Composition

OOP Principles, encapsulation, composition, has-a, delegation

Central Principles of OOP

- **Abstraction** – Focus on essential qualities of an entity/system rather than a specific example: focus on *what* rather than *how*.
- **Encapsulation** – Bundling together an object's data and operations on that data. Implementation details of objects are hidden from the clients that use them.
- **Inheritance** – Deriving new (child) classes from (parent) classes. The data and methods of the parent are inherited by the child.
- **Polymorphism** – Objects behave according to their actual types (rather than their declared types).

Encapsulation

- Hiding the details of the implementation of an object from client code using the object.
  - Client code knows only what it must know.
  - Protects class from improper access (the class determines its access, not the client code).
- Achieving encapsulation.
  - Declare class data as private (or at least static final if public).
  - Define methods for changing/reading data as needed.
- Program for **class invariants** (property that holds true for every instance of the class and every state of the instance). Invariants constrain the object so that "methods can always reference the object without the risk of making inaccurate presumptions."
  - Avoid public setters when possible.
  - Methods should assure invariants are met.

Encapsulation - example

```
package edu.ncsu.csc216.samples;

public class Account {
   private String owner;
   private double balance;

   public Account(String owner, double balance) {
      if (owner == null || owner.equals(""))
         throw new IllegalArgumentException("Empty name");
      this.owner = owner;
      if (balance < 0)
         balance = 0;
      this.balance = balance;
   }

   public String getOwner() { return owner; }
   public double getBalance() { return balance; }

   // continued  next slide
}
```

Encapsulation - example

```
public void deposit(double amt) {
  if (amt < 0)
   throw new IllegalArgumentException("Negative deposit");
  balance += amt;
}
public void withdraw(double amt) {
  if (amt > balance)
   throw new IllegalArgumentException(”NSF");
  balance -= amt;
}
```

Encapsulations principles illustrated:

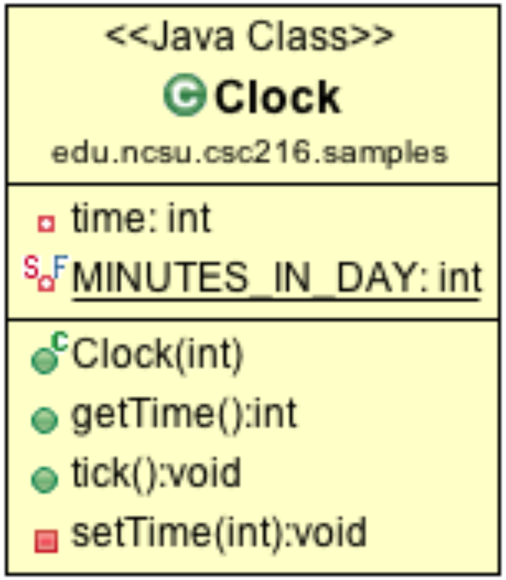
- Private instance variables.
- No direct setters.
- Class invariants:
    - owner is never null or blank
    - balance is never negative

Composition

- One way of creating classes from other classes.
  - Client code knows only what it must know.
  - Protects class from improper access (the class determines its access, not the client code).

1: Composition example: Clock

Task: construct a class named **`Oven`** with instance variables of other class types: **`Clock`** and **`Thermometer`**.



Clock design:

- Attributes:
  - time
  - MINUTES_PER_DAY
- Behaviors:
  - Constructor
  - getTime
  - tick
  - setTime (private)

```java
public class Clock {
   private int time;
   private static final int MINUTES_IN_DAY = 1440;

   public Clock(int time) {
      setTime(time);
   }
   public int getTime(){
      return time;
   }
   public void tick() {
      setTime(++time);
   }
   private void setTime(int time) {
      this.time = Math.abs(time) % MINUTES_IN_DAY;
   }
}
```

2: Composition: Thermometer

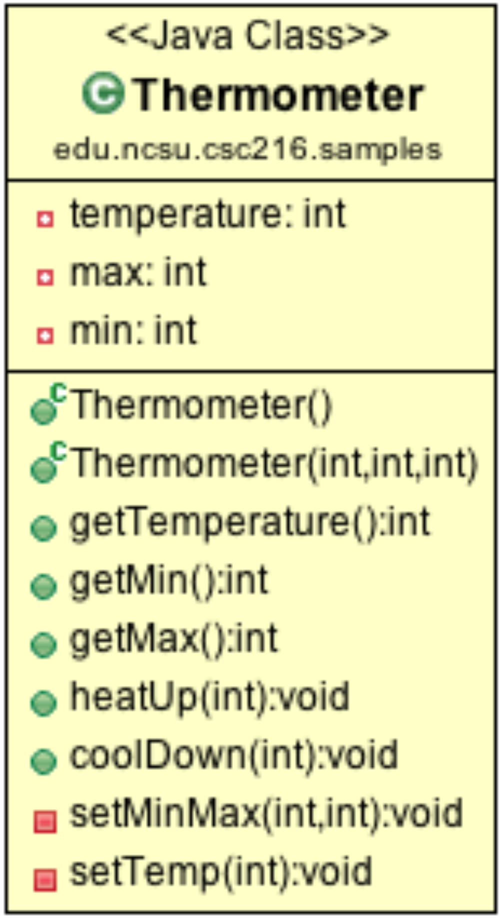
Thermometer design:

Attributes:

- temperature
- max
- min

Behaviors:

- Constructors
- getTemperature
- getMin
- getMax
- heatUp
- coolDown
- setMinMax (private)
- setTemp (private)



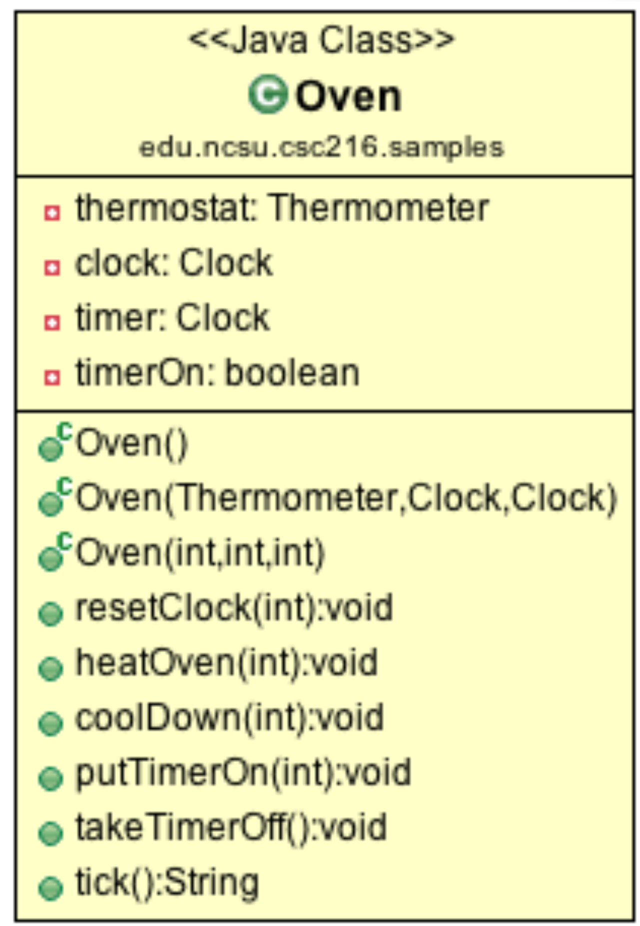
```java
public class Thermometer {
   private int temperature;
   private int max;
   private int min;

   public Thermometer() {
      this(70, -25, 130);
   }
   public Thermometer(int temperature, int min, int max) {
      this.temperature = temperature;
      setMinMax(min, max);
   }
   public int getTemperature() { return temperature;}
   public int getMin() { return min; }
   public int getMax() { return max;}

   public void heatUp(int degrees) {
      temperature += degrees;
      if (temperature > max)
         temperature = max;
   }

 // Continued next page
```

```java
// .. Continue Thermometer definition

   public void coolDown(int degrees) {
      temperature -= degrees;
      if (temperature < min)
         temperature = min;
   }

   private void setMinMax(int min, int max) {
      if (min > max)
         throw new IllegalArgumentException("Max can't be < Min");
      this.min = min;
      this.max = max;
      setTemp(temperature);
   }

   private void setTemp(int temp) {
      temperature = Math.max(temp, min);
      temperature = Math.min(temp, max);
   }
}
```

3. Composition: Oven

Oven design

- Attributes:
    - thermostat
    - clock
    - timer
    - timerOn
- Behaviors:
    - Constructors
    - resetClock
    - heatOven
    - turnOffOven
    - putTimerOn
    - takeTimerOff
    - tick



```java
public class Oven {
   private Thermometer thermostat;
   private Clock clock;
   private Clock timer;
   private boolean timerOn = false;

   public Oven () {
      this(100, 500, 100);
   }
   public Oven (int temperature, int max, int min){
      thermostat = new Thermometer(temperature, max, min);
      clock = new Clock(0);
      timer = new Clock(0);
   }

   public void resetClock(int time) {
      clock = new Clock(time);
   }
   public void heatOven(int degrees){
      thermostat.heatUp(degrees - thermostat.getTemperature());
   }

   // Continued next page
```

```java
// ... Continue Oven definition
   public void coolDown(int degrees) {
      thermostat.coolDown(degrees);
   }
   public void putTimerOn(int amountOfTime) {
      if (amountOfTime > 0) {
         timerOn = true;
         timer = new Clock(clock.getTime() + amountOfTime);
      }
   }
   public void putTimerOff(){
      timerOn = false;
   }
   public String tick() {
      clock.tick();
      if (timerOn && clock.getTime() == timer.getTime()){
         putTimerOff();
         return "Dinner is ready!";
      }
      return "";
   }
}
```

Composition concepts

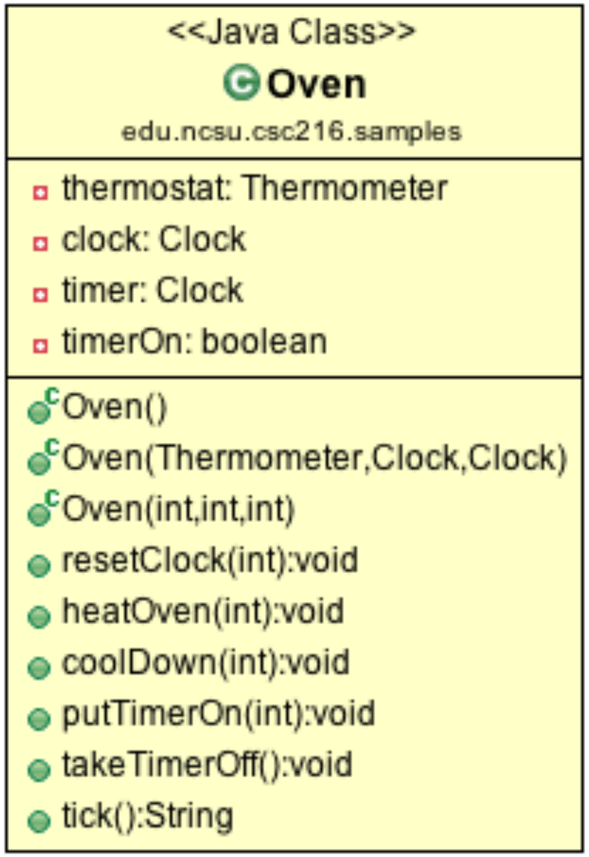
Composition means creating a class with objects for instance variables. This is one way to reuse existing code.

- ***has-a*** relationship. An Oven has-a Clock.

```
private class Clock {
   private Clock timer;
   private Clock clock;
   private Thermometer thermostat;
```

- **Delegation** - passing the task on to the object instance variable – an implementation detail.

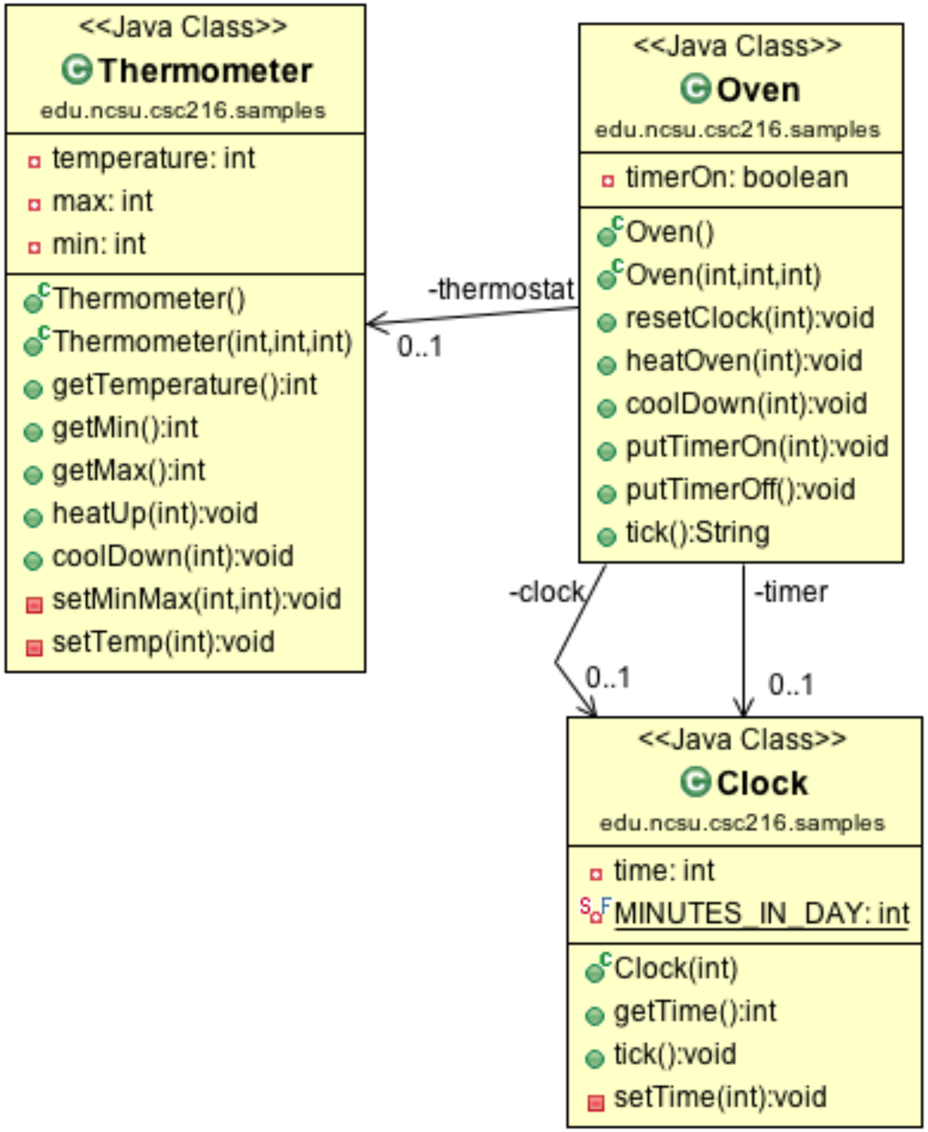
```
public void resetClock(int time) {
   clock.setTime(time);
 }
```



Oven system class diagram

This is a UML class diagram. It shows three classes and the relationships among them:

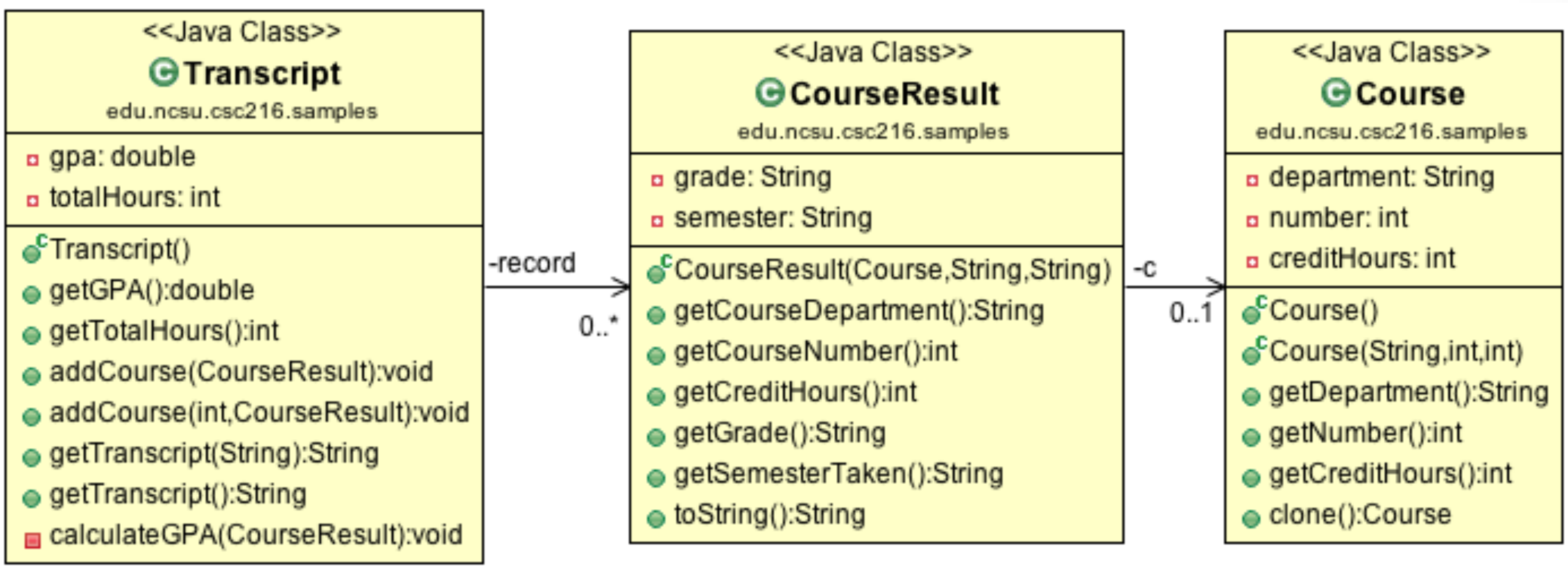
- Oven has two Clocks, which are named clock and timer. They're private.
- Oven has a Thermometer named thermostat. It is also private.



Course/Transcript example

Composition can be nested:

- A Course consists of a department name, number, credit hours.
- A CourseRecord consists of a Course, a grade, a semester
- A Transcript consists of many CourseRecords, a gpa, and total number of credit hours earned



Course

```
public class Course {
   private String department = "CSC";
   private int number = 216;
   private int creditHours = 3;

   public Course() {}
   public Course(String department, int number, int hours){
      this.department = department;
      this.number = number;
      this.creditHours = hours;
   }

   public String getDepartment() { return department; }
   public int getNumber() { return number; }
   public int getCreditHours() { return creditHours; }

   public Course clone() {
      return new Course(department, number, creditHours);
   }
}
```

CourseResult

```
public class CourseResult {
   private Course c;
   private String grade;
   private String semester;

   public CourseResult(Course c, String grade,
                       String semester){
      this.c = c.clone();
      this.grade = grade;
      this.semester = semester;
   }

   public String getCourseDepartment(){
      return c.getDepartment();
   }
   public int getCourseNumber() { return c.getNumber();}
   public int getCreditHours()  { return c.getCreditHours(); }
   public String getGrade()     { return grade;}
   public String getSemesterTaken() { return semester; }

   public String toString() {
      return getCourseDepartment()+ getCourseNumber() + "  "
            + getCreditHours() + "   " + grade;
   }
}
```

Transcript

```
import java.util.Arraylist;

public class Transcript {
   private double gpa;
   private int totalHours;
   private ArrayList<CourseResult> record;

   public Transcript() {
      record = new ArrayList<CourseResult>();
   }

   public double getGPA() { return gpa; }
   public int getTotalHours() { return totalHours; }

   public void addCourse(CourseResult c){
      record.add(c);
      calculateGPA(c);
   }
   public void addCourse(int position, CourseResult c){
      record.add(position, c);
      calculateGPA(c);
   }

   // Transcript continued ...
```

```java
// Continue Transcript definition

public String getTranscript(String semester) {
      String r = "";
      for (CourseResult c : record){
         if (c.getSemesterTaken().equals(semester))
            r += c.toString()+ "\n";
      }
      return r;
   }
   public String getTranscript() {
      String r = "";
      for (CourseResult c : record)
         r += c.toString()+ "\n";
      return r;
   }

   private void calculateGPA(CourseResult c) {
      // calculation code goes here
   }
}
```

Composition summary

- You can create a class using instance variables that are different class types.
- Typically start with primitives types and build more complex types through them.
- Methods of a class that is created through composition can often delegate part of their tasks to the class type instance variables.

Class design best practices

- Do not provide any functionality that does not have clear use or that is not essential to the class.
- Avoid direct setters when possible. Limit mutators.
- Classes should have **cohesion**.
  - Cohesion is the extent to which the code for a class represents a single abstraction.
  - Cohesion allows for reusability of classes in different contexts.
- The `Thermometer` class represents only the things a thermometer knows and can do.
  - A `Thermometer` should be able to give temperature rise and fall.
  - A `Thermometer` should not do console input or output.

Program design best practices

- Classes should not have unnecessary dependencies.
  Coupling is the degree to which one part of a program depends on another.
- Related data and behaviors should be in the same class.
- Use packages to group related classes.