NC STATE UNIVERSITY

OO Programming and Composition

Zybooks: Chapters 1-10

1

NC STATE UNIVERSITY

Object-Oriented Programming

- Reasoning about a program as a set of objects rather than as a set of actions
- ***Object***: A programming entity that contains state and behavior
  - ***State***: internal data of an object (fields or instance variables)
    - What an object knows
  - ***Behavior***: set of actions performed by the object (instance [non-static] methods)
    - What an object can do
- ***Class***: a blue print for an Object
  - Defines how we construct and work with objects

2

NC STATE UNIVERSITY

Constructors

- Classes can have multiple constructors
  - Different signatures – number and type(s) of parameters
- Default constructor (no parameters) automatically available if you have no defined constructors
- After you define one constructor, you have to write a default constructor

- Programming Paradigm – *One-Path of Construction*: one constructor contains initialization code – all other constructors call it
  - Every object goes through a common code path

NC STATE UNIVERSITY

Object Construction

```
Recipe r = new Recipe("Chili", 4); //name, number of servings
```

1. Creates a `Recipe` reference, `r`
2. Creates a `Recipe` object
3. Calls one of the possibly many `Recipe` constructors and passes any parameters
4. Assigns the newly created object to be stored in reference variable `r`

NC STATE UNIVERSITY

Object Construction

```
Recipe r = new
  Recipe("Chili", 4);
```

1. Creates a `Recipe` reference, `r`
2. Creates a `Recipe` object
3. Calls one of the possibly many `Recipe` constructors and passes any parameters
4. Assigns the newly created object to be stored in reference variable `r`



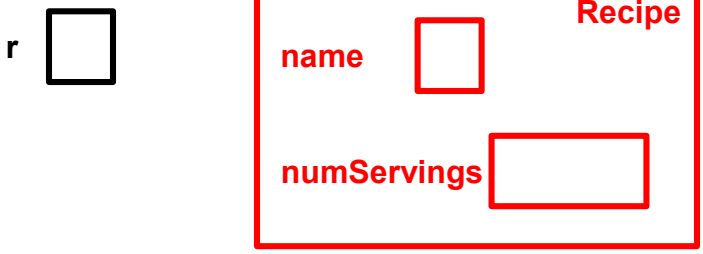
5

NC STATE UNIVERSITY

Object Construction

```
Recipe r = new
  Recipe("Chili", 4);
```

1. Creates a `Recipe` reference, `r`
2. Creates a `Recipe` object
3. Calls one of the possibly many `Recipe` constructors and passes any parameters
4. Assigns the newly created object to be stored in reference variable `r`



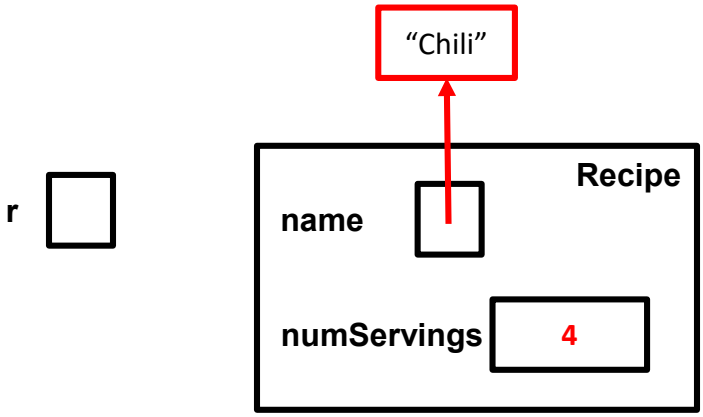
6

NC STATE UNIVERSITY

Object Construction

```
Recipe r = new
  Recipe("Chili", 4);
```

1. Creates a `Recipe` reference, `r`
2. Creates a `Recipe` object
3. Calls one of the possibly many `Recipe` constructors and passes any parameters
4. Assigns the newly created object to be stored in reference variable `r`



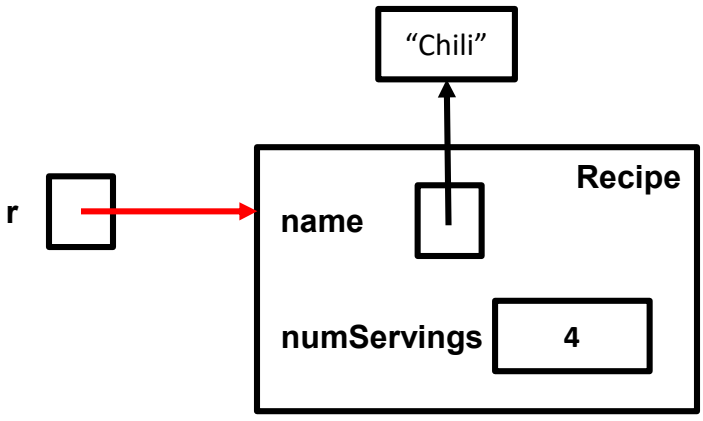
7

NC STATE UNIVERSITY

Object Construction

```
Recipe r = new
  Recipe("Chili", 4);
```

1. Creates a `Recipe` reference, `r`
2. Creates a `Recipe` object
3. Calls one of the possibly many `Recipe` constructors and passes any parameters
4. Assigns the newly created object to be stored in reference variable `r`

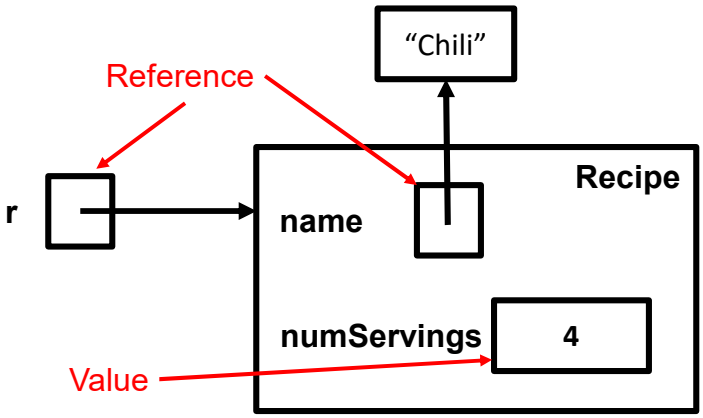


8

Reference Semantics

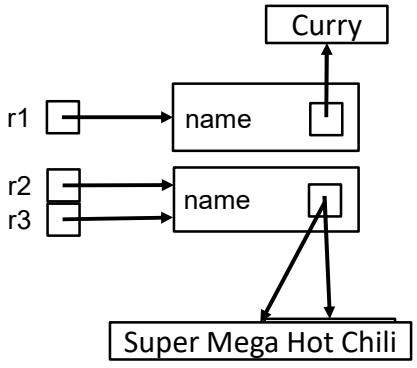
- **Reference Semantics:** behavior where variables refer to a common value when assigned to each other or when passed as a parameter
  - Object *variables* don't store the object directly; instead they store the object's *location in memory*

- Objects in Java use *reference semantics*
  - *Efficiency:* copying large objects takes time
  - *Sharing:* can share information when calling methods
- Primitives use *value semantics*



Reference Semantics

```
public class Recipe {
   private String name;
   public void updateName(String name) {
      if (name != null && name.length() != 0)
         this.name = name;
   }
   public static void main(String [] args) {
      Recipe r1 = new Recipe("Curry");
      Recipe r2 = new Recipe("Chili");
      Recipe r3 = r2;
      r3.updateName("Super Mega Hot Chili!");
   }
}
```



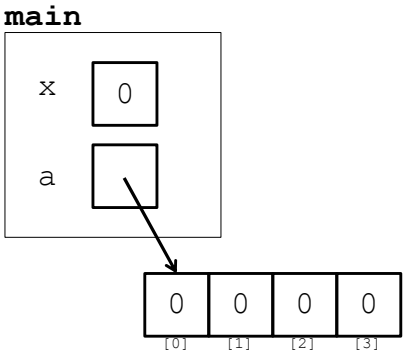
NC STATE UNIVERSITY

Reference Semantics Exercise

Current Line

```
int x = 0;
int [] a = new int [4];
```

output



11

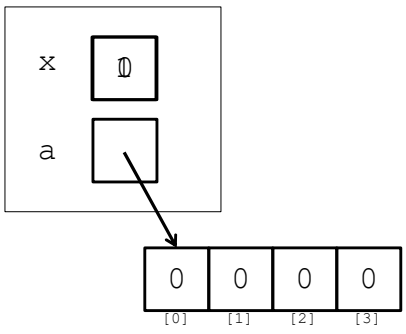
NC STATE UNIVERSITY

Reference Semantics Exercise

Current Line

```
x = x + 1;
```

output



12

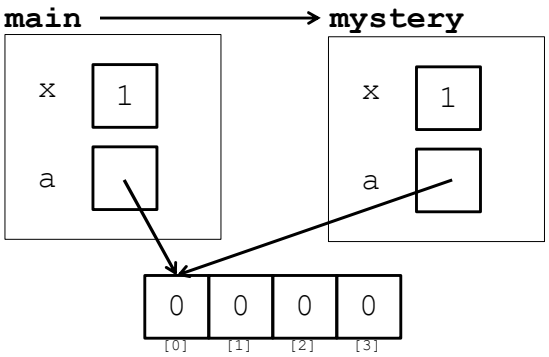
NC STATE UNIVERSITY

Reference Semantics Exercise

Current Line

```
mystery(x, a);
```

output



13

NC STATE UNIVERSITY

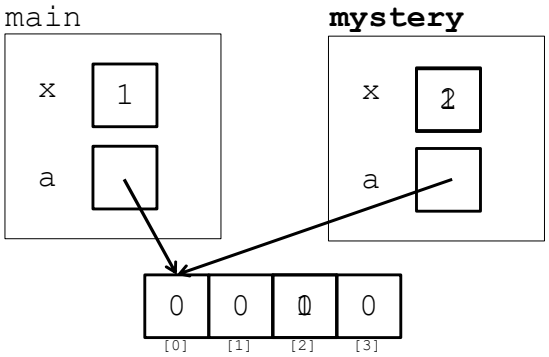
Reference Semantics Exercise

Current Line

```
x = x + 1;
a[x] = a[x] + 1;
sysout(x + " " + Arrays.toString(a);
```

output

```
2 [0, 0, 1, 0]
```



14

NC STATE UNIVERSITY

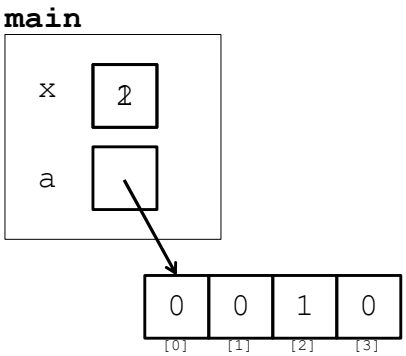
Reference Semantics Exercise

Current Line

```
sysout(x + " " + Arrays.toString(a);
x = x + 1;
```

output

```
2 [0, 0, 1, 0]
1 [0, 0, 1, 0]
```



15

NC STATE UNIVERSITY

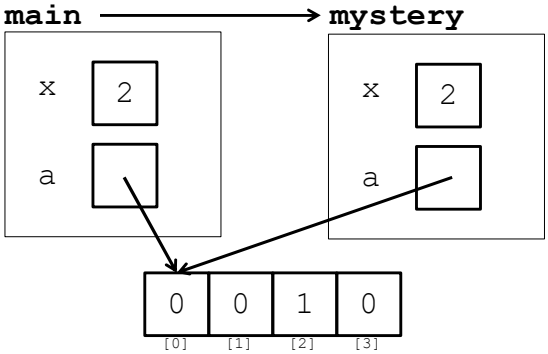
Reference Semantics Exercise

Current Line

```
mystery(x, a);
```

output

```
2 [0, 0, 1, 0]
1 [0, 0, 1, 0]
```



16

NC STATE UNIVERSITY

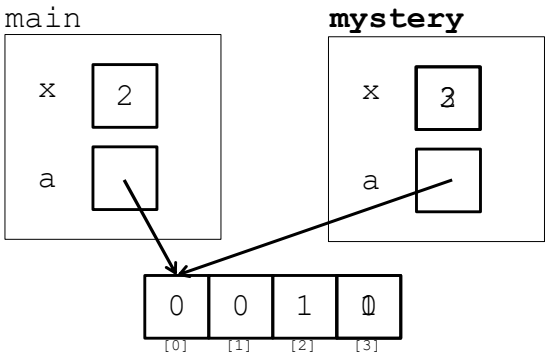
Reference Semantics Exercise

Current Line

```
x = x + 1;
a[x] = a[x] + 1;
sysout(x + " " + Arrays.toString(a);
```

output

```
2 [0, 0, 1, 0]
1 [0, 0, 1, 0]
3 [0, 0, 1, 1]
```



17

NC STATE UNIVERSITY

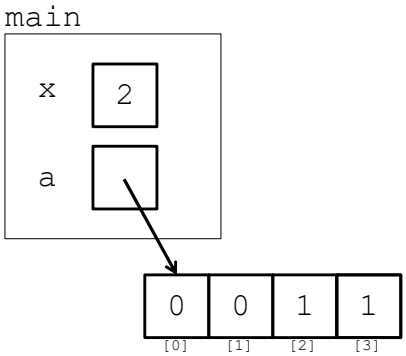
Reference Semantics Exercise

Current Line

```
sysout(x + " " + Arrays.toString(a);
```

output

```
2 [0, 0, 1, 0]
1 [0, 0, 1, 0]
3 [0, 0, 1, 1]
2 [0, 0, 1, 1]
```



18

NC STATE UNIVERSITY

Dereferencing an Object

- **Dereference:** to access data or methods of an object
  - Use the dot notation
  - Specify which object to dereference and what field or method to access

```
Recipe r = new Recipe("Chili");
String name = r.getName().toUpperCase();
r.updateName("Super Mega Hot Chili");
```


NC STATE UNIVERSITY

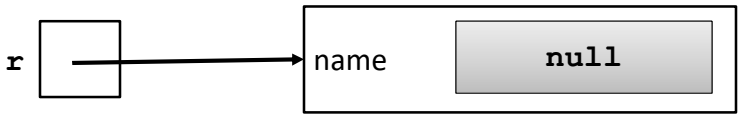
Null References

- `null`: A built in value that does not refer to any object
  - Uninitialized object variables are set to `null`
- Uses of null
  - Initialize a local object variable
    ```
    Recipe r = null;
    ```
  - Check if an object is null
    ```
    if (r == null) { … }
    ```
  - Pass null as a parameter
    - Useful in some cases when use of parameter is unknown
    - Be prepared for that method to throw an exception!
  - Return null from a method
    - Indicate failure or that something wasn't found

NullPointerException

- `null` cannot be dereferenced!
  - `null` has no methods or data

```
Recipe r = new Recipe(); //default const.
String s = r.getName().toUpperCase();
```



Composition and Delegation

- Creating a class from existing classes
  - Classes have members (fields) that are instances of other classes
- Our class works with the object fields by calling methods (and fields) of the object
  - An instance of the original class provides functionality for the class you're writing
  - You're delegating to the field through its methods
  - Delegation is a way of extending and reusing a class

NC STATE UNIVERSITY

Abstraction & Encapsulation

- **Abstraction:** Distancing between ideas and details
  - Can use objects without knowing how they work
  - Example: Phone
- **Encapsulation:** Hiding the implementation details of an object from the clients of the object
  - Protects data from unwanted access
  - Clients cannot directly access of modify its internal workings – nor do they need to do so
  - Encapsulation leads to abstraction

NC STATE UNIVERSITY

Steps for Achieving Encapsulation

- Make fields private
- Ensure class invariants
  - **Class Invariant:** a property that is true of every object/instance of a class
  - An assertion about an object's state that is true for the lifetime of that object
  - Enforce within public methods and call public methods within class (e.g. from constructor)
  - Example: a student id number should never be negative
- If class contains `Object` fields, defensively copy the object to ensure immutability

NC STATE UNIVERSITY

Method Overriding

- Every object you create is an `Object`
- `Object` has useful methods that you can override
  - `toString()`
  - `equals()`
  - `hashCode()`

25

NC STATE UNIVERSITY

The `Object` Class

- `Object` is a (direct or indirect) superclass of all Java classes
  - Every object you construct is an `Object`
- Contains defines nine methods that you can override
  - **Override:** provide your own implementation of the method

Method	Purpose
`String toString()`	Returns a String representation of the object.
`boolean equals(Object o)`	Returns true if some other object is “equal to” this one.
`int hashCode()`	Returns a hash code value for the object.

26

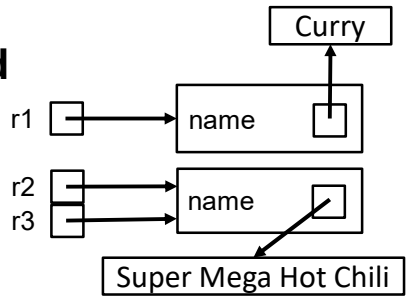
NC STATE UNIVERSITY

The `toString()` method

- Called automatically when ever a `String` is concatenated with an `Object`
- If you don't override `toString()`, the default output is something like
  - "ClassName@d24606bf"
  - Name of the class followed by the hash code
- Have to override `toString()` so that interesting and relevant information is printed
  - Convention: "`ClassName[fieldName=fieldValue,…]`"

NC STATE UNIVERSITY

The `equals()` method



- Called when want to compare objects
- Default behavior of `equals()`?
  - The same thing as ==, which compares object references to see if two objects point to SAME object
- The `equals()` method has a parameter of type `Object`
  - We must ***cast*** the parameter to the same type of object as the class to compare the fields

NC STATE UNIVERSITY

The `hashCode()` method

- Method that uses an algorithm to compute an unsigned integer value that is *likely* to be unique for different objects
- When two or more objects that we would consider different have the same hash code we have a collision
  - Try to write hash functions that minimize collision
  - Hash functions scramble info about the instance fields to create the integer value
    - Typically prime numbers are used
- `hashCode()` should be compatible with `equals()`
  - Objects that are the same should hash to the same value
  - Use the same fields to compare for equality and create the hash code
- If you override `equals()`, you must override `hashCode()`
  - This is a Java invariant!

NC STATE UNIVERSITY

`equals()` and `hashCode()`

- Let Eclipse write your `equals()` and `hashCode()` methods for you
- You should know how to write `equals()` methods without code generation help
  - See the textbooks for suggested structure
- You do not need to know how to write `hashCode()` method, but need to know what it is

NC STATE UNIVERSITY

Implicit Parameter

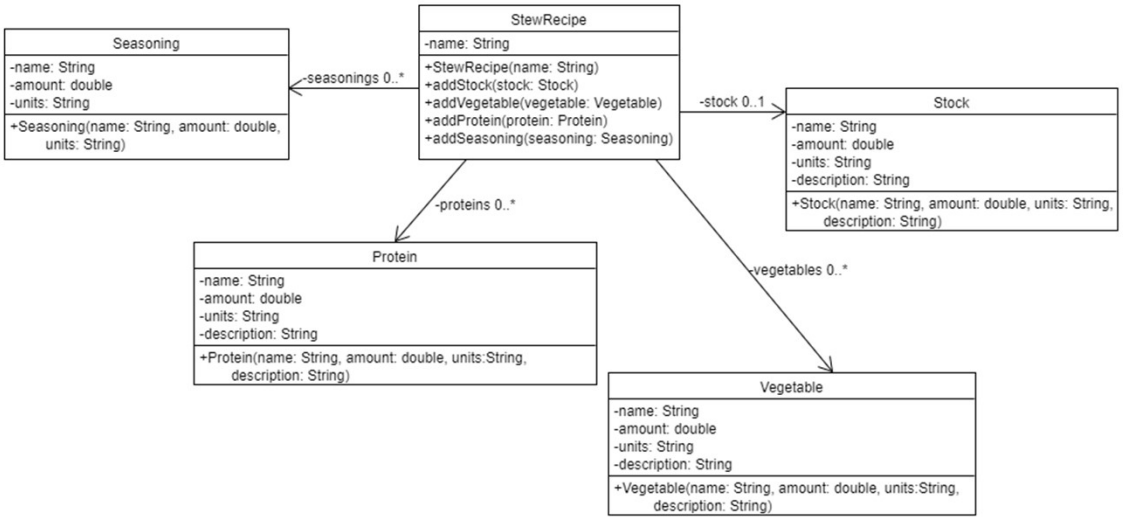
- **Implicit Parameter:** the object being referenced during the call of an instance method
- Provides knowledge of what *specific object* the method will operate on
- We can access the implicit parameter using the keyword `this`
- Calling an object's field, like `x`, is compiled to `this.x`

NC STATE UNIVERSITY

Welcome to Recipe Book

- The Recipe Book is an electronic cookbook. You can collect recipes. Maybe one day, we'll use the info to create an automated chef?
- We're going to focus on a module for Stews
- What makes a stew?

Stew Composition



Stock Implementation (POJO)

```
public class Stock {
   private String name;
   private double amount;
   private String description;

   public Stock(String name, double amount, String description) {
       this.name = name;
       this.amount = amount;
       this.description = description;
   }

   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
```

NC STATE UNIVERSITY

Stock Implementation (POJO)

```
    public double getAmount() { return amount; }
    public void setAmount(double amount) {
          this.amount = amount;
       }
       public String getDescription() { return description; }
       public void setDescription(String description) {
          this.description = description;
       }
       public boolean equals() { … }
       public int hashCode() { … }
       public String toString() { … }
    }
}
```


NC STATE UNIVERSITY

Stew Implementation

Construction

```
import java.util.ArrayList;
public class StewRecipe {
   // Fields – composition relationship
   private Stock stock;
   private ArrayList<Protein> proteins;
   private ArrayList<Vegetable> vegetables;
   private ArrayList<Seasoning> seasonings;

   // Field – composition with Java API
   private String name;

   // Default constructor
   public StewRecipe() {
      this("Stew");
   }
```

Declaration

```
   // Parameterized constructor
   public StewRecipe(String name) {
       this.name = name;
       proteins = new ArrayList<Protein>();
       vegetables = new AL<Vegetable>();
       seasonings = new AL<Seasoning>();
   }
   public String getRecipe() {
      String rtn = stock.toString() + "\n";
      rtn += proteins.toString() + "\n";
      rtn += vegetables.toString() + "\n";
      rtn += seasonings.toString();
      return rtn;
   }
}
```

Use

NC STATE UNIVERSITY

Composition Summary

- Using objects within another class allows for the creation of custom objects
  - Example: `Stew` has `Stock`, `Proteins`, `Vegetables`, `Seasonings`
- Start with primitive types and build more complex objects from those types
  - Example: `Stock` has `name` (`String`), `amount` (`double`), `description` (`String`)
- The container class can define its own methods to call the methods of its instance variables
  - This is *delegation*!
  - Example: `Stew.toString()` delegates to the objects' `toString()` methods

NC STATE UNIVERSITY

Class Design Paradigm

- Do not provide any functionality that does not have a clear use
- Control access to an object's state!
  - Omit mutators (methods that change state) if they are not needed
  - Limit object creation to the constructor (not always possible)
- "Classes should be immutable unless there's a very good reason to make them mutable… If a class cannot be made immutable, limit it's mutability as much as possible." Joshua Bloch

NC STATE UNIVERSITY

Class Design Paradigm

- Classes should have ***cohesion***
  - The extent to which the code for a class represents a single abstraction (idea)
  - Allows for reusability of the class in other programs
- The `Recipe` class represents only things a Recipe can do and knows.
  - A `Recipe` should not perform console input and output
  - A `Recipe` should not ensure that a specific ingredient is constructed correctly.

NC STATE UNIVERSITY

Lab 00: Installation

- Lab 00 checks your environment setup.
  - Individual lab
  - Due
    - All Labs: Wednesday, January 21st at 11:45pm
  - CSC 216 ONLY students – you're encouraged to make sure your environment is up to date!
- Demo Functionality
  - Demo in person during office hours
  - Record and submit a video demo
- Use provided GitHub repo (csc217-2026-spring-L0-<repo#>)

NC STATE UNIVERSITY

Guided Project 1

- Install Eclipse and plug-ins needed for CSC 216
- Learn how to use Eclipse & GitHub
- Start WolfScheduler program
  - Guided Project 1
  - Review pre-req materials (CSC 116 and equivalent) including composition & file I/O
  - Due Thursday, January 29th @ 11:45pm
  - Late Deadline Saturday, January 31st @ 11:45pm
- Use provided GitHub repo (csc216-2026-spring-GP1-<section#>-<repo#>)