Objectives

- Object-oriented programming in Java
  - Constructors
  - Default constructors
  - Garbage collection
  - Static methods, variables
  - Inherited methods

Danger of Semicolons

- What does this code do?

```
if ( x > 4 ) ;
     System.out.println("x is " + x);
```


Danger of Semicolons

- What does this code do?

```
if ( x > 4 ) ;
     System.out.println("x is " + x);
```

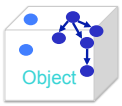
- ; is a valid statement
- Print statement *always* executes
- Indentation doesn't matter

Review

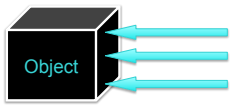
- Why OO programming?
  - What are its components?
- What's wrong with "white-box" programming?
- What is the syntax for defining a method?
- What is the syntax for defining a constructor?

Review: 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**"



- Can see and manipulate object's internals



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

Review: General Java Class Structure

```
public class ClassName {

   // --------- INSTANCE VARIABLES ---------------
   // define variables that represent object's state
   private int inst_var;

   // --------- CONSTRUCTORS ---------------
   public ClassName() {
      // initialize data structures
   }

   // ----------- METHODS ------------
   public int getInfo() {
      return inst_var;
   }
}
```

Note: instance variables are `private` and methods are `public`

More on Constructors

- A class can have **more than one** constructor
  - Whoa! Let that sink in for a bit
- A constructor can have zero, one, or multiple parameters
- A constructor has no return value
- A constructor is always called with the new operator

Overloading

- Allowing > 1 constructor (or any method) with the same name is called **overloading**
  - Constraint: Each of the methods that have the same name must have different parameters
    - "different" → *Number* and/or *type*
- Compiler handles **overload resolution**
  - Process of matching a method call to the correct method by matching the parameters
- No function overloading in Python
  - Why wasn't that possible?

Default Initialization

- If instance field is not explicitly set in constructor, automatically set to default value
  - Numbers are set to zero
  - Booleans are set to `false`
  - Object variables are set to `null`
  - Local variables are not assigned defaults
- **Do not** rely on defaults
  - Code is harder to understand
  - **Set all instance fields in the constructor(s)**

Explicit Field Initialization

- If more than one constructor needs an instance field set to same value, 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

- Explicit field initialization happens before any constructor runs
- A constructor can change an instance field that was set explicitly
- If the constructor does not set the field explicitly, explicit field initialization is used

final keyword

- An instance field can be `final`
- `final` instance fields **must** be set in the constructor or in the field declaration
  - Cannot be changed after object is constructed

```
private final String dbname = "invoices";
private final String id;
…
public MyObject( String id ) {
      this.id = id;
}
```


Default Constructor

- **Default constructor:** constructor with no parameters
- If class has *no constructors*, compiler provides a default constructor
  - Sets all instance fields to their default values
- However, if a class has at least one constructor and no default constructor, the default constructor is NOT provided

Default Constructor

- Chicken class has one constructor:

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

➡ No default constructor

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

- Is a compiler error

Constructors Calling Constructors

- Can call a constructor from inside another constructor
- The **first** statement of constructor must be

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

to call another constructor of the same class
  - `this` refers to the object being constructed

Constructors Calling Constructors

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

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

- Example: base case constructor

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

Not in example code online

Parent Class: `Object`

- Every new class you create automatically inherits from the `Object` class
  - See Java API
- Useful methods to customize your class
  - `String toString()`
    - Returns a string representation of the object
    - Like Python's `__str__`
  - `boolean equals(Object o)`
    - Return `true` iff this object and o are equivalent
    - Like Python's `__eq__ or __cmp__`
  - `void finalize()`
    - Called when object is destroyed
    - Clean up resources

Method signature

More on `toString()`

- Automatically called when object is passed to print methods
- Default implementation: Class name followed by @ followed by unsigned hexidecimal representation of hashcode
  - Example: `Chicken@163b91`
- General contract: "A concise but informative representation that is easy for a person to read"
- Document the format

Examples: `Chicken.java`

- What would be a good String representation of a Chicken object?
  - Look at output before and after `toString` method implemented
- How would we know if two Chickens are equal?

GARBAGE COLLECTION

Memory Management

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

Garbage Collector

- Garbage collector is low-priority thread
  - Or runs when available memory gets tight
- Before GC can clean up an object, the object may have opened resources
  - Ex: generated temp files or open network connections that should be deleted/closed first
- GC calls object's `finalize()` method
  - Object's chance to clean up resources

Discussion: Benefits and costs of garbage collection?

Garbage Collection

Benefits

- Fewer memory leaks
  - Less buggy code
  - But, memory leaks are still possible
- Code is easier to write

Costs

- Garbage collection may not be as efficient as explicit freeing memory

finalize()

- Called before garbage collector sweeps away the object and reclaims the memory
- Should not be used for reclaiming resources
  - i.e., *close resources as soon as possible*
  - Why?
    - *When* method is called is not deterministic or consistent
    - Only know it will run sometime before garbage collection
- Clean up anything that cannot be atomically cleaned up by the garbage collector
  - Close file handles, network connections, database connections, etc.
- Note: no finalizer chaining
  - Must explicitly call parent object's `finalize` method

Alternatives to finalize

- Recall: unknown when `finalize` will execute—or *if* it will execute
  - Also *heavy performance cost*
- Solution: create your own terminating method
  - User of class terminates when done using object
- Examples: `File`'s or `Window`'s `close` method
- May still want a finalize method as a safety net if user didn't call the terminate method
  - Log a warning message so user knows error in code

Can You Spot the Memory Leak?

- Handout: `MemoryLeak.java`
  - An implementation of a `Stack`
- Try drawing a picture of typical uses of the class

Eliminating the Memory Leak

- In pop()

```
elements[size] = null;
```

- Added benefit: if dereferenced later by mistake, program will fail immediately with `NullPointerException`
  - Detect error as quickly as possible→Good!
- BUT, don't overcompensate by nulling out references all the time
  - Often unnecessary; let variable fall out of scope

STATIC METHODS AND FIELDS

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 sin(double a)`

Static Methods

- Do not operate on objects
- Cannot access instance fields of their class
- Can access *static fields* of their class
- Similar to Python *functions* that are associated with the class

Static Fields

- A static field is used when only one such field **per class** (not object!)
- All objects of a class share **one copy** of the static field

Static Fields Example

```
public 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 exists even when no `Student` objects have been constructed

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

Static Field Example

- One option:

```
public class Student {
      private static int nextID = 1;
      private int id = assignID();

      private static int assignID() {
            int r = nextID;
            nextID++;
            return r;
      }
…
```


Constant Static Fields

- We used a static field to designate a ***class constant***:

```
public class Converter {
      public static final double CM2IN = 2.54;
```

- The `Math` class has a static constant, `PI`
  - Value can be accessed using the `Math` class:
    `area = Math.PI * r * r;`
- Notice we do not need to create an object of the `Math` class to use this constant
  - What is another benefit of a class constant?

`main()`

- Most common static method we have seen
- `main()` does not operate on any objects
  - Runs when a program starts…there are no objects yet
- `main()` executes and constructs the objects the program needs and will use
  - Like the *driver function* for the program

Analyzing java.lang.String

- `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
- Discussion: Why can the second method be `static`?

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 make a method `static`?
  - When a method does not have to access an object's state (fields) because all needed data are passed into the method
  - When a method only needs to access static fields in the class