CSE1030 – Introduction to Computer Science II

Lecture #2

Introduction to
Object Oriented Programming

Goals for Today

- Theory:
  - Learn a little about
    Object Oriented Programming

- Practical: (Assignment #1!)
  - How to create a Java class
  - What makes a class a Static or Utility Class
  - JavaDocs

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- We're Done!

Idea Behind OOP

- Make it easier to develop and maintain large or complex software systems

- Originated in the original Graphical User Interface research projects (complex!)

- Fundamental Ideas:
  - Organise Data and Code into Modules
  - Formalise the way one module interacts with another
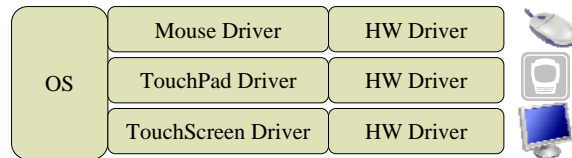   (We call this the **Interface** between the Modules)
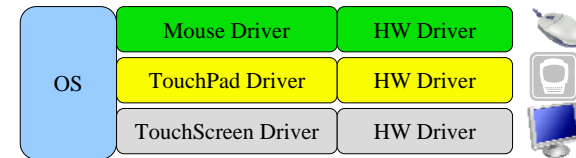
Sketchpad (1963)

Object Oriented Programming

- Object Oriented Programming
  - Example:

OS	Mouse Driver	HW Driver
	TouchPad Driver	HW Driver
	TouchScreen Driver	HW Driver

Object Oriented Programming

- Object Oriented Programming
  - **Is Not**: Task Oriented
  - **Is Not**: Library Oriented

OS	Mouse Driver	HW Driver
	TouchPad Driver	HW Driver
	TouchScreen Driver	HW Driver

Object Oriented Programming

- Object Oriented Programming
  - **Is**: A Different Way of **Thinking About** the Organisation of the Program and its Data

OS	Mouse Driver	HW Driver
	TouchPad Driver	HW Driver
	TouchScreen Driver	HW Driver
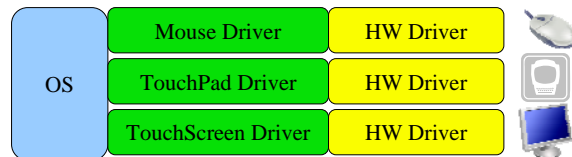
Why OOP?

- Encapsulation
  - Data & Code* in single well-defined location
  - Hide complexity away, on expose a simple API**

- Take Advantage of Inherent Relationships
  - Polymorphism
    - Objects that do similar things are often used similarly
  - Inheritance
    - Many things are "a kind of…" something else

 *Code = Software
 **API = Application Programming Interface

Java and OOP

- Java is an Object Oriented Language

- Big Idea:
  In Java, **Everything** is an **Object***

  (* almost, we'll talk more about this later)

- And a Java **Class** is how Objects are Defined

Java Classes

- Classes describe Objects ←Important Idea!
  (Every Object has a Class)

- Java Class Definition: (we'll come back to this)
  1. **Names** the Class
  2. Describes **How to Construct** an Object of the Class
  3. Stipulates **Who** can use our Objects, and **How**
  4. **Defines the Data** in the Objects (and in the Class)
  5. **Contains all of the Code** pertaining to the Objects

How it Normally Looks when Running
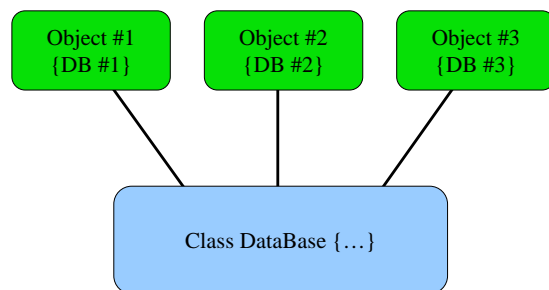
| Object #1 {DB #1} | Object #2 {DB #2} | Object #3 {DB #3} |

Class DataBase {…}

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- Parameters and Parameter Passing
- We're Done!

Elements of a Java Class

```
// any needed package statement
// any needed import statements

public class ClassName
{
   // data declarations
   private int i;

   // constructor
   ClassName(){ i = 0; };

   // method definitions
   int getI() { return i; }
   void setI(int pi) {i = pi; }
}
```

Elements of a Java Class

```
// any needed package statement
// any needed import statements

public class ClassName
{
   // data declarations
   private int i;

   // constructor
   ClassName(){ i = 0; };

   // method definitions
   int getI() { return i; }
   void setI(int pi) {i = pi; }
}
```

1. Name the Class

2. How to Construct an Object

3. Who can use, and How

4. Defines the Data

5. Contains the Code

Elements of a Java Class - Name

1. Name the Class

- The name should be unique and meaningful
- Class name must be the same as the name of the file, for example this class should be saved in a file called "ClassName.java"

```
// any needed package statement
// any needed import statements

public class ClassName
{
   // data declarations
   private int i;

   // constructor
   ClassName(){ i = 0; };

   // method definitions
   int getI() { return i; }
   void setI(int pi) {i = pi; }
}
```

Elements of a Java Class - Constructor

2. How to Construct an Object

- This function is called the **Constructor**
- Its primary purpose is to initialise the data elements (like the variable **i**)
- It must have the same name as the class
- It does not have a "return type", its return type is implicit
- If you do not write a constructor for a class, then one is automatically generated for you
- Consequently, if you do not want people to be able to create objects of this type, you must use an access specifier (e.g., "private")

```
// any needed package statement
// any needed import statements

public class ClassName
{
   // data declarations
   private int i;

   // constructor
   ClassName(){ i = 0; };

   // method definitions
   int getI() { return i; }
   void setI(int pi) {i = pi; }
}
```

Elements of a Java Class - Access

3. Who can use, and How

- Access Specifiers control who has access to what parts:
  - public – everybody
  - protected – only related classes (this class, subclass, package*)
  - private – only this class
  - 'none' ~ protected (no sub classes)
- Only 1 public class per file (class with the same name as the file)
- Can be applied to members too:
  private int i;

* A **Package** is a group of Classes. Classes without package specifiers go into the **default** package.

```
// any needed package statement
// any needed import statements

public class ClassName
{
    // data declarations
    private int i;

    // constructor
    ClassName(){ i = 0; };

    // method definitions
    int getI() { return i; }
    void setI(int pi) {i = pi; }
}
```

Elements of a Java Class - Data

4. Defines the Data

- Data within a class is usually made private
- Accessor functions are used to control access to this internal data, examples
- Mutators change data values
- The data members are initialised by the constructor

```
// any needed package statement
// any needed import statements

public class ClassName
{
    // data declarations
    private int i;

    // constructor
    ClassName(){ i = 0; };

    // method definitions
    int getI() { return i; }
    void setI(int pi) {i = pi; }
}
```

Elements of a Java Class - Code

5. Contains the Code

- All operations pertaining to the objects of this class should be performed by functions defined right here, in this class.
- This way, everybody can see all of the related data and code in one place.

```
// any needed package statement
// any needed import statements

public class ClassName
{
    // data declarations
    private int i;

    // constructor
    ClassName(){ i = 0; };

    // method definitions
    int getI() { return i; }
    void setI(int pi) {i = pi; }
}
```

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- We're Done!

Definition of a Utility Class

- A Class that contains a common often re-used function (or family of functions)…

- No Objects – usually they are collections of functions

- Examples:
  - java.lang.Math
  - java.lang.System
  - java.util.Collections

No Objects?

- Why would we use an Object Oriented Language to write code that doesn't have any objects?

- ANSWER: What if I have a very simple little thing that just doesn't need classes? Like Adding a Couple of Numbers? What code do I actually need?

Example: Objects Not Necessary

```
public class DeclareVariables
{
    public static void main(String[] args)
    {
        int A = 10;
        int B = 20;
        int C = A + B;

        System.out.println("The answer is: " + C);
    }
}
```

Example: But it's still a class

Declare the Class

```
public class DeclareVariables
{
    public static void main(String[] args)
    {
        int A = 10;
        int B = 20;
        int C = A + B;

        System.out.println("The answer is: " + C);
    }
}
```

Define a Member: **main()**

The `main()` Function

- The main function is where execution of all java programs begins

- All classes can have a main function
  - Even if there are more than one class, each can have it's own main function
  - The only main function that matters is the one in the controlling class – that is the one that will be run

- The main function is labelled static, meaning that an object is not needed to run the main function
  - That's great if we don't want the added complexity of having objects around

Problem...

Even though we haven't provided a constructor in our example, Java will automatically create one for us.

So to ensure that nobody creates an object of a class we don't want them to, we have to *disable* the constructor by making it **private**

Now we have a Class, but no Objects

```
public class DeclareVariables
{
    private DeclareVariables() {};

    public static void main(String[] args)
    {
        int A = 10;
        int B = 20;
        int C = A + B;

        System.out.println("The answer is: " + C);
    }
}
```

Characteristics of Utility Classes

- Want to make this functionality available to others

- Usually Utilities are collections of useful functions, rather than stand-alone programs

The AdditionUtility Class

```java
public class AdditionUtility
{
    private AdditionUtility() {};

    public static int add(int A, int B)
    {
        return A + B;
    }
}
```
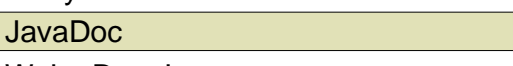
Summary: Utility Classes

- Private Constructor

- **All** members, data and code, must be labelled **static**

- Usually does not contain a main() function

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- We're Done!

Intro to JavaDoc

- How do we make it easy for other programmers (or even ourselves) to use our classes?

- Users need to know the API

- JavaDoc provides semi-automatic generation of API documentation suitable for viewing in a browser

JavaDoc Comments

```
/**
 * This class defines a function for
 * adding two numbers
 */
public class AdditionUtility
{
    private AdditionUtility() {};

    /**
     * This function adds two numbers.
     */
    public static int add(int A, int B)
    {
        return A + B;
    }
}
```

Running javadoc

- **javadoc AdditionUtility.java**

- This command reads the Java source file and generates the API documentation.

- Several files are created, open the one called "index.html" in your web browser

Class AdditionUtility

Note! This is an old-style javadoc

java.lang.Object
  └ AdditionUtility

public class **AdditionUtility**
extends java.lang.Object

This class defines a function for adding two numbers

Method Summary

static int	add(int A, int B)
	This function adds two numbers.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

add

public static int **add**(int A,
                    int B)

This function adds two numbers.

JavaDoc Tags…

- It would be nice to include more information regarding our function

- How about, what the parameters are, what's the return value?

Adding Details to `add()`

```
/**
 * This function adds two numbers.
 *
 * @param A A number to add
 * @param B Another Number to add
 * @return The sum, A + B
 */
public static int add(int A, int B)
{
    return A + B;
}
```

Method Detail

add

```
public static int add(int A,
                      int B)
```

This function adds two numbers.

Parameters:
    A - A number to add
    B - Another number to add
Returns:
    The sum, A + B

Preconditions

- Preconditions are instructions made to the users of your function

- You should **always check** the validity of **your function's parameters**

- But **if you have limits** in what you can handle, **tell the user** – use a precondition!

Adding javadoc Preconditions

```
/**
 * This function adds two numbers.
 *
 * @param A A number to add
 * @param B Another Number to add
 * @pre. There are no preconditions
 * @return The sum, A + B
 */
public static int add(int A, int B)
{
    return A + B;
}
```

Preconditions in JavaDoc

- Preconditions are not directly supported in this version of javadoc

- So we use a custom tag: **@pre.**

- And we run a more complicated javadoc command:
  ```
  javadoc –tag param –tag pre.:a:"Precondition: "
    –tag return AdditionUtility.java
  ```

Method Detail

add

```
public static int add(int A,
                      int B)
```

This function adds two numbers.

Parameters:
   A - A number to add
   B - Another number to add
Precondition:
   There are no preconditions
Returns:
   The sum, A + B

Final JavaDoc Notes

- There are other tags:
  - @author, @version, @see, @throws, etc.

- You can use HTML tags in the comments

- More information about defining your own custom tags appears in the online javadoc documentation:
  http://docs.oracle.com/javase/1.4.2/docs/tooldocs/windows/javadoc.html#tag

---

CSE1030 – Lecture #2

- Intro to Object Oriented Programming
- Elements of a Java Class
- Utility Classes
- JavaDoc
- We're Done!

Next topic…

Non-Static Object Features