Session 7 – OOP Concepts

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Class & Objects
- ✓ Why OOP?
- ✓ OOP vs. Structured
- ✓ OOP Features

OOP Concepts

Introduction to OOPs Concepts

Java is designed around the principles of object-oriented programming.

What is Object-Oriented Programming?

To put it simply, object-oriented programming focuses on data before anything else. How data is modeled and manipulated through the use of objects is fundamental to any object-oriented program.

OOP is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

To perform object-oriented programming, one needs an object-oriented programming language (OOPL). Java, C++ languages are one of the major popular OOP languages.

Let us understand what Class & Objects are-

Class –

A "**Class**" can be understood in the following way:

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors.

(or)

A class is a concept which has definitions of data and definitions of functionality.

(or)

A class is collection of data members and member functions.

(or)

Class defines the encapsulation of an object. Class contains only definitions.

Classes are understood as –

- A class is defined as a collection of Objects with same type of data and functions
- Once a class is defined we can create number of objects belonging to that class
- Class is a user defined data type

Object

Objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. A software object exists in the memory, so it needs memory allocation for its existence.

Object is understood as –

- Object is defined as the basic run time entity that contains data and its related methods.
- Object may be a person, place or any item that the program has to handle.
- The methods are operating on the data.
- The objects may be either physical or logical.

Why OOP?

Object-oriented programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. It is based on several techniques, including encapsulation, modularity, polymorphism, and inheritance.

Why OOPs is used for programming?

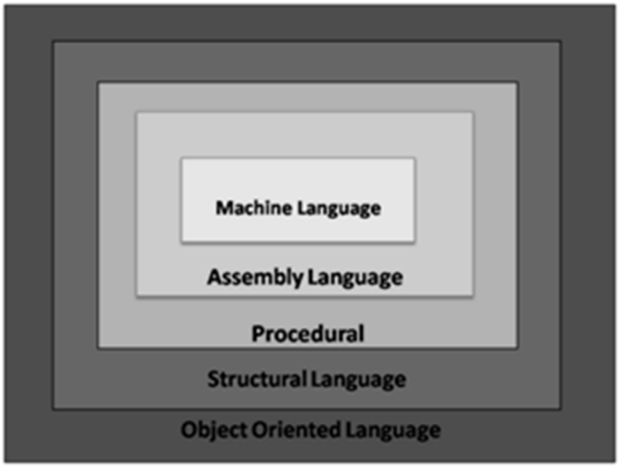
1. OOPs are really flexible in terms of using implementations.
2. It is much easier in implementing security.
3. It makes the coding more organized - We all know that a Clean Program is a Clean Coding. Using OOP instead of procedural makes things more organized and systematized.

The concepts used in OOPs are

- ✓ Objects
- ✓ Classes
- ✓ Data Abstraction and Data Encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic Binding

The paradigms of Programming Language give the model to the programmer to write the programs. The different paradigms of the programming language are as bulleted below:-

- ✓ Unstructured Programming (or) Monolithic Programming
- ✓ Procedural Programming
- ✓ Structural Programming
- ✓ Object Oriented Programming

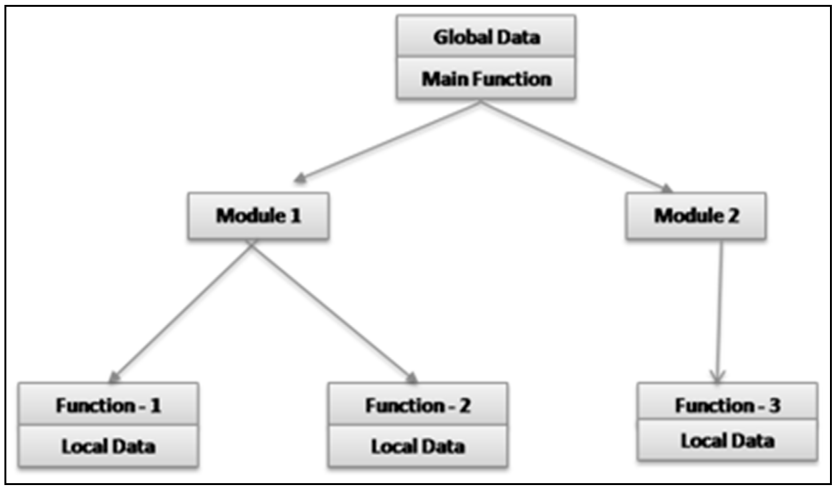


Let us understand about the difference between Structured Programming & Object Oriented Programming;

Structural Programming:

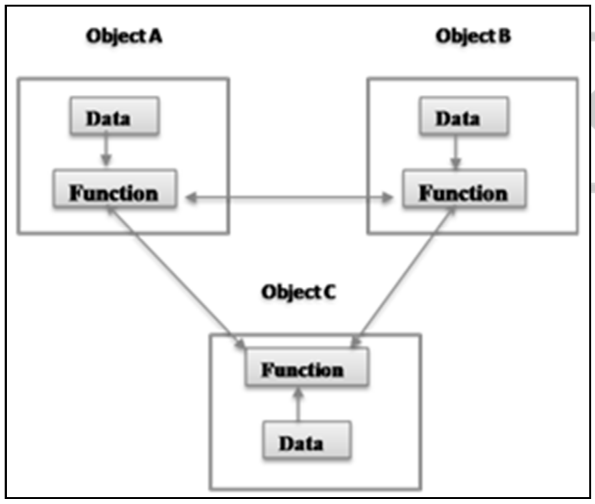
- ✓ The program is divided into modules and the modules are then divided into functions.

- ✓ The usage of goto statement is removed or reduced.
- ✓ Each module can work independent of one another.
  - Example: ADA, C++.



Object Oriented Programming:

- ✓ The Program is divided into number of small units called Object. The data and function are built around these objects.
- ✓ The data of the objects can be accessed only by the functions associated with that object.
- ✓ The functions of one object can access the functions of other object.



Difference between Structured and Object Oriented Programming

Structured Programming	Object-oriented Programming
Top-down approach is followed.	Bottom-up approach is followed.
Focus is on algorithm and control flow.	Focus is on object model.
Program is divided into a number of sub modules or functions or procedures	Program is organized by having a number of classes and objects.
Functions are independent of each other.	Each class is related in a hierarchical manner.
No designated receiver in the function call.	There is a designated receiver for each message passing.
Views data and functions as two separate entities.	Views data and functions as a single entity.
Maintenance is costly.	Maintenance is relatively cheaper.
Software reuse is not possible.	Helps in software reuse.
Function call is used.	Message passing is used.
Function abstraction is used.	Data abstraction is used.
Algorithm is given importance.	Data is given importance.

Solution is solution-domain specific.	Solution is problem-domain specific.
No encapsulation. Data and functions are separate.	Encapsulation packages code and data altogether. Data and functionalities are put together in a single entity.

OOP Features:

- ✓ Emphasis is given on data rather than procedures.
- ✓ Problems are divided into objects.
- ✓ Data structures are designed such that they organize the object.
- ✓ Data and function are tied together.
- ✓ Data hiding is possible.
- ✓ New data and functions can be easily loaded.
- ✓ Object can communicate with each other using functions.
- ✓ Bottom-up programming approach is used

Session 8 – OOP Concepts Contd...

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Encapsulation
- ✓ Data Abstraction
- ✓ Writing Classes
- ✓ Creating Objects
- ✓ Constructor Overloading

OOP Concepts

Encapsulation

Encapsulation is the mechanism that binds together code (member functions) and the data (data members) it manipulates, and keeps both safe from outside interference and misuse. It means it bundles the data with the methods that operate on it as a set, that we call as object. Programs written in non-object-oriented languages did not enforce any property/method relationship. This often resulted in side effects where variables had their contents changed or reused in unexpected ways and that was difficult to unravel, understand and maintain. Encapsulation is one of three fundamental principles in object oriented programming.

Data hiding

Data hiding is the ability of objects to shield variables from external access. It is a useful consequence of the encapsulation principle. Those variables marked as private can only be seen or modified through the use of public accessor and mutator methods. This permits validity checking at run time. Access to other variables can be allowed but with tight control on how it is done.

Abstraction

Abstraction is the process of recognizing and focusing on important characteristics of an object and leaving/filtering out the un-wanted characteristics of the object. Let us take a person as example and see how that person is abstracted in various situations.

- A doctor sees (abstracts) the person as patient. The doctor is interested in name, height, weight, age, blood group, previous or existing diseases etc. of a person
- An employer sees (abstracts) a person as Employee. The employer is interested in name, age, health, degree of study, work experience etc. of a person.
- A Banker sees (abstracts) a person as Customer. The banker is interested in name, age, pan card no etc. of a person.

So, we see that Abstraction is the basis for software development. It is through abstraction we define the essential aspects of a system. The process of identifying the abstractions for a given system is called as Modeling (or object modeling).Abstraction serves as the foundation for determining the classes for a particular system (which is called object model).

Moreover, Data Abstraction

- ✓ Refers to the act of representing essential features without including the back ground details.
- ✓ Also by the feature of data abstraction it is possible to create user defined data types and thus increase the power of programming language

Now let us start writing of Classes and see how we write a simple Java Class

Writing the Class

We may write the following in a class

- ✓ Data Members
- ✓ Constructors
- ✓ Destructors
- ✓ Accessors
- ✓ Methods

Data Members

Data Member is a variable declared in a class body that holds the object's state(instance members). We also have other type of data members "static data member" which holds the class's state.

Constructors

A java constructor has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value. Constructor may have parameters.
Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided. It makes the initialization of each member with the default value of the corresponding data type.

Destructors

Unlike in C++ we don't need to write the destructors to free the memory associated with the objects after its usage. Java has a feature "Automatic Garbage Collection" which will free the memory automatically after the object scope is completed. However if we have some other resource to free before our object is removed we can implement finalize() method of "Object" class. This will be discussed in more detail in "Memory Management" Chapter.

Accessors

Encapsulation and information hiding are fundamental principles that lead to the design of robust classes and components. An important technique that we can employ to increase encapsulation, and to support information hiding, is the appropriate use of accessor (getter &mutator) methods.

Accessors - member functions that directly manipulate the value of fields -- come in two flavors: setters or mutators and getters.
A setter modifies the value of a field, whereas a getter obtains its value. Although accessors add minimal overhead to your code, the loss in performance is often trivial compared to other factors (such as questionable database designs).
Accessors help to hide the implementation details of your classes and, thus, increase the robustness of your code. By having, at most, two control points from which a field is accessed, one setter and one getter, you are able to increase the maintainability of your classes by minimizing the points at which changes need to be made.

Methods

Methods are functions written in class body that represents the behaviors of the class.
Methods receive the input through the arguments. There are two types of methods supported -"instance methods" and "static methods".

Let us write a simple example and see how a class looks like;

Simple Example of Class:
//Account.java

```
class Account{
protectedintano;
protected String title;
protected double balance;
public Account(inta,Stringt,double b){
        ano=a;
        title=t;
        balance=b;
   }
publicintgetAno(){
        returnano;
   }
public void setTitle(String t){
        title=t;
   }
public String getTitle(){
        return title;
   }
public double getBalance(){
        return balance;
   }
public void deposit(double amt){
        balance+=amt;
        System.out.println("Balance after deposit is "+balance);
   }
public void withdraw(double amt){
        if ((balance-amt)>=0){
        balance-=amt;
        System.out.println("Balance after withdrawl is "+balance);
        }else
        System.out.println("Sorry insufficient funds");
    }
}
```

Creating Objects

A class defines what data members an object will have when it's created. When new object is created for a class, a block of memory is allocated in the heap and it is referenced from a stack variable.

"new" keyword

To create a new object, we use "new" followed by name of the class (ex:Account), followed by parenthesis (This represents the constructor arguments). The new keyword creates the memory in the heap.

Example class to test the functionality of the Account class:

```
classTestAccount{
public static void main(String args[]){
        Account a=new Account(101,"krsna",50000);
        a.deposit(5000);
        a.withdraw(1500);
        a.withdraw(3500000);
   }
```

Another Example of class:

```
Employee.java
class Employee{
privateintempNo;
private String name;
private double basic;
private String panNo;
public Employee(inte,Stringn,doubleb,String p){
        empNo=e;
        name=n;
        basic=b;
        panNo=p;
  }
public Employee(inte,Stringn,double b){
        this(e,n,b,null);
  }
publicintgetEmpNo(){
        returnempNo;
  }
public void setName(String n){
        name=n;
  }
public double getBasic(){
        return basic;
  }
public void setPanNo(String p){
        panNo=p;
  }
public String getPanNo(){
        returnpanNo;
  }
```

```
private  doublecalDA(){
        return basic*0.1;
  }
private double calHRA(){
        return basic *0.15;
  }
private double calTA(){
        return basic *.12;
  }
private double calProfessionalTax(){
        return basic * 0.05;
  }
public void printPaySlip(){
        double total;
        total=basic+calDA()+calTA()+calHRA()-calProfessionalTax();
        System.out.println("Empno        :"+empNo);
        System.out.println("Emp Name   :"+name);
        System.out.println("Basic Sal     :"+basic);
        System.out.println("DA             :"+calDA());
        System.out.println("HRA            :"+calHRA());
        System.out.println("Total Sal      :"+total);
  }  public static void main(String args[]){
        inteno=Integer.parseInt(args[0]);
        Double bsc=Double.parseDouble(args[2]);
        Employee e=new  Employee(eno,args[1],bsc);
        e.printPaySlip();
  }
}
```


Unlike in previous example here we have written the main method in the Employee class itself. You can execute this program as below:
>java Employee 7499 "Krsna Prasad" 5400

Constructor Overloading

Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists. Constructor Overloading gives the flexibility to create objects of the class with variety of input combinations.

```
//Account.java
class Account{
privateintano;
private String title;
private double balance;
public Account(inta,Stringt,double b){
        ano=a;
        title=t;
        balance=b;
    }
    /*
```

This overloaded constructor takes only two arguments. when no initial balance is given we assume minimum 5000 as balance and initializes the balance with 5000. And instead of writing the initialization code again we can make call to

```
the other constructor by using "this(...)" keyword
    */
public Account(inta,String t){
        this(a,t,5000); // calls the above constructor.
    }
     /*
This overloaded constructor takes only one argument. When no title is given
we assume null for it.
    */
public Account(int a){
        this(a,"",5000); // calls the above constructor. }}
```

We can create the objects of the Account class in the following way

```
classTestAccount{
public static void main(String args[]){
        Account a1,a2,a3;
        a1=new Account(101,"nimai",100000);
        a2=new Account(102,13000);
        a3=new Account(103);
   }
}
```

Session 9 – Class Internals

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Instance & Static Members
- ✓ Instance & Static Methods
- ✓ Static blocks
- ✓ Overloading of methods
- ✓ Passing Objects as Arguments

Class Internals

Instance members vs. Static Members

Instance members are specific to instance or object and will come to existence only when instance is created for the class. That means until object is created no instance member is created in the memory.

Unlike instance members static members gets created and initialized as and when the class is loaded into memory irrespective of object creation. Static members are specific to the class.

Instance members will have separate memory copy for each instance of the class and any modification to an instance member of an instance will not show any reflection on other instances.

Static members will have only one copy for the class and entire instances so any modification made the static member that will be reflected to all.

Instance members only can be referred on instances or objects and cannot be referred on class directly. Static members can be referred on class directly. However static members also can be referred on objects in java.

Instance Methods vs. Static Methods

Instance methods represent a behavior of the class and its implementation. Usually it works with encapsulated content. Static methods though defined in class it represents individual behavior and doesn't work with encapsulated content. Mostly it takes inputs as arguments.

Instance Methods can be referred on objects and cannot be referred on class. Static methods can be referred on class however also can be referred on object.

Instance methods can refer both the instance and static members.

Static methods can refer only static members and cannot refer the instance members.

Static block / Static Initializer block

Static initializer blocks are usually used to initialize the static variables when the class is loaded. The code in the static block can also be quite complex. It resembles a method with no name, no arguments, and no return type.

Like a constructor, a static initializer block cannot contain a return statement.

You may include any number of static initializer blocks in your class definition, and they can be separated by other code such as method definitions and constructors. The static initializer blocks will be executed in the order in which they appear in the code.

Example Code block:

```
class Account{
  ......
static double floatInterestRate;
  .......
static{
floatInterestRate=findTodaysInterestRate();
 }
 ......
 // a static method calculates the todays float interest.
static double findTodaysInterestRate(){
  .....
 }
 .........}
```


"this" pointer (An implicit argument)

"this" pointer is an immutable reference which refers to the current object."this" can be used in instance methods to refer to the object on which the currently executing method has been invoked. "this" pointer is dropped to all instance methods as implicit argument.

"this" pointer will be useful in name overloading case for example if parameters of a method and members of the class shares the same name this can be used to refer the members distinctly. Since all instance methods are virtual in Java, this can never be null.

Example Code block:

```
class Account{
privateintano;
private String title;
private double balance;
public Account(....){
   ...
  }
  /*
herethis.balance refers to the balance of the object on which the method
deposit is invoked. Even you refer the balance directly without "this" pointer
by default it refers on this pointer only.
  */
public void deposit(double amt){
this.balance+=amt; //here balance+=amt  also will do the same thing.
System.out.println("Balance after deposit is "+this.balance);
  }
  ....
}
```

Method Overloading

Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call.

//MethodOverloadingDemo.java

```
public class MethodOverloadingDemo {
void adding() { // First Version
System.out.println("No parameters");
}
void adding(String a,String b) { // Second Version
```

```
        String c;
        c=a+b;//+ operators b/w strings concatenates.
        System.out.println("Two String Parameters concatenation: " + a);
        }
        int adding(int a, int b) { // Third Version
                System.out.println("Two parameters: " + a + " , " + b);
                return a + b;
        }double adding(double a, double b) { // Fourth Version
                System.out.println("Two double parameters: " + a + " , " + b);
                return a + b;
        }
        public static void main(String args[]) {
                MethodOverloadingDemo mod = new MethodOverloadingDemo();
                intintResult;
                doubledoubleResult;
                mod.adding();
                System.out.println();
                mod.adding("Nimai ","Krsna");
                System.out.println();
                intResult = mod.adding(10, 20);
                System.out.println("Sum is  " + intResult);
                System.out.println();
                doubleResult = mod.adding(1.1, 2.2);
                System.out.println("Sum is  " + doubleResult);
                System.out.println();
        }
}
```


Passing objects as arguments

The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter. Infact, java is strictly "pass by value". Objects are not "passed by reference" rather Object references are "passed by value". A copy of the reference's value is what is passed to the method. Java does manipulation on objects by reference. All object variables in java are reference variables.

Example Code block:

```
class Test{
  .....
  /* Here "act" is the formal parameter gets the value of the actual parameter "ac" i.e the address of the Account object pointed by "ac". As "act" is now holding the address of Account object passed, the deposit(50000) called on "act"  increments the actual object, but new object created with "new Account(500000)" will changes the value of "act" stack pointer to the newly created object and will not do any thing on actual object. So "act.deposit(6000)" will not work on actual object.
  */
```

```
public void inrement(Account act){
act.deposit(50000);
System.out.println(act.getBalance()); //prints 1,50,000 as balance
act=new Account(500000);
act.deposit(6000);
System.out.println(act.getBalance()); //prints 5,06,000
 }
 ....
public static void main(String[] args){
    Account ac;
    //Here a is a pointer to Account Object, not Account Object itself.
ac=new Account(101,"Krsna Prasad",100000);
    //Account is created with 1,00,000 initial balance
    Test t=new Test();
t.increment(ac);
    /* Here we are passing the value of 'ac' i.e address reference of the Account
object.
    */
System.out.println(ac.getBalance()); //prints 1,50,000 only not 5,06,000.
 }
 .....
}
```

Session 10 – Strings

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Strings
- ✓ String Pool Concept
- ✓ String Methods
- ✓ StringBuffer& Builder
- ✓ String Manipulations

Strings

Strings - The String class represents character strings. Strings are constant; their values cannot be changed after they are created. The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

Strings are immutable so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

equals() vs "=="

"equals()" method is a instance method, will compare the content of passed String and the existing String, where as "==" will compare the references of stack's.

Note: When we use "==" between primitives it compares the values directly and in case of objects it compares the references/address.

What is String Pooling in Java?
The string pool is the JVM's particular implementation of the concept of string interning:

String interning is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a string intern pool. Basically, a string intern pool allows a runtime to save memory by preserving immutable strings in a pool so that areas of the application can reuse instances of common strings instead of creating multiple instances of it.

This prints' true (even though we don't use equals method: correct way to compare strings)

```
String s = "a" + "bc";
String t = "ab" + "c";
System.out.println(s == t);
```

When compiler optimizes your string literals, it sees that both s and t have same value and thus you need only one string object. It's safe because String is immutable in Java.
As result, both s and t point to the same object and some little memory saved.
Name 'string pool' comes from the idea that all already defined string are stored in some 'pool' and before creating new String object compiler checks if such string is already defined.

If you have multiple Strings who's values are the same, they'll all point to the same string literal in the string pool.
String s1 = "Hari"; //case 1
String s2 = "Hari"; //case 2
In case 1, literal s1 is created newly and kept in the pool. But in case 2, literal s2 refer the s1, it will not create new one instead.
if(s1 == s2) System.out.println("equal"); //Prints equal.

String n1 = new String("Hari");
String n2 = new String("Hari");
if(n1 == n2) System.out.println("equal"); //No output.

==, .equals(), compareTo(), and compare()

Equality comparison: One way for primitives, Four ways for objects		
Comparison	**Primitives**	**Objects**
a == b, a != b	Equal values	**Compares references, not values.** The use of == with object references is generally limited to the following: • Comparing to see if a reference is null. • Comparing two enum values. This works because there is only one object for each enum constant.

		• You want to know if two references are to the *same object*
a.equals(b)	N/A	Compares values for equality. Because this method is defined in the Object class, from which all other classes are derived, it's automatically defined for every class. However, it doesn't perform an intelligent comparison for most classes unless the class overrides it. It has been defined in a meaningful way for most Java core classes. If it's not defined for a (user) class, it behaves the same as ==. It turns out that defining equals() isn't trivial; in fact it's moderately hard to get it right, especially in the case of subclasses.
a.compareTo(b)	N/A	**Comparable interface.** Compares values and returns an int which tells if the values compare less than, equal, or greater than. If your class objects have a natural order, implement the *Comparable<T>* interface and define this method. All Java classes that have a natural ordering implement this (String, Double, BigInteger, ...).
compare(a, b)	N/A	**Comparator interface.** Compares values of two objects. This is implemented as part of the *Comparator<T>* interface, and the typical use is to define one or more small utility classes that implement this, to pass to methods such as sort() or for use by sorting data structures such as TreeMap and TreeSet. You might want to create a Comparator object for the following. • **Multiple comparisions.** To provide several different ways to sort somthing. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort() method. • **System class.** To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length. • **Strategy pattern.** To implement a *Strategey* pattern, which is a situation where you want to represent an *algorithm* as an object that you can pass as a parameter, save in a data structure, etc. If your class objects have one natural sorting order, you may not need this.

Comparing Object references with the == and != Operators

The two operators that can be used with object references are comparing for equality (**==**) and inequality (**!=**). These operators compare two values to see if they **refer to the same object**. Although this comparison is very fast, it is often not what you want.

Usually you want to know if the objects have the same *value*, and not whether two objects are a *reference* to the same object. For example,

```
if (name == "Mickey Mouse")   // Legal, but ALMOST SURELY WRONG
```

This is true only if name is a reference to the *same object* that "Mickey Mouse" refers to. This will be false if the String in name was read from input or computed (by putting strings together or taking the substring), even though name really does have exactly those characters in it.

Many classes (eg, String) define the equals() method to compare the *values* of objects.

Comparing Object values with the equals() Method

Use the equals() method to compare object values. The equals() method returns a boolean value. The previous example can be fixed by writing:

```
if (name.equals("Mickey Mouse"))  // Compares values, not refererences.
```

Because the equals() method makes a == test first, it can be fairly fast when the objects are identical. It only compares the values if the two references are not identical.

Other comparisons - Comparable<T> interface

The equals method and == and != operators test for equality/inequality, but do not provide a way to test for relative values. Some classes (eg, String and other classes with a natural ordering) implement the *Comparable<T>* interface, which defines a compareTo method. You will want to implement *Comparable<T>* in your class if you want to use it with Collections.sort() or Arrays.sort() methods.

Defining a Comparator object

As described in the table above on compare(), you can create Comparators to sort any arbitrary way for any class. For example, the String class defines the CASE_INSENSITIVE_ORDER comparator.

If you override equals, you should also override hashCode()

Overriding hashCode(). The hashCode() method of a class is used for *hashing* in library data structures such as HashSet and HashMap. If you override equals(), you should override hashCode() or your class will not work correctly in these (and some other) data structures.

Shouldn't .equals and .compareTo produce same result?

The general advice is that if a.equals(b) is true, then a.compareTo(b) == 0 should also be true. Curiously, BigDecimal violates this. Look at the Java API documentation for an explanation of the difference. This seems wrong, although their implementation has some plausibiliby.

Other comparison methods

String has the specialized equalsIgnoreCase() and compareToIgnoreCase(). String also supplies the constant String.CASE_INSENSITIVE_ORDER Comparator.

The === operator (Doesn't exist - yet?)

Comparing objects is somewhat awkward, so a === operator has been proposed. One proposal is that a === b would be the same as ((a == b) || ((a != null) &&a.equals(b)))

Common Errors

Using == instead of equals() with Objects

When you want to compare objects, you need to know whether you should use == to see if they are the *same object*, or equals() to see if they may be a different object, but have the *same value*. This kind of error can be very hard to find.

StringBuffer

A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

Imp Constructors

- StringBuffer()
- StringBuffer(String t)
- StringBuffer(int i)

Ex:

StringBuffer sb1=new StringBuffer() -allocates the memory of 16 bytes
StringBuffer sb2=new StringBuffer("Nimai"); -allocates the memory of 21 bytes.
StringBuffer sb3=new StringBuffer(45);-allocates 45 bytes.

Most of the String methods are available for StringBuffer. In addition to it we have the following methods

- void append(String)
- void reverse()
- void setCharAt(char c,intind)
- void deleteCharAt(intind)
- String toString()

Example to work with append() method –

```
classappendDemo {
        public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffersb = new StringBuffer(40);
        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
        }
}
```

The output is: a = 42!

The **append( )** method is most often called when the **+** operator is used on **String** objects. Java automatically changes modifications to a **String** instance into similar `operations on a **StringBuffer**instance. Thus, a concatenation invokes **append( )** on a **StringBuffer**object. After the concatenation has been performed, the compiler inserts a call to **toString( )** to turn the modifiable **StringBuffer**back into a constant **String**.

Example to work with reverse() method –

```
public class StringReverseExample {
        public static void main(String args[]){
        //declare orinial string
        String strOriginal = "Hello World";
```

```
        System.out.println("Original String : " + strOriginal);
        strOriginal = new StringBuffer(strOriginal).reverse().toString();
        System.out.println("Reversed String : " + strOriginal);
        }
}
```

Output of the program is:

Original String : Hello World
Reversed String :dlroWolleH

Example to work with charAt() method-

```
public class CharAtExample{
        public static void main(String args[]) {
        StringBuffersb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
         }
}
```

Output of the program is:

buffer before = Hello
charAt(1) before = e
buffer after = Hillo
charAt(1) after = i

Example to work with deleteCharAt() method –

```
public class DelCharAtExample {
        public static void main(String args[]) {
        StringBuffersb = new StringBuffer("This is a test.");
        sb.delete(4, 7);
        System.out.println("After delete: " + sb);
        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
        }
}
```

Output of the program is:

After delete: This a test.
After deleteCharAt: his a test.

Example to work with toString() method –

```
public class ToStringExample {
        public static void main(String args[]) {
        StringBuffersb = new StringBuffer(40);
        sb.append(5);
        String s = sb.toString();
```

```
        System.out.println(s);
        }
}
```

Output of the program is: 5

StringBuilder
A mutable sequence of characters. This class provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.

The principal operations on a StringBuilder are the append and insert methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string builder. The append method always adds these characters at the end of the builder; the insert method adds the characters at a specified point.

For example, if z refers to a string builder object whose current contents are "start", then the method call z.append("le") would cause the string builder to contain "startle", whereas z.insert(4, "le") would alter the string builder to contain "starlet".

Imp Constructors

- StringBuilder()
- StringBuilder(CharSequenceseq)
- StringBuilder(int capacity)
- StringBuilder(String str)

Example to work with capacity() method –

```
/* Creates a few StringBuilder objects with different capacities */
public class capacityExample{
public static void main(String args[]){
StringBuilder builder1 = new StringBuilder();
StringBuilder builder2 = new StringBuilder(0);
StringBuilder builder3 = new StringBuilder(100);

System.out.println(builder1.capacity());
System.out.println(builder2.capacity());
System.out.println(builder3.capacity());
    }
  }
```

Output of the program is:

```
16
0
100
```

Example to work with insert() method –

```
/**
    * Creates a few StringBuilder objects and uses the append and
    * insert methods to add text to them.
    * We use the \n character for line breaks.
    */
public class insertTextExample {
public static void main(String args[]){
     //Create the StringBuilder
StringBuilder builder = new StringBuilder("Line 1\n");
       //Append text to the end of the buffer
builder.append("Line 3\n");
       //Now we want to add text in between line 1 and line 3
       String lineToInsert = "Line 2\n";
int index = builder.indexOf("Line 3");
builder.insert(index, lineToInsert);
System.out.println(builder.toString());
      }
   }
```

Output of the program is:
Line 1
Line2
Line3

Example to work with delete() method –

```
/**
    * Deletes text from the StringBuilder object
    */
public class deleteTextExample{
public static void main(String args[]){
       //Create the StringBuilder
StringBuilder builder = new StringBuilder("Line 1\n");
       //Append text to the end of the buffer
builder.append("Line 3\n");
       //Now we want to add text in between line 1 and line 3
       String lineToInsert = "Line 2\n";
int index = builder.indexOf("Line 3");
builder.insert(index, lineToInsert);
System.out.println(builder.toString());
       //Now we want to delete the text we just inserted
       //Note that the method returns a new instance of StringBuilder
builder = builder.delete(index, index + lineToInsert.length());
System.out.println(builder.toString());
       }
   }
```

Output of the program is:
Line 1
Line 2
Line 3

Line 1
Line 3

StringBuildervsStringBuffer

- StringBuffer is Thread Safe. It can be used in multithreaded applications.
- StringBuilder is added to java in 1.6 version and it is unthread safe. You can prefer to use StringBuilder if it is not used in multithreaded sequence.
- StringBuilder works faster than StringBuffer

Session 11 – Wrapper Classes

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Wrapper Classes
- ✓ Primitives as Objects
- ✓ Important Methods
- ✓ Autoboxing & Unboxing

Wrapper Classes

Wrapper Classes

Java uses simple or primitive data types, such as int, char and Boolean etc. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. However, at times there is a need to create an object representation of these simple data types. To address this need, Java provides classes that correspond to each of these simple types. These classes encapsulate, or wrap, the simple data type within a class. Thus, they are commonly referred to as wrapper classes.

Wrapper classes corresponding to respective simple data types are as given in table below.

Primitive Datatypes	Wrapper Classes
byte	Byte
short	Short
int	Int
long	Long
char	Character
float	Float
double	Double
boolean	Boolean
void	Void

Wrapper Class API – Boolean

java.lang
Class Boolean

java.lang.Object
└ java.lang.Boolean

The Boolean class wraps a value of the primitive type `boolean` in an object. An object of type `Boolean` contains a single field whose type is `boolean`. In addition, this class provides many methods for converting a `boolean` to a `String` and a `String` to a `boolean`, as well as other constants and methods useful when dealing with a `boolean`.

All Implemented Interfaces:
Serializable, Comparable<Boolean>

---

public final class **Boolean**
extends Object
implementsSerializable, Comparable<Boolean>

Field Summary	
`static Boolean`	`FALSE` -The `Boolean` object corresponding to the primitive value `false`.
`static Boolean`	`TRUE`-The `Boolean` object corresponding to the primitive value `true`.
`static Class<Boolean>`	`TYPE`-The Class object representing the primitive type boolean.

Constructor Summary	
`Boolean(boolean value)`	Allocates a `Boolean` object representing the `value` argument.
`Boolean(String s)`	Allocates a `Boolean` object representing the value `true` if the string argument is not `null` and is equal, ignoring case, to the string `"true"`.

Method Summary	
`boolean`	`booleanValue()` -Returns the value of this `Boolean` object as a boolean primitive.
`int`	`compareTo(Boolean b)`- Compares this `Boolean` instance with another.
`boolean`	`equals(Object obj)` - Returns `true` if and only if the argument is not `null` and is a `Boolean` object that represents the same `boolean` value as this object.
`static boolean`	`getBoolean(String name)`- Returns `true` if and only if the system property named by the argument exists and is equal to the string `"true"`.
`int`	`hashCode()`- Returns a hash code for this `Boolean` object.
`static boolean`	`parseBoolean(String s)` -Parses the string argument as a boolean.
`String`	`toString()`- Returns a `String` object representing this Boolean's value.
`static String`	`toString(boolean b)` -Returns a `String` object representing the specified boolean.
`static Boolean`	`valueOf(boolean b)` -Returns a `Boolean` instance representing the specified `boolean` value.
`static Boolean`	`valueOf(String s)` -Returns a `Boolean` with a value represented by the specified string.

Wrapper Class API - Float

java.lang
Class Float
java.lang.Object
└ **java.lang.Number**
└ **java.lang.Float**

The `Float` class wraps a value of primitive type `float` in an object. An object of type `Float` contains a single field whose type is `float`.
In addition, this class provides several methods for converting a `float` to a `String` and a `String` to a `float`, as well as other constants and methods useful when dealing with a `float`.

All Implemented Interfaces:
Serializable, Comparable<Float>

---

public final class **Float**
extends Number
implements Comparable<Float>

Field Summary	
`static int`	`MAX_EXPONENT` Maximum exponent a finite `float` variable may have.
`static float`	`MAX_VALUE` A constant holding the largest positive finite value of type `float`, $(2-2^{-23})\cdot 2^{127}$.
`static int`	`MIN_EXPONENT`Minimum exponent a normalized `float` variable may have.
`static float`	`MIN_NORMAL` A constant holding the smallest positive normal value of type `float`, 2^{-126}.
`static float`	`MIN_VALUE` A constant holding the smallest positive nonzero value of type `float`, 2^{-149}.
`static float`	`NaN` A constant holding a Not-a-Number (NaN) value of type `float`.
`static float`	`NEGATIVE_INFINITY` A constant holding the negative infinity of type `float`.
`static float`	`POSITIVE_INFINITY` A constant holding the positive infinity of type `float`.
`static int`	`SIZE` The number of bits used to represent a `float` value.
`static Class<Float>`	`TYPE` The`Class` instance representing the primitive type `float`.

Constructor Summary	
`Float(double value)`	Constructs a newly allocated `Float` object that represents the argument converted to type `float`.
`Float(float value)`	Constructs a newly allocated `Float` object that represents the primitive `float` argument.
`Float(String s)`	Constructs a newly allocated `Float` object that represents the floating-point value of type `float` represented by the string.

Method Summary	
byte	byteValue() Returns the value of this Float as a byte (by casting to a byte).
static int	compare(float f1, float f2) Compares the two specified float values.
int	compareTo(Float anotherFloat) Compares two Float objects numerically.
double	doubleValue() Returns the double value of this Float object.
boolean	equals(Object obj) Compares this object against the specified object.
static int	floatToIntBits(float value) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout.
static int	floatToRawIntBits(float value) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout, preserving Not-a-Number (NaN) values.
float	floatValue() Returns the float value of this Float object.
int	hashCode() Returns a hash code for this Float object.
static float	intBitsToFloat(int bits) Returns the float value corresponding to a given bit representation.
int	intValue() Returns the value of this Float as an int (by casting to type int).
boolean	isInfinite() Returns true if this Float value is infinitely large in magnitude, false otherwise.
static boolean	isInfinite(float v) Returns true if the specified number is infinitely large in magnitude, false otherwise.
boolean	isNaN() Returns true if this Float value is a Not-a-Number (NaN), false otherwise.
static boolean	isNaN(float v) Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.
long	longValue() Returns value of this Float as a long (by casting to type long).
static float	parseFloat(String s) Returns a new float initialized to the value represented by the specified String, as performed by the valueOf method of class Float.
short	shortValue() Returns the value of this Float as a short (by casting to a short).
static String	toHexString(float f) Returns a hexadecimal string representation of the float argument.
String	toString() Returns a string representation of this Float object.
static String	toString(float f) Returns a string representation of the float argument.
static Float	valueOf(float f) Returns a Float instance representing the specified float value.
Static Float	valueOf(String s) Returns a Float object holding the float value represented by the argument string s.

If for example you want to store a set of int values in the elements of a Vector, the values in a Vector must be objects and not primitives. When you want to retrieve the values wrapped up in the Integers that are in the Vector elements you will need to cast the elements back to Integers and then call the toxxValue in order to get back the original number

Example to work with Wrapper class - Integer

```
importjava.util.*;
public class VecNum{
public static void main(String argv[]){
   Vector v = new Vector();
v.add(new Integer(1));
v.add(new Integer(2));
for(int i=0; i <v.size();i ++){
      Integer iw=(Integer) v.get(i);
System.out.println(iw.intValue());
   }  }
}
```

Output of the program is:

1
2

Note: Once assigned a value, the value of a wrapper class cannot be changed

Example to work with Wrapper class - Character

```
public class CharacterDemo {
        public static void main(String[] args) {
        char a[] = {'a','b','5','?','A',' '};
        for(int i=0;i<a.length;i++){
                if(Character.isDigit(a[i]))
                System.out.println(a[i] + "is a digit ");
                if(Character.isLetter(a[i]))
                System.out.println(a[i] + "is a letter ");
                if(Character.isWhitespace(a[i]))
                System.out.println(a[i] + "is a White Space ");
                if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + "is a lower case ");
                if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + "is a upper case ");
                }
        }
}
```

Output of the program is:

ais a letter
ais a lower case
ais a upper case
bis a letter
bis a lower case
bis a upper case
5is a digit
Ais a letter
is a White Space

Autoboxing & Unboxing in Java

To add any primitive to a collection, you need to explicitly box (or cast) it into an appropriate wrapper class.It is not possible to put any primitive values, such as int or char, into a collection. Collections can hold only object references.

While taking out an object out of the collection, it needs to be unboxed. Hence the autoboxing and unboxing features of Java 5 can be used to avoid these steps.

From Java 5 onwards, it supports automatic conversion of primitive types (int, float, double etc.) to their object equivalents (Integer, Float, Double,...) in assignments and method and constructor invocations. This conversion is known as Autoboxing.

Java 5 also supports automatic unboxing, where wrapper types are automatically converted into their primitive equivalents if needed for assignments or method or constructor invocations.

For example:

```
int i = 0;
i = new Integer(5); // auto-unboxing

Integer i2 = 5;  // autoboxing
```

Although the Java programming language is an object-oriented language, it's often the case that when using the language you need to work with primitive types. Before J2SE 5.0, working with primitive types required the repetitive work of converting between the primitive types and the wrapper classes. Here you will see how the new autoboxing feature in J2SE 5.0 handles conversions -- for example, between values of type int and values of type Integer. We also discuss some autoboxing-related considerations in determining when two numerical values are equal.

Here is a simple example that uses the printf() method:

```
public class FormatPrint {
  public static void main(String[] args) {
    System.out.printf("There is only %d thing.", 1);
  }
}
```


The signature of the printf() method in the FormatPrint example is:
printf(String format, Object... args)

The number 1 is a primitive and not an object, so you might think that the line:
System.out.printf("There is only %d thing.", 1);

should not compile. However autoboxing takes care of the situation by automatically wrapping the integer value in the appropriate wrapper object. In J2SE 1.4 you would have needed to manually wrap the primitive value using something like new Integer(1).

Another example of where automatically converting from a primitive might be useful is when you use the Collections APIs. The collections classes are designed to store objects. Consider the following simple example of storing int values from 0 to 9 in an ArrayList:

```
import java.util.ArrayList;
 public class Autoboxing {
   public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<Integer>();
      for(int i = 0; i < 10; i++){
        list.add(i);
      }
   }
```

```
}
```

The comparable program for J2SE 1.4.2 would be the following:

```
import java.util.ArrayList;
public class ManualBoxing {
  public static void main(String[] args) {
    ArrayList list = new ArrayList();
    for(int i = 0; i < 10; i++){
      list.add(new Integer(i));
    }
  }
}
```

With ManualBoxing you need to explicitly create the Integer object using list.add(new Integer(i)). Contrast that with Autoboxing, where the int i is autoboxed to an Integer object in the line list.add(i).
Autoboxing works well with other new J2SE 5.0 features. For example, the autoboxing feature allows seamless integration between generic types and primitive types. In the ManualBoxing example, the elements of the ArrayList are of type Object. By comparison, in the Autoboxing example, the elements of list are of type Integer.

Let's extend the Autoboxing example to iterate through the elements in the ArrayList and calculate their sum. Notice that this new version also uses the new J2SE 5.0 enhanced for loop to iterate through the elements.

```
import java.util.ArrayList;
  public class Autoboxing {
    public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<Integer>();
      for(int i = 0; i < 10; i++){
        list.add(i);
      }
      int sum = 0;
     for ( Integer j : list){
       sum += j;
     }
     System.out.printf("The sum is %d.", sum );
    }
  }
```

Autoboxing is used in a number of places in the updated Autoboxing example. First, ints are boxed to Integers as they are added to the ArrayList. Then Integers are unboxed to ints to be used in calculating the sum. Finally, the int representing the sum is boxed for use in the printf() statement.

The transparency of the boxing and unboxing makes autoboxing easy to use. However using the autoboxing feature requires some care. In particular, testing for the equality of objects created by autoboxing is not the same as testing for the equality of objects that are not created by autoboxing. To see this, look at the following BoxingEquality class:

```
import java.util.ArrayList;
  public class BoxingEquality {
    public static void main(String[] args) {
```

```
        int i = 2;
        int j = 2;
        ArrayList <Integer> list = new ArrayList<Integer>();
        list.add(i);
        list.add(j);
        System.out.printf("It is %b that i ==j.\n",
                    (i==j)); //(1)
        System.out.printf("It is %b that " +
                    "list.get(0) == list.get(1).\n",
                    list.get(0)==list.get(1)); //(2)
        System.out.printf("It is %b that " +
                   "list.get(0).equals(list.get(1)).",
                   list.get(0).equals(list.get(1))); //(3)
    }
}
```

The first print statement in BoxingEquality compares the equality of the primitives i and j. The second print statement compares the equality of the objects created by autoboxing i and j. The third print statement compares the value of the objects created by autoboxing i and j. You would expect the first and the third print statements to return true, but what about the second? The output from running the BoxingEquality program is:

```
It is true that i ==j.
It is true that list.get(0) == list.get(1).
It is true that list.get(0).equals(list.get(1)).
```


Now change the values of i and j to 2000.

```
import java.util.ArrayList;

public class BoxingEquality {
  public static void main(String[] args) {
    int i = 2000;
    int j = 2000;
    // . . .
}
```

Save, recompile, and rerun BoxingEquality. This time the results are different:

```
It is true that i ==j.
It is false that list.get(0) == list.get(1).
It is true that list.get(0).equals(list.get(1)).
```

The primitives are equal and the values of the boxed ints are equal. But this time the ints point to different objects. What you have discovered is that for small integral values, the objects are cached in a pool much like Strings. When i and j are 2, a single object is referenced from two different locations. When i and j are 2000, two separate objects are referenced. Autoboxing is guaranteed to return the same object for integral values in the range [-128, 127], but an implementation may, at its discretion, cache values outside of that range. It would be bad style to rely on this caching in your code.

In fact, testing for object equality using == is, of course, not what you normally intend to do. This cautionary example is included in here because it is easy to lose track of whether you are dealing with objects or primitives when the compiler makes it so easy for you to move back and forth between them.

NKXGEN