**OOP Structure**

# **Objects**

## Storage: Objects store state at runtime

## Behavior: Objects will also in some sense take an active role. Each object has a set of operations that it can perform, usually on itself. (Methods)

## Class:

## Every object belongs to a class that defines its storage and behavior.

### An object always remembers its class (in Java).

### "Instance" is another word for object -- an "instance" of a class.

### Procedural variables are passive -- they just sit there. A procedure is called and it changes the variable.

### Objects are has both storage and behavior to operate on that state.

## String example

### We could have a couple String objects, each of which stores a sequence of characters. The objects belong to the String class. The String class defines the storage and operations for the String objects...

String a = “Hello”

public class String{

length(){

-----

-----

}

reverse(){

-----

-----

}

}

Sending a length()

Message to a String Object..

# **Class**

## Exists once -- there is one copy of the class in memory.

## Defines the storage and behavior of its objects

## **Storage:**

### Define the storage that objects of this class will have.

### "instance variables" -- the variables that each object will use for its own storage. Instance variables are usually just called "ivars".

## **Behavior:**

### Define the behaviors that objects of this class can execute (methods).

## **String example**

### The String class defines the storage structure used by all String objects -- probably an array of chars of some sort

### The String class also defines the operations that String objects can perform on themselves -- length(), reverse(), ...

# **Message / Receiver**

## Suppose we have Student objects, each of which has a current number of units. The message getUnits() requests the units from a student.

## Java syntax: a.getUnits() send the "getUnits()" message to the receiver "a"

## **Receiver:**

### The "receiver" is the object receiving the message. Typically, the operation uses the receiver's memory.

# **Method (code)**

## A "method" is executable code defined in a class.

## The objects of a class can execute all the methods their class defines.

## The String class defines the code for length() and reverse() methods. The methods are run by sending the "length()" or "reverse()" message to a String object.

# **Message -> Method resolution**

# 

## Suppose a message is sent to an object --- x.reverse();

## The receiver, x, is of some class -- suppose x is of the String class

## Look in that class of the receiver for a matching reverse() method (code)

## Execute that code "against" the receiver-- using its memory (instance variables)

## In Java this is "dynamic" -- the message/method resolution uses the true, run-time class of the receiver.

# **OOP Design - Anthropomorphic, Modular**

## Objects responsible for their own state -- as much as possible, objects do not reach in to read or write the state of other objects.

## Objects can send messages to each other -- requests

## The object/message paradigm makes the program more modular internally. Each class deals with its own implementation details, but can be largely independent of the details of the other classes. They just exchange messages.

# **OOP Design Rule #1 -- Encapsulation**

## Objects "protect" their own state from direct access by other objects -- "encapsulation". Other objects can send requests, but only the receiver actually changes its own state. This allows more reliable software -- once a class is correct and debugged, putting it in a new context should not create new bugs.

## Abstraction vs. Implementation

### This is the old Abstract Data Type (ADT) style of separating the abstraction from the implementation, but structured as messages (abstraction) vs. methods (implementation)

# **OOP Design Process**

## Think about the objects that make up an application

## Think about the behaviors or capabilities those objects should have

## Provide the objects with those abilities as methods

## If a capability does not occur to you in the initial design, that's ok. Add it to the appropriate class when needed -- they just needs to go in the right class

## Co-operation

### Objects send each other messages to co-operate

## Tidy style

* + Experience shows that having each object operate on its own state is a pretty intuitive and modular way to organize things

# **Advantages of Object-oriented Programming**

* Elimination of redundant code due to inheritance, that is, we can use the same code in a class by deriving a new class from it.
* Modularize the programs. Modular programs are easy to develop and can be distributed independently among different programmers.
* Data hiding is achieved with the help of encapsulation.
* Data centered approach rather than process centered approach.
* Program complexity is low due to distinction of individual objects and their related data and methods.
* Clear, more reliable, and more easily maintained programs can be created.

###### **Classes and Objects**

* As we have discussed earlier, a class is a framework that specifies what data and what methods will be included in objects of that class.
* A class is also called user defined data type or programmers defined data type because we can define new data types according to our needs.
* Syntax:

[Access][Modifiers] class className [extends superClass]

[implements interface1[, interface2][, ……]]{

//variables and methods

}

* **Components:**
  + **Access:** public, private, and others discussed later.
  + **Modifiers:** static, final and others discussed later.
  + **Class name:** The class name, with the initial letter capitalized by convention.
  + **Superclass:** The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class (subclass) can only extend one parent (superclass).
  + **Interfaces:** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
  + **Body:** The class body, surrounded by braces, {}.
* **Adding Variables:**
  + Data is encapsulated in a class by placing data fields inside the body of the class definition.
  + Syntax:

[Access][Modifiers] data-type variableName;

* **Components**:
  + **Access:** public, private, and others discusses later.
  + **Modifiers:** static, final and others discussed later.
  + **Data type:** type of the variable. It may be primitive or user defined.
  + **Variable name:** name of the variable
* **Adding** **Methods**:
  + To manipulate data contained in the class we add methods inside the body of the class definition.
  + Syntax:

[Access][Modifiers] return-type methodName(ArgumentList)

[exceptions lists]{

//body

}

**Components:**

* **Access:** public, private, and others discussed later.
* **Modifiers:** static, final and others discusses later
* **Return-type:** the data type of the value returned by the method or void if the method does not return a value.
* **Method name:** valid identifier with convention of starting with small letter, first word as a verb and the later words start with capital letter.
* **The parameter list in parenthesis:** a comma-separated list of input parameters, preceded by their data types, enclosed by parentheses. If there are no parameters, you must use empty parentheses.
* **Exception list:** discussed later.
  + **The method body, enclosed between braces:** the method's code, including the declaration of local variables, goes here.

Example:

class Rectangle {

private int length; //variable declaration

private int breadth; //variable declaration

public void setData(int l, int b) { //method definition

length = l;

breadth = b;

}

public int findArea() { // method definition

return length \* breadth;

}

public int findPerimeter() { // method definition

return 2 \* (length + breadth);

}

}

* **Creating Objects**
  + Objects are created using the **new** operator.
  + The **new** operator creates an object of the specified class and returns a reference to that object.
  + The main idea of using **new** is to create the memory that is required to hold an object of the particular type in run time i.e. to dynamically allocate the memory at the run time.
  + To create a usable object we must finish two steps:
    - Declaring the variable of its type and
    - Instantiating the object.

Rectangle myRectangle; //declaring the variable of type Rectangle

myRectangle = new Rectangle(); //Instantiating an object

* + - The execution of first statement creates the variable that holds reference of the class Rectangle using the name myRectangle. It points nowhere (i.e. null) as shown below:

**null**

myRectangle

myRectangle

length

breadth

Rectangle Object

* + - When the second statement is executed then the actual assignment of object reference to the variable is done as shown in above figure (right). The above two steps are equivalently written as:

Rectangle myRectangle = new Rectangle();

* + - We can create more than one reference for same object and using different name we can manipulate same object. For example,

Rectangle myRectangle = new Rectangle();

Rectangle myRectangle1 = new Rectangle();

myRectangle = myRectangle1;

* + - The pictorial representation of above code segment is as below:

myRectangle

length

breadth

Rectangle Object

myRectangle1

* Using Class Members
  + When an object of the class is created then the members are accessed using the ‘.’ dot operator.

myRectangle.setData(5, 2);

myRectangle.findArea();

myRectangle.findPerimeter();

**Remember:** Not all the members can be accessed using the dot ‘.’ operator.

* + Complete Example

class Rectangle {

private int length;

private int breadth;

public void setData(int l, int b) {

length = l;

breadth = b;

}

public int findArea() {

return length \* breadth;

}

public int findPerimeter() {

return 2 \* (length + breadth);

}

}

class MainRectangle {

public static void main(String[]args) {

Rectangle myRectangle = new Rectangle();

Rectangle myRectangle1 = new Rectangle();

myRectangle.setData(5, 2);

myRectangle1.setData(9, 4);

System.out.println("First rectangle:");

System.out.println("Area = " + myRectangle.findArea());

System.out.println("Peremeter = " + myRectangle.findPerimeter());

System.out.println("Second rectangle:");

System.out.println("Area = " + myRectangle1.findArea());

System.out.println("Peremeter = " + myRectangle1.findPerimeter());

}

}

**Output:**

First rectangle:

Area = 10

Perimeter = 14

Second rectangle:

Area = 36

Perimeter = 26

* Setters and Getters
  + These are also called set and get methods.
  + We use setters and getters to provide the interface to modify and use fields (private) within the class respectively.

class Rectangle {

private int length;

private int breadth;

public int getLength() { return length;} //getter

public int getBreadth() {return breadth;} //getter

public void setBreadth(int b) {breadth = b;} //setter

public void setLength(int l) {length = l;} //setter

public int findArea() {return length\*breadth;}

public int findPerimeter() {return 2\*(length+breadth);}

}

class MainRectangle {

public static void main(String [] args) {

Rectangle rect1 = new Rectangle();

rect1.setLength (10);

rect1.setBreadth(5);

System.out.println("Length: "+ rect1.getLength());

System.out.println("Breadth: "+ rect1.getBreadth());

System.out.println("Area: " + rect1.findArea());

System.out.println("Perimeter: " + rect1.findPerimeter());

}

}

**Output:**

Length: 10

Breadth: 5

Area: 50

Perimeter: 30

* + **Remember:** If a method changes the value of variable(s), it is called **mutator method**. For example, set method in a class. The method that does not change but access value(s) is called **accessor** or **query method**. For example get method in a class.
* **Constructors**
  + The constructor initializes the variables of the object immediately after its creation.
  + Constructor has same name as of class and looks like a method except that it has no return type.
  + We have already seen the use of the constructor in above programs as Rectangle() [default constructor]. Though we have used the constructor we have not defined it and if we do not define the constructor the default constructor is called automatically.
  + if we define constructor ourselves then there will be no default constructor so in case of our need of default constructor we have to create by ourselves.
  + Since constructor looks like method we can also provide arguments (parameters) to the constructor.

class Rectangle {

private int length;

private int breadth;

public Rectangle() { // default constructor

}

public Rectangle(int l, int b) { //parameterized constructor

length = l;

breadth = b;

}

public void setLength(int l) {

length = l;

}

public void setBreadth(int b) {

breadth = b;

}

int findArea() {

return length\*breadth;

}

int findPerimeter() {

return 2\*(length+breadth);

}

}

class MainRectangle {

public static void main(String [] args) {

Rectangle rect1 = new Rectangle(10, 5);

Rectangle rect2 = new Rectangle();

rect2.setLength (7);

rect2.setBreadth( 3);

System.out.println("First rectangle");

System.out.println("Area:" + rect1.findArea());

System.out.println("Perimeter:" + rect1.findPerimeter());

System.out.println("\nSecond rectangle");

System.out.println("Area:" + rect2.findArea());

System.out.println("Perimeter:" + rect2.findPerimeter());

}

}

**Output:**

First rectangle

Area: 50

Perimeter: 30

Second rectangle

Area: 21

Perimeter: 20

* **Parameters to the Methods and Return Type**
  + In Java method can be defined to take unknown number of parameters.
  + Java methods can take any type of data (primitive as well as objects) and return any type of value (primitive as well as objects).
  + If the method takes parameter as primitive type then passing of argument from the caller is done as **call by value**
  + If the parameters are of object type the calling method is **call by reference**. The same process happens for the return type.
  + **Call by value**
    - In this method the value of the argument passed to the method. In this case, actual parameter is copied to the formal parameter of the method so that changes made to the parameter inside the method are not reflected to the actual parameter.
  + **Call** **by** **reference**
    - In this method, unlike the value, reference to an argument (actual parameter) is passed to the formal parameter so that they refer to the same location. In this process the modification to the parameter inside the method is reflected to the actual parameter.

**Example:**

class ParameterPass {

public static void main(String[]args) {

int a = 7;

Rectangle r1 = new Rectangle(10,5);

Rectangle.increment(a); //method call with primitive type parameter

Rectangle.increment(r1); //method call with object type parameter

System.out.println("a = " + a);

r1.display();

}

}

class Rectangle {

private int length;

private int breadth;

Rectangle(int l, int b) {

length = l;

breadth = b;

}

static void increment(int aa) {

aa++;

}

static void increment(Rectangle r) {

r.length++;

r.breadth++;

}

public void display() {

System.out.println("Length = " + length);

System.out.println("Breadth = " + breadth);

}

}

**Output:**

a = 7

Length = 11

Breadth = 6

* **This Keyword**
  + The keyword **this** can be used within any method to represent the current object i.e. **this** is the reference to the object on which method is getting invoked.

class Rectangle {

private int length;

private int breadth;

public Rectangle(int length, int breadth) {

this.length = length;

this.breadth = breadth;

}

………

………

}

* + Though normally in java declaring two variables within the same scope is illegal it is permissible to have same name as parameter of a method and instance variables. In this case, we use **this** keyword to identify the instance variables.
* **Access Control**
  + Public
  + private,
  + protected,
  + No modifier (package-private).
* If we need our class to be visible to all other packages then we must declare it as public using “public” specifier.
* If we need it to be visible in the same package only, we use no specifier. The use of other specifiers is not allowed.
* At member level we can use all the possible specifiers
* If the member has no specifier then it is visible to all the classes within the package. This is often called package access.
* If the member has **public** specifier then it is visible to all the classes in all the packages.
* If the member has **private** specifier then it is visible only inside the top level class and nowhere else.
* If the member has **protected** specifier then it is visible only inside the package and in the subclass outside the package.

**Remember:** If a class in a package is not public, then none of its members will be visible outside that package.

* **Static**
  + The static modifier in java can be used in the following parts:
* Variables : class variables
* Methods : class methods
* Blocks – These are blocks within a class that are executed only once, usually for some initialization. They are like instance initializers, but execute once per class, not once per object.
* Classes – Classes nested in another class.
* **Static Blocks**
  + If we need to do computation in order to initialize static variables, we can declare a static block, which gets executed exactly once, when a class is first loaded.
  + Static blocks are also called **static initialization blocks**. A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

class StaticBlock {

static int a = 3;

static int b;

static {

System.out.println("Staic block initialized");

b = a \* 4;

}

public static void main(String[]args) {

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

**Output:**

Staic block initialized

a = 3

b = 12

**Note:** There is an alternative to static blocks —you can write a private static method:

class Whatever {

public static varType myVar = initializeClassVariable();

private static varType initializeClassVariable() {

//initialization code goes here

}

}

* **Final**
* The final modifier in java can be used in the following ways:
* Using final with variables
  + A variable (primitive or user defined) can be declared as final to prevent its content from being modified.
  + If we try to change the content of the final variable the compile time error occurs.

final static String CLASS = “lftechnology”;

final Student S1 = new Student(“John”, 1);

final int VAR1 = 20;

* + It is a common coding convention to choose all uppercase identifiers for final variables
  + Variables declared as final do not occupy memory on a per-instance basis.
  + Java also supports the blank final variables i.e. final variables without initialize value at declaration. When you initialize final variable later then you cannot change it later.
  + If all variables of a class are final, then the class is immutable and values in the variables of the objects never change after they are initialized.

class MyClass {

final String name; //blank final variable

MyClass (String s) {

name = s; //initialization of final variable

}

………

………

}

* **Nested and Inner Classes**
  + It is possible to define a class within another class. Such classes are known as **nested classes**. They are same as any other class except that they are defined inside the body of another class

class Outer {

int outer\_x = 100;

void test() {

Inner in = new Inner();

in.display();

}

class Inner {

void display() {

System.out.println(outer\_x);

}

}

}

* + There are two types of nested classes: **static** and **non-static**.
  + An **inner class** is a non-static nested class and it can be defined outside a method and inside a method. Some of the reasons behind using nested classes are listed below:
    - **Logical grouping of classes**
    - **Increased encapsulation**
    - **More readable, maintainable code**.
* **Garbage Collection**
  + We use **new** keyword to dynamically allocate the memory.
  + Since the memory is created it must be destroyed to free the space if the object is not referenced by any variable.
  + The process of freeing up memory used by objects that are not used is called garbage collection and this process is automatically done by java. It is not true that whenever the object is not used java runs the garbage collector but the garbage collector is run occasionally. Normally, we do not have to take care about this thing in our programming.
  + If you want to call the garbage collector by youself then you can use the static method **gc** of the **System** class as:

System.gc();

* **Life Cycle of Object**

Steps:

* Object memory allocation
* Initialization of variables
* Call of constructor (appropriate one)
* Use of object in program
* Disassociation of reference to the memory by an object
* Garbage collection run
* Calling the object's finalize (if any) method (by garbage collector).
* Memory freeing by garbage collector.

**Lab:**

1. Write a Guessing a number game application: User will select the guessing value from the keyboard and application will generate a random no between 1 and 100. Check if the guess value matched with random no. or not, if matched show the guessing value, random value and the no. of attempts.
2. Create a class called Distance that includes two private data members feet (type int) and inches (type float). Use a method setData to set values to instance variables. Provide a method display that displays the feet and inches, a member function addDistance for adding two distances, and a member function compareDistance for comparing two distances. Implement your program in Java.
3. Write a program that calculates percentage and GPA of 5 students. Each of the students has following attributes: name, class, rollnum, and marks obtained in 5 subjects.
4. Create a class called Time with three attributes hours, minutes, and seconds. Use appropriate constructor (s) to initialize instance variables. Also, use a display method to display the time in hh:mm:ss format. Modify the class to add two time objects that correctly results in addition of times.
5. Create a class called Employee that includes three pieces of information as instance variables a first name (type String), a last name (type String) and a monthly salary (double). Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.
6. Create a class called Date that includes three pieces of information as instance variables a month (type int), a day (type int) and a year (type int). Your class should have a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a set and a get method for each instance variable. Provide a method displayDate that displays the month, day and year separated by forward slashes (/). Write a test application named DateTest that demonstrates class Date's capabilities.
7. Create class SavingsAccount. Use a static variable annualInterestRate to store the annual interest rate for all account holders. Each object of the class contains a private instance variable savingsBalance indicating the amount the saver currently has on deposit. Provide method calculateMonthlyInterest to calculate the monthly interest by multiplying the savingsBalance by annualInterestRate divided by 12. This interest should be added to savingsBalance. Provide a static method modifyInterestRate that sets the annualInterestRate to a new value. Write a program to test class SavingsAccount. Instantiate two savingsAccount objects, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set annualInterestRate to 4%, then calculate the monthly interest and print the new balances for both savers. Then set the annualInterestRate to 5%, calculate the next month's interest and print the new balances for both savers.