WEEK THREE

Acknowledgements: Slides created based off material provided by Dr. Travis Doom,
Dr. Michael Raymer, and Reese Hatfield

SIMPLIFYING COMPARATOR CODE

- Do I have to have a separate class for every comparator I want to make?
  - Yes…
  - For each, separate implementation, I need a separate class
  - But this class *can* be defined inside my existing class

INNER CLASS

- A class defined *inside* another class
- Inner classes can access any field/method in the outer class (including private fields)
- May need to be static
  - If we don't want to have to create an outer class object to instantiate the inner class

INNER CLASSES EXAMPLE

```
public class Employee implements Comparable<Employee>
{
    private String name = "";
    private int ssn = 0;
    private String dob = "";
    private double salary = 0.0;

    static class SalaryCompare implements Comparator<Employee> {
        @Override
        public int compare(Employee o1, Employee o2) {
            return Double.compare(o1.getSalary(), o2.getSalary());
        }
    } . . .
}
```

INNER CLASS EXAMPLE CONTINUED

```
public class Testing {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee("bob", 1234, "04-03-1988", 100);

        ArrayList<Employee> roster = new ArrayList<>();
        roster.add(e1);
        roster.add(e2);

        Collections.sort(roster, new Employee.SalaryCompare());
    }
}
```

ANONYMOUS INNER CLASSES

- If I don't need to reuse the Comparator, I can define it via an *anonymous* inner class:

```
Collections.sort(roster, new Comparator<Employee>() {
    @Override
    public int compare(Employee o1, Employee o2) {
        return Double.compare(o1.getSalary(), o2.getSalary());
    }
});
```

LAMBDA EXPRESSIONS

ADDED IN JAVA 8 (2014)

- Anonymous inner classes are often written in an even simpler form called a lambda expression
  - `Collections.sort(roster, (o1, o2) -> Double.compare(o1.getSalary(), o2.getSalary()));`
- Can only use when interface being implemented has ***exactly one*** method
- Syntax (note the *expression* or *body* serve as the code for the method)
  - No parameters: `() -> expression`
  - One parameter: `paramName -> expression`
  - Two parameters: `(paramOneName, paramTwoName) -> expression`
  - Multi-line body:

```
() -> {
    body
}
```

OBJECT DECLARATION

- When an object is created, the class type often matches the constructor called
  - `Scanner scnr = new Scanner(System.in);`
- However, via inheritance and polymorphism, a parent class may be used as the class type
  - `Object input = new Scanner(System.in);`
- So can I do this?
  - `String name = input.next();`
  - No, next() will cause a “Cannot resolve method” error
  - But Scanner class has the next() method, so why can’t we call it?
  - Because the declared type (Object) does not have a next() method

DECLARED VS. ACTUAL TYPE

Declared Type

- Appears on LHS of assignment
  - `Object input =`
- Checked at compile time (before code is run)
  - Compiler ensures no "rules" are broken based on the declared type
  - If the method called is not in the class of the declared type, compiler gives an error
  - Casting may be used to pass checks
- Can be an interface or abstract class

Actual Type

- Appears on RHS of assignment
  - `Scanner(System.in);`
- Used at runtime
  - Controls what method implementation is run, if multiple (dynamic dispatch)
  - While the object may be referenced as a parent type, under the hood, code from the *actual* type will be executed when possible
- Must be concrete

DYNAMIC DISPATCH

- When a polymorphic method is called, which version is run?
  - A polymorphic method is one that is overloaded or **overridden**
  - Which overloaded method to execute can be determined at compile time
  - Which overridden method to execute is determined at run time (based on actual type)
- Dynamic dispatch is runtime determination of which method implementation to call/execute
  - Why is this useful?
  - Collections with parent types as the declared type
  - Method parameters with abstract or interface declared types
  - Etc.

DYNAMIC DISPATCH EXAMPLE

- If the following code is run, which method is run?

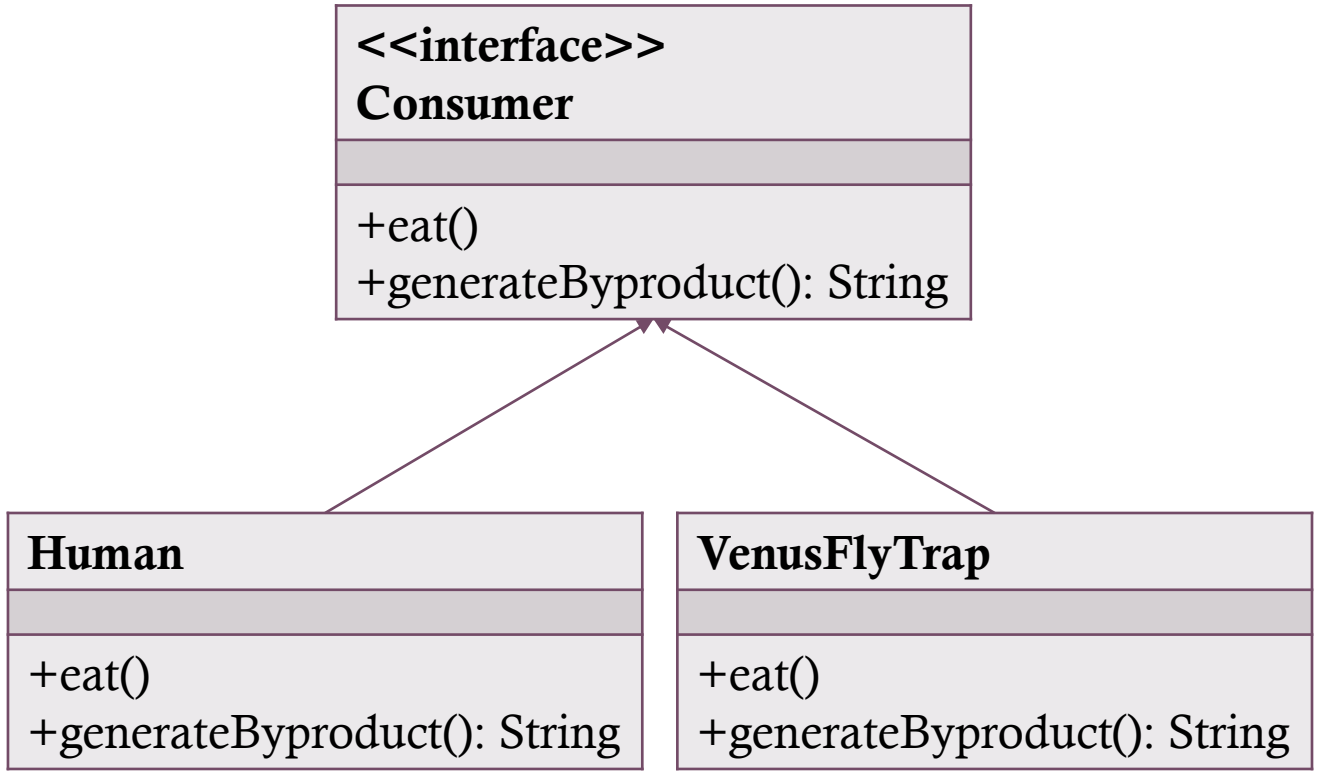
```
Human h = new Human();

dinner(h);


public static void dinner(Consumer c) {

        c.eat();

}
```



INHERITING FROM AN EXISTING CLASS

- Inheritance is extremely powerful and useful
  - What happens if a class extends a pre-existing class like ArrayList?
  - The new class gains all the functionality of ArrayList
- Suppose I want to create a class called EditHistory that holds a set of String objects that describe all the edits made to a document
  - I want to be able to add new edits (String objects), access edits at a certain location, etc.
  - I could have my class extend ArrayList<String>, so I don't have to rewrite any code!
  - But what if I don't want any edits to be removed/replaced?
  - When a class extends another, it gets *all* the functionality of the parent
  - Instead, I can have an ArrayList<String> as a field in the class

COMPOSITION VS. INHERITANCE

Composition

- "Has-a" relationship
- Better control over functionality
- More flexible/future-proof
- More straightforward

Inheritance

- "Is-a" relationship (specialization)
- Shares functionality with *all* parent classes
- Can reduce code duplication (not a primary reason to choose inheritance)
- Polymorphic benefits

WELL DEVELOPED CLASSES (USING OOP PRINCIPLES)

- Encapsulation
  - Good use of access modifiers
  - Getters/setters
- Inheritance (from Object)
  - toString()
  - equals() and hashCode()
- Interface implementation
  - Comparable
- Shallow/deep copies
  - Deep copiable
- Exceptions
  - Handles common exceptions
  - Descriptive robust exceptions