Ellucian

Java and OO Overview

March 21, 2012
www.ellucian.com

Course Overview

- **This is a 3-day course that provides an overview of Java and Object-Oriented Programming**

- **The course consists of lecture and hands-on labs to reinforce concepts**

- **Its primary purpose is to provide enough background so you will succeed in the subsequent Groovy/Grails course**
  - It's **not** a full-scale Java programming course

Course Objectives

- **Understand the Java environment and its importance**

- **Learn how to write basic-to-intermediate Java code**
  - Including how to organize code into classes

- **Learn the fundamentals of OOP**

- **Learn and employ best practices**

Target Audience and Prerequisites

- **This course is for developers that need a basic understanding of Java and OO**
  - To succeed in the subsequent Groovy/Grails and Banner 9 training
    - Banner 9 uses Groovy and Grails
    - Therefore, some Java/OO knowledge is needed first
  - Sometimes development managers take the course, also

- **Course prerequisites:**
  - Familiarity with programming concepts
  - Experience with Java/OO helpful, but not required

Course Agenda

End of Section

- **This slide is intentionally devoid of any useful content**

Ellucian

Section 1
Java Environment

Architecture
Development Cycle and JDK

Section Objectives

- Understand how Java is both a language and a platform
- Explain portability and how Java achieves it
- Understand the Java development and runtime lifecycle
- Become familiar with the Java Development Kit and some of its tools
- Write your first Java program

Ellucian

Java Environment Architecture

Architecture

Development Cycle and JDK

Java Is a Language

- **Java is a modern, strongly-typed, OO programming language**
  - Invented by Sun Microsystems in 1995
  - Oracle has acquired Sun

- **Built-in support for:**
  - Networking and database access
  - GUI and internationalization
  - Multithreading
  - Security, error handling, and much more

- **Syntax based on C/C++**
  - But some things from C++ were deliberately left out(!)

Java Is a Platform

- ***Java Virtual Machine (JVM)*** **executes the code**

- ***Java Runtime Environment (JRE)*** **includes the JVM and other runtime facilities outside the JVM**

- ***Java Core API***
  - ***Thousands*** of built-in classes to help you
  - The "built-in support" on the previous page is mostly done via these
  - Getting to know them and how to use them is one of your challenges
    - But once you know OO and how to read an API, you're there

Java Is Standardized and Portable

- ***Standardized*** **– Sun (with input from others) specifies the language and its Core API**
  - Sun was once the owner, but Java is now open source and specified via the Java Community Process (JCP)

- ***Portable*** **– runs on any platform (without recompiling)**
  - Sun's motto: "Write Once Run Anywhere"

- **But there's lots of different computing platforms out there...**
  - Unix (different flavors), Windows, MacOS, MVS, etc.
  - So how does Java accomplish this?

- **Code runs on the JVM, *not* directly on the operating system**
  - The JVM, though, does run on the target operating system

Java High-Level Architecture

HelloWorld.java	source code	standardized
↓	↓	
javac	compiler	vendor-provided (Sun, IBM, others)
↓	↓	
HelloWorld.class	bytecode	standardized

HelloWorld.class	bytecode loaded into JVM
Java Core API + Java Runtime Environment (JVM + other services) — **java**	vendor-provided JVM translates bytecode into executable form for the target operating system
target operating system	OS executes instructions

Ellucian

Java Environment Development Cycle and JDK

Architecture

Development Cycle and JDK

Java Development Cycle

1. Write source file with any text editor

- *FirstExample.java*
  - Your write the class definition (`class`) in here
  - NOTE: classes are named using **CamelCase**

2. Compile it with ***`javac`***

```
javac FirstExample.java
```

- This will give error(s) or *FirstExample.class*

3. Run it with ***`java`***

```
java FirstExample
```

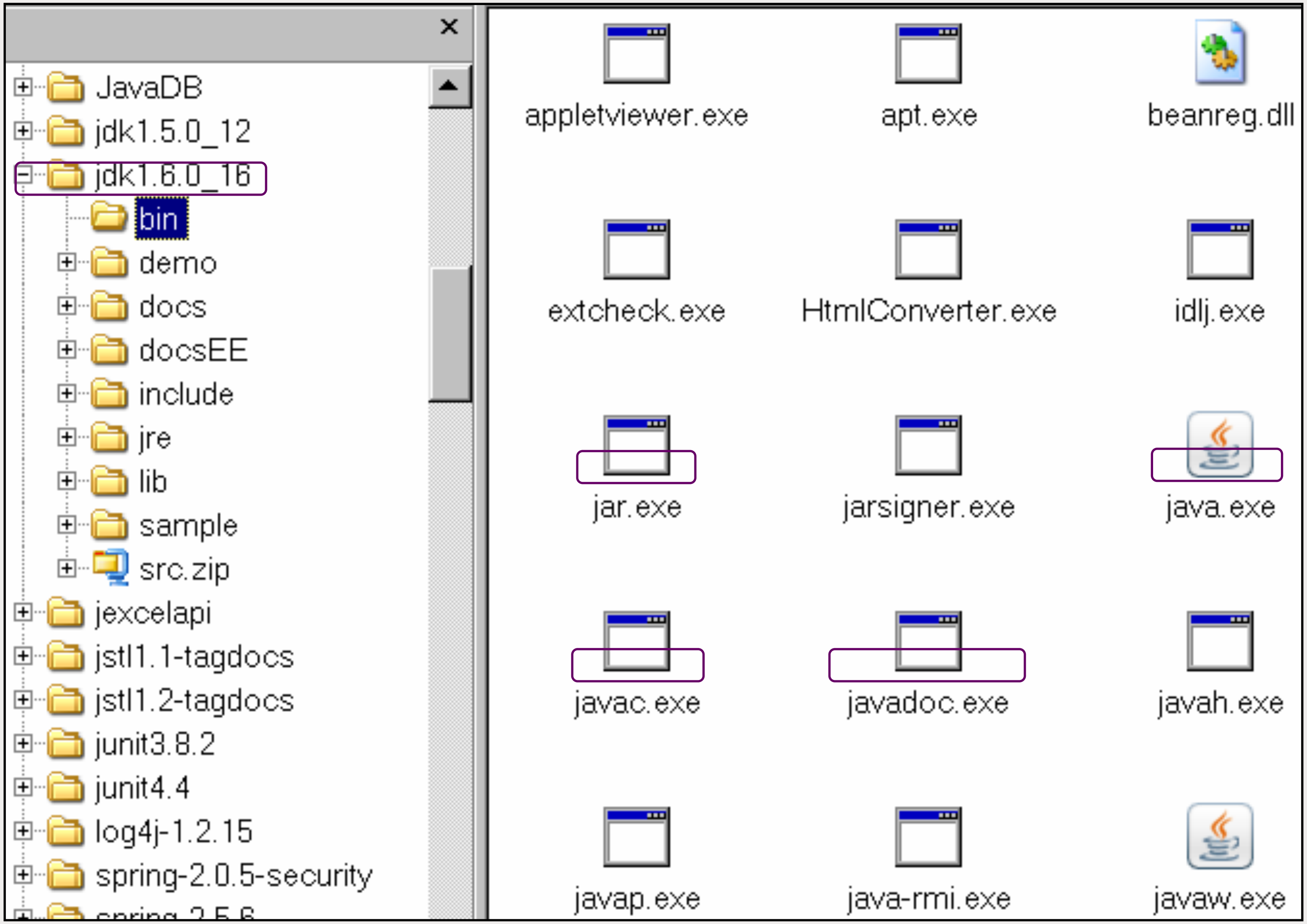
- This launches the JVM and executes *FirstExample.class*
- NOTE: you **don't** include the *.class* extension when running it

Java Developer's Kit (JDK)

- **JRE + command line tools for compiling and other tasks**
  - Free
  - Often obtained from operating system vendor
    - Remember, the VM itself is platform-specific
    - IBM provides a JDK for AIX, Sun provides one for Solaris, etc.
  - Sun, IBM, others provide a JDK for Windows

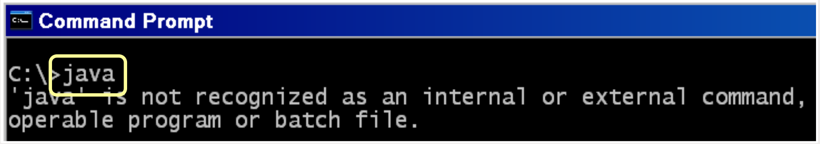
- **There is also a JRE-only package**
  - No compiler or other tools, just the runtime
  - For compiled Java code, all you need is a JRE to run it
  - Vendors that provide Java-based applications usually bundle the JRE with their installer

JDK Illustrated



JDK Post-Installation Tasks

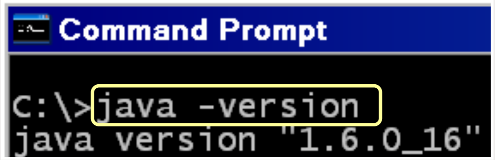
- **KEY: command line tools need to be on your system path**
  - Or you get "command not found"



- **As seen on previous page, the tools are in *<java>\bin***
  - <java> represents where your JDK is installed

1. Set *JAVA_HOME* environment variable (recommended)
  - Point this to your JDK installation directory

2. Include *%JAVA_HOME%\bin* in *PATH* environment variable
  - So you can run the tools from any directory



Diagnosing Problems

- **Common problems:**

`javac YourClass.java`	path
`java YourClass`	path or classpath (more later)

- **If you have multiple JDKs installed on your machine and you're not getting the desired one**
  - Use **`java -version`** to see which one you're getting
  - Check JAVA_HOME and PATH
  - Delete any *java.exe* and *javaw.exe* files located other than in JDK installation directory, especially on Windows (see notes)

Ellucian

Lab 1.1
Setting up the JDK (Class Exercise)

Lab 1.1 - Setting up the JDK

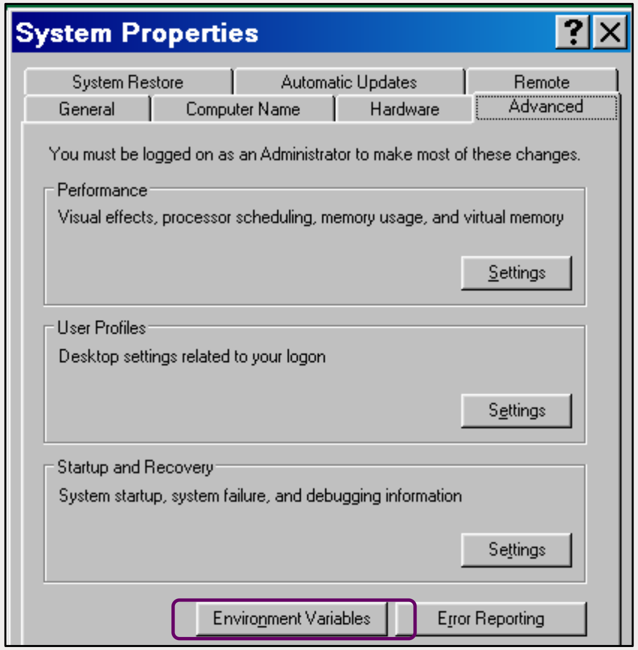
- **Together with the instructor, you will set up the JDK**

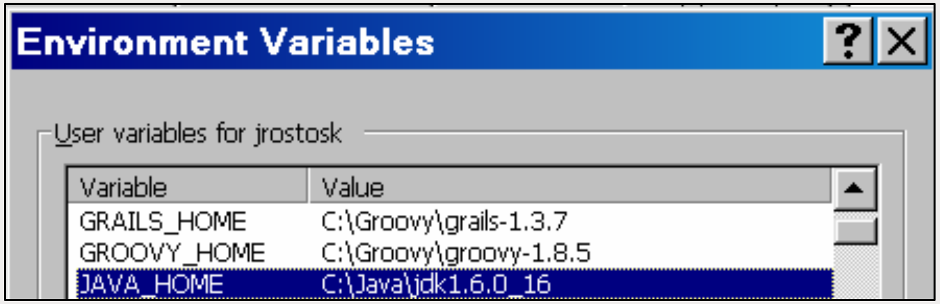
- **NOTE for IDE users:**
  – Even if you use an IDE, understanding this is important
  – The IDE shields you from these details (good), but then you don't understand what's going on underneath (bad)
  – You can start using the IDE in the next section

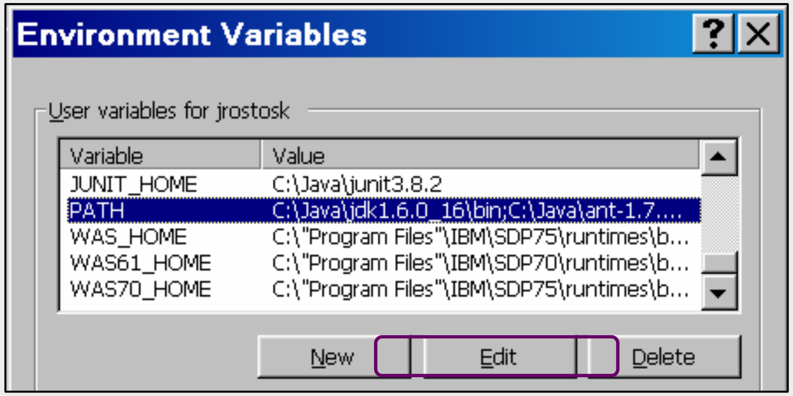
- **Determine where your JDK is installed**
  – IDE users, you must have one, too
    - Bundled with the IDE or it relies on you to provide one
  – My JDK is installed in ___________________
    - NOTE: this is your value for JAVA_HOME

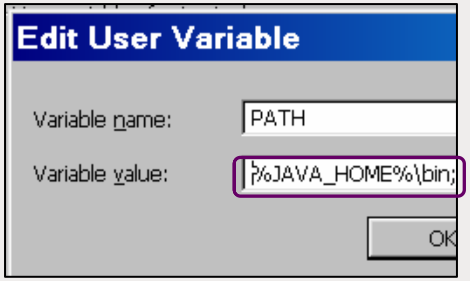
Lab 1.1 - Environment Variables

- **Set JAVA_HOME and PATH**



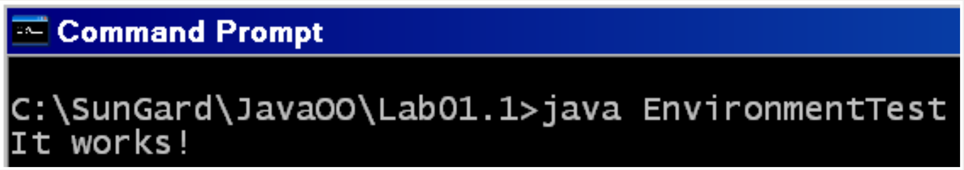






Lab 1.1 - Compiling and Running

- **Extract the *JavaOO_LabSetup.zip* file to *C:***
  - This creates a directory structure under ***C:\Ellucian\JavaOO***
    - This will be the working directory for all the work in this course
- **Open a command window to *C:\Ellucian\JavaOO\Lab01.1***
  - Type `java -version` and make sure it's correct
  - Type `javac`. Did you get the usage banner?
  - There is a Java source file there called *EnvironmentTest.java*
    - Compile it and look for *EnvironmentTest.class*
    - Execute `EnvironmentTest` in the JVM



Java Source Files

- **Filename must have *.java* extension**
  - *HelloWorld.java*

```
class HelloWorld {

}
```

- **Contains 1 or more class definitions**
  - You **can** put more than one class in a source file, **but don't**
- **Max of one *public* class (details later)**
  - Should only be one class in a source file anyway
- **Classname should match filename, *including case***
  - If public class, classname **must** match filename

`main()` Method

- Entry point for every Java application

```
class HelloWorld {

  public static void main(String[] args) {
    // code goes here
  }

}
```

- Signature is important (args variable name is not critical)
- Memorize it

- The `void` means `main()` doesn't return any value
  - To whom? Who calls `main()`, anyway?

- We'll talk later about `public` and `static`

Classpath

- **Review: JDK tools (`java`, `javac`, etc.) are located by looking on your operating system path**
  - Which needs to contain *<java>\bin*

- **At runtime, JVM looks for classes on the *classpath***

- **Much like a system path, classpath is a list of directories and/or JAR files (more on these later)**
  - If you get errors at runtime like "cannot find class," this is a classpath issue

- **Can be specified by *CLASSPATH* environment variable**
- **Can be provided with *`-classpath`* flag**

```
java -classpath C:\Student\Java FirstExample
```

Classpath Rules

- **If you have no classpath, your classpath is implicitly "dot" (`.`) (current directory)**

- **If you set the classpath, you give up the free "dot" (`.`)**
  - If you want the current directory on the classpath, include `.`

    ```
    set CLASSPATH=.;C:\Student\Java
    ```

- **If you have a CLASSPATH environment variable *and* use the `-classpath` flag, `-classpath` overrides**

Ellucian

Lab 1.2
Your First Java Program

Lab 1.2 - Hello World

- **With any text editor, create a new file called *HelloWorld.java***
  - Save it in the ***C:\Ellucian\JavaOO\Lab01.2*** directory
    - **NOTE: you need to manually create this directory**

  - Type the following into it (**exactly** as it appears here):

```
class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello from Java!");
  }
}
```

Lab 1.2 - Compiling and Running

- **Open a command prompt in *C:\Ellucian\JavaOO\Lab01.2***
  - Compile your source file `javac HelloWorld.java`
  - Run it `java HelloWorld`
  - Try running it this way: `java helloworld`

Lab 1.2 - Classpath

- **Now we will experiment with classpath**
  - Make sure you have no classpath

```
Command Prompt
C:\SunGard\JavaOO\Lab01.2>set CLASSPATH
Environment variable CLASSPATH not defined
```

    - Type "`set CLASSPATH`" at the command line
      If you have a value for it, clear it by typing "`set CLASSPATH=`"
    - This means your classpath is now "dot" (`.`) or "current directory"
    - Change to any other directory and try to run `HelloWorld` (fails)
  - Now set the classpath to *C:\Ellucian\JavaOO\Lab01.2*
    - Type "`set CLASSPATH=C:\Ellucian\JavaOO\Lab01.2`"
    - Try to run `HelloWorld` (works) – works from any directory, too

- **Experiment with `-classpath` flag (preferred over env var)**
  Remember that this overrides any CLASSPATH setting
  - Open a **new** command window to any directory, and type
    `java -classpath`
    `C:\Ellucian\JavaOO\Lab01.2 HelloWorld` (all one line)

Lab 1.2 - JAR Files (optional)

- **JAR = *J*ava *AR*chive**
  - Zip file containing Java classes (*.class* files)
  - Created with the JDK ***jar*** tool
  - Have a ***.jar*** extension

- **Open a command window to *C:\Ellucian\JavaOO\Lab01.2***
  - Create *hello.jar* by typing the following:

```
jar -cvf hello.jar HelloWorld.class
```

- ***Delete HelloWorld.class* (we want to use only the JAR)**

- **Run `HelloWorld` by specifying *hello.jar* on the classpath**

```
java -classpath hello.jar HelloWorld
```

Section Review

1. Explain how Java is both a language and a platform.
2. What do we mean by "portability?" How does Java accomplish this?
3. If you compile your source code with a Sun compiler, you have to run it with the Sun JVM. [T/F]
4. What is bytecode?
5. What is the starting point for every Java program?
6. Explain the difference between path and classpath.
7. Java currently has a limitation in that you can only define one class per source file. [T/F]
8. How does the JRE locate classes at runtime?
9. What is a JAR file? What's in it? How do you create one?

End of Section

- **This slide is intentionally devoid of any useful content**

Ellucian

Section 2
Java and OO Fundamentals

Classes and Objects
Data Types, Variables, and Operators
Attributes and Methods
Data Encapsulation
Strings, Arrays, and Wrapper Classes

Section Objectives

- **Understand classes and objects, and how to write classes**
- **Learn the core Java language syntax elements**
- **Understand and use Java primitive data types and variables**
- **Learn the Java operators and how they are used**
- **Write methods and implement data encapsulation**
- **Understand the `String` class, arrays, and wrapper classes**

Ellucian

Java and OO Fundamentals
Classes and Objects

Classes and Objects

Data Types, Variables, and Operators

Attributes and Methods

Data Encapsulation

Strings, Arrays, and Wrapper Classes

What Is a Class?

- **In the real world, you'll often find many individual objects all of the same "kind" or "type"**
  - For example, Automobile, Person, Teacher, Meeting, Planet

- **A *class* defines a *type* of object, and consists of two fundamental things:**
  - **Variables**
  - **Methods**

- **A class *describes* its objects – it tells you *what* you will have *when* you have one**
  - The data (variables) that each object of the class will have
  - The operations (methods) defined on those objects

What's in a Class?

- A class defines a set of ***data*** (variables) along with its ***behaviors*** (methods)

```
class Automobile {

  // VARIABLES
  String makeModel;       // NOTE: String is a class
  String licensePlate;

  // METHODS
  void start() {
    System.out.println("Starting a " + makeModel +
      " with license plate number " + licensePlate);
  }
}
```

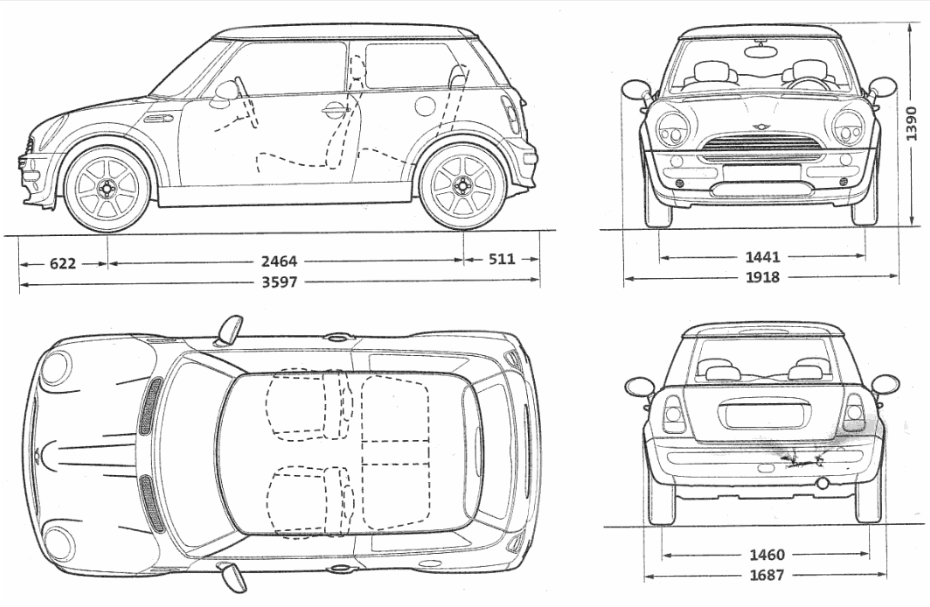
Difference between Class and Object

- A ***class*** is like a "blueprint," from which individual objects of the class are created

- A specific "car" object is an ***instance*** of the `Automobile` class

Automobile class (1)

Automobile objects (many)



Creating Objects

- **Create a new instance of the class `Automobile`**

```
// car is a reference or pointer to an Automobile object
Automobile car = new Automobile();
```

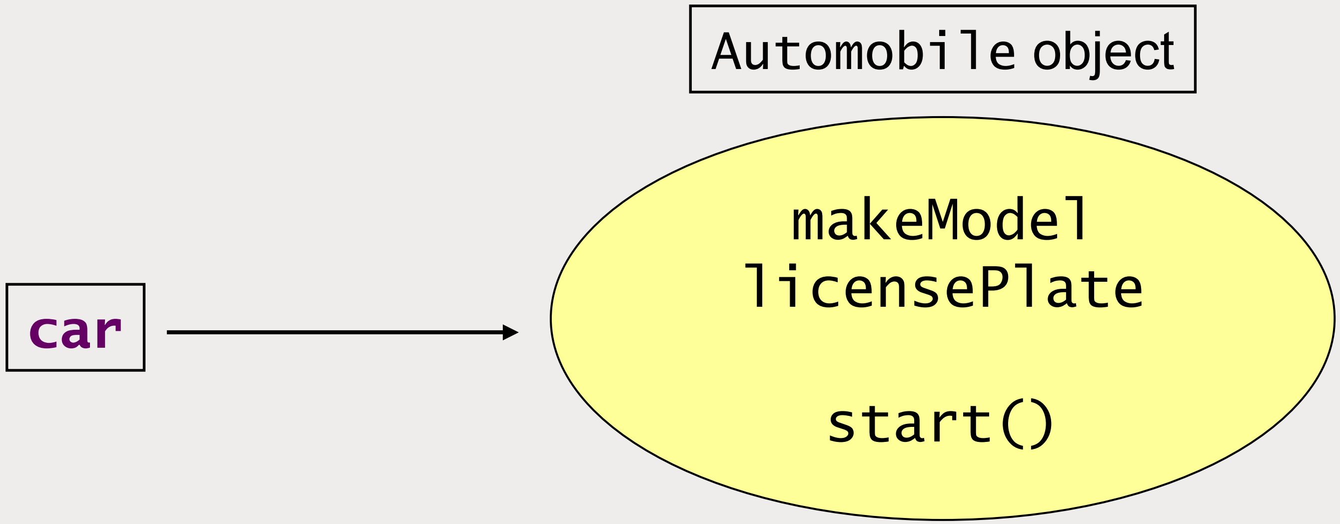
- **What we are actually doing is:**
  - Creating an `Automobile` instance (our object)
  - Creating a variable called **car** that **refers** to the `Automobile` instance
    - "car" is really an address location, e.g., 0x99f775
    - But you never deal with this, only with "car"

- **In OO terms, we are *instantiating* the `Automobile` class**
  - That is, creating an instance (object) of the class

Creating Objects - Illustrated

- The **car** variable below ***references*** or ***points to*** our `Automobile` object

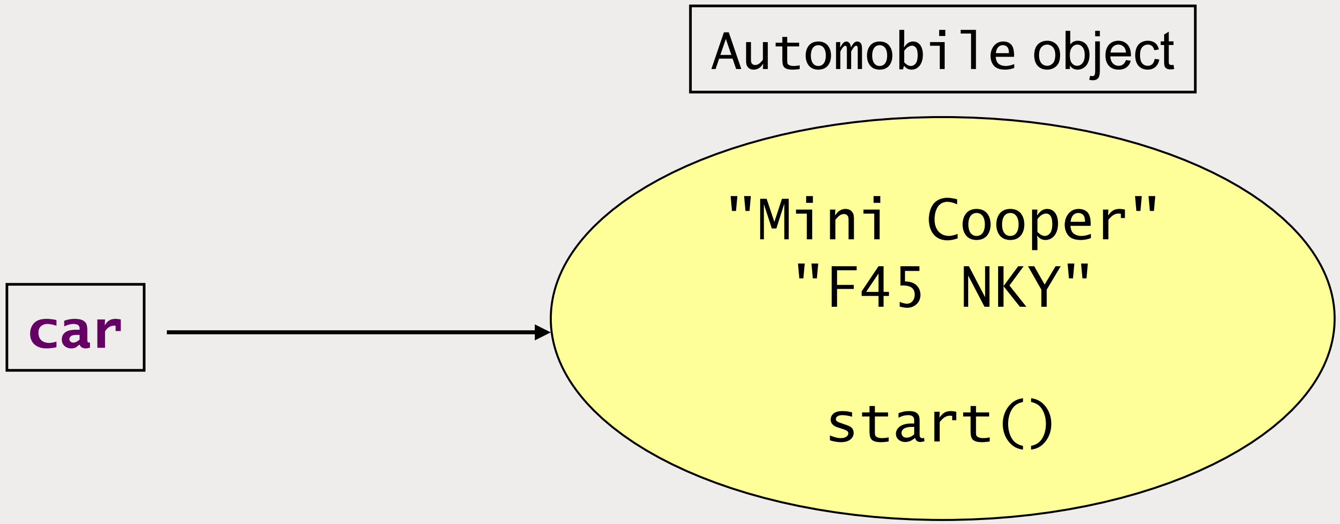
```
Automobile car = new Automobile();
```



Setting the Data in Objects

- **We still need to set the data in our `Automobile` object**
  - We assign string values to each of its variables

```
Automobile car = new Automobile();
car.makeModel = "Mini Cooper";
car.licensePlate = "F45 NKY";
```



- **See notes about setting the data this way**

Invoking Behavior on Objects

- **Execute our `Automobile` object's `start()` method**
  - As before, we use our `car` reference variable

```
Automobile car = new Automobile();
car.makeModel = "Mini Cooper";
car.licensePlate = "F45 NKY";
car.start();
```

```
car.start();
```

- **Which results in this output**

```
Starting a Mini Cooper with license plate number F45 NKY
```

Writing Class Definitions

- **In Java, everything is in classes, i.e., inside**
```
class {
   ...
}
```

- **There should generally be only *one class per source file***
  - And the names should match – `class Student` in source file *Student.java*

- ***JAVA IS CASE SENSITIVE*** ***Java Is Case Sensitive***
  - You must accept this reality
  - There are a few things that can be expressed both in upper and lower case, but not many

Packages

- **Java classes are organized into *packages***

- **A class that not in a package is in the *default package***
  - A package with no name
  - More on packages later

- **The Java Core API library is divided up into packages**
  - **`java.lang`**
  - **`java.net`**
  - **`java.io`**, etc.

Identifiers

- **Names used for classes, variables, and methods**

- **Can include:**
  - Letters, digits, underscore (_), dollar ($)

- **Cannot:**
  - Begin with a digit
  - Be a Java keyword (we'll see these soon)

- ***Java is case sensitive***
  - **`User`** and **`user`** are completely different identifiers

Java Keywords

abstract	continue	for	new	switch
assert (1.4)	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum (5.0)	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp (1.2)	volatile
const*	float	native	super	while

* not used

Java Naming Conventions

- **Not strictly required by the compiler**
  - But pretty much law
- ***Class names*** **begin with a** ***capital letter***
  - `String`, `System`, `Person`, `HelloWorld`, `SecurityManager`
- ***Variable and method names*** **begin with a** ***lower case letter***
  - `name`, `age`, `initialCount`, `numWords`, `bufferSize`
  - `getAge()`, `setAge()`, `println()`, `flushBuffer()`, `maxVal()`
- **In both cases, subsequent "words" in the name are capitalized**

Comments

- **Multi-line or "block"**
  - Begin with `/*` and end with `*/`
- **Single line**
  - Begin with `//`
  - Continues to end of line

```
// single-line comment
/*
 this comment spans
 several lines
*/
```

- **Javadoc documentation comments**
  - Begin with `/**` and end with `*/`
  - Read by **`javadoc`** tool to generate HTML-based API doc

Ellucian

Lab 2.1
Creating a Simple Class (Class Exercise)

Lab 2.1 - Using an IDE

- **We will now switch to using an IDE called *SpringSource Tool Suite (STS)***
  – Based on **Eclipse** – a very popular open source development environment

- **Your instructor will get you started with STS**
  – Launch the tool, set workspace directory to ***C:\Ellucian\JavaOO\workspace***
  – Give an overview of the following Eclipse concepts:
    - Workspace, Workbench, Perspectives, Views, Editors, etc.

- **You should use the *Java* perspective for this course**
  – The instructor will show you how to customize the perspective and save it via **Window → Save Perspective As**

Lab 2.1 - Automobile Class

- **With the instructor guiding, create a Java project called *Lab02.1_Automobile***

- **For now, our classes will not be in packages**
- **The New Class wizard defaults to making classes public**
  - This is okay, it's quite common for classes to be public
  - More about packages and access protection later

- **With the instructor guiding, create an `Automobile` class**
  - Declare two variables, both as `String`s, as in the example
    `makeModel`
    `licensePlate`
  - Write a `start()` method, as in the example
  - See notes

Lab 2.1 - Automobile Client Class

- **With the instructor guiding, create a class called `AutomobileClient`**
  - Write a `main()` method in it
    - Your instructor can show you an IDE shortcut for generating the `main()` method automatically

- **In the `main()` method, create two instances of `Automobile`**
  - Each one will have its own reference variable, e.g., **`car1`**
    - Set each one's `makeModel` and `licensePlate` properties

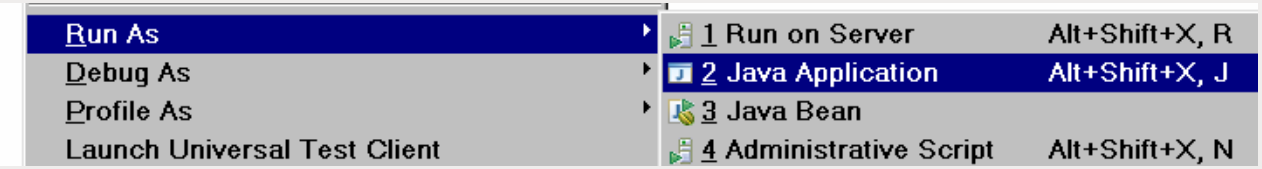
```
Automobile car1 = new Automobile();
car1.makeModel = "Toyota Matrix";
car1.licensePlate = "ABC 123";
```

- **Call each one's `start()` method:**

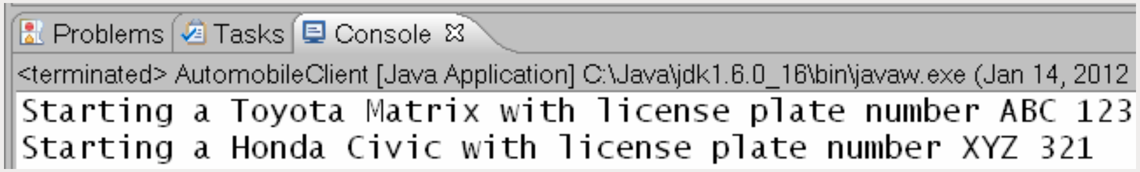
```
car1.start();
```

Lab 2.1 - Running the Client

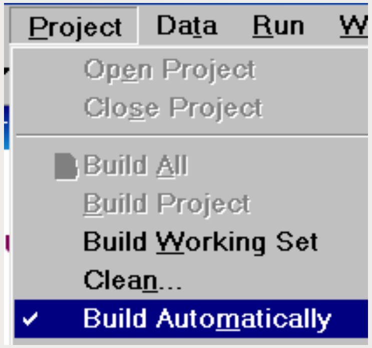
- **Right-click on *AutomobileClient.java***
  - **Run As → Java Application**, or [Alt+Shift+X, J]



- **Console view, with your output, should appear at the bottom**



- **But I never compiled it...**
  - Every save [Ctrl+S] does a compile ("build")
  - If you turn this off, you then use [Ctrl+B] to build



Ellucian

Java and OO Fundamentals
Data Types, Variables, and Operators

Classes and Objects

Data Types, Variables, and Operators

Attributes and Methods

Data Encapsulation

Strings, Arrays, and Wrapper Classes

Data Types in Java

- All data in Java has a type
- Class or interface type (covered later) – `String` is a class
- Primitive types (no class for these)
  - `byte` `short` `int` `long` — integers (`int` is default)
  - `float` `double` — floating point numbers (`double` is default)
  - `char` — single character (stored as 16-bit Unicode)
  - `boolean` — true/false

Primitives - Details

Type	Size (bits)	Range of Values
`byte`	8	-128 to 127
`short`	16	-32768 to 32767
`int`	32	-2147483648 to 2147483647
`long`	64	-9223372036854775808 to 9223372036854775807

- **Generally, only `int` and `long` are used for integers**

- **`float` is 32-bit, `double` is 64-bit**
  - `double` generally used over `float`
  - Most business apps use the `BigDecimal` class (see notes)

- **`String` generally used over char**

Assignment

- **Assignment takes this form:**
  `variable-name = value;`

- **Declaration and assignment together:**
  `variable-type variable-name = value;`

- **Assignment is done from right to left**
  - Value can be a literal, or returned from a method

```
int sum;            // declaration
sum = 3 + 7;        // assignment
```

```
// declaration and assignment together
int size = 9;

int age = p.getAge();  // p is a Person object
```

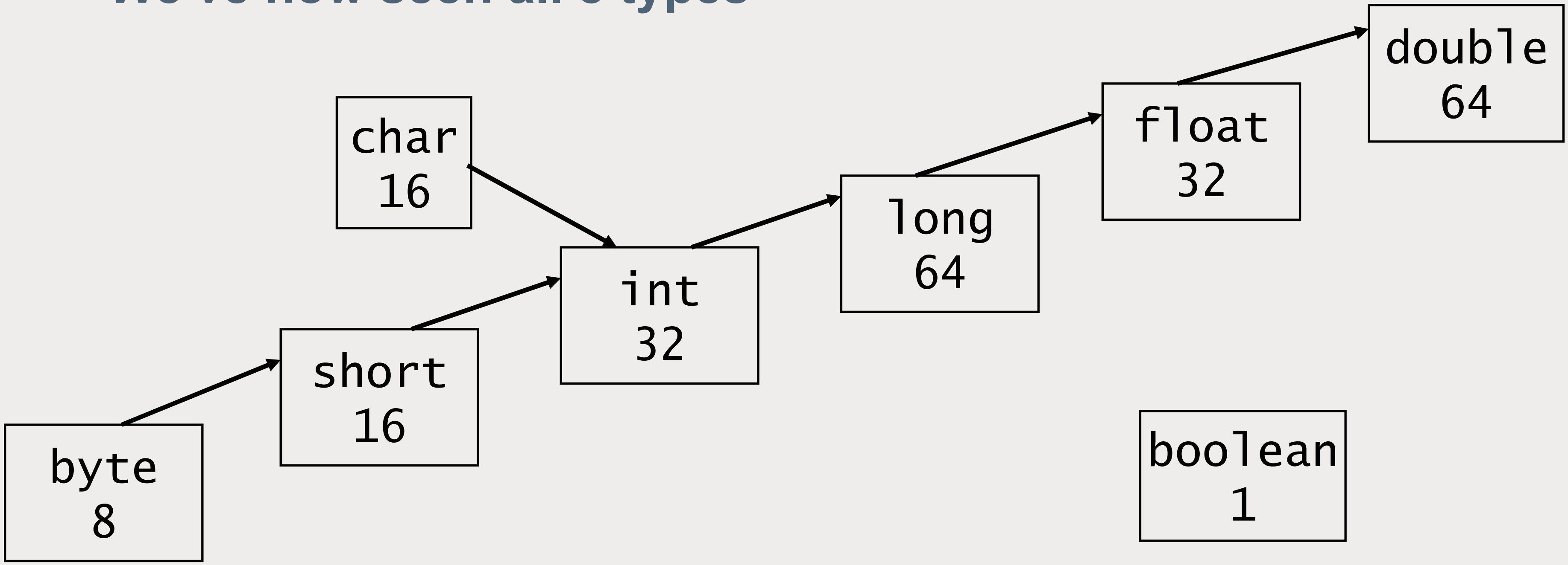
Assignment - Example

```
int x = 3;
long id = 112;
long y = 123456789123L;    // 123456789123 is too big for int

float weight = 1.23F;      // 1.23 is double by default
double pi = 3.14;

boolean isRaining = true;  // no quotes, just true or false

char gender = 'M';         // single quotes, single character only

String name = "Fred";      // double quotes, String is a class name
```

Why Have Primitive Types in Java?

- Avoid overhead associated with objects and references

- We've now seen all 8 types



Primitive Type Conversion and Casting

- "Upcasts" to a larger type are implicit and automatic
- "Downcasts" to a smaller type must be made explicitly
  - Because you might lose data and/or precision

```
float length = 2.1F;
float width = 4.51F;
double area = length * width;     // auto upcast to double

float approxArea = (float) area;  // explicit downcast to float
```

- "Upcasts" or conversions may occur during operations

```
int   + long        // int converted to long, result is long
float + double      // float converted to double, result is double
int   + float       // int converted to float, result is float
```

Ellucian

Lab 2.2
Primitives

Lab 2.2 - Primitives

- **In a new project *Lab02.2_Primitives*, create a class called `Primitives`**
  - As before, class is not in a package, okay for it to be public
  - Write a `main()` method in it (or use a shortcut)

- **In `main()`, declare and initialize some primitive variables**
  - Integers, floating points, character, boolean
  - Also do one for `String` (see notes)

- **For each, print a message that shows its value**
  - Here's an example code snippet (inside the `main()` method):

```
int i = 42;
System.out.println("i is " + i);  // + does concatenation here
```

Lab 2.2 - Local Variables Must Be Initialized

- **Sometimes you want to declare, but not initialize a variable**
  - You might later initialize it in an `if` statement, for example

- **Below all your print statements (still in `main()`), try this:**

```
int x;    // declared but not initialized
System.out.println("x is " + x);
```

- **What's the problem? What is the compiler error message?**

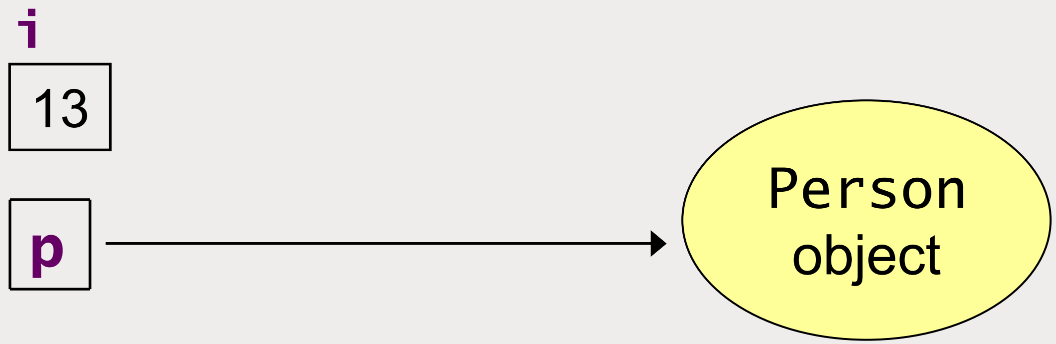
- **This behavior occurs with *local variables***
  - Variables declared and used **within a method**
  - There are other types of variables that behave differently (more later)
- **After you've seen the error, delete the problem code**
  - Or leave it, but assign a value of 0 to the variable

Primitive Variables and Reference Variables

- **Primitive variable – stores a data value directly**
  - Used for the 8 primitive data types we've just seen

- **Reference variable – stores a *reference* to an object**
  - Used for objects, which is everything else

```
int i = 13;   // i is a primitive variable that holds the value 13

Person p = new Person();   // p is a reference or handle to a Person
```



Equality and Object Identity

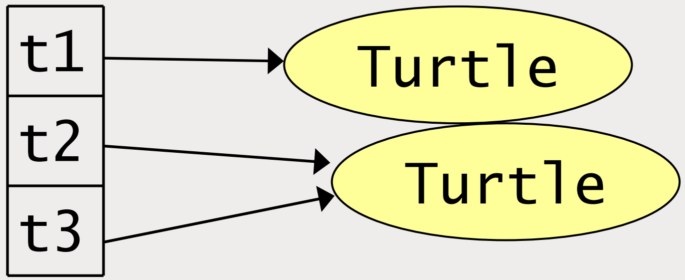
- **With primitives, == tests for *equality***
  - Indicates if two **values** are equal

```
int x = 3;
x == 3  // true
```

- **With reference variables, == tests for *identity***
  - Indicates if two **references** point to the **same physical object**

```
Turtle t1 = new Turtle();   // brand new Turtle
Turtle t2 = new Turtle();   // brand new Turtle

Turtle t3 = t2;             // t3 refers to same Turtle object as t2
```

```
t1 == t2     // false
t3 == t2     // true
```

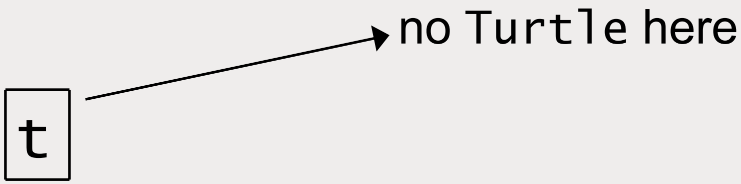


`null`

- **Indicates that a reference variable doesn't point to an object**
- **Can initialize to `null`, can test references for `null`**

```
Turtle t = null;  // no new means no object, t is null
t == null         // true
```

- **Can't call methods if you have a `null` reference**
  - There's no object to execute the code!
  - Results in `NullPointerException` at runtime



Primitives Are Passed by Value (Copy)

```
class PassByValue {

  public static void main(String[] args) {
    int x = 10;
    changeIt(x);
    System.out.println("The value of x is " + x);
  }

  static void changeIt(int n) {
    n = n + 5;  // add 5 to n, then assign that new value to n
  }

}
```

a **copy** of x is sent

```
The value of x is 10    // see notes for how to get the 15 back
```

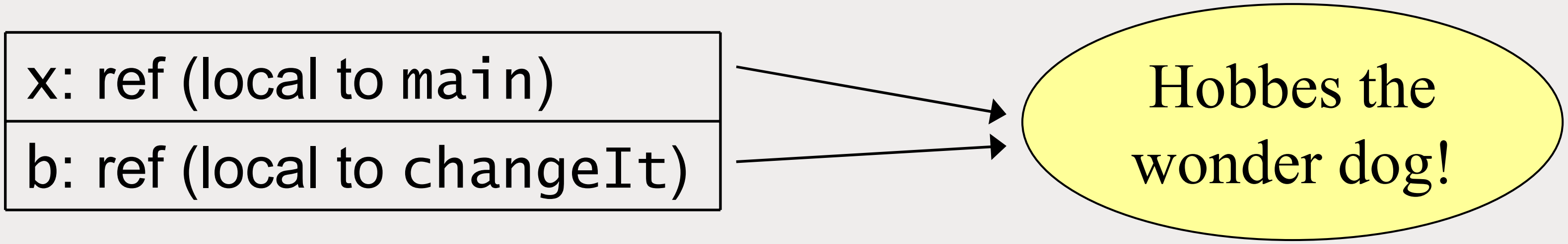
x: 10 (local to main)
n: 15 (local to changeIt)

Objects Are Passed by Reference

```
class PassByReference {

  public static void main(String[] args) {
    StringBuffer x = new StringBuffer("Hobbes");
    changeIt(x);
    System.out.println("The value of x is " + x);
  }

  static void changeIt(StringBuffer b) {
    b.append(" the wonder dog!");
  }

}
```

a copy of the **reference** is sent

```
The value of x is Hobbes the wonder dog!
```



Arithmetic Operators

+ addition
- subtraction
* multiplication
/ division
% modulo (remainder after division)

- As in school, multiplication and division have precedence over addition and subtraction
  - Use () when needed, and for clarity

    4 + 5 * 6 = 34

    4 + (5 * 6) = 34

    (4 + 5) * 6 = 54

Increment and Decrement

++ --

- **Modify value by 1**
- **They have "pre" or "post" behavior**
  - If ++x, increment **before**
  - If x++, increment **after**

```
int x;
int y = 0;
x = y++;    // what is the value of x? ("post")
```

```
int x;
int y = 0;
x = ++y;    // what is the value of x? ("pre")
```

+ Has Two Meanings

- **If two numbers, + *adds* them**

- **If either side is a `String`, + *concatenates* them together**
  - The other side is automatically converted to a `String` if necessary

```
String s = "2";
int r = 2;
System.out.println(r + s);  // what will the output be?
```

- **Called *operator overloading***
  - Only the + operator offers it

Comparison Operators

`>` `>=` `<` `<=` `==` `!=`

- **Operate on numbers and return a boolean: `true` or `false`**

```
int age = 34;
if (age >= 21) {
  ...
}
```

- **Common mistake for beginners:**
  - **You don't use = to test equality in Java, you use ==**
  - Remember, = is for assignment

- **== won't do what you think on `Strings`**
  - It compares the two object references, **not** the string values
  - Use the **`equals()`** method to compare `Strings` (more later)

Boolean Operators

&&   & || | ^   !

AND   OR   XOR   NOT

- **Operate on booleans and return a boolean**

- **&& and || work "short-circuit" style**
  - Evaluate the left side, and only look at the right side if necessary ("short-circuiting" it)
    - Since `false` AND `<anything>` is `false`, don't need to look at `<anything>`
    - Likewise for `true` OR `<anything>`

Ellucian

Lab 2.3
Object References

Lab 2.3 - Object References

- **Open your *Lab02.1_Automobile* project**

- **Create another class called `IdentityTest`**
  - Put a `main()` method in it, and do all your work in `main()`

- **Create two `Automobile` objects and test them for identity using ==**
  - Print a message to the console to see the result

```
Automobile car1 = new Automobile();
Automobile car2 = new Automobile();
System.out.println("car1 == car2: " + (car1 == car2));
```

Lab 2.3 - `null`

- **Still in the `main()` method, create an `Automobile` reference variable, but *do not* point it to a new `Automobile` object**
  - First, try it this way:

```
Automobile auto;
```

- **Test to see if auto points to `null` using ==**

```
Automobile auto;
System.out.println("auto == null: " + (auto == null));
```

  - What's the problem? Have we seen this before?

  - Now declare it this way and test for `null`

```
Automobile auto = null;
```

Ellucian

Java and OO Fundamentals
Attributes and Methods

Classes and Objects
Data Types, Variables, and Operators
Attributes and Methods
Data Encapsulation
Strings, Arrays, and Wrapper Classes

Data Members in a Class

- **Also called *instance variables*, *fields*, *attributes*, *properties***
  - They hold data values for the objects of that class
- ***Each instance has its own set of values***
  - And those values **vary from instance to instance**
    - Hence the name "instance variables"
  - Each object is an instance or copy of the class at runtime – changes in one do not affect the others
- **Person example:**
  - The Person class is defined to have two data members:
    - name (string)
    - age (integer)
  - Each Person object in the room has their own name and age
    - In one instance, we have "Sue" 26; in another instance, "Joe" 48

Default Values for Data Members

- **Data members get initialized with default values**
  - "Zero" for that type

- **Primitive types:**
  - Integers **`0`**
  - Floating points **`0.0`**
  - Booleans **`false`**
- **Class types:** **`null`**

```
// these are equivalent
String name;

String name = null;
```

- **This is *different than local variables* (those declared locally in individual methods)**
  - Remember that they don't get the "free zero" – they must be explicitly initialized before used

Scope and Lifetime of Variables

- **Variables "live" or "exist" only in the braces in which declared**
  - But instance variables are in the outer-most braces
    - They have **class-level scope**
  - They are available to all the methods

- **They are "there" in memory for the lifetime of the object**

```
class Automobile {

  String makeModel;
  String licensePlate;

  void start() {
    System.out.println("Starting a " + makeModel +
      " with license plate number " + licensePlate);
  }
}
```

```
makeModel "Mini Cooper"
licensePlate "F45 NKY"
start() { ... }
```

Methods

- **Objects have functionality – they "do stuff"**
- **This is defined in *methods***
  - Methods "get the work done"
- **Methods break up the object's functionality into distinct, cohesive units**
  - Cohesive means "about one thing"
    - Method should have a tightly defined job, not "do it all here"

Defining Methods

- **Anatomy of a method:**

```
return-type method-name(param-type param-name, ...) {
  // method "body" - implementation code goes here
}
```

- **Return type – what it gives back**
  - `void` if it returns nothing
  - **`return`** keyword to return a value
- **Method name – *camelCase* by convention**
- **Parameters – each one just like a variable declaration**
  - Enclosed in parentheses, separated by comma
- **Code block in { }**

Invoking Methods

```
Turtle t = new Turtle();     // t is a reference to a Turtle object
t.jump();                    // takes no arguments, returns nothing

// using a variable for the 1st argument, a literal for the 2nd
String target = "San Francisco";
String result = t.moveTo(target, 5);
```

```
class Turtle {

  void jump() {
    System.out.println("Turtles can jump");
  }

  String moveTo(String location, int speed) {
    return "Moved to " + location + " at speed " + speed;
  }
}
```

`main()` Method - Revisited

- **`main()` is a somewhat special method**
  - It is the entry point for every Java application
    - The JVM starts your program by invoking `main()`

```
public static void main(String[] args) {
  // ...
}
```

- **When you write a method, you get to pick its signature**
  - The calling code must call it properly

- **You can't pick the signature of `main()`, because we have an agreement with the JVM (which is outside our control)**
- **It will call `public static void main(String[])`**
  - You **must** have this method **exactly** as the JVM is expecting it, or it can't start your program

APIs

- **To use an object you only need to know its *Application Programming Interface* (API)**

- **An API includes a description of:**
  - All available methods and what they do
  - The parameters they take and the values that they return

- **The API is usually given in a special format called *Javadoc***

- **Javadoc for the built-in Java Core API classes may be installed with your JDK or IDE**
  - Readily available on the Web

Javadoc API Documentation

- **From this info, how would you call the `charAt()` method?**

java.lang

Class String

java.lang.Object
  └ *java.lang.String*

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Method Summary	
`char`	*`charAt`*`(int index)` Returns the char value at the specified index.
`int`	*`codePointAt`*`(int index)` Returns the character (Unicode code point) at the specified index.
`int`	*`codePointBefore`*`(int index)` Returns the character (Unicode code point) before the specified index.

Ellucian

Lab 2.4
Attributes and Methods

Lab 2.4 - Writing Methods

- **In a new project *Lab02.4_AttributesMethods*, create a class called `Calculator`**
  - With the instructor guiding, you'll write these methods:
    - `add()`
    - `subtract()`
    - `multiply()`
    - `divide()`
    - `squareRoot()`

- **Each should take input data as `double`, and return a `double`**
  - For `squareRoot()`, delegate to the `Math` class's `sqrt()` method

- **Create `CalculatorClient` class, with `main()` method**
  - With the instructor guiding, test drive a `Calculator` object

Lab 2.4 - Attributes and Methods in a Class

- **Create another class called `Person`**
  - Give it four instance variables
    - **name** `String`
    - **age** `int`
    - **height** `double` (inches)
    - **weight** `double` (pounds)

- **Write a method called `calculateBMI()`**
  - It should take no parameters
  - It should return the weight divided by height

- **Write another method called `eat()`**
  - It should take a `String` parameter called **food**
  - Just print a message to the console, including the person's name and the food (see notes)

Lab 2.4 - Testing the Person

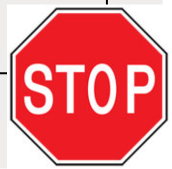
- **Create a class called `PersonClient`**
  - Give it a `main()` method and do all work in `main()`

- **Create a Person object and set all four variables**

```
Person p = new Person();
p.name = "Martin";
p.age = 33;
...
```

- **Call its methods**

```
// calling this method returns a value, which we store in a variable
double bmi = p.calculateBMI();
System.out.println(p.name + " has a BMI of " + bmi);

// nothing is returned from calling this method
p.eat("pizza");
```



Ellucian

Java and OO Fundamentals
Data Encapsulation

Classes and Objects
Data Types, Variables, and Operators
Attributes and Methods
Data Encapsulation
Strings, Arrays, and Wrapper Classes

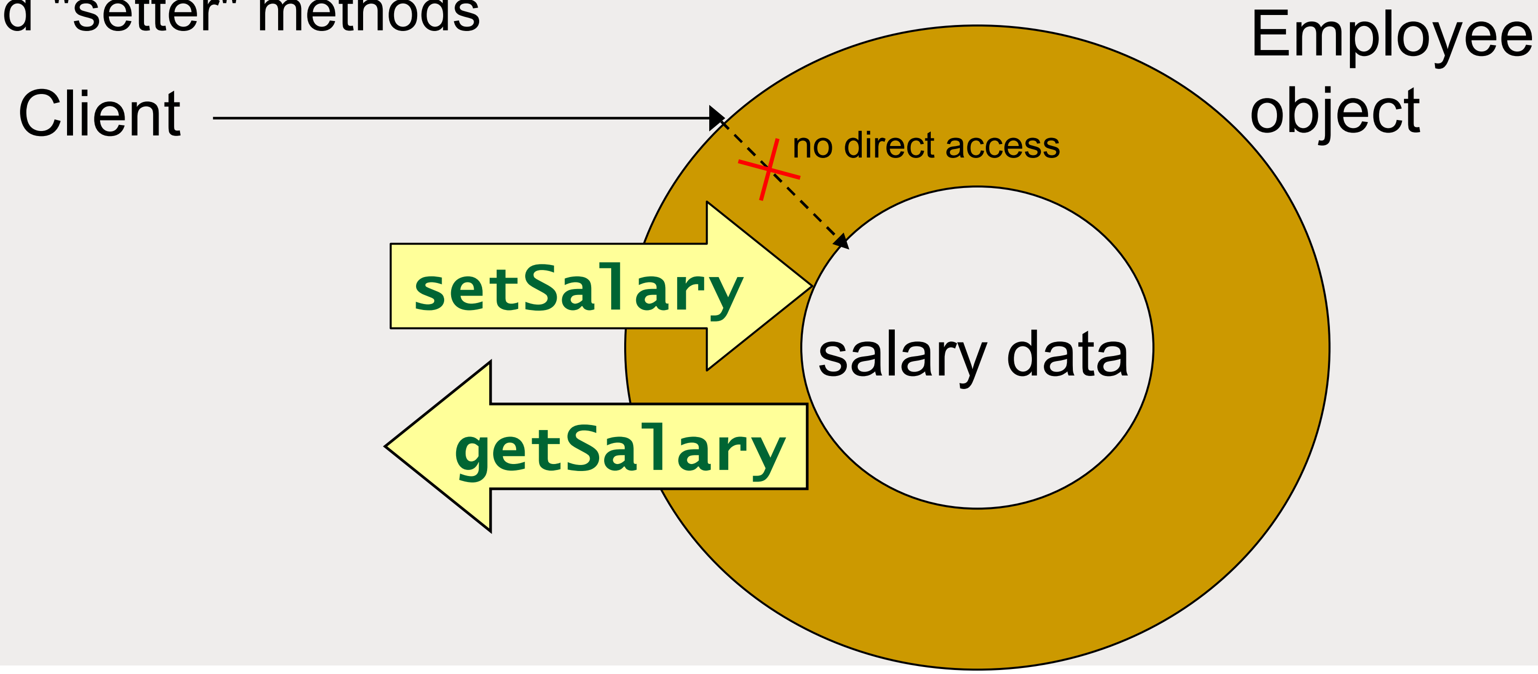
Encapsulation

- **An object need not reveal all of its attributes and behaviors**
  - In good OO design, an object should only reveal the minimum needed to interact with it
- **Internal details should be hidden from other objects**
- **This is called *encapsulation***
- **Encapsulation pertains to data and behavior**
  - Data can be hidden from the client
    - Only accessed internally (by methods in the object itself)
  - Methods can be hidden from the client
    - Only invoked internally (by other methods in the object itself)

Data Encapsulation - Illustrated

- **The infamous donut diagram (the donut is an object)**
  - If you've read any OO books, you've seen something like this before

- **In the "center" is a piece of data (salary)**
  - The only way a client can get at salary is through the "getter" and "setter" methods



Data Encapsulation - Example

- We mark the salary data as **private**
- We provide **public** getter/setter methods to access it

```
class Employee {
  // private so clients can't see it
  private int salary;

  // public so clients can call them
  public int getSalary() {
    return salary;
  }

  public void setSalary(int sal) {
    salary = sal;
  }
}
```

- Client code can only access salary via the methods

```
Employee emp = new Employee();
emp.setSalary(1000);
```

Benefits of Data Encapsulation

- **Enforce business rules, validate incoming data**
  - Example: salary cannot be negative
    - Without encapsulation, a client could do this:

```
Employee emp = new Employee();
emp.salary = -100;  // you pay us to work here, ha ha
```

- **Shield client code from the object's internal data structures**
  - The less clients know about how an object works, the better
  - For clients, ignorance is bliss
- **Adaptability to changing requirements**
  - Example: you need to handle fractional salaries, but you can't break existing client code that works with integer salaries

Ellucian

Lab 2.5
Data Encapsulation

Lab 2.5 - Encapsulate Person Data

- **Make a copy of the *Lab02.4_AttributesMethods* project and name the copy *Lab02.5_DataEncapsulation***
  – Close the Lab02.4 project, then **delete** the calculator classes

- **In `Person`, make the data members `private`**
  – This will create compile errors in `PersonClient` (why?)

- **The instructor will show you how to generate accessor methods with the IDE**
  – They will default to **`public`** access, which is fine
  – The instructor will also show you how to write a very common method, called **`toString()`** (which can also be generated)

- **Make the other methods `public`, too**
- **See notes about using accessor methods inside the class**

Lab 2.5 - Use the Accessor Methods

- In **PersonClient**, call the setter methods to set the data

```
Person p = new Person();
p.setName("Martin");
p.setAge(33);
...
```

- Call getter methods to read the data

```
// calling this method returns a value, which we store in a variable
double bmi = p.calculateBMI();  // as before
System.out.println(p.getName() + " has a BMI of " + bmi);
```

- Try out the toString() method

```
System.out.println(p.toString());
```

Lab 2.5 - Default Values

- **Recall that instance variables are initialized to "zero" for that type**
  - Integers `0`
  - Decimals `0.0`
  - Booleans `false`
  - Class types `null`

- **In the `main()` method of `PersonClient`, create another new instance of `Person`**
  - But **don't** set any of its data
  - Then call its `toString()` method – what do you notice?
  - In the `Person` class, change the `age` variable to have a default value of 10 and run the client again
    - After you've seen the results, remove the 10

Ellucian

Java and OO Fundamentals
Strings, Arrays, and Wrapper Classes

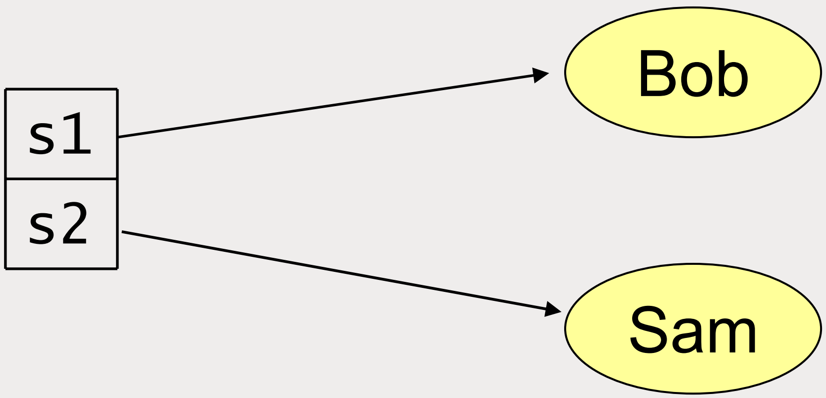
Classes and Objects
Data Types, Variables, and Operators
Attributes and Methods
Data Encapsulation
Strings, Arrays, and Wrapper Classes

String Is an Object

- **`Strings` are special**
  - Don't need **new** (the **only** class that doesn't't)
  - Immutable (details next)

```
String s1 = "Bob";
String s2 = "Sam";
```

- **We think of them as values, but they're really objects**
  - Which means they have references, just like all objects



Strings Are Immutable

- ***Immutable*** **means unchangeable**
  - Once `String` object is created, its characters can **never change**

- **NOTE: this means unchangeable even if it looks like it's changing**

> **toUpperCase**
>
> public String *toUpperCase*()
>
> Converts all of the characters in this String to upper case

```
String s1 = "abc";
s1.toUpperCase();          // converts "abc" to "ABC" (doc says so)
System.out.println(s1);    // OUTPUT IS "abc"  HUH?!
```

- **Read the API – toUpperCase()** ***returns*** **a String**

```
String s2 = s1.toUpperCase();
System.out.println(s2);    // OUTPUT is "ABC" (s1 is still "abc")
```

String Class and String Comparison

- **The `String` class is *rich* with functionality**
  - You can do all sorts of things with strings
  - **But remember: the original string is unchanged**
  - The "string manipulation" methods all **return** a `String`
    - They return the result of your manipulation – original is unchanged

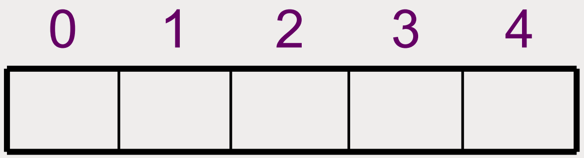
- **To compare two strings for *equality* (same characters)**
  - Use the **`equals()`** method, **not** ==
    - Remember, == compares ***identity***, i.e., point to same object

```
public void register(String username, String password) {
  // need to determine if username/password are the same
  if (username.equals(password)) {
    ...
  }
}
```

Arrays

- **An array is an ordered group of items**
  - Can contain primitive values, or object references
  - Elements must be all the same type

- **Arrays are actually objects**
  - But there is no class for them
  - You create them with **new**



- **Zero-indexed – all arrays have a `length` variable**
  - Be careful here, the length of this array is **5**
  - But the buckets are numbered **0 - 4**

- **Fixed length – cannot change size once created**

- **NOTE: Java Collections API used *much* more often (later)**

Creating Arrays

- Create an array object with **new[ ]**

```
String[] names = new String[2];  // array object with length 2
```

- Fill it with values

```
names[0] = "Jason";  // bucket 0 contains "Jason"
names[1] = "Edwin";  // bucket 1 contains "Edwin"
```

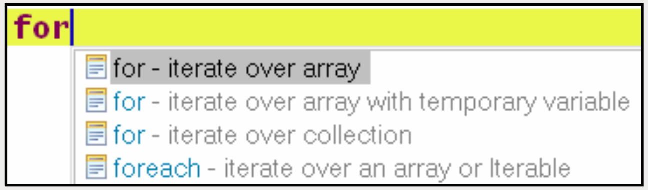
- Shortcut:

```
double[] shoeSizes = {7, 9.5, 11, 8.0, 12.5};
String[] cities = {"New York", "San Francisco", "Seattle"};
Person[] people = {new Person(), new Person()}
```

Iterating over Arrays

- **We'll officially discuss loops in the next section**

- **Two standard techniques for iterating over an array**
  - Both can be generated by the IDE, using [Ctrl+Space]




```
int[] ages = {11, 38, 45};

// use the loop's counter variable to hit buckets 0 thru length-1
for (int i = 0; i < ages.length; i++) {
  System.out.println(ages[i]);
}

// "for-each" loop (Java 5)
// read as: "for each int age in ages ..."
for (int age : ages) {
  System.out.println(age);
}
```

Command Line Arguments

- **As you know, `main()` takes a string array**

```
public static void main(String[] args)
```

- **Contains arguments passed in on the command line**

```
java HelloWorld arg1 arg2 "arg3 contains spaces"
```

- **Arguments are separated by a space**
  - Use quotes if an argument contains a space

Ellucian

Lab 2.6
Command Line Arguments

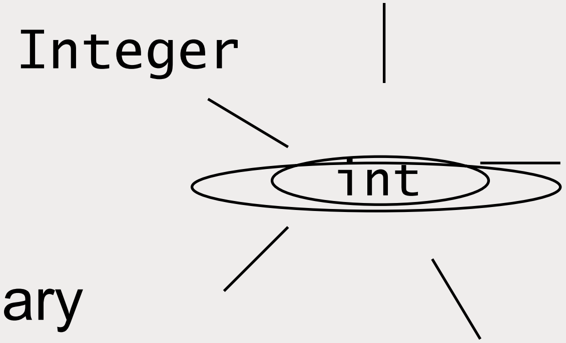
Lab 2.6 - Processing Command Line Arguments

- **In a new project *Lab02.6_Arguments*, create a class called `Arguments`**
  - Give it a `main()` method and do all work inside `main()`
  - The instructor will show you how to pass in arguments
- **Print a message to the console, indicating the # of args**
- **Iterate over them and print them out – use a "for-each" loop**
  - The instructor will show you how to generate one in the IDE
  - In the loop, also print the length of each argument (`String`)
    - See the Javadoc for `String` to find which method to call

Wrapper Classes - Overview

- **Primitive types are not objects**
  - Only things that are not objects in Java
  - Done for efficiency – objects incur overhead in the JVM

- **Corresponding *wrapper classes* for each primitive**
  - They "wrap" a single primitive value



- **Provide useful functionality**
  - Data conversion to/from `String`
  - Treat primitives as objects when necessary
    - Storing primitive values in a collection (more later)
    - Collections are more flexible than arrays, but only store objects

- **Java 5 introduced *autoboxing* (covered later)**
  - Wrappers still used behind the scenes

Meet the Wrapper Classes

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Using Wrapper Classes

- **Pass in the primitive value to be wrapped at creation time**

```
Integer ageInteger = new Integer(42);
Boolean isSunny = new Boolean(true);
```

- **Or pass in a string representation of the value**
  - Causes an error if the string value doesn't make sense

```
Double sizeDouble = new Double("9.5");
Boolean isFriday = new Boolean("false");
Long qLong = new Long("Q");  // causes an error
```

- **To get the primitive value back out**

```
int age = ageInteger.intValue();
double size = sizeDouble.doubleValue();
```

Data Conversion via Wrapper Classes

- Use for data conversion from string to primitive

```
String ageString = "35";  // want this as an int

// parseInt() is a static method, so we call it using the class name
// more on this later
int age = Integer.parseInt(ageString);
```

- There are patterns to these method names
  - `Long.parseLong()` returns a `long`
  - `Double.parseDouble()` returns a `double`
  - `Float.parseFloat()` returns a `float`
  - See the pattern?

Wrapper Facts

- **Can check wrapper objects for equality via `equals()`**
  - Returns `true` if they contain the same primitive value
  - You **don't** use **==** to compare (why not?)

```
Integer i = new Integer(7);
Integer j = new Integer(7);

if (i.equals(j)) {
   ...
}
```

- **Wrappers are immutable!**
  - You pass in the value when creating one
  - And can't change the value after that
    - Because there are no methods available to do this

Ellucian

Lab 2.7
Wrapper Classes

Lab 2.7 - Work with Wrappers

- **Continue to work in the *Lab02.6_Arguments* project**

- **Calculate the sum of the arguments, then print it**
  - Pass in "numeric" values to the program when running it
    - Remember, command line args come into `main()` as `Strings`
    - Then see what happens if you pass in a "Q"

CHALLENGE (optional)

- **Open your *Lab02.5_DataEncapsulation* project**

- **Create a new class called `PersonClientArgs` – with `main()`**
  - Create a new `Person` object and set its data from command line arguments
  - Then call its `toString()` method

Section Review

1. What are the two fundamental things found in a class definition?
2. Class names should begin with a capital letter. [T/F]
3. Java is "case aware," but not case sensitive. [T/F]
4. ______ variables are those that are declared inside a method block.
5. Explain pass by value (copy) and pass by reference.
6. How do you invoke a method on an object?
7. Explain how the + operator works.
8. Explain the difference between == on primitives and == on objects.
9. There is no way to tell if a reference variable points to an object or not. [T/F]
10. Why are `Strings` special?
11. How can you tell the size of an array?
12. All arrays are instances of the `Array` class. [T/F]
13. What is a wrapper class? Why are they useful?
14. What happens when a `String` is changed?
15. The size of an array cannot change. [T/F]

End of Section

- **This slide is intentionally devoid of any useful content**

Ellucian

Section 3
Flow of Control

Conditional Statements
Looping

Section Objectives

- Use Java conditional logic statements
- Learn how to write loops in Java

Ellucian

Flow of Control
Conditional Statements

Conditional Statements

Looping

`if` Statement

```
if (boolean-expr)
  // do an action

if (boolean-expr) {
  // do an action
  // do another action
}

// remember: = is assignment, == is comparison
if (x = 5) {    // ERROR: should be x == 5
  // do something
}
```

- Braces not required
- Recommend ***always*** using braces

if-else Statement

```
if (boolean-expr) {
  // do an action
  // do another action
}
else {
  // do something else
}
```

- Compact alternative: ***ternary operator*** (3 operands)

    `boolean-expr ? true-value : false-value`

```
public static void main(String[] args) {
  String str = (args.length > 0) ? "got args" : "no args";
}
```

if-else Statement - Example

- You can nest them

```
public double pay(int hours, double rate) {

  double basePay = hours * rate;
  double multiplier = 0;  // why declare it here?

  if (hours > 50) {
    multiplier = 1.75;
  }
  else if (hours > 40) {
    multiplier = 1.5;
  }
  else {
    multiplier = 1;
  }
  return basePay * multiplier;
}
```

switch Statement

- Like `if-else`, but can make multiple choices
- Execution begins at the ***first case*** that matches the variable
  - And continues until it reaches a **break**
- If no case matches, executes the **`default`** case (if present)

```
// NOTE:can only switch on byte, short, int, char, enum (NOT String)
switch (variable) {
  case value1:
    // statements
    break;

  case value2:
    // statements
    break;

  default:
    // statements
}
```

switch Statement - Example

```
void determineRange(int i) {
  switch (i) {
    case 1: case 2: case 3:
      System.out.println("i is between 1 and 3");
      break;
    case 4: case 5: case 6:
      System.out.println("i is between 4 and 6");
      break;
    case 7: case 8: case 9:
      System.out.println("i is between 7 and 9");
      break;
    default:
      System.out.println("i is out of range");
  }
}
```

- **Notice how the cases are used here**

Ellucian

Lab 3.1
Conditional Logic

Lab 3.1 - Validating Person Data

- **Make a copy of the *Lab02.5_DataEncapsulation* project and name the copy *Lab03.1_Conditionals***
  - Then close the Lab02.5 project

- **In Person's `setAge()` method, do some simple validation**
  - If the value passed in is positive, set the instance variable
  - Otherwise, do nothing

- **Test it out from `PersonClient` (see notes)**
  - Create a new `Person` object and set its age to 10
    - Set its other properties, too
  - Call its `toString()` method – age should be 10

```
System.out.println(p.toString());
```

  - Now set its age to -10 and call `toString()`
    - What do you notice?

Lab 3.1 - More Conditional Logic

- **Continuing in `PersonClient`**
  - Call the new `Person`'s **`calculateBMI()`** method
  - If the value returned is at least 3, output a message
    - "The person's BMI is 3 or greater"
  - Otherwise, print a different message
    - "The person's BMI is less than 3"

- **In `Person`'s `eat()` method, check the parameter passed in**
  - If it's "pizza", output a different message
    - "Sorry, I don't eat pizza"
  - Otherwise, output the same message as before
  - Remember: use **`equals()`** for string comparisons, **not ==**
  - Test it from `PersonClient`, first by passing in "pizza", and then by passing in some other value

Ellucian

Flow of Control
Looping

Conditional Statements

Looping

`while` Loop

```
while (boolean_expr)
  // do an action

while (boolean_expr) {
  // do an action
  // do another action
}
```

- **Braces not required**
- **Recommend *always* using braces**

`while` Loop - Example

- **Processing a `ResultSet` in JDBC**
  - JDBC = Java Database Connectivity API
  - `ResultSet`'s **`next()`** method advances the row cursor and returns true/false, indicating if there's a row there
    - It conveniently returns false after the last row

```
Statement stmt = ...
ResultSet rs = stmt.executeQuery("SELECT * FROM ...");

// for N rows, this will execute N times (nice)
while (rs.next()) {
   ...
}
```

for Loop

```
for (initialization; boolean_expr; iteration) {
  // statements
}
```

```
for (int i = 0; i < 10; i++) {
  // loop executes 10 times
}
```

- **Initialization**
  - Sets initial value, can also include a variable
- **Boolean expression**
  - Loop repeats until false
- **Iteration/increment**
  - Fires just after body of the loop (just before the closing **}**)

for-each Loop

- **Introduced in Java 5 – easier way to iterate over arrays and collections**

```
int[] ages = {11, 38, 45};

// read as: "for each int age in ages ..."
for (int age : ages) {
  System.out.println(age);
}
```

- **Sometimes you still need to use the classic for loop**
  - When you need the iteration counter value

```
// use the loop's counter variable to hit buckets 0 thru length-1
for (int i = 0; i < ages.length; i++) {
  if (ages[i] >= 21) {
    System.out.println("Found an age over 21 at bucket " + i);
  }
}
```

break

- **Terminates a loop early**

- **Often used for efficiency reasons**
  - If you're looping to search for a value, once you find it, stop looping

```
public void searchForName(String[] names, String searchName) {
  // search for searchName in names array
  for (String name : names) {
    if (name.equals(searchName)) {
      // found one, stop looping
      break;
    }
  }
}
```

Section Review

1. What are the conditional statements? How do they differ?
2. What is the fastest loop?
3. What is the (classic) standard technique for iterating over an array?
4. How does array iteration change in Java 5?
5. Explain the difference between the following two `if` statements:

```
if (day.equals("Friday")) {
  leaveWorkEarly();
  goToTheatre();
}

if (day.equals("Friday"))
  leaveWorkEarly();
  goToTheatre();
```

Ellucian

Section 4
More about Classes and Objects

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Section Objectives

- Understand the benefits of enums, and how to use them
- Understand method overloading and why it's useful
- Learn about constructors and object creation
- Explain the JavaBeans naming convention, and how it enables useful abstractions
- Understand packages, public classes, and package-level encapsulation
- Understand `static` variables and methods, and how they differ from instance members

Ellucian

More about Classes and Objects
Enums

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

What is an Enum?

- **Added in Java 5 – a mechanism to constrain a data type to a fixed set of values**
  - The set of all possible values is specified
  - Compiler-enforced
  - Type-safe

- **They extend the `Enum` class**

```
enum Gender { MALE, FEMALE }
```

- **Defines a new type (much like a class)**
  - There are only two possible instances of `Gender`
    - **`Gender.MALE`** and **`Gender.FEMALE`**
  - Note the naming convention

Life before and after Enums

- **Before enums, you'd probably represent gender as a string**
  - You'd have make sure your application **only** uses "M" or "F"
    - The compiler can't enforce this, so you have to

```
public void setGender(String gender) {
  // any string could be passed – forces me to validate it
}
```

- **With enums, we are guaranteed only two possible values**

```
public void setGender(Gender gender) {
  // can only be Gender.MALE or Gender.FEMALE
}
```

Using Enums

- They can be used in `if` statements

```
public void getLost(Gender gender) {
  if (gender == Gender.FEMALE) {  // NOTE: ok to use == here
    // stop and ask for directions (some men will never do this)
  }
}
```

- They can be used in `switch` blocks

```
public void teeOff(Gender gender) {
  switch (gender) {
    case MALE:
      // use the white tees
      break;
    case FEMALE:
      // use the red tees
      break;
  }
}
```

Ellucian

More about Classes and Objects
Method Overloading

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Overloading Methods

- **Different methods with the *same name* in the same class**
  - So how do we distinguish between them?

- **They differ in their parameters (number and/or type)**

```
void eat() { /* eat whatever you want */ }
void eat(String food) { /* eat the given food */ }
```

```
void watchTV(int channel) { /* watch the given channel number */ }
void watchTV(String network) { /* watch the given network name */ }
```

- **At runtime, the correct method is chosen**
  - Based on the arguments passed

```
p.eat();            // calls the 1st eat() method above
p.eat("salad");  // calls the 2nd one
```

Overloading - Why?

- **Saves you the time and trouble of both thinking up and remembering method names when coding**
  - Makes an API easier to remember and use (this is important)

- **Without overloading, the previous slide might be:**

```
void eatSomething() { ... }
void eatSpecifiedFood(String food) { ... }
```

```
void watchTVChannel(int channel) { ... }
void watchTVNetwork(String network) { ... }
```

Ellucian

Lab 4.1
Method Overloading

Lab 4.1 - Method Overloading

- **Create a new project *Lab04.1_Overloading***
  - Copy the `Person` class from the ***Lab03.1_Conditionals*** project into this new project (then close the Lab03.1 project)
  - Your instructor will show you how to do this

- **Add another `eat()` method – one that takes no arguments**
  - Output a message something like the following

```
System.out.println(this.getName() + " is eating my favorite food");
```

- **Create a new `PersonClient` class – with a `main()` method**
  - Create a new instance of `Person` and set the name property
  - Call both `eat()` methods
  - Notice how the correct one is called each time

Ellucian

More about Classes and Objects
Constructors

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Constructors - What Are They?

- **Special kind of method invoked when you say new**
  - They initialize newly created objects

- **They have the *same name as the class***

```
Person p = new Person();  // calling the method named Person
```

- **They have no true return type**
  - **But:** an instance of the class is returned

- **Like other methods, they can be overloaded**

```
class Person {
  private int age;

  Person() { ... }          // method named Person
  Person(int age) { ... }  // same name, different parameter list
}
```

Implicit Constructor

- Java language says that ***all classes must have at least one constructor***

- If you type in no constructors for a class, the compiler ***automatically inserts*** a "default" or no-arg constructor
  - Unknowingly, we've been using it in the labs

```
Calculator calc = new Calculator();
Person p = new Person();
```

- If you type in at least one constructor, the "default" constructor is ***no longer provided*** by the compiler
  - Because the "at least one constructor" rule has been met
  - If you want a no-argument constructor, just type it in

Uses of Constructors

- **Allows client to pass in initial data during instantiation**
  - Much easier than calling a bunch of individual setters
  - How many calls does it take to fully initialize a `Person` with data?

```
class Person {
  private String name;
  private int age;

  public void setName(String name) { ... }
  public void setAge(int age) { ... }
}
```

- **Wouldn't you rather be able to do it in one shot?**

```
Person p = new Person("Jason", 42);
```

Writing Constructors

- **Often, you simply take in parameters and assign their values to the instance variables (just like setters do)**
  - Here's what a `Person` constructor might look like:

```
class Person {
  private String name;
  private int age;

  // if this one isn't provided, what does that mean for the client?
  public Person() {
  }

  // NOTE: best practice to call your own setters (why?)
  public Person(String name, int age) {
    this.setName(name);
    this.setAge(age);
  }
}
```

Ellucian

Lab 4.2
Constructors

Lab 4.2 - Writing Constructors

- **Create a new project *Lab04.2_Constructors***
  – Copy the `Person` class from the ***Lab04.1_Overloading*** project into this new project (then close the Lab04.1 project)

- **Write some constructors – at least the first two below:**
  – No-argument
  – Takes **name** parameter

  Optionally, add some more:
  – Takes **name** and **age** parameters
  – Takes all 4 parameters – **name**, **age**, **height**, **weight**

Lab 4.2 - Calling Constructors from the Client

- Create a new `PersonClient` class – with a `main()` method
  - Create some instances of `Person` using your new constructors
  - Then call `toString()` to make sure the data were set

```
Person p1 = new Person("Leanne");
System.out.println(p1.toString());

Person p2 = new Person("John", 17);  // if you provided this ctor
System.out.println(p2.toString());
```

Ellucian

More about Classes and Objects JavaBeans

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Java Naming Conventions - Review

- Classes – begin with a capital letter, then CamelCase
  - `JavaDeveloper`
- Fields – begin with a lowercase letter, then CamelCase
  - `baseSalary`
- Methods – begin with a lowercase letter, then CamelCase
  - `writeCode()`
- Getter methods should be prefixed with **get**
  - `getBaseSalary()`
- Setter methods should be prefixed with **set**
  - `setBaseSalary()`

JavaBeans - Naming Conventions Become Law

- **JavaBean = class that *strictly* follows these conventions**

- **Must have a default or no-arg constructor** less important

  more important
- **Must be a *public* class (we'll talk about this next)**
- **Getters/setters *must* use standard naming convention**
  - getSomething/setSomething
    - For booleans, can also use isSomething/setSomething
    - For example, isMarried/setMarried
  - Setters must be public and have `void` return type
  - Getters must be public and have return type matching property
    - See notes for details

JavaBeans Provides a Useful Abstraction

- An object has ***properties*** instead of get/set methods

```
public class Person {
  // only getter/setter methods shown
  public int    getAge() ...                 age
  public void   setAge() ...
  public String getName() ...                name
  public void   setName() ...
  public double getShoeSize() ...            shoeSize
  public void   setShoeSize() ...
}
```

- Allows tools and other Java technologies to interact with these objects at a higher level, e.g., on a JSP page:

```
<%-- get person object somehow (details in Servlets/JSP course) --%>
Hello and welcome to the Web site, ${person.name}
```

calls `person.getName()`

JavaBeans Caveats

- **Often, only the getter/setter naming rule matters**
  - If the tool/framework relying on this abstraction doesn't need to instantiate the class, it doesn't care about the no-arg ctor
- **`getSomething`/`setSomething` → `something` property**
  - This does **not** require you to have a field named `something`
  - You might calculate the value on the fly, or fetch it from somewhere
- **Read-only and write-only properties:**
  - Read-only: getter but no setter
  - Write-only: setter but no getter

Ellucian

Lab 4.3
JavaBeans

Lab 4.3 - JavaBeans

- **Is our Person class from the labs a JavaBean?**
  – If so, what are its properties?

- **The class below is a JavaBean – list its properties**

```
public class AuctionItem {
  private Long id;
  private double price;
  private double minPrice;

  public Long getId() { ... }
  public void setId(Long id) { ... }

  public double getSalePrice() { ... }

  public boolean isAntique() { ... }
  public void setAntique(boolean old) { ... }
}
```

Ellucian

More about Classes and Objects
Packages and Imports

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Packages

- **Collection of related classes**

- **Without them, would be like having all of your files in C:**
  – Hard to find what you want – and no duplicate filenames

- **Packages provide:**
  – Organization of classes
  – Unique classnames
  – More flexibility in access control

- **The Java Core API classes are in packages – `java.lang`, `java.io`, `java.util`, etc.**
  – Any real code is in a package, including vendor classes

package Statement

- Must be the ***first*** thing in the class definition (not including comments)

- Placing your classes in a package is as simple as this:

```
package com.ellucian.training;

class Instructor {
  ...
}
```

Package Naming Conventions

- **Reverse domain name – all lowercase**
- **What is your organization's domain name?**
  - Reverse it to get your base package name
  - **`com.ibm`**, **`gov.irs`**, **`edu.mit`**, **`org.npr`**

- **Globally unique – your company owns your domain name, and no one else would really use it for their package name**

- **Often add project name or other identifiers after the base**
  - `com.ibm.websphere.portal`
  - `gov.irs.taxes`

- **Some organizations don't adhere**
  - Oracle uses `oracle.jdbc`, not `com.oracle.jdbc`

Fully-Qualified Classname

- **Fully-qualified name = package name + classname**

```
package com.ellucian.training;

// fully-qualified classname is com.ellucian.training.Instructor
class Instructor {
  ...
}
```

- **Packages provide unique names**
  - Seems everybody has a `Connection` class

```
package com.ellucian.server;

class Connection {

}
```

```
package java.sql;

class Connection {

}
```

import Statement

- **Without them:**

```
// somewhere in the class:
com.ellucian.server.Connection c =
  new com.ellucian.server.Connection();  // you must be kidding

java.util.Date now = new java.util.Date();
```

- **With them:**

```
// at the top of the file, after the package statement
import com.ellucian.server.Connection;
import java.util.Date;

// then somewhere in the class:
Connection c = new Connection();        // that's better
Date now = new Date();
```

What `import` Does - and Doesn't Do

- **All it does is save typing**
  - You are spared from typing the fully-qualified classname

- **`import` gives you an "alias" or "short form" for the fully-qualified classname**

- **It does not:**
  - Load any external code into your class
  - Load any code into memory
  - Copy any code into your class (like a `#include` in C/C++)

Automatic Imports

- `java.lang` is automatically imported
  - Takes care of `String`, `Integer`, `System`, etc.
  - These classes are the "core" of the Java Core API

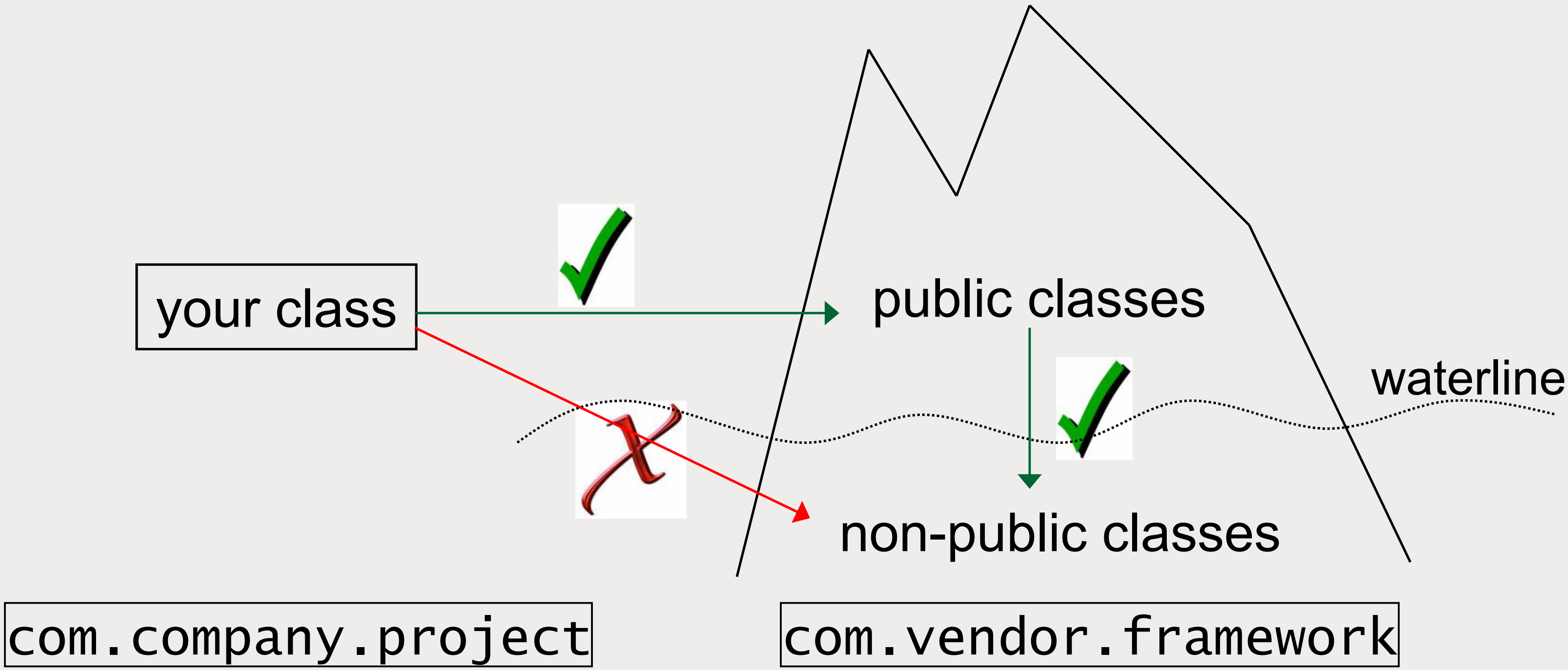
- Your class's package is automatically imported

```
package com.ellucian.training;

// NOT NECESSARY if Computer also in package com.ellucian.training
import com.ellucian.training.Computer;

class Instructor {
  private Computer computer;
}
```

Visibility - Classes

- A ***public*** class is visible to classes in other packages
- A ***non-public*** class is only visible to classes in the ***same package***

- Provides for package-level encapsulation



Visibility - Methods and Variables

- `public`
  - Accessible everywhere the class is accessible

- `private`
  - Only accessible within the class

- **default** **(not a keyword)**
  - This is what it is if you don't say `public` **or** `private`
  - Accessible from other classes in the **same package**
  - You'll see several terms for this level of visibility:
    - "package-private"
    - "package"
    - "default"

Ellucian

Lab 4.4
Packages and Imports

Lab 4.4 - Putting Classes in a Package

- **Make a copy of the *Lab04.2_Constructors* project and name the copy *Lab04.4_Packages***
  - Your instructor can show you how to do this
  - Then close the Lab04.2 project

- **In the new project, we'll put `Person` in a package (but not `PersonClient`)**
  - Right-click on the **src** folder and choose **New → Package**
  - Name the package **`com.hr.personnel`**
  - Drag the `Person` class into the package
  - The IDE should automatically provide the appropriate `package` and `import` statements
    - Review both `Person` and `PersonClient` to see this

- **Run the client – should work as before**

Ellucian

More about Classes and Objects
Static Members

Enums
Method Overloading
Constructors
JavaBeans
Packages and Imports
Static Members

Class-Wide or `static` Data Members

- **We know about instance data**
  - Values vary from instance to instance
  - And are stored in each individual object (inside the "donut")
- **Sometimes you want to store a data member "class-wide"**
  - Not in each individual instance, but in a class-level "shared" area
- **So how do we tell the difference between an instance-specific value and a class-wide "common" value?**

Class-Wide or `static` Data Members

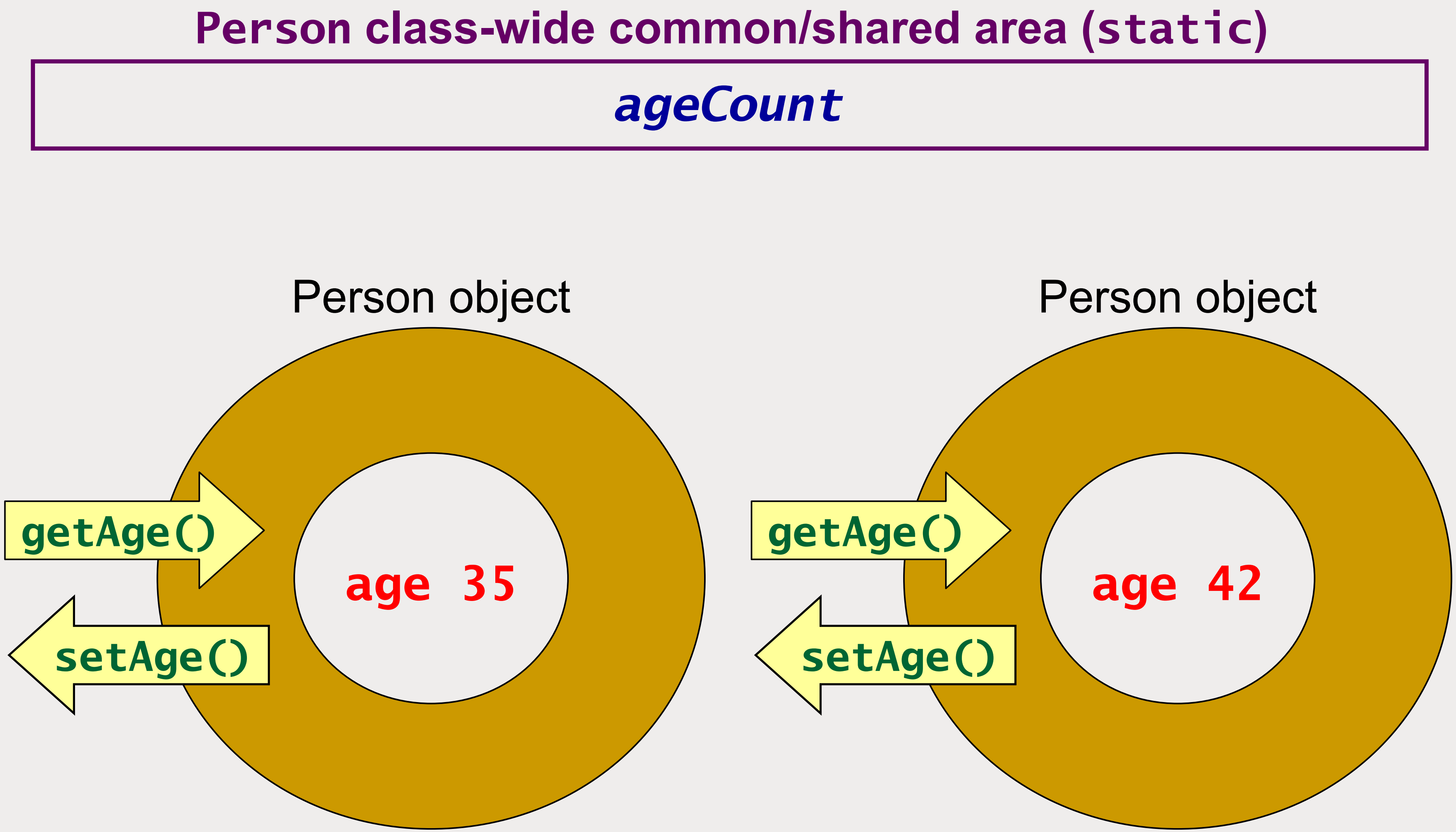
- **We use the keyword `static` to indicate this**

- **Careful: "static" in English means "does not change"**
  - A `static` data member's value **can** change
  - But it **doesn't vary from instance to instance**
    - It's **common** to all the instances, or "shared" by them

- **Using our person example, we need to store the number of times any `Person` has been asked its age (via `getAge()`)**
  - They're sensitive about age and want to keep track of this

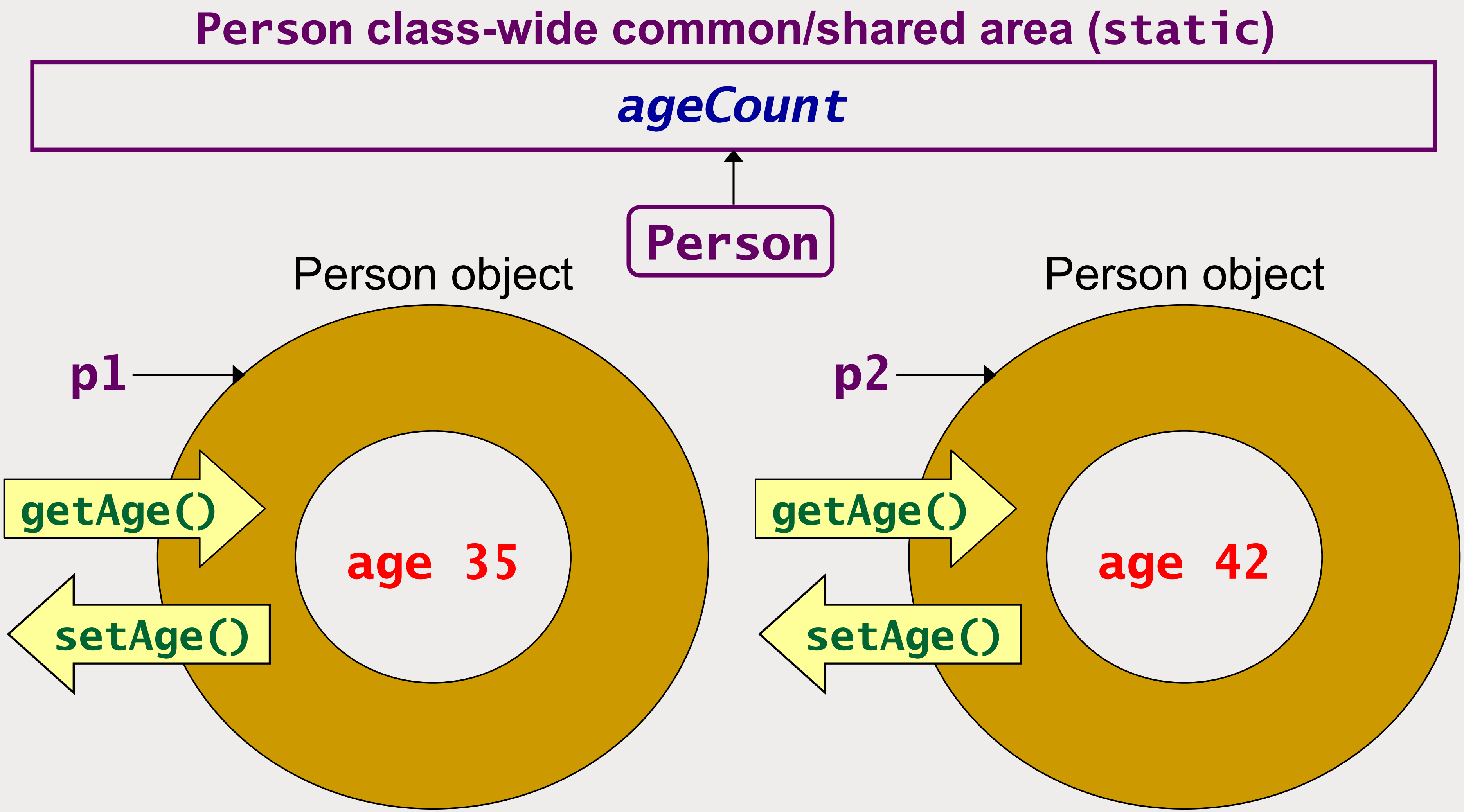
```
class Person {
  static int ageCount;  // common or shared among all Persons
  private int age;      // specific value for a specific Person
  // rest of class not shown
}
```

Instance Storage vs. Class-Wide Storage



Referencing the Class-Wide Common Area

- We use the ***class name***, ***not*** an object reference



Class-Wide or static Data Members - Example

```
Person p1 = new Person(35);
Person p2 = new Person(42);
p1.getAge();  // Person.ageCount is now 1 (p1 incremented it)
p2.getAge();  // Person.ageCount is now 2 (p2 incremented it)
System.out.println("Person ageCount is: " + Person.ageCount);
```

```
class Person {
  static int ageCount;  // common or shared among all Persons
  private int age;      // specific value for a specific Person

  public Person(int age) {
    this.setAge(age);
  }

  public int getAge() {
    Person.ageCount++;  // increment the class-wide ageCount
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

`static` Methods

- **Encapsulate `static` data, just like we do with instance data**
  - Make the variable `private`
  - Provide `static` methods for clients to access it

- **Methods that are really more like "functions"**
  - Take input data and return some computed value
  - No need to create an entire object just to execute the function
  - No object data involved
    - Function gets all its input as parameters, computes return value

- **Also called "utility" methods**
- **Examples:**
  - `Integer.parseInt(String)` – converts `String` into `int`
  - `Math.sqrt(double)` – returns square root of supplied value

How the JVM Starts Your Application

- **The JVM starts your program by calling `main()` (review)**

- **`main()` is `public` and `static`**
  - It's `public`, so the JVM can call it in the first place
  - It's `static`, so the JVM can call it as shown below

command line

```
java HelloWorld HI
```

JVM response

```
HelloWorld.main(["HI"]);
```

Class Constants

- **Recall that class-level or `static` data members *can* change**
  - They just don't change from instance to instance
    - Not stored in any instance, but rather in the common area

- **What if you wanted one that *couldn't* change value?**

- **We do this by marking the `static` variable as `final`**
  - Final means final – it's a ***class constant***

- **Since it can't change value, we can also make it `public`**
  - Thus, they are marked **`public static final`**
  - And use **`ALL_CAPS`** naming convention

All-`static` Classes

- **The `Math` class is a great example**
  - It's just a bunch of math functions
    - `Math.abs()`
    - `Math.max()` and `Math.min()`
    - `Math.sin()`
    - `Math.random()`
  - It also has two class constants
    - `Math.PI`
    - `Math.E`

- **It has no instance variables, and you don't create any `Math` objects**
  - You just call the functions using the class name

```
double result = Math.sqrt(25.0);
```

Ellucian

Lab 4.5
Static Members

Lab 4.5 - Static Members

- **Remember our `Calculator` class?**
  - It doesn't have any instance variables – it only has methods
    - Thus, there's nothing that makes one `Calculator` object "different" than another
    - In fact, all its operations involve "one-off" calculations that don't involve any "state" (data stored in the `Calculator` object itself)
  - It's a classic candidate for an all-`static` class

- **Create a new project *Lab04.5_StaticMembers***
  - Copy the **`Calculator`** and **`CalculatorClient`** classes from the ***Lab02.4_AttributesMethods*** project into this new project
    - The instructor can help you with this, or do it as a class
    - Then close the Lab02.4 project

Lab 4.5 - More Practice with Packages

- **In the new project, we'll put `Calculator` in a package (but not `CalculatorClient`)**
  - Right-click on the **src** folder and choose **New → Package**
  - Name the package **`com.math.util`**
  - Drag the `Calculator` class into the package
  - The IDE should automatically provide the appropriate `package` and `import` statements
    - Review both `Calculator` and `CalculatorClient` to see this

- **NOTE: you may have compile errors in `CalculatorClient`**
  - Depending on how you wrote the methods
    - They need to be public now (why?)

```
public static void main(String[] args) {
  Calculator calc = new Calculator();
  System.out.println(calc.add(2, 3));
  System.out.println(calc.
```

The method add(double, double) from the type Calculator is not visible

Lab 4.5 - Static Methods

- **Make `Calculator`'s methods static**

```
package com.math.util;

public class Calculator {

  public static double add(double a, double b) {
    return a + b;
  }
```

- **In `CalculatorClient`:**
  - Remove the instantiation of the `Calculator` object
    - The line with "`new Calculator()`" in it
  - Use the ***class name*** when calling `Calculator`'s methods
    - `Calculator.`*`add`*`()`, etc.

- **Run the client – should work as before**

STOP

Section Review (1 of 2)

1. What benefits do enums provide? What is their naming convention?
2. Describe method overloading. How does it provide convenience?
3. What do you do to call a constructor?
4. Constructors, though convenient, cannot be overloaded. [T/F]
5. Explain the "free default constructor."
6. List the rules for a class to be a JavaBean.
7. What's the most important benefit of JavaBeans?
8. What is a Java package? Why do we have them?
9. Which two packages do you never need to import?
10. Why do we do imports, anyway? What happens when we do one?

Section Review (2 of 2)

1. What is the standard package naming convention?
2. Explain what we mean by a "fully-qualified" class name.
3. What visibility modifiers are available for classes? Explain them.
4. Inside a class, what visibility modifiers are available for data members and methods? Explain them.
5. What do we mean by package-level encapsulation?
6. What is the difference between a `static` data member and an instance variable?
7. `static` data members cannot be marked `private`. [T/F]
8. To access a `static` data member, you prefix it with `static`. [T/F]

Ellucian

Section 5
Composition and Inheritance

Composition and Delegation
Inheritance

Section Objectives

- Understand composition, delegation, and the HAS-A relationship
- Learn how inheritance works, the IS-A relationship, and explain its benefits
- Understand what overriding is, and when you use it
- Understand polymorphism and encapsulation of type
- Explain constructor chaining, and gain a deeper understanding of object instantiation
- Understand the significance of class `Object` and its methods

Ellucian

Composition and Inheritance
Composition and Delegation

Composition and Delegation

Inheritance

Relationships between Classes - HAS-A

- **Objects can be composed of other objects**
  - We see this all the time in the real world
  - Computer has a DisplayScreen, Keyboard, Mouse, etc.
  - Customer has an Address – Employee also has an Address
    - We've built more complex objects from simpler objects

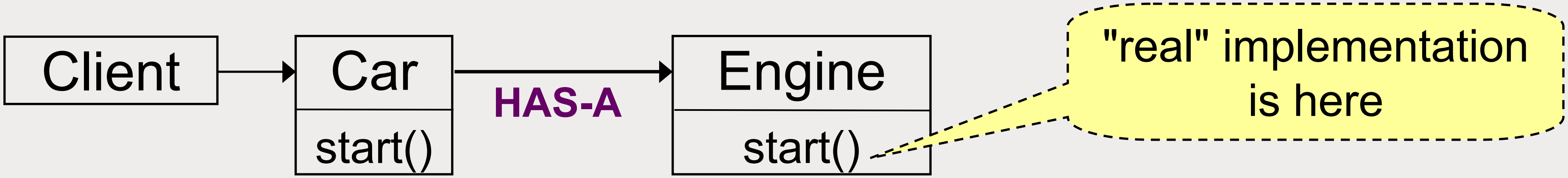
- **Called the *HAS-A* relationship**

- **Using the example above, Computer has *collaborators***
  - DisplayScreen, Keyboard, and Mouse

- **Promotes *code reuse* and *cohesion***
  - Reuse: you can use prebuilt components
  - Cohesion: instead of one class doing many unrelated things, you get smaller, more tightly-defined classes

Delegation

- Instead of repeating the work, reuse what's already there
- One class delegates responsibility to another class




```
public class Car {
  private Engine engine;  // Car HAS-A Engine

  public Car(Engine engine) {
    this.engine = engine;
  }

  public void start() {
    engine.start();
  }
}
```

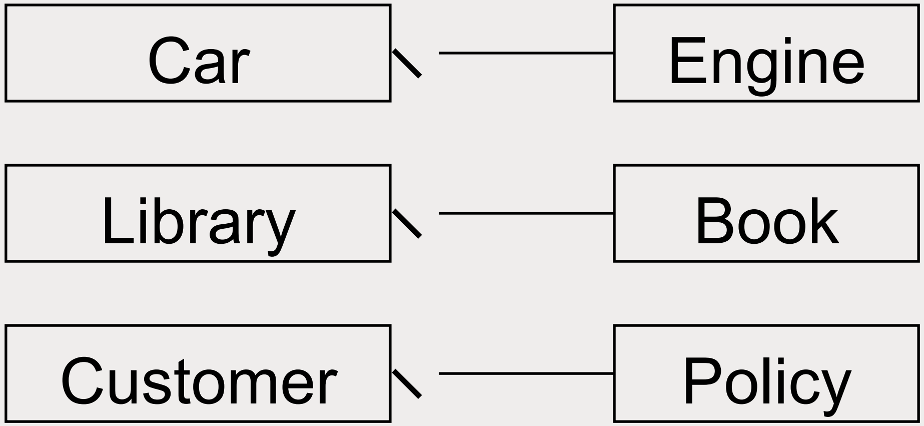
Delegation - Why?

- **Why separate out Engine responsibilities from Car?**
- **It depends**
  – Can the car use different engines?
  – Can the engine be used by more than one car?
- **In our example, the Car class *encapsulates* Engine**
  – The client knows only about Car, not its internal Engine
- **This has consequences**
  – Dependency between Car and Engine
  – And therefore ***coupling*** – one class has knowledge of another

Composition in UML

- **UML is the *Unified Modeling Language***
  - A graphical way to show classes and their relationships

- **A diamond appears on the class that "owns" or HAS-A the other class**



Ellucian

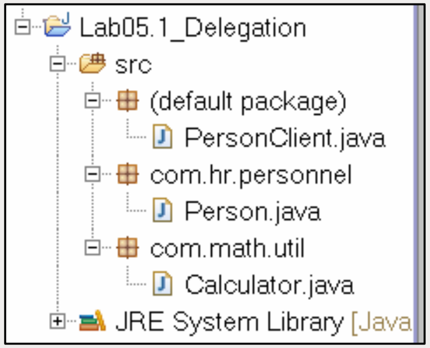
Lab 5.1
Delegation

Lab 5.1 - Delegation

- **To show a simple delegation, our `Person` class will use `Calculator` in the BMI calculation**
  – Strictly speaking, this is not composition, just delegation
  – Sometimes called a ***USES*** relationship

- **Make a copy of the *Lab04.4_Packages* project and name the copy *Lab05.1_Delegation***
  – Then close the Lab04.4 project

- **Copy the `com.math.util` package from *Lab04.5_StaticMembers* into this new project**
  – The instructor can help you with this, or do it as a class
  – Then close the Lab04.5 project



Lab 5.1 - Delegation

- **In Person's `calculateBMI()` method, delegate the math work to `Calculator`**
  - `Calculator` is in package `com.math.util`, so make sure that you have the appropriate `import` statement in `Person`
    - [Ctrl-Shift-O] ("organize imports") should take care of this

```
public double calculateBMI() {
  // Person USES Calculator to help with the calculation
  return Calculator.divide(weight, height);
}
```

- **Test it out from `PersonClient` (see notes)**
  - Create an instance of `Person` and set all 4 properties
    - Use your constructors/setters to set them
  - Get its BMI and print it out – should work as before

```
double bmi = p.calculateBMI();  // as before
System.out.println(p.getName() + " has a BMI of " + bmi);
```

Ellucian

Composition and Inheritance
Inheritance

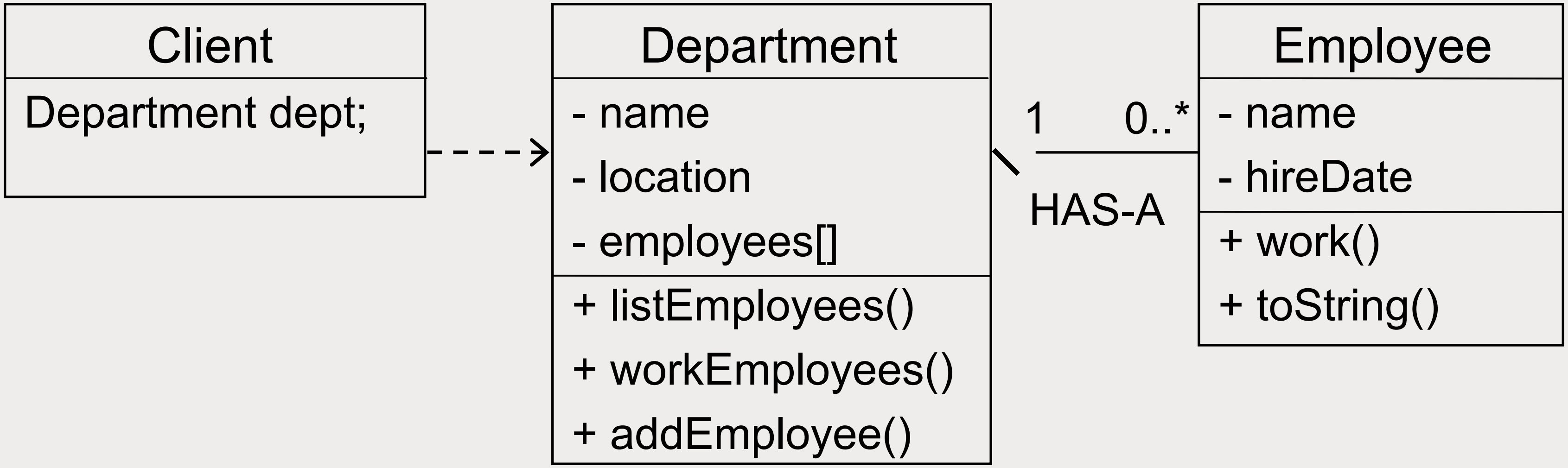
Composition and Delegation

Inheritance

Business Problem - Introduction

- **Simple HR system**
  - Department and Employee classes
- **Department**
  - name and location properties
  - Array of Employees
  - Methods to list its Employees, and to make them work
- **Employee**
  - name and hireDate properties
  - Method to do work
- **HAS-A relationship between them**
  - A Department has many Employees (1-to-many)

HR System - UML Diagram



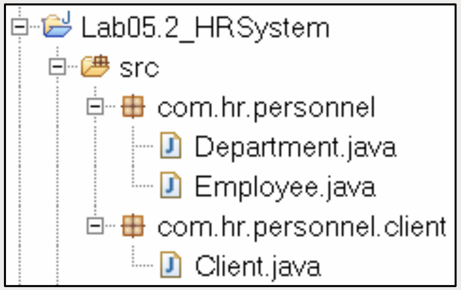
Ellucian

Lab 5.2
HR System

Lab 5.2 - HR System

- **In this lab, we'll review the basic components of the system**

- **Create a new project *Lab05.2_HRSystem***
  - With the instructor guiding, import the code from the ***LabSetup/Lab05.2*** directory



- **Review the classes with the instructor**
  - Start with `Employee`
  - Then look at `Department`
  - Finally, look at `Client`

- **Run the client and note the output**

Inheritance Defined

- **Inheritance allows a subclass to gain the attributes and behaviors from its parent class**
  - Can also add new attributes and behavior
    - Subclass can do everything superclass can do – **and more**

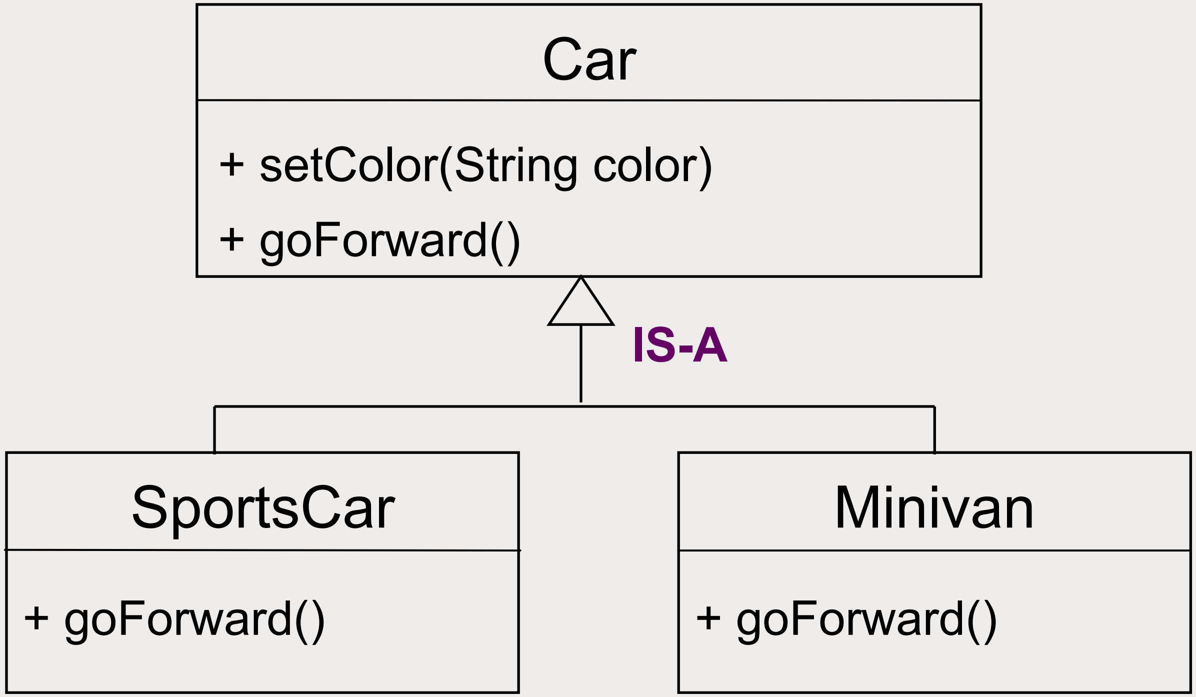
- **Called *specialization***
  - Subclass is a specialization of its superclass

- ***Principle of Substitutability***
  - Anywhere the superclass is expected, a subclass will do

- **We leverage code through inheritance**
  - Useful in "families" of classes – some everyday examples:
    - Shapes: circles, squares, triangles, etc.
    - Cars: sports cars, minivans, trucks, etc.

Relationships between Classes - IS-A

- **Inheritance forms the *IS-A* relationship**
  - Relationship between two classes where one class is derived from another
  - `SportsCar` IS-A `Car`, `Minivan` IS-A `Car`



Inheritance in Code

- **Use the extends keyword to inherit**
  - SportsCar inherits color and setColor()

```
class Car {
  private String color;

  public void setColor(String color) {
    this.color = color;
  }
}
```

```
class SportsCar extends Car {
  private int numDoors;                    // new stuff for SportsCar

  public void setNumDoors(int n) {         // new stuff for SportsCar
    numDoors = n;
  }
}
```

What Gets Inherited?

- **All data members are inherited**
  - Even `private` ones
  - **But:** subclass can't access them directly (private is private)
    - However, it can use getter/setter methods, if provided

- **All methods are inherited *except***
  - Private methods are **not** inherited
  - Constructors are **not** inherited
    - But we'll see later that they can be "borrowed"

- **You do *not* repeat the superclass code in the subclass**
  - You get it for free – it's just there

Overriding Methods

- In addition to adding new behavior, a subclass can ***replace*** inherited behavior
  - The replacing method ***overrides*** the inherited one

- When you want the same method but different behavior

Overriding in Code

- **We want our `SportsCar` to go forward in a different way**
  - `SportsCar` inherits `goForward()` from `Car`
  - But **replaces** it with its own version

```
class Car {
  public void goForward() {
    engageDrive();
  }
}
```

```
class SportsCar extends Car {
  public void goForward() {
    engageHyperDrive();
  }
}
```

Overriding - Caveats

- **Must match the method signature *exactly***
  - Identical parameters – both in type and order
    - Argument list must **not** change
    - If arguments change, it's **overloading**, not overriding

- **Cannot narrow visibility**
  - Can't make the method "more private" (breaks IS-A)

- **Cannot override `final` methods**
  - Final is final

Ellucian

Lab 5.3
Inheritance 1

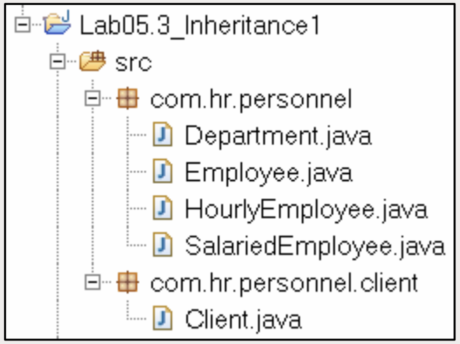
Lab 5.3 - Introducing Subclasses

- **We have two kinds of employees – hourly and salaried**
  - Both have `name` and `hireDate`, just like `Employee`
  - Both work, and they work the **same way**
    - Therefore, it makes sense that they inherit these properties and the `work()` behavior from `Employee`
  - Each one adds subclass-specific data and behavior
    - **`HourlyEmployee`** adds **`rate`** and **`hours`** properties
    - **`SalariedEmployee`** adds **`salary`** property

- **We will see the concepts we just introduced in action**
  - Inheritance of data and behavior
  - Method overriding
  - Principle of Substitutability and the IS-A relationship

Lab 5.3 - Import the Code and Finish It

- **Create a new project *Lab05.3_Inheritance1***
  - With the instructor guiding, import the code from the ***LabSetup/Lab05.3*** directory



- **Review the classes with the instructor**
  - **`Employee` and `Department` have not changed at all**
    - **This is key**: we've extended the system **without touching them**
  - `HourlyEmployee` and `SalariedEmployee`
    - Each one adds specific properties
    - Each one **overrides** the inherited **`toString()`** method

- **Finally, look at `Client`**
  - Look for the TODO and complete it
  - Then run it and note the output

Lab 5.3 - Observations

- **Inheritance**
  - The subclasses get all the `name` and `hireDate` code for free
  - As well as the `work()` method

- **Principle of Substitutability and IS-A**
  - You can add `HourlyEmployee` and `SalariedEmployee` objects to the `Department`
    - Just like you can with regular `Employees`
    - Because an `HourlyEmployee` IS-A `Employee`
  - Department doesn't "know" what kind of `Employees` it has
    - It only "sees" them as `Employee` objects, when in reality they are a mix of several different kinds

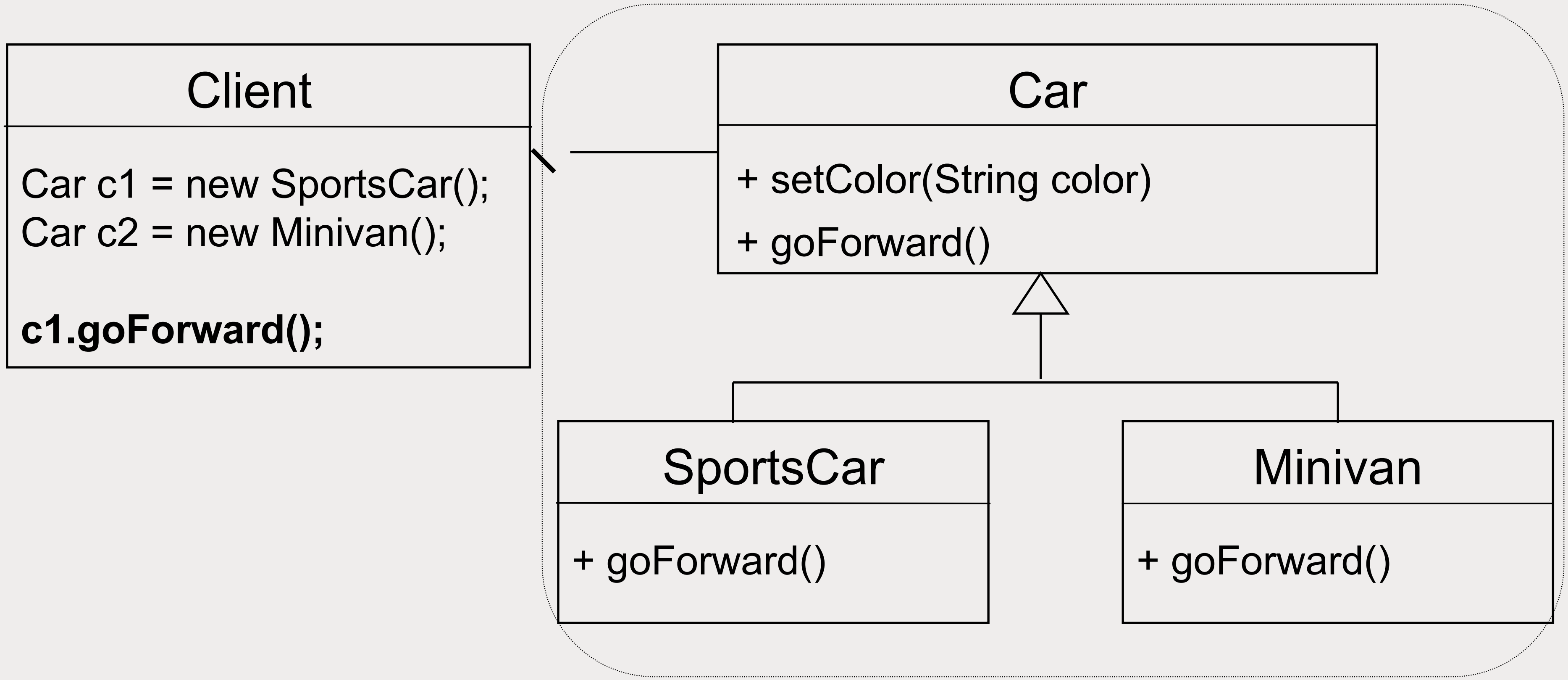
- **Overriding and *polymorphism***
  - Different variations of the `toString()` method

Polymorphism - Defined

- **Different objects respond to the *same message* in *different ways***
- **Another kind of encapsulation – encapsulation of *type***
  - Very helpful for extensibility and clarity of a system
- **The ability to treat "many types" as a single type**
- **To make it work requires:**
  - Inheritance
  - Overriding
  - Can also use ***interfaces*** or ***abstract classes*** (coming next)

Polymorphism - UML Example

- **Which method runs when `c1.goForward()` is called?**
  - Hint: left side creates the reference, right side creates the object



Polymorphism - Runtime Binding

- **Polymorphism is also called *late* or *runtime binding***
- **In the previous example, the `goForward()` method to execute gets determined at *runtime* (vs. compile time)**
- **A reference to a `Car` is an *abstraction***
  - You don't know if it's a `SportsCar`, a `Minivan`, or a regular `Car` – you just know it's a "Car"
  - But at runtime, the correct implementation is invoked
    - Because the JVM knows
- **This is extremely powerful**
  - New "cars" can be introduced to the system, and client code would not have to change

Constructors - Review

- **Special kind of method invoked when you say new**
  - They initialize newly created objects

- **They have the same name as the class**

```
Person p = new Person();  // calling the method named Person
```

- **They have no true return type**
  - **But:** an instance of the class is returned

- **Like other methods, they can be overloaded**

- ***New info* – they are not inherited**

Constructor Chaining - super

- The **super** keyword is a reference to the superclass
- You can call a superclass constructor with **super()**

```
class Car {
  private String color;
  public Car() {
  }
  public Car(String color) {
    this.color = color;
  }
}
```

```
class SportsCar extends Car {
  private int numDoors;
  public SportsCar() {
    super();
  }
  public SportsCar(String color, int n) {
    super(color);  // pass color to superclass constructor
    numDoors = n;  // deal with numDoors here
  }
}
```

Constructor Chaining - Be Careful

- **There are limits – cannot chain what is not there**
  - Note the removal of the no-arg constructor in Car

```
class Car {
  private String color;
                 ?  // can't call methods that don't exist
  public Car(String color) {
    this.color = color;
  }
}
```

```
class SportsCar extends Car {
  private int numDoors;
  public SportsCar() {
    super();        // ERROR: no no-arg constructor in superclass
  }
  public SportsCar(String color, int n) {
    super(color);   // this will still work
    numDoors = n;
  }
}
```

Constructor Chaining ALWAYS Happens

- There's a secret line of code in ***every constructor***

```
class Car {
  private String color;
  public Car(String color) {
    this.color = color;
  }
}
```

```
class SportsCar extends Car {
  // ERROR: won't compile - how can that be? there's no code here!
}
```

- Remember the implicit default constructor?

- ***This*** is your free default constructor
  - See why it won't compile?

```
public SportsCar() {
  super();
}
```

Objects Are Built "Top Down"

- **An object cannot exist until its parent exists**

- **Logically, this makes sense**
  - Inheritance means you get stuff from your parent, right?
  - So how can you get it if your parent doesn't exist yet?

- **Therefore, *a superclass constructor is <u>always</u> called***

- **A superclass constructor call always comes first – *always***
  - To reinforce this for a while, physically type in the call to `super()` every time you write a constructor
  - Because even if you don't type in it, **it's there** (see notes)

Ellucian

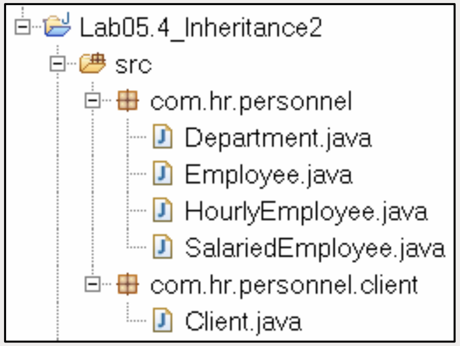
Lab 5.4
Inheritance 2

Lab 5.4 - Variance in the Subclasses

- All employees get paid, but ***not*** the same way
  - Hourly: rate * hours
  - Salaried: salary

- We'll write **`pay()`** methods in the subclasses
- We'll write a **`payEmployees()`** method in **`Department`**
  - Which iterates over its `Employees` and calls **`pay()`** on each
  - Another example of polymorphism in action

- First, let's import the code – create a new project ***Lab05.4_Inheritance2***
  - With the instructor guiding, import the code from the ***LabSetup/Lab05.4*** directory



Lab 5.4 - Complete the Classes

- **Look for the TODOs and complete them**
  - `HourlyEmployee` and `SalariedEmployee` **`pay()`** methods
  - `Department` **`payEmployees()`** method
    - You'll notice a problem here when you try to call `pay()`

```
pay();
The method pay() is undefined for the type Employee
```

    - `Department`'s reference is to **`Employee`**, so it can **only** call methods defined in **`Employee`**
    - Thus, we need to have a `pay()` method in `Employee` itself
    - Go ahead and implement one – but how do you pay a "generic" `Employee`, when you don't have any rate/hours or salary info?

```
public void pay() {
  System.out.println(this.getName() + " is paid...somehow");
}
```

- **Review the client, and then run it and note the output**

Lab 5.4 - Observations

- **Overriding and *polymorphism***
  - Different variations of the `pay()` method

- **Using `super()` to call superclass constructors in the subclasses**
  - Common data (`name`, `hireDate`) passed to superclass ctor
  - Subclass data (`rate`, `hours`; `salary`) dealt with in subclass

- **Department doesn't "know" what kind of `Employees` it has**
  - It only "sees" them as `Employee` objects, when in reality they are a mix of several different kinds
  - This is a good thing, but we ran into a snag
    - We had to write a "dummy" `pay()` method in `Employee`
    - You should not feel comfortable with this solution
    - We will discuss this in the next section

Everything Inherits from `Object`

- **All classes inherit from `Object`**
  - If you don't say `extends`, your class extends `Object`
  - Since all classes ultimately extend `Object`, its methods are guaranteed to be present in **every** object
    - These methods are extremely generic and some of them should be overridden in many of the classes you write

- **Methods from `Object` you will likely need to deal with:**
  - **`equals()`** (seen that one before)
  - **`hashCode()`**
  - **`toString()`** (seen that one before)

Making `Object` Methods Work Right for You

- **You override them to make them work the way you want**

- **`equals()` determines object "equality"**
  - Are two object's "the same"
  - The `equals()` you inherit from `Object` uses `==`
    - This method should often be overridden
    - Generally, you compare the instance variables of the objects

- **`toString()` provides a string representation of the object**
  - Some "sentence" that describes it
  - The `toString()` you inherit from `Object` returns this:
    - Person@876d5e4 (for a `Person` object – see notes)
    - Generally, you want something like this:
      Person: name=Leanne, age=23

Overriding the equals() Method

- **A Person class with name and age properties**
  - Person objects are "the same" if name and age are the same

```
class Person {
  private String name;
  private int age;
  // rest of class not shown

  public boolean equals(Object obj) {
    boolean result = false;
    if (obj instanceof Person) {  // see notes
      Person other = (Person) obj;

      // they are "the same" if name and age are the same
      result = this.getName().equals(other.getName()) &&
        this.getAge() == other.getAge();
    }
    return result;
  }
}
```

Visibility - `protected`

- **We know `public`, `private`, and "package-private" visibility**
  - Subclasses cannot access `private` members

- **The last visibility modifier is `protected`**

- **Two kinds of things can access a `protected` member:**
  1. Subclasses
  2. Other classes in the same package
  - Even if not a subclass

Section Review

1. To build a HAS-A relationship, you declare your class as follows: [T/F]
   `class A has-a B { ... }`
2. Explain delegation.
3. Why is inheritance useful?
4. What's the purpose of overriding?
5. To override a superclass method, the subclass method must _____.
6. What is polymorphism? Explain encapsulation of type.
7. What is constructor chaining? When does it happen?
8. How does a subclass access code from its superclass?
9. An array of `Car` objects can hold `SportsCar` objects, too. [T/F]
10. What is the significance of class `Object` and its methods?

End of Section

- **This slide is intentionally devoid of any useful content**

Ellucian

Section 6
Interfaces and Abstract Classes

Abstract Classes
Interfaces

Section Objectives

- Gain an understanding of abstract classes
- Understand interfaces and their uses
- Learn how to define and implement an interface
- See how interfaces and abstract classes are depicted in UML

Ellucian

Interfaces and Abstract Classes
Abstract Classes

Abstract Classes

Interfaces

What Is an Abstract Class?

- **Usually has one or more `abstract` methods**
  - Though not required
- **Declared using the `abstract` keyword**
- **Cannot be instantiated**
  - No objects of this type can be created
- **If an instance cannot be created, what good is it?**
  - Leveraging code commonality goes in the superclass
  - Polymorphism variance goes in the subclasses
  - Provides an ***abstract type*** for reference variables

```
Car c = new SportsCar();
```

reference type **actual object type**

When to Use an Abstract Class

- **When behavior is *common* to all the subclasses and implemented the *same way*:**
  - Code it in the superclass and it gets inherited by subclasses

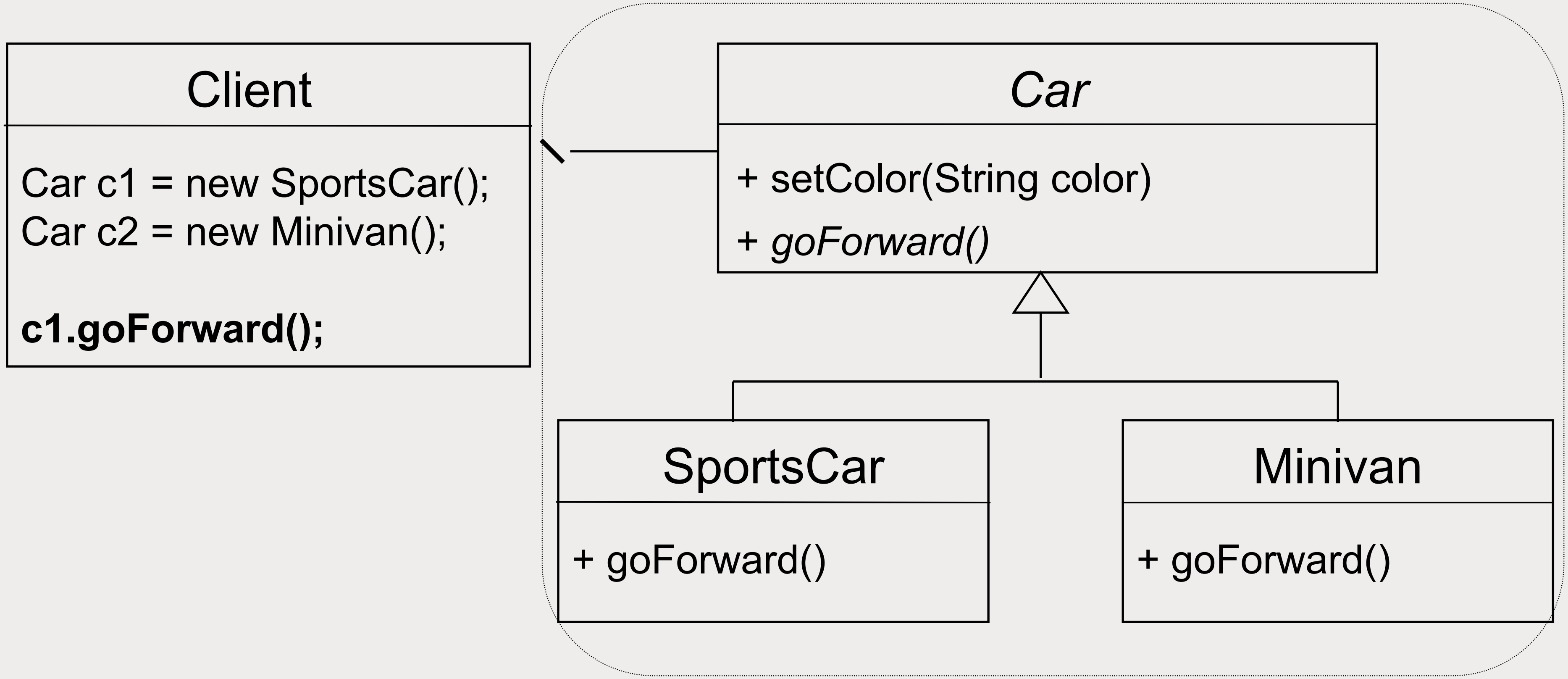
- **When behavior is *common* to all the subclasses but implemented *differently*:**
  - Put the "interface" (abstract method) in the superclass
  - Put the implementation in the subclasses
  - We declare **that** all the subclasses do it, but we don't say **how** they do it
  - Since the method is declared in the superclass, we can call it through an abstract reference type

```
Car c = new SportsCar();
c.goForward();
```

Abstract Class – UML

- **All Cars `goForward()`, but they do it differently**
  - **Declare** the method in `Car`, **implement** it in the subclasses

- **Abstract classes and methods are shown in *italics***



Abstract Class in Code

```
abstract class Car {
  private String color;

  public void setColor(String color) {
    this.color = color;
  }

  public abstract void goForward();  // NOTE: no method body
}
```

```
class SportsCar extends Car {
  private int numDoors;

  public void setNumDoors(int numDoors) {
    this.numDoors = numDoors;
  }

  public void goForward() {
    // override abstract method with concrete implementation
  }
}
```

Ellucian

Lab 6.1
Abstract Classes

Lab 6.1 - Abstract Behavior in Employee

- **We left our HR lab with a "dummy" `pay()` method in `Employee`**

```
public void pay() {
  System.out.println(this.getName() + " is paid...somehow");
}
```

- **We have to put some sort of `pay()` method in `Employee`**
  - Because `Department` has only an `Employee` reference
    - It only "sees" them as `Employee` objects, so it can **only** call methods defined in `Employee`

- **Solution: abstract `Employee` class, abstract `pay()` method**
  - We say **that** `Employee` does `pay()`, but we don't say **how**
  - That's left to the subclasses

Lab 6.1 - Completing the Classes

- **Make a copy of the *Lab05.4_Inheritance2* project and name the copy *Lab06.1_AbstractClasses***
  – Then close the Lab05.4 project

- **Make the following changes:**
  – Declare `Employee` to be an abstract class
  – Make the `pay()` method in `Employee` abstract
  – In the client, change any `Employee` objects to `HourlyEmployee` or `SalariedEmployee`
    - We can't have a "plain" `Employee` – that doesn't make sense, because we need to know how they get paid
    - We need to have a specific kind – hourly or salaried

- **Run the client and note the output**

Ellucian

Interfaces and Abstract Classes
Interfaces

Abstract Classes

Interfaces

Interface - Defined

- **Used to define (but not implement) some behavior**
  - **What** needs to be done, but not **how** it needs to be done
- **Structurally, an interface is a group of related methods, *all* of which are *abstract* (no implementation)**
  - Similar to an abstract class, but **all** methods are abstract
  - An interface cannot define instance variables
    - Instance variables are "implementation"
- **Like a *contract***
  - If a class ***implements*** an interface, it "signs the contract"
    - Agreeing to implement all of its methods with code

Interfaces Define Roles

- **Can also think of interfaces as *roles* that a class can play**
- **When I am teaching, I am in the role of `Instructor`**
  - My students "see" only my `Instructor` behavior
- **When I am home, I am in the role of `Parent`**
  - My kids "see" only my `Parent` behavior
- **Interfaces are about coupling, or rather decoupling**
  - The client sees the **interface type**, not the class type
- **A class can implement more than one interface**
  - I can sign multiple contracts as long as I fulfill them all
  - I can **not** promise to be a subclass to more than one class

IS-A but Not Inheritance

- **Different classes that are not related inheritance-wise, can still sign the same interface contract**

- **In the next lab, we'll examine a `Taxpayer` interface**
  - It has one method: `payTaxes()`

- **Our HR Employees pay taxes – they implement Taxpayer**
  - Employee IS-A Taxpayer

- **Corporations also pay taxes – they implement Taxpayer**
  - Corporation IS-A Taxpayer

- **Employee and Corporation are not in the same hierarchy**
  - Corporation IS-A Employee? (no)
  - Employee IS-A Corporation? (no)

Specifications Use Interfaces

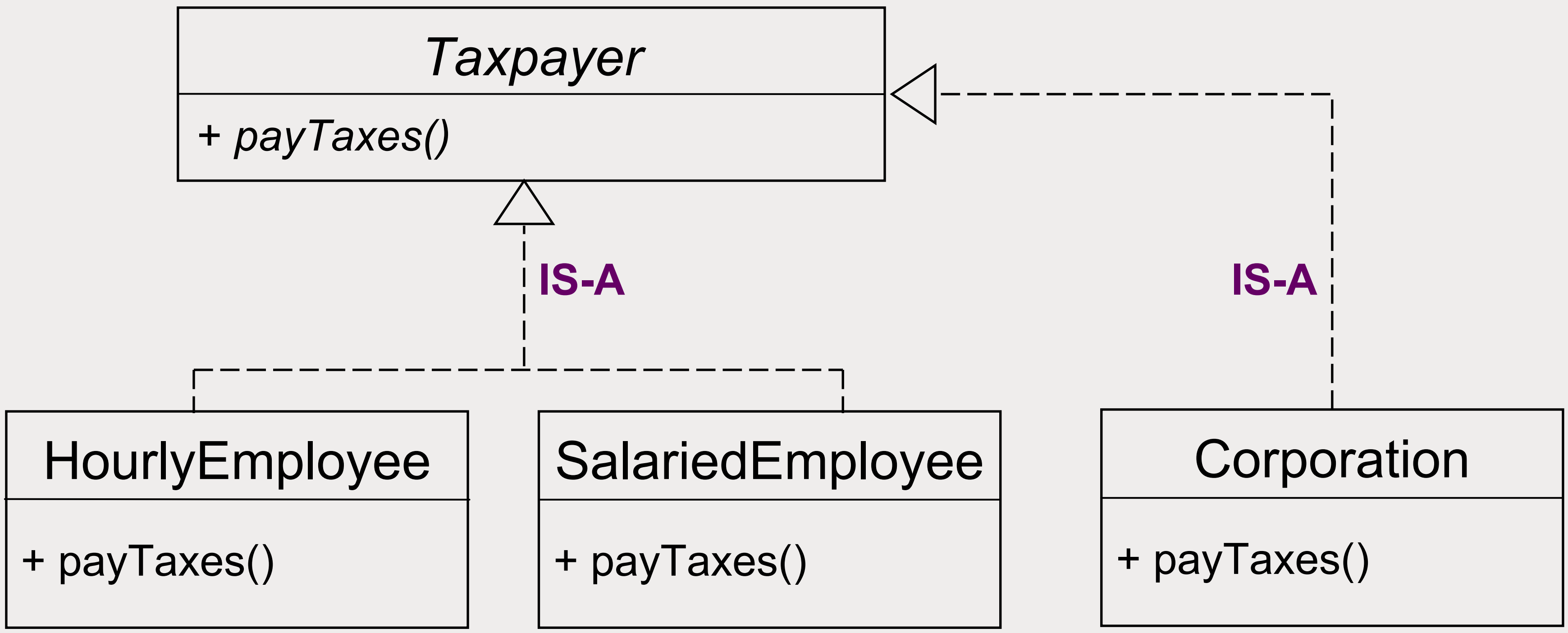
- **Many Java specifications and APIs are based on interfaces**

- **JDBC (Java Database Connectivity) was the first (1996)**
  - The `java.sql` package is almost all interfaces
    - It's an **API** for talking to databases from Java
  - Sun defined **what** should be done, not **how** it should be done
    - That's left to the database vendors
  - Vendor classes **implement** the interfaces for their particular database – collectively, these classes are called a "driver"
  - You benefit, because you're coupled only to JDBC, not to Oracle, IBM, or Sybase classes

- **Most JavaEE specifications are defined in packages that consist mostly of abstract classes and interfaces**

Interfaces - UML

- **This diagram shows only the *interface realization* of these classes**
  - `HourlyEmployee` and `SalariedEmployee` still subclass `Employee` (but `Corporation` does not)



Interface in Code

- Use the **implements** keyword to implement an interface

```
public interface Taxpayer {
  public void payTaxes();     // no need to say abstract (see notes)
}
```

```
class HourlyEmployee implements Taxpayer {
  ...
  public void payTaxes() {    // must implement the interface method
    // code to pay taxes on hourly wages
  }
}
```

```
class Corporation implements Taxpayer {
  ...
  public void payTaxes() {    // must implement the interface method
    // code to pay taxes on net income
  }
}
```

Concrete Superclass, Abstract Class, or Interface?

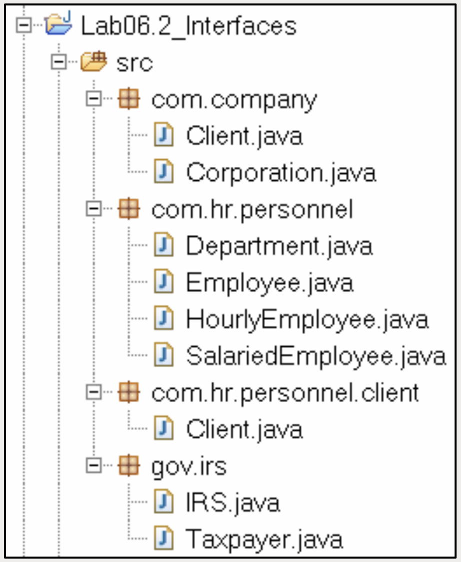
- **Motivation for all three is the same – decoupling client from actual implementation class**
- **Concrete superclass works best when *all behavior* is common**
- **Interface works best when *all behavior* varies**
- **Abstract class works best when there is a *combination* of commonality and variability**
  - Some common behavior has the same implementation
    - Concrete method in superclass, gets inherited by subclasses
  - Some common behavior has different implementations
    - Abstract method in superclass, overridden in subclasses

Ellucian

Lab 6.2
Interfaces

Lab 6.2 - Interfaces

- **Create a new project *Lab06.2_Interfaces***
  - Import the code from ***LabSetup/Lab06.2***



- **It's a good number of files, but it's not really that complicated – let's break it down**
  - **Taxpayer** is an interface with one method
    - **`payTaxes()`**
  - `IRS` is a class that references `Taxpayers`
    - That is, **any class** that **implements** `Taxpayer` – `IRS` calls `payTaxes()` on all its `Taxpayers`
  - `HourlyEmployee`/`SalariedEmployee` **implement** `TaxPayer`
  - `Corporation` also **implements** `Taxpayer`
  - `com.hr.personnel.client.Client` creates employees and registers them with the IRS
  - `com.company.Client` does the same for `Corporation`

Lab 6.2 - See It Work

- **Review the classes with the instructor**
  - The key takeaway is that `IRS` "sees" its objects only as **`Taxpayers`** (interface type)
    - They could be `HourlyEmployees`, `SalariedEmployees`, `Corporations`, or any other class that implements `Taxpayer`
    - The IRS doesn't really know ... or care! As long as they `payTaxes()`

- **Review, and then run the HR client**
  - Our existing employee classes work with the IRS

- **Review, and then run the corporate client**
  - The `Corporation` class also works with the IRS

Section Review

1. What makes a class abstract? Why do we have them?
2. What is defined in an interface?
3. What is the difference between an interface and an abstract class?
4. How does a class implement an interface?
5. Explain why an interface is like a "contract."
6. Can a class implement more than one interface?
7. When should you use an interface?
8. When should you use an abstract class?
9. How are interfaces visually depicted in UML?

Ellucian

Section 7
Collections

Collections Overview
Lists, Sets, and Maps
Using Collections
Odds and Ends

Section Objectives

- Gain an understanding of the collection classes in Java
- Explain the differences between `List`, `Set`, and `Map`
- Learn how to use collections, and how to iterate over them
- Explain the benefits of type-safe collections
- Explain the benefits of autoboxing
- Describe how collections are sorted

Ellucian

Collections
Collections Overview

Collections Overview

Lists, Sets, and Maps

Using Collections

Odds and Ends

Collection Framework

- **The *Java Collections Framework* is about storing and manipulating groups of objects**

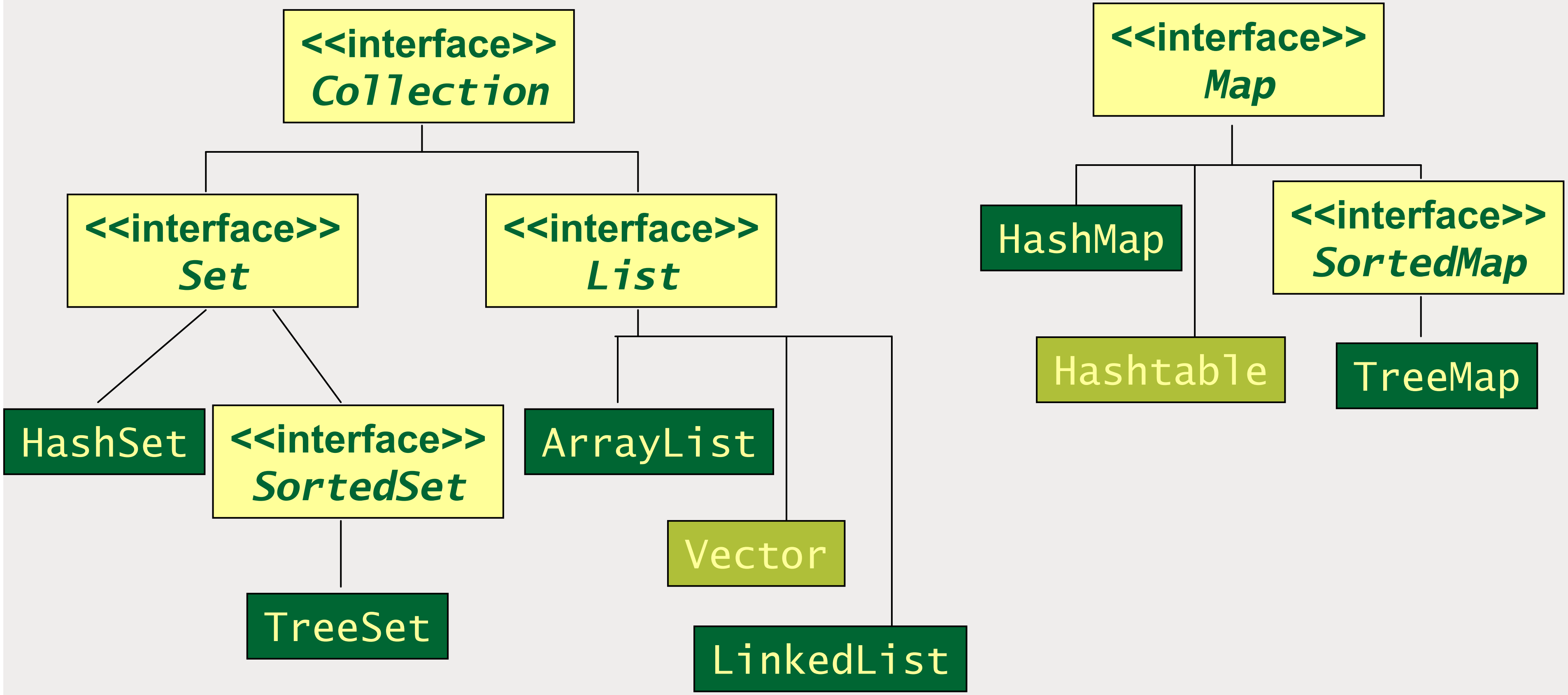
- ***Any type of object* can be stored, retrieved, and manipulated as an element of a collection**
  - "Any type of object" means `Object`
    - Since everything is ultimately an `Object`, anything can be stored

- **Follows common interfaces for interoperability**
  - The designers followed good OO practices here

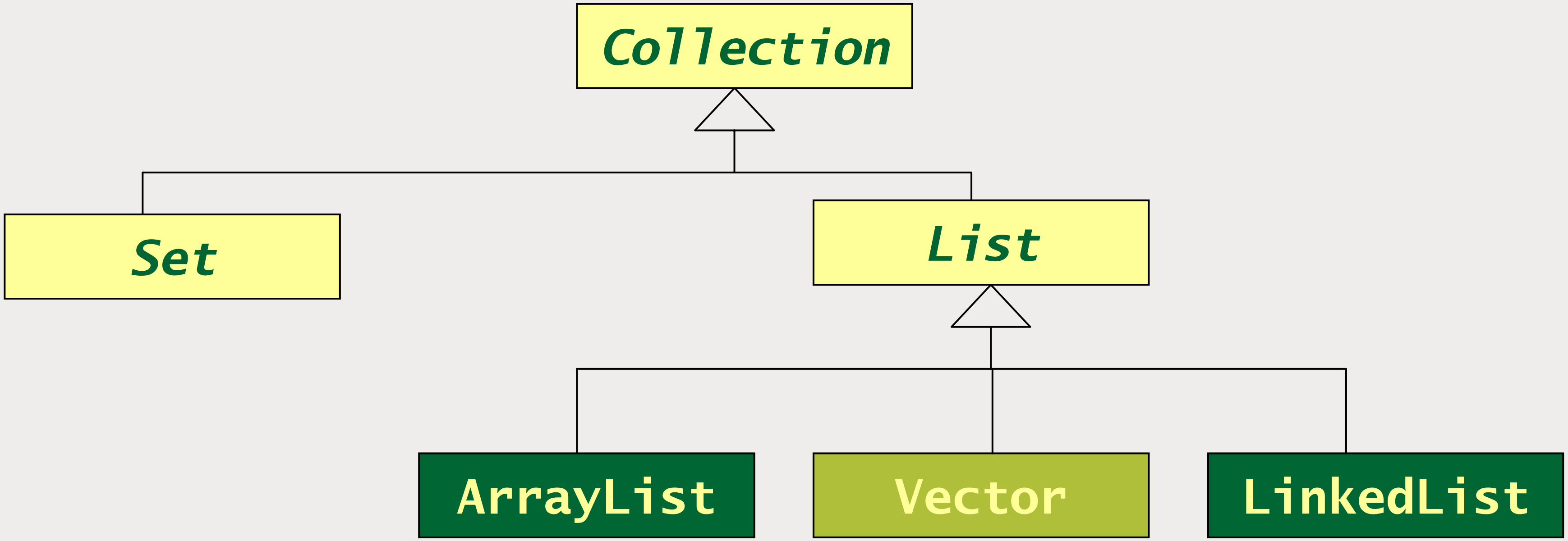
Collections API

- **Found in `java.util` package**
- **Some good OO concepts modeled**
- **`Collection` is an interface – the top of the hierarchy**
- **`Collections` is a class**
  - An all-`static` class providing utility methods, e.g., to sort a collection
- **The Javadoc is *extremely* useful here**
  - Tells you how each type of collection works, and what the methods do

Collections Hierarchy



Code to the Interface



Do this:

```
List list = new ArrayList();
```

Not this:

```
ArrayList list = new ArrayList();
```

Ellucian

Collections
Lists, Sets, and Maps

Collections Overview

Lists, Sets, and Maps

Using Collections

Odds and Ends

Interfaces: `Collection`

- **`Collection` interface is extended by the `List` and `Set` interfaces**
  - Methods used most often are bolded
  - Note that the parameter and return types are all `Object`

```
public interface Collection {
  boolean add(Object element);
  boolean addAll(Collection c);
  void clear();
  boolean contains(Object o);
  boolean containsAll(Collection c);
  boolean isEmpty();
  Iterator iterator();
  boolean remove(Object o);
  boolean removeAll(Collection c);
  int size();
  Object[] toArray();
}
```

Interfaces: `List`, `Set`, `Map`

- **`List`**
  - **Ordered**, or sequenced
    - Using an index, just like an array
  - **Allows duplicates**

- **`Set`**
  - **Unordered**
    - No index
  - **No duplicates**
    - "Duplicate" objects are "the same" according to the class's `equals()` method

- **`Map`**
  - **key-value** pairs

Lists

- ***Indexed*, like arrays**
  - Lists maintain the positions of the objects in it
  - Zero-indexed, like arrays

- ***Allows duplicates***
  - The `equals()` and `hashCode()` methods are not used (they are with `Set` – we'll see that soon)

- **Implementations:**
  - **`ArrayList`** (most popular)
  - `LinkedList`
  - `Vector` (legacy)

List Interface

- **List interface extends Collection to add more methods**
  - Specifically, you can add/retrieve objects by index
  - Notice the use of interface inheritance
  - A List IS-A Collection

```
public interface List extends Collection {
  void add(int index, Object element);
  Object get(int index);
  Object set(int index, Object element);
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator listIterator();
  ListIterator listIterator(int index);
  List subList(int from, int to);
}
```

Sets

- ***Unordered*** **(no index),** ***no duplicates***
  - Relies on your objects' **`equals()`** and **`hashCode()`** methods to determine if an object to be added is already in there
    - More on this later

- **If you are going to put objects in a `Set`, then *always* implement the `equals()` and `hashCode()` methods**
  - Or your sets will contain duplicates

- **Implementations:**
  - **`HashSet`** (most popular)
  - `LinkedHashSet` (maintains "add order")
  - `TreeSet` (sorts by "natural order")

  $\left.\begin{array}{l}\\ \\ \end{array}\right\}$ see notes

Maps

- **Store *key-value pairs* (`Object`, `Object`)**
  - Each object added to the map sits next to a key
  - Key is used to find the object in the map

Key	Value
"student1"	"Jason"

"entry"

- ***No duplicate keys***
  - Adding an object with the same key **replaces** the old object with the new one
  - Key's `equals()` and `hashCode()` methods are used to determine if two keys are the same

- **Implementations:**
  - **`HashMap`** (most popular)
  - `TreeMap` (also implements `SortedMap`)
  - `Hashtable` (legacy)

Map Interface

- **Add entries as key-value pairs** `map.put(key, value)`
- **Retrieve values by key** `map.get(key)`

```
public interface Map {
  void clear();
  boolean containsKey(Object o);
  boolean containsValue(Object o);
  Set entrySet();
  Set keySet();
  Object get(Object key);
  Object put(Object key, Object value);
  void putAll(Map map);
  Object remove(Object key);
  Collection values();
}
```

Ellucian

Collections
Using Collections

Collections Overview
Lists, Sets, and Maps
Using Collections
Odds and Ends

List - Example

```
public class ListExample {

  public static void main(String[] args) {

    // code to the interface
    List states = new ArrayList();

    // objects can be added to the end of the list
    states.add("PA");  // index 0
    states.add("WA");  // index 1

    // they can be inserted at a particular point
    states.add(1, "GA");  // "WA" shifts to index 2

    int indexOfPA = states.indexOf("PA");  // returns 0

    // required to cast Object to String (see notes)
    String state = (String) states.get(1);  // "GA"

    int size = states.size();  // size is 3
  }
}
```

Set - Example

```
public class SetExample {

  public static void main(String[] args) {

    // code to the interface
    Set names = new HashSet();

    names.add("Susan");
    names.add("Danny");
    names.add("Danny");  // NOT added - duplicate

    int size = shapes.size();  // size is 2

    if (names.contains("Susan")) {
      ...
    }
  }
}
```

Map - Example

```
public class MapExample {

  public static void main(String[] args) {

    // code to the interface
    Map stateCaps = new HashMap();

    stateCaps.put("GA", "Atlanta");                  // key-value pair
    stateCaps.put("WA", "Olympia");                  // "WA"-"Olympia"

    // required to cast Object to String (see notes)
    String gaCap = (String) stateCaps.get("GA");  // get by key "GA"
    String waCap = (String) stateCaps.get("WA");  // "Olympia"

    if (stateCaps.containsKey("FL")) {
      ...
    }
  }
}
```

Iterators

- Used to walk through collections – every `Collection` has an `Iterator` object
  - Get the `Iterator` by calling the `iterator()` method on the `Collection`

- `Iterator` methods
  - `boolean hasNext()`
  - `Object next()`

Iterator - Example

```
public class IteratorExample {

  public static void main(String[] args) {

    // code to the interface
    List states = new ArrayList();

    // add some objects to it
    states.add("PA");
    states.add("WA");

    Iterator i = states.iterator();

    while (i.hasNext()) {
      String state = (String) i.next();
      ...
    }
  }
}
```

- **Iteration can also be done using a for loop (see notes)**

Generics

- **Introduced in Java 5**
- **For our purposes, you just need to think of them as *type-safe collections***
  - Instead of being able to hold any type of object, they are only allowed to hold the type you define
- **Unless you need a collection of mixed types (rare), *always* use a type-safe collection**
- **The `Collection` below can only hold `Car` types**
  - Or what else could it hold? (think IS-A here)

```
// usually pronounced "collection of cars"
Collection<Car> cars = new ArrayList<Car>();
```

Type-Safe Collections - Example

```
public class GenericListExample {

  public static void main(String[] args) {

    // code to the interface
    List<String> states = new ArrayList<String>();

    states.add("PA");
    states.add("WA");

    // not allowed to add anything but Strings - compiler checks
    states.add(new Integer(5));  // ERROR: not a String

    // no casting needed - because it can only hold Strings
    String state = states.get(0);
  }
}
```

for-each Loop

- **Works just like it does for arrays**
- **Automatically gets an `Iterator` and iterates through your `Collection`**

```
public class ForEachListExample {

  public static void main(String[] args) {

    Collection<String> states = new ArrayList<String>();

    states.add("PA");
    states.add("WA");
    states.add("FL");

    // pronounced "for each String state in states"
    for (String state : states) {
      System.out.println(state);
    }
  }
}
```

Ellucian

Lab 7.1
Using Collections

Lab 7.1 - Using Collections

- **Arrays are old and fixed length**
  - We've been fudging the length by just making it 100
  - Adding/removing elements requires us to keep track of index
    - Hence our `currentIndex` variable in `Department`

- **Let's replace our array of `Employees` with a collection**

- **Make a copy of the *Lab06.1_AbstractClasses* project and name the copy *Lab07.1_Collections***
  - Then close the Lab06.1 project

Lab 7.1 - Using Collections

- **In Department, change the `Employee` array to be a `Collection<Employee>`**
  - Remove the `currentIndex` variable – no longer needed
  - This:

```
private Employee[] employees = new Employee[100];
private int currentIndex = 0;  // for dealing with array
```

  - Becomes this: (see notes)

```
private Collection<Employee> employees = new ArrayList<Employee>();
```

- **Change `listEmployees()`, `workEmployees()`, and `payEmployees()` to use a for-each loop**

```
for (Employee emp : employees) {
  ...
}
```

Lab 7.1 - Using Collections

- **Change the `addEmployee()` method, and add a `removeEmployee()` method**
  - Because we're using a collection now, these are very simple

```
// helper methods to work with the collection
public void addEmployee(Employee emp) {
  employees.add(emp);
}
public void removeEmployee(Employee emp) {
  employees.remove(emp);
}
```

- **Run the client and note the output – should work as before**

Ellucian

Collections
Odds and Ends

Collections Overview
Lists, Sets, and Maps
Using Collections
Odds and Ends

`equals()` and `hashCode()`

- **We know what `equals()` means – "the same"**
- **What does `hashCode()` mean?**
  - `hashCode()` should return a ***unique integer*** for that object
    - If two objects have **different** hash codes, they are "different"
    - If two objects have **the same** hash code, this does **not** mean they are "the same" – this could just be a coincidence

- **Sets disallow duplicates by *first* calling `hashCode()`, and then *possibly* calling `equals()`, when an object is added**
  - If the hash code of the object being added **doesn't** match any of those already in the `Set`, the object is added
    - Because it has a different hash code, it's definitely "different"
  - If the hash code **does** match one of the existing elements, then `equals()` is called
    - Because equal hash codes could just be a coincidence

Autoboxing

- **Introduced in Java 5**

- **Collections hold objects – *not* primitives**
  - What if you want a `Collection` of **`ints`**?
  - You can't have one – **but** you **can** have a `Collection` of **`Integer`** objects
    - This is one of the reasons we have the wrapper classes!

- **With autoboxing, you can (pretend to) add primitives to a collection**
  - Behind the scenes, wrapper objects are created and added

- **When you take them out, they are auto-unboxed**
  - "Unwrapped" and returned to you as the primitive

Autoboxing - Example

- **Without autoboxing**

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(new Integer(38));        // wrap 38 in an Integer object
...
Integer ageInteger = ages.get(0);
int age = ageInteger.intValue();  // unwrap it
```

- **With autoboxing**

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(38);           // 38 is autoboxed into new Integer(38)
...
int age = ages.get(0);  // 38 is auto-unboxed from Integer(38)
```

Sorting Collections - Overview

- **Collections can be sorted two ways**
  1. "Natural" order
     - Strings "alphabetically" (see notes), numbers increasing, etc.
  2. Any way you want

- **Determining natural order requires the class to implement the `Comparable` interface**

```
public int compareTo(Object other)
```

  – From the Javadoc:
    - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

- **To sort some other way, you write a separate class that implements the `Comparator` interface**

Sorting Collections - Making It Happen

- A `List` can be sorted with `Collections.sort()`
  - `static void sort(List list)` – natural order
  - `static void sort(List list, Comparator c)`
    - By supplied `Comparator`
- For a `Set`, use an implementation of `SortedSet` (`TreeSet`)
  - Maintains order at all times, defined at instantiation (ctor)
  - `TreeSet()` – natural order
  - `TreeSet(Comparator c)` – by supplied `Comparator`
- For a `Map`, use an implementation of `SortedMap` (`TreeMap`)
  - Maintains order at all times, defined at instantiation (ctor)
  - `TreeMap()` – natural order
  - `TreeMap(Comparator c)` – by supplied `Comparator`

Ellucian

Lab 7.2 (optional)
Odds and Ends

Lab 7.2 - Odds and Ends (optional)

- **If you wish, you can look over some examples that demonstrate `Sets`, autoboxing, and sorting**

- **Create a new project *Lab07.2_OddsAndEnds***
  - Import the code from ***LabSetup/Lab07.2***

Section Review

1. Which collection would you use if the position of the objects is important?
2. Which collection acts like a "table" of key-value pairs?
3. Which collection does not allow duplicates? What's a duplicate?
4. What scenario would allow the same object to be in a `Map` twice?
5. Type-safe collections are useful when you have a collection of just one type, but this is a rare situation. [T/F]
6. Explain the significance of the `equals()` and `hashCode()` methods when dealing with a `Set`.
7. Explain autoboxing.
8. How do you sort a collection?

End of Section

- This slide is intentionally devoid of any useful content

Ellucian

Section 8
Exceptions

Overview
Exception Handling

Section Objectives

- **Gain some understanding of exceptions**
  - Including the difference between checked and unchecked exceptions
- **Understand how to create your own exception classes**
- **Learn how to declare and throw exceptions**
- **Explain why we would use a `try-catch` block or a `finally` block**
- **Learn how to handle exceptions**

Ellucian

Exceptions Overview

Overview

Exception Handling

Reporting and Trapping Errors

- What do you do when something "bad" happens in your code?

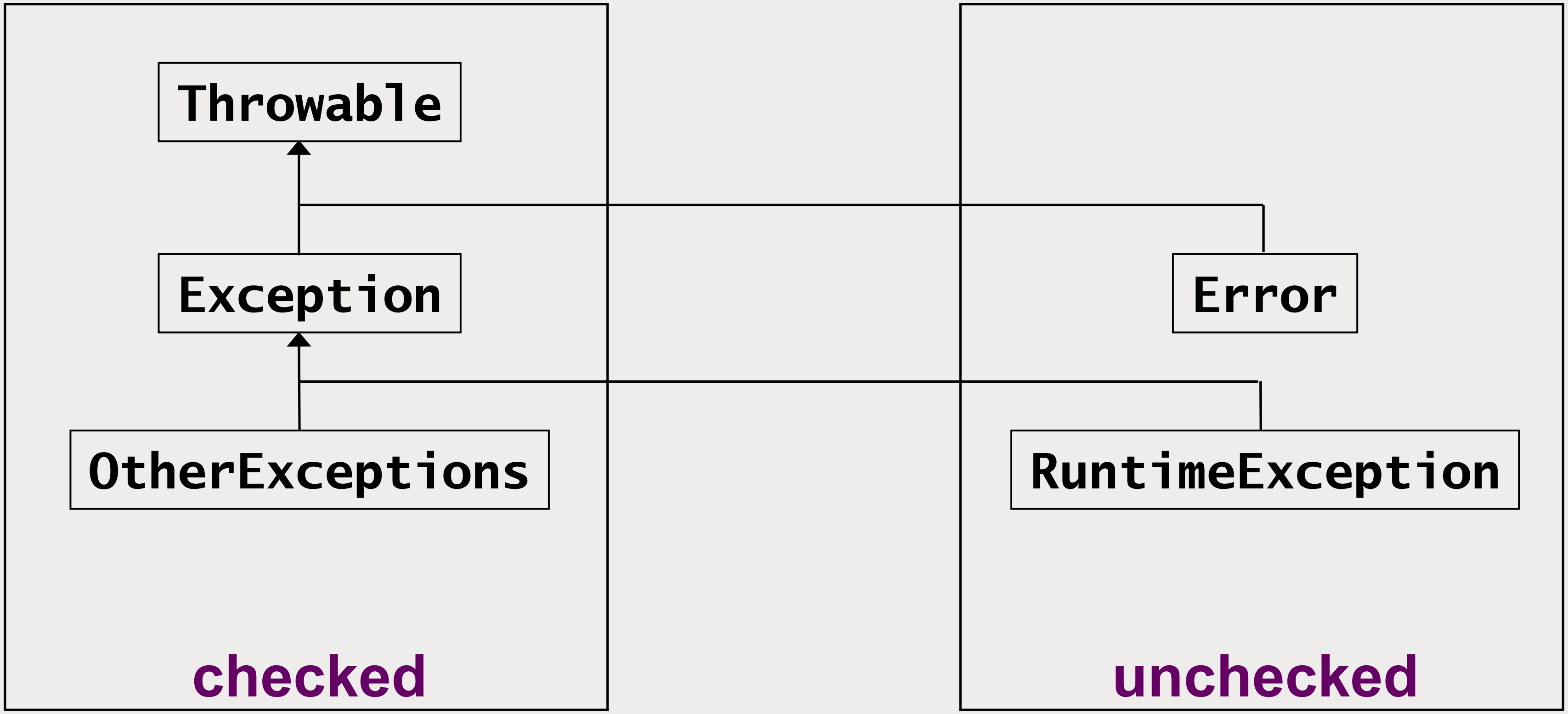
- ***Exceptions*** is Java's mechanism for handling this
  - Two types of exceptions
    - ***Checked*** (compiler forces you to deal with it)
    - ***Unchecked*** (compiler leaves this up to you)

- `Exception` is a class with many subclasses
  - `IOException`, `NullPointerException`, etc.

- If you are calling a method that might throw an exception, it will have a `throws` clause in the declaration

```
public String readLine() throws IOException
```

Checked vs. Unchecked Exceptions - Illustrated



Common Exceptions

Exception	Why or when it happens
`ArrayIndexOutOfBoundsException`	Accessing array with invalid index value
`ClassCastException`	Cast a reference that is not an IS-A
`IllegalArgumentException`	Method receives argument different than expected
`IllegalStateException`	State of object not suitable for the operation
`NullPointerException`	Accessing an object whose current value is `null`
`NumberFormatException`	Data conversion method received a string that was not a number, e.g., `Integer.parseInt()`
`OutOfMemoryError`	Self-explanatory
`StackOverflowError`	Infinite loop
`NoClassDefFoundError`	JVM can't find the class on the classpath

Defining Custom Exceptions

- **Give it a good name and provide a few constructors**
  - Default or no-arg constructor
  - One that takes a message (indicating why it was thrown)
  - One that takes a nested exception
  - One that takes a message and a nested exception

```
package com.company;
class MyOwnException extends Exception {
  public MyOwnException() {
  }
  public MyOwnException(String message) {
    super(message);
  }
  public MyOwnException(Throwable cause) {
    super(cause);
  }
  public MyOwnException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

Common Methods from `Throwable`

- `getMessage()` – returns the string message indicating why it was thrown
- `toString()` – returns classname of exception followed by the string message
- `printStackTrace()` – prints to stdout all the information associated with this exception
  - Including all the methods involved between the initial thrower and the ultimate catcher

Ellucian

Lab 8.1
Defining a Custom Exception

Lab 8.1 - Custom Exception

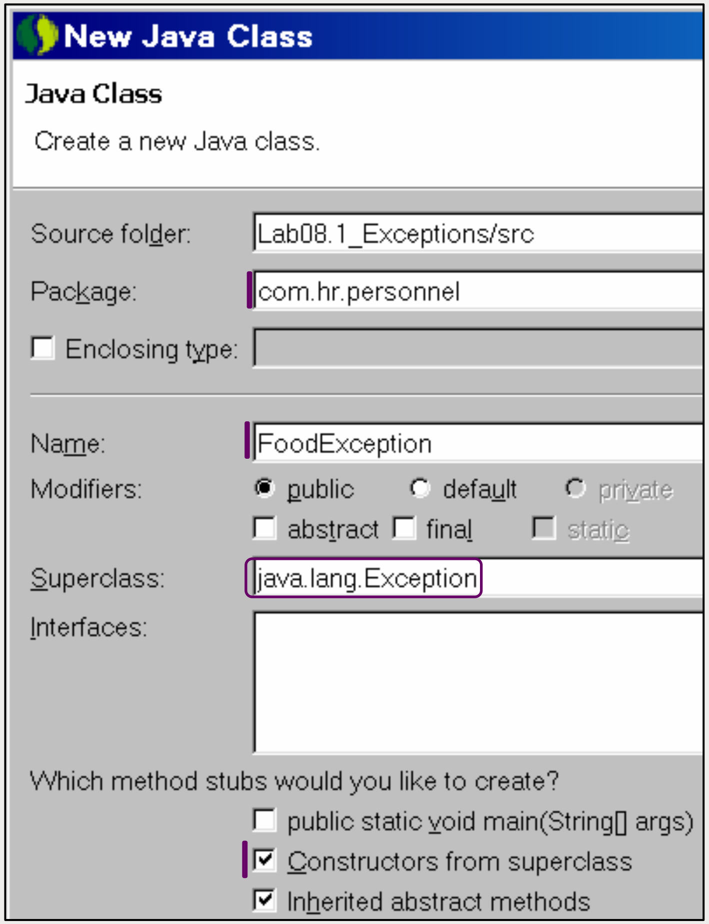
- **Make a copy of the *Lab04.4_Packages* project and name the copy *Lab08.1_Exceptions***
  - Then close the Lab04.4 project

- **Recall that `Person` has an `eat(String food)` method**
  - If a "disliked" food is passed in, we want to throw a **`FoodException`**

- **We need to create this custom exception class**
  - We'll put it in the **`com.hr.personnel`** package (where `Person` is)
  - Right-click on the package and choose **New → Class**

Lab 8.1 - Custom Exception

- **Fill in the wizard as shown**
- **Then click Finish**
- **Review the exception class**
  - It's very simple – just has a few constructors
  - This is true of most exception classes



Ellucian

Exceptions
Exception Handling

Overview

Exception Handling

Exception Handling

- **To handle an exception there are two choices:**
  - Catch (and possibly re-throw)
  - Don't catch

- **Method callers "handle" them by *catching* thrown exceptions**

- **Exceptions that are not caught by anyone will end the program**
  - Perfectly legal to do this
  - Error message will be printed to the console and JVM exits
  - Call stack available for review

The Throwing Side

- **A method that might throw an exception declares this fact with the `throws` keyword**
  - **Checked** exception → **must** declare this
  - **Unchecked** exception → **may** declare this, but don't have to
    - Sometimes declared anyway, to indicate to the caller that this could happen to them when the method executes

```
class Car {
  void drive() throws FlatTireException {   // a checked exception
    if (tirePressure < 10) {
      throw new FlatTireException("Tire pressure: " + tirePressure);
    }
    else {
      // do driving behavior
    }
  }
}
```

Declaring Multiple Exceptions

- **What if several things could possibly go wrong?**

- **Method declares all exceptions it might throw**
  - Just separate with commas

```
class Car {
  void drive() throws FlatTireException, OutOfGasException,
    RegistrationExpiredException {
    ...
  }
}
```

- **NOTE: *only one* of them will be thrown upon execution**
  - It's **not** throwing multiple exceptions at once
  - It's just a list of all possible exceptions **could be thrown**

The Catching Side

- A method that calls another method that might throw an exception plans for this possibility with **try-catch** blocks
  - **Checked** exception → **must** `try-catch` (caveat later)
  - **Unchecked** exception → **may** `try-catch` (not required to)

- `catch` block takes the thrown exception as a parameter

```
void goToWork() {
  try {
    Car c = new Car();
    c.start();
    c.drive();                    // this might throw an exception
    c.stop();
  }
  catch (FlatTireException e) {  // e points to the exception object
    // handle it somehow
    System.out.println(e);        // what does this do?
  }
}
```

Exceptions and Program Flow - Example

- **In this example, assume the exception is thrown**
  - Control jumps to `catch` block
  - `c.stop()` never executes

```
void goToWork() {
  try {
    Car c = new Car();
    c.start();
    c.drive();              // EXCEPTION
    c.stop();
  }
  catch (FlatTireException e) {
    // handle it somehow
    System.out.println(e);  // what does this do?
  }
}
```

Handling Multiple Exceptions - Example

- **Exception type thrown dictates the catch block chosen**
  - If exception is not thrown, `try` completes normally
  - Order matters – most general exception should be last

```
void goToWork() {
  try {
    Car c = new Car();
    c.start();
    c.drive();   // ONE of 3 possible exceptions might be thrown
    c.stop();
  }
  // NOTE: these "handle" the exception "silently" (which is cheap)
  catch (FlatTireException e) {
  }
  catch (OutOfGasException e) {
  }
  catch (Exception e) {
  }
}
```

What does this catch?

Catch and Re-Throw

- **Can catch and re-throw the same exception, or a new one**
  – Catch it and do something
  – Then throw it back up the call stack

- **Since it may propagate out of your method, must declare it**

```
void goToWork() throws FlatTireException {
  try {
    Car c = new Car();
    c.start();
    c.drive();
    c.stop();
  }
  catch (FlatTireException e) {
    Logger.logMessage(e);  // do something here first
    throw e;               // then re-throw it back up the stack
  }
}
```

try-catch-finally

- **The `finally` block will *always* execute – guaranteed**
  - If present, it executes whether or not there is an exception
  - Remember, as soon as an exception is thrown, control jumps to a matching `catch` block
  - In this example, we need to `stop()` the `Car` in all cases

```
void goToWork() {
  Car c = new Car();       // variable scope: need c in finally block
  try {
    c.start();
    c.drive();
  }
  catch (FlatTireException e) {
    System.out.println(e.getMessage());
  }
  finally {
    c.stop();              // always want to turn off the car
  }
}
```

Not Catching an Exception

- **On the catching side, we said this before:**
  - **Checked** exception → **must** `try-catch` (caveat later)

- **Now for the caveat:**
  - **You can *ignore* the `try-catch` completely and declare it in your `throws` clause**
  - But if an exception is thrown back at you, then what?
  - It propagates back to whomever called **you**
  - Goes back up the call stack until it's caught (see notes)

```
void goToWork() throws FlatTireException {  // not handled here
  Car c = new Car();
  c.start();
  c.drive();                  // this STILL might throw an exception
  c.stop();
}
```

Ellucian

Lab 8.2
Handling Exceptions

Lab 8.2 - Throwing FoodException

- Continue to work in the ***Lab08.1_Exceptions*** project
  - First, delete any code in `PersonClient`'s `main()` method

- In **`Person`**, find the **`eat(String food)`** method
  - Declare that the method might throw a `FoodException`
    - By adding a `throws` clause to the method's signature
  - If "pizza" is passed in, throw a `FoodException`
    - This is done with `throw new FoodException(...)`

```
public void eat(String food) throws FoodException {
   if (food.equals("pizza")) {
      throw new FoodException("Sorry, I don't eat pizza");
   }
   else {
      System.out.println(this.getName() + " is eating " + food);
   }
}
```

Lab 8.2 - Catching FoodException in the Client

- **In PersonClient's main(), create a new Person object**
  - You must put the following "eat" calls in a try-catch (why?)
  - Call the eat(String food) method, passing in "salad"
  - After passing in "salad", call the method again, this time with an argument of "pizza" – this will trigger the exception
    - Since the "pizza" call throws an exception, the "chicken" call is never reached – control immediately jumps to the catch block

```
Person p = new Person(...);
try {
  p.eat("salad");
  p.eat("pizza");  // throws exception, control jumps to catch block
  p.eat("chicken");
}
catch (FoodException e) {
  e.printStackTrace();
}
```

- **Run the client and note the results**

Lab 8.2 - Runtime Exceptions

- Back in **Person**, find the **setAge()** method
  - If an invalid age is passed in, throw an **IllegalArgumentException** (unchecked exception)
  - Because it's unchecked, you don't need to declare it in a throws clause (although you can)

```
public void setAge(int age) {
  if (age > 0) {
    this.age = age;
  }
  else {
    throw new IllegalArgumentException("Invalid age: " + age);
  }
}
```

Lab 8.2 - Runtime Exceptions

- **In `PersonClient`'s `main()` method, call `setAge()` with a negative value *before* the `try-catch` block**
  - Because `setAge()` throws an unchecked exception, you don't need to call it in a `try-catch` block (although you can)

```
Person p = new Person(...);

p.setAge(-10);  // throws exception, unhandled, program terminates

try {
  p.eat("salad");
  p.eat("pizza");
  ...
```

- **Run the client and note the results**

Section Review

1. What's the difference between checked and unchecked exceptions?
2. When will the compiler **not** tell you about an exception that could occur?
3. How many `catch` blocks can a `try` block have?
4. When does the `finally` block run? What is its purpose?
5. How do you create your own exception classes?
6. The nice thing about exception handling in Java is that, ultimately, it's optional because you can always just rethrow everything. [T/F]

End of Section

- This slide is intentionally devoid of any useful content

Ellucian

Section 9
Unit Testing Overview

Writing Unit Tests
Test Run Lifecycle

Section Objectives

- Understand the basic principles of unit testing
- Learn how to write JUnit test cases and how assertions work
- Understand the JUnit test run lifecycle

Ellucian

Unit Testing Overview
Writing Unit Tests

Writing Unit Tests

Test Run Lifecycle

Overview of JUnit

- ***JUnit*** **is an automated unit testing framework for Java**
  – Standard framework used in the industry

- **Free and open source**

- **JUnit was the first of a series of tools called *xUnit***
  – HttpUnit is one example
  – All work the same way, they just test different things
  – Once you learn JUnit, using the others is a snap

Writing JUnit Tests

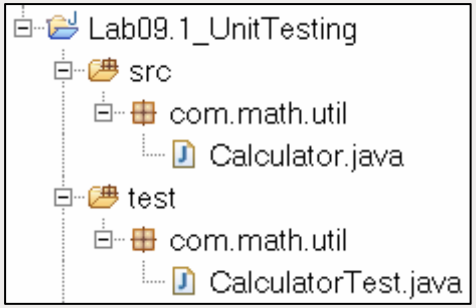
- Write a ***test class***
  - It should extend `junit.framework.TestCase`

- Add a ***fixture*** to set up your test environment (optional)
  - With `setUp()` and `tearDown()` methods

- Add a series of ***test methods***
  - Empty test methods will succeed by default
  - You need to exercise the class under test to give the test methods a reason to fail
  - If a test method reaches the closing curly brace without an assertion failure, the test succeeds

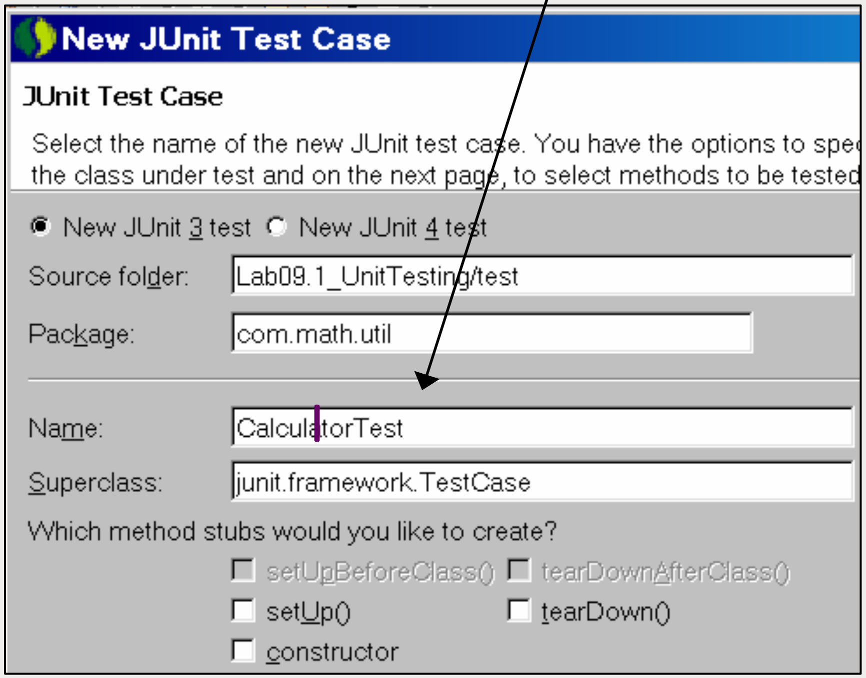
Naming Conventions

- `TargetClass` is tested by `TargetClassTest`
- `TargetClassTest` is in ***same package*** as `TargetClass`
- Use ***separate source folders*** for target and test code
  - ***src*** contains *TargetClass.java*
    test contains *TargetClassTest.java*
  - Allows you to test package-private methods (when necessary), yet still keep target and test code separate
  - You deploy/ship only the target code (in *src*)



Creating the Test Class

- **Create a new *JUnit Test Case***
  - Right-click on package and choose **New → JUnit Test Case**
  - Provide classname for the test class



Writing Test Methods

- Name your test methods `testSomething()`, and make sure that they:
  - Are **`public`**
  - Have a **`void` return type**
  - Accept **no arguments**
  - Start with **"test"** (lower case)

- The JUnit test runner will find all methods that meet the above requirements, and will run them all

- Order of test method execution ***cannot be predicted or relied upon***
  - Do not write test methods that rely on other test methods having been run first

Test Methods

- **In the body of your test method, invoke the target method**
  - Use varying arguments and verify the return values and/or side effects

- **You must give the test a way to fail**
  - Assert that your expected results are what actually occurred

- **The core principle of JUnit testing is:**
  compare expected results to actual results

JUnit Test Case - Example

```
import junit.framework.TestCase;

public class CalculatorTest
extends TestCase {

  // test method
  public void testAdd() {
    // assertEquals() is the workhorse of JUnit
    // arguments are (expected, actual)
    // execute target methods and assert that return values are good
    assertEquals(8, Calculator.add(5, 3));  // expect 8, got ?
    assertEquals(3, Calculator.add(1, 2));  // expect 3, got ?
  }
}
```

Running JUnit Tests

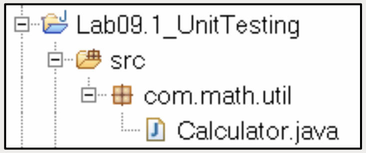
- **Run the test case using a test runner**
  - Modern IDEs come with a graphical test runner
    - Usually provide a progress bar
    - The bar is **green** if all tests have succeeded
    - The bar is **red** if any of the tests have failed
- ***"If the bar is green, the code is clean"***
- **Text-based test runners are also provided**
  - Often used for reporting test results, or as part of a nightly build process
- **Right-click on the test and choose Run As → JUnit Test**
  - Or just press [Alt+Shift+X] T

Ellucian

Lab 9.1
Unit Testing with JUnit

Lab 9.1 - Getting Started

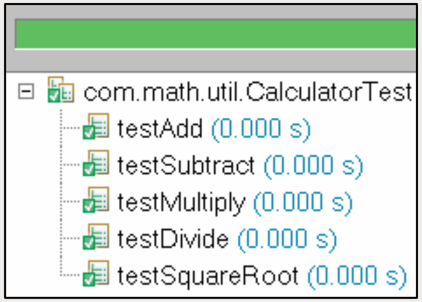
- **Make a copy of the *Lab04.5_StaticMembers* project and name the copy *Lab09.1_UnitTesting***
  - Then close the Lab04.5 project
  - In the new project, delete the default package, leaving you with only the **`com.math.util.Calculator`** class



- **Right-click on the project and choose New → Source Folder**
  - Name it ***test***
- **Right-click on the *test* folder and choose New → Package**
  - Name it **`com.math.util`** – same package as `Calculator`
- **Right-click on the package, choose New → JUnit Test Case**
  - Select **JUnit 3 test**, and call it **`CalculatorTest`** (see notes)
  - Click Finish and put JUnit 3 on the classpath (see notes)

Lab 9.1 - Testing Our Calculator

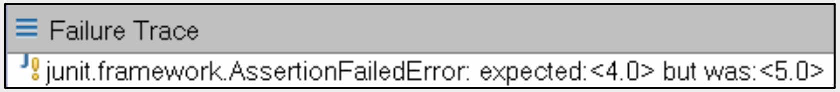
- **With the instructor guiding, you'll write test methods for the `Calculator` target class**
  - Should have at least one test method in `CalculatorTest` for each of the target methods in `Calculator`
    - **`add()`**
    - **`subtract()`**
    - **`multiply()`**
    - **`divide()`**
    - **`squareRoot()`**



- **Run the tests and note the results**

- **Force a failure**

```
public void testAdd() {
  assertEquals(4.0, Calculator.add(2, 3));
}
```



Ellucian

Unit Testing Overview
Test Run Lifecycle

Writing Unit Tests

Test Run Lifecycle

Test Fixture - `setUp()` Method

- A ***test fixture*** can be created with the **`setUp()`** method
  - **`public`**
  - **void return type**
  - Accepts **no arguments**

- Runs ***before*** each test method
  - Set up your object under test
  - As well as any collaborating objects

```
public class CalculatorTest extends TestCase {

  public void setUp() {
    // set up test environment
  }
}
```

Test Fixture - `tearDown()` Method

- **The `tearDown()` method runs *after* each test method**
  - Can be used to clean up what happened in a test
    - Close files, database connections, etc.
    - Roll back transactions

- **Can often be omitted**
  - If your setup and test methods simply create objects, the garbage collector will delete them – no need for `tearDown()`

```
public class CalculatorTest extends TestCase {
  // a file needed during the test
  private File file;

  public void tearDown() {
    file.close();
  }
}
```

Order of Invocation

- ***Each test*** **is bracketed with `setUp()` and `tearDown()`**

- **For example, if you have `setUp()`, `tearDown()`, `testOne()`, and `testTwo()`, either of the following is possible:**

1. setUp, **testOne**, tearDown	setUp, **testTwo**, tearDown
2. setUp, **testTwo**, tearDown	setUp, **testOne**, tearDown

- **Remember – the order of test method execution cannot be predicted**

Ellucian

Lab 9.2
Test Run Lifecycle

Lab 9.2 - Test Run Lifecycle

- **Continue to work in the current project**

- **Create a new unit test class in the `com.math.util` package**
  - Name it `LifecycleTest`

- **Write `setUp()`, `tearDown()`, `testOne()`, and `testTwo()` methods**
  - Each one should simply print a message to stdout, indicating which method is being executed

```
public void setUp() {
  System.out.println("setUp");
}
```

- **Run the tests and note the output**

Section Review

1. Explain the naming convention used for target class and test class.
2. What are the rules for test methods with respect to signature?
3. Why is the test client often in the same package as the target code?
4. Why do we use separate source folders for target and test code?
5. The JUnit test runner executes your tests alphabetically? [T/F]
6. The JUnit test runner executes your tests in the order in which they appear in the test class. [T/F]
7. What is a fixture? What methods comprise one? Why use them?

End of Section

- **This slide is intentionally devoid of any useful content**