



Core Java

Object Oriented Programming in Java

Lesson Objective

- Introduction to Object Orientation
- Objects and Classes
- Object Oriented Principles
 - Abstraction
 - Encapsulation
 - Modularity
 - Hierarchy
 - Polymorphism
- Implementation of Object Oriented Principles
- Abstract Classes and Interfaces
- Inner Classes

Introduction to Object Orientation

- OOP is a paradigm of application development where programs are built around objects and their interactions with each other.
 - An Object Oriented program can be viewed as a collection of co-operating objects
- An OO program is made up of several objects that interact with each other to make up the application.
- **For example:** In a Banking System, there would be Customer objects pertaining to each customer. Each customer object would own its set of Account Objects, pertaining to the set of Savings and Current Accounts that the customer holds in the bank.
- Today, most programming languages are object oriented.
For example: Java, C++, C#

Object Oriented Approach

- OO Approach is based on the concept of building applications and programs from a collection of “reusable entities” called “objects”.
 - Each object is capable of receiving and processing data, and further sending it to other objects.
 - Objects represent real-world business entities, either physical, conceptual, or software.
- **For example:** a person, place, thing, event, concept, screen, or report

Why Object Oriented Programming?

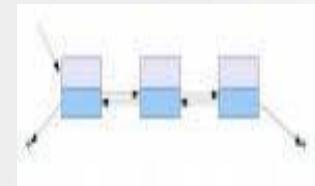
- There are problems associated with structured language, namely:
 - Emphasis is on doing things rather than on data
 - Most of the functions share global data which lead to their unauthorized access
 - More development time is required
 - Less reusability
 - Repetitive coding and debugging
 - Does not model real world well

Advantages with OOPS

- **Simplicity:** Software objects model the real world objects. Hence the complexity is reduced and the program structure is very clear.
- **Modularity:** Each object forms a “separate entity” whose internal workings are decoupled from other parts of the system.
- **Modifiability:** Changes inside a class do not affect any other part of a program, since the only “public interface” that the external world has to a class is through the use of “methods”.
- **Extensibility:** Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
- **Maintainability:** Objects can be separately maintained, thus making locating and fixing problems easier.
- **Re-usability:** Objects can be reused in different programs.

Objects

- An object is a tangible, intangible, or software entity.
- It is characterized by Identity, State, and Behavior.
 - **Identity:** It distinguishes one object from another.
 - **State:** It comprises set of properties of an object, along with its values.
 - **Behavior:** It is the manner in which an object acts and reacts to requests received from other objects



Object State

- **State** of an object is one of the possible conditions in which the object may exist.



Bank Account Modeled as a Software Object

Attributes of the object:

Account Number: A10056

Type: Savings

Balance: 40000

Customer Name: Sarita Kale

The state of an object is not defined by a “state” attribute or a set of attributes. Instead the “state” of an object gets defined as a total of all the attributes and links of that object.

- **Behavior** of an object determines how an object reacts to other objects.



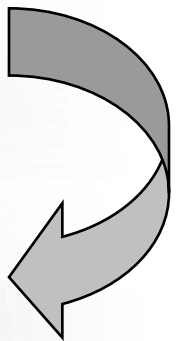
Behaviors of the object

- Withdraw
- Deposit
- Check Balance
- Get Monthly statement



These are the operations that the object can perform, and represents its behavior.

Through these operations or methods, an object controls its state.



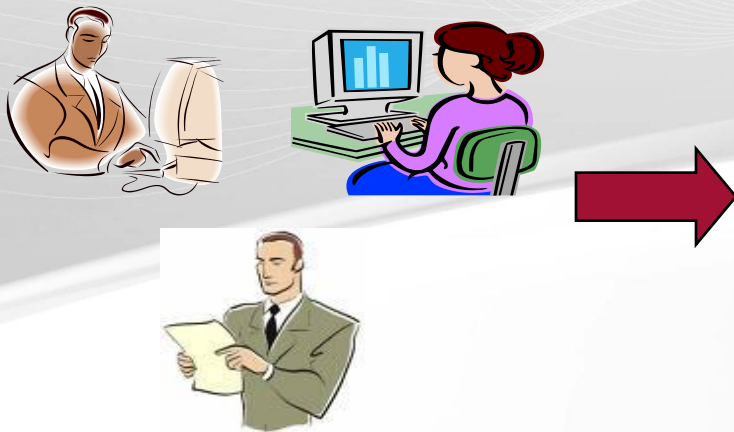
Object Identity

- Two objects can possess identical attributes (state) and yet have distinct identities.



Class

- A Class characterizes common structure and behavior of a set of objects.
- It constitutes of Attributes and Operations.
- It serves as a template from which objects are created in an application.



Class name	Customer
Class attributes	Name, Address, Email-ID, TelNumber
Class operations	displayCustomerDetails() changeContactDetails()

Class Attributes and Operations

- Each class has a name, attributes, and operations.
 - **Attribute** is a named property of a class that describes a range of values.
 - **Operation** is the implementation of a service that can be requested from any object of the class to affect behavior.

	Class Name	Account
Attributes {	Class attributes	AccountNumber, Type, Balance
Operations {	Class operations	withdraw(), checkBalance(), displayAccountDetails, deposit()

- **Constructors:**

- They enable instantiation of objects against defined classes.
- Memory is set aside for the object. Attribute values can be initialized along with object creation (if needed).

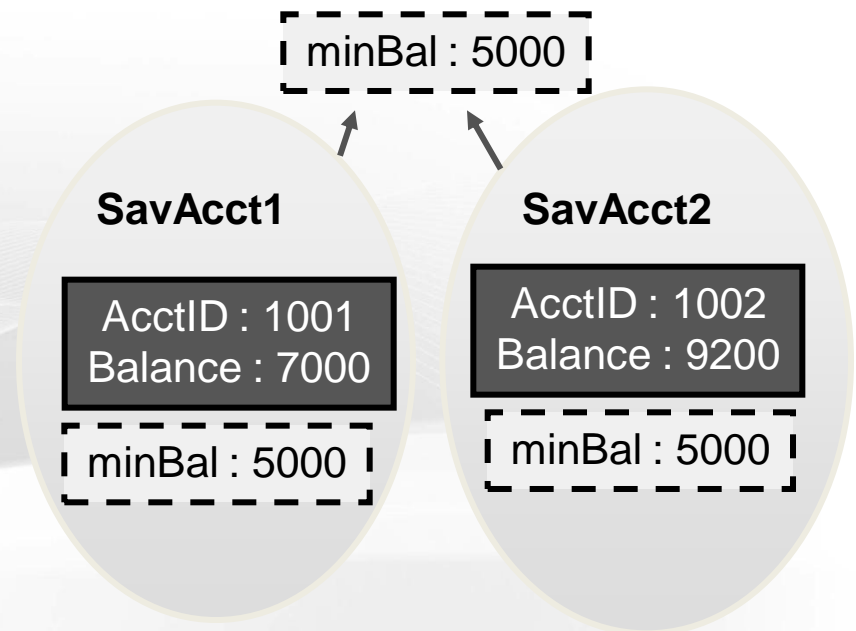
- **Destructors:**

- They come into play at end of object lifetime. They release memory and other system locks.

Static Members

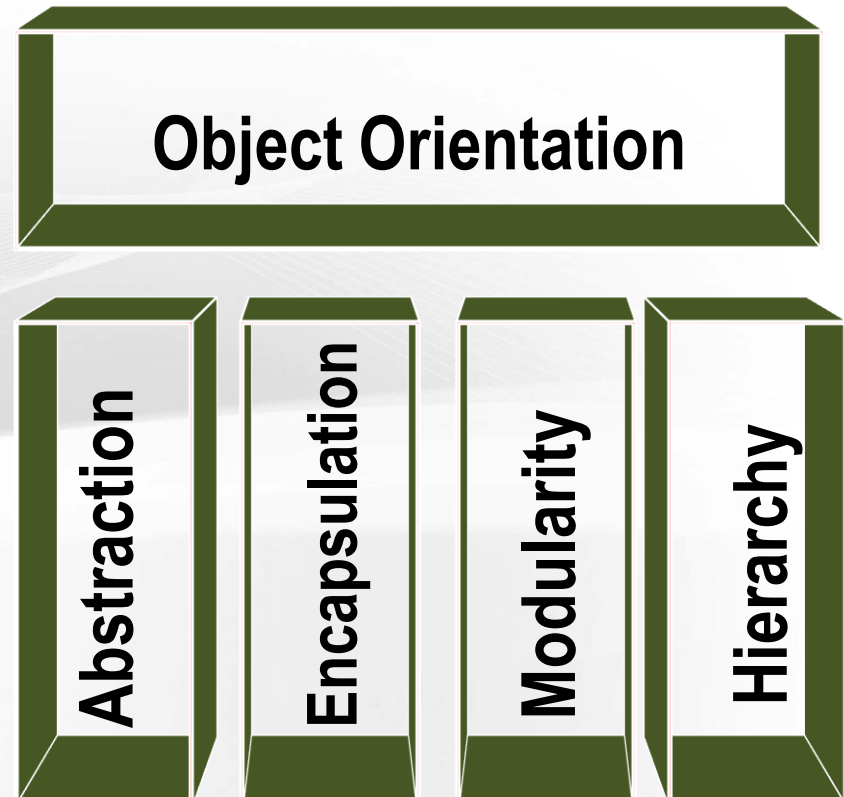
- Static data members are those that are shared amongst all object instances of the given type.

➤ **Example:** Rate of Interest for Savings Account will be the same in the bank, so common copy is sufficient. Not so for Account Balances!



- Static Member Functions can be invoked without an object instance.
 - **For example:** Counting the number of Customer Objects created in the Banking System - this is not specific to one object!

- OO is based on four basic principles, namely:
 - Principle 1: Abstraction
 - Principle 2: Encapsulation
 - Principle 3: Modularity
 - Principle 4: Hierarchy

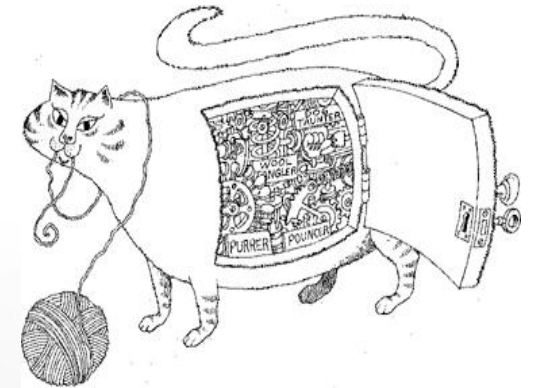


Concept of Abstraction

- Focus only on the essentials, and only on those aspects needed in the given context.
 - **For example:** Customer Height / Weight not needed for Banking System!
- Abstraction is determining the essential qualities. By emphasizing on the important characteristics and ignoring the non-important ones, one can reduce and factor out those details that are not essential, resulting in less complex view of the system.
- Abstraction means that we look at the external behavior without bothering about internal details. We do not need to become car mechanics to drive a car!

Concept of Encapsulation

- “To Hide” details of structure and implementation
 - **For example:** It does not matter what algorithm is implemented internally so that the customer gets to view Account status in Sorted Order of Account Number.
- Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object.
- Encapsulation allows restriction of access of internal data.




Encapsulation versus Abstraction

- Abstraction and Encapsulation are closely related.
 - Abstraction can be considered as User's perspective.
 - Encapsulation can be considered as Implementer's perspective.
- Why Abstraction and Encapsulation?
 - They result in “Less Complex” views of the System.
 - Effective separation of inside and outside views leads to more flexible and maintainable systems.



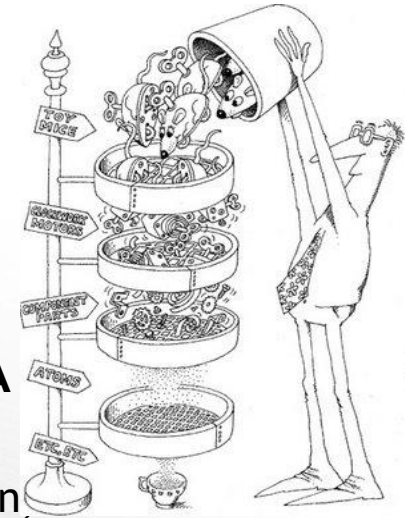
Concept of Modularity

- Decomposing a system into smaller, more manageable parts
 - **Example:** Banking System can have different modules to take care of Customer Management, Account Transactions, and so on.
 - Why Modularity?
 - Divide and Rule! Easier to understand and manage complex systems.
 - Allows independent design and development of modules.
 - As modules are groups of related classes, it is possible to have parallel developments of modules. Changes in one may not affect the other modules. Modularity is an essential characteristic of all complex systems. Well designed modules can be reused in similar situations in other designs.
- 



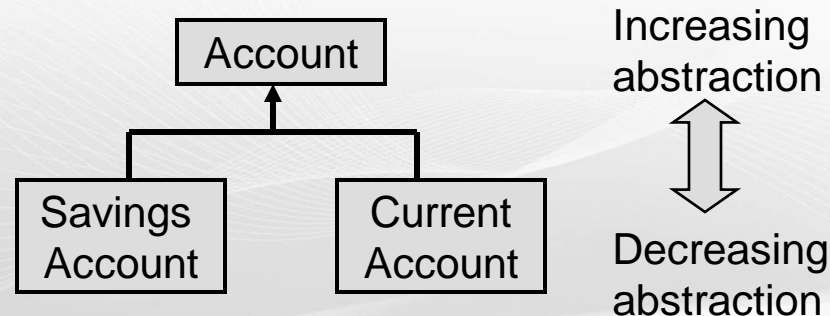
Concept of Hierarchy

- A ranking or ordering of abstractions on the basis of their complexity and responsibility
- It is of two types:
 - **Class Hierarchy:** Hierarchy of classes, Is A Relationship.
 - Example: Accounts Hierarchy
 - **Object Hierarchy:** Containment amongst Objects, Has A Relationship.
 - Example: Window has a Form seeking customer information which has text boxes and various buttons.



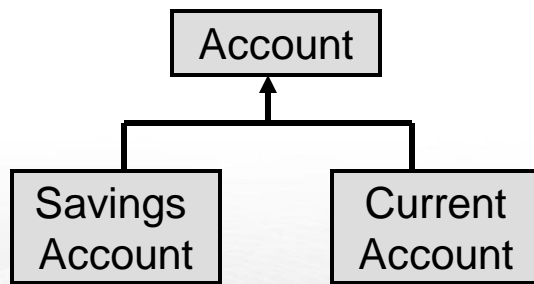
● Inheritance

- It is a powerful technique that enables code reuse resulting in increased productivity, and reduced development time.
- It allows for designing extensible software.

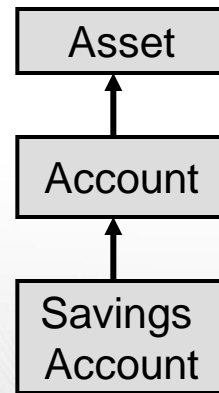


- Inheritance is the process of creating new classes, called *derived classes*, from existing or base classes. The derived class inherits all the capabilities of the base class, but can add some specificity of its own. The base class is unchanged by this process.

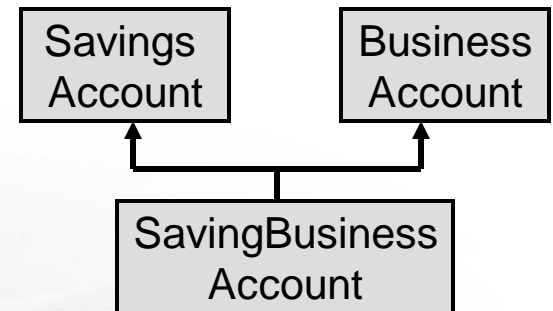
Types of Inheritance



Single-level inheritance



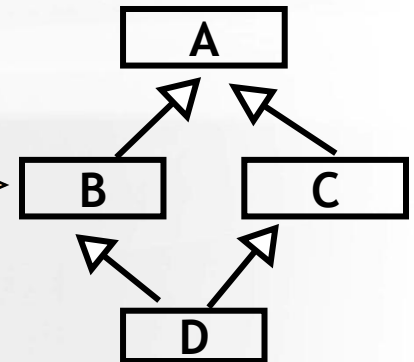
Multilevel inheritance



Multiple inheritance

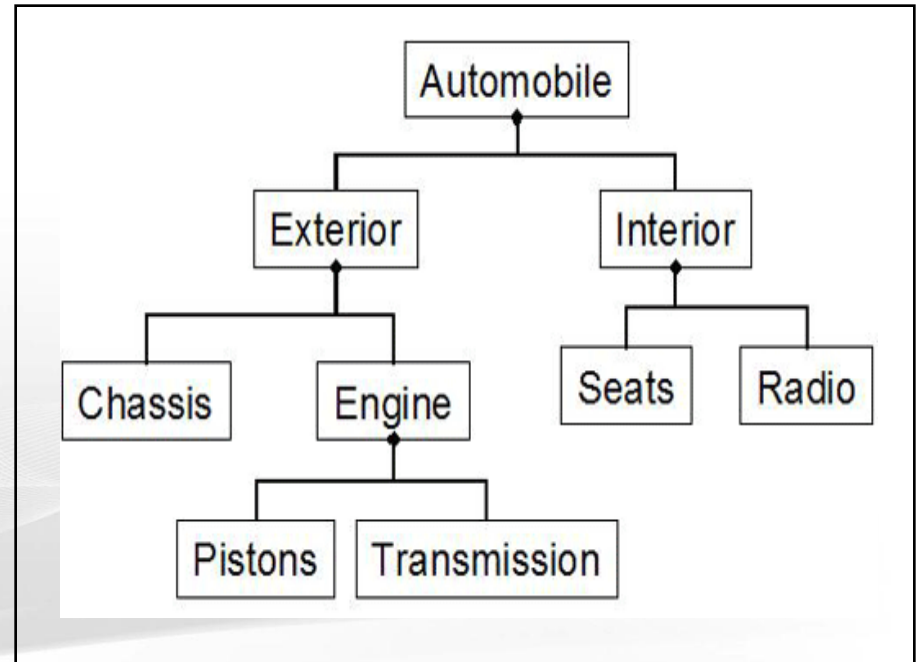
Multiple inheritance challenges:

A name conflict introduced by a shared super-class (A) of super-classes (B and C) used with "multiple inheritance".



Object Hierarchy

- Composition
- “Has-a” hierarchy is a relationship where one object “belongs” to (is a part or member of) another object, and behaves according to the rules of ownership.



- It implies One Name, Many Forms.
- It is the ability to hide multiple implementations behind a single interface.
- Polymorphism allows different objects to respond to the same message in different ways!
- There are two types of Polymorphism, namely:
 - Static Polymorphism
 - Dynamic Polymorphism

- Static Polymorphism:

- The “Form” can be resolved at compile time, achieved through **overloading**.

- For example: An operation “Sort” can be used for sorting integers, floats, doubles, or strings.

- Dynamic Polymorphism:

- The “Form” can be resolved at run time, achieved through **overriding**.

- For example: An operation “calculateInterest” for Accounts, where an Account can be of different types like Current or Savings.

Class and Object Implementation

```
public class Employee {  
    private String name; // instance variables  
    private String tel;  
  
    public Employee(String n, String t){ //constructor  
        name = n;  
        tel = t;  
    }  
    public void printinfo() {  
        System.out.println(name+"'s number is "+tel);  
    }  
  
    public static void main(String[] args) // Entry Point  
    {  
        //Object Creation  
        Employee gao=new Employee("Gao", "4548");  
        gao.printinfo();  
    }  
}
```

Inheritance Implementation

```
public class Faculty extends Employee {
    private String secName;
    public Faculty(String n, String t, String s) {
        super(n, t);           //have to be the first line  constructor
        secName = s;
    }
    public void changeSec (String s) {
        secName = s;
    }
    public void printinfo() {
        super.printinfo();
        System.out.println("Secretary's name is " + secName);
    }
    public static void main(String[] args) {
        Faculty lixin=new Faculty("Lixin", "4548", "June");
        lixin.printinfo();
    }
}
```

- Two or more methods within the same class share the *same* name.
- They should differ either in number of arguments or types of arguments.
- Java implements compile-time polymorphism by:
 - Overloading Constructors
 - Overloading Normal Methods

Function Overloading

```
public class Calculator {  
  
    public int add(int x, int y){  
        return(x+y);  
    }  
    public int add(int x,int y, int z) {  
        return(x+y+z);  
    }  
  
    public static void main(String[] args)  
    {  
        Calculator c=new Calculator();  
        int d=c.add(10,20);  
        int e=c.add(10,20,30);  
    }  
}
```

- In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method.
- Overridden methods allow Java to support run-time polymorphism.
- Base class and derived class method has same signature.
- Final methods can't be overridden

Function Overriding

```
public class Calculator_New extends Calculator {  
  
    public int add(int x, int y){  
        x++;  
        y++;  
        return(x+y);  
    }  
    public static void main(String[] args)  
    {  
        Calculator c;  
        c=new Calculator_New();  
        int d=c.add(10,20);           //32  
        c=new Calculator();  
        int e=c.add(10,20);           //30  
    }  
}
```


Abstract Classes

- Abstract Method

- Methods do not have implementation.
- Methods declared in interfaces are always abstract.
 - Example:

```
abstract int add(int a, int b);
```

- Abstract class

- Provides common behavior across a set of subclasses.
- Not designed to have instances that work.
- One or more methods are declared but may not be defined
- Advantages:
 - Code reusability.
 - Help at places where implementation is not available.

Abstract Classes

- Declare any class with even one method as abstract as *abstract*.
- Cannot be instantiated.
- Use *Abstract* modifier for:
 - Constructors
 - Static methods
- Abstract class' subclasses should implement all methods or declare themselves as *abstract*.
- Can have concrete methods also.

Abstract classes

```
class AccountAbstract {  
    public int balance;  
    abstract public void deposit(int amt);           //abstract method  
    public void display()  
    {    System.out.println("Balance = "+balance);  
    }  
}  
  
public class Saving extends AccountAbstract {  
    public void deposit(int amt){  
        balance=balance +amt;  
    }  
    public static void main(String[] args)  
    {  
        Saving s=new Saving();  
        s.deposit(500)  
    }  
}
```

Interface

- Interface defines a data-type without implementation.
- The interface approach is sometimes known as programming by contract.
- It's essentially a collection of constants and abstract methods.
- An interface is used via the keyword “implements”. Thus, a class can be declared as follows:

```
class MyClass implements MyInterface{  
    ... }
```

Interfaces

- A Java interface definition looks like a class definition that has only abstract methods, although the abstract keyword need not appear in the definition

```
public interface Testable {  
    void method1();  
    void method2(int i, String s);  
    int x=10;  
}
```

note no bodies for the methods, public by default

Static final variable

Declaring and Using Interfaces

```
public interface simple_cal {  
    int add(int a, int b);  
    int i=10;  
}  
//Interfaces are to be implemented.  
class calci implements simple_cal {  
    int add(int a, int b){  
        return a+b;  
    }  
}
```

- Methods in an interface are always public and abstract.
- Data members in a interface are always public, static and final.
- Interface is a purely abstract class; has only signatures and no implementation.
- Interface members are implicitly public abstract.
- Interface members must not be static.
- Interfaces can extend other interfaces.
- A class can inherit from a single base class, but can implement multiple interfaces.

Interfaces and Abstract Classes

Abstract classes	Interfaces
Abstract classes are used only when there is a “is-a” type of relationship between the classes.	Interfaces can be implemented by classes that are not related to one another.
You cannot extend more than one abstract class.	You can extend more than one interface.
Abstract class can contain abstract as well as implemented methods.	Interfaces contain only abstract methods.
With abstract classes, you grab away each class’s individuality.	With Interfaces, you merely extend each class’s functionality.

Nested Classes

- Class within another class.
- Scope is bounded by the scope of the enclosing class.
- Use to reflect and enforce the relationship between two classes.

```
class EnclosingClass{  
    ...  
    class ANestedClass {           //known only within Enclosing class  
        ...  
    }  
}
```

- Nested classes are of two types:
 - Static:
 - Access members of the enclosing class through an object.
 - Cannot refer to members of the enclosing class directly.
 - Non-static:
 - Also called as *Inner* classes.
 - Access to all members, including private members, of the class in which they are nested.
 - Can refer to the members of enclosing class in the same way as non-static members.

Inner class Example

```
class Outer{
int intOuter = 100;
void Test(){
    Inner inObj = new Inner();
    inObj.display();
}
class Inner{
    void Display(){
        System.out.println("Value of outer variable = " + intOuter);
    } } // end of class Inner
} // end of class Outer
class Demo{
public static void main(String args[]){
    Outer outObj = new Outer();
    outObj.test();
}
}
```

Output: Value of outer variable =100

Anonymous Inner Class

- Do not have a name.
- Defined at the location they are instantiated using additional syntax with the *new* operator.
- Used to create objects “on the fly” in contexts such as:
 - Method return value.
 - Argument in a method call.
 - Variable initialization.



Client.java

Final Variable

- Variable declared as *final* acts as a constant.
- Once initialized, it's value cannot be changed.
- Example:
 - `final int i = 10;`

- Method declared as final cannot be overridden in subclasses.
- Their values cannot change their value once initialized.
 - Example:

```
class A {  
    public final int add (int a, int b)  
    {  
        return a+b;  
    }  
}
```

- Cannot be sub-classed at all.
- Examples: *String* and *StringBuffer* classes in Java.
- A class or method cannot be abstract and final at the same time.

```
final class A {  
    public int add (int a, int b)  
    {  
        return a+b;  
    }  
}
```

- Packages are used to group related classes and interfaces.
- Packages are useful method of grouping classes to avoid name clashes.
- Classes with same name can be put into different packages.
- A *package* is a collection of related types providing access protection.

Packages

- You should bundle these classes and the interface in a package for several reasons:
- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that provide graphics-related functions.
- The names of your types won't conflict with types names in other packages, because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating package

- Create the class or interface.
- Put the first statement of that class or interface as below.

```
package <package_name>;
```

Example : `package pack1;`
- compile the above class or interface which will generate .class file.
- Create folder with the package name and put the class file in the folder.
- If you are having sub packages the corresponding sub folders need to be created

Create a package

```
package pkg1;  
public class Student  
{  
    public int sno;  
    public void getSno()  
    {  
    }  
}
```

Using Packages

- Write the import statement in the file where you want use the package classes or interfaces.

Example: `import pkg1.Student;`

- Set the classpath from command prompt to get package available to you.

Example :

Set classpath=.;d:\assignments;

Assuming that pkg1 folder is in side the “d:\assignment” folder

- Running application whose main class is in package
`javac pkg1.Student;`

```
import pkg1.*;

class MainClass
{
    public static void main(String a[])
    {
        Student s1=new Student();
        s1.getStudent();
    }
}
```

Packages and Name Space Collision

package pack1;

class Teacher

class Student

package pack2;

class Student

class Courses

- Namespace collision can be avoided by accessing classes with the same name in multiple packages by their fully qualified name.

Commonly used Java Packages

Package Name	Description
java.lang	Classes that apply to the language itself, which includes the Object class, the String class, and the System class. It also contains the special classes for the primitive types (Integer, Character, Float, and so on). Classes belonging to java.lang package need not be explicitly imported.
java.util	Utility classes, such as Date, as well as collection classes, such as Vector and Hashtable
java.io	Input and output classes for writing to and reading from streams (such as standard input and output) and for handling files
java.net	Classes for networking support, including Socket and URL (a class to represent references to documents on the World Wide Web)
java.applet	Classes to implement Java applets, including the Applet class itself, as well as the AudioClip interface

Static Import

- Static import enables programmers to import static members.
- Class name and a dot (.) are not required to use an imported static member.

```
import static java.lang.Math.*;  
public class StaticImportTest  
{  
    public static void main( String args[] )  
    {  
        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );  
    } // end main  
}
```

Note: It's not Math.sqrt



Access Specifiers and Modifiers

	Private	No Modifier	Protected	Public
Same class	Y	Y	Y	Y
Same Package Subclass	N	Y	Y	Y
Same Package non-sub class	N	Y	Y	Y
Different Package Subclass	N	N	Y	Y
Diffrent Package non-subclass	N	N	N	Y