Object-Oriented Programming

Sara Sprenkle

Questions?

- Review…
  - Java is …?
  - Assignment 0 due on Tuesday

Answers to your questions

- Different Java Editions
  - SE: Standard Edition
  - EE: Enterprise Edition
    - Server-side applications
    - Web applications, Communication (mail)
  - ME: Micro Edition
    - For PDAs, mobile devices, etc.
- `goto` statement
  - Unimplemented keyword
  - Variables can't be called `goto`

Your Surveys

- JavaScript vs. Java
  - JavaScript is *not* Java
- Web application programming
  - This class: briefly on client-side (Applets) and server-side programming (Servlets, JSPs)
  - Good starting point to learn more on your own
  - CISC474
- C# *wishes* it were Java

Course Topics

- Java Fundamentals
- APIs
  - GUIs
  - Threads
  - XML
  - Network programming (RMI)
  - Applets
  - Web applications: Servlets, JSPs

A Little More Arrays

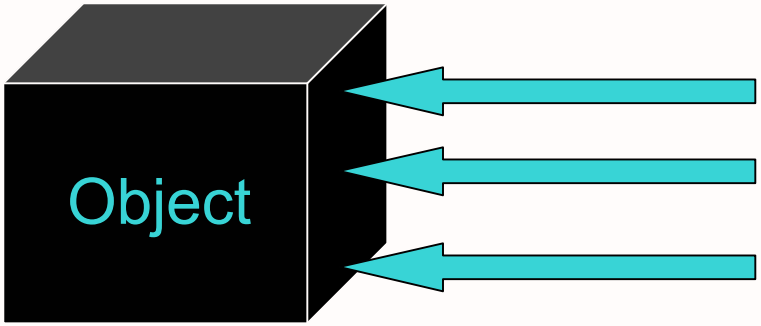
- `Arrays` is a class in java.util
- Methods for sorting, searching, “deepEquals”, fill arrays
  - Wouldn’t these methods have been useful in 105 or 181?

Object-Oriented Programming

- Programming that models real life
  - Consider an ATM…
    - Implicitly agreed upon interface between user and the calculator
    - What, not how

Objects

- How object does something doesn't matter
- **What** object does matters (its **functionality**)
  - What object *exposes* to other objects
  - Referred to as "black-box programming"



- Has public interface that others can use
- Hides state from others

Objects: Black-box programming

- If an object allows you to access and store data, you don't care if the underlying data type is an array or hashtable, etc.
  - It just has to ***work***!
  - And, *you* don't need to implement a new object to do the job.
- Similarly, if object *sorts*, does not matter if uses merge or quick sort

Classes & Objects

- Classes define template from which objects are made
  - "cookie cutters"
  - Define **state -** data, usually private and **behavior** - *methods* for an object, usually public
- Many objects can be created for a class
  - Object: the cookie!
  - E.g., many Mustangs created from Ford's "blueprint"
  - Object, a.k.a. an instance of the class

Classes & Objects

- Java is pure object-oriented programming
  - all data and methods in a program must be contained within an object/class

Classes, Objects, Methods

- Classes define template from which objects are made.
  - **State -** data, usually private
  - **Behavior** - *methods* for an object, usually public
- Many objects can be created for a class
  - E.g., many Mustangs created from Ford's "blueprint"
- **Method**: sequence of instructions that access/modify an object's data
  - **Accessor**: accesses (doesn't modify) object
  - **Mutator**: changes object's data

Encapsulation Terminology

- **Encapsulation** is the combining of data and behavior (functionality) into one package (the object) and hiding the implementation of the data from the user of the object.
- **instance fields:** the object's data/variables
- **methods**: the functions & procedures which operate on the object's data
- The set of data contained in an object's instance fields is the current **state** of the object.

Example: `Chicken` class

- State
  - Weight, height, name
- Behavior
  - Accessor methods
    - getWeight, getHeight, getName
    - Convention: “get” for “getter” methods
  - Mutator methods
    - eat: adds weight
    - sick: lose weight
    - changeName

Designing the `Pizza` Class

- State
- Behavior

Constructors

- **Constructor:** a special method which constructs and initializes an object
  - After construction, you can call methods on an object
- Constructors have the same name as their classes
- To create an object of a certain type (a certain class), use the **new** operator, much like in C++

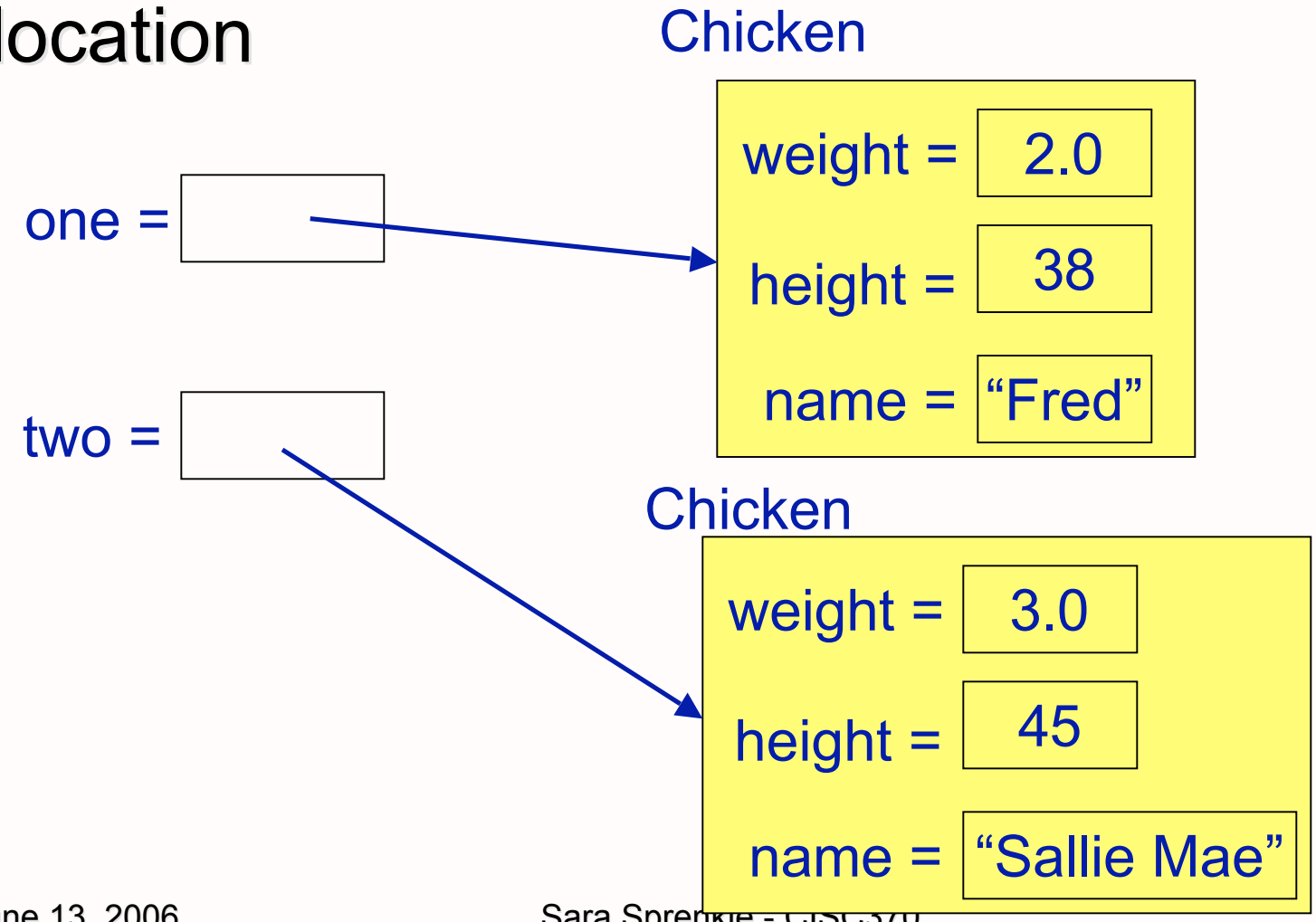
Constructing objects using new

- Create three chickens
  - “Fred”, weight: 2.0, height: 38
  - “Sallie Mae”, weight: 3.0, height: 45
  - “Momma”, weight: 6.0, height: 83
- Chicken **constructor**
  - Chicken(String name, float weight, float height)
- Create new objects (Chickens)
  - Chicken one = new Chicken(“Fred”, 2.0, 38);
  - Chicken two = new Chicken(“Sallie Mae”, 3.0, 45);

And Momma? …

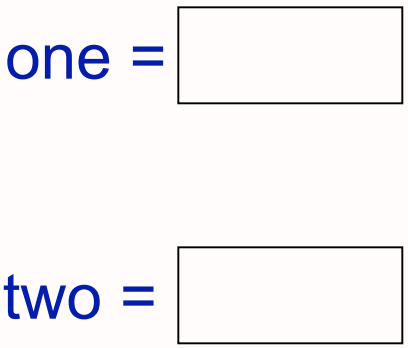
Object References

- Variable of type object: value is memory location



Object References

- Variable of type object: value is memory location



If I haven't called the constructor, only

```
Chicken one;
Chicken two;
```

Both `one` and `two` are equal to `null`

Null Object Variables

- An object variable can be explicitly set to null.
  - indicates that this object variable does not currently refer to any object.
- It is possible to test if an object variable is set to null

```
Chicken chick = null;
    … … …
if (chick == null)
{
   . . .
}
```

Designing the `Pizza` Class

- Constructors?

Relationship to C++

- Java object variables as similar to object pointers in C++.
- For example,

```
String myString;      // Java
```

  is really the same as:

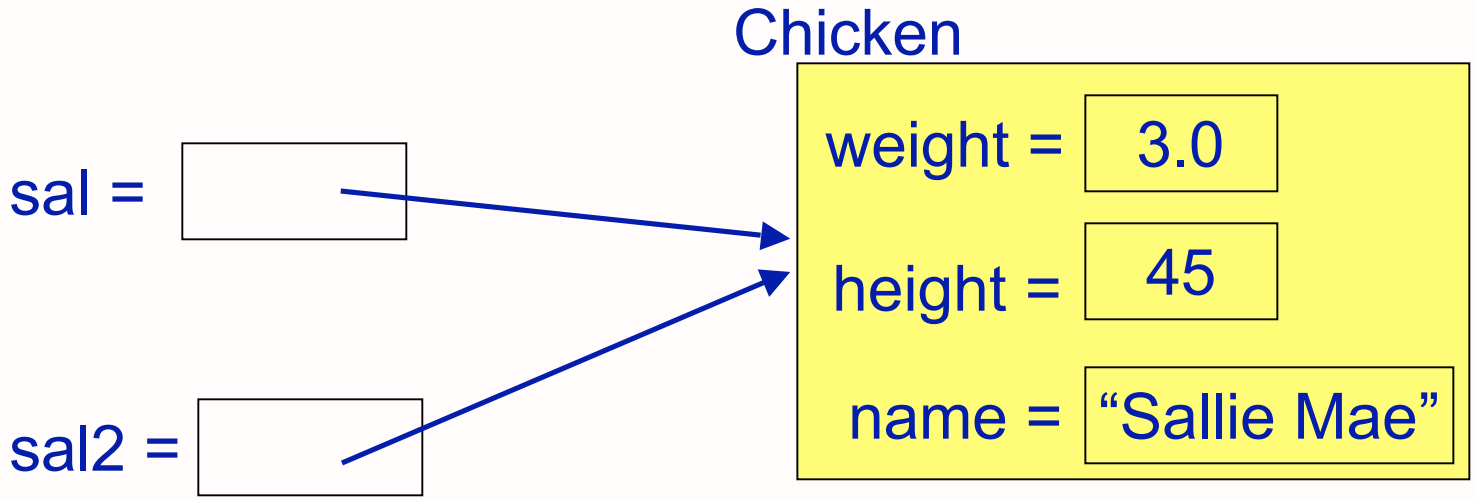
```
String * my_string;   // C++
```

- Thus, we can see that every object variable in Java is like a pointer to an object in C++.

Multiple Object Variables

- more than one object variable can refer to the same object

```
Chicken sal = new Chicken("Sallie Mae");
Chicken sal2 = sal;
```



What happens here?

```
Chicken x, y;
Chicken z = new Chicken("baby", 1.0,
  5);
x = new Chicken("ed", 10.3, 81);
y = new Chicken("mo", 6.2, 63);
Chicken temp = x;
x = y;
y = temp;
z = x;
```

Encapsulation Revisited

- Objects should hide their data and only allow other objects to access this data through **accessor** and **mutator** methods.
- Common programmer mistake:
  - creating an accessor method that returns a reference to a mutable (changeable) object.

What is "bad" about this class?

```
class Farm
{
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster()
      {
            return headRooster;
      }
      . . .
}
```

Fixing the Problem: Cloning

```
class Farm
{
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster()
      {
            return (Chicken) headRooster.clone();
      }
      . . .
}
```

Another Chicken object, with the same data as headRooster is created and returned to the user.

If the user modifies (e.g., feeds) that object, it does not affect headRooster.

Cloning

- Cloning is a more complicated topic than it seems from the example.
- We will examine cloning in more detail at a later point.

Access Modifiers

- A **public** method (or instance field) means that any object *of any class* can directly access the method (or field)
  - Least restrictive
- A **private** method (or instance field) means that any object *of the same class* can directly access this method (or field)
  - Most restrictive
- There are also other access modifiers, which will be discussed with Inheritance

Method Parameters in C

- **Call-by-value** – a copy of the parameter is passed into the function/method.
- **Call-by-reference** – the memory location of the parameter is passed into the function/method.

Method Parameters in Java

- Java always passes parameters into methods **call-by-value**.
  - methods cannot change the variables used as input parameters.

What's the output?

```
int x;
x = 27;
System.out.println(x);
doubleValue(x);
System.out.println(x);

. . .

void doubleValue(int p)
{
     p = p * 2;
}
```

What's the output?

```
Farm farm = new Farm("OldMac");
Chicken sal = new Chicken("Sallie Mae", 50, 10);
System.out.println(sal.getWeight());
farm.feedChicken(sal);
System.out.println(sal.getWeight());
. . .
void feedChicken(Chicken c)
{
     c.setWeight( c.getWeight() + .5);
}
```

What's the difference?

- `sal` is an object variable, not an object
  - *refers* to an object (think of it as being a pointer to an object of the `Chicken` class).
- a copy of the object variable (the reference) is passed into the method.
  - a copy of a reference refers to the same object!
  - we *can* modify the state of an object in this manner

Summary of Method Parameters

- Everything is passed **call-by-value** in Java.
- An **object variable** (not an object) is passed into a method
  - changing the state of an object in a method changes the state of that object outside the method
  - the method does not see a copy of the original object

Varargs

- New in Java 1.5
- A method can take a variable number of arguments

```
void varargs_method(String ... args)
{
   for (int i=0;i < args.length; i++)
      System.out.println("arg "+ i +
        " is " + args[i]);
}
```

Why would you want to do this?

Hint: Java now has a System.out.printf function

Constructor Conventions

- A *constructor* is a special method that defines an object's initial state
- Always has the same name as the class
- A class can have more than one constructor
- A constructor can have zero, one, or multiple parameters
- A constructor has no return value
- A constructor is always called with the `new` operator.

Constructor Overloading

- Allowing more than one constructor (or any method with the same name) is called **overloading**.
  - each of the methods that have the same name must have different parameters
- *Overload resolution* is the process of matching a method call to the correct method by matching the parameters
  - handled by the compiler

Default Initialization

- Unlike C and C++, if an instance field is not set explicitly in a constructor, it is automatically set to a default value…
  - Numbers are automatically set to zero.
  - Booleans are automatically set to false.
  - Object variables are automatically set to null.
- It is a bad idea to rely on defaults
  - Code is harder to understand
  - **Set all instance fields in the constructor(s).**

Default Constructor

- A *default constructor* is a constructor that has no parameters.
- If no constructors are present in a class, a default constructor is provided by the compiler
  - default sets all instance fields to their default values
- If a class supplies at least one constructor but no default constructor, the default constructor is NOT provided

Default Constructor

- Chicken class has only one constructor:

```
Chicken(String name, float weight, float height)
```

- So, there is no default constructor

```
Chicken chicken = new Chicken();
```

  - Is a syntax error

Explicit Field Initialization

- If more than one constructor needs to set a certain instance field to the same value, the initial state of the field can be set explicitly, in the field declaration

```
class Chicken {
      private String name = “”;
      . . .
}
```

Explicit Field Initialization

- Or in a static method call

```
class Employee
{
      private int id = assignID();
      . . .
      private static int assignID()
      {
            int r = nextID;
            nextID++;
            return r;
      }
}
```

More on static later…

Explicit Field Initialization

- The ordering of explicit field initialization with relation to the constructor is important.
- Explicit field initialization happens before any constructor runs.
- if one constructor wants to change an instance field that was set explicitly, it can.
- If the *constructor* does not set the field explicitly, explicit field initialization is used

`final` keyword

- An instance field can be made *final*.
- If an instance field is `final`, it must be set in the constructor or in the field declaration and cannot be changed after the object is constructed

```
private final String id;
```

Constructors calling constructors

- It is possible to call a constructor from inside another constructor.
- If the first statement of a constructor has the form

```
this( . . . );
```

  the constructor calls another constructor of the same class.
  - `this` keyword works just like in C++; an *implicit parameter*
  - refers to the object being constructed

Constructors calling constructors

- Why would you want to call another constructor?
  - Reduce code size/reduce duplicate code
- Ex: if name not provided, use default name

```
Chicken( float height, float weight ) {
      this( "Bubba", height, weight);
}
```

- Example: base case constructor

```
Chicken( float height, float weight ) {
      this();
      this.height = height;
      this.weight = weight;
}
```

Object Destructors

- In C++ (and many other OOP languages), classes have explicit destructor methods that run when an object is no longer used.
- Java does not support destructors, as it provides *automatic garbage collection*
  - Watches/waits until there are no references to an object
  - Reclaims the memory allocated for the object that is no longer used

`finalize()`

- Java supports a method named *finalize()*.
- method is called before the garbage collector sweeps away the object and reclaims the memory
- This method should not be used for reclaiming any resources
  - the timing when this method is called is not deterministic or consistent
  - only know it will run sometime before garbage collection

Using `finalize()`

- What do we do in a finalize() method?
- We clean up anything that cannot be atomically cleaned up by the garbage collector
  - Close file handles
  - Close network connections
  - Close database connections
  - etc…

Static Methods/Fields

- For related functionality/data that isn't specific to any particular object
- java.lang.Math
  - No constructor (what does that mean?)
  - Static fields: PI, E
  - Static methods:
    - `static double ceil(double a)`

Static Methods

- Do not operate on objects
- Cannot access instance fields of their class.
- Can access *static fields* of their class.

Static Fields

- A static field implies that there is only one such field per class (not object!).
- All objects of a class with a static field share **one copy** of that field.
- For example, if we wanted to have a unique studentID for a Student class…

```
class Student {
      private int id;
      private static int nextID = 1;
      . . .
}
```

Static Fields

```
class Student {
  private int id;
  private static int nextID = 1;
  . . .
}
```

- Each Student object has an `id` field, but there is only one `nextID` field, shared among all instances of the class
  - `nextID` field is present even when no Students have been constructed.

How would we use the nextID field to create unique IDs?

Constant Static Fields

- We can also use a static field to make a constant in a class…

```
public class Math {
   . . .
   public static final
     double PI = 3.14..;
}
```

- The Math class has a static constant, PI
  - The value can be accessed using the Math class:

```
a = b * Math.PI;
```

- Notice we do not need to create an object of the Math class to use this constant.

`main()`

- The most popular static method we have seen so far is the `main()` method.
- The `main()` method does not operate on any objects
  - It runs when a program starts…there are no objects yet.
- `main()` executes and constructs the objects the program needs and will use.

Analyzing java.lang.String

- Look at Java Docs for Constructors
- `String toUpperCase()`
  - Converts all of the characters in this String to upper case
- `static String valueOf(boolean b)`
  - Returns the string representation of the boolean argument.

What methods/fields should be static for the `Pizza` class?

Static Summary

- Static fields and methods are part of a class and not an object
  - do not require an object of their class to be created in order to use them.
- When would we use a static method?
  - When a method does not have to access an object state (fields) because all needed data are passed into the method.
  - When a method only needs to access static fields in the class

Explicit Field Initialization

- Or in a static method call

```
class Employee
{
      private int id = assignID();
      . . .
      private static int assignID()
      {
            int r = nextID;
            nextID++;
            return r;
      }
}
```

More on static later…

Class Design/Organization

- Fields
  - Chosen first
  - Placed at the beginning or end of the class
  - Has an accessor modifier, data type, variable name and some optional other modifiers
    - If no accessor modifer --> public
  - Use `this` keyword to access the object
- Constructors
- Methods
  - Maybe `public static void main`