Why User-Defined Classes?

Primitive data types (int, double, char, ..) are great …

… but in the real world, we deal with more complex objects: products, Web sites, flight records, employees, students, ..

Object-oriented programming enables us to manipulate real-world objects.

User-Defined Classes

• Combine data and the methods that operate on the data
• Advantages:
  – Class is responsible for the validity of the data.
  – Implementation details can be hidden.
  – Class can be reused.
• Client of a class
  – A program that instantiates objects and calls methods of the class
Syntax for Defining a Class

```java
accessModifier class ClassName
{
    // class definition goes here
}
```

- Class or members can be referenced by
  - methods of the same class,
  - methods of other classes
  - methods of subclasses,
  - methods of classes in the same package

Software Engineering Tip

- Use a noun for the class name.
- Begin the class name with a capital letter.

Important Terminology

- Fields
  - *instance variables*: data for each object
  - *class data*: *static* data that all objects share
- Members
  - fields and methods
- Access Modifier
  - determines access rights for the class and its members
  - defines where the class and its members can be used

Access Modifiers

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Class or members can be referenced by…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public</code></td>
<td>methods of the same class, and</td>
</tr>
<tr>
<td></td>
<td>methods of other classes</td>
</tr>
<tr>
<td><code>private</code></td>
<td>methods of the same class only</td>
</tr>
<tr>
<td><code>protected</code></td>
<td>methods of the same class, methods of</td>
</tr>
<tr>
<td></td>
<td>subclasses, and methods of classes in</td>
</tr>
<tr>
<td></td>
<td>the same package</td>
</tr>
<tr>
<td>No access modifier (package access)</td>
<td>methods in the same package only</td>
</tr>
</tbody>
</table>
**public VS. private**

- Classes are usually declared to be `public`
- Instance variables are usually declared to be `private`
- Methods that will be called by the client of the class are usually declared to be `public`
- Methods that will be called only by other methods of the same class are usually declared to be `private`
- APIs of methods are published (made known) so that clients will know how to instantiate objects and call the methods of the class

---

**Defining Instance Variables**

Syntax:

```
accessModifier dataType identifierList;
```

dataType can be primitive data type or a class type

identifierList can contain:
- one or more variable names of the same data type
- multiple variable names separated by commas
- initial values

- Optionally, instance variables can be declared as `final`

---

**Examples of Instance Variable Definitions**

```java
private String name = "";
private final int PERFECT_SCORE = 100,
    PASSING_SCORE = 60;
private int startX, startY,
    width, height;
```

---

**Software Engineering Tips**

- Define instance variables for the data that all objects will have in common.
- Define instance variables as `private` so that only the methods of the class will be able to set or change their values.
- Begin the identifier name with a lowercase letter and capitalize internal words.
**The Auto Class**

```java
public class Auto {
    private String model;
    private int milesDriven;
    private double gallonsOfGas;
}
```

**Writing Methods**

Syntax:

```
accessModifier returnType methodName(
    parameter list ) // method header
{
    // method body
}
```

- **parameter list** is a comma-separated list of data types and variable names.
  - To the client, these are *arguments*
  - To the method, these are *parameters*
- Note that the method header is the method API.

**Software Engineering Tips**

- Use verbs for method names.
- Begin the method name with a lowercase letter and capitalize internal words.

**Method Return Types**

- The return type of a method is the data type of the value that the method returns to the caller. The return type can be any of Java's primitive data types, any class type, or *void*.

- Methods with a return type of *void* do not return a value to the caller.
Method Body

- The code that performs the method's function is written between the beginning and ending curly braces {...}.
- Unlike if statements and loops, these curly braces are required, regardless of the number of statements in the method body.
- In the method body, a method can declare variables, call other methods, and use any of the program structures we've discussed, such as if/else statements, while loops, for loops, switch statements, and do/while loops.

main is a Method

```java
public static void main( String [] args )
{
    // application code
}
```

Let's look at `main`'s API in detail:
- `public main` can be called from outside the class. (The JVM calls `main`.)
- `static main` can be called by the JVM without instantiating an object.
- `void main` does not return a value
- `String [] args main`'s parameter is a `String` array

Value-Returning Methods

- Use a `return` statement to return the value
- Syntax:
  ```java
  return expression;
  ```

Constructors

- Special methods that are called when an object is instantiated using the `new` keyword.
- A class can have several constructors.
- The job of the class constructors is to initialize the instance variables of the new object.
### Defining a Constructor

Syntax:
```java
public ClassName( parameter list )
{
   // constructor body
}
```

Note: no return value, not even `void`!
- Each constructor must have a different number of parameters or parameters of different types
- **Default constructor**: a constructor that takes no arguments.
- See Examples 7.1 and 7.2, Auto.java and AutoClient.java

### Default Initial Values

- If the constructor does not assign values to the instance variables, they are auto-assigned default values depending on the instance variable data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte, short, int, long</code></td>
<td>0</td>
</tr>
<tr>
<td><code>float, double</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>char</code></td>
<td><code>space</code></td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td>Any object reference (for example, a <code>String</code>)</td>
<td><code>null</code></td>
</tr>
</tbody>
</table>

### Common Error Trap

Do not specify a return value for a constructor (not even `void`).

Doing so will cause a compiler error in the client program when the client attempts to instantiate an object of the class.

### Class Scope

- Instance variables have **class scope**
  - Any constructor or method of a class can directly refer to instance variables.
- Methods also have class scope
  - Any method or constructor of a class can call any other method of a class (without using an object reference).
Local Scope

• A method's parameters have **local scope**, meaning that:
  – a method can directly access its parameters.
  – a method's parameters cannot be accessed by other methods.
• A method can define local variables which also have local scope, meaning that:
  – a method can access its local variables.
  – a method's local variables cannot be accessed by other methods.

Summary of Scope

• A method in a class can access:
  – the instance variables of its class
  – any parameters sent to the method
  – any variable the method declares from the point of declaration until the end of the method or until the end of the block in which the variable is declared, whichever comes first
  – any methods in the class

Accessor Methods

• Clients cannot directly access private instance variables, so classes provide public accessor methods with this standard form:
  ```java
  public returnType getInstanceVariable(  )
  {
    return instanceVariable;
  }
  ```
  
  *(returnType is the same data type as the instance variable)*

• Example: the accessor method for model.
  ```java
  public String getModel( )
  {
    return model;
  }
  ```

*See Examples 7.3 Auto.java & 7.4 AutoClient.java*
Mutator Methods

• Allow client to change the values of instance variables

```java
public void setInstanceVariable(
    dataType newValue )
{
    // validate newValue,
    // then assign to instance variable
}
```

• Example: the mutator method for `milesDriven`

```java
public void setMilesDriven( int newMilesDriven )
{
    if ( newMilesDriven >= 0 )
        milesDriven = newMilesDriven;
    else
    {
        System.err.println( "Miles driven 
            cannot be negative." );
        System.err.println( "Value not changed." );
    }
}
```

• See Examples 7.5 Auto.java & 7.6 AutoClient.java

Software Engineering Tip

• Write the validation code for the instance variable in the mutator method and have the constructor call the mutator method to validate and set initial values
• This eliminates duplicate code and makes the program easier to maintain

Common Error Trap

• Do not declare method parameters.
  – Parameters are defined already and are assigned the values sent by the client to the method.
• Do not give the parameter the same name as the instance variable.
  – The parameter has name precedence so it "hides" the instance variable.
• Do not declare a local variable with the same name as the instance variable.
  – Local variables have name precedence and hide the instance variable.
Data Manipulation Methods

• Perform the "business" of the class.
• Example: a method to calculate miles per gallon:

```java
public double calculateMilesPerGallon()
{
    if (gallonsOfGas != 0.0)
        return milesDriven / gallonsOfGas;
    else
        return 0.0;
}
```

• See Examples 7.7 Auto.java & 7.8 AutoClient.java

The Object Reference this

• How does a method know which object's data to use?
• **this** is an implicit parameter sent to methods and is an object reference to the object for which the method was called.
• When a method refers to an instance variable name, this is implied
• Thus:
  variableName is understood to be this.variableName
  model is understood to be this.model

Using this in a Mutator Method

```java
public void setInstanceVariable(dataType instanceVariableName)
{
    this.instanceVariableName = instanceVariableName;
}
```

• Example:
```java
public void setModel(String model)
{
    this.model = model;
}
```

this.model refers to the instance variable.
model refers to the parameter.

The toString Method

• Returns a String representing the data of an object
• Client can call toString explicitly by coding the method call.
• Client can call toString implicitly by using an object reference where a String is expected.
• Example client code:
  ```java
  Auto compact = new Auto();
  // explicit toString call
  System.out.println(compact.toString());
  // implicit toString call
  System.out.println(compact);
  ```
The **toString** API

<table>
<thead>
<tr>
<th>Return value</th>
<th>Method name and argument list</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>toString( )</td>
<td>returns a String representing the data of an object</td>
</tr>
</tbody>
</table>

**Auto Class toString Method**

```java
public String toString() {
    DecimalFormat gallonsFormat =
    new DecimalFormat( 
        "#0.0"
    );
    return "Model: " + model + 
        "; miles driven: " + milesDriven + 
        "; gallons of gas: " + gallonsOfGas ;
}
```

The **equals** Method

- Determines if the data in another object is equal to the data in this object

<table>
<thead>
<tr>
<th>Return value</th>
<th>Method name and argument list</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>equals( Object obj )</td>
<td>returns true if the data in the Object obj is the same as in this object; false otherwise.</td>
</tr>
</tbody>
</table>

- Example client code using *Auto* references `auto1` and `auto2`:
  ```java
  if ( auto1.equals( auto2 ) )
      System.out.println( "auto1 equals auto2" );
  ```

**Auto Class equals Method**

```java
public boolean equals( Auto autoA ) {
    if ( model.equals( autoA.model )
        && milesDriven == autoA.milesDriven
        && Math.abs( gallonsOfGas - autoA.gallonsOfGas ) < 0.0001 )
        return true;
    else
        return false;
}
```

- See Examples 7.10 Auto.java & 7.11 AutoClient.java
**static Variables**

- Also called *class variables*
- One copy of a *static* variable is created per *class*
- *static* variables are not associated with an *object*
- *static* constants are often declared as *public*
- To define a *static* variable, include the keyword *static* in its definition:
  
  Syntax:
  
  ```java
  accessSpecifier static dataType variableName;
  ```

  Example:
  
  ```java
  public static int countAutos = 0;
  ```

**static Methods**

- Also called *class methods*
- Often defined to access and change *static* variables
- *static* methods cannot access instance variables:
  - *static* methods are associated with the *class*, not with any *object*.
  - *static* methods can be called before any *object* is instantiated, so it is possible that there will be no instance variables to access.

**Rules for static and Non-static Methods**

<table>
<thead>
<tr>
<th>Access instance variables?</th>
<th>static Method</th>
<th>Non-static Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access static class variables?</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Call static class methods?</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Call non-static instance methods?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Use the object reference <em>this</em>?</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

* See Examples 7.12 and 7.13

**Reusable Graphical Objects**

- In Chapter 4, the applet code and the Astronaut were *tightly coupled*: we couldn't draw the Astronaut without running the applet.
- To separate the Astronaut from the applet, we can define an *Astronaut* class.
  - The starting x and y values become instance variables, along with a new scaling factor.
  - We move the code for drawing the Astronaut into a *draw* method.
- The applet instantiates an *Astronaut* object in *init* and calls *draw* from the *paint* method.
- *See Examples 7.14, 7.15, and 7.16.*
enum Types

- Special class definition designed to increase the readability of code
- Allows you to define a set of objects that apply names to ordered sets
- Examples of ordered sets:
  - Days of the week
  - Months of the year
  - Playing cards

enum

- Built into java.lang (no import statement needed)
- Syntax:
  ```java
enum EnumName { obj1, obj2,.. objn };
```
- Example
  ```java
enum Days { Sun, Mon, Tue, Wed, Thurs, Fri, Sat };
```
- When this statement is executed A constant object is instantiated for each name in the list. Thus, each name is a reference to an object of type Days

Using an enum Object

- Referring to an enum object reference
  - Syntax:
    ```java
    EnumType.enumObject
    ```
  - Example:
    ```java
    Days.Mon
    ```
- Declaring an object reference of an enum type
  - Syntax:
    ```java
    EnumType referenceName
    ```
  - Example:
    ```java
    Days d; // d is null initially
d = Days.Thurs;
    ```

Useful enum Methods

<table>
<thead>
<tr>
<th>Return value</th>
<th>Method name and argument list</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>compareTo(Enum eObj)</td>
</tr>
<tr>
<td></td>
<td>compares two enum objects and returns a negative number if this object is less than the argument, a positive number if this object is greater than the argument, and 0 if the two objects are the same.</td>
</tr>
<tr>
<td>int</td>
<td>ordinal()</td>
</tr>
<tr>
<td></td>
<td>returns the numeric value of the enum object. By default, the value of the first object in the list is 0, the value of the second object is 1, and so on.</td>
</tr>
</tbody>
</table>
More Useful *enum* Methods

<table>
<thead>
<tr>
<th>Return value</th>
<th>Method name and argument list</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>equals( Enum eObj )</td>
</tr>
<tr>
<td></td>
<td>returns true if this object is equal to the argument eObj; returns false otherwise.</td>
</tr>
<tr>
<td>String</td>
<td>toString( )</td>
</tr>
<tr>
<td></td>
<td>returns the name of the <em>enum</em> constant</td>
</tr>
<tr>
<td>Enum</td>
<td>valueOf( String enumName )</td>
</tr>
<tr>
<td></td>
<td>Convert a string to an object</td>
</tr>
<tr>
<td></td>
<td>static method that returns the <em>enum</em> object whose name is the same as the String argument enumName.</td>
</tr>
</tbody>
</table>

- See Example 7.17 EnumDemo.java

Using *enum* Objects with *switch*

- Using *enum* objects for *case* constants makes the code more readable.
- Use the *enum* object reference without the *enum* type
  - Example:
    ```java
    case Fri:
    ```
- See Example 7.18 DailySpecials.java

Creating Packages

- A *package* is a collection of related classes that can be imported into a program.
- Packages allow reuse of classes without needing the class in the same directory as the other source files.
- To include a class in a package, precede the class definition with the *package* statement:
  ```java
  package packageName;
  ```

A Reusable Class

- For example, we can create a class that provides type-safe reading of input from the console that can be reused by our programs.
- We will name this class *ConsoleIn.java*
- See Example 7.20
Naming Packages

- To avoid name collisions, which can occur when multiple programmers define packages, we use this naming convention:
  - Use the reverse of the domain name, excluding "www".
- For example, for a domain name:
  www.jbpub.com
  the package name would begin with:
  com.jbpub
  then add the package name:
  com.jbpub.af

Create the Directory Structure

- For the package com.jbpub.af, we create three directories and place ConsoleIn.java into the af directory and compile it:

Modify the CLASSPATH

- The CLASSPATH environment variable tells the compiler where to look for packages.
- Set the CLASSPATH to include the directory in which you created the com directory for your package.
- On Windows, if com is created in My Documents, the CLASSPATH might be:
  .\Documents and Settings\User\My Documents
- On Linux, if com is created in myClasses in your home directory, the CLASSPATH might be:
  ./usr/local/java/jre/lib/\bin/home/user/myClasses

Client Use of Package

- To reuse the classes in a package, use the import statement.
  import com.jbpub.af.ConsoleIn;
- See Example 7.21 ConsoleInClient.java
Javadoc Documentation

- The Java class library documentation on Sun's Web site (www.java.sun.com) helps us learn how to instantiate objects and call methods for the classes.
- This documentation was generated using Javadoc, a tool provided in the Java Software Development Toolkit (SDK).
- We can also use Javadoc to generate Web pages that provide documentation on our class’s fields and methods.

To Use Javadoc

- We need to add Javadoc comments and special tags to our classes.
- Javadoc comments begin with /** and end with */ (Note that this is similar to a Java block comment, but with an extra * in the opening syntax.)
- Example:

```
/**
 * Auto class
 * Anderson, Franceschi
 */
```

Block Tags

- Identify parameters and return values
- HTML tags can be used in the descriptions
  - For example, <BR> to insert a new line

<table>
<thead>
<tr>
<th>Tag</th>
<th>Common syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>@param</td>
<td>@param variableName description</td>
</tr>
<tr>
<td>@return</td>
<td>@return description</td>
</tr>
</tbody>
</table>

Sample equals Method Documentation

```
/**
 * equals method:<BR>
 * Compares the fields of two Auto objects
 * @param a1 another Auto object
 * @return a boolean, true if this object
 * has the same field values as the parameter a1
 */
public boolean equals( Auto a1 )
{
    return ( model.equals( a1.model ) &&
             milesDriven == a1.milesDriven &&
             Math.abs( gallonsOfGas - a1.gallonsOfGas ) < 0.001 );
```
Executing Javadoc

- `javadoc.exe` is located in the `bin` directory of the Java SDK
- To generate documentation for a class:
  ```
  javadoc Class.java
  ```
  Example:
  ```
  javadoc Auto.java
  ```
- To generate documentation for all classes in a directory:
  ```
  javadoc *.java
  ```
- See Example 7.22