Programming is about Naming and Manipulating Information

Objectives

- *Understand the meaning of words like program, object, class, and so on*
- Understand the role of naming in different components of a program
- Create and initialize objects using the ***new*** keyword and initialize those objects by putting in important information at creation time.
- Send a message to objects asking them to do something using a series of Java statements.

Object-oriented programs

- Java is an object-oriented programming language. Object-oriented programs consist of a multitude of objects, each of which represents a set of related concepts. These objects communicate with each other to carry out the tasks set forth by the programmer. This is similar to assigning tasks to multiple people who work on a large problem together (such as the people involved in running a restaurant).
- To create an object, we ask the class that defines that object to create it. Each object keeps a reference to the class that created it. That class is like a cookie cutter: It knows how much space is required to store the object, and many such objects can be created from it. We ask a class to create an object using the keyword ***new*** className.

- Class
  - (definition) A class holds all of the information defining how an object should work.
  - A class is like a cookie cutter that can make many objects.
- Instance
  - (definition) An instance of a class is an object created from that class. To continue our cookie cutter metaphor, an instance is like a specific cookie.

Examining your First Application

- Program Organization
  - Every Java program is a class definition.
  - A class definition is a blueprint that is used to construct an object; in the case of an applet, you may want to think of that object as a "program object."
  - Not every Object-Oriented language works like this. In C++, for instance, every program consists of a main() function that may, or may not create different kinds of objects. In Java, there is no separate program entity; there are only classes and object. The basic structure for every Java program looks like this:

- The basic structure for every Java program looks like this:

```
… some stuff
public class SomeClassName {

    … some more stuff

}
```

- The file name for this program must be SomeClassName.java and the file can contain only one public class definition. (You can define other classes in the same file, but they cannot be public classes.)
- Note that this is different than many other programming languages where the program name and the name of the file are not related. In Java, the name of the file and the name of the program must be identical (including case). *The file name and the public class it contains must match exactly.*

The Class Header

- A class definition has two parts: a header and a body.
  - The header "describes" or "declares" the class for the compiler,
  - The body, [the portion of the class in braces { }], is where the attributes and methods for the class will be defined.
- Let's start by examining the header for MyFirstProgram, and then we'll come back to its body. The MyFirstProgram class header is written as:

  public class MyFirstProgram {
- The first part of this is the declaration *public class*. This simply tells the compiler:

  "Hey! a class definition is coming up!" so the compiler knows how to interpret the words that follow.

- Both public and class are two of the fifty-or-so Java keywords.
- *Keywords* [also sometimes called *reserved words*] are the built-in vocabulary of a language.
- In Java:
  - Keywords are always *lowercase*
  - You CANNOT use keywords as identifiers [the names you supply when you create classes, methods, and attributes]
  - You must spell the keywords correctly and use them in the correct context. If you use a keyword inappropriately, or if you misspell it, you will generate a syntax error: a violation of Java's internal rules of grammar.

Name of the Class

- When you write the declaration public class, this puts that Java compiler into such a state of anticipation, that it won't be satisfied unless you tell it exactly what class is about to be described. Thus, the next part of the class header must be the class name. Here we've named our class MyFirstProgram .
- While public and class are keywords, MyFirstProgram is a different kind of beast; MyFirstProgram is an *identifier*. An identifier is a word that you make up to name classes, attributes, methods, and variables.
- You can use any name you like, as long as you follow a few simple rules:
  - You can use letters, digits, the currency symbol [$] and the underscore character in your name.
  - Your identifier cannot start with a digit.
  - You must remember that identifiers are case sensitive . Unlike keywords, you may use uppercase as well as lowercase, but the identifiers cat, Cat, and CAT are not the same!

- If you've programmed in another language, such as C, Pascal, or Visual Basic, these rules don't seem too different. [They're not the same, of course. C doesn't allow the $ character, and Pascal identifiers are not case sensitive.] The biggest difference [when it comes to identifiers] between Java and these other languages is not readily apparent, but pops to the surface when you ask the question, *"What's a character?"*
- Unlike these other languages Java isn't limited to using the ASCII character set [the most widely used computer-character encoding scheme]. As mentioned before, it uses Unicode, the international character set that provides some 65,000 different characters.
- So, although Java identifiers are limited to characters and digits, those characters and digits can be any valid Unicode character or digit.

Naming Conventions

- In programming, as in dining, what is legal is not always appropriate. While it is usually not illegal to eat with your hands, it's often inappropriate, and so most of us obey the social conventions.
- Much like social conventions, identifier naming conventions allow you to infer a great deal about an object by the way it is named. Also like social conventions, identifier naming conventions can become prissy and pedantic, ignoring the more important considerations of clarity and simplicity.

- It is always more important to make sure that your variable names impart meaning to your audience than that they adhere to some rigid naming rule. To that end, you should follow the "golden rules" that are true in any programming language:
  - Try to make your names understandable
  - Use names that accurately describe what your identifier does. For instance, the variable names:

    quarterlySales = quarterlySales + monthlySales;
  - are much easier to understand than

    x = x + x2;

Recommendations

- Before we get to the normal conventions, I should mention two other recommendations. Java allows you to begin a name with an underscore as well as a dollar sign ($), but you should avoid doing so. Java itself uses the dollar sign to create names for "inner classes". To avoid some unusual and hard to find bugs, you should avoid using the dollar sign ($) in your names at all.
- It is also legal to start an identifier with an underscore, but you should avoid that as well. It is just too easy to miss a leading underscore when you are reading code, and if you use them you'll spend many more fruitless hours staring at code that "should" work, but doesn't for some reason.

Reminder

- Here are the more-or-less agreed upon naming standards in the Java world:
  - Class names: CapitalizeEveryWord
  - Method and field names: startLowThenCaps
  - Constants: CAPS_WITH_UNDERSCORES

The Class Body

- Now that we've taken care of the class header, we're ready to look at the class body, where you'll define the attributes and methods that make up the class. We'll start by looking at some general style and punctuation issues.
- In Java, braces {} are used to mark the beginning and end of the class body. Braces are also used to delimit the body of a method, as well as the bodies of loops and selection statements. Just think of braces as markers that say:
  - { "this is the beginning of a section"
  - } "this is the end of a section"
- In Java, such a delimited "section" is called a *block*.

- Braces, along with *indentation*, allow you to quickly see and understand the general structure or "shape" of a program. To facilitate this, every time you use a set of braces, the "stuff" inside the braces should be indented as well.
- There are at least two styles of indenting you can use with your braces. Which one you select is a matter of taste. You should be consistent, however. If you use one style in one part of your code, and the other style in another part of your code, it makes your programs harder to read.
- Below we'll cover the two most popular styles: Classic Indenting and Vertical-style Indenting.

Classic Indenting

- In *classic indentation*, [also called C-style indentation], the "begin" brace is placed on the same line as the block header, while the "end" brace is lined up with the header.
- Everything else inside the braces is indented like this:

```
public class MyFirstProgram {
   // Define your attributes here
   // Define your methods here
}
```

- The advantage of classic style indentation is that you can see more lines of code on your screen at once.

Vertical-style Indenting

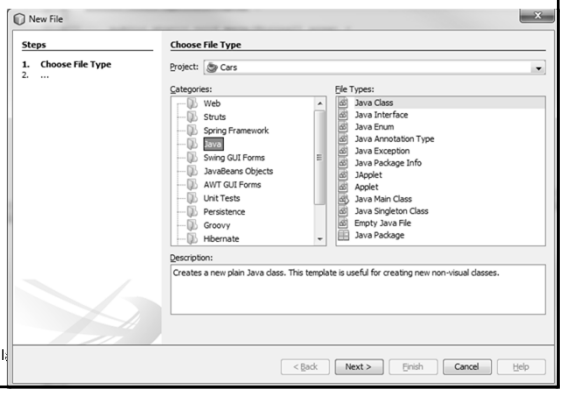
- In vertical-style indenting, the opening brace is placed on a line of its own instead of on the line that contains the block header.
- The advantage of vertical-style indenting is that it makes it easy to match all of the begin-end braces in your program, something that is difficult to do using classic indentation. As you might expect, each opening brace must be matched with a closing brace, otherwise your program will not compile.
- Here's an example of vertical-style indentation:

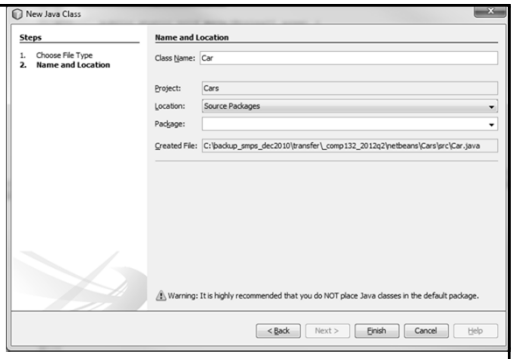
```
public class MyFirstProgram
{
   // Define your attributes here
   // Define your methods here
}
```

- How much should the code inside your braces be indented? Probably between 3 and 4 spaces.

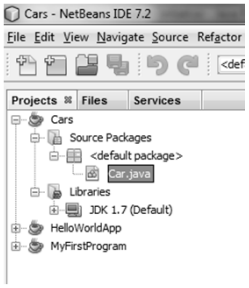
Example

- We are going to walk through the process of creating a simple class called Cars and how Objects work.
- First, create a new project in Netbeans and call it Cars. This time, uncheck the Create Main Class box. You will have NO source files because we told Netbeans to not include the main method.
- Under the File menu, select New File.
- Make sure the Project Line says Cars
- Select Java
- Select Java Class
- Hit Next





- Name your class Car
- Hit Finish.
- Ignore the warning.
- When you are done, you should see a file called Car.java under the <default package>



Car.java

- Our class has nothing in it other than some default comments and our class header.
- Before we continue, what are some things we know about cars? Well, there are many that I am sure we can think of but let us keep our class simple.
- We will use the following information:
  - Attributes:
    - year, make,
  - Functionality:
    - running

Creating Objects

- The first step in creating a new class is to define attributes.
- In Java this is done by creating *fields* or *instance* variables.
  - A field is simply a storage location where you'll store each object's state. If you created a Car class, for instance, you might create a field to hold each Car object's color. The actual color--red or blue--is the state of each of the individual cars. Each field you define can hold only one particular type of thing. With your Car class, you couldn't use the color field to hold information about each Car's gas mileage.

- In Car.java we are going to create our attributes and determine if our car is running or not. Copy the following code into Car.java.
- This is what the whole class will look like:

```
public class Car {
   public int year;
   public String make;
   public boolean running;
   public void isRunning(boolean on) {
      running = on;
   }
}
```



```
public class Car {

   public int year;
   public String make;
   public boolean running;

   public void isRunning(boolean on) {
      running = on;
   }
}
```

Attribute Definition Syntax

- When you create a field, you define it by explicitly describing three things:
  - What kind of thing will be stored in the field
  - What name you want to use for the field
  - What initial value the field should have
- Let's walk through these three easy steps, one-by-one:

- Step 1: Start with a class or type
  - The first step, you recall is to describe what kind of thing will be stored in the field. Thus, the first part of any field definition is a class or type name like this:
  - Ex: Car …
- Step 2: Give your field a name
  - You can't send an object a message if you don't know its name. You can make up any name you like, as long as you obey the rules for naming identifiers, like this:
  - Ex: Car ford
- Step 3: Assign a value to your field
  - To store a value in your new field you use the assignment operator [=]. The assignment operator copies the value on its right, and stores it in the object on its left.
  - Ex: Car ford = <make Car here>

Constructing An Object

- At this point, we're kind of in a quandry. How do we create an object? What do we put on the right-hand-side of the assignment operator?
- If our fields were numbers or characters, for instance, we'd expect to do something like this:
  - int someNumber = 27;
- Since our fields are objects, however, we need to learn how to construct an object; we can't just use a literal value like 27.
- Fortunately, constructing an object is easy. To construct an object--any object--you just use a constructor, along with the keyword new.
- If a class represents a blueprint that describes the attributes and methods of an object, then the constructor is the factory that creates the goods. To make things even easier, a constructor always has the same name as the thing you're trying to create. Now we can finish what we started:
  - Car ford = new Car();

- The first thing you should notice is that every definition ends in a semicolon. Although it was mentioned before, I should reiterate that every Java statement ends with a semicolon. If you omit the semicolons in these statements, the code will not compile.
- The second thing you should notice is the parentheses following the constructor names. Constructors are like methods because they perform an action. In Java, every name that performs an action must be followed by parentheses.
- Look again at the code in the example. Do you notice that the word Car appears twice? The first time, the word is not followed by parentheses, so you are not telling the compiler to do anything; the second time, the word is followed by parentheses. This tells the compiler to perform the code contained in the Car constructor.
- Finally, if the constructors has a value inside its parentheses. These values are called arguments. Arguments allow a constructor--or any other method--to customize its operation. That's how the same Car constructor can be used to turn out red and blue cars .

Attribute Recap

- You must define every attribute like this:
  - Begin with the class name
  - Name your field, subject to rules and conventions
  - Create a new object using new and a constructor
  - Constructor will have same name as class
  - Customize by passing arguments in parentheses

Creating a new car

- To create a Car, you can type new Car();
- When you type new Car(), you are asking the Car class to create a car object for you. The Car class can create many car objects. Each car that is created is an allocated space in memory where it can store custom information about its make, year, and running
- Although you have successfully caused a car to be created, you don't have any way to refer to it. You have to name this object to make use of it.
- Let's try again with World earth = new World(). Now you have created a variable, called ford that you can use to refer to your new Car object. You can create more cars if you want...
  - Car ford = new Car();
  - Car toyota= new Car();

Using the "new" keyword

- Here's an example of using the new keyword.
- Car ford = new Car (); creates a variable named ford which refers to the object created by the Car class.
- Remember, classes are like object factories: you can create numerous objects from a single class, so that those objects have same characteristics defined by that class.
- For instance, you can do the following:
  1. Car ford = new Car ();
  2. Car toyota = new Car ();
  3. Car VW = new Car ();
- These lines of code will create three objects (ford, toyota , and VW) that are all instances of the Car class.

- This is a very simplified version of a Car class.
- In java, to maintain security and to make sure we don't accidentally overwrite information we do information hiding.
- Information hiding is when we only have access to an object by calling a method to set or get the data. Public violates this principle.
- Now that we have a class but that is all. We cannot execute this code because we do not have a way to start the program and we do not have objects.

- In the Cars package, create a new java file and name it ParkingLot.
- This will be the file where we create Car objects and run the program.
- Enter the following code into the ParkingLot class:

```
public static void main(String[] args) {
    Car ford = new Car();
    ford.year = 1967;
    ford.make = "Mustang";
    ford.isRunning(true);
    System.out.println("Make: " + ford.make + " year: " + ford.year
        + " is running: " + ford.running);
}
```


- Hopefully, you will not have any errors.
- Try running the code. The first time it will ask you which class to run, make sure you select ParkingLot.
- Your output should look like this:

```
Make: Mustang year: 1967 is running: true
```

- You sent a message to the isRunning method.
- You told us that the car is running by passing “true” into the parameter.
- We are going to come back to this shortly.

Summary

- To summarize, we have discussed the importance of naming in creating human-readable connections between the code and data stored in a computer and the programs that a person is writing. We have discussed the importance of objects in object-oriented programs.
- We also discussed how objects are instantiated (using the syntax new **Name()**). We explored the difference between a *class*, which defines what an object will do, and the objects it can create.
- Finally, we discussed how to use a variable name to refer to an object we created (Car ford = new Car())
- You might be confused about all of this. The main point to take out of this is that you can create many objects from one class with different attributes.