CS 61B Data Structures and Programming Methodology

June 24 2008

David Sun
Announcements

• Ben’s Office Hours:
  – Monday 4-5pm, Tuesday 2-3pm in Soda 611.

• Need a thorough set of notes for this class
  – $160 for the entire semester.
  – Quality and speed counts.
  – If interested, email me with your notes for today and yesterday @ davidsun@cs.berkeley.edu
From last time...

```java
public class FirstApp {
    public static void main(String[] args) {
        String[] greeting = new String[3];
        greeting[0] = "Welcome to CS 61b";
        greeting[1] = "from David, Ben, Adam";
        greeting[2] = "and George";

        for (String g : greeting) {
            System.out.println(g);
        }
    }
}
```

Class with a single **main** method
Today

• Object oriented programming
• Defining Classes
• Constructors
• Access Privileges
Object Oriented Programming
Classes and Objects

• A class is a template or blueprint from which you construct objects.

• A constructed object is an instance of a class
• The standard Java library supplies thousands of classes.
• You still need to create your own to describe the objects of the problem domain of your application.
Encapsulation (data hiding)

- Combining data and behavior in one package.
- Data = *Instance fields/instance variables*
- Behavior = *Methods*
- An object’s current values for the instance variables = *State of the object*
- Programs only interact with object through the object’s *methods*, never directly through the object’s *instance variables*.

= reuse and reliability
Objects

• **Key characteristics:**
  – Object’s Behavior: what can you do with the object, what methods can you call?
  – Object’s State: how does the object react to methods applied to it? What does it look like?
  – Object’s Identity: how to distinguish the object from others with same behavior and state?

• All objects that are instances of the same class support the *same* behavior.

• Each object stores information about its current state.

• Identity issue covered future lectures
Defining Classes
A Minimal Class

class Minimalist
{

}

class MinimalistTest
{
    public static void main(String[] args) {
        Minimalist mini = new Minimalist();
    }
}
A Human Class

Age

Name

“I’m name and I’m age years old!”
class Human
{
    public int age;
    public String name;
}

class HumanTest
{
    public static void main(String[] args) {
        Human wendy = new Human();
        wendy.age = 12;
        wendy.name = "Wendy";
        System.out.println("I’m “ + wendy.name + “ and I’m “ + wendy.age + “ years old.");
    }
}

What about encapsulation??
A (Better) Class Definition

class Human
{
    private int age;
    private String name;

    public Human(int a, String n) {
        age = a;  // Write code to initialize the
        name = n; // object.
    }

    public void introduce() {
        System.out.println("I’m “ + name + “ and I’m “ + age + “ years old.");
    }
}
class HumanTest {
    public static void main(String[] args) {
        Human wendy = new Human(12, "Wendy");
        wendy.introduce();  //"I’am Wendy and I’m 12 years old"
    }
}
public v.s private

class Human
{
  public int age;
  public String name;
}

class HumanTest{
  public static void main(String[] args){
      ...
      wendy.age = 12;
      ...
  }
}

class Human
{
  private int age;
  private String name;
  ...
}

private: only methods that can access the field are methods defined within the class.

public void introduce() {
    System.out.println("I’m “ + name + “ and I’m “ + age + “ years old.");
}

any method in any class has access.
public Fields

• You *could* use `public` keyword with fields
• Bad idea
  – Any part of the program can read and modify the instance field.
  – Ruins encapsulation.
Constructors
public Human(int a, String n) {
    age = a;
    name = n;
}

• Constructor has the same name as the class.
• Constructor can only be called in conjunction with the **new** operator.
  – wendy.Human(21, "wendy"); //error

• A class can have more than one constructor
• A constructor can take zero or more parameters
• A constructor has no return value.
class Human
{
    public int age;
    public String name;
}

class HumanTest
{
    public static void main(String[] args) {
        Human wendy = new Human();
        wendy.age = 12;
        wendy.name = "Wendy";
        System.out.println("I’m " + wendy.name + " and I’m " + wendy.age + " years old.");
    }
}
Default Constructor

• If a class is not defined with a constructor, Java provides a default constructor
  — Takes no parameter.

• A default constructor can be explicitly defined:
  ```java
  public Human() {
  }
  ```

• If you defined *any* constructor, the default constructor provided by Java goes away.
  — To have both the default and other constructors, you must define both explicitly.

• Overriding a default constructor:
  ```java
  public Human() {
    age = 0;
    name = "no name";
  }
  ```
Field Initialization

• The fields are initialized to their default values in the default constructor:
  age = 0
  name = null;

• It’s a good idea to always set the values of the fields to some default value:
  class Human
  {
      private int age = -1;
      private String name = "";
      ...
  }
Parameter Names

```java
public Human(int a, String n) {
    age = a;
    name = n;
}

public Human(int age, String name) {
    age = age;
    name = name;
}

public Human(int aAge, String aName) {
    age = aAge;
    name = aName;
}
```

Parameter variables shadow instance fields. Must read the program to decipher the meaning of the parameters.
class Human
{
    ...
    public void introduce()
    {
        System.out.println("I’m " + name + " and I’m " + age + " years old.").
    }
}

class HumanTest
{
    public static void main(String[] args)
    {
        Human wendy = new Human(12, "Wendy");
        wendy.introduce(); //"I’am Wendy and I’m 12 years old"
    }
}

• **An implicit** parameter precedes the field instance.
• The implicit parameter can be *explicitly* referred to by the keyword **this**.

System.out.println("I’m " + this.name + " and I’m " + this.age + " years old.").
Revisit Parameter Names

```java
public Human(int a, String n) {
    age = a;
    name = n;
}

public Human(int aAge, String aName) {
    age = aAge;
    name = aName;
}

public Human(int age, String name) {
    this.age = age;
    this.name = name;
}
```
More on this

• **this** refers to an implicit parameter.
• **this** has a second meaning if it’s the first statement of a constructor...

```java
class Human {
    int height;

    public Human(int age, String name, int height) {
        this(age, name);
        this.height = height;
    }
}
```

• You write a common constructor only once.
Construction Process

• All the fields are initialized to default value.
• All field initializers are executed.
• If the first line of the constructor calls a second constructor, then the second constructor is executed.
• The body of the constructor is executed.
Access Privileges
Class-Based Access Privileges

• Method can access private data of the object on which it is invoked.

• Method can access the private data of all objects of its class:

```java
Class Human {
    ...

    boolean equals(Human other) {
        return name.equals(other.name);
    }
}
```
Mutators and Accessor Methods

- Private fields are not directly accessible, how can we access and/or change their values:

```java
class Human {
    private int age;
    private String name;
    ...
    public int getAge() {
        return age;
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

- Not all fields need mutators.
private methods

• public data fields are dangerous, what about methods?
• Most methods are public, but helper methods can be made private.
• private methods are not accessible by other classes
static Fields

- If a field is declared `static`, then there is only one such field per class.
- Think of it as a global state shared by all instances of the class.
- For example, need a counter to keep track of the number of humans created:

```java
class Human {
    private int age = -1;
    private String name = ""
    public static int numberOfHumans = 0;

    public Human(int age, String name) {
        this.age = age;
        this.name = name;
        numberOfHumans++;
    }
}
```

- Another example: `System.out.println()`
static methods

- **static methods do not operate on an object:**
- **Has no implicit parameter**
- **Cannot access instance fields from a static method.**
- **Static methods can access static field**

```java
class Human {
    public static int getNumOfHumans() {
        return numOfHumans;
    }
}
```

- **Useful when:**
  - A method doesn’t need to access the object state because all needed parameters are supplied as explicit parameters
    - Math.pow(a, b)
  - When a method only needs to access static fields.
Life Cycle of a Variable

• Local variable is gone as soon as the method in which it is declared finishes executing.
• 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 run.
Next Class

• Reading:
  – Chapters 3, 4, and 9 (except pages 250-255) in *Head First Java*
  – "Boxes and arrows" (available on UCWise)

• Next Class:
  – Primitive Types
  – Arrays and loops.