Classes and Objects
In Java

CMSC 202H (Honors Section)

John Park

A Class Is a Type

- A class is a programmer-defined type.
- Variables can be declared of a class type.
- A value of a class variable type is called an ***object*** or an ***instance*** of the class.
  - If A is a class, then the phrases
    - “X is of type A“
    - “X is an object of the class A"
    - “X is an instance of the class A"

  mean the same thing

Objects

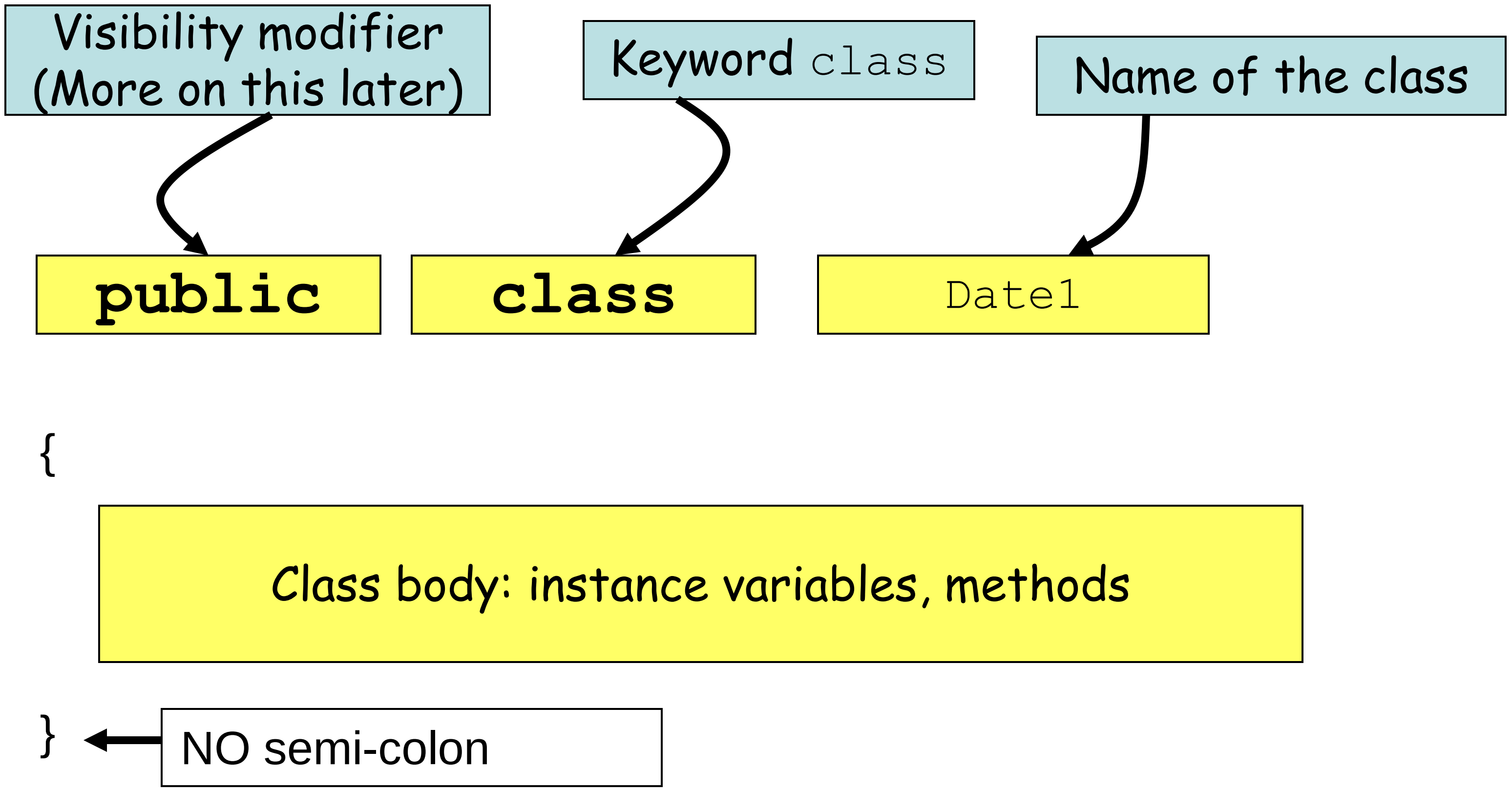
- All objects of a class have the same methods.

- All objects of a class have the same attributes (i.e., name, type, and number).
  - For different objects, each attribute can hold a different value.
  - The values of the attributes define the object ***state***, which is what makes each object unique.

The Class Definition

- A ***class definition*** implements the class model.
  - The class behaviors/services/actions/operations are implemented by class ***methods***.
  - The class attributes (data items) are called ***fields*** or ***instance variables***.
- In Java, classes are defined in files with the .java extension.
- The name of the file must match the name of the class defined within it.
  - e.g. class ‘Baker’ must be in Baker.java

Anatomy of a Java Class

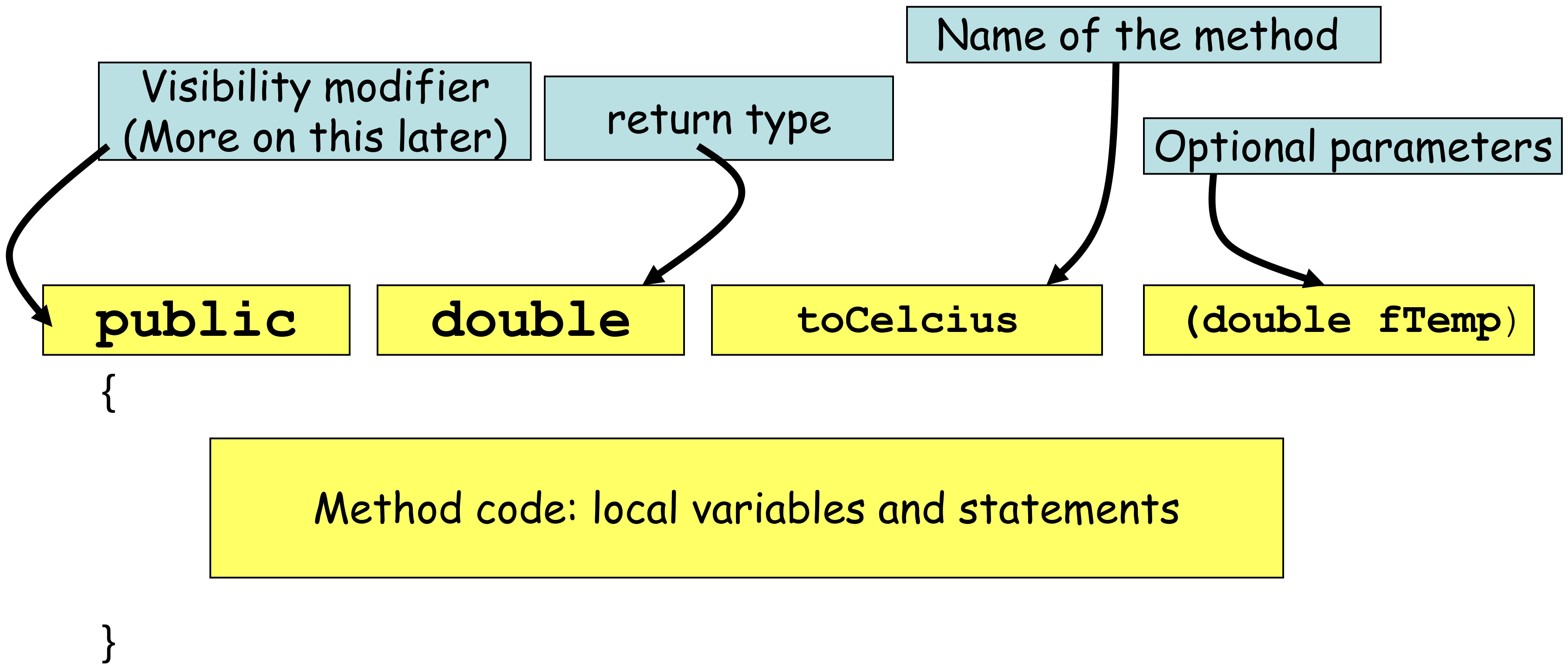

Visibility modifier (More on this later)
Keyword class
Name of the class
public
class
Date1
{
Class body: instance variables, methods
}
NO semi-colon

Instance Variables

- Defined inside the class definition
- May be
  - primitive types
  - other class types
- Are accessible by all methods of the class
  - have ***class scope***
- Given the services identified for the red-green-yellow traffic light, the garage door opener and the bank account, what instance variables might be defined for each?

Anatomy of a Method

Are very much like functions



```
public double toCelcius (double fTemp)
{
    Method code: local variables and statements
}
```

Example: A Date Class

This class definition goes in a file named Date1.java.

```
public class Date1
{
   public String month;
   public int day;
   public int year;

   public String toString( )
   {
       return month + " " + day + ", " + year;
   }
}
```

These are the (public)“data members” or “instance variables” of the class

This is a method definition and its implementation

A method may use the class instance variables

Date1 toString Method

- toString is a method of the Date1 class.
  - Its definition and implementation are part of the Date1 class.
- Class methods may
  - be void or return a value, and
  - (optionally) have parameters, which may be
    - primitive types passed by value, and/or
    - objects (discussed later).
- All of a class' methods have access to all of the class' instance variables (class scope).

Using Date1

This class definition goes in a file named Date1Demo.java.

```
public class Date1Demo
{
   public static void main( String[ ] args )
   {
       Date1 myDate;
       myDate = new Date1( );

       myDate.month = “July”;
       myDate.day = 4;
       myDate.year = 2007;

       String dateString = myDate.toString( );
       System.out.println(dateString);

   }
}
```

Create a Date1 object named myDate

Give values to the data members

Invoke the toString method

Using Date1

This class definition goes in a file named Date1Demo.java.

```
public class Date1Demo
{
   public static void main( String[ ] args )
   {
         Date1 myDate;
         myDate = new Date1( );

         myDate.month = “July”;
         myDate.day = 4;
         myDate.year = 2007;

         String dateString = myDate.toString( );
         System.out.println(dateString);


   }
}
```

Create a Date1 object named myDate

Creating the Date1 Object

- The statement `Date1 myDate;` defines a variable of type Date1.
  - But there is no Date1 object yet!

- The statement `myDate = new Date1( );` creates a “new” Date1 object and assigns a reference to it, to the variable “myDate”.
  - Now “myDate” ***refers to*** a Date1 object.

- For convenience, these statements can be combined.

```
Date1 myDate = new Date1( );
```

Using Date1

This class definition goes in a file named Date1Demo.java.

```
public class Date1Demo
{
   public static void main( String[ ] args )
   {
         Date1 myDate;
         myDate = new Date1( );

         myDate.month = “July”;
         myDate.day = 4;
         myDate.year = 2007;

         String dateString = myDate.toString( );
         System.out.println(dateString);

   }
}
```

Give values to the data members

“Dot” Notation

- Public instance variables of an object are referenced using the “dot” operator.

```
myDate.month = “July”;
myDate.day = 4;
myDate.year = 2007;
```

- Instance variables can be used like any other variable of the same type.

- The set of values stored in all instance variables define the ***state*** of the myDate object.

Using Date1

This class definition goes in a file named Date1Demo.java.

```
public class Date1Demo
{
  public static void main( String[ ] args )
  {
      Date1 myDate;
      myDate = new Date1( );

      myDate.month = “July”;
      myDate.day = 4;
      myDate.year = 2007;

      String dateString = myDate.toString( );
      System.out.println(dateString);

  }
}
```

Invoke the toString method

More “Dot” Notation

- The statement

  ```
  myDate.toString( );
  ```

  invokes the `toString` method of `myDate`, which refers to an object of type Date1.

- In OO terminology, we say that we are “sending the `toString` message” to the object referred to by `myDate`.

- The object `myDate` is referred to as the ***calling object*** or ***host object***.

Calling Object

- Generally invoke methods from the calling object
  - E.g.: “myDate.toString();”
- Method is being invoked on behalf of calling object
- Method’s body has easy direct access to calling object
  - Can easily access its instance variables
  - Can invoke calling object’s other methods

Accessing Members Inside Class

- Accessing instance variables:
  - In many cases, can access instance variables by simple name:
    - when any (non-static) method is called, references from within the method body to instance variables by name are assumed to apply to the calling/host object
- Accessing class methods:
  - When invoking any other method of the calling object, inside the same class, you just use the method name (no “dot” notation)
    - Invoked method also retains same calling object

Other Date Methods

Some other possible services that the Date1 class might provide:

- incrementDay - changes the date to "tomorrow"
- DMYString – creates a different string format
- setDate - initialize/change the year, month, and/or day
- What others ?

New Date1 Methods

```
// change the month (using an int), day, and year.
public void setDate( int newMonth, int newDay, int newYear )
{
     month = monthString( newMonth );
     day = newDay;
     year = newYear;
}

// change month number (int) to string - used by setDate
public String monthString( int monthNumber ) {
     switch ( monthNumber )  {
              case 1:  return "January";
              case 2:  return "February";
              case 3:  return "March";
              case 4:  return "April";
              case 5:  return "May";
              case 6:  return "June";
              case 7:  return "July";
              case 8:  return "August";
              case 9:  return "September";
              case 10: return "October";
              case 11: return "November";
              case 12: return "December";
              default: return “????”;
     }
}
```

Confusion?

- In the preceding **setDate** method it's tempting to define the method using the common terms "month", "day" and "year" as the parameters.

```
public void setDate( int month, int day, int year)
{
    month = monthString( month );// which month is which?
    day = day;                   // which day is which?
    year = year;                 // which year is which?
}
```

However, parameter variables take precedence over instance variables, so the compiler will assume in this case that all uses of **day**, **month**, and **year** refer to the *method parameters,* and hence <u>this code has no effect</u>.

Calling Object

What the code in **setDate** is really trying to do is

```
public void setDate( int month, int day, int year)
   {
       "calling object".month = monthString( month );
       "calling object".day = day;
       "calling object".year = year;
   }
```

It's handy (and sometimes necessary) to have a name for the calling object.

In Java, we use the reserved keyword **this** as the generic name of the calling object.

Using **this**

So, if we want to name our parameters the same as our instance variables:

```
public void setDate( int month, int day, int year)
{
    this.month = monthString( month );  // notice "this"
    this.day = day;
    this.year = year;
}
```

Note:

- Many examples in the text use this technique for class methods.
- Some Java programmer tools (including Eclipse) use this technique when writing code for you.

this Again

Recall the **toString** method from Date1:

```
public void toString( )
{
     return month + " " + day + " " + year;
}
```

It's clear that **month, day**, and **year** refer to the instance variables of the calling object because there are no parameters.

We could have written:

```
public void toString( )
{
     return this.month + " " + this.day + " " + this.year;
}
```

Even when the prefix this is not strictly necessary, it is often included for clarity..

Sample Code Segment Using Date1

```
Date1 newYears = new Date1( );
newYears.month = “January”;
newYears.day = 1;
newYears.year = 2008;

Date1 birthday = new Date1( );
birthday.month = “July”;
birthday.day = 4;
birthday.year = 1776;

System.out.println(newYears.toString( ));             // line 1
System.out.println(birthday.toString( ));             // line 2
System.out.println(birthday.monthString(6));          // line 3
birthday.setDate( 2, 2, 2002);                        // line 4
System.out.println(birthday.toString( ));             // line 5
newYears.day = 42;                                    // line 6
System.out.println(newYears.toString( ));             // line 7
```

January 42, 2008

- It appears that classes allow the user to change the data anytime he or she chooses, possibly making the data invalid.
- That's true so far because we have defined our instance variables with **`public`** access.
- This is rarely the case in real applications.

More About Methods

- Different classes can define a method with the same name.
- Java can determine which method to call based on the type of the calling object.
- Example:

```
Date1 birthday = new Date1( );
Dog fido = new Dog( );
System.out.println(birthday.toString( ));
System.out.println(fido.toString( ));
```

  - `birthday.toString( )` will call the `toString`( ) method defined in the Date1 class because birthday's type is Date1.
  - `fido.toString`( ) will call the `toString`( ) method defined in the Dog class because fido's type is Dog.

Method Overloading

- Two or more methods *in the same class* may also have the same name.

- This technique is known as ***method overloading***.

Overloaded setDate

- The Date1 class `setDate` method:

```
public boolean setDate( int month, int day, int year )
```

- Suppose we wanted to change only the day and year?
  - Define another method named setDate:

```
public boolean setDate( int day, int year )
```

(After all, setDate is a good descriptive name for what this method does.)

Date2 Class - Overloaded setDate Method

```
public class Date2
{
   public String month;
   public int day;              // 1 - 31
   public int year;             // 4 digits

   public boolean setDate( int newMonth, int newDay, int newYear )
   {
        // code here
   }

   public boolean setDate( int newDay, int newYear );
   {
        // code here, doesn't change month
   }

   // toString( ), monthString( ), etc. follow
}
```

Date2Demo Class

```
public class Date2Demo
{
   public static void main (String[ ] args)
   {
        Date2 myDate = new Date2( );

        myDate.setDate( 1, 23, 1982 );
        System.out.println( myDate.toString( ) );
        myDate.setDate( 4, 1999 );
        System.out.println( myDate.toString( ) );
   }
}
```

How does Java know which setDate method to invoke?

Method Signature

- A method is uniquely identified by
  - its name and
  - its parameter list: more specifically, just the sequence of types.
- This is known as its ***signature***.

Examples:

```
public boolean setDate(int newMonth, int newDay, int newYear)
public boolean setDate(String newMonth, int newDay, int newYear)
public boolean setDate(int newDay, int newYear)
public boolean setDate(int newDay, String newMonth)
// But adding this next one will cause problems:
public boolean setDate(int newYear, int dayOfYear)
```

Return Type is Not Enough

- Suppose we attempt to create an overloaded `setDay()` method by using different return types.

```
public void setDay( int day )    { /* code here */ }
public boolean setDay( int day ) { /* code here */ }
```

- This is NOT valid method overloading because the code that calls `setDay( )` can ignore the return value.

```
birthday.setDay( 22 );
```

- The compiler can’t tell which `setDay( )` method to invoke.
- Just because a method returns a value doesn’t mean the caller has to use it.

Too Much of a Good Thing

Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler. Example:

```
public class X {
    //version 1
    public void printAverage ( int a, double b) {
        /*code*/
    }

    //version 2
    public void printAverage ( double a, int b) {
        /*code*/
    }
}
```

Why might this be problematic?

Too Much of a Good Thing

```
public void printAverage ( int a, double b) {/*code*/}
public void printAverage ( double a, int b) {/*code*/}
```

- Now, consider this:

```
    X myX = new X( );
    myX.printAverage( 5, 7 );
```

- The Java compiler can’t decide whether to:
  - promote 7 to 7.0 and invoke the first version of printAverage(), or
  - promote 5 to 5.0 and invoke the second.
- It will throw up its hands and complain
- Take-home lesson: don’t be too clever with method overloading

CMSC202 Standards: Comments

- CMSC202 has fairly strict documentation standards.
- In Advanced Sect., we will be a little more Javadoc-compliant.
- Separate templates for class header and method headers

Class-level Documentation

- Class header format:

```
/**
 * File: Table.java
 * Project: CMSC 202 Project 3, Fall 2010
 * Date: 9/26/2010
 * Section: 13
 * E-mail: jdoe22@umbc.edu
 * Class Description:
 * Class Invariant
 * 1. number of legs is either 3 or 4
 * 2. shape is one of ROUND, RECTANGLE or OVAL
 * @author Bob Smith
 */
```

Method-level Documentation

- Method header format:

```
/**
 * Name: circleArea
 * PreCondition: the radius is greater than zero
 * PostCondition: none
 * @param radius - the radius of the circle
 * @return the calculated area of the circle
 * (@throws – optional)
 */
double circleArea ( double radius ) {
    // handle unmet precondition
    if (radius < 0.0) {
        return 0.0;
    } else {
        return Math.PI * radius * radius;
    }
}
```

Instance Variable Documentation

- Javadoc wants the variable descriptions on line before actual declaration:

```
/** first name of the account holder */
String firstName;
/**
 * the last name of the account holder
 * (note we can have a multi-line description).
 */
String lastName;
```

Method Documentation

- Clear communication with the class user is of paramount importance so that he can
  - use the appropriate method, and
  - use class methods properly.
- Method comments:
  - explain what the method does, and
  - describe how to use the method.
- Two important types of method comments:
  - ***precondition*** comments
  - ***post-conditions*** comments

Preconditions and Postconditions

- Precondition
  - What is assumed to be true when a method is called
  - If any pre-condition is not met, the method may not correctly perform its function.
- Postcondition
  - States what will be true after the method executes (assuming all pre-conditions are met)
  - Describes the side-effect of the method, e.g. if state of instance changes

An Example

Very often the precondition specifies the limits of the parameters and the postcondition says something about the return value.

```
/*
  Pre-condition:
       1 <= month <= 12
       day appropriate for the month
       1000 <= year <= 9999
   Post-condition:
       The month, day, and year of the calling object
   have been set to the parameter values.
   @return true if the calling object has been changed,
           false otherwise
*/
public boolean setDate(int month, int day, int year)
{
   // code here
}
```