
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)
Professor Paul Corazza

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
- Three Examples of Java Classes: `Label, Employee, Customer`
  - Objects: Creating, Using, Destroying
    - Creating new objects
    - Accessing data and operations in an object
    - Destroying objects
- Some Classes in the Java Library: `Date, GregorianCalendar, LocalDate`
- Template for User-Defined Classes in Java
- Advanced Example: Employee
  - The '`this`' keyword
  - Access Modifiers: `private, public`
  - Mutators (setters) and Accessors (getters)
  - Best Practice: Never return mutable objects using a getter
- How to Make a Class Immutable
- Static Fields And Methods
  - Application: Constants in Java (and using enums in place of constants)
- How Objects and Variables Are Stored in Memory
- Call by Reference vs Call by Value
- Miscellaneous Topics
- Principles of Good Class Design

The Object-Oriented Paradigm

Historical background

- In the early days of programming, the task was to provide a solution by translating data from the real world, and procedures for manipulating it, into "computer language", which, in the early days was *assembly language*.

- "Higher level" languages eventually emerged – like FORTRAN, BASIC, and C – which made the translation process easier

- Eventually, languages emerged that made it possible to model the problem at hand directly, instead of asking the developer to model the machine. With this approach, one relies on "under the hood" implementations to take care of the mapping to the machine. Examples include LISP, Prolog, and many others.

- 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.)

- Think of every object as providing a set of services, which are specified in its *interface*. Analogy: Ignition system on a car – very specific interface (the keyhole) to provide a very specific service (starting the car).

  The automotive engineer figures out how to make the turning of the key produce the result of starting the car. Likewise, in the world of objects, once the services that an object should provide have been specified, you, as the developer, write the code to ensure that these services are available, and users of your object rely only on the interface to get your object to do things it is supposed to do.

OO Concepts

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

- *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"

- *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.

continued

- *Instance fields* – the *fields* in a class are the types of data values that objects (instances of this class) are responsible for; a Customer class might have a *name* field, a *streetAddress* field and a *telephoneNumber* field.

- *Instance methods* – the *methods* in a class are the behaviors that instances of this class are capable of performing on the data; a Customer class might provide methods *getName*, *updateStreetAddress,* and *lookupTelephoneNumber*

- *State of an object* – the state of an object is the set of values currently stored in its fields

continued

- *Inheritance* – OO languages, and in particular, Java, support the idea that one type is a "subtype" of another; the Triangle type is a subtype of the Shape type. If class A represents a subtype of class B, this relationship can be realized in the language; class A is said to *inherit* from class B; in code, we write

```
class A extends B
```

Fields and methods defined in class B are automatically available to instances of class A. (Much more on this later on.)

- *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.

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.

Three Examples

1. ***Reference Example*** from Lesson 2
2. `lesson3.objectdemo`
3. Java's `Label` class (`lesson3.javalabel.Label.java`)

In each example, we see:

- Instance variables to store data (in `Label: alignment` and `text`)
- `"get"` and `"set"` methods to read and write the data in these variables – these support *encapsulation* of data in instance variables
- A *constructor* to create an instance of the class
- The need for `import` statements
- Different Java classes are placed in different files – same name as the class
- These examples do not illustrate *inheritance* – see Lesson 7.
- *For later:* static variables and exception handling

Java's Label Class

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

    String text;
    int alignment = LEFT;
    public Label(String text, int alignment)  {
            this.text = text;
            setAlignment(alignment);
    }
    public int getAlignment() {
            return alignment;
    }
    public String getText() {
        return text;
    }
```

continued

```
    public synchronized void setAlignment(int alignment) {
            switch (alignment) {
                case LEFT:
                case CENTER:
                case RIGHT:
                    this.alignment = alignment;
                    return;
            }
            throw new IllegalArgumentException("improper alignment: " +
                                               alignment);
    }

    public void setText(String text) {
            synchronized (this) {
            if (text != this.text && (this.text == null ||
                      !this.text.equals(text))) {

                    this.text = text;
        }
      }
    }
}
```

Working with the Label Class

- Constructing a new `Label` instance uses the "new" operator and the "constructor"

  Example: `Label l = new Label("Hi There!",LEFT);`

- After the constructor is called, the state of the instance of `Label` is

  `[text = "Hi There!", alignment = LEFT]`

Declaring and Initializing Objects

- When you create an object from a class like this

```
MyClass cl = new MyClass();
```

you are invoking the *constructor* of that class

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

```
Label myLabel;
myLabel.getText();//throws NullPointerException
```

- Constructor rules:
  - Constructors may accept parameters – see `Employee` (Ref Example) and `Customer` (obect demo)
  - The *default* constructor of any class is the parameter-free constructor
  - 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)
  - Examples:
    - Reference Example: the Employee class explicitly lists its default constructor (must do so because a parametrized constructor is present)
    - Reference Example and `lesson3.objectdemo: Main` class's default construtor is not shown explicitly
  - Constructors cannot be called like ordinary methods, but only with `new`

- *Initializing* an object variable is done in one of two ways: with `new` or by assigning to an already existing object variable

```
Label myLabel = new Label();
Label anotherLabel = myLabel;
```

- The `new` operator does two things: it *creates* an object and it *returns* a reference to that object. **Important**: object variables do not contain objects, only *references* to objects.

- When an object variable is initialized by assigning it to another object variable (one that refers to an existing object), then both variables refer to the same object

```
Label myLabel = new Label();
Label anotherLabel = myLabel;
myLabel.setText("hello");
String text = anotherLabel.getText(); //text is "hello"
```

Accessing Data and Operations in an Object

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

```
//Label class
public static void main(String[] args)  {
    Label label = new Label("Hi there!", LEFT);
    label.setAlignment(RIGHT);
    //Can access instance variables that
    //are visible with "dot" notation
    System.out.println(label.alignment);
    //Better to access data using getters
    System.out.println(label.getAlignment());
}
```

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").
- *When does the garbage collector run*? Garbage collection is initiated at the JVM's discretion, specifically to free up memory.
- *Sometimes garbage collector is not enough*. Sometimes an unreferenced object ties up more than just a memory location; it may also tie up other resources, like a file or a database connection. In such cases, the garbage collector itself does not know how to – and therefore will not -- free up these resources. To free them up, the developer must write code to handle this need and make sure that it will execute before all references to this object are lost. (Code for this will be discussed in Lesson 12.)
- *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.

Dates and Calendars in Java 7 and Java 8

- **`Date`** represents a point in time. It is the number of milliseconds since the beginning of the day 1/1/1970 ("the epoch").

- **`GregorianCalendar`** is responsible for calendar operations – calendars are the way one culture represents points in time (examples: Gregorian Calendar, lunar calendar, Mayan calendar) in terms of days, weeks, months, etc.

- The `Date` class has a small API:

  Two constructors:

```
Date()  //today's date, in number of milliseconds
        //from the epoch
Date(long numMilliseconds) //another date, passed in
                           //in the form of millisecs
```

  Methods:

```
boolean after(Date d)
boolean before(Date d)
Object clone()
int compareTo(Date d)
boolean equals(Date d)
long getTime()
void setTime(long millisecs)
```

- Use `GregorianCalendar` to change values like day, month, year, hour, locale

  Examples:

```
new GregorianCalendar()  // today at this moment
new GregorianCalendar(1999, 11, 31) //11 is December
new GregorianCalendar(1999, 11, 31, 23, 12, 58)
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
GregorianCalendar cal = new GregorianCalendar();
int month = cal.get(Calendar.MONTH);
int weekday = cal.get(Calendar.DAY_OF_WEEK);
int daynum = cal.get(Calendar.DATE);
cal.set(Calendar.YEAR, 2001);
cal.set(Calendar.DATE, 23);
```

- *Conversions:*

```
//get the Date (point in time) represented by cal instance
GregorianCalendar cal = new GregorianCalendar();
Date d = cal.getTime();

//get the GregorianCalendar that corresponds to this Date
Date d = new Date();
GregorianCalendar cal = new GregorianCalendar();
cal.setTime(d);
```

- *Formatting.* To format a `Date`, send it to a formatter

```
//j2se5.0: can use String.format with printf options

String format = "%tD";
Date d = new Date();
String formattedDate = String.format(format, d);
// has form MM/dd/yy


//pre - j2se5.0: use DateFormat and/or SimpleDateFormat

Date d = new Date();
DateFormat f =
DateFormat.getDateInstance(DateFormat.SHORT);
String formattedDate = f.format(d);
// has form: MM/dd/yy
```

See demo package: `lesson3.dates`

Java 8 Date and Time API

- New in Java 8, replaces `Date` and `GregorianCalendar`
- `LocalDate` manages dates that do not require timezone data, like birthdays and a single-timezone company intranet

  `LocalDateTime` is like `LocalDate`, but includes time information.

  `ZonedDate (ZonedDateTime)` handles dates (date and time) and takes into account time zones. They directly replace usages of `GregorianCalendar`
- The new Data and Time classes are *immutable;* operations that act on instances produce new instances – this is like Java's String class
- In this course, sometimes we use `Date` and `GregorianCalendar`, sometimes `LocalDate`, to handle date needs. In Java 8's new Date and Time API, `LocalDate` is the easiest to learn; the other classes in that API are more complex variants of this one.

LocalDate Sample Code

```
System.out.println("Today's date: " + LocalDate.now());
System.out.println("Specified date: " + LocalDate.of(2000, 1, 1));


//Formatting LocalDates as strings and reading date strings as LocalDates
public static final String DATE_PATTERN = "MM/dd/yyyy";
public static LocalDate localDateForString(String date)      {
          return LocalDate.parse(date, DateTimeFormatter.ofPattern(DATE_PATTERN));
}

public static String localDateAsString(LocalDate date) {
          return date.format(DateTimeFormatter.ofPattern(DATE_PATTERN));
}

//// LocalDate <--> GregorianCalendar conversions
public static LocalDate LocalDateFromGregCalendar(GregorianCalendar cal) {
          return LocalDate.of(cal.get(Calendar.YEAR), 1 + cal.get(Calendar.MONTH),
                              cal.get(Calendar.DATE));
}

public static GregorianCalendar GregorianCalendarFromLocalDate(LocalDate locDate) {

          return new GregorianCalendar(locDate.getYear(),
                    locDate.getMonth().getValue()-1,
                      locDate.getDayOfMonth());
}
```

See demo package: lesson3.dates

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)
  - We have seen three examples: Reference Example, `Customer`, `Label`

Advanced Example

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

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

    // constructor
    Employee(String aName, String aNickName, double aSalary, int aYear,
            int aMonth, int aDay) {
        name = aName;
        nickName = aNickName;
        salary = aSalary;
        GregorianCalendar cal = new GregorianCalendar(aYear, aMonth - 1, aDay);
        hireDay = cal.getTime();
    }
```

```java
// instance methods
public String getName() {
    return name;
}
public String getNickName() {
    return nickName;
}
public void setNickName(String aNickName) {
    nickName = aNickName;
}
public double getSalary() {
    return salary;
}
// needs to be improved
public Date getHireDay() {
    return hireDay;
}
public void raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
    salary += raise;
}
private String format = "name = %s, salary = %.2f, hireDay = %s";

public String toString() {
    return String.format(format, name, salary, Util.dateAsString(hireDay));
}
```

```
public class EmployeeTest {
    public static void main(String[] args) {
        Employee[] staff = new Employee[3];
        staff[0] = new Employee("Carl", "Jones", 75000, 1987, 12, 15);
        staff[1] = new Employee("Harry", "Rogers", 50000, 1989, 10, 1);
        staff[2] = new Employee("Tony", "Atkinson", 40000, 1990, 3, 15);

        // From EmployeeTest -- can access raiseSalary method
        for (Employee e : staff) {
            e.raiseSalary(5);
        }
        for (Employee e : staff) {
            System.out.printf(e.toString() + "%n");
        }
    }
}
```

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:

```
Employee(String name,
         double salary,
         int year,
         int month,
         int day) {
   this.name = name;
   this.salary = salary;
   GregorianCalendar cal = new GregorianCalendar(. . .);
   this.hireDay = cal.getTime();
}
```

- 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).`)
- See Demo in package: `lesson3.thiskeyword`

Access Modifiers: Private, Public

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

- `private` instance 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` instance variables can be accessed from any other class

  Example: The names of the months are public fields in the `Calendar` class:

```
int monthNum = Calendar.DECEMBER;  //this is legal
```

- 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;
}
```

```
//From EmployeeTest -- can access raiseSalary method
for(Employee e : staff) {
    e.raiseSalary(5);
}
```

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 Advanced 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.

  Sample benefit: Ability to change the implementation without undermining client code:

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

Security: Careless Use of Getters

Example from `Employee` (in Advanced Employee example)

```
// Getter in Advanced Employee example
public Date getHireDay() {
    return hireDay;
}
```

```
//Rogue programmer could do this:
    Employee harry = . . . //get instance
    Date d =  harry.getHireDay();
    long tenYearsInMilliseconds =
        10 * 365 * 24 * 60 * 60 * 1000L;
    long time = d.getTime();
    d.setTime(time - tenYearsInMilliseconds);
```

(See package `lesson3.employee.rogue`)

Question: How can this be prevented?

A Solution: Use `clone()` to correct `getHireDay()`:

```
//corrected code
public Date getHireDay() {
    return (Date)hiredDay.clone();
}
```

Moral: Do not return *mutable data fields* directly, via getter methods. Instead, return a copy of such fields.

(See package: `lesson3.employee.j7rogue_soln`)

Another Solution: Immutable Classes

- If `GregorianCalendar`s were *immutable*, it would be impossible to modify an instance of a `GregorianCalendar` by changing the `Date` in the way the rogue programmer did.
- **Java 8 Solution**. Use `LocalDate` in place of `GregorianCalendar` and `Date`: The `hireDay` should now have type `LocalDate`, and the year, month, day passed into the constructor should be used to construct this `LocalDate`. Then `getHireDay()` will return an immutable `LocalDate`.

  (See package: `lesson3.employee.j8rogue_soln`)

How to Make a Class Immutable

- A class is immutable if the data it stores cannot be modified once it is initialized.
- Java's `String` and number classes (such as `Integer`, `Double`, `BigInteger`), as well as `LocalDate`, are immutable. Immutable classes provide good building blocks for creating more complex objects.
- Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a "mutable companion" should also be created (for example, the mutable companion for `String` is `StringBuilder`) to handle the multiplicity without hindering performance.
- Guidelines for creating an immutable class (from Bloch, *Effective Java*, 2nd ed.)
  - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
  - **Provide *getters* but no *setters* for all fields**. Not providing setters is essential for making the class immutable.
  - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class in another way – to be discussed in Lesson 6.)
  - **Make sure that getters do not return mutable objects**.
- See demo package lesson3.immutable. Also, see Lab 3, Problem 4.

Static Fields And Methods

- *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
  - Example: Pretest question (Fall 2005):

> "Create a Java class that keeps track of how many instances of itself have been created. This data should be stored in a variable and should be accessible by a public accessor method (this should be an instance method). Write a main method that constructs several instances of the class and then outputs the number of instances created by calling this accessor method."

continued

Solution:

```
public class CountInstances {
    private static int count;
    CountInstances() {
        ++count;
    }

    public int getCount(){
        return count;
    }

    public static void main(String[] args){
        for(int i = 0; i < 10; ++i){
            new CountInstances();
        }
        System.out.println("Num instances created: "
          + CountInstances.count);
    }
}
```

About Static Methods

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

```
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);
```

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. 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*. Recall the `Label` class:

```
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`.

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) applyDiscount();
```

  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:
}
```

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.

How Objects and Variables Are Stored in Memory

- **Registers** – fastest area of memory (within the processor) but no access with Java programs

- **The Stack** – second fastest area of RAM because of direct support from processor for the stack pointer. This is literally a stack data structure that the JVM maintains. It stores local variables and method names; if as variables are initialized inside a method, they are also added to the stack. When method exits, all of these are popped. Object references are also stored on the stack, but the objects themselves are not.

- **The Heap** – general purpose pool of memory (in RAM still) where Java objects are placed. Much more flexible than the Stack – no bookkeeping required to allocate or de-allocate memory blocks in the heap. But the price for this flexibility is that it is a bit slower.

  The String Pool. The String class maintains in heap memory a table of Strings that have been "interned". When a string literal is defined, the table is checked; if the string already exists, it is returned, otherwise it is added to the table. Two interned strings that have the same values will always be considered equal using == since they are literally the same object. See package lesson3.stringinterning.

- **Static Storage** – another area of RAM that holds object references that have been declared "static" in the Java program. These references remain "alive" through the entire execution of the program. (Public static variables are therefore the same as global variables.)

Call by Reference vs Call by Value

- A programming language supports a "call by reference" idiom for method calls if the values stored in the variables that are passed into the method may be modified in the method body.

  A programming language supports a "call by value" idiom for method calls if the values stored in the variables that are passed into the method represent *copies* of the original values, so that they may *not* be modified in the method body. ***Java uses call by value***.

- Call By Value for Primitives

```
public static void main(String[] args) {
    CallByValuePrimitives c = new CallByValuePrimitives();
    int num = 50;
    c.triple(num);
    //value of num is still 50
    System.out.println(num);

}
public void triple(int x) {
    x = 3 * x;
}
```

- Change Value Stored in an Object Reference

```
public static void main(String[] arg) {
    ChangeValueInReference c = new ChangeValueInReference();
    Employee harry = new Employee("Harry","Rogers", 50000, 1989, 10, 1);
    c.tripleSalary(harry);
    //salary has been tripled
    System.out.println("Harry's salary now: " + harry.getSalary());
}

public void tripleSalary(Employee e) {
    e.raiseSalary(200);
}
```

- Call by Value for Objects

```
public class CallByValueObjects {
    public static void main(String[] args) {
        CallByValueObjects c = new CallByValueObjects();
        Employee a = new Employee("Alice","Thompson", 60000, 1995, 2, 10);
        Employee b = new Employee("Bob","Rogers", 70000, 1997, 10, 1);
        c.swap(a, b);
        //To which Employee does the reference a point?
    }
    public void swap(Employee x, Employee y) {
        Employee temp = x;
        x = y;
        y = temp;
    }

}
```

Moral: all method calls in Java are **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".

Miscellaneous: Overloading

Overloading a constructor

Example: Recall two of the constructors from GregorianCalendar

```
GregorianCalendar()
GregorianCalendar(int year, int month, int day)
```

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

For example, the following additional constructor would not be allowed by the compiler

```
GregorianCalendar(int hour, int minute, int second)
```

- 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: Calling One Constructor from Another

Example:

```
Label(String text, int alignment) {
      this.text = text;
      this.alignment = alignment;
 }
```

Java's `Label` class has another constructor `Label(String text)`, that uses LEFT as the default alignment. Does this by letting `Label(String text)` *call* the other constructor.

To call one constructor from another, use the '`this`' keyword:

```
Label(String text) {
      this(text, LEFT);
}
```

If you do this, 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 fo the constructor.

Miscellaneous: Initialization Blocks

- Initialization blocks

Example of *object initialization block*

```
private static int nextId;
private int id;
private String name;

//object initialization block
{
   id = nextId;
   nextId++;
}
```

Usually, object initialization could be (and should be) done in the constructor instead of in a separate block.

Example of *static initialization block*

```
static String[] arr;
static final int CA = 0;
static final int NY = 1;
static final int IA = 2;
static int [] indexes = {CA,NY,IA};

//static initialization block
static {
        arr = new String[indexes.length];
        arr[CA] = "California";
        arr[NY] = "New York";
        arr[IA] = "Iowa";
}
```

Can be useful when the initialization needs to happen before the body of the constructor executes.

Miscellaneous: Sequence of Execution

Sequence of execution when a constructor is called

a) If this is the first time the class is loaded into memory, all static fields are given default initialization, and then all static fields are initialized and static initialization blocks are run in the order in which they appear in the class file.
(Static fields are initialized only once; static blocks executed only once.)

b) All instance variables are initialized with their default values

c) All instance variable initialization is performed and object initialization blocks executed, in the order in which they occur in the class file

d) If the first line of the constructor calls another constructor, the body of the second constructor is executed

e) The body of the constructor is executed

Note: See package lesson3.orderofexec.demo. The sequence of steps becomes more complex when *inheritance* is involved – see Lesson 7.

Miscellaneous: When Is 'this' Initialized?

- *When is* `this` *initialized*? It is initialized immediately after all static initialization has occurred (and before any instance variables are initialized) – see demo in package `lesson3.thiskeyword`

Miscellaneous: Making a Constructor Private

Occasionally, constructors are declared to be private

a) A private constructor cannot be accessed by any other class. The only way for another class to communicate with such a class is by way of *static methods*.
b) Useful for utility classes which provide static methods only – example: Math (this class has a private constructor).
c) Private constructor can be used as part of a strategy to ensure that *only one instance* of a class is ever used. Such a class is called a *singleton*. Making a class a singleton during design is called *using the Singleton design pattern*.

See the example: lesson6_singletons.SingleThreadedSingleton.java, which will be discussed in Lesson 6.

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).
- It is not *necessary* to use packages in a Java application, and for small applications, there is often no need to use them. In that case, the JVM creates a *default package* and views your classes as belonging to this package.
- For medium to large applications, it is important to organize code in packages for several reasons:
  - Packaging according to a systematic scheme (so that classes that are related to each other belong to the same package and unrelated classes belong to different packages) supports parallel development of code and makes code easier to maintain.
  - Packages can be defined so that they can be reused by other applications (example: a Statistics package)
  - A package creates a *namespace* that helps to prevent naming conflicts. For example, it is possible to have two classes with the same name as long as they belong to different packages.

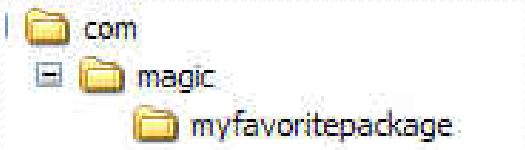
- Conventions concerning packages
  - The name of a package should consist of *all lower case letters.*

    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.

  Example: All the instance variables in Sun's `Label` class have package level accessibility.

  Note: It is generally better for the sake of encapsulation to label every instance variable private, though this approach is not always taken.

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);
    }
}
```

Miscellaneous: Static Imports

- Static imports (new with j2se5.0) allows you to import static fields and methods.

  Example:

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

  Caution: Use of static imports is often considered a bad practice because of readability – it is too hard to determine the origin of a method that has been statically imported.

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.