreamReader(System.in));
                System.out.println(keybd.readLine());
        }
}
```

Classes for Web Access

- Read a line of text from the White House Web page.

```
import java.net.*;
import java.io.*;

class WHWWW {
        public static void main(String[] arg) throws Exception {
                URL u = new URL("http://www.whitehouse.gov/");
                InputStream ins = u.openStream();
                InputStreamReader isr = new InputStreamReader(ins);
                BufferedReader whiteHouse = new BufferedReader(isr);
                System.out.println(whiteHouse.readLine());
        }
}
```

DEFINING CLASSES

```
class Human {
       public int age;                // The Human's age (an integer).
       public String name;            // The Human's name.

        public void introduce() {     // This is a _method_definition_.
                System.out.println("I'm " + name + " and I'm " + age + " years old.");
        }
}
```

Using the class:

```
        Human amanda = new Human();    // Create amanda.
        amanda.age = 6;                // Set amanda's fields.
        amanda.name = "Amanda";
        amanda.introduce();            // _Method_call_ has amanda introduce herself.
```

- The output is: I'm Amanda and I'm 6 years old.

Constructor

```
class Human {     // Include all the stuff from the previous definitions here.
        public Human(String givenName) {
                age = 6;
                name = givenName;
        }
}
```

Now, we can shorten amanda's declaration and initialization to

```
        Human amanda = new Human("Amanda");
        amanda.introduce();
```

JAVA Class Constructors

- Java provides every class with a default constructor, which takes no parameters and does no initializing. Hence, when we wrote

```
Human amanda = new Human();
```

- We created a new, blank Human. If the default constructor were explicitly written, it would look like this:

```
public Human() {
}
```

- Warning: if you write your own Human constructor, even if it takes parameters, the default constructor goes away.

```
class Human {   // Include all the stuff from the previous definitions here.
  public Human() {
        age = 0;
        name = "Untitled";
  }
}
```

Overriddng

The "this" Keyword

- A method invocation, like "amanda.introduce()", implicitly passes an object (in this example, amanda) as a parameter called "this". So we can rewrite our last constructor as follows without changing its meaning.

```
public Human() {
   this.age = 0;
   this.name = "Untitled";
}

public void change(int age) {
   String name = "Tom";
   this.age = age;
   this.name = name;
  }
```

- What happens when we call "amanda.change(11)”?

The "static" Keyword

- A _static_field_ is a single variable shared by a whole class of objects; its value does not vary from object to object.

```
class Human {
  public static int numberOfHumans;

  public int age;
  public String name;

  public Human() {
  numberOfHumans++;    // The constructor increments the number by
                          one.
  }
}
```

The "static" Keyword

- If we want to look at the variable numberOfHumans from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object.

```
int kids = Human.numberOfHumans / 4;  // Good.
int kids = amanda.numberOfHumans / 4; // This works too, but has nothing to
                   // do with amanda specifically.  Don't
                   // do this; it's bad (confusing) style.
```

- System.in and System.out are other examples of static fields.

Static Methods

- Methods can be static too. A _static_method_ doesn't implicitly pass an object as a parameter.

```
class Human {
  ...
  public static void printHumans() {
    System.out.println(numberOfHumans);
  }
}
```

- Now, we can call "Human.printHumans()" from another class. We can also call "amanda.printHumans()", and it works, but it's bad style, and amanda will NOT be passed along as "this".

- The main() method is always static, because when we run a program, we are not passing an object in.

Lifetimes of Variables

- A local variable (declared in a method) is gone forever as soon as the method in which it's declared finishes executing.
  - If it references an object, the object might continue to exist, though.
- An instance variable (non-static field) lasts as long as the object exists.
- An object lasts as long as there's a reference to it.
- A class variable (static field) lasts as long as the program runs.

Comparison

	Object/Reference types	Primitive types
Variable contains a	reference	value
How defined?	class definition	built into Java
How created?	"new"	"6", "3.4", "true"
How initialized?	constructor	default (usually zero)
How used?	methods	operators ("+", "*", etc.)

Object-Oriented Terminology

- In the words of Turing Award winner Nicklaus Wirth,
  - *"Object-oriented programming (OOP) solidly rests on the principles and concepts of traditional procedural programming. OOP has not added a single novel concept ... along with the OOP paradigm came an entirely new terminology with the purpose of mystifying the roots of OOP."*

Procedural Programming	Object-Oriented Programming
record / structure	object
record type	class
extending a type	declaring a subclass
procedure	method
procedure call	sending a message to the method