Objectives

- Eclipse features
  - Debugger
  - Search
- Decorator design pattern

Review

1. What is the singleton design pattern?
   - When is it useful? How is it implemented?
2. What is the `instanceof` code smell? Why is it a smell?
   - What is the solution?
3. What is the process for evaluating an expression?
   - Consider `floor(y)` and `floor( floor(y) )`
     - Resulting image will not be different
   - Name the components, methods called
     - Template: A calls B's c method, passing in d and e; the method returns f
   - Map back to what these components represent, as appropriate
4. Are you having fun yet?

Review: Singleton Design Pattern

- Goal: Only one object of a class
- How to achieve
  - Make the constructor private
  - Make a public method for accessing the one and only instance

Review: `instanceof` Code Smell

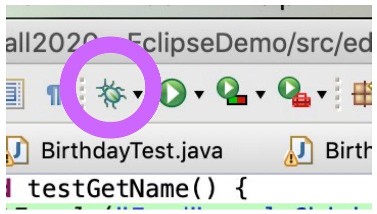
- Problem:
  - Code specific to each possible type → Hard to update as add new types
- Solution: Refactor! Add abstraction! (as usual)
  - Specifically: make a method for that functionality in the classes
  - Let dynamic dispatch call the appropriate method.

Picasso Notes

- Given code base is not perfect but pretty good
- Example imperfections
  - Missing comments/Javadocs
  - Incorrect comments
  - Less-than-ideal naming
  - `CharToken` takes an `int` (rather than a `char`) as a parameter?
- Project goal: you're gaining ***experience***
  - You'll work with imperfect code bases in the future

ECLIPSE DEBUGGER

Eclipse Debugger


all2020 EclipseDemo/src/ed

BirthdayTest.java Birth

```
testGetName() {
```

1. Set breakpoint
   - Near and BEFORE point of failure
2. Run program in debug mode
   - Program pauses when it hits a breakpoint
3. Inspect variables
4. Step through program, inspecting variables
   - Step into, over, and return

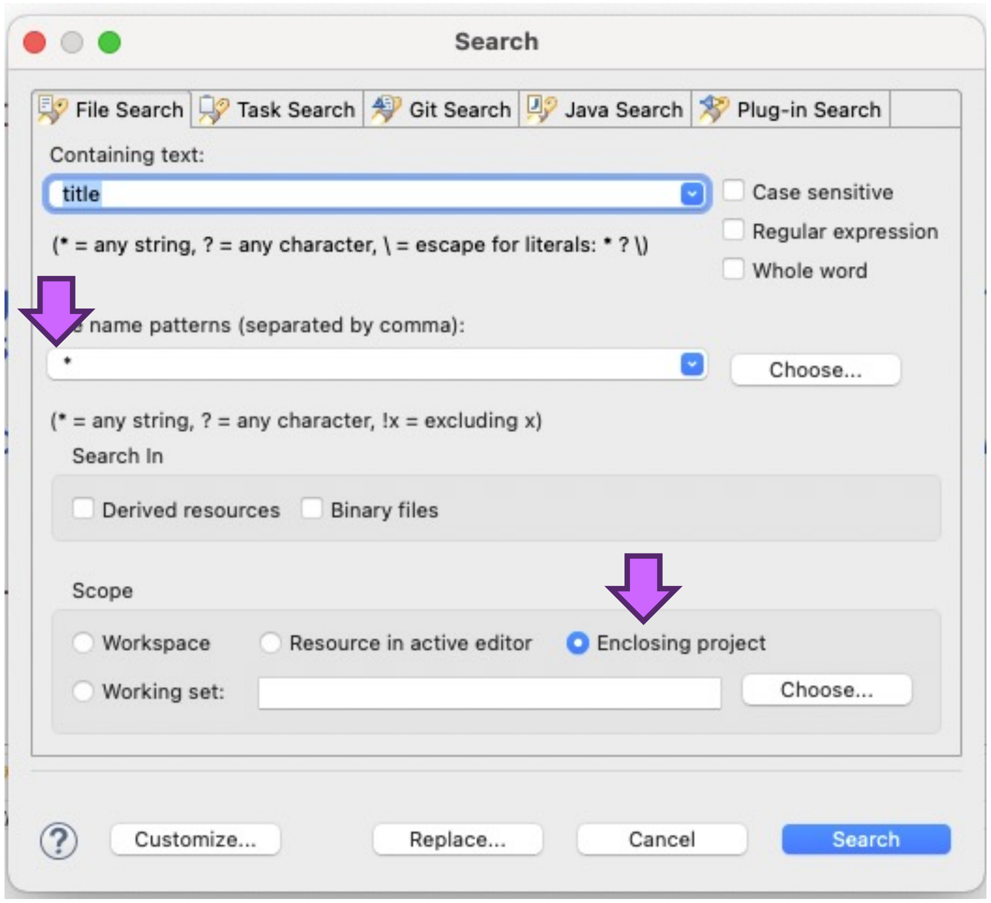
Commands

- Step Into
  - Executes the current line
  - If the current line is a method call, the debugger steps into the method's code
- Step Over
  - Executes a method without stepping into it in the debugger
- Step Return
  - Steps out to the *caller* of the currently executing method
  - Finishes the execution of the current method and returns to the caller of this method

ECLIPSE SEARCH

Eclipse Search: File Search

- More general search


Search

Containing text: title

(* = any string, ? = any character, \ = escape for literals: * ? \)

[ ] Case sensitive
[ ] Regular expression
[ ] Whole word

name patterns (separated by comma): *

(* = any string, ? = any character, !x = excluding x)

Search In

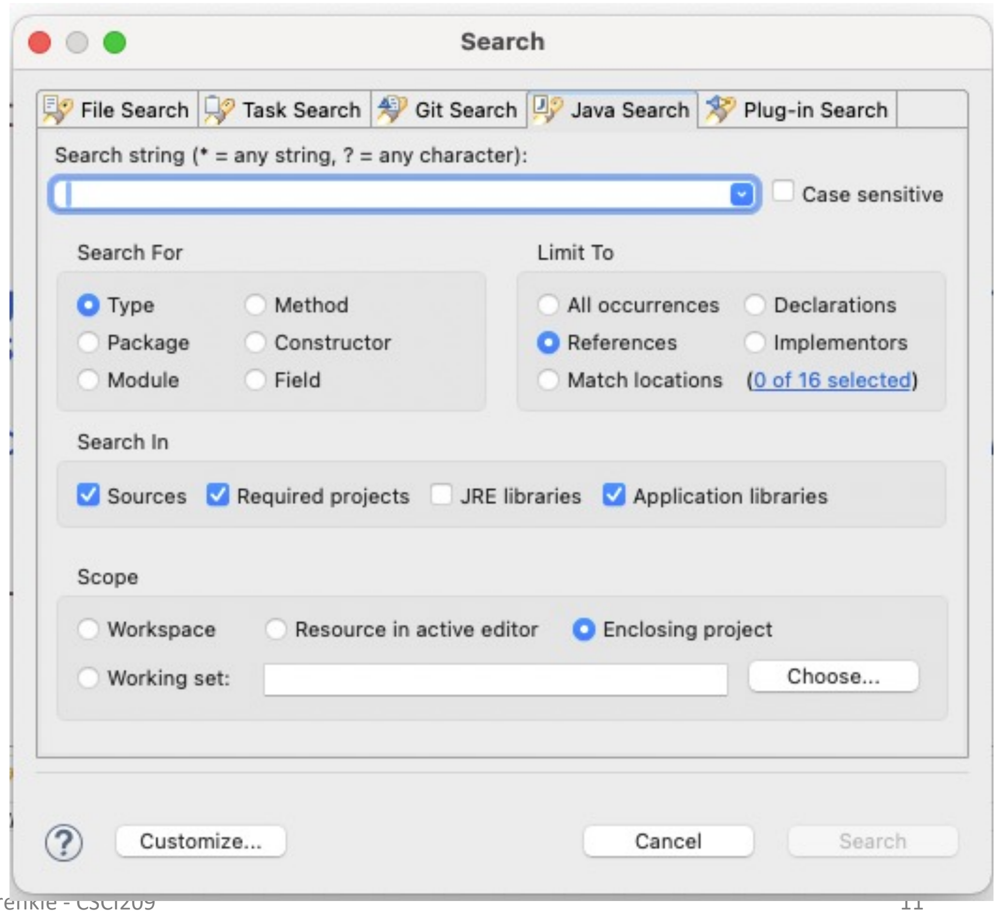
[ ] Derived resources [ ] Binary files

Scope

Workspace / Resource in active editor / [✓ Enclosing project] / Working set:

Eclipse Search: Java Search

- Specific to the Java programming language



Search string (* = any string, ? = any character):

[ ] Case sensitive

Search For

[✓ Type] Method
Package Constructor
Module Field

Limit To

All occurrences Declarations
[✓ References] Implementors
Match locations (0 of 16 selected)

Search In

[✓ Sources] [✓ Required projects] [ ] JRE libraries [✓ Application libraries]

Scope

Workspace Resource in active editor [✓ Enclosing project]
Working set:

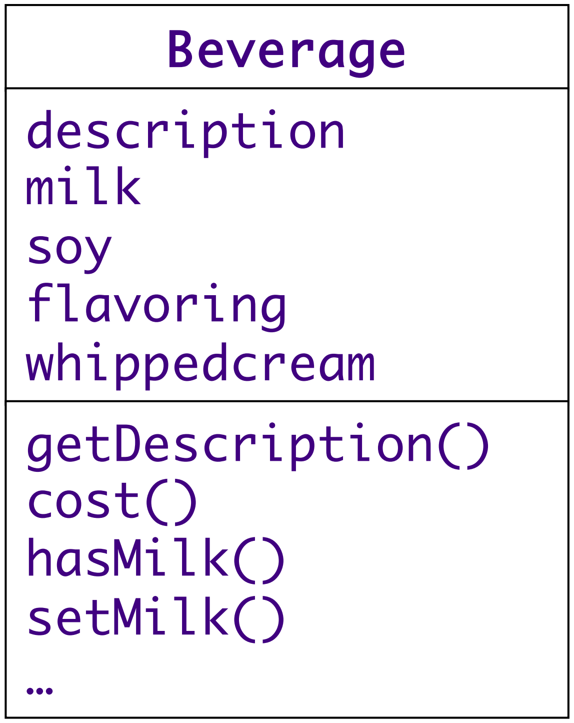
DECORATOR DESIGN PATTERN

What’s Your Drink?

- You go into a coffee shop: what is your drink?

- How can we represent the various beverages in code?
- What are the possible implementation issues?

What’s Your Coffee Drink?



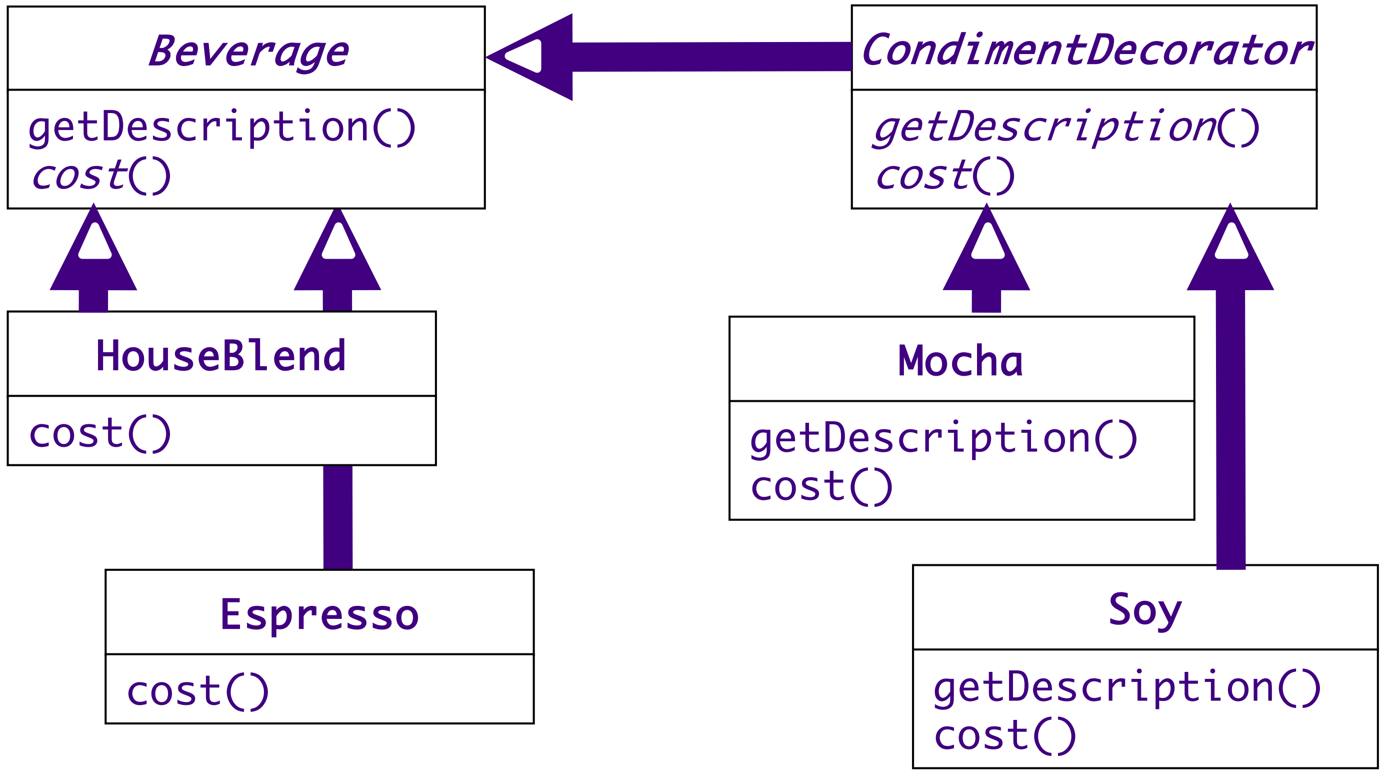
```
Beverage
---------------
description
milk
soy
flavoring
whippedcream
---------------
getDescription()
cost()
hasMilk()
setMilk()
…
```

How many additional methods will we need to add to create a comprehensive beverage object?

How will we compute cost?

What happens when a new beverage feature is added?

One Solution: **Decorator**



UML Diagram

Latte's Implementation

```java
public class Latte extends Beverage {

   private double cost;

   public Latte() {
      this.cost = 3.75;
   }

   public String getDescription() {
      return "Latte";
   }

   public double cost() {
      return this.cost;
   }
}
```

One possibility
(could keep state differently)

Mocha’s Implementation

```
public class Mocha extends CondimentDecorator {

   private Beverage beverage;

   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

   public String getDescription() {
      return beverage.getDescription() + ", Mocha";
   }

   public double cost() {
      return .20 + beverage.cost();
   }
}
```

What design patterns are used within this class?
How would we use this class?
How would we create other beverages?

Using Beverages

```java
public class CoffeeGeneral {

   public static void main(String[] args) {
       Beverage b = new DarkRoast();
       System.out.println(b.getDescription() + " $" + b.getCost());

       Beverage b2 = new DarkRoast();
       b2 = new Mocha(b2);
       b2 = new Mocha(b2);
       b2 = new Whip(b2);
       System.out.println(b2.getDescription() + " $" + b2.getCost());
   }
}
```

Mocha's Implementation

```
public class Mocha extends CondimentDecorator {

   private Beverage beverage;

   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

   public String getDescription() {
      return beverage.getDescription() + ", Mocha";
   }

   public double cost() {
      return .20 + beverage.cost();
   }
}
```

Handles part it knows about,
Delegates rest to Beverage;
Example of OCP

Generalize: when to use the Decorator pattern, tradeoffs of this design pattern?

Design Pattern: **Decorator**

- Adds behavior to an object dynamically
  - Typically added by doing computation before or after an existing method in the object
- Benefits:
  - Alternative to inheritance
  - Can add any number of decorators
  - Each class is responsible for just one thing
- Possible drawback:
  - Could add many small classes → less than straightforward for others to understand

Have we seen decorators used in practice?

Represent Thanksgiving?

```
dinner = new Turkey( new Duck( new Chicken() ) );
```

Not-always-culturally-relevant: Christmas Tree

Picasso: Your Team's Javadocs

- Automatically generated from `main` branch at 3:58 a.m. every day
- Linked from Documentation section of Picasso project page

Reload the page to see changes/updates

FAQ for Picasso

- Linked from the specification page
- Updated as I get new questions

Reload the page to see changes/updates

Preliminary Implementation

- Goals
  - Get your team working together, familiar/comfortable with pull requests
    - No one left out, no one dominating
  - Find kinks in design
    - Rework now instead of later
- Tag your version
- Can keep working after that
  - Return to the tagged version for Friday's demo

Ungraded Objectives

- Think about what you need to complete for the final implementation.
- With your current design, how well does your design extend for the next steps?
  - Next steps include other/different types of expressions/functions, extensions
  - What could be designed better (i.e., make it easier to add these other parts)?
- An hour of thinking about the design and changing the code to improve the design will be worth hours of time later.

Looking Ahead

- Friday: Preliminary Deadline and Demos
- Order of teams will be randomly generated on Friday
  - Schedule: 8:35, 8:47, 9:00, 9:15
  - Schedule: 11:05, 11:17, 11:30, 11:45
- Next steps:
  - How will you add reading expressions from a file?
  - How will you add other components?

Secondary Project Goals

- You’re going to figure out that your final design isn’t perfect—maybe not even good!
- Fix smaller and/or more critical things
  - Refactoring!
- Note larger things
  - Analysis/post-mortem due at end of finals week

Good judgment comes from experience.
How do you get experience?
Bad judgment works every time.