Java Programming

Java Classes in depth

Review of Lecture 3

- **Declaring static methods**: public static double maximum(double x, double y, double z)
- Calling a static method outside the class:

ClassName.methodName(arguments)

Ex: **Math.abs**(2.3)

Calling a static method inside the class:

double result = maximum(number1, number2, number3);

- **main** method is static – allows JRE to call main without creating an instance of the class.
- public methods are available to client code.
- **Return type** - specifies the type of data a method returns.
- Methods that do not return any information are **void**
- **Method header**
- **Method body**

Review of Lecture 3

- **Instance** methods: non static
- **Static** methods need a an object of the class to access instance variables and methods.
- the statement:

  return *expression;*

  evaluates the *expression*, then **returns the result to the caller**
- declare constants as public final static
- Arguments can be **promoted** to higher types (ex: float to double).
- Successive return addresses are pushed by JRE onto the **stack** in LIFO order.
- Predefined Java classes are grouped into categories of related classes called ***packages***
- Enhanced for statement
- Passing arrays as arguments to methods

Lesson 4 Objectives

- Create and use Java classes as a means of developing component-based applications.
- Use 'this' keyword.
- Use **composition** in Java applications.
- Define and use enum type in Java classes.
- Use **package access**.
- Use **final instance variables**.
- Design Java applications composed of **multiple classes**.

Java Classes in depth

- Deeper look at building classes, **controlling access to members** of a class and **creating constructors**.
- Show how to throw an **exception** to indicate that a problem has occurred.
- **Composition** - a capability that allows a class to have references to objects of other classes as members.
- More details on enum types.
- Discuss **static class members** and **final instance variables** in detail.
- Show how to organize classes in **packages** to help manage large applications and **promote reuse**.

Time Class Case Study

- Class Time1 represents the time of day.
  - private int **instance variables** *hour*, *minute* and *second* represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23, and minutes and seconds are each in the range 0–59).
  - public methods *setTime*, *toUniversalString* and *toString*.
  - Called the **public services** or the **public interface** that the class provides to its clients.

Time Class Case Study

```
public class Time1
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59
   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void setTime(int hour, int minute, int second)
   {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60)
      {
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }
```

Time Class Case Study

```
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  // convert to String in universal-time format (HH:MM:SS)
  public String toUniversalString()
  {
    return String.format("%02d:%02d:%02d", hour, minute, second);
  }
  // convert to String in standard-time format (H:MM:SS AM or PM)
  public String toString()
  {
    return String.format("%d:%02d:%02d %s", ((hour == 0 || hour == 12) ? 12 : hour % 12),
      minute, second, (hour < 12 ? "AM" : "PM"));
  }
} // end class Time1
```

Time Class Case Study

- Class Time1 **does not declare a constructor**, so the **compiler supplies a default constructor**.
- Each instance variable implicitly receives the **default int value**.
- Instance variables also can be initialized when they are declared in the class body, using the same initialization syntax as with a local variable.

TimeTest class

```
public class Time1Test
{
   public static void main(String[] args)
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor
      // output string representations of the time
      displayTime("After time object is created", time);
      System.out.println();
      // change time and output updated time
      time.setTime(13, 27, 6);
      displayTime("After calling setTime", time);
      System.out.println();
```

TimeTest class

```
   // attempt to set time with invalid values
   try
   {
      time.setTime(99, 99, 99); // all values out of range
   }
   catch (IllegalArgumentException e)
   {
      System.out.printf("Exception: %s%n%n", e.getMessage());
   }
   // display time after attempt to set invalid values
   displayTime("After calling setTime with invalid values", time);
}
```

TimeTest class

```
// displays a Time1 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time1 t)
   {
      System.out.printf("%s%nUniversal time: %s%nStandard 
time: %s%n",
         header, t.toUniversalString(), t.toString());
   }
} // end class Time1Test
```

Using Exceptions to validate arguments

- Method **setTime** (lines 12 - 25) declares three int parameters and uses them to set the time.
- Lines 15 -16 test each argument to **determine whether the value is outside the proper range**.
- For incorrect values, setTime **throws an exception of type IllegalArgumentException** (lines 18–19)
- Notifies the client code that **an invalid argument was passed to the method**.

Java Exception Handling model

```
try
{
// code that may generate an exception
}
catch (Exception ex)
{
//code to handle the exception
}
```

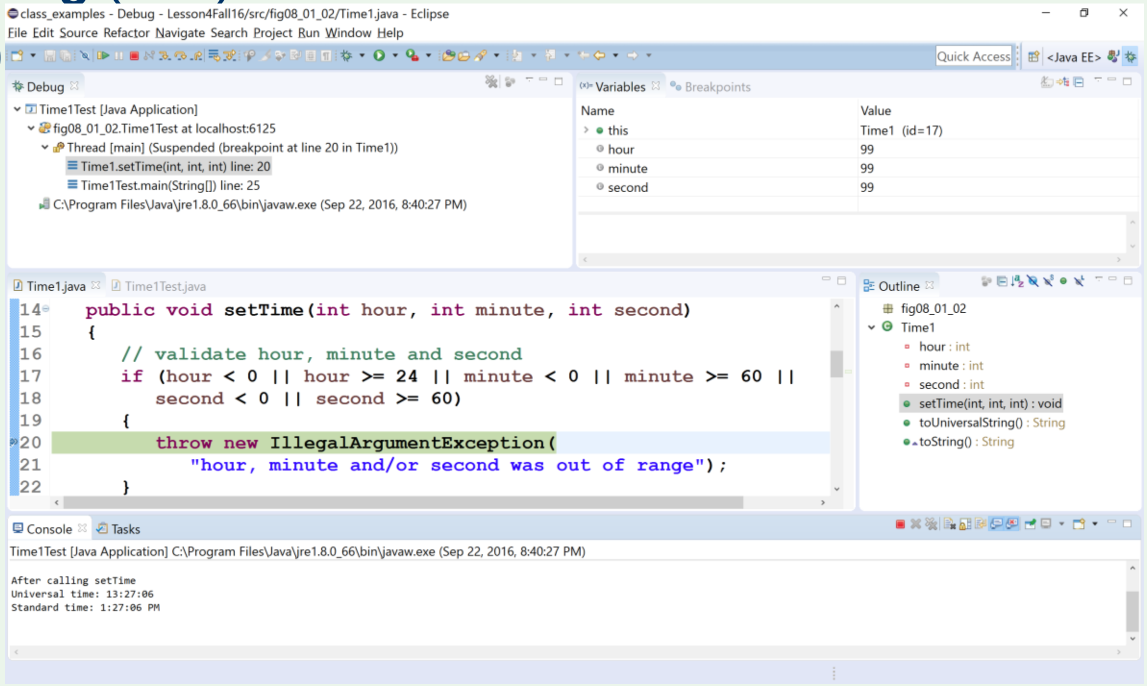
- throw statement - throws an exception which is returned to the calling method

Using Exceptions to validate arguments

- Use try...catch to catch exceptions and attempt to recover from them.
- The class instance creation expression in the **throw** statement (line 18) creates a new object of type **IllegalArgumentException**.
- In this case, we call the constructor of **IllegalArgumentException** that allows us to specify a **custom error message**.
- After the exception object is created, the throw statement immediately **terminates method setTime** and **the exception is returned to the calling method that attempted to set the time**.

Using Debug window

- Switch to **Debug perspective**
- Set a **break point** to line 20 (Time1.java)
- Select Run/**Debug** (F11)
- **Step over** (F6)



class_examples - Debug - Lesson4Fall16/src/fig08_01_02/Time1.java - Eclipse

Debug

```
Time1Test [Java Application]
  fig08_01_02.Time1Test at localhost:6125
    Thread [main] (Suspended (breakpoint at line 20 in Time1))
      Time1.setTime(int, int, int) line: 20
      Time1Test.main(String[]) line: 25
  C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (Sep 22, 2016, 8:40:27 PM)
```

Variables

Name	Value
this	Time1 (id=17)
hour	99
minute	99
second	99

Time1.java

```
14    public void setTime(int hour, int minute, int second)
15    {
16       // validate hour, minute and second
17       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
18          second < 0 || second >= 60)
19       {
20          throw new IllegalArgumentException(
21             "hour, minute and/or second was out of range");
22       }
```

Outline

```
fig08_01_02
Time1
  hour : int
  minute : int
  second : int
  setTime(int, int, int) : void
  toUniversalString() : String
  toString() : String
```

Console

Time1Test [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (Sep 22, 2016, 8:40:27 PM)

```
After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Software Engineering of the Time1 Class Declaration

- The **instance variables** hour, minute and second are each declared private.
- The actual data representation used within the class is of no concern to the class's clients.
- Reasonable for Time1 to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight.
- Clients could use the same **public methods** and get the same results without being aware of this.

Java SE 8 - Date/Time API

- Java SE 8 introduces a new Date/Time API—defined by the classes in the package java.time - applications built with Java SE 8 should use the Date/Time API's capabilities, rather than those in earlier Java versions.
  - provides more robust, easier-to-use capabilities for manipulating dates, times, time zones, calendars and more.

LocalDateTime timePoint = LocalDateTime.now(); // The current date and time

LocalDate.of(2012, Month.DECEMBER, 12); // from values

LocalDate.ofEpochDay(150); // middle of 1970

LocalTime.of(17, 18); // the train I took home today

LocalTime.parse("10:15:30"); // From a String

Controlling Access to Members

- Access modifiers public, private and protected control access to a class’s variables and methods.
- **public methods** present to the class’s clients a view of the services the class provides (the class’s public interface).
- Clients need not be concerned with how the class accomplishes its tasks.
- For this reason, the **class’s private variables and private methods** (i.e., its implementation details) are **not accessible to its clients**.
- **private class members** are not accessible outside the class (MemberAccessTest example).

Referring to the Current Object's Members with the this Reference

- Every object can access a reference to itself with keyword this.
- When a an instance method is called for a particular object, the method's body implicitly uses keyword this to **refer to the object's instance variables and other methods**.
- Enables the class's code to know which object should be manipulated.
- Can also use keyword this explicitly in an instance method's body.
- Can use the this reference **implicitly and explicitly**.

Referring to the Current Object’s Members with the this Reference

- ThisTest example:
- SimpleTime declares three private instance variables—hour, minute and second.
- We use the this reference to refer to the instance variables if parameter names of the constructor are identical to the class’s instance-variable names.
- .

Time Class Case Study: Overloaded Constructors

- **Overloaded constructors** enable objects of a class to be initialized in different ways.
- To overload constructors, simply **provide multiple constructor declarations** with different signatures.
- Recall that the compiler differentiates signatures by the **number of parameters**, the **types** of the parameters and the **order** of the parameter types in each signature.

Referring to the Current Object's Members with the this Reference (Cont.)

- When you compile a .java file containing more than one class, the compiler produces a separate class file with the **.class** extension for every compiled class.
- When one source-code **(.java**) file contains multiple class declarations, the compiler places both class files for those classes in the same directory.
- **A source-code file can contain only one public class -** otherwise, a compilation error occurs.
- **Non-public classes** can be used only by other classes in the same package.

Time Class Case Study: Overloaded Constructors (Cont.)

- Class Time2 (Fig. 8.5) contains **five overloaded constructors** that provide convenient ways to initialize objects.
- The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration.

Time Class Case Study: Overloaded Constructors (Cont.)

- A program can declare a so-called **no-argument constructor** that is invoked without arguments.
- Such a constructor simply initializes the object as specified in the constructor’s body.
- Using this in method-call syntax as the first statement in a constructor’s body **invokes another constructor of the same class**.
  - Popular way to reuse initialization code provided by another of the class’s constructors rather than defining similar code in the no-argument constructor’s body.
- Once you declare any constructors in a class, the compiler will not provide a default constructor.

Time Class Case Study: Overloaded Constructors (Cont.)

- Methods can access a class's private data directly without calling the get methods.
- However, consider changing the representation of the time from three int values (requiring 12 bytes of memory) to a single int value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory).
- If we made such a change, only the bodies of the methods that access the private data directly would need to change - in particular, the three-argument constructor, the setTime method and the individual set and get methods for the hour, minute and second.
- There would be no need to modify the bodies of methods toUniversalString or toString because they do not access the data directly.

Time Class Case Study: Overloaded Constructors (Cont.)

- Designing the class in this manner reduces the likelihood of programming errors when altering the class’s implementation.
- Similarly, each Time2 constructor could be written to include a copy of the appropriate statements from the three-argument constructor.
- Doing so may be slightly more efficient, because the extra constructor calls are eliminated.
- But, duplicating statements makes changing the class’s internal data representation more difficult.
- Having the Time2 constructors call the constructor with three arguments requires any changes to the implementation of the three-argument constructor be made only once.

Notes on Set and Get Methods

- Set methods are also commonly called **mutator methods**, because they typically change an object’s state - i.e., modify the values of instance variables.
- Get methods are also commonly called **accessor methods** or **query methods**.
- If an instance variable is declared private, a **public get method** certainly allows other methods to access it, but the get method **can control how the client can access it**.
- A **public set method** can - and should - carefully scrutinize attempts to modify the variable’s value to **ensure valid values**.

Notes on Set and Get Methods (Cont.)

- **Validity Checking in Set Methods**
- The benefits of data integrity do not follow automatically simply because instance variables are declared private - you **must provide validity checking**.
- **Predicate Methods**
- Another common use for accessor methods is to **test whether a condition is true or false** - such methods are often called predicate methods.
- Example: ArrayList's isEmpty method, which returns true if the ArrayList is empty and false otherwise.

Composition – Employee example

- A class can have **references to objects of other classes as members**.
- This is called **composition** and is sometimes referred to as a **has-a relationship**.
- Example:
  - An **AlarmClock** object needs to know the current time and the time when it’s supposed to sound its alarm, so it’s reasonable to **include two references to Time** objects in an AlarmClock object.
  - An **Employee** object needs to know the birth date and hire date and it **includes two references** to **Date** object.

enum Types

- The basic enum type defines a set of constants represented as unique identifiers.
- Like classes, all enum types are **reference types**.
- An enum type is declared with an enum declaration, which is a comma-separated list of enum constants
- The declaration may optionally include other components of traditional classes, such as **constructors**, **fields** and **methods**.

Enum Types (Cont.)

- Each enum declaration declares an enum class with the following restrictions:
- enum constants are **implicitly final**.
- enum constants are **implicitly static**.
- Any attempt to create an object of an enum type with operator new results in a compilation error.
- enum constants can be used anywhere constants can be used, such as in the case labels of switch statements and to control enhanced for statements.

Enum Types (Cont.)

- enum declarations contain two parts - the enum constants and the other members of the enum type.
- An enum constructor can specify any number of parameters and can be overloaded.
- For every enum, the compiler generates the static method values that returns an array of the enum’s constants.
- When an enum constant is converted to a String, the constant’s identifier is used as the String representation.
- enum example

Enum Types (Cont.)

- Use the static method ***range*** of class **EnumSet** (declared in package java.util) to access a range of an enum's constants.
- Method range takes two parameters - the first and the last enum constants in the range
- Returns an EnumSet that contains all the constants between these two constants, inclusive.
- The enhanced for statement can be used with an EnumSet just as it can with an array.
- Class EnumSet provides several other static methods.

Garbage Collection

- Every object uses system resources, such as **memory**.
- Need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur.
- The JVM performs automatic **garbage collection** to reclaim the memory occupied by objects that are **no longer used**.
- When there are no more references to an object, the object is **eligible to be collected**.
- Collection typically occurs when the JVM executes its **garbage collector**, which may not happen for a while, or even at all before a program terminates.

Garbage Collection (Cont.)

- Every class in Java has the methods of class **Object** (package java.lang), one of which is method **finalize**.
- You should **never use method finalize**, because it can cause many problems and there's uncertainty as to whether it will ever get called before a program terminates.
- The original intent of finalize was to allow the garbage collector to perform termination housekeeping on an object just before reclaiming the object's memory.

Garbage Collection (Cont.)

- Now, it’s considered better practice for any class that uses system resources - such as files on disk - to **provide a method that programmers can call to release resources** when they’re no longer needed in a program.
- **AutoClosable** objects reduce the likelihood of resource leaks when you use them with the try-with-resources statement.
- As its name implies, **an AutoClosable object is closed automatically**, once a try-with-resources statement finishes using the object.

static Class Members

- In certain cases, only one copy of a particular variable should be shared by all objects of a class.
- A static field - called a class variable - is used in such cases.
- A static variable represents **classwide information** - all objects of the class share the same piece of data.
- The declaration of a static variable begins with the keyword static.

static Class Members (Cont.)

- **Static variables have class scope** - they can be used in all of the class's methods.
- Can access a class's public static members through a reference to any object of the class, or by qualifying the member name with the class name and a dot (.), as in **Math.random()**.
- private static class members can be accessed by client code only through methods of the class.
- static class members are available as soon as the class is loaded into memory at execution time.
- To access a public static member when no objects of the class exist (and even when they do), prefix the class name and a dot (.) to the static member, as in **Math.PI**.

static Class Members (Cont.)

- To access a private static member when no objects of the class exist, provide a public static method and call it by qualifying its name with the class name and a dot.
- A static method cannot access a class's instance variables and instance methods, because a static method can be called even when no objects of the class have been instantiated.
- For the same reason, the this reference **cannot be used in a static method**.
- The this reference must refer to a specific object of the class, and when a static method is called, there might not be any objects of its class in memory.
- If a static variable is not initialized, the compiler assigns it a default value - in this case 0, the default value for type int.

Employee example 08_12_13

static Class Members (Cont.)

- **String** objects in Java are **immutable** - they cannot be modified after they are created.
- Therefore, it's safe to have many references to one String object.
- This is not normally the case for objects of most other classes in Java.
- If String objects are immutable, you might wonder why are we able to use operators + and += to concatenate String objects.
- **String-concatenation actually results in a new String** object containing the concatenated values - the original String objects are not modified.

static Class Members (Cont.)

- In a typical app, the garbage collector might eventually reclaim the memory for any objects that are eligible for collection.
- The JVM does not guarantee when, or even whether, the garbage collector will execute.
- When the garbage collector does execute, it’s possible that no objects or only a subset of the eligible objects will be collected.

static Import

- A static import declaration enables you **to import the static members of a class or interface** so you can access them via their unqualified names in your class - that is, the class name and a dot (.) are not required when using an imported static member.
- Two forms
  - One that imports a particular static member (which is known as single static import)
  - One that imports all static members of a class (which is known as static import on demand)

static Import (Cont.)

- The following syntax imports a particular static member:

```
import static packageName.ClassName.staticMemberName;
```

- where *packageName* is the package of the class, ClassName is the name of the class and *staticMemberName* is the name of the static field or method.
- The following syntax imports all static members of a class:

```
import static packageName.ClassName.*;
```

- where *packageName* is the package of the class and *ClassName* is the name of the class.
  - * indicates that *all* static members of the specified class should be available for use in the class(es) declared in the file.

static Import (Cont.)

- static import declarations import only static class members.
- Regular import statements should be used to specify the classes used in a program.
- StaticImportTest example

final Instance Variables

- The principle of least privilege is fundamental to good software engineering.
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but **no more**.
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.
- Keyword final specifies that a variable is not modifiable (i.e., it’s a constant) and any attempt to modify it is an error.

```
private final int INCREMENT;
```

  - Declares a final (constant) instance variable INCREMENT of type int.

final Instance Variables (cont.)

- final variables can be **initialized when they are declared** or by each of the class's constructors so that each object of the class has a different value.
- If a class provides multiple constructors, **every one would be required to initialize each final variable**.
- A final variable **cannot be modified by assignment after it's initialized**.
- If a final variable is not initialized, a compilation error occurs.

Package Access

- If no access modifier is specified for a method or variable when it’s declared in a class, the method or variable is considered to have **package access**.
- In a program uses multiple classes from the same package, these **classes can access each other’s package-access members directly through references to objects of the appropriate classes**, or in the case of static members through the class name.
- Package access is rarely used.
- PackageDataTest example

Using BigDecimal for Precise Monetary Calculations

- Any application that requires precise floating-point calculations—such as those in financial applications—should instead use class BigDecimal (from package java.math).
- **Interest** example

References

- Textbook
- Java documentation
  - https://docs.oracle.com/javase/tutorial/java/javaOO/index.html
  - https://docs.oracle.com/javase/tutorial/java/package/index.html
- http://www.vogella.com/tutorials/EclipseDebugging/article.html