TOPIC 12

CREATING CLASSES
PART 1

Notes adapted from Introduction to Computing and Programming with Java: A Multimedia Approach by M. Guzdial and B. Ericson, and instructor materials prepared by B. Ericson.

Outline

- Identifying objects and classes
- Defining a **class**
- Defining **attributes**
  - Also called **fields**
  - Also called **instance variables**
- Defining **constructors**
  - Overloading constructors
- Defining **methods**

Recall: Java is Object-Oriented

- In Java, the focus is on **objects**
- **Objects** are entities that can do actions or be acted upon in a Java program
- All objects have
  - **Properties**
    - These are the data about an object
    - In Java we call them **attributes** or **fields**
  - **Behaviors** (**actions**)
    - In Java they are implemented as **methods**
- Examples: Picture objects, Picture methods

Objects and Classes

- Every object belongs to a specific **class**
  - Objects that belong to the same class share properties and behaviors
  - Example: we can invoke the methods of the Picture class on any Picture object
- We can think of a class as being a template or pattern or model for objects of that class

Objects and Classes

5

- **Object-oriented programs** consist of interacting objects
  - Which are defined by classes
  - And created by other classes
- Example:
  - Picture class defines the attributes of a Picture object and the methods that can be invoked on a Picture object
  - We can write programs to perform some task, that create and use objects of the Picture class

Object-Oriented Design

6

- To identify the objects in a task:
  - What are the things that are doing the work or being acted upon?
  - How do you classify them?
  - What data do they need to know to do the task? What **attributes** describe them? (**fields**)
  - What actions do they need? What can they do / what can be done to them? (**methods**)

Identifying the Objects and Classes

7

- Say that we want to write a program to track the grades of students
- One way to start is to identify the nouns
  - We get grades and student
  - Then we decide if something should be
    - a **class** (template for objects)
    - or a **field** (data)
    - Does it have more than one piece of data associated with it?
  - A student has a name and grades, so it should be defined by a class

Identifying the Objects and Classes

8

- Decide on the **fields (attributes)** of the class:
  - A particular student has
    - a name, which is a string
    - grades, which are numbers
    - these will be **fields** of the student class
- Decide on the **actions (methods)** to be performed on the student objects
  - We need to analyze what we might want to do with the data
  - Examples:
    - Show the student name and grades
    - Get the average of the grades for the student
- We will do this example in detail later

Class Definition

- A **class definition** consists of
  - Field (attribute) definitions
  - Constructor definitions
  - Method definitions
- A class definition is stored in a **file**
  - With the same name as the class
  - With a .java extension on the file
  - Examples: Turtle.java, Picture.java

Class Definition – Class Names

- **Class names**
  - Should be singular
    - Why? we are describing an object of that type
    - Examples: Turtle, Picture, Student
  - Start with an uppercase letter
  - The rest of the word is lowercase, except the first letter of each additional word should be uppercase
    - Examples: ColorChooser, SimpleTurtle

Class Definition - Syntax

- The syntax for a **class definition** is:

```
visibility class ClassName
{

  field (=attribute) definitions

  constructor definitions

  method definitions

}
```

Class Definition - Attributes

- **Field (attribute) definitions**
  - Example: the SimpleTurtle class of our textbook declares many attributes for an object of this class, some of which are:

```
int width = 15;  // width of this turtle
int height = 18; // height of this turtle
int xPos = 320; // x coordinate of current position
int yPos = 240; // y coordinate of current position
```

  - Attributes can be **initialized to default values**
  - What is SimpleTurtle? See next slide

Brief Digression: Inheritance

13

- In our textbook examples, we have used the class Picture
- The intention is that we can add methods to this class
- The authors have put the parts that they do not want changed into a class called SimplePicture, and Picture is just an "extension" of SimplePicture
  - We say Picture **inherits from** SimplePicture and that SimplePicture is the **parent** of Picture
  - An object of type Picture is also of type SimplePicture, so it has its attributes and methods
- This is the structure of Turtle and SimpleTurtle also

Class Definition - Attributes

14

- Syntax for **attribute/field definition**:
  visibility type name;
  visibility type name = expression;
  - We usually use private for the visibility
  - The type is a primitive type or a class name
  - Field names start with a lowercase letter

- Examples:

```
private int width = 15;
private  int height = 18;
```

Class Definition - Attributes

- Why are attributes usually private?
  So that other classes cannot access them directly (i.e. can only access them through methods of the class)
  - Example: getRed(), setRed() methods of the Pixel class
    - Users of the Pixel class do not need to know how/where the red value for a pixel is stored
    - In fact, the representation for the colors of a pixel could change in a new version of Pixel.java, and users do not even need to know that

Class Definition - Attributes

- The variables that we define for the attributes are also known as **instance variables**
  - Why? They are variables that describe an **instance** of the class, i.e. an object of the class
  - Example: width, height, xPos, yPos, etc. are instance variables for the SimpleTurtle class

Class Definition - Constructors

17

- **Constructor definitions**
  - A **constructor** is a special method that is called automatically when an object is created with the new operator
  - Its purpose is to initialize the attributes of an object when the object is created
  - Examples:

```
World world1 = new World();
Turtle turtle1 = new Turtle(100,200, world1);
Turtle turtle2 = new Turtle(world1);
```

  - Constructors have the same name as the class name

Class Definition - Constructors

18

- Syntax for a ***constructor definition***

```
visibility  ClassName (paramList)
{
   // assign values to instance variables
}
```

  - Note that constructor has no return type
- Example: constructor that initializes turtle start position to the parameter values

```
public SimpleTurtle(int x, int y) {
    xPos = x;            // initialize xPos
    yPos = y;            // initialize yPos
          …
}
```

Class Definition - Constructors

- **Default Field Values**
  - If a constructor does not initialize an instance variable to some specific value, it has as its default value the value from the attribute definition
  - Example:

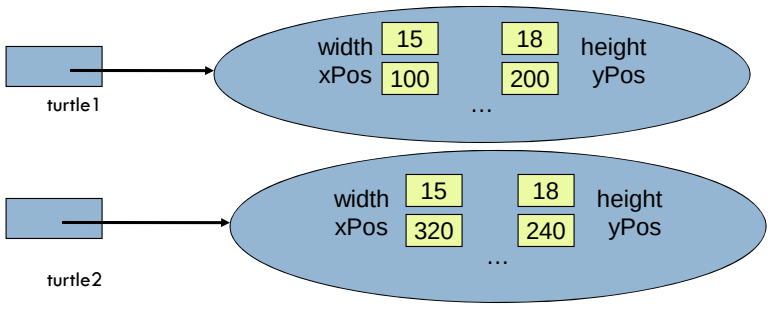
```
int width = 15;      // width of this turtle
int height = 18;     // height of this turtle
```

- When an object is created, it has its own set of instance variables

Objects and their Attributes

- Example:

```
World world1 = new World();
Turtle turtle1 = new Turtle(100,200, world1);
Turtle turtle2 = new Turtle(world1);
```



Class Definition - Methods

- **Method definitions**
  - Recall the syntax for **defining a method**

```
visibility returnType name(parameterList)
{
    body of method
}
```

  - Example from Picture class:

```
public void changeRed(double howMuch)
{
  Pixel[] pixelArray = this.getPixels();
  // etc.
}
```

Example: Student Class

- We will now define a class Student that models keeping track of a student's grades
  - A Student class could be much more complex
  - We will define a very simple class for now

Class Definitions in DrJava

- To define the Student class
  - Click on the New button in DrJava
  - Type in the Definitions pane:

```
public class Student
{
  // fields, constructors, methods go here
}
```

  - Save it in Student.java

Example: Student Class

- A student should have a name and some grades associated with it, so the name and grades should be **fields** in our Student class
- What type should we use for each of these?
  - The field name can be a String
  - We will have a collection of grades, and each can have a decimal point in it
    - So, we will use double as the type
    - Stored in an array of grades

Example: Student Class - Fields

25

```
public class Student
{
   //  fields (attributes)
   private String name;
   private double[ ] gradeArray;

   …
}
```

Example: Student Class - Constructor

26

- Add the constructor definition to the Student class after the field definitions:

```
public Student(String theName)
{
    this.name = theName;
}
```

Class Definitions – Using Attributes

- Within the methods of a class, you can access the attributes (and other methods) of the class directly
- Example: constructor on previous slide

```
public Student(String theName) {
    this.name = theName; }
```

- We could also write this as

```
public Student(String theName) {
    name = theName; }
```

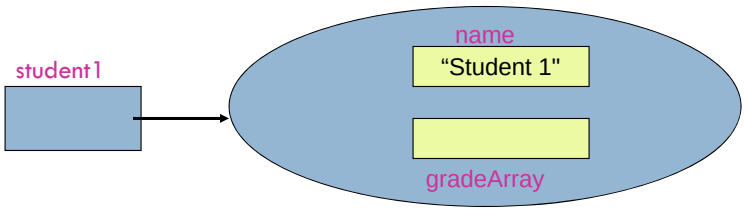
  - The "this" is implicit here

Example: Create a Student object

- Suppose a new Student object is created in some program by

```
Student student1 = new Student("Student 1");
```

  - What is its name field initialized to?
  - What is its gradeArray field initialized to? Why?



Constructor Overloading

- You can have more than one constructor
  - As long as the parameter lists are different
  - This is called **constructor overloading**
  - We have seen method overloading before, in the methods we wrote for the Picture class

Example: Another Constructor

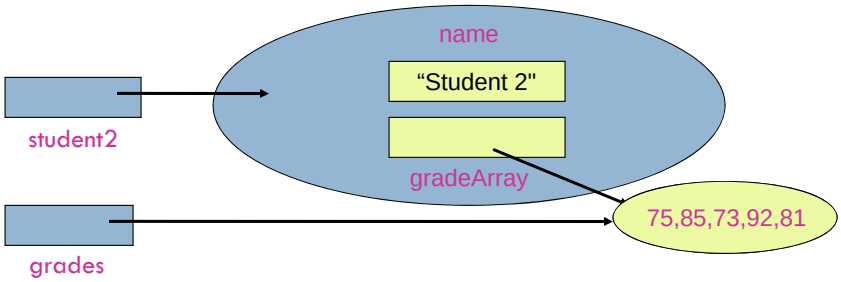
- Add another constructor to the Student class that takes both the name and an array of grades as parameters:

```
public Student(String theName, double[] theGrades)
{
    this.name = theName;
    this.gradeArray = theGrades;
}
```

Example: Create a Student object

- We could create another new Student object in some program by:

```
double [ ] grades = {75,85,73,92,81};
Student student2 = new Student("Student 2", grades);
```



Example: Student Class - Methods

- What methods might we want in the Student class?
- We need to decide what we might want to do with the data:
  - Show the student name and grades
  - Get the average of the grades for the student

Example: toString Method

- It is conventional to have a toString() method in every class
- It returns a string containing the object's data
- Which can then be printed

```
public String toString()
  {
     String s = "Student " + this.name + " has grades ";
     for (int i = 0; i < this.gradeArray.length; i ++)
          s = s + this.gradeArray[i] + " ";
     return s;
  }
```

Example: toString Method

- We can now print a student object's data using System.out.println(student2.toString());

- We can also do this using System.out.println(student2);
  - Why? Java automatically invokes the toString() method for the class of which student2 is an object
    - If that class does not have a toString() method, Java will use the toString() method of the Object class, which is the parent class of every other class
    - This does not, however, provide useful information about the data of the object

Example: toString Method

- Now consider our previous example object:
  Student student1 = new Student("Student 1");
  - What is its name field initialized to?
  - What is its gradeArray field initialized to?
- What will happen now if we type
  System.out.println(student1.toString());
  - How can we handle that?

Example: Revised toString Method


```
public String toString()
{
    String s = "Student " + this.name;
    if (this.gradeArray != null)
    {
        s = s + " has grades ";
       for (int i = 0; i < this.gradeArray.length; i ++)
            s = s + this.gradeArray[i] + " ";
    }
    return s;
}
```

Example: Calculate Grade Average

- To calculate an average:
  - Sum the grades
  - Divide by the number of grades (the length of the grade array)
- We need to be careful of:
  - A null gradeArray
  - A 0 length gradeArray

Example: getAverage Method

- Create a method getAverage that calculates and returns the average of the grades in the grade array
- Algorithm:
  - Return 0 if the grade array is null
  - Return 0 if the grade array length is 0
  - Otherwise return
    sum of grades / number of grades

The getAverage Method


```
public double getAverage()
{
   double average = 0.0;
   if (this.gradeArray != null && this.gradeArray.length > 0)
   {
      double sum = 0.0;
      for (int i = 0; i < this.gradeArray.length; i++)
      {
         sum = sum + this.gradeArray[i];
      }
      average = sum / this.gradeArray.length;
   }
   return average;
}
```

Example: Testing our Student class

- Try this in the Interactions pane:

```
Student student1 = new Student("Student 1");
System.out.println(student1.toString());
System.out.println("Average = " + student1.getAverage());
double [] grades = {75,85,73,92,81};
Student student2 = new Student("Student 2", grades);
System.out.println(student2.toString());
System.out.println("Average= " + student2.getAverage());
```

Testing equality

- Suppose we have another student

  Student student3 = new Student("Student 2", grades);
- Do we mean that student2 and student3 are the same student?
- They are both reference variables, but to different Student objects
- However, the two objects have the same contents
- This is state equality, as opposed to identity equality

Testing equality

- Testing identity equality is done through the expression (student1 == student2)
- Important use: testing if a Student object is null

  if (student1 == null)

  System.out.println("Student object not initialized");
- To test state equality (between students), one writes a method

  public boolean equals(Student otherStudent)

  that returns true if the attributes are the same
- Not easy to write correctly

Rules to keep in mind

- Make sure that you're not trying to access some methods or attributes of a null object
- Make sure that you're not trying to access the length of a null array
- Concretely: test whether they are null or not

- If you need to test whether two arrays have the same contents, you should not use ==

Testing state equality for students


```
public boolean equals(Student otherStudent)
    if (otherStudent == null)
        return false;      // the current student cannot be null
    // first we check if the names are the same
    if ( ! this.name.equals(otherStudent.name) )
        return false;      // equals exists in the String class
    // to compare the arrays of grades, we first deal
    // with the case where one could be null
    if (this.gradeArray == null && otherStudent.gradeArray == null)
        return true;
```

Testing state equality for students


```
    if (this.gradeArray == null || otherStudent.gradeArray == null)
        return false;
    // so now, none of the arrays is null
    if (this.gradeArray.length != otherStudent.gradeArray.length)
        return false;
    // now, both arrays have the same length
    for (int i = 0; i < this.gradeArray.length; i++)
        if (this.gradeArray[i] != otherStudent.gradeArray[i])
            return false;
    return true;
}
```

Summary

- Identifying objects and classes
- Defining a **class**
- Defining **attributes**
  - Also called **fields**
  - Also called **instance variables**
- Defining **constructors**
  - Overloading constructors
- Defining **methods**
- Understanding **equality**