Object Oriented Programming with Java

What is Object Oriented Programming?
Object Oriented Programming consists of creating outline structures that are easily reused over and over again. There are four elements of Object Oriented Programming:

- Abstraction – Generalization for Relationship Inheritance
- Encapsulation – Encoding of Data
- Inheritance – Relationships between "parent-child" objects
- Polymorphism – Flexibility of interchangeable parts / reusability

Imagine that there is an object known as "Automobile." An automobile has properties such as a year, model, make, color, etc. An automobile also has actions, such as drive forward, turn right, turn left, etc. The properties are known as "Attributes," and the actions are known as "Methods."

We can also create multiple instances of the object "Automobile," one for each student in a classroom. Each student owns a car, and no two students have the same vehicle; however, each student's car has a model and make and each car has the same action abilities. We can use a **UML Class Diagram** as a development tool to design abstract ideas of objects; we will refer to objects as classes.

```
                 Class Automobile
Attributes:
     Year  :: number
     Model :: text
     Make  :: text
     Color :: text

Methods:
     Drive(distance, speed, direction)
     Turn(speed, direction)
     PrintLocation()
```

The UML Class Diagram is a table that breaks a class into two categories: Attributes and Methods. The example UML Class diagram above follows the Automobile model discussed earlier. It contains the attributes *Year*, *Model*, *Make*, and *Color* as well as three methods: *Drive*, *Turn*, and *PrintLocation*. The each attribute has a **data type** next to it to describe the format of information stored for that attribute. The *Year* attribute will have a number, and the *Model*, *Make* and *Color* attributes will all have text for a data type.

The Methods are actions that an "Automobile" object can perform. Methods often expect data before performing a function. For example, an *Automobile* cannot drive forward without knowing the *Speed*, *Distance*, and *Direction* of the movement. *PrintLocation* does not require previous information; it will be used simply to output the current location of the car.

You can navigate to specific sections of this handout by clicking the links below.

Data Types vs. Classes
We mentioned data types in the UML diagram; data types are the set of possible values that could exist for an instance of data. For example, the data type of a student's age would be a number, and the data type of their name would be text. Below is a table of common data types within various programming languages. The Type column contains variable types used in Java.

Type	Description	Example Usage
char	A single ASCII character	Letter Grade of a student
String	A series of ASCII characters	Student's Name
int	A short integer	Age of a student
long	A long integer	Student ID number
float	A decimal number	Balance of Student Account
double	A precise decimal	Grade Point Average of Student
boolean	A true or false value	Commuting/Non-commuting Student

We can create a class of anything we want; however, we still have to create the values within a class based on the principles of these basic data types. Additionally we can create classes of previously created classes and objects. This process is actually very beneficial to abstract critical thinking. We could create a "Student" class that includes our previously created "Automobile" class. For example: each student has a name(String), an age(int), an ID number(long), a GPA(double), and a car(Automobile).

Variables: Scope, regulations and class over-loading
The "Automobile" class consists of attributes and methods. Attributes are known as **variables**. Algebraically speaking, variables represent an unknown—and more importantly, they are used to perform calculations regardless of the individual value. In programming, variables are actually memory locations; the memory is used to store values assigned to a variable. Variables, classes, functions, etc. have a few rules regarding usage and assignment.

- Variable names are case-sensitive:
    - studentAGE is a different variable instance than STUDENTage
- Variable names cannot be pre-existing *reserved* commands:
    - Cannot name a variable char, String, int or any other data type…
- Variable names cannot begin with a number, contain symbols, or spaces other than an underscore ("_"):
    - Cannot name a variable: 08Students, #phoneNumber, student age…

The scope of a variable refers to the life-time and locality of a value. For example, if we create five students that each have a name, GPA, class schedule, etc. it is unnecessary for studentA to have access to studentB's GPA. We would make the GPA a *Private* variable to isolate its availability to within the instance of a student. The process of making variables *Private* vs. *Public* can help prevent unnecessary repetition, recursion, and memory allocation. Usually, all attributes are created as *Private* and all methods are *Public*. This process then calls for two additional methods that allow a program to read and write to the *Private* variable from outside the class.

Class **overloading** is the process of calling a function and giving it different **parameters**. Parameters are the required information for a function/class to operate. The "Automobile" class created on the first page requires distance, speed, and direction to call the function *Drive*. We can set up the *Drive* function to have a default direction, "Forward," and only require a distance and speed. This is an example of *polymorphism*, adding to the overall flexibility of the class. When the *Drive* function is called, the class will automatically decide which version of *Drive* to use based on the number of parameters that were given to it. A common error that occurs while overloading classes and functions is **Data Type Mismatch**. This error is caused by a function receiving a different data type value than what it was expecting. For example, we cannot perform mathematic operations on Strings and text.

Command Syntax
Syntax is the proper way to call or use a function. The following are a few important general Java formatting syntax requirements and tips:

- Complete statements of code should always have a semi-colon(;) at the end

- Grouped code segments such as statements within a loop or class, should be enclosed within curly braces "{" and "}"
- Order of Operations (Order of Precedent) is a non-configurable guideline for mathematical procedures
- Parameters sent to a function are enclosed in parentheses and should have the same data types as what is expected
- Creating **constructors** is a good way to control class creation; constructors are a default setting for a created class.

The following are examples of proper syntax and description of common functions & commands within the Java language.

Declarations:
Main Operational Class: This is used to create a "main executable" class to operate a program. It operates as the main code routine.

```
Public class Example ()
{
     Public static void main (String args[])
     {
          ... Insert main code here
     }
}
```

Variable Declarations/Assignments: The following creates integer variables x, y, and z, as well as two String variables. These variables are local or private, and cannot be seen outside of the class. String variable test is actually assigned a value during declaration. We can declare multiple variables in the same line of code as long as they are of the same data type; they should be separated by a comma.

```
Public class example ()
{
     Private int x;
     Private int y,z;
     Private String test= "This is a String Variable";
     Private String Name;
}
```

Documentation – adding comments and reminders is important in designing and re-designing a program. Documentation also proves useful when debugging and trying different features.

Single-Line Comment- used for documenting the rest of a line of text.

```
//    Comment
```

Block Comment – used for larger comments such as describing a process or documenting authorship.

```
/*    Comment Line 1
      Comment Line 2
      Comment Line 3      */
```

Import – used to include previously designed classes/functions within a class. Imports are the very first lines of a program.

```
Import Java.Util.Scanner; // Used to import class Scanner
Import Java.Awt;          // imports a graphical class
```

Conditional Operands and Arithmetic Symbols

Expression	Description	Expression	Description
`n1 + n2`	*Addition*	`n1 == n2`	*Equality*
`n1 - n2`	*Subtraction*	`n1 != n2`	*Does not equal*
`n1 * n2`	*Multiplication*	`n1 > n2`	*Greater than*
`n1 / n2`	*Division, integer quotient after division*	`n1 >= n2`	*Greater than or equal to*
`n1 % n2`	*Modulus, integer remainder after division*	`n1 < n2`	*Less than*
`n1 = n2`	*Assignment*	`n1 <= n2`	*Less than or equal to*

Loops & repetition: Looping and repetition is used to reuse sections of code that are required multiple times. Looping is very useful for reviewing conditions, counting, and handling user-entry error. The only major caution with looping is to avoid infinite loops; infinite variable creation and storage can cause serious computer problems.

Finite Loop – For/Next Loop – best used for counting; the exit condition is checked before executing the statements within the loop.

```
For (variable; condition; increment)
{
      statements in loop…
}
```

While/Do Loop – best used for indefinite loops where an unknown quantity of code executions is required; the exit condition is checked before executing the statements within the loop.

```
While ( condition )
{
      Statements in loop…
}
```

Conditional Statements: Conditional statements are used to make decisions within a program or application. There are multiple conditional statement types; which one to use depends on the situation. A simple yes/no question would best be decided by an *If/Else* statement. If a decision needs to be made regarding different values such as outputting a response based on what grade a high school student is in, a *Switch-Case* statement would be more efficient. A process known as nesting can also be used by embedding *If/Else* statements within *If/Else* statements.

If/Else Statement

```
If ( condition )
     {
          Statements if condition is true
     }
Else
     {
          Statements if condition is false
     }
```

Switch-Case Statement – use *break* to exit switch-case as it will continue to test cases.

```
Switch ( variable )
{
     Case (value1):
          Statement if variable = value1;
          Break;
     Case (value2):
          Statement if variable = value2;
          Break;
     …
     Default:
          Statement if variable is anything else;
}
```

More Information:
For more information regarding specific Java functions, visit the online tutorials and documentation from the creators of Java technology.

Tutorials:
http://java.sun.com/docs/books/tutorial/index.html

API Documentation:
http://java.sun.com/j2se/1.5.0/docs/api/