Lecture 3: Objects and Classes

Wholeness of the Lesson

In the OO paradigm of programming, execution of a program involves objects interacting with objects. Each object has a type, which is embodied in a Java *class*. The intelligence underlying the functioning of any object in a Java program resides in its underlying class, which is the silent basis for the dynamic behavior of the objects. Likewise, pure consciousness is the silent level of intelligence that underlies all expressions of intelligence in the form of thoughts and actions in life.

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

The Object-Oriented Paradigm

- The OO paradigm, supported by many different OO languages, is an approach in which the elements of the problem domain are viewed as "objects" and then represented within the software design and code as "objects", so a minimum of translation from real world to machine (or some other conceptual framework) is required. As the programmer, you create software objects that correspond to real-world objects (like "Customer", "Record", "Balance", "Credit Card") and equip them with the behaviors that the real-world objects actually have ("withdrawAmount", "changeName", etc.)

OO concept: Class and Object

- Recall the reference example
- *Class* – this is the way a particular "type" is created in the Java language, such as Customer, Employee, CreditCard, Triangle

OO concept: Class and Object

- *Object construction and instances* – objects in Java are created as the program executes; objects are instances of a class; the class is like a template; the object is a realization of the template. Example: One instance of a Customer class may produce an object representing "Joe Smith"; another instance may represent "Susan Brown". We can use `new` to create objects.
  - When you create an object from a class like this -

```
Employee e = new Employee();
```

  you are invoking the *constructor* of that class.

Declaring and Creating Objects

- Constructor rules:
  - Constructors may accept parameters

```
public Employee(String fName, String lName) {
      firstName = fName;
      lastName = lName;
}
```

  - The *default* constructor of any class is the parameter-free constructor

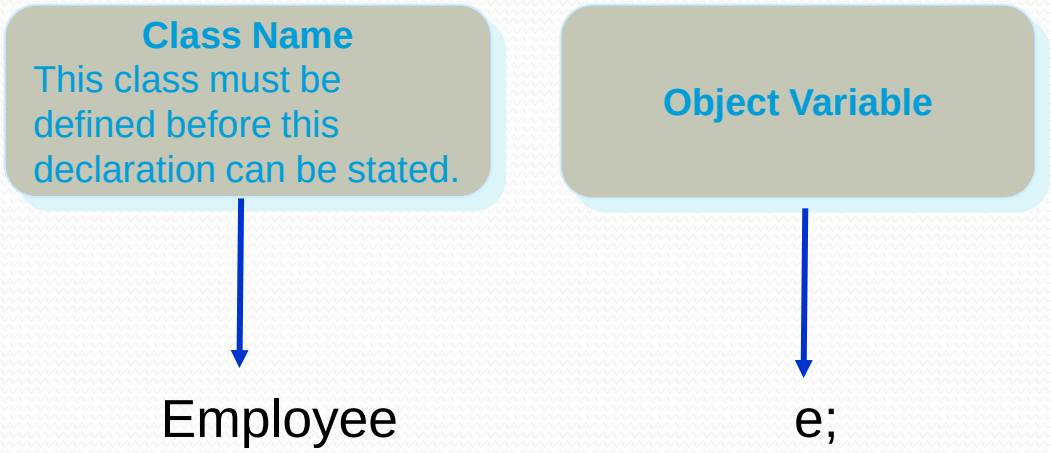
```
public Employee() {
}
```

  - The default constructor does not need to be explicitly coded, unless the class has other parametrized constructors (i.e. constructors that take one or more arguments) and the default constructor needs to be accessed.

Note: Simply declaring an object variable is not enough to create an object

```
Employee e;
e.getSalary();//throws NullPointerException
```

Object Declaration



More Examples

```
Account   account;
Student   jan, jim, jon;
```

Object Creation

Object Variable
Variable that refers to the object we are creating here.

Class Name
An instance of this class is created.

Argument
Arguments are used to construct an object.

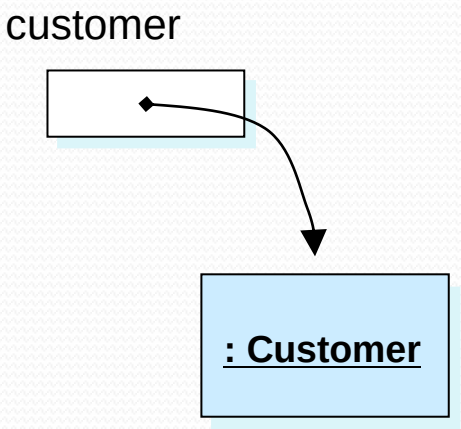
```
e =   new Employee("Joe", "Smith", 50000 );
```

More Examples

```
jon      = new Student("134267");
car1     = new Car( );
```

Declaration vs. Creation

```
1  Customer   customer;
2  customer   =  new  Customer( );
```

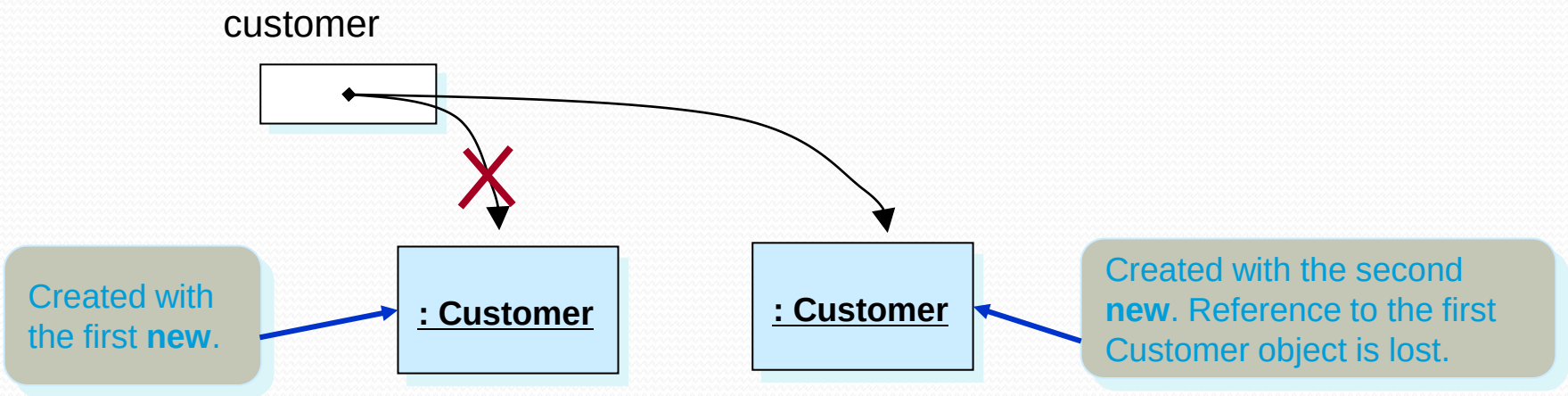


1. The object variable customer is declared and space is allocated in memory.

2. A Customer object is created and the identifier customer is set to refer to it.

Name vs. Objects

```
Customer   customer;
customer   =  new  Customer( );
customer   =  new  Customer( );
```



OO concept: Instance field/method

- *Instance fields* – the *fields* in a class are the types of data values that objects (instances of this class) are responsible for. Instance field for Employee class in the reference example:

```
private String firstName;
private String lastName;
private double salary;
private Address home;
private Address work;
```

- *Instance methods* – the *methods* in a class are the behaviors that instances of this class are capable of performing on the data. Examples of instance method in same Employee class:

```
public String toString() {
        return "[" + firstName + " " + lastName + "]";
}
public double getSalary() {
        return salary;
}

public double raiseSalary(int increase) {
        return salary += increase;
}
```

Accessing Data and Operations in an Object

Object Variable

Method Name
The name of the message we are sending.

Argument
The argument we are passing with the message.

```
e.setSalary(70000);
```

The data and methods of an object are accessed using "dot" notation, subject to visibility constraints.

More Examples

```
account.deposit( 200.0 );
car1.startEngine(  );
```

OO Concepts: Encapsulation

- *Encapsulation* – objects in a Java program interact with other objects, by way of their interfaces (list of services); the data that an object owns and the way that it manages that data are hidden from view; only the public services provided by the object are visible on the outside. The data and the way it is managed are said to be *encapsulated* in the object.

OO Concepts: State and Identity

- *State of an object* – the state of an object is the set of values currently stored in its fields. The `toString` method provides the state of an object. (More on `toString` method in Lesson 5.)

- *Identity* – Every object in Java has its own identity. Even if two objects have identical values in their fields, they can be distinguished as different objects.

Destroying Objects: Garbage Collection

- "Destructors" in languages like C++ . But there are no destructors in Java
- JVM provides a garbage collector.
  - When objects that have been created during execution of an application are no longer referenced anywhere in the application, they are considered to be garbage. Periodically, the JVM will invoke its garbage collector to determine which objects are still being referenced (“mark”) and then to free up the memory used up by all the remaining objects (“sweep”).
- *Assisting the garbage collector*. To ensure that an object variable no longer refers to a particular object (and to thereby set up the object for garbage collection) it suffices to set the value of the variable to null or to another object

Template for Creating a Class

- *General structure of a class file:*
  - one or more constructors
  - fields (which store data)
  - methods (which act on the data)

Main Point

The OO paradigm is a shift from old design and programming styles which are focused on machine-centric language models. In the OO paradigm, the focus shifts to mapping real world objects and dynamics to software objects and behavior; this parallel structure has proven to be more robust, less error prone, more scalable, and more cost-effective. In SCI we see that a more profound paradigm is discovered when the point of reference moves from the individual to the unbounded level – this is the CC paradigm in which self-sufficiency is based on true knowledge of the Self as universal, rather than the view of the self as a separate individual.

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Another Example

We discuss more elements of working with objects with reference to another more advanced example: see package `lesson3.employee`

```
public class Employee {
    // instance fields
    private String name;
    private String nickName;
    private double salary;
    private LocalDate hireDay;

    // constructor
    public Employee(String aName, String aNickName, double aSalary, int aYear,
            int aMonth, int aDay) {
        name = aName;
        nickName = aNickName;
        salary = aSalary;
        hireDay = LocalDate.of(aYear, aMonth, aDay);
    }

    public Employee() {
        this("DEFAULT","DEFAULT",0.0, 0,0,0);
    }
```

```
    // instance methods
    public String getName() {
        return name;
    }
    public String getNickName() {
        return nickName;
    }
    public void setNickName(String aNickName) {
        nickName = aNickName;
    }
    public double getSalary() {
        return salary;
    }

    public LocalDate getHireDay() {
        return hireDay;
    }
    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }
}
```

The 'this' Keyword

Each method in the `Employee` class passes in an *implicit parameter* which is the current instance of `Employee`. Called *implicit* because we don't actually display it as an argument. But it is accessible through the use of the keyword '`this`'.

- Example: You can rewrite the constructor of `Employee` like this:

```
public Employee(String aName, String aNickName, double
            aSalary, int aYear, int aMonth, int aDay) {
    this.name = aName;
    this.nickName = aNickName;
    this.salary = aSalary;
    this.hireDay = LocalDate.of(aYear, aMonth, aDay);
}
```

- Example: `e.raiseSalary(5)` actually entails two parameters, one explicit – the number 5 – and one implicit, which is the object reference `e`. (Again, "implicit" because the it does not appear in the method declaration `raiseSalary(double x).`)

The 'this' Keyword: Calling One Constructor from Another

```
public Employee(String aName, String aNickName, double aSalary, int
                aYear, int aMonth, int aDay) {
     name = aName;
     nickName = aNickName;
     salary = aSalary;
     hireDay = LocalDate.of(aYear, aMonth, aDay);
}

public Employee() {
     this("DEFAULT","DEFAULT",0.0, 0,0,0);
}
```

When calling one constructor from another, the line containing 'this' must be the first line in the constructor's body.

Best Practice: It's better to reuse constructor code than to rewrite identical sections of code in each version of the constructor.

Access Modifiers: Private, Public

Variables and methods may be assigned *access modifiers*: `private` and `public`

- `private` variables can be accessed only by methods within the class.

  Example: The following code inside the main method of `EmployeeTest` does not compile:

```
Employee e = new Employee("Carl", 75000, 1987, 12, 15);
String name = e.name;  //error: name field is not visible
```

- `public` variables can be accessed from any other class

- Likewise, methods in a class can be declared as `public` or `private`, with the same meaning

```
//From Advanced Employee example
public void raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
    salary += raise;
}

public class EmployeeTest {
    public static void main(String[] args) {
        Employee e = new Employee("Carl", "Jones", 75000, 1987, 12, 15);
        e.raiseSalary(5);
        System.out.println(e.toString());
    }
}
```

Accessors (getters) and Mutators (setters)

- Important examples of *public methods*.
- They play the role of *getting* values stored in instance variables, and *setting* values in instance variables, respectively.

Example:

```
//from Employee Example
public String getNickName() {
      return nickName;
}
public void setNickName(String aNickName) {
      nickName = aNickName;
}
```

continued

- Getters and setters support "encapsulation". Permits the class that owns the data to control access to the data.

Example: Notice "name" has been made "read-only" since there is no setter, whereas "salary" is modifiable.

```
private String firstName;
private String lastName;
public String getName() {
  return firstName + " " + lastName;
}
```

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Static Fields

- *Static fields* – uses the keyword `static` as part of the declaration
  - A value for a static variable is the same for all instances of the class
  - Can be accessed without an instance of enclosing class
  - Example: interestRate in CheckingAccount class. (Our assumption is interestRate is the same for all instances of this class.)
  - Demo: package lesson3.staticfield;

Application: Defining Constants

- When the `final` keyword is used for an instance variable, it means the variable may not be used to store a different value after it has been initialized explicitly. Using this keyword requires that the variable is initialized when declared or when the object is constructed.

  Example: Since the `name` field in `Employee` can never change, we could make it `final`

```
private final String name;
```

- *Static final* instance variables are considered to be *constants*. Example: `Label` class in `lesson3.javalabel`

```
public static final int LEFT      = 0;
public static final int CENTER    = 1;
public static final int RIGHT     = 2;
```

- Note: Variables that are declared *final* but not *static* are not technically considered *constants*. It is possible to have "blank final" whose value is not set till the constructor is called. See the package `lesson3.blankfinal`.

- Sometimes it is useful to store constants that may be useful for several classes in a separate place. One way to do this in the Label example is to create a class `LabelConstants` as follows:

```
class LabelConstants {

   public static final int LEFT      = 0;
   public static final int CENTER    = 1;
   public static final int RIGHT     = 2;
}
//then access the constants like this:
   LabelConstants.LEFT
```

- Problem with this approach:

```
Label lab = new Label("hello", 5); //set text to "hello",
                                   //and alignment to 5
```

There is no compiler-based control over the possible alignment values in `LabelConstants`.
Demo: `lesson3.javalabel`

Consider Using enums When Defining Constants

- A more reliable way to store constants is to use an *enumerated type* (also called an *enumeration type*). An enumerated type is a class all of whose possible instances are explicitly enumerated during initialization.

  Example:

```
public enum Size { SMALL, MEDIUM, LARGE};
//usage:  if(requestedSize==Size.SMALL) // do something
```

  Here, the `enum Size` has been declared to have precisely three instances, named `SMALL`, `MEDIUM`, and `LARGE`.

- *Application to the* `Label` *class*. The constants in Label could be put into an enum like this:

```
public enum LabelConstant{ LEFT, CENTER, RIGHT };
```

  Now compiler checks all inputs to `setAlignment`. See package `lesson3.labelwithenums` for details of the implementation.

One final note about `enum`s: The values of an enumerated type can be used like an `int` or `char` in a `switch` statement:

```
switch(alignment) {
   case LEFT:
        System.out.println(LEFT);  //prints out "LEFT"
        break;
   case CENTER:
        System.out.println(CENTER); //prints out "CENTER"
        break;
   case RIGHT:
        System.out.println(RIGHT); //prints out "RIGHT"
        break;
   default:
}
```

Static Methods

- Typically static methods are utility methods that provide a service of some kind, like a computation.
- Example of a static method: `Math.pow(x,y)`
- Typical form of static method:

```
public static <return_val_type> method(params)
```

  and typically the method does not save or read data in order to perform its function.

- How to make static method calls:

  By convention, always use <class_name>.<method_name>

  Example:

```
Math.pow(2,5);
```

- Can be accessed without an instance of enclosing class
- Cannot access instance variables
- Does not have an implicit parameter (so, cannot be used with "this")

Main Point

Static fields and methods are fields and methods whose lifetime persists throughout execution of the application, and when used with the public keyword, are globally accessible. The notion of "static" parallels the recognition that there is a field in life that is globally available and is always located in the same place in “memory”: namely, pure consciousness.

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Arguments and Parameters

```
class Sample {

    public static void
           main(String[] arg) {

      Account acct = new Account();
      . . .

      acct.add(400);
      . . .
    }

    . . .
 }
```

argument

```
class Account {

    . . .

    public void add(double amt) {

        balance = balance + amt;
    }

    . . .
}
```

parameter

- An argument is a value we pass to a method
- A parameter is a placeholder in the called method to hold the value of the passed argument.

Matching Arguments and Parameters

```
Demo demo = new Demo( );
int i = 5; int k = 14;
demo.compute(i, k, 20);
```

3 arguments

Passing Side

```
class Demo {
  public void compute(int i, int j, double x) {
      . . .
    }
}
```

3 parameters

Receiving Side

- The number of arguments and the parameters must be the same
- Arguments and parameters are paired left to right
- The matched pair must be assignment-compatible (e.g. you cannot pass a double argument to a String parameter)

Call-by-Value Parameter Passing

- When a method is called,
  - the value of the argument is passed to the matching parameter, and
  - separate memory space is allocated to store this value.
- This way of passing the value of arguments is called a *pass-by-value* or *call-by-value scheme*.
- Since separate memory space is allocated for each parameter during the execution of the method,
  - the parameter is local to the method, and therefore
  - changes made to the parameter will not affect the value of the corresponding argument.

Call-by-Value Example

```
class Tester {
    public void myMethod(int one, double two ) {
        one = 25;
        two = 35.4;
    }
}
```

```
Tester tester = new Tester();
int x, y;
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);
```

produces

```
10 20
```

Memory Allocation for Parameters

1

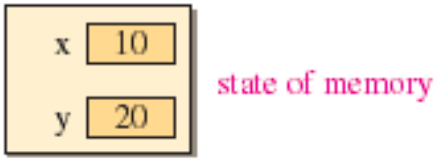
```
x = 10;
y = 20;          (1)
tester.myMethod( x, y );
```

execution flow

```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

at (1) before calling **myMethod**



state of memory

Local variables do not exist before the method execution.

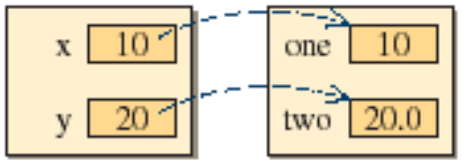
2

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

```
public void myMethod( int one, double two ) {   (2)

   one = 25;
   two = 35.4;
}
```

values are copied at (2)

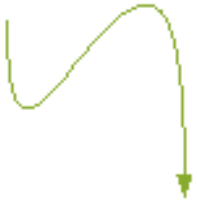


Memory space for myMethod is allocated, and the values of arguments are copied to the parameters.

Memory Allocation for Parameters (cont'd)

3

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

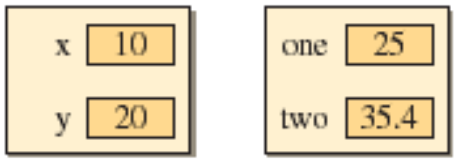


```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

3

at 3 before return



The values of parameters are changed.

4

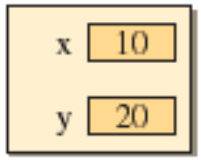
```
x = 10;
y = 20;
tester.myMethod( x, y );
```

4

```
public void myMethod( int one, double two ) {

   one = 25;
   two = 35.4;
}
```

at 4 after **myMethod**



Memory space for myMethod is deallocated, and parameters are erased. Arguments are unchanged.

Parameter Passing: Key Points

1. *Arguments are passed to a method by using the pass-by- value scheme.*
2. *Arguments are matched to the parameters from left to right.The data type of an argument must be assignment-compatible with the data type of the matching parameter.*
3. *The number of arguments in the method call must match the number of parameters in the method definition.*
4. *Parameters and arguments do not have to have the same name.*
5. *Local copies, which are distinct from arguments, are created even if the parameters and arguments share the same name.*
6. *Parameters are input to a method, and they are local to the method. Changes made to the parameters will not affect the value of corresponding arguments.*

Passing Objects to a Method

- As we can pass int and double values, we can also pass an object to a method.
- When we pass an object, we are actually passing the reference of the object
  - it means a duplicate of an object is NOT created in the called method

Passing a Student Object

```
Library l = new Library();
Student student = ……;
l.update(student);
```

Passing Side

1

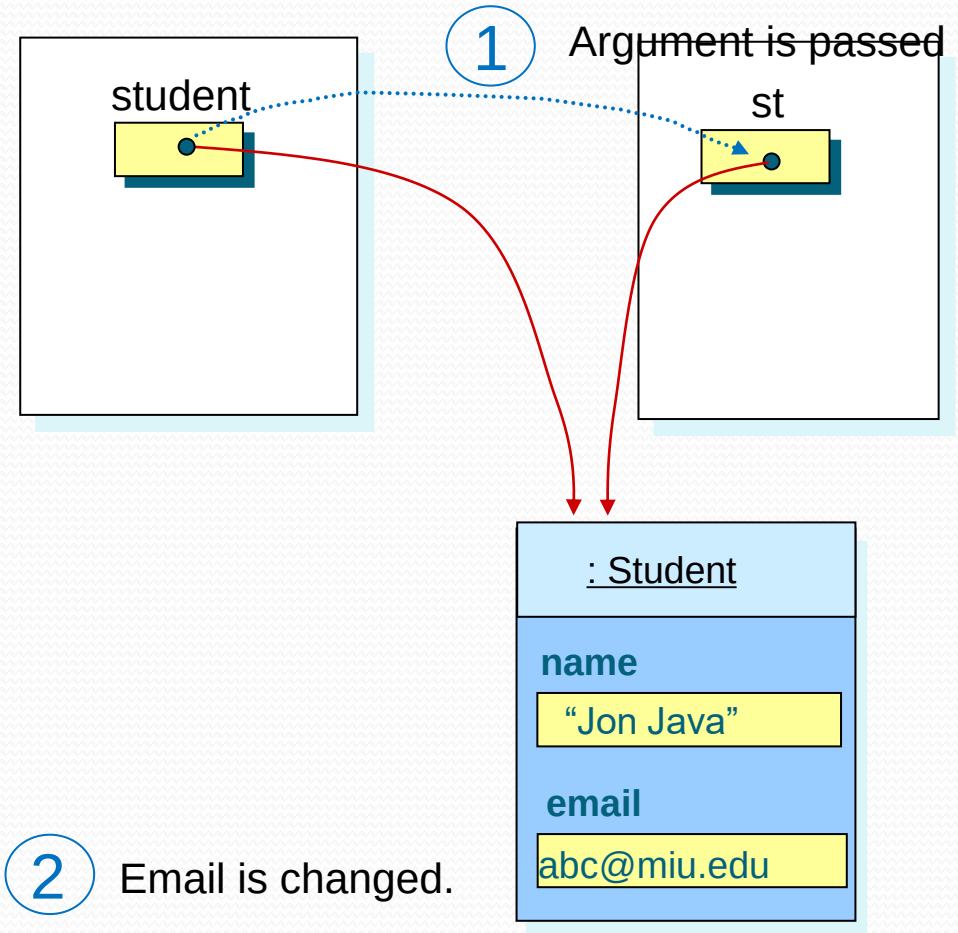
```
class Library {

  public void update(Student st) {

      st.setEmail("abc@miu.edu");
  }
}
```

2

Receiving Side



State of Memory

Call by Reference vs Call by Value

- A programming language supports a "call by value" idiom for method calls if a reference stored in a passed variable cannot be replaced with a different reference inside the method body.

- A programming language supports a "call by reference" idiom for method calls if a reference stored in a passed variable may be replaced with a different reference inside the method body.

- ***Java uses call by value***.

Main Point

Java method calls are in every case *call by value* (and never *call by reference*). Even though an object reference can be passed into a method, the variable that stores the reference cannot be made to point to a different reference within the method. Therefore, only a *copy* of such a variable is ever passed to a method (in other words, call by value). Call by value is reminiscent of the incorruptible quality of pure consciousness – "fire cannot burn it, nor water wet it".

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Miscellaneous: Overloading

Overloading a constructor

Example: Recall two of the constructors from Employee

```
public Employee(String fName, String lName) {
     firstName = fName;
     lastName = lName;
}
public Employee() {
}
```

This is called *overloading the constructor* – each version of constructor must have a different sequence of argument types from the others.

- Methods can be overloaded using the same rules. To say a method is overloaded means that there are two methods in a class having the same name but having different *signatures*.

  The signature of a method is the combination of the method's name along with the number and types of the parameters , and the order in which they occur.

- Also: Cannot have two methods with identical signatures but different return types:

```
//not allowed:
int myMethod(int input1, String input2)
String myMethod(int input1, String input2)
```

Miscellaneous: Field Initialization

- *Explicit field initialization.* When a class is constructed, all instance variables are initialized in the way you have specified, or if initialization statements have not been given, they are given default initialization.
  - primitive numeric type variables (including `char`) -- default value is 0;
  - `boolean` type variable – default value is `false`
  - object type variables (including `String` and array types) – default value is `null`
- It is not necessary to initialize *instance* variables in your code – you can use the default assignments. However, it is good practice to initialize always.
- By contrast, local variables (variables declared within a method body) *should always* be initialized (compiler issues a warning if not). Local variables are *never provided with default values;* if your code attempts to access an uninitialized local variable, a compiler error will be generated. (See package lesson3.uninitializedlocal.)

Miscellaneous: Packages in Java

- Packages represent units of organization of the classes in an application. The basic rules for packaging Java classes are like the rules in a file system: A package may contain Java classes and other packages (and other resources, like image files).
- Conventions concerning packages
  - The name of a package should consist of *all lower case letters*.

    Example: `myfavoritepackage  //correct`
    `myFavoritePackage  //incorrect`

  - To avoid naming conflicts between packages developed in different places (even possibly different parts of the world), a package "nesting" convention has developed in the Java community: Name your package by using your company's domain name in reverse as the prefix.

    Example: Your company's domain name is `magic.com`. Your top- level package is `myfavoritepackage`. So, for production, name this package `com.magic.myfavoritepackage`

- *Package-level access modifier.* We have seen `public` and `private` already. If no access modifier is specified for an instance variable or method, it is considered to have *package-level accessibility.* This means that the variable/method is visible only to other classes belonging to the same package.

Miscellaneous: Importing Classes

Can use fully qualified class names or imports or both

```
package mypackage;
import java.util.Math;
MyClass {
    public static void main(String[] args) {
            Math.sqrt(4);
    }
}
```

OR

```
package mypackage;

MyClass {
    public static void main(String[] args) {
            java.util.Math.sqrt(4);
    }
}
```

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
- Static Fields/Methods and Applications of them
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

Principles of Good Class Design

1. Keep data private and represent the services provided by a class with public methods.
2. Always initialize variables.
3. Divide big classes into smaller classes (if too many fields or too many responsibilities).
4. Not all fields need their own accessor and mutator methods; use this flexibility to control access to fields – e.g. can make a field read-only by providing a getter but no setter.
5. Use a consistent style for organizing class elements within each class.
6. Follow naming conventions for packages, classes, and methods.
7. ***Ambler's Book.*** Scott Ambler has systematized many best coding practices and an optimal coding style in his book *The Elements of Java Style.* (This has been included in your syllabus.)

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Object identity and identifying with unboundedness

1. A Java class specifies the type of data and the implementation of the methods that any of its instances will have.
2. Every object has not only state and behavior, but also identity, so that two objects of the same type and having the same state can be distinguished.

---

3. **Transcendental Consciousness:** TC is the identity of each individual, located at the source of thought.
4. **Wholeness moving within itself**: In Unity Consciousness, one's unbounded identity is recognized to be the final truth about every object. All objects are seen to have the same ultimate identity, even though differences on the surface still remain.