OOD Principles

Object-Oriented Programming

Programming style in which a program is split into self-contained objects, with each object representing different parts of the program and contains its own data and logic. These objects interact with each other.

Pros: Allows for modular programs so that programmers can create modules that don't need to be changed when a new type of object is added (means programs are easier to modify). Instead, they can create a new object that inherits features from the existing objects.

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Reusability: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Object – Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). An object stores its state in variables and exposes its behavior through methods.

Class – Blueprint from which objects are created. Includes local, instance and class variables. *Constructor* is a method that has the same name as the class name – job is to create instance of class.

```
public class Dog{
        String breed;
        int ageC
        String color;
        void barking(){}
```

```
        void hungry(){}
        void sleeping(){}
}
```

Packages – categorizing similar classes and interfaces

Principles of OOP

**Inheritance* – When classes inherit commonly used variables and methods from other classes. E.g. Animals includes subclasses dog and cat. Use "extends" keyword.

Pros: reusability, extensibility, data hiding
Cons: classes (base and inherited) gets tightly coupled and so isn't independent

**Encapsulation* – Used to refer to two notions
1) Encapsulation of Data – process of encapsulating variables and methods together in a unit. Variables of a class will be hidden from other classes depending on which access modifiers are used – know as **data** hiding. Each class in Java is an ex of encapsulation.

2) Restricting access to object's components - variables of a class will be hidden from other classes by making variables private, and can be accessed only through the methods of their current class. *Setter and getter method* used to update the private variables via public methods. **Allows for data hiding** because protects variable from outside world.

Pros: data hiding, can make class read-only or write-only by providing only setter and getter methods, also, provides you control over the data b/c you set the values through setter and getters

```
public class EncapsulationDemo{
        private int ssn;
        private String empName;
        private int empAge;

        //Getter and Setter methods
        public int getEmpSSN(){ return ssn; }
        public String getEmpName(){ return empName; }
        public int getEmpAge(){ return empAge; }
        public void setEmpAge(int newValue){ empAge = newValue; }
        public void setEmpName(String newValue){ empName = newValue; }
    public void setEmpSSN(int newValue){ ssn = newValue; }
}
```

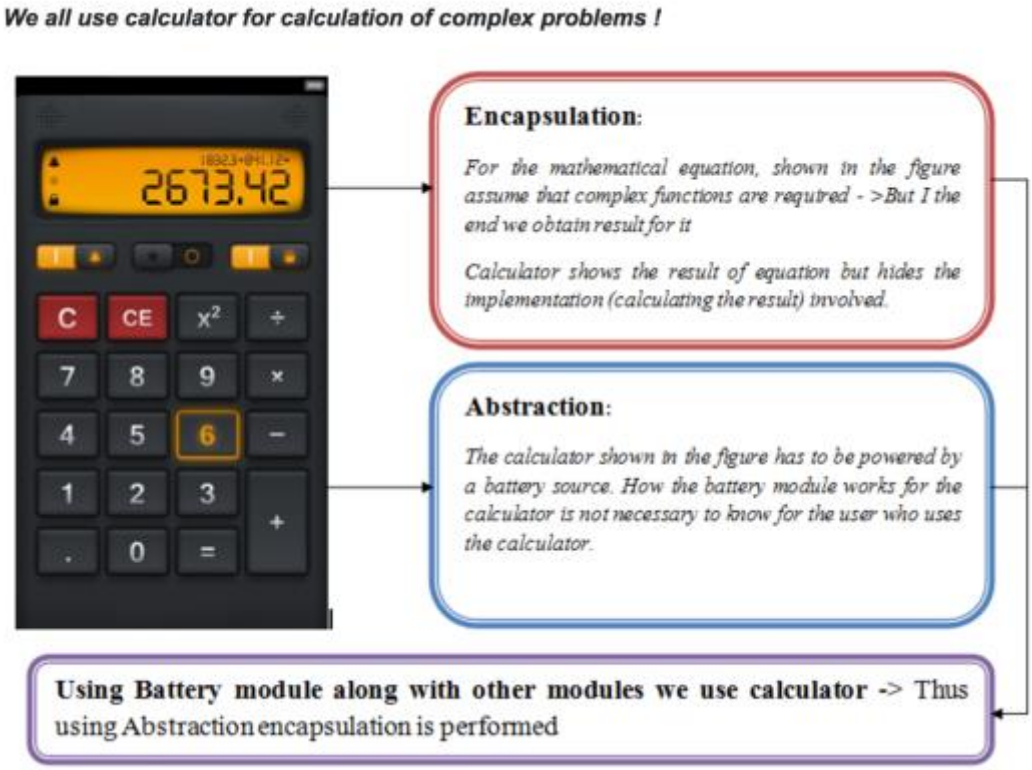
Access Modifiers

	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	n
no modifier	y	y	n	n
private	y	n	n	n

y: accessible
n: not accessible

Variables declared private can be accessed outside class, if public getter methods are present in class. Using private modifier is main way that an object encapsulates itself and hides data from the outside world.

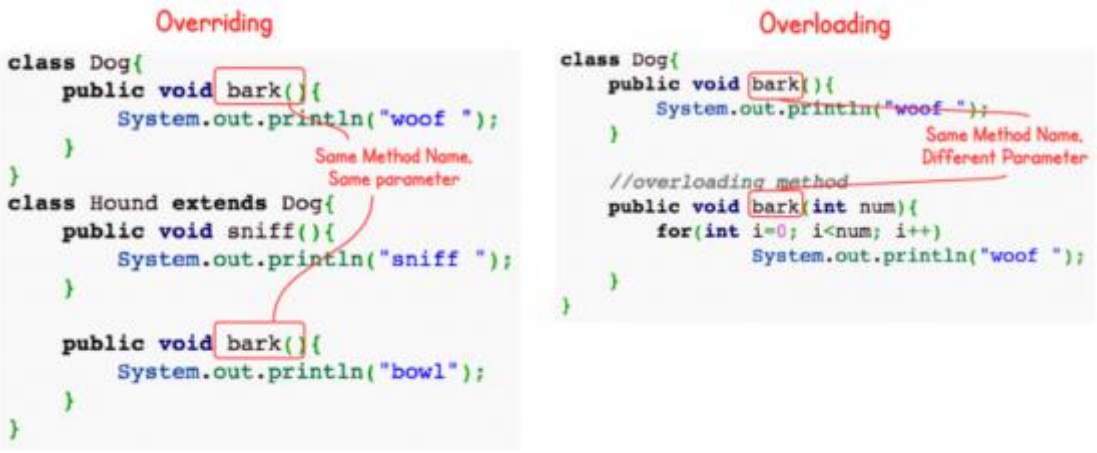
**Abstraction* – Process of hiding implementation details from user, only functionality will be provided to user. In that way, user will only have info on what object does instead of how it does it.



- More about abstract classes and methods in next page

Pros: makes code more readable and simple since complexity is hidden, easier to maintain code since program is very complex without abstraction

**Polymorphism* – Allows us to perform single action in different ways (to process objects differently based on their data type). Two types: *Compile time* polymorphism (static binding) and *Runtime polymorphism* (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism. Static polymorphism in Java is achieved by method overloading. Dynamic polymorphism in Java is achieved by method overriding. Overriding is run-time while overloading is compile-time.



* “super” keyword: used inside a sub-class method definition to call a method defined in the **super** class. Private methods of the **super**-class cannot be called. Only public and protected methods can be called by the **super keyword**. It is also used by class constructors to invoke constructors of its parent class.

```
class Vehicle {
    public void move () {
         System.out.println ("Vehicles are used for moving from one place
    }
}

class Car extends Vehicle {
    public void move () {
      super. move (); // invokes the super class method
      System.out.println ("Car is a good medium of transport ");
    }
}

public class TestCar {
    public static void main (String args []){
        Vehicle b = new Car (); // Vehicle reference but Car object
        b.move (); //Calls the method in Car class
    }
}
```

Output:

```
Vehicles are used for moving from one place to another
Car is a good medium of transport
```

Pros: **overloading** allows methods that perform similar functions to be accessed through a common name, **overriding** allows sub class to use all definitions that a super class provides and also can add more definitions through overridden methods, **overriding** works together w/ inheritance to allow for reusability without the need for re-compile

Abstract Classes and Methods + Interface

http://beginnersbook.com/2013/05/java-abstract-class-method/

A class that is declared using “abstract” keyword is known as abstract class. It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with abstract methods (without an implementation, without braces, and followed by a semicolon). An

abstract class cannot be **instantiated** (you are not allowed to create **object** of Abstract class).

Remember two rules:
1) If the class is having few abstract methods and few concrete methods: declare it as abstract class.
2) If the class is having only abstract methods: declare it as **interface**

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** (discussed below) in a class then you must declare the class abstract as well. you can't have abstract method in a **non-abstract class**. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. Abstract class can have non-abstract method (concrete) as well.

Points to remember about abstract method:
1) Abstract method has no body.
2) Always end the declaration with a **semicolon**(;).
3**) It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.**
4) Abstract method must be in an abstract class.

http://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/

Used for: to provide common definition of a base class that multiple derived classes can share.

*** Interface ***

http://beginnersbook.com/2013/05/java-interface/

What is an interface?
Interface looks like class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default.

What is the use of interfaces?
Used to achieve full abstraction (whereas Abstract Classes is partial). Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not support multiple inheritance, using interfaces we can achieve this as a class can implement more than one interfaces, however it cannot extend more than one classes.

Interface and Inheritance
Interface cannot implement another interface. It must extend the other interface if required. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf1 and Inf2.

http://beginnersbook.com/2014/01/abstract-method-with-examples-in-java/

**** When to use Abstract vs Interface?

An **abstract class's purpose** is to provide an appropriate superclass from which other classes can inherit and thus share a common design.
As opposed to an interface:

An **interface describes** a set of methods that can be called on an object, but **does not provide concrete implementations for all the methods**... Once a class implements an interface, all objects of that class have an is-a relationship with the interface type, and **all objects of the class are guaranteed to provide the functionality described by the interface.** This is true of all subclasses of that class as well.

Implement vs. Extends
extends is for when you're inheriting from a base class to extend its functionality
implements is for *implementing* an interface
The difference between an interface and a regular class is that in an interface you cannot implement any of the declared methods. Only the class that "implements" the interface can implement the methods.

Also, java doesn't support **multiple inheritance** for classes. This is solved by using multiple interfaces.

```
public interface ExampleInterface{
        public void do();
        public String doThis(int number);
}

public class sub implements ExampleInterface{
        public void do(){
                //specify what must happen
        }
        public String doThis(int number){
                //specfiy what must happen
```

```
        }
}
```

now extending a class

```
public class SuperClass{
        public int getNb(){
                //specify what must happen
                return 1;
        }
        public int getNb2(){
                //specify what must happen
                return 2;
        }
}

public class SubClass extends SuperClass{
        //you can override the implementation
        @Override
        public int getNb2(){
                return 3;
        }
}
```

in this case

```
Subclass s = new SubClass();
s.getNb(); //returns 1
s.getNb2(); //returns 3
SuperClass sup = new SuperClass();
sup.getNb(); //returns 1
sup.getNb2(); //returns 2
```