

Java Tutorial

- Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial gives a complete understanding on Java.
- This reference will take you through simple and practical approach while learning Java Programming language.



Prerequisites

- Before you start doing practice with various types of examples given in this reference, I'm making an assumption that you are already aware about what is a computer program and what is a computer programming language.



Where It Is Used

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.



Types Of Java Applications

There are mainly 4 type of applications that can be created using Java:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install

on every machine such as media player, antivirus etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in Java.



Cont..

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.



Java Overview

- Java programming language was originally developed by Sun Microsystems, which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems Java platform (Java 1.0 [J2SE]).
- As of December 08 the latest release of the Java Standard Edition is 6 (J2SE).
- With the advancement of Java and its wide spread popularity, multiple configurations were built to suite various types of platforms.
- Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.



Features of Java

1. Simple
2. Object-oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed



Features Of Java

- **Object Oriented** : In java everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent**: Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.



Features Of Java Cont..

- **Simple** :Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
- **Secure** : With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural- neutral** :Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.



Features Of Java Cont..

- **Portable** :being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary means We may carry the Java byte code to any platform.
- **Robust** : Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in Java. There is exception handling and type checking mechanism in Java. All these points makes Java robust.
- **Multi-threaded** : With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.



Features Of Java Cont..

- **Interpreted** :Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance**: With the use of Just-In-Time compilers Java enables high performance.
- **Distributed** :Java is designed for the distributed environment of the internet.
- **Dynamic** : Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

My First Java Programme

```
public class MyFirstJavaProgram {  
    public static void main(String []args) {  
        System.out.println("Hello World");  
    }  
}
```



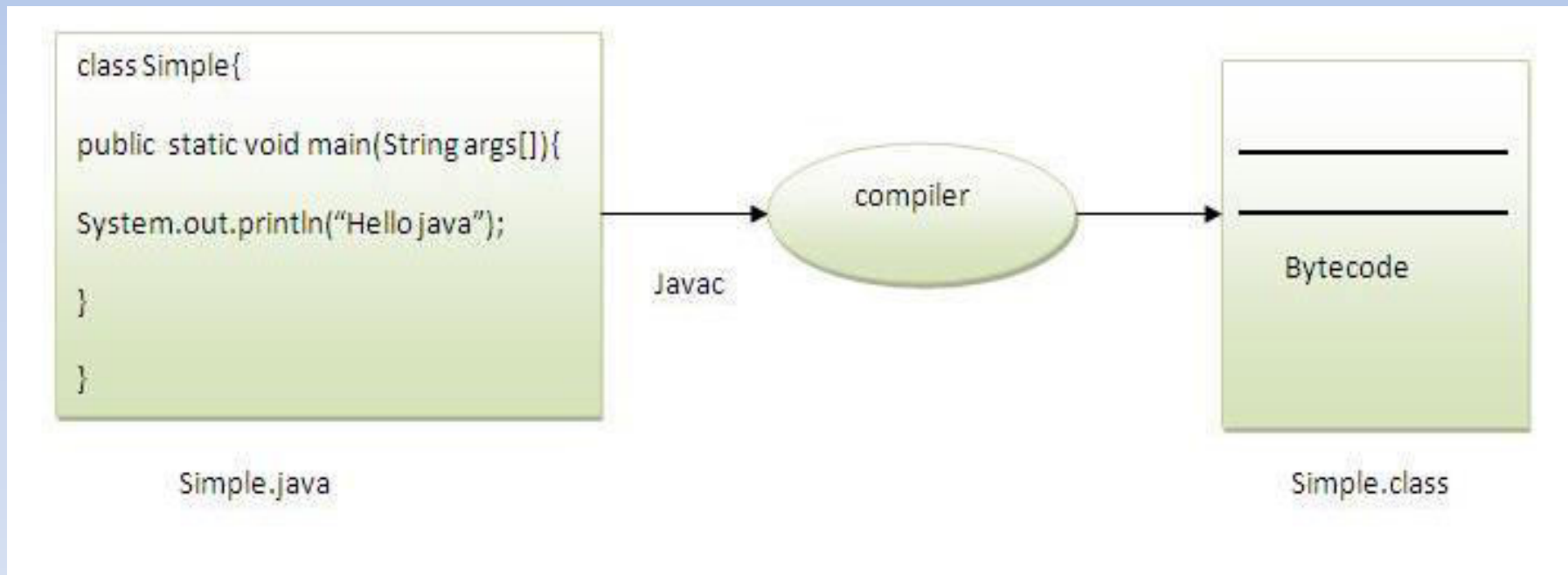
Understanding first java program

- class is used to declare a class in Java.
- public is an access modifier which represents visibility, it means it is visible to all.
- static is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- void is the return type of the method, it means it doesn't return any value, main represents startup of the program.
- String[] args is used for command line argument. We will learn it later.
System.out.println() is used print statement.

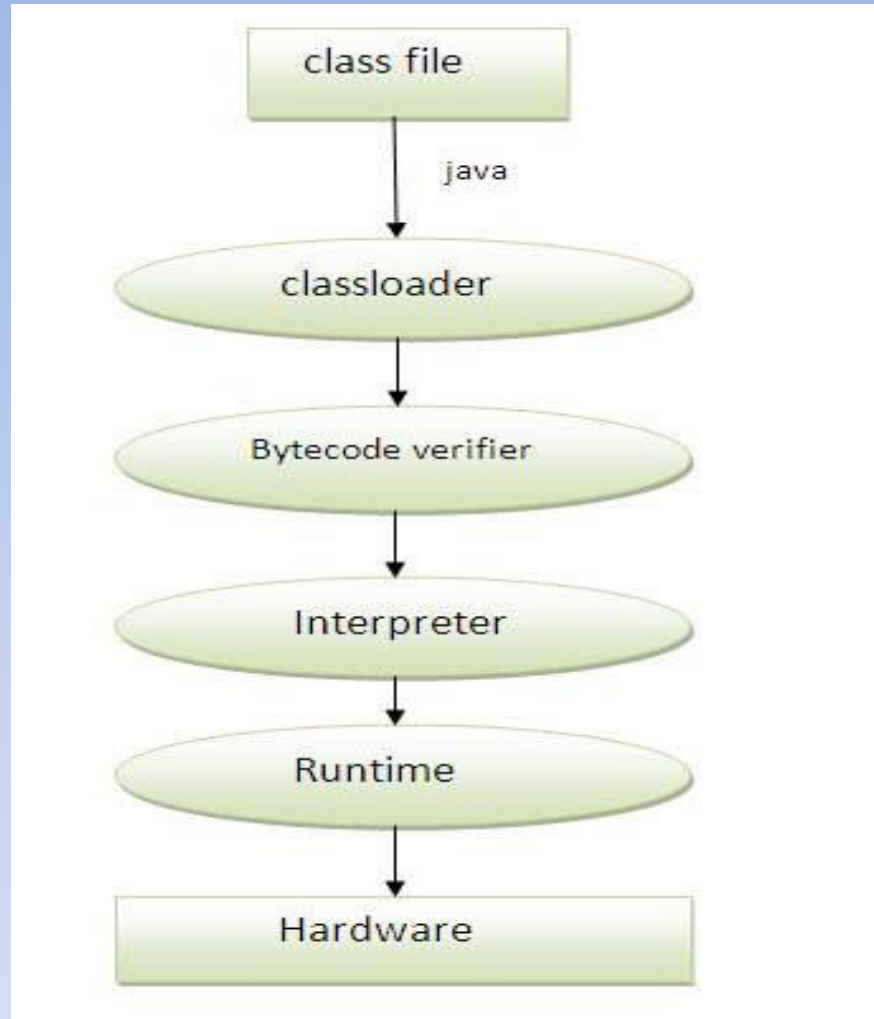


What happens at compile time?

At compile time, Java file is compiled by Java Compiler (It does not interact with OS) and converts the Java code into bytecode.



What happens at runtime?



ClassLoader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Questions

- Can you save a java source file by other name than the class name?
- Can you have a empty java file?
- What if no one class have main function?
- How to print “Hello World” without main function?
- Can you have multiple classes in a java source file?



Java is an Object Oriented Language

As a language that has the Object Oriented feature Java supports the following fundamental concepts:

- 1) Classes
- 2) Objects
- 3) Encapsulation
- 4) Abstraction
- 5) Inheritance
- 6) Polymorphism



Objects

- Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans etc. All these objects have a state and behavior.
- An object is an instance of a [class](#). The relationship is such that many objects can be created using one class. Each object has its own data but its underlying structure (i.e., the type of data it stores, its behaviors) are defined by the class.



Classes

- A *class*--the basic building block of an object-oriented language such as Java
- A class is a blue print from which individual objects are created.
- A class specifies the design of an object. It states what data an object can hold and the way it can behave when using the data.
- Class is a template that describes the data and behavior associated with *instances* of that class.



Classes

- The data associated with a class or object is stored in *variables*.
- The behavior associated with a class or object is implemented with *methods*. (Methods are similar to the functions or procedures in procedural languages such as C).

Class Example

```
public class Book {  
    private String title;  
    private String author;  
    private String publisher;  
    public Book(String bookTitle, String authorName,  
        String publisherName) {  
        title = bookTitle;  
        author = authorName;  
        publisher = publisherName;  
    }  
    //all getter and setter methods  
}
```



Object Example

An instance of this class will be a book object:

```
Book firstBook = new Book("Complete  
Reference","ABC","XYZ");
```

Objects can be created by using **new** keyword in java.

As mentioned previously a class provides the blueprints for objects. So basically an object is created from a class. In java the new key word is used to create new objects.

There are three steps when creating an object from a class:

- **Declaration** . A variable declaration with a variable name with an object type. Like Book firstBook
- **Instantiation** . The 'new' key word is used to create the object.
- **Initialization** . The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

eg. new Book();



More Details Of Class

- A class can contain any of the following variable types.
 - 1) **Local variables**
 - 2) **Instance variables**
 - 3) **Class variables**

Local Variable

Local variable are variables defined inside methods, constructors or blocks.

The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Eg.

```
Public void setTitle(String title){  
    this.title=title;  
}
```



Rules For Local Variable

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.



Example Of Local Variable

```
public class Test{
    public void pupAge(){
        int age = 0;
        age = age + 7;
        System.out.println("Puppy age is : " + age);
    }

    public static void main(String args[]){
        Test test = new Test();
        test.pupAge();
    }
}
```

This would produce following result:

```
Puppy age is: 7
```

Problem With Local Variable

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test{
    public void pupAge() {
        int age;
        age = age + 7;
        System.out.println("Puppy age is : " + age);
    }

    public static void main(String args[]){
        Test test = new Test();
        test.pupAge();
    }
}
```

This would produce following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
           ^
1 error
```

Instance variables

Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Eg.

```
private String title;  
private String author;  
private String publisher;
```



Rules For Instance Variable

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present through out the class.
- Instance variables can be declared in class level before or after use.

Rules For Instance Variable

- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) the should be called using the fully qualified name *.ObjectReference.VariableName*.



Example Of Instance Variable

```
public class Employee{
    // this instance variable is visible for any child class.
    public String name;
    // salary variable is visible in Employee class only.
    private double salary;|
    // The name variable is assigned in the constructor.
    public Employee (String empName){
        name = empName;
    }

    // The salary variable is assigned a value.
    public void setSalary(double empSal){
        salary = empSal;
    }

    // This method prints the employee details.
    public void printEmp(){
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }

    public static void main(String args[]){
        Employee empOne = new Employee("Ransika");
        empOne.setSalary(1000);
        empOne.printEmp();
    }
}
```

This would produce following result:

```
name : Ransika
salary :1000.0
```

Class variables

Class variables are variables declared with in a class, outside any method, with the static keyword.

Eg.

```
private static int count=0;
```


Rules For Class/Static Variable

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.

Rules For Class/Static Variable

- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *.ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.



Example Of Class/Static Variable

```
public class Employee{
    // salary variable is a private static variable
    private static double salary;

    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development ";

    public static void main(String args[]){
        salary = 1000;
        System.out.println(DEPARTMENT+"average salary:"+salary);
    }
}
```

This would produce following result:

Development average salary:1000

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

- A class can have any number of methods to access the value of various kind of methods.

Like Book Class can have

bookIssue()

addBook()

deleteBook()

updateBook()

Source file declaration rules

- There can be only one public class per source file.
- A source file can have multiple non public classes.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is `public class Employee{}` Then the source file should be as `Employee.java`.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.



A Simple Case Study

- For our case study we will be creating two classes. They are Employee and EmployeeTest.
- First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now save this source file with the name Employee.java.
- The Employee class has four instance variables name, age, designation and salary. The class has one explicitly defined constructor which takes a parameter.



A Simple Case Study

```
public class Employee{
    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name){
        this.name = name;
    }
    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge){
        age = empAge;
    }
    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig){
        designation = empDesig;
    }
    /* Assign the salary to the variable salary.*/
    public void empSalary(double empSalary){
        salary = empSalary;
    }
    /* Print the Employee details */
    public void printEmployee(){
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}
```

A Simple Case Study

```
public class EmployeeTest{

    public static void main(String args[]){
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();

        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}
```


How To Run

```
C : \programme> javac Employee.java  
C : \programme> java EmployeeTest
```

Name:James Smith

Age:26

Designation:Senior Software Engineer

Salary:1000.0

Name:Mary Anne

Age:21

Designation:Software Engineer

Salary:500.0

Java Keywords

abstract	assert	boolean	break	byte	case
catch	char	class	const*	continue	default
double	do	else	enum	extends	false
final	finally	float	for	goto*	if
implements	import	instanceof	int	interface	long
native	new	null	package	private	protected
public	return	short	static	strictfp	super
switch	synchronized	this	throw	throws	transient
true	try	void	volatile	while	

Java Basic Datatypes

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

Primitive Datatypes

Data type	Bytes	Min Value	Max Value	Literal Values
byte	1	-2^7	$2^7 - 1$	123
short	2	-2^{15}	$2^{15} - 1$	1234
int	4	-2^{31}	$2^{31} - 1$	12345, 086, 0x675
long	8	-2^{63}	$2^{63} - 1$	123456
float	4	-	-	1.0
double	8	-	-	123.86
char	2	0	$2^{16} - 1$	'a', '\n'
boolean	-	-	-	true, false

Reference Data Types

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example : `Animal animal = new Animal("giraffe");`



Java Modifier Types

- Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:
- [Java Access Modifiers](#)
- [Non Access Modifiers](#)

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones)



Java Modifiers

Modifier	Class	Class Variables	Methods	Method Variables
public	✓	✓	✓	
private		✓	✓	
protected		✓	✓	
<i>default</i>	✓	✓	✓	
final	✓	✓	✓	✓
abstract	✓		✓	
strictfp	✓		✓	
transient		✓		
synchronized			✓	
native			✓	
volatile		✓		
static	✓	✓	✓	

Modifier Example

```
public class className {  
    // ...  
}  
private boolean myFlag;  
static final double weeks = 9.5;  
protected static final int BOXWIDTH = 42;  
public static void main(String[] arguments) {  
    // body of method  
}
```


Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Default:-Visible to the package. No modifiers are needed.
- Private:-Visible to the class only .
- Public:-Visible to the world.
- Protected:-Visible to the package and all subclasses.



Non Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.



Naming convention

A naming convention is a rule to follow as you decide what to name your identifiers (e.g. class, package, variable, method, etc.), but it is not mandatory to follow that is why it is known as convention not rule.

Advantage:

By using standard Java naming conventions they make their code easier to read for themselves and for other programmers. Readability of Java code is important because it means less time is spent trying to figure out what the code does.

Class Name	should begin with uppercase letter and be a noun e.g.String,System,Thread etc.
Interface Name	should begin with uppercase letter and be an adjective (wherever possible), e.g.Runnable,ActionListener etc.
Method Name	should begin with lowercase letter and be a verb. e.g.main(),print(),println(),actionPerformed() etc.
Variable Name	should begin with lowercase letter e.g. firstName,orderNumber etc.
Package Name	should be in lowercase letter, e.g. java.lang.sql.util etc.
Constant Name	should be in uppercase letter, e.g. RED YELLOW MAX PRIORITY etc.

Method Overloading

Method overloading means when two or more methods have the same name but a different signature.

Signature of a method is nothing but a combination of its name and the sequence of its parameter types.

Advantages of method overloading

It allows you to use the same name for a group of methods that basically have the same purpose.

Method overloading increases the readability of the program.

Method Overloading

Different ways to overload the method

There are two ways to overload the method in Java

- 1. By changing number of arguments
- 2. By changing the data type

Note: In Java, Method Overloading is not possible by changing return type of the method.

```
public class Calculation
{
    public int addition(int i, int j)
    {
        return i + j ;
    }

    public String addition(String s1, String s2)
    {
        return s1 + s2;
    }

    public double addition(double d1, double d2)
    {
        return d1 + d2;
    }
}
```

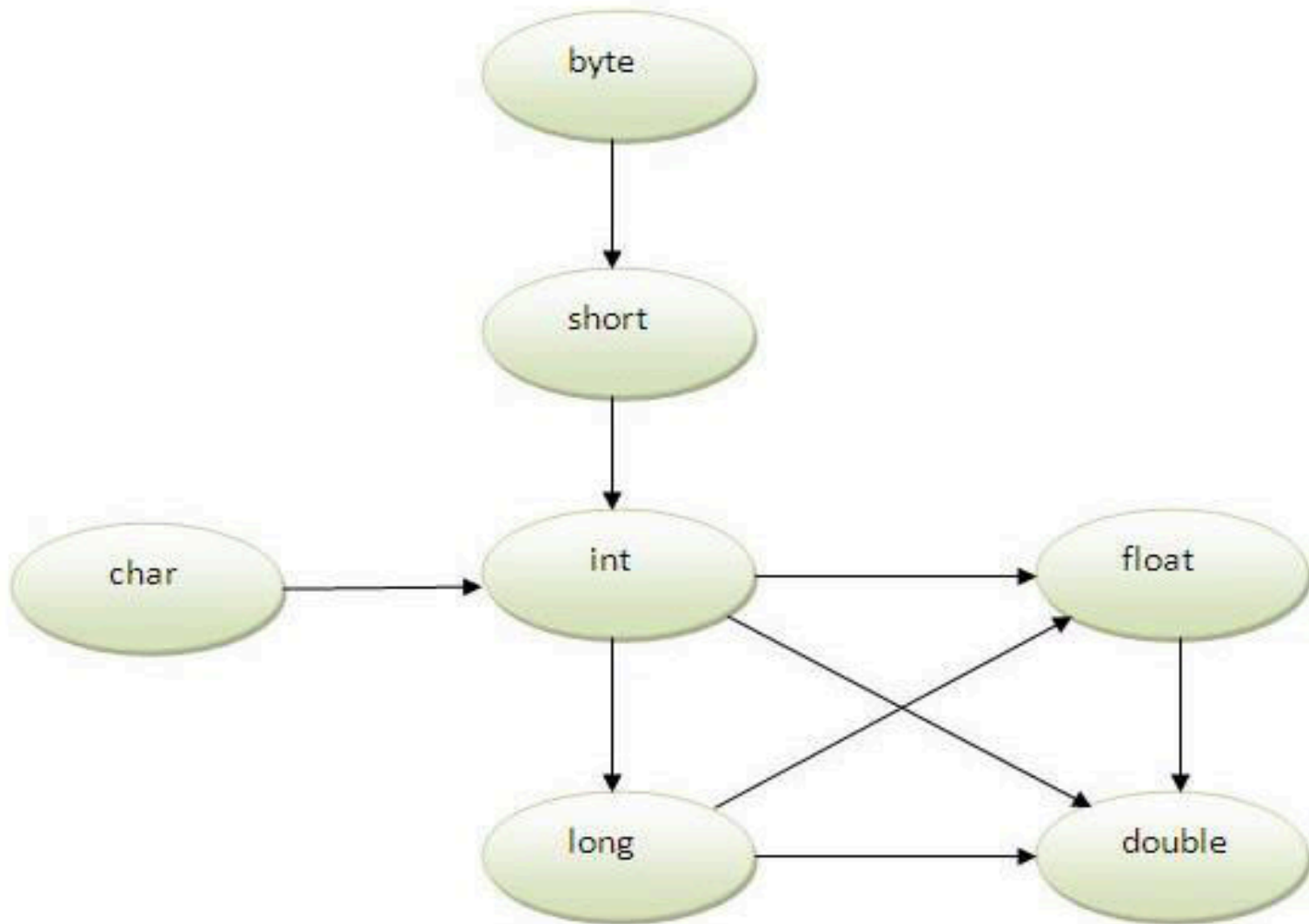
```
public class CalculationMain
{
    public static void main(String args[])
    {
        Calculation sObj = new Calculation();
        System.out.println(sObj.addition(1,2));
        System.out.println(sObj.addition("Hello ", "World"));
        System.out.println(sObj.addition(1.5,2));
    }
}
```


Questions

Que) Why Method Overloading is not possible by changing the return type of method?

Que) Can we overload main() method?

Method Overloading and TypePromotion



Example

```
class Calculation{
    void sum(int a,long b){
        System.out.println(a+b);
    }
    void sum(int a,int b,int c){
        System.out.println(a+b+c);
    }
    void sum(int a,int b,float c){
        System.out.println(a+b+c);
    }
    public static void main(String args[]){
        Calculation obj=new Calculation();
        obj.sum(20,20); //now second int literal will be promoted to long
        obj.sum(20,20,20);
        obj.sum(20,20,20.76);
    }
}
```

Example

```
class Calculation
{
    void sum(int a,int b){
        System.out.println("int arg method invoked");
    }
    void sum(long a,long b){
        System.out.println("long arg method invoked");
    }
    void sum(double a,long b){
        System.out.println("double long arg method invoked");
    }
    public static void main(String args[]){
        Calculation obj=new Calculation();
        obj.sum(20,20);
        obj.sum(20,201);
        obj.sum(20.75,20);
        obj.sum(20.75,20.89);
    }
}
```

Example

```
class Calculation
{
    void sum(int a,long b){
        System.out.println("int long arg method invoked");
    }
    void sum(long a,long b){
        System.out.println("long int arg method invoked");
    }
    void sum(double a,long b){
        System.out.println("double long arg method invoked");
    }
    public static void main(String args[]){
        Calculation obj=new Calculation();
        obj.sum(20,20);
        obj.sum(20,201);
        obj.sum(20.75,20);
    }
}
```

Constructor

- Constructor is a special type of method that is used to initialize the state of an object/initialize a value to instance variable.
- Constructor is invoked at the time of object creation. It constructs the values i.e. data for the object that is why it is known as constructor.
- Constructor is just like the instance method but it does not have any explicit return type.

Characteristics or Rules for Constructor

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type.
3. Constructor can have **arguments**
4. Constructor Can be **overloaded**.
5. Constructor should be public but it can have other **access specifier** too.
5. Constructor will **automatically** invoke when you create object of class using **new** keyword.
6. Constructor cannot be call explicitly.
7. Constructor cannot be inherited.

Types Of Constructor

- There are two types of constructors:
 - 1) default constructor (no-arg constructor)
 - 2) parameterized constructor

Remember Point:-

- 1) If you don't provide any constructor in your class compiler will provide default constructor.
- 2) moment add any constructor in class u will loose to get default constructor from compiler.
- 3) Your class should have default constructor in case of **Inheritance**
- 4) Same constructor can be call to other constructor using **this** and derived class constructor using **super** keyword

Questions

- What If there is no constructor in a class, is compiler automatically creates a default constructor?
- What default constructor do.
- What the use of parametrize constructor
- What the difference between constructor and methods.
- Does constructor can have void return type?
- Does copy constructor is available in java.
- What the alternative of copy constructor in java?
- Can constructor perform other tasks instead of initialization?

Static

- The static keyword is used in Java mainly for memory management. We may apply static keyword with variables, methods and blocks. The static keyword belongs to the class rather than instance of the class.

The static can be:

- 1.variable (also known as class variable)
- 2.method (also known as class method)
- 3.block

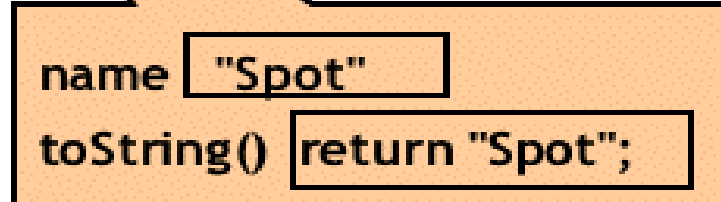
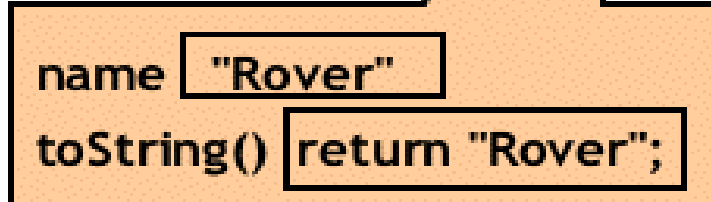
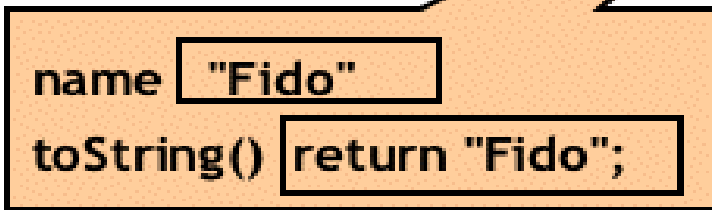
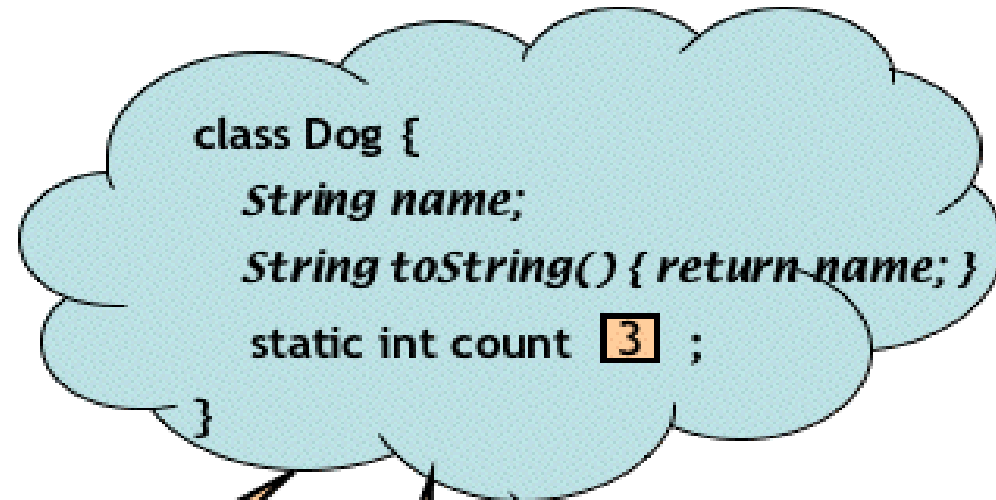
Static Variable

- If you declare any variable as static, it is known as a static variable
- It is a variable which belongs to the class and not to an object instance.
- The static variable can be used to refer to the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in the class area at the time of class loading.
- It can be initialized at the time of object creation.

Static Variable

- Static variables are initialized only once , at the start of the execution . These variables will be initialized first, before the initialization of any instance variables.
- A **single copy** to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn't need any object.
- Syntax : **<class-name>.<variable-name>**.
- *Static variable can be final to make constant.*
- *Syntax: public static final double
RATE_OF_INT=15.5;*

Static Variable Memory Diagram



Static Method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.
- It is a method which **belongs to the class** and **not** to the **object**(instance)
- A static method **can access only static data**. It can not access non-static data (instance variables)

Static Method

- A static method **can call only** other **static methods** and can't call a non-static method from it.
- A static method can be **accessed directly** by the **class name** and doesn't need any object but can be call by object.
- Syntax : *<class-name>.<method-name>*
- A static method cannot refer to “this” or “super” keywords in anyway.
- main method is static , since it must be accessible for an application to run , before any instantiation takes place.

Static Block

The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM

```
class Test{  
    static {  
        //Code goes here  
    }  
}
```

A static block helps to initialize the static data members, just like constructors help to initialize instance members

Questions

Que) Can we execute a program without main() method?

This reference in java

- **this keyword in Java** is a special keyword which can be used to represent current object or instance of any [class in Java](#).
 - “this” can also call constructor of same class in Java and used to call overloaded constructor.
 - if used than it must be first statement in constructor `this()` will call no argument constructor.
 - and `this(parameter)` will call one argument constructor with appropriate parameter.
- Example continue in next slide

example

```
class Loan{
    private double interest;
    private String type;

    public Loan() {
        this("personal loan");
    }

    public Loan(String type) {
        this.type = type;
        this.interest = 0.0;
    }
}
```

If member variable and local variable name conflict than **this** can be used to refer member variable. here is an example of this with member variable:

```
public Loan(String type, double interest){  
    this.type = type;  
    this.interest = interest;  
}
```

Here local variable interest and member variable interest conflict which is easily resolve by referring member variable as **this.interest**

this is a [final variable in Java](#) and you can not assign value to this. this will result in compilation

```
this = new Loan();  
//cannot assign value to final variable : this
```

you can call methods of class by using **this keyword** as shown in below example.

```
public String getName() {  
    return this.toString();  
}
```


- **this** can also be passed as method parameters since it represent current object of class.
- Java This can be used to get the handle of the current class

```
Class className = this.getClass();
```

Though this can also be done by, `Class className = ABC.class;` // here ABC refers to the class name and you need to know that!

Java - String Class

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings:

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```



```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!".

As like any other object, you can create String objects by using the new keyword and a constructor.

The String class has eleven constructors that allow you to provide the initial value of the string using different sources

```
public class StringDemo{  
  
    public static void main(String args[]){  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};  
        String helloString = new String(helloArray);  
        System.out.println( helloString );  
    }  
}
```

This would produce following result:

hello.

Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters then you should use [String Buffer & String Builder](#) Classes.

Cont....

String Length:

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object.

```
String palindrome = "Dot saw I was Tod";  
int len = palindrome.length();  
System.out.println( "String Length is : " + len );  
o/p:- String Length is : 17
```

Concatenating Strings:

The String class includes a method for concatenating two strings:

- 1) method concat
- 2) + operator

- 1) `string1.concat(string2);`
- 2) `"My name is ".concat("Zara");`
- 3) `"Hello," + " world" + "!"`

char charAt(int index)

Returns the character at the specified index.

int compareTo(Object o)

Compares this String to another Object.

int compareTo(String anotherString)

Compares two strings lexicographically.

int compareToIgnoreCase(String str)

Compares two strings lexicographically, ignoring case differences.

String concat(String str)

Concatenates the specified string to the end of this string.

boolean contentEquals(StringBuffer sb)

Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.

static String copyValueOf(char[] data)

Returns a String that represents the character sequence in the array specified.

static String copyValueOf(char[] data, int offset, int count)

Returns a String that represents the character sequence in the array specified.

boolean endsWith(String suffix)

Tests if this string ends with the specified suffix.

boolean equals(Object anObject)

Compares this string to the specified object.

boolean equalsIgnoreCase(String anotherString)

Compares this String to another String, ignoring case considerations.

byte getBytes()

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

byte[] getBytes(String charsetName

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

int hashCode()

Returns a hash code for this string.

int indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character.

int indexOf(int ch, int fromIndex)

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

int indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring.

int indexOf(String str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

int lastIndexOf(int ch, int fromIndex)

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

int lastIndexOf(int ch, int fromIndex)

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

int lastIndexOf(String str)

Returns the index within this string of the rightmost occurrence of the specified substring.

int lastIndexOf(String str, int fromIndex)

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

int length()

Returns the length of this string.

boolean matches(String regex) :-Tells whether or not this string matches the given regular expression.

String replace(char oldChar, char newChar) :-Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

String replaceAll(String regex, String replacement)
Replaces each substring of this string that matches the given regular expression with the given replacement.

String replaceFirst(String regex, String replacement)
Replaces the first substring of this string that matches the given regular expression with the given replacement.

String[] split(String regex) :-Splits this string around matches of the given regular expression.

boolean startsWith(String prefix) :-Tests if this string starts with the specified prefix.

boolean startsWith(String prefix, int toffset) :-Tests if this string starts with the specified prefix beginning a specified index.

String substring(int beginIndex) :-Returns a new string that is a substring of this string.

String substring(int beginIndex, int endIndex) :-Returns a new string that is a substring of this string.

char[] toCharArray() :-Converts this string to a new character array.

String toLowerCase() :-Converts all of the characters in this String to lower case using the rules of the default locale.

String toString() :-This object (which is already a string!) is itself returned.

String toUpperCase() :-Converts all of the characters in this String to upper case using the rules of the default locale.

String trim() :-Returns a copy of the string, with leading and trailing whitespace omitted.

static String valueOf([int][float][.....] x) :-Returns the string representation of the passed data type argument.

Continue with more details about String later

Inheritance

- Inheritance can be defined as the process where one object acquires the properties of another.
- Inheritance is a mechanism in which one object acquires all the properties and behaviours of parent object.
- The idea behind inheritance is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you reuse (or inherit) methods and fields.
- Inheritance represents the IS-A relationship.
- **extends** keyword is used to achieve inheritance.

Inheritance Example

```
public class Animal{  
}  
  
public class Mammal extends Animal{  
}  
  
public class Reptile extends Animal{  
}  
  
public class Dog extends Mammal{  
}
```

Animal is the superclass of Mammal class.

Animal is the superclass of Reptile class.

Mammal and Reptile are subclasses of Animal class.

Dog is the subclass of both Mammal and Animal classes.

```
public class Animal{  
}  
  
public class Mammal extends Animal{  
}  
  
public class Reptile extends Animal{  
}  
  
public class Dog extends Mammal{  
}
```

Mammal IS-A Animal

Reptile IS-A Animal

Dog IS-A Mammal

Hence : Dog IS-A Animal as well

Instanceof Operator

- The instanceof operator compares an object to a specified type.
- You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
- The instanceof operator is also known as type comparison operator because it compares the instance with type.
- It returns either true or false.
- If we apply the instanceof operator with any variable that have null value, it returns false.

Instanceof Operator Example

```
public class Dog extends Mammal{

    public static void main(String args[]) {

        Animal a = new Animal();
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

This would produce the following result:

```
true
true
true
```

Types of Inheritance

- There exists basically three types of [inheritance](#).
- Single Inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance

In **single inheritance**, one class extends one class only.

In **multilevel inheritance**, the ladder of single inheritance increases.

In **multiple inheritance**, one class directly extends more than one class.

In **hierarchical inheritance** one class is extended by more than one class.

Single Inheritance

```
class Aves {
    public void nature() {
        System.out.println("Generally, Aves fly");
    }
}

class Bird extends Aves {
    public void eat() {
        System.out.println("Eats to live");
    }
}
```

Multilevel Inheritance

```
class Aves {
    public void nature() {
        System.out.println("Generally, Aves fly");
    }
}

class Bird extends Aves {
    public void eat() {
        System.out.println("Eats to live");
    }
}

public class Parrot extends Bird {
    public void eat() {
        System.out.println("Parrot eats seeds and fruits");
    }
}

public static void main(String args[]) {
    Parrot p1 = new Parrot();
    p1.eat();
}
}
```

Multilevel Inheritance

```
class Aves {
    public void nature() {
        System.out.println("Generally, Aves fly");
    }
}

class Bird extends Aves {
    public void eat() {
        System.out.println("Eats to live");
    }
}

public class Parrot extends Bird {
    public void eat() {
        System.out.println("Parrot eats seeds and fruits");
    }
}

public static void main(String args[]) {
    Parrot p1 = new Parrot();
    p1.eat();
}
}
```

Multiple Inheritance

- In multiple inheritance, **one class extends multiple classes**. Java **does not support multiple inheritance** but C++ supports. The above program can be modified to illustrate multiple inheritance. The following program does not work.

```
class Aves { }
```

```
class Bird { }
```

```
class Parrot extends Aves, Bird { }
```

Note:-Java supports multiple inheritance partially through interfaces.

Hierarchical Inheritance

```
class Aves {
    public void nature() {
        System.out.println("Generally, Aves fly");
    }
}

class Bird extends Aves {
    public void eat() {
        System.out.println("Eats to live");
    }
}

public class Parrot extends Bird {
    public void eat() {
        System.out.println("Parrot eats seeds and fruits");
    }
}

class Vulture extends Aves {
    public void vision() {
        System.out.println("Vulture can see from high altitudes");
    }
}
```

Disadvantages of Inheritance

- Both classes (super and subclasses) are tightly-coupled.
- As they are tightly coupled (binded each other strongly with extends keyword), they cannot work independently of each other.
- Changing the code in super class method also affects the subclass functionality.
- If super class method is deleted, the code may not work as subclass may call the super class method with super keyword. Now subclass method behaves independently.-

Aggregation In Java

- If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship
- Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

Aggregation In Java

```
class Employee{  
    int id;  
    String name;  
    Address address;//Address is a class  
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

Ans:-For Code Reusability.

When use Aggregation?

- ◆ Code reuse is also best achieved by aggregation when there is no is-a relationship.
- ◆ Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Overriding

- In the previous chapter, we talked about super classes and sub classes.
- If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.
- The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.
- In object-oriented terms, overriding means to override the functionality of an existing method.

```
class Animal{

    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal{

    public void move() {
        System.out.println("Dogs can walk and run");
    }
}
```

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.

Rules for method overriding:

- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.

Rules for method overriding:

- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not.
- However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Questions

1) Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism so we will learn it later.

2) Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Compare Overloading and Overriding

Method Overloading	Method Overriding
<p>Method overloading is used to increase the readability of the program.</p>	<p>Method overriding is used to provide the specific implementation of the method that is already provided by its super class.</p>
<p>method overloading is performed within a class.</p>	<p>Method overriding occurs in two classes that have IS-A relationship.</p>
<p>In case of method overloading parameter must be different.</p>	<p>In case of method overriding parameter must be same.</p>

Applying access modifier with method overriding

- If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

Access Levels are in sequence of

1)Private

2)Default

3)Protected

4)Public

Example

Cont.. Next page

```
class A{
    protected void msg() {
        System.out.println("Hello Java");
    }
}

public class Simple extends A{
    void msg() {
        System.out.println("Hello java");
    }
}

}
```

the **default** modifier is more restrictive than **protected**.
That is why there is compile time error.

Covariant Return Type

- The covariant return type specifies that the return type may vary in the same direction as the subclass.
- Before Java5, it was not possible to override any method by changing the return type.
- But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.

Let's take a simple example:

```
class A{
public A get(){return this;}
}
class B extends A{
    public B get(){return this;}
    public void message()
    {
        System.out.println("welcome to covariant return type");
    }
    public static void main(String args[]){
        new B().get().message();
    }
}
```

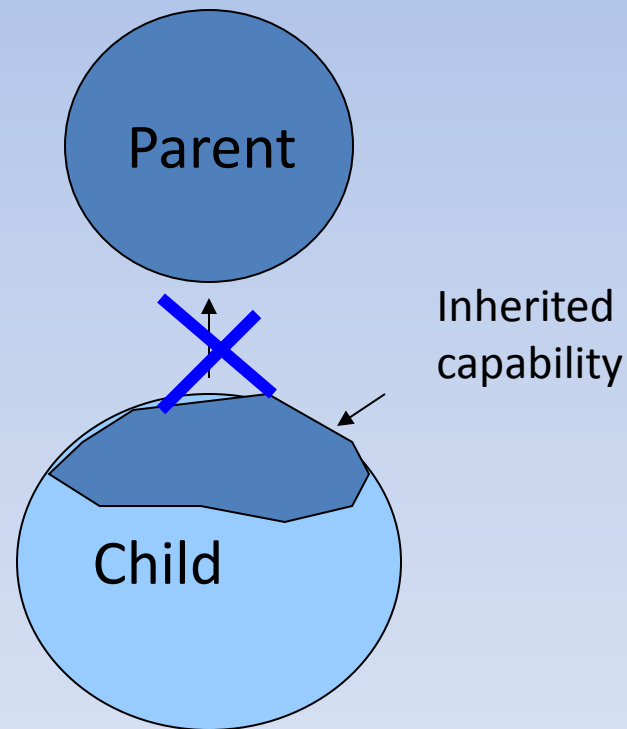
super keyword

- The super is a reference variable that is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly because it is referred by super reference variable.

Usage of super Keyword:-

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

Restricting Inheritance



Final Members: A way for Preventing Overriding of Members in Subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:
 - final int marks = 100;
 - final void display();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Final Classes: A way for Preventing Classes being extended

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword `final` as follows:

```
final class Marks
```

```
{ // members  
}
```

```
final class Student extends Person
```

```
{ // members  
}
```

- Any attempt to inherit these classes will cause an error.

Abstract Classes

- When we define a class to be “final”, it cannot be extended. In certain situation, we want to properties of classes to be always extended and used. Such classes are called Abstract Classes.
- An *Abstract* class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package:

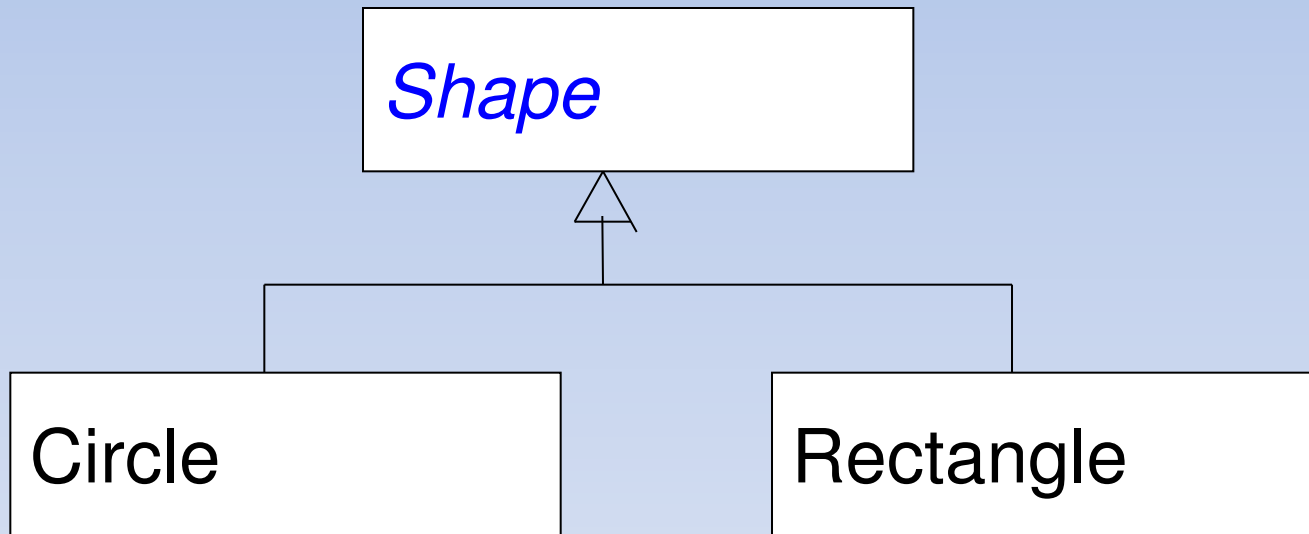
Abstract Class Syntax

```
abstract class ClassName
{
    ...
    ...
    abstract Type MethodName1();
    ...
    ...
    Type Method2()
    {
        // method body
    }
}
```

- **When a class contains one or more abstract methods, it should be declared as abstract class.**
- **The abstract methods of an abstract class must be defined in its subclass.**
- **We cannot declare abstract constructors or abstract static methods.**

Abstract Class -Example

- Shape is a abstract class.



The Shape Abstract Class

```
public abstract class Shape {  
    public abstract double area();  
    public void move() { // non-abstract  
method  
        // implementation  
    }  
}
```

- **Is the following statement valid?**
 - **Shape s = new Shape();**
- **No. It is illegal because the Shape class is an abstract class, which cannot be instantiated to create its objects.**

Abstract Classes

```
public Circle extends Shape {  
    protected double r;  
    protected static final double PI =3.1415926535;  
    public Circle() { r = 1.0; }  
    public double area() { return PI * r * r; }  
    ...  
}
```

```
public Rectangle extends Shape {  
    protected double w, h;  
    public Rectangle() { w = 0.0; h=0.0; }  
    public double area() { return w * h; }  
}
```

Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.

Summary

- If you do not want (properties of) your class to be extended or inherited by other classes, define it as a final class.
 - Java supports this is through the keyword “final”.
 - This is applied to classes.
- You can also apply the final to only methods if you do not want anyone to override them.
- If you want your class (properties/methods) to be extended by all those who want to use, then define it as an abstract class or define one or more of its methods as abstract methods.
 - Java supports this is through the keyword “abstract”.
 - This is applied to methods only.
 - Subclasses should implement abstract methods; otherwise, they cannot be instantiated.

Interface

- Since we have a good understanding of the **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship.
- The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.
- An interface in the Java programming language is an abstract type that is used to specify an interface that classes must implement.

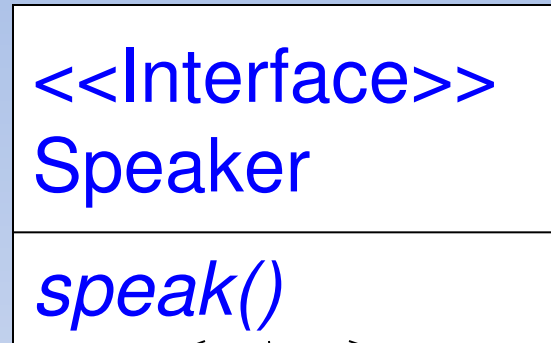
Interface

- Interface is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

Interface

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realising multiple inheritance in Java.

Interface Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {  
    // Constant/Final Variable Declaration  
    // Methods Declaration – only method body  
}
```

- Example:

```
interface Speaker {  
    public void speak( );  
}
```

Implementing Interfaces

- Interfaces are used like super-classes whose properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, ...]  
{  
    // Body of Class  
}
```

Implementing Interfaces Example

```
class Politician implements Speaker {  
    public void speak(){  
        System.out.println("Talk politics");  
    }  
}
```

```
class Priest implements Speaker {  
    public void speak(){  
        System.out.println("Religious Talks");  
    }  
}
```

```
class Lecturer implements Speaker {  
    public void speak(){  
        System.out.println("Talks Object Oriented Design and  
Programming!");  
    }  
}
```


Extending Interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1 {  
    // Body of InterfaceName2  
}
```

Inheritance and Interface Implementation

- A general form of interface implementation:

```
class ClassName extends SuperClass implements  
InterfaceName [, InterfaceName2, ...]  
{  
    // Body of Class  
}
```

- This shows a class can extended another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Interface Cont..

- Interfaces are declared using the **interface** keyword.
- Interface may only contain method signature and constant declarations (variable declarations that are declared to be both static and final).
- An interface may never contain method definitions.
- Interfaces cannot be instantiated, but rather are implemented.
- A class that implements an interface must implement all of the methods described in the interface, or be an abstract class.

Interface Cont..

- Object references in Java may be specified to be of an interface type.
- One benefit of using interfaces is that they simulate multiple inheritance.
- All classes in Java must have exactly one base class because multiple inheritance of classes is not allowed.
- A Java class may **implement** n number of interface.
- Interface may **extends** n number of interface.

Interface Example

```
public interface Animal {}  
  
public class Mammal implements Animal{  
}  
  
public class Dog extends Mammal{  
}
```

Interface Example

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
    public static void main(String args[]){

        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

This would produce the following result:

```
true
true
true
```

Comparable Interface

1. Provides an interface for comparing any two objects of same class.

2. General Form :

```
public interface Comparable
{
int compareTo(Object other );
}
```

```
public interface Comparable<T>
{
int compareTo(<T> other );
}
```



Note : other parameter should be type casted to the class type implementing Comparable interface

3. Collections. sort method can sort objects of any class that implements comparable interface.

4. By implementing this interface , programmers can implement the logic for comparing two objects of same class for less than, greater than or equal to.

Examples for Implementation

class BOX Implements Comparable

{

.....

.....

.....

public int compareTo(Object other)

{

BOX box = (BOX) other;

.....Logic for comparison

}

.....

}

class Student Implements Comparable

{

.....

.....

.....

public int compareTo(Object other)

{

Student std = (Student) other;

.....Logic for comparison

}

.....

}

Examples for Implementation

class BOX Implements Comparable<BOX>

```
{  
.....  
.....  
.....
```

public int compareTo(BOX other)

```
{  
.....Logic for comparison ....  
}  
.....  
}
```

class Student Implements Comparable<Student>

```
{  
.....  
.....  
.....
```

public int compareTo(Student other)

```
{  
.....Logic for comparison ....  
}  
.....  
}
```

Examples

class BOX implements Comparable

```
{
private double length;
private double width;
private double height;
BOX(double l,double b,double h)
{
length=l;width=b;height=h;
}
public double getLength() { return length;}
public double getWidth() { return width;}
public double getHeight() { return height;}
public double getArea()
{
return 2*(length*width + width*height+height*length);
}
public double getVolume()
{
return length*width*height;
}
```



Unparametrized Comparable

```
public int compareTo(Object other)
{
BOX b1 =(BOX) other;
if(this.getVolume() > b1.getVolume())
return 1;
if(this.getVolume() < b1.getVolume())
return -1;
return 0;
}
public String toString()
{
return "Length:"+length+" Width :"+width +" Height :"+height;
}
} // End of BOX class
```

```
import java.util.*;
class ComparableTest
{
public static void main(String[] args)
{
BOX[] box = new BOX[5];
box[0] = new BOX(10,8,6);
box[1] = new BOX(5,10,5);
box[2] = new BOX(8,8,8);
box[3] = new BOX(10,20,30);
box[4] = new BOX(1,2,3);
Arrays.sort(box);
for(int i=0;i<box.length;i++)
System.out.println(box[i]);
}
} // End of class
```

```
Import java.util.*;
class ComparableTest
{
public static void main(String[] args)
{
ArrayList box = new ArrayList();
box.add( new BOX(10,8,6));
box.add( new BOX(5,10,5));
box.add( new BOX(8,8,8));
box.add( new BOX(10,20,30));
box.add( new BOX(1,2,3));
Collections.sort(box);
Iterator itr = ar.iterator();
while(itr.hasNext())
{
BOX b =(BOX) itr.next();
System.out.println(b);
}
}
} // End of class
```

Problems With Comparable Interface

- Method `int compareTo(Object obj)` needs to be included in the base class itself.
- We can include only single ordering logic.
- Different order requires logic to be included and requires changes in the base class itself.
- Each type we need different order we need to change the code itself.

```
import java.util.*;
class Student implements Comparable
{
private String name;
private String idno;
private int age;
private String city;
.....
.....
public int compareTo(Object other)
{
Student std = (Student) other;
return this.name.compareTo(other.name);
}
public String toString()
{
Return "Name:"+name+"Id
No:"+idno+"Age:"+age;
}
} // End of class
```

```
Student[] students = new
Student[10];
.....
.....
.....
Arrays.sort(students);
for(int i=0 ; i<students.length;i++)
System.out.println(students[i]);
```

OUTPUT List sorted by Name

```
import java.util.*;
class Student implements Comparable
{
private String name;
private String idno;
private int age;
private String city;
.....
.....
public int compareTo(Object other)
{
Student std = (Student) other;
return this.idno.compareTo(other.idno);
}
public String toString()
{
Return "Name:"+name+"Id
No:"+idno+"Age:"+age;
}
} // End of class
```

```
Student[] students = new
Student[10];
.....
.....
.....
Arrays.sort(students);
for(int i=0 ; i<students.length;i++)
System.out.println(students[i]);
```

OUTPUT List sorted by IdNo

Comparator Interface

- Allows two objects to compare explicitly.
- Syntax :

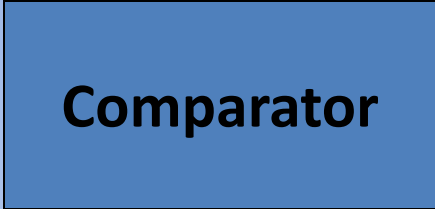
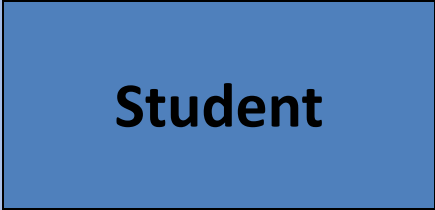
```
public interface Comparator
```

```
{  
int compare(Object O1, Object O2);           Unparametrized Comparator  
}
```

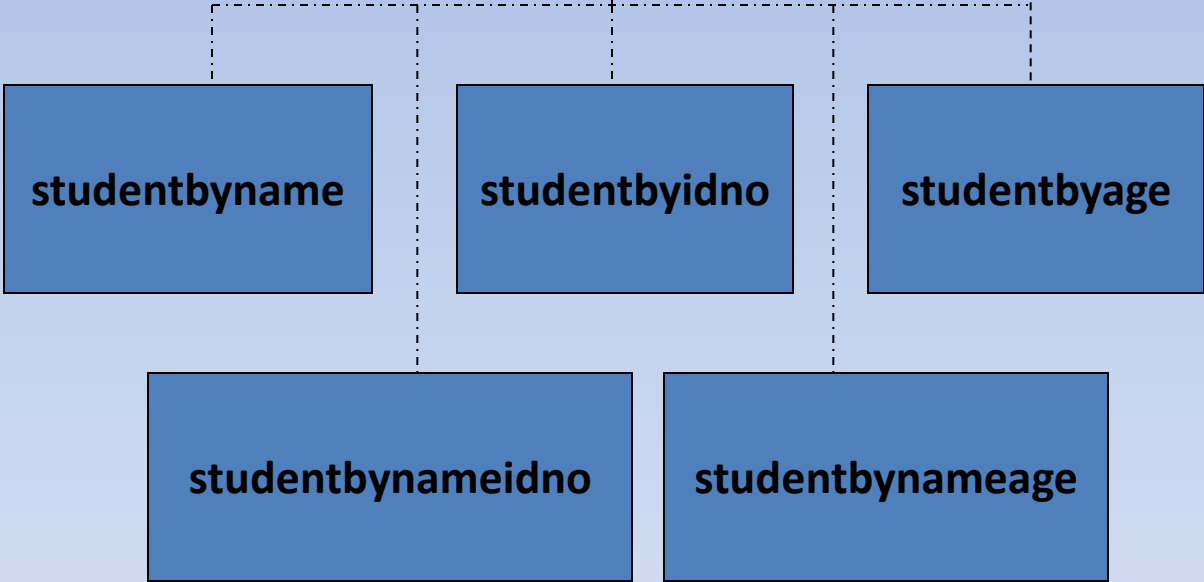
```
public interface Comparator<T>
```

```
{                                           Parametrized Comparator  
int compare(T O1, T O2);  
}
```

- Does not require change in the base class.
- We can define as many comparator classes for the base class.
- Each Comparator class implements Comparator interface and provides different logic for comparisons of objects.
- But as we are passing both parameters explicitly, we have to type cast both Object types to their base type before implementing the logic **OR**
Use the second form



```
class Student
{
private String name;
private String idno;
private int age;
private String city;
.....
.....
```



```
class studentbyname implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
return s1.getName().compareTo(s2.getName());
}
}
```

```
class studentbyidno implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
return s1.getIdNo().compareTo(s2.getIdNo());
}
}
```

```
class studentbyage implements comparator
```

```
{
```

```
public int compare(Object o1,Object o2)
```

```
{
```

```
Student s1 = (Student) o1;
```

```
Student s2 = (Student) o2;
```

```
if( s1.getAge() > s2.getAge() ) return 1;
```

```
if( s1.getAge() < s2.getAge() ) return -1;
```

```
return 0;
```

```
}
```

```
}
```

```
class studentbynameidno implements comparator
```

```
{
```

```
public int compare(Object o1,Object o2)
```

```
{
```

```
Student s1 = (Student) o1;
```

```
Student s2 = (Student) o2;
```

```
if( s1.getName().compareTo(s2.getName()) == 0)
```

```
return s1.getIdNo().compareTo(s2.getIdNo());
```

```
else
```

```
return s1.getName().compareTo(s2.getName());
```

```
}}
```

```
class studentbynameage implements comparator
{
public int compare(Object o1,Object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if( s1.getName().compareTo(s2.getName()) == 0)
return s1.getAge() - s2.getAge();
else
return s1.getName().compareTo(s2.getName());
}
}
```

```
import java.util.*;
class comparatorTest
{
public static void main(String args[])
{
Student[] students = new Student[5];
Student[0] = new Student("John","2000A1Ps234",23,"Pilani");
Student[1] = new Student("Meera","2001A1Ps234",23,"Pilani");
Student[2] = new Student("Kamal","2001A1Ps344",23,"Pilani");
Student[3] = new Student("Ram","2000A2Ps644",23,"Pilani");
Student[4] = new Student("Sham","2000A7Ps543",23,"Pilani");

// Sort By Name
Comparator c1 = new studentbyname();
Arrays.sort(students,c1);
for(int i=0;i<students.length;i++)
System.out.println(students[i]);
```

```
// Sort By Idno
```

```
c1 = new studentbyidno();
```

```
Arrays.sort(students,c1);
```

```
for(int i=0;i<students.length;i++)
```

```
System.out.println(students[i]);
```

```
// Sort By Age
```

```
c1 = new studentbyage();
```

```
Arrays.sort(students,c1);
```

```
for(int i=0;i<students.length;i++)
```

```
System.out.println(students[i]);
```

```
// Sort by Name & Idno
```

```
c1 = new studentbynameidno();
```

```
Arrays.sort(students,c1);
```

```
for(int i=0;i<students.length;i++)
```

```
System.out.println(students[i]);
```

```
// Sort by Name & Age
```

```
c1 = new studentbynameage();
```

```
Arrays.sort(students,c1);
```

```
for(int i=0;i<students.length;i++)
```

```
System.out.println(students[i]);
```

```
} // End of Main
```

```
} // End of test class.
```

*Exception in thread "main" java.lang.ClassCastException:
A*

*at java.util.Arrays.mergeSort(Arrays.java:1156)
at java.util.Arrays.sort(Arrays.java:1080)
at ctest.main(ctest.java:21)*

```
import java.util.*;  
class A  
{ int a;  
}  
class ctest  
{  
public static void main(String args[])  
{  
String[] names = {"OOP","BITS","PILANI"};  
Arrays.sort(names);  
int[] data = { 10,-45,87,0,20,21 };  
Arrays.sort(data);  
A[] arr = new A[5];  
arr[0] = new A();  
arr[1] = new A();  
arr[2] = new A();  
arr[3] = new A();  
arr[4] = new A();  
Arrays.sort(arr);  
}}}
```

Ok As String class
implements Comparable

Ok As Integer class
implements Comparable

NOT Ok as A class does
not implements
Comparable.

Unparametrized Comparator

```
import java.util.*;
class A implements Comparable
{
int a;
public int compareTo(Object other)
{
A a1 = (A) other;
if(this.a == a1.a ) return 0;
if(this.a < a1.a ) return -1;
return 1;
}
}
```

*Type cast Object type to Base
Type Before use*

```
class ctest
{
public static void main(String args[])
{
String[] names = {"OOP","BITS","PILANI"};
Arrays.sort(names);
int[] data = { 10,-45,87,0,20,21 } Will Work
Arrays.sort(data);
Will Work

A[] arr = new A[5];
arr[0] = new A();
arr[1] = new A();
arr[2] = new A();
arr[3] = new A();
arr[4] = new A();
Arrays.sort(arr);
}
} Will Work
```

Unparametrized Comparable


Parametrized Comparator

```
import java.util.*;
class A implements Comparable<A>
{
int a;
public int compareTo(A other)
{
// A a1 = (A) other; //No need of cast
if(this.a == other.a ) return 0;
if(this.a < other.a ) return -1;
return 1;
}
}
```

Parametrized Comparable

```
class ctest
{
public static void main(String args[])
{
String[] names = {"OOP","BITS","PILANI"};
Arrays.sort(names);
int[] data = { 10,-45,87,0,20,21 } Will Work
Arrays.sort(data);
Will Work
A[] arr = new A[5];
arr[0] = new A();
arr[1] = new A();
arr[2] = new A();
arr[3] = new A();
arr[4] = new A();
Arrays.sort(arr);
}
} Will Work
```

```
import java.util.*;
class BOX implements Comparable<BOX>
{
private double l,b,h;
// Overloaded Constructors
BOX(double a)
{ l=b=h=a;
}
BOX(double l,double b,double h)
{ this.l=l; this.b=b; this.h=h;
}
// Acessor Methods
public double getL()
{ return l;
}
public double getB()
{ return b;
}
public double getH()
{ return h;
}
```



**Parametrized
Comparable of
type BOX**

Cont....

```
// area() Volume() Methods
double area()
{
return 2*(l*b+b*h+h*l);
}
double volume()
{
return l*b*h;
}
// isEqual() method
boolean isEqual(BOX other)
{
if(this.area() == other.area()) return true;
return false;
/* OR
if(area() == other.area()) return true
return false;
*/
}
static boolean isEqual(BOX b1, BOX b2)
{
if(b1.area() == b2.area()) return true;
return false;
}
```

```
// compareTo method
public int compareTo(BOX other)
{
    if(area() > other.area()) return 1;
    if(area() < other.area()) return -1;
    return 0;
}
public String toString()
{
    String s1="length:"+l;
    String s2="width:"+b;
    String s3="area:"+h;
    String s4="Area:"+area();
    String s5="Volume:"+volume();
    return s1+s2+s3+s4+s5;
}
} // End of class BOX
```

```
class comparableTest10
{
public static void main(String args[])
{
ArrayList<BOX> boxes = new ArrayList<BOX>();
boxes.add(new BOX(10));
boxes.add(new BOX(20));
boxes.add(new BOX(10,6,8));
boxes.add(new BOX(4,6,10));
boxes.add(new BOX(10,12,14));
```

```
Iterator itr = boxes.iterator();
while(itr.hasNext())
System.out.println((BOX)itr.next());
```

Collections.sort(boxes);

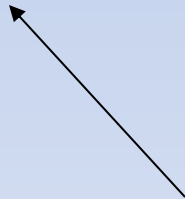
```
Iterator itr1 = boxes.iterator();
while(itr1.hasNext())
System.out.println((BOX)itr1.next());
}
}
```

Converting a Class To an Interface Type

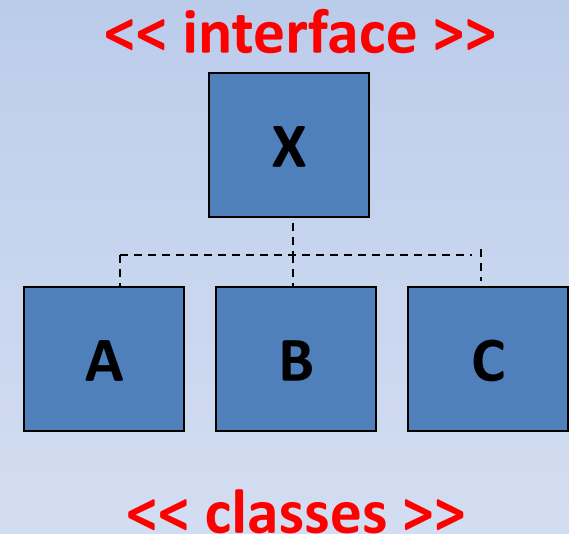
1. Interface acts as a super class for the implementation classes.
2. A reference variable belonging to type interface can point to any of the object of the classes implementing the interface.

```
A a1 = new A();
```

```
X x1 = a1;
```



Class to interface type Conversion



Converting an Interface to a class Type

```
X x1 = new A();
```

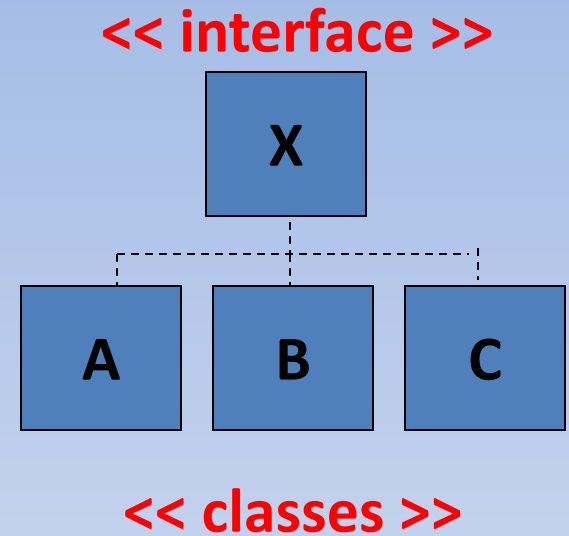
```
A a1 = (A) x1;
```

```
X x1 = new B();
```

```
B b1 = (B) x1;
```

```
X x1 = new C();
```

```
C c1 = (C) x1;
```



Interface to Class type Conversion

Comparator Example

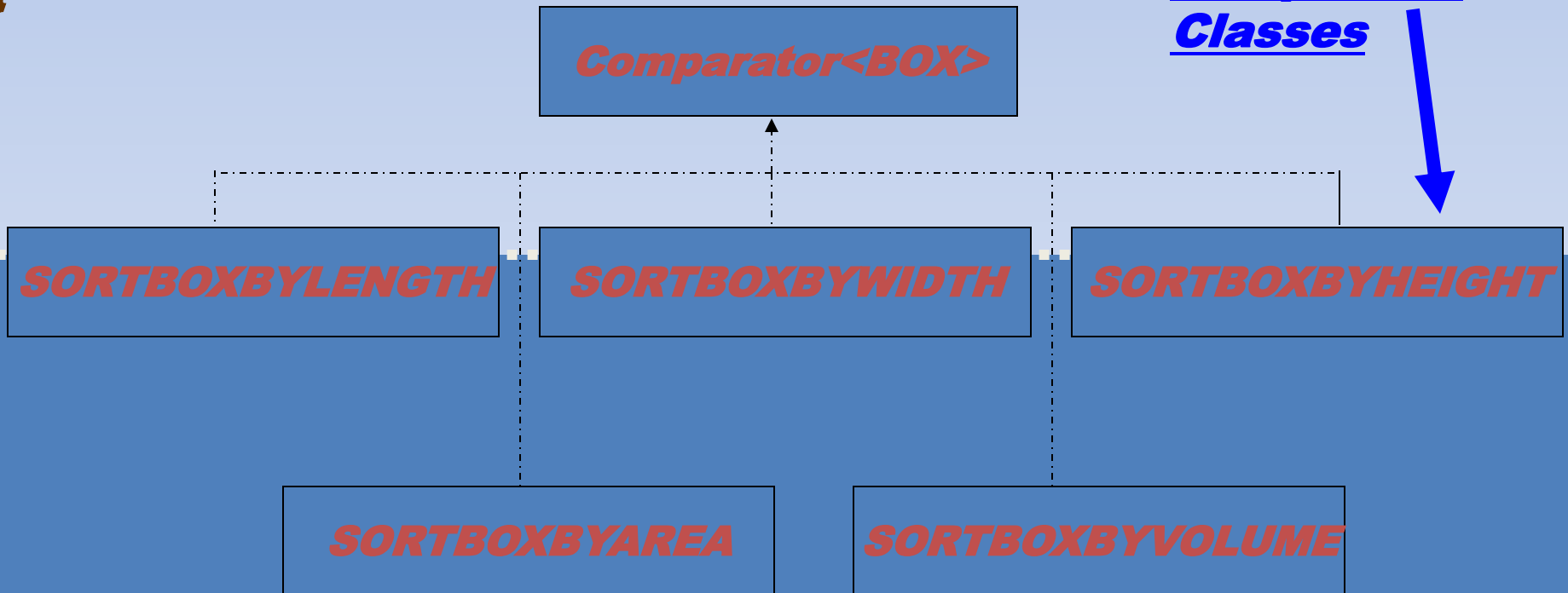
- *Supply comparators for BOX class so that BOX[] OR ArrayList<BOX> can be sorted by any of the following orders:*
 1. *Sort By Length Either in Ascending or descending order*
 2. *Sort By Width Either in Ascending or descending order*
 3. *Sort By Height Either in Ascending or descending order*
 4. *Sort By Area Either in Ascending or descending order*
 5. *Sort By Volume Either in Ascending or descending order*

BOX is base class whose references stored either in Arrays or in Any Collection class such as ArrayList, Vector or LinkedList Needs to be sorted

```
class BOX  
{  
.....instance fields  
.....instance methods  
.....  
}
```

BOX class does not implement any comparable or comparator interface

Comparator Classes



```
import java.util.*;
class BOX
{
private double l,b,h;
// Overloaded Constructors
BOX(double a)
{ l=b=h=a;
}
BOX(double l,double b,double h)
{
this.l=l;
this.b=b;
this.h=h;
}
// Acessor Methods
public double getL()
{ return l;
}
public double getB()
{ return b;
}
public double getH()
{ return h;
}
```

```
// area() Volume() Methods
double area()
{
return 2*(l*b+b*h+h*l);
}
double volume()
{
return l*b*h;
}
// isEqual() method
boolean isEqual(BOX other)
{
if(this.area() == other.area()) return true;
return false;
/* OR
if(area() == other.area()) return true
return false;
*/
}
```

Cont

```
static boolean isEqual(BOX b1, BOX b2)
{
if(b1.area() == b2.area()) return true;
return false;
}
public String toString()
{
String s1="length:"+l;
String s2="width:"+b;
String s3="area:"+h;
String s4="Area:"+area();
String s5="Volume:"+volume();
return s1+s2+s3+s4+s5;
}
} // End of class BOX
```

NOTE :

BOX class is base class
whose references needs to
be sorted. It does not
implement either
comparable or comparator
class

Cont

// Comparator class for Sorting by BOX references By length

```
class SORTBOXBYLENGTH implements Comparator<BOX>
{
private int order; // Defines Order of sorting 1 for Ascending -1 for Descending
SORTBOXBYLENGTH(boolean isAscending)
{
if(isAscending)
order =1;
else
order =-1;
}
public int compare(BOX b1,BOX b2)
{
if(b1.getL() > b2.getL()) return 1*order;
if(b1.getL() < b2.getL()) return -1*order;
return 0;
}
} // End of class
```

// Comparator class for Sorting by BOX references By Width

```
class SORTBOXBYWIDTH implements Comparator<BOX>  
{  
private int order;  
SORTBOXBYWIDTH(boolean isAscending)  
{  
if(isAscending)  
order =1;  
else  
order =-1;  
}  
public int compare(BOX b1,BOX b2)  
{  
if(b1.getB() > b2.getB()) return 1*order;  
if(b1.getB() < b2.getB()) return -1*order;  
return 0;  
}  
} // End of class
```

// Comparator class for Sorting by BOX references By Height

class SORTBOXBYHEIGHT implements Comparator<BOX>

```
{  
private int order;  
SORTBOXBYHEIGHT(boolean isAscending)  
{  
if(isAscending)  
order =1;  
else  
order =-1;  
}  
public int compare(BOX b1,BOX b2)  
{  
if(b1.getH() > b2.getH()) return 1*order;  
if(b1.getH() < b2.getH()) return -1*order;  
return 0;  
}  
} // End of class
```

// Comparator class for Sorting by BOX references By Area

```
class SORTBOXBYAREA implements Comparator<BOX>  
{  
  private int order;  
  SORTBOXBYAREA(boolean isAscending)  
  {  
    if(isAscending)  
      order =1;  
    else  
      order =-1;  
  }  
  public int compare(BOX b1,BOX b2)  
  {  
    if(b1.area() > b2.area()) return 1*order;  
    if(b1.area() < b2.area()) return -1*order;  
    return 0;  
  }  
} // End of class
```

// Comparator class for Sorting by BOX references By Volume

class SORTBOXBYVOLUME implements Comparator<BOX>

{

private int order;

SORTBOXBYVOLUME(boolean isAscending)

{

if(isAscending)

order =1;

else

order =-1;

}

public int compare(BOX b1,BOX b2)

{

if(b1.volume() > b2.volume()) return 1*order;

if(b1.volume() < b2.volume()) return -1*order;

return 0;

}

} // End of class


```
class comparatorTest
{
public static void main(String args[]) {
ArrayList<BOX> boxes = new ArrayList<BOX>();
boxes.add(new BOX(10));
boxes.add(new BOX(20));
boxes.add(new BOX(10,6,8));
boxes.add(new BOX(4,6,10));
boxes.add(new BOX(10,12,14));

// SORT BY LENTH ORDER:Ascending
Comparator<BOX> c1 = new SORTBOXBYLENGTH(true);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");

// SORT BY LENTH ORDER:Descending
c1 = new SORTBOXBYLENGTH(false);
Collections.sort(boxes,c1);
for(int i=0;i<boxes.size();i++)
System.out.println(boxes.get(i));
System.out.println("");
```

```
// SORT BY Volume ORDER:Ascending  
c1 = new SORTBOXBYVOLUME(true);  
Collections.sort(boxes,c1);  
for(int i=0;i<boxes.size();i++)  
System.out.println(boxes.get(i));  
System.out.println("");  
// SORT BY Volume ORDER:Descending  
c1 = new SORTBOXBYVOLUME(false);  
Collections.sort(boxes,c1);  
for(int i=0;i<boxes.size();i++)  
System.out.println(boxes.get(i));  
System.out.println("");  
}  
} // End of Main class
```

Exercise 1

- Suppose *C* is a class that implements interfaces *I* and *J*. Which of the following Requires a type cast?

C *c* =?

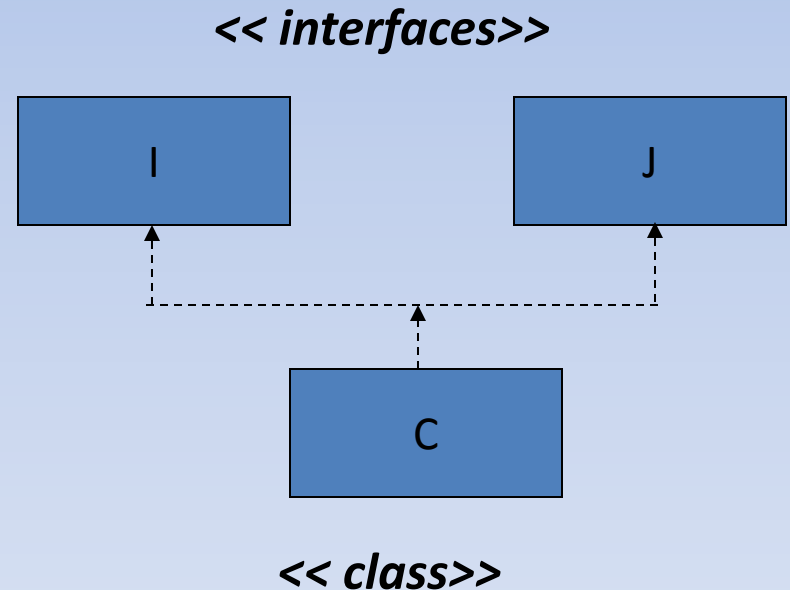
I *i* =?

J *j* =?

1. *c* = *i*

2. *j* = *c*

3. *i* = *j*



First *c* = (*C*) *i*

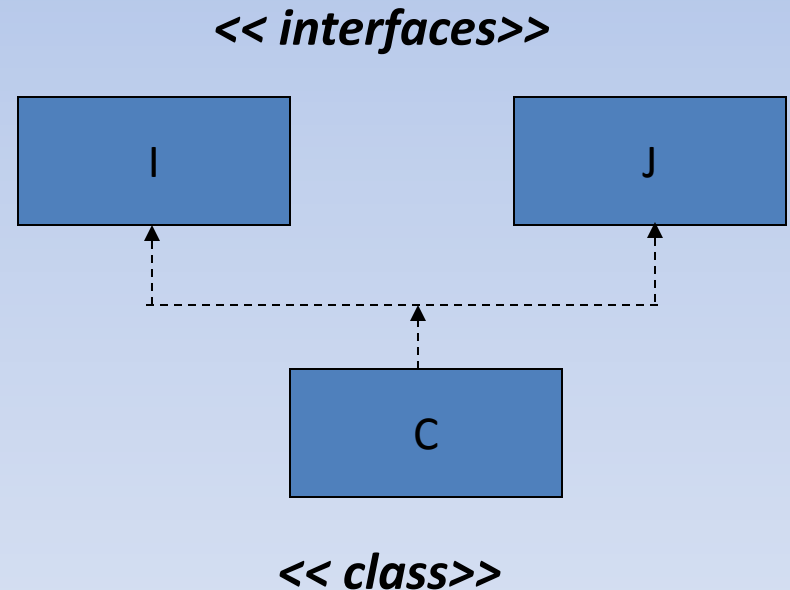
Third *i* = (*I*) *j*

Exercise 2

- Suppose *C* is a class that implements interfaces *I* and *J*. Which of the following will throw an Exception?

C `c = new C();`

1. `I i = c;`
2. `J j = (J) i;`
3. `C d = (C) i;`



Second

Exercise 3

- *Suppose the class Sandwich implements Editable interface. Which if the following statements are legal?*
 1. *Sandwich sub = new Sandwich();* **OK**
 2. *Editable e = sub;* **OK**
 3. *sub = e* ***Illegal***
 4. *sub = (Sandwich) e;* **OK**

Polymorphism

- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one IS-A test is considered to be polymorphic.
- In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

Polymorphism

- It is important to know that the only possible way to access an object is through a reference variable.
- A reference variable can be of only one type.
- Once declared, the type of a reference variable cannot be changed.
- The reference variable can be reassigned to other objects provided that it is not declared final.
- The type of the reference variable would determine the methods that it can invoke on the object.

Polymorphism

- A reference variable can refer to any object of its declared type or any subtype of its declared type.
- A reference variable can be declared as a class or interface type.

refer example from next slide.

Polymorphism

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

public class Main
{
    public static void main(String args[])
    {
        Deer d = new Deer();
        Animal a = d;
        Vegetarian v = d;
        Object o = d;
    }
}
```

Method Overriding Cont..

1. *Sub class can override the methods defined by the super class.*
2. *Overridden Methods in the sub classes should have same name, same signature , same return type and may have either the same or higher scope than super class method.*
3. *Java implements Run Time Polymorphism/ Dynamic Method Dispatch by Method Overriding. [Late Binding]*
4. *Call to Overridden Methods is Resolved at Run Time.*
5. *Call to a overridden method is not decided by the type of reference variable Rather by the type of the object where reference variable is pointing.*
6. *While Overriding a Method, the sub class should assign either same or higher access level than super class method.*

EXAMPLE METHOD OVERRIDING

```
class A
{
void show()
{
System.out.println("Hello This is show() in A");
} // End of show() Method
} // End of class A
```

*B class overrides show() method from
super class A*

```
class B extends A
{
void show()
{
System.out.println("Hello This is show() in B");
} // End of show() Method
} // End of class B
```

Call to show()
of A class

Call to show()
of B class

```
class override
{
public static void main(String args[])
{
// super class reference variable
// can point to sub class object
```

```
A a1 = new A();
a1.show();
```

```
a1 = new B();
a1.show();
```

```
}
```

```
}
```

```
class A
{
void show(int a)
{
System.out.println("Hello This is show()
in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is show()
in B");
}
}
```

Is this Method
Overriding

NO

```
class override1
{
public static void main(String args[])
{
/*
A a1 = new B();
a1.show(); */

A a1 = new A();
a1.show(10);

B b1 = new B();
b1.show(10);
b1.show(); }
}
```

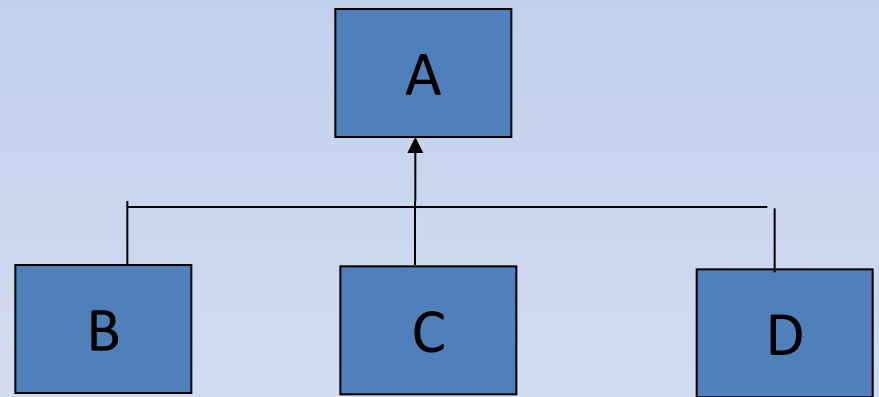
OUTPUT

Hello This is show() in A
Hello This is show() in A
Hello This is show() in B

Dynamic Method Dispatch

1. Super class reference variable can refer to a sub class object.
2. **Super class variable if refers to sub class object can call only overridden methods.**
3. Call to an overridden method is decided by the type of object referred to.

```
A a1 = new B();  
a1.show(); // call to show() of B  
a1 = new C();  
a1.show(); // call to show() of C  
a1 = new D();  
a1.show(); // call to show() of D
```



Assume show() Method is Overridden by sub classes

DYNAMIC METHOD DISPATCH

```
class A
{
void show()
{
System.out.println("Hello This is
show() in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is
show() in B");
}
}
```

```
class C extends A
{
void show()
{
System.out.println("Hello This
is show() in C");
}
}
class D extends A
{
void show()
{
System.out.println("Hello This
is show() in D");
}
}
```

CONTINUED.....

```
class override2
{
public static void main(String
args[])
{
A a1 = new A();
a1.show();
a1 = new B();
a1.show();
a1 = new C();
a1.show();
a1 = new D();
a1.show();
}
}
```

Hello This is show() in A

Hello This is show() in B

Hello This is show() in C

Hello This is show() in D

Method Overriding Cont..

1. *Sub class can override the methods defined by the super class.*
2. *Overridden Methods in the sub classes should have same name, same signature , same return type and may have either the same or higher scope than super class method.*
3. *Java implements Run Time Polymorphism/ Dynamic Method Dispatch by Method Overriding. [Late Binding]*
4. *Call to Overridden Methods is Resolved at Run Time.*
5. *Call to a overridden method is not decided by the type of reference variable Rather by the type of the object where reference variable is pointing.*
6. *While Overriding a Method, the sub class should assign either same or higher access level than super class method.*


```
class override3
{
public static void main(String args[])
{
A a1 = new B();
B b1 = (B) a1;

/*
A a1 = new B();
C c1 = (C) a1;

Exception in thread "main"
java.lang.ClassCastException: B
at override3.main(override3.java:39)
*/
}
}
```

Examples Overriding

```
class A
{
void show() { .... }
}
class B extends A
{
void show() { .... }
void show(int x) { ... }
void print() { ... }
}
```

```
A a1 = new B();
a1.show();      // Valid
// a1.show(10); // Invalid
//a1.print();   // Invalid
```

When a super class variable points to a sub class object, then it can only call overridden methods of the sub class.

```
class A
{
protected void show()
{
System.out.println("Hi");
}
}
class B extends A
{
void show()
{
System.out.println("Hi");
}
}
```

```
D:\Java1>javac AB.java
AB.java:10: show() in B cannot
override show() in A; attempting
to assign weaker
access privileges; was protected
void show()
      ^
```

1 error

IS THIS METHOD OVERRIDING

```
class A
{
private void show()
{
System.out.println("Hi");
}
}
class B extends A
{
int show()
{
System.out.println("Hi");
return 10;
}
}
```

NO

***CODE WILL COMPILE
& RUN SUCCESSFULLY***

What's Wrong Here

```
class A  
{  
static int show()  
{  
System.out.println("class A");  
return 0;  
}  
}
```

```
class B extends A  
{  
void show()  
{  
System.out.println("class B");  
}  
}
```

***sample.java:12: show() in B
cannot override show() in A;
overridden method is static
void show()***

^

1 error

Nested Classes

Java programming language allows you to define a class within another class

```
class OuterClass  
{ ...  
    class NestedClass { ... }  
}
```

Enclosing Class OR Outer Class

A nested class is a member of its enclosing class

Nested Class

1. Nested has access to other members of the enclosing class, even if they are declared private
2. Can be private, public, protected or friendly access

Nested Class Types

Static nested classes

1. **Static keyword applied for class declaration**
2. **Static nested class can use the instance fields/methods of the outer class only through object reference.**
3. **Static nested class can be accessed**

OuterClass.StaticNestedClass

4. ***To create an object for the static nested class, use this syntax:***

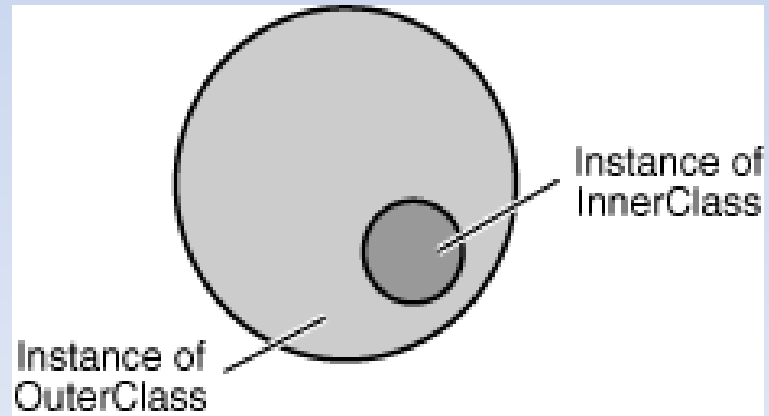
```
OuterClass.StaticNestedClass nestedObject = new  
OuterClass.StaticNestedClass();
```

Nested Class Types cont..

- Non-Static nested classes
 1. These nested classes do not have *static* keyword applied
 2. Non-Static nested class can use the instance fields/methods of the outer class directly.
 3. *To create an object for the non-static nested class, use this syntax:*

OuterClass.NestedClass nestedObject = Outerobjectreference.
new innerclass();

Inner class instance can only exist inside Outer class instance.



Example 1 [Non-static Nested Class]

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a)
;
}
```

Outer Class

***Call to
print() of
outer class***

```
class B
{
int b;
B(int b)
{
int c = b+10;
this.b = c;
}
void show()
{
print();
System.out.println("b="+b);
} // End of class B
} // End of class A
```

***Nested
class with
friendly
access***

Example 1 [Non-static Nested Class]

cont....

```
class innertest1
```

```
{
```

```
public static void main(String args[])
```

```
{
```

```
A a1 = new A(10);
```

```
A.B b1 = a1.new B(100);
```

```
b1.show();
```

```
}
```

```
}
```

Inner class Name

Outer class Reference

To create an inner class instance for non-static classes you need an outer class reference.

Inner class Reference

Outer class Name

If class B is Private then it is not visible in main().

A.B b1 = a1.new B(100); is WRONG/INVALID

Example 2

```
class A
{
private int a;
private int b=10;
```

Outer class

```
A(int a)
{
this.a=a;
}
```

Nested Inner class [Non-static Type]

```
class B
{
private int b;
B(int b)
{
this.b =b;
}
void show()
{
int b=20;
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("this.b="+this.b);
System.out.println("Outer b="+A.this.b);
}
} // End of B inner class
```

Instance Field of B

Outer Class A's a

Local b
B's instance Field b
A's instance Field b

```
void show()
{
B b1 = new B(30);
b1.show();
}
} // End of Outer class A
```

```
class innerTest
{
public static void main(String args[])
{
// Create an inner class B's instance
// Call show() method

// STEP 1
// Create an Outer Instance first
```

```
A a1 = new A(20);
A.B b1 = a1.new B(-30);
b1.show();
```

```
a=20
b=20
this.b=-30
Outer b=10
```

```
// inner class object instantiation thru anonymous
outer
// reference
```

```
A.B b2 = new A(30).new B(-40);
b2.show();
```

```
a=30
b=20
this.b=-40
Outer b=10
```

```
}
}
```

Static Inner class / Static Nested class Example

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a
);
}
```

Static nested class can refer to outer members only through outer reference

static class B

```
{
int b;
B(int b)
{
int c = b+10;
this.b = c;
}
void show()
{
// print(); INVALID
A a1 = new A(10);
a1.print();
System.out.println("b="+b);
}
} // End of class B
} // End of class A
```

Static inner class

Example cont....

```
class innertest10
{
public static void main(String args[])
{
A.B b1 = new A.B(100);
b1.show();
}
}
```

***Instance of static Inner
class***

Static Nested class Example 2

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
public void show()
{
System.out.println("a="+a);
display();
}
public static void display()
{
System.out.println("b="+b);
}
```


Example 2 cont....

```
static class B
{
private int a;
protected static int b=100;
B(int a)
{
this.a=a;
}
void show()
{
// A.this.show(); // Won't work show() is non-static in outer
display(); // Will work as method is static in outer
System.out.println("a="+a);
// System.out.println("a="+A.this.a);
// Won't work a is non-static in outer
System.out.println("b="+b); // Will refer to its own b
System.out.println("A'sb="+A.b); // will refer to outer class B

new A(40).show();
// This is how you can call non static methods of outer

}
} // End of inner class B
} // End of class A
```

Example 2 cont....

```
class innerTest1
{
public static void main(String
args[])
{
A.B b1 = new A.B(-30);
b1.show();
}
}
```

D:\jdk1.3\bin>java innerTest1

b=10

a=-30

b=100

A'sb=10

a=40

b=10

Local Inner classes [Classes Within method body]

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
    class B
    {}
}
}
```

Class declared within a method body.

Here method is show()

Local inner classes Can not be declared as public,private or protected

1. **Class B is visible only in method show().**
2. **It can be used within this show() method only**
3. **Local inner classes can only use final variables from its enclosing method.**
4. **However inner classes can refer to its fields of enclosing class.**

```

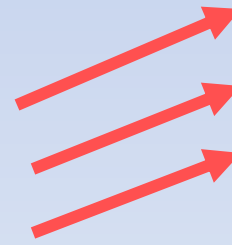
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
int x=10;
    class B
    {
private int b;
B(int b)
{
this.b=b;
}
void display()
{
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("x="+x);
}
} // End of class B
} // End of show() method
} // End of A class

```

```

D:\jdk1.3\bin>javac
innerTest2.java
innerTest2.java:23: local
variable x is accessed from
within inner class;
to be declared final
System.out.println("x="+x);
                        ^
1 error

```



Reference for A's a
Reference for B's b
Reference is wrong / erroneous
'x' is local variable inside the local
method. Local classes can use only
final fields from enclosing method

```
class innertest
{
public static void
main(String args[])
{
final int a1=10;
```

```
new A(20).show();
print();
} // End of main
static void print()
{
/*
A a1 = new A(30);
a1.show();
*/
System.out.println("Hello"
);
}
}
```

```
class A
{
private int a;
private int b;
int c;
A(int a)
{
this.a =a;
b = a+20;
c = a+40;
}
void show()
{
System.out.println("a1="+a1) ;
System.out.println("a="+a) ;
System.out.println("b="+b) ;
System.out.println("c="+c) ;
}
} //End of A
```

OUTPUT

```
E:\oop>java innertest
```

```
a1=10
```

```
a=20
```

```
b=40
```

```
c=60
```

```
Hello
```

Anonymous Inner classes

- *Another category of local inner classes*
- *Classes without any name i.e classes having no name*
- *Can either implements an interface or extends a class.*
- *Can not have more than one instance active at a time.*
- *Whole body of the class is declared in a single statement ending with ;*

Cont...

- Syntax [If extending a class]

```
[variable_type_superclass =] new superclass_name() {  
    // properties and methods  
} [;]
```

- Syntax [If implementing an interface]

```
[variable_type_reference =] new reference_name() {  
    // properties and methods  
} [;]
```


Anonymous Inner Class Example

```
class A  
{  
private int a;  
A(int a)  
{  
this.a =a;  
}  
void show()  
{  
System.out.println("a="+a);  
} // End of show()  
// End of class A
```

```
class innertest1
{
public static void main(String args[])
{
```

Anonymous inner class extending super class A

```
A a1 = new A(20) {
    public void show()
    {
        super.show();
        System.out.println("Hello");
    }
    public void display()
    {
        System.out.println("Hi");
    }
};
```

```
a1.show();
// a1.display();
}
```



Calling show from inner class

```
interface X
{
int sum(int a,int b);
int mul(int x,int y);
}
class innertest2
{
public static void main(String args[])
{
```

Anonymous inner class implementing an interface

```
X x1 = new X()
{
    public int sum(int a,int b)
    {
        return a+b;
    }
    public int mul(int a,int b)
    {
        return a*b;
    }
};
```

```
System.out.println(x1.sum(10,20));
System.out.println(x1.mul(10,20));
} // End of main
} // End of innertest2
```

Wrapper Classes

- Java uses primitive types, such as int, char, double to hold the basic data types supported by the language.
- Sometimes it is required to create an object representation of these primitive types.
- These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class.

- To satisfy this need, Java provides classes that correspond to each of the primitive types. Basically, these classes encapsulate, or wrap, the primitive types within a class.
- Thus, they are commonly referred to as type wrapper. Type wrapper are classes that encapsulate a primitive type within an object.
- The wrapper types are Byte, Short, Integer, Long, Character, Boolean, Double, Float.
- These classes offer a wide array of methods that allow to fully integrate the primitive types into Java's object hierarchy.

- Wrapper Classes are based upon the well-known software engineering design pattern called the Wrapper pattern.
 - A design pattern is a template solution to a common problem. It describes the problem and identifies the recommended solution(s) to that problem.
 - Because design patterns deal with common problems, they are often quite abstract!
- The problem:e.g. we want to store an int in a Vector, but Vectors do not accept primitive types.
- The Solution:
 - We create another class that wraps the underlying class/type and provides an appropriate interface for the client.
 - e.g. we create an Integer class that subclasses Object (as all classes do), allowing us to store wrapped int's in Vectors.

Example that will *not* work

```
import java.util.Vector;
```

```
public class MyApp {
```

```
    public static void main(String[] args) {
```

```
        int myValue = 2;
```

```
        Vector myVector = new Vector();
```

```
        myVector.addElement(myValue);
```

```
        for (int x=0; x < myVector.size(); x++) {
```

```
            System.out.println("Value: " +  
myVector.get(x));
```

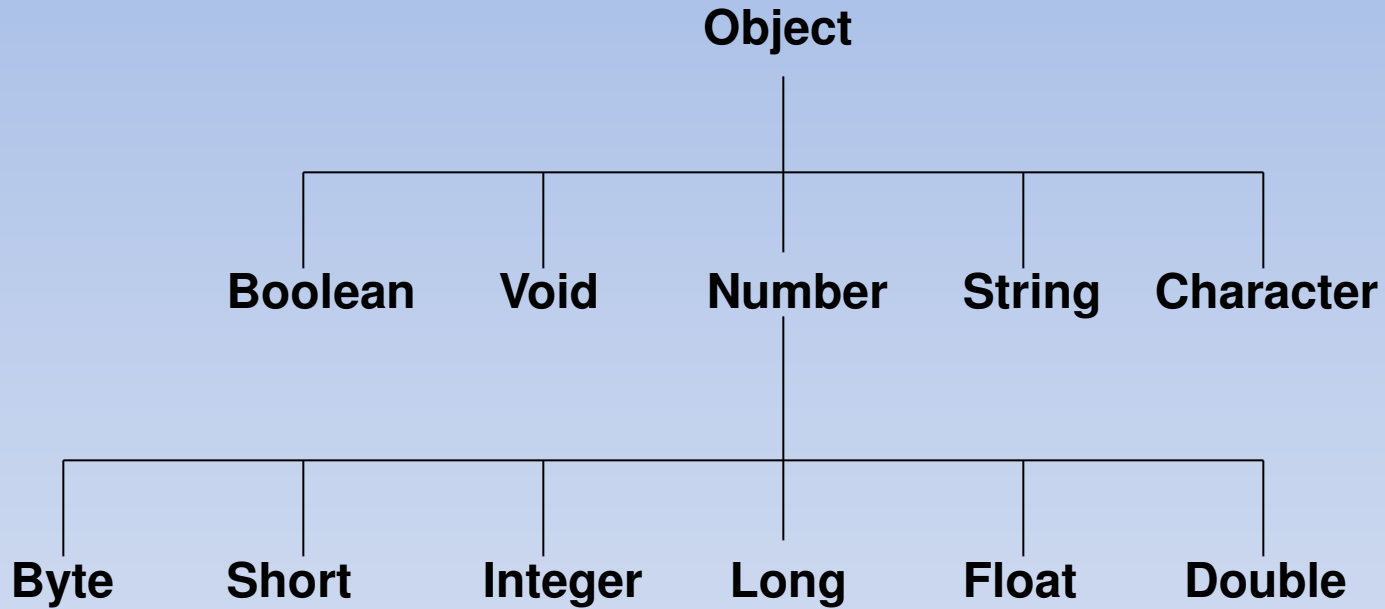
```
        }
```

```
    }
```

```
}
```

The compiler detects
an error here

Java Wrapper Classes



Primitives & Wrappers

- Java has a *wrapper* class for each of the eight primitive data types:

Primitive Type	Wrapper Class	Primitive Type	Wrapper Class
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short

Converting Primitive to Object

- Constructor calling Conversion Action
- Integer IntVal = new Integer(i);
Primitive integer to Integer object
- Float FloatVal = new Float(f);
Primitive float to Float object
- Double DoubleVal = new Double(d);
Primitive double to Double object
- Long LongVal new Long(l);
Primitive long to Long object

Converting Numeric Strings to Primitive

- `int i = Integer.parseInt(str);`
Converts String `str` into primitive integer `i`
- `long l = Long.parseLong(str);`
Converts String `str` into primitive long `l`
- `Double d = Double.parseDouble(str);`
Converting String to primitive double

Note:

`parseInt()` and `parseLong()` ... methods throw a `NumberFormatException` if the value of the `str` does not represent an integer.

Some Useful Methods

- The Java Wrapper Classes include various useful methods:
 - Getting a value from a String

e.g. `int value = Integer.parseInt("33");`

sets value to 33.

- Converting a value to a String:

e.g. `String str = Double.toString(33.34);`

sets str to the String "33.34".

- Getting a wrapper class instance for a value:

e.g. `Integer obj = Integer.getInteger("12");`

creates a new Integer object with value 12 and makes obj refer to that object.

Reading a Double

```
import java.io.*;

class MyProgram {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

        String line = null;

        System.out.println("Input Something:");
        try {
            line = in.readLine();
        } catch (IOException ie) {
            System.out.println("Exception caught: " + ie);
        }

        try {
            double value = Double.parseDouble(line);
            System.out.println("Value: " + value);
        } catch (NumberFormatException nfe) {
            System.out.println("You didn't enter a double number");
        }
    }
}
```

Use of the Wrapper Classes

- Java's *primitive* data types (boolean, int, etc.) are not classes.
- Wrapper classes are used in situations where objects are required, such as for elements of a Collection:

```
List<Integer> a = new ArrayList<Integer>();  
methodRequiringListOfIntegers(a);
```

Value => Object: Wrapper Object Creation

- *Wrapper.valueOf()* takes a value (or string) and returns an object of that class:

```
Integer i1 = Integer.valueOf(42);  
Integer i2 = Integer.valueOf("42");
```

```
Boolean b1 = Boolean .valueOf(true);  
Boolean b2 = Boolean .valueOf("true");
```

```
Long n1 = Long.valueOf(42000000L);  
Long n1 = Long.valueOf("42000000L");
```

Object => Value

- Each wrapper class Type has a method `intValue` to obtain the object's value:

```
Integer i1 = Integer.valueOf(42);  
Boolean b1 = Boolean.valueOf("false");  
System.out.println(i1.intValue());  
System.out.println(b1.intValue());
```

=>

42

false

String => value

- The Wrapper class for each primitive *type* has a method `parseType()` to parse a string representation & return the literal value.

```
Integer.parseInt("42")           => 42  
Boolean.parseBoolean("true")    => true  
Double.parseDouble("2.71")      => 2.71  
//...
```

- Common use: Parsing the arguments to a program:

Parsing argument lists

```
// Parse int and float program args.  
public parseArgs(String[] args) {  
    for (int i = 0; i < args.length; i++) {  
        try {  
            ...println(Integer.parseInt(args[i]));  
        } catch (Exception e) {  
  
        }  
    }  
}
```

Sample values:

```
boolObj new Boolean(Boolean.TRUE);  
charObj = new Character('a');  
byteObj = new Byte("100");  
shortObj = new Short("32000");  
intObj = new Integer(2000000);  
longObj = new Long(5000000000000000000L);  
floatObj = new Float(1.42);  
doubleObj = new Double(1.42);  
  
printWrapperInfo(); //method to print objects above
```

Sample values (output from previous slide):

=>

For Boolean & Character Wrappers:

Boolean:true

Character:a

For Number wrappers:

Byte:100

Short:32000

Integer:2000000

Long:5000000000000000000

Float:1.42

Double:1.42

Each Number Wrapper has a MAX_VALUE constant:

```
byteObj = new Byte(Byte.MAX_VALUE);  
shortObj = new Short(Short.MAX_VALUE);  
intObj = new Integer(Integer.MAX_VALUE);  
longObj = new Long(Long.MAX_VALUE);  
floatObj = new Float(Float.MAX_VALUE);  
doubleObj = new Double(Double.MAX_VALUE);  
  
printNumValues("MAXIMUM NUMBER VALUES:");
```

MAX values (output from previous slide):

=>

Byte:127

Short:32767

Integer:2147483647

Long:9223372036854775807

Float:3.4028235E38

Double:1.7976931348623157E308

Byte Example

- The Byte class encapsulates a byte value. It defines the constants `MAX_VALUE` and `MIN_VALUE` and provides these constructors:
 - `Byte(byte b)`
`Byte(String str)`
- Here, `b` is a byte value and `str` is the string equivalent of a byte value.

Byte Example

```
import java.util.*;
public class Byte_Demo
{
    public static void main(String args[])
    {
        Byte b1 = new Byte((byte)120);
        for(int i = 125; i<=135; i++)
        {
            Byte b2 = new Byte((byte)i);
            System.out.println("b2 = " + b2);
        }
        System.out.println("b1 Object = " + b1);
        System.out.println("Minimum Value of Byte = " + Byte.MIN_VALUE);
        System.out.println("Maximum Value of Byte = " + Byte.MAX_VALUE);
        System.out.println("b1* 2 = " + b1*2);
        System.out.println("b1* 2 = " + b1.byteValue()*2);
        Byte b3 = new Byte("120");
        System.out.println("b3 Object = " + b3);
        System.out.println("(b1==b3)? " + b1.equals(b3));
        System.out.println("(b1.compareTo(b3)? " + b1.compareTo(b3));
        /*Returns 0 if equal. Returns -1 if b1 is less than b3 and 1 if b1 is
        greater than 1*/
    }
}
```


Short Example

- The Short class encapsulates a short value. It defines the constants MAX_VALUE and MIN_VALUE and provides the following constructors:
Short(short s)
Short(String str)

Short Example

```
import java.util.*;
public class Short_Demo
{
    public static void main(String args[])
    {
        Short s1 = new Short((short)2345);
        for(int i = 32765; i<=32775; i++)
        {
            Short s2 = new Short((short)i);
            System.out.println("s2 = " + s2);
        }
        System.out.println("s1 Object = " + s1);
        System.out.println("Minimum Value of Short = " + Short.MIN_VALUE);
        System.out.println("Maximum Value of Short = " + Short.MAX_VALUE);
        System.out.println("s1* 2 = " + s1.shortValue()*2);
        Short s3 = new Short("2345");
        System.out.println("s3 Object = " + s3);
        System.out.println("(s1==s3)? " + s1.equals(s3));
        Short s4 = Short.valueOf("10", 16);
        System.out.println("s4 Object = " + s4);
    }
}
```

Integer Example

- The Integer class encapsulates an integer value. This class provides following constructors:
 - Integer(int i)
 - Integer(String str)
- Here, i is a simple int value and str is a String object.

Integer Example

```
import java.util.*;
public class Int_Demo
{
    public static void main(String args[])
    {
        Integer i1 = new Integer(12);
        System.out.println("I1 = " + i1);
        System.out.println("Binary Equivalent = " + Integer.toBinaryString(i1));
        System.out.println("Hexadecimal Equivalent = " + Integer.toHexString(i1));
        System.out.println("Minimum Value of Integer = " + Integer.MIN_VALUE);
        System.out.println("Maximum Value of Integer = " + Integer.MAX_VALUE);
        System.out.println("Byte Value of Integer = " + i1.byteValue());
        System.out.println("Double Value of Integer = " + i1.doubleValue());
        Integer i2 = new Integer(12);
        System.out.println("i1==i2 " + i1.equals(i2));
        System.out.println("i1.compareTo(i2) = " + i2.compareTo(i1));
        // Compareto - if it is less than it returns -1 else 1, if equal it return 0.
        Integer i3 = Integer.valueOf("11", 16);
        System.out.println("i3 = " + i3);
    }
}
```

Character Example

- The Character class encapsulates a char value. This class provides the following constructor.
- `Character(char ch)`
- Here, `c` is a char value. `charValue()` method returns the char value that is encapsulated by a Character object and has the following form:
- `char charValue()`

Character Example

```
import java.util.*;

public class Char_Demo
{
    public static void main(String args[])
    {
        Character c1 = new Character('m');
        char c2 = 'O';
        System.out.println("Object C1 = " + c1);
        System.out.println("char value of Character Object = " + c1.charValue());
        System.out.println(c2 + " is defined character set ? " + Character.isDefined(c2));
        System.out.println("c2 is digit = " + Character.isDigit(c2));
        System.out.println("c2 is lowercase character = " + Character.isLowerCase(c2));
        System.out.println("c2 is uppercase character = " + Character.isUpperCase(c2));
    }
}
```

Boolean Example

- The Boolean class encapsulates a Boolean value. It defines FALSE and TRUE constants.
- This class provides following constructors:
 - Boolean(Boolean b)
 - Boolean(String str)
- Here, b is a Boolean value and str is the string equivalent of a Boolean value.
- The methods associated with Boolean Class are as follows:
 1. Boolean booleanValue()
 2. Boolean equals(Boolean b)
 3. String toString(Boolean b)

Boolean Example

```
import java.util.*;

public class Boolean_Demo
{
    public static void main(String args[])
    {
        Boolean b1 = new Boolean(true);
        Boolean b2 = new Boolean(false);
        System.out.println("Object B1 = " + b1);
        System.out.println("Object B2 = " + b2);
        Boolean b3 = new Boolean("true");
        Boolean b4 = new Boolean("false");
        System.out.println("Object B3 = " + b3);
        System.out.println("Object B4 = " + b4);
        System.out.println("Boolean Value = " + b1.booleanValue());
        System.out.println("(b1==b2)? " + b1.equals(b2));
        String S1 = b1.toString();
        System.out.println("String S1 " + S1);
    }
}
```


The Object Class

- Every java class has Object as its superclass and thus inherits the Object methods.
- Object is a non-abstract class
- Many Object methods, however, have implementations that aren't particularly useful in general
- In most cases it is a good idea to override these methods with more useful versions.
- In other cases it is **required** if you want your objects to correctly work with other class libraries.

Clone Method

- Recall that the “=” operator simply copies Object **references**. e.g.,
 - >> Student s1 = new Student(“Smith”, Jim, 3.13);
 - >> Student s2 = s1;
 - >> s1.setName(“Sahil”);
 - >> System.out.println(s2.getName());OP:- Sahil
- What if we want to actually make a copy of an Object?
- Most elegant way is to use the clone() method inherited from Object.
 - Student s2 = (Student) s1.clone();

About clone() method

- First, note that the clone method is *protected* in the Object class.
- This means that it is *protected* for subclasses as well.
- Hence, it cannot be called from within an Object of another class and package.
- To use the clone method, you must override in your subclass and upgrade visibility to *public*.
- Also, any class that uses clone must implement the *Cloneable* interface.
- This is a bit different from other interfaces that we've seen.
- There are no methods; rather, it is used just as a marker of your intent.
- The method that needs to be implemented is inherited from Object.

Issue With clone() method

- Finally, clone throws a CloneNotSupportedException.
- This is thrown if your class is not marked Cloneable.
- This is all a little odd but you must handle this in subclass.

Steps For Cloning

- To reiterate, if you would like objects of class C to support cloning, do the following:
 - implement the Cloneable interface
 - override the clone method with public access privileges
 - call `super.clone()`
 - Handle `CloneNotSupportedException` Exception.
- This will get you default cloning means shallow copy.

Shallow Copies With Cloning

- We haven't yet said what the default clone() method does.
- By default, clone makes a *shallow copy* of all iv's in a class.
- *Shallow copy* means that all native datatype iv's are copied in regular way, but iv's that are objects are not recursed upon – that is, references are copied.
- This is not what you typically want.
- Must override clone explicitly for Deep Copying.

Deep Copies

- For *deep copies* that recurse through the object iv's, you have to do some more work.
- `super.clone()` is first called to clone the first level of iv's.
- Returned cloned object's object fields are then accessed one by one and clone method is called for each.
- See `DeepClone.java` example

Additional clone() properties

- Note that the following are typical, but not strictly required:
 - `x.clone() != x;`
 - `x.clone().getClass() == x.getClass();`
 - `x.clone().equals(x);`
- Finally, though no one really cares, `Object` does not support `clone()`;

toString() method

- The Object method

```
String toString();
```

is intended to return a readable textual representation of the object upon which it is called. This is great for debugging!

- Best way to think of this is using a print statement. If we execute:

```
System.out.println(someObject);
```

we would like to see some meaningful info about someObject, such as values of iv's, etc.

default toString()

- By default toString() prints total garbage that no one is interested in
`getClass().getName() + '@' + Integer.toHexString(hashCode())`
- By convention, print simple formatted list of field names and values (or some important subset).
- The intent is not to overformat.
- Typically used for debugging.
- Always override toString()!

equals() method

- Recall that boolean == method compares when applied to object compares references.
- That is, two object are the same if the point to the same memory.
- Since java does not support operator overloading, you cannot change this operator.
- However, the equals method of the Object class gives you a chance to more meaningful compare objects of a given class.

equals method, cont

- By default, `equals(Object o)` does exactly what the `==` operator does – compare object references.
- To override, simply override method with version that does more meaningful test, ie compares iv's and returns true if equal, false otherwise.
- See `Equals.java` example in course notes.

equals subtleties

- As with any method that you override, to do so properly you must obey contracts that go beyond interface matching.
- With equals, the extra conditions that must be met are discussed on the next slide:

equals contract

- ◆ It is *reflexive*: for any reference value x , $x.equals(x)$ should return true.
- ◆ It is *symmetric*: for any reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.
- ◆ It is *transitive*: for any reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true.
- ◆ It is *consistent*: for any reference values x and y , multiple invocations of $x.equals(y)$ consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- ◆ For any non-null reference value x , $x.equals(null)$ should return false.

hashCode() method

- Java provides all objects with the ability to generate a hash code.
- By default, the hashing algorithm is typically based on an integer representation of the java address.
- This method is supported for use with `java.util.Hashtable`
- Will discuss `Hashtable` in detail during Collections discussion.

Rules for overriding hashCode

- ◆ Whenever invoked on the same object more than once, the hashCode method must return the same integer, provided no information used in equals comparisons on the object is modified.
- ◆ If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- ◆ It is *not* required that if two objects are unequal according to the [equals\(java.lang.Object\)](#) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

finalize() method

- Called as final step when Object is no longer used, just before garbage collection
- Object version does nothing
- Since java has automatic garbage collection, finalize() does not need to be overridden reclaim memory.
- Can be used to reclaim other resources – close streams, database connections, threads.
- However, it is strongly recommended *not* to rely on this for scarce resources.
- Be explicit and create own dispose method.

Polymorphism

- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one IS-A test is considered to be polymorphic.
- In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

Polymorphism

- It is important to know that the only possible way to access an object is through a reference variable.
- A reference variable can be of only one type.
- Once declared, the type of a reference variable cannot be changed.
- The reference variable can be reassigned to other objects provided that it is not declared final.
- The type of the reference variable would determine the methods that it can invoke on the object.

Polymorphism

- A reference variable can refer to any object of its declared type or any subtype of its declared type.
- A reference variable can be declared as a class or interface type.

refer example from next slide.

Polymorphism

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

public class Main
{
    public static void main(String args[])
    {
        Deer d = new Deer();
        Animal a = d;
        Vegetarian v = d;
        Object o = d;
    }
}
```

Method Overriding Cont..

1. *Sub class can override the methods defined by the super class.*
2. *Overridden Methods in the sub classes should have same name, same signature , same return type and may have either the same or higher scope than super class method.*
3. *Java implements Run Time Polymorphism/ Dynamic Method Dispatch by Method Overriding. [Late Binding]*
4. *Call to Overridden Methods is Resolved at Run Time.*
5. *Call to a overridden method is not decided by the type of reference variable Rather by the type of the object where reference variable is pointing.*
6. *While Overriding a Method, the sub class should assign either same or higher access level than super class method.*

EXAMPLE METHOD OVERRIDING

```
class A
{
void show()
{
System.out.println("Hello This is show() in A");
} // End of show() Method
} // End of class A
```

*B class overrides show() method from
super class A*

```
class B extends A
{
void show()
{
System.out.println("Hello This is show() in B");
} // End of show() Method
} // End of class B
```

Call to show()
of A class

Call to show()
of B class

```
class override
{
public static void main(String args[])
{
// super class reference variable
// can point to sub class object
```

```
A a1 = new A();
a1.show();
```

```
a1 = new B();
a1.show();
```

```
}
```

```
}
```

```
class A
{
void show(int a)
{
System.out.println("Hello This is show()
in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is show()
in B");
}
}
```

Is this Method
Overriding

NO

```
class override1
{
public static void main(String args[])
{
/*
A a1 = new B();
a1.show(); */

A a1 = new A();
a1.show(10);

B b1 = new B();
b1.show(10);
b1.show(); }
}
```

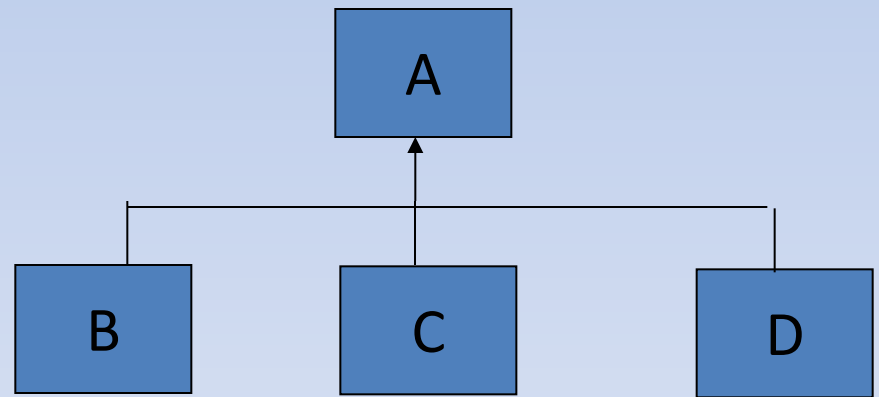
OUTPUT

Hello This is show() in A
Hello This is show() in A
Hello This is show() in B

Dynamic Method Dispatch

1. Super class reference variable can refer to a sub class object.
2. **Super class variable if refers to sub class object can call only overridden methods.**
3. Call to an overridden method is decided by the type of object referred to.

```
A a1 = new B();  
a1.show(); // call to show() of B  
a1 = new C();  
a1.show(); // call to show() of C  
a1 = new D();  
a1.show(); // call to show() of D
```



Assume show() Method is Overridden by sub classes

DYNAMIC METHOD DISPATCH

```
class A
{
void show()
{
System.out.println("Hello This is
show() in A");
}
}
class B extends A
{
void show()
{
System.out.println("Hello This is
show() in B");
}
}
```

```
class C extends A
{
void show()
{
System.out.println("Hello This
is show() in C");
}
}
class D extends A
{
void show()
{
System.out.println("Hello This
is show() in D");
}
}
```

CONTINUED.....

```
class override2
{
public static void main(String
args[])
{
A a1 = new A();
a1.show();
a1 = new B();
a1.show();
a1 = new C();
a1.show();
a1 = new D();
a1.show();
}
}
```

Hello This is show() in A

Hello This is show() in B

Hello This is show() in C

Hello This is show() in D

Method Overriding Cont..

1. *Sub class can override the methods defined by the super class.*
2. *Overridden Methods in the sub classes should have same name, same signature , same return type and may have either the same or higher scope than super class method.*
3. *Java implements Run Time Polymorphism/ Dynamic Method Dispatch by Method Overriding. [Late Binding]*
4. *Call to Overridden Methods is Resolved at Run Time.*
5. *Call to a overridden method is not decided by the type of reference variable Rather by the type of the object where reference variable is pointing.*
6. *While Overriding a Method, the sub class should assign either same or higher access level than super class method.*

```
class override3
{
public static void main(String args[])
{
A a1 = new B();
B b1 = (B) a1;

/*
A a1 = new B();
C c1 = (C) a1;

Exception in thread "main"
java.lang.ClassCastException: B
at override3.main(override3.java:39)
*/
}
}
```

Examples Overriding

```
class A
{
void show() { .... }
}
class B extends A
{
void show() { .... }
void show(int x) { ... }
void print() { ... }
}
```

```
A a1 = new B();
a1.show();      // Valid
// a1.show(10); // Invalid
//a1.print();   // Invalid
```

When a super class variable points to a sub class object, then it can only call overridden methods of the sub class.

```
class A
{
protected void show()
{
System.out.println("Hi");
}
}
class B extends A
{
void show()
{
System.out.println("Hi");
}
}
```

```
D:\Java1>javac AB.java
AB.java:10: show() in B cannot
override show() in A; attempting
to assign weaker
access privileges; was protected
void show()
      ^
```

1 error

IS THIS METHOD OVERRIDING

```
class A
{
private void show()
{
System.out.println("Hi");
}
}
class B extends A
{
int show()
{
System.out.println("Hi");
return 10;
}
}
```

NO

***CODE WILL COMPILE
& RUN SUCCESSFULLY***

What's Wrong Here

```
class A  
{  
static int show()  
{  
System.out.println("class A");  
return 0;  
}  
}
```

```
class B extends A  
{  
void show()  
{  
System.out.println("class B");  
}  
}
```

***sample.java:12: show() in B
cannot override show() in A;
overridden method is static
void show()***

^

1 error

Nested Classes

Java programming language allows you to define a class within another class

```
class OuterClass  
{ ...  
    class NestedClass { ... }  
}
```

Enclosing Class OR Outer Class

A nested class is a member of its enclosing class

Nested Class

1. Nested has access to other members of the enclosing class, even if they are declared private
2. Can be private, public, protected or friendly access

Nested Class Types

Static nested classes

1. **Static keyword applied for class declaration**
2. **Static nested class can use the instance fields/methods of the outer class only through object reference.**
3. **Static nested class can be accessed**

OuterClass.StaticNestedClass

4. ***To create an object for the static nested class, use this syntax:***

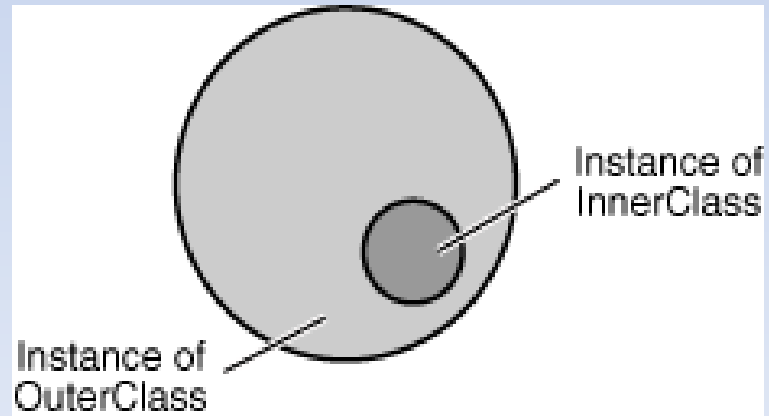
```
OuterClass.StaticNestedClass nestedObject = new  
OuterClass.StaticNestedClass();
```

Nested Class Types cont..

- Non-Static nested classes
 1. These nested classes do not have *static* keyword applied
 2. Non-Static nested class can use the instance fields/methods of the outer class directly.
 3. *To create an object for the non-static nested class, use this syntax:*

OuterClass.NestedClass nestedObject = Outerobjectreference.
new innerclass();

Inner class instance can only exist inside Outer class instance.



Example 1 [Non-static Nested Class]

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a)
;
}
```

Outer Class

***Call to
print() of
outer class***

```
class B
{
int b;
B(int b)
{
int c = b+10;
this.b = c;
}
void show()
{
print();
System.out.println("b="+b);
} // End of class B
} // End of class A
```

***Nested
class with
friendly
access***

Example 1 [Non-static Nested Class]

cont....

```
class innertest1
```

```
{
```

```
public static void main(String args[])
```

```
{
```

```
A a1 = new A(10);
```

```
A.B b1 = a1.new B(100);
```

```
b1.show();
```

```
}
```

```
}
```

Inner class Name

Outer class Reference

To create an inner class instance for non-static classes you need an outer class reference.

Inner class Reference

Outer class Name

If class B is Private then it is not visible in main().

A.B b1 = a1.new B(100); is WRONG/INVALID

Example 2

```
class A
{
private int a;
private int b=10;
```

Outer class

```
A(int a)
{
this.a=a;
}
```

Nested Inner class [Non-static Type]

```
class B
{
private int b;
B(int b)
{
this.b =b;
}
void show()
{
int b=20;
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("this.b="+this.b);
System.out.println("Outer b="+A.this.b);
}
} // End of B inner class
```

Instance Field of B

Outer Class A's a

Local b
B's instance Field b
A's instance Field b

```
void show()
{
B b1 = new B(30);
b1.show();
}
} // End of Outer class A
```



```
class innerTest
{
public static void main(String args[])
{
// Create an inner class B's instance
// Call show() method

// STEP 1
// Create an Outer Instance first
```

```
A a1 = new A(20);
A.B b1 = a1.new B(-30);
b1.show();
```

```
a=20
b=20
this.b=-30
Outer b=10
```

```
// inner class object instantiation thru anonymous
outer
// reference
```

```
A.B b2 = new A(30).new B(-40);
b2.show();
```

```
a=30
b=20
this.b=-40
Outer b=10
```

```
}
}
```

Static Inner class / Static Nested class Example

```
class A
{
private int a;
A(int a)
{
this.a =a;
}
void print()
{
System.out.println("a="+a
);
}
```

Static nested class can refer to outer members only through outer reference

static class B

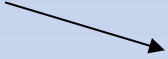
```
{
int b;
B(int b)
{
int c = b+10;
this.b = c;
}
void show()
{
// print(); INVALID
A a1 = new A(10);
a1.print();
System.out.println("b="+b);
}
} // End of class B
} // End of class A
```

Static inner class

Example cont....

```
class innertest10
{
public static void main(String args[])
{
A.B b1 = new A.B(100);
b1.show();
}
}
```

***Instance of static Inner
class***



Static Nested class Example 2

```
class A  
{  
private int a;  
protected static int b=10;  
A(int a)  
{  
this.a=a;  
}  
public void show()  
{  
System.out.println("a="+a);  
display();  
}  
public static void display()  
{  
System.out.println("b="+b);  
}
```

Example 2 cont....

```
static class B
{
private int a;
protected static int b=100;
B(int a)
{
this.a=a;
}
void show()
{
// A.this.show(); // Won't work show() is non-static in outer
display(); // Will work as method is static in outer
System.out.println("a="+a);
// System.out.println("a="+A.this.a);
// Won't work a is non-static in outer
System.out.println("b="+b); // Will refer to its own b
System.out.println("A'sb="+A.b); // will refer to outer class B

new A(40).show();
// This is how you can call non static methods of outer

}
} // End of inner class B
} // End of class A
```

Example 2 cont....

```
class innerTest1
{
public static void main(String
args[])
{
A.B b1 = new A.B(-30);
b1.show();
}
}
```

D:\jdk1.3\bin>java innerTest1

b=10

a=-30

b=100

A'sb=10

a=40

b=10

Local Inner classes [Classes Within method body]

```
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
    class B
    {}
}
}
```

Class declared within a method body.

Here method is show()

Local inner classes Can not be declared as public,private or protected

1. **Class B is visible only in method show().**
2. **It can be used within this show() method only**
3. **Local inner classes can only use final variables from its enclosing method.**
4. **However inner classes can refer to its fields of enclosing class.**

```

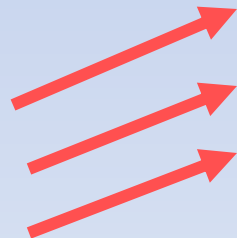
class A
{
private int a;
protected static int b=10;
A(int a)
{
this.a=a;
}
void show()
{
int x=10;
    class B
    {
private int b;
B(int b)
{
this.b=b;
}
void display()
{
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("x="+x);
}
} // End of class B
} // End of show() method
} // End of A class

```

```

D:\jdk1.3\bin>javac
innerTest2.java
innerTest2.java:23: local
variable x is accessed from
within inner class;
to be declared final
System.out.println("x="+x);
                        ^
1 error

```



Reference for A's a
Reference for B's b
Reference is wrong / erroneous
'x' is local variable inside the local
method. Local classes can use only
final fields from enclosing method


```
class innertest
{
public static void
main(String args[])
{
final int a1=10;
```

```
new A(20).show();
print();
} // End of main
static void print()
{
/*
A a1 = new A(30);
a1.show();
*/
System.out.println("Hello"
);
}
}
```

```
class A
{
private int a;
private int b;
int c;
A(int a)
{
this.a =a;
b = a+20;
c = a+40;
}
void show()
{
System.out.println("a1="+a1) ;
System.out.println("a="+a) ;
System.out.println("b="+b) ;
System.out.println("c="+c) ;
}
} //End of A
```

OUTPUT

```
E:\oop>java innertest
```

```
a1=10
```

```
a=20
```

```
b=40
```

```
c=60
```

```
Hello
```

Anonymous Inner classes

- *Another category of local inner classes*
- *Classes without any name i.e classes having no name*
- *Can either implements an interface or extends a class.*
- *Can not have more than one instance active at a time.*
- *Whole body of the class is declared in a single statement ending with ;*

Cont...

- Syntax [If extending a class]

```
[variable_type_superclass =] new superclass_name() {  
                                // properties and methods  
                                } [;]
```

- Syntax [If implementing an interface]

```
[variable_type_reference =] new reference_name() {  
                                // properties and methods  
                                } [;]
```

Anonymous Inner Class Example

```
class A  
{  
private int a;  
A(int a)  
{  
this.a =a;  
}  
void show()  
{  
System.out.println("a="+a);  
} // End of show()  
// End of class A
```

```
class innertest1
{
public static void main(String args[])
{
```

Anonymous inner class extending super class A

```
A a1 = new A(20) {
    public void show()
    {
        super.show();
        System.out.println("Hello");
    }
    public void display()
    {
        System.out.println("Hi");
    }
};
```

```
a1.show();
// a1.display();
}
```



Calling show from inner class

```
interface X
{
int sum(int a,int b);
int mul(int x,int y);
}
class innertest2
{
public static void main(String args[])
{
```

Anonymous inner class implementing an interface

```
X x1 = new X()
{
    public int sum(int a,int b)
    {
        return a+b;
    }
    public int mul(int a,int b)
    {
        return a*b;
    }
};
```

```
System.out.println(x1.sum(10,20));
System.out.println(x1.mul(10,20));
} // End of main
} // End of innertest2
```

Wrapper Classes

- Java uses primitive types, such as int, char, double to hold the basic data types supported by the language.
- Sometimes it is required to create an object representation of these primitive types.
- These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class.

- To satisfy this need, Java provides classes that correspond to each of the primitive types. Basically, these classes encapsulate, or wrap, the primitive types within a class.
- Thus, they are commonly referred to as type wrapper. Type wrapper are classes that encapsulate a primitive type within an object.
- The wrapper types are Byte, Short, Integer, Long, Character, Boolean, Double, Float.
- These classes offer a wide array of methods that allow to fully integrate the primitive types into Java's object hierarchy.

- Wrapper Classes are based upon the well-known software engineering design pattern called the Wrapper pattern.
 - A design pattern is a template solution to a common problem. It describes the problem and identifies the recommended solution(s) to that problem.
 - Because design patterns deal with common problems, they are often quite abstract!
- The problem:e.g. we want to store an int in a Vector, but Vectors do not accept primitive types.
- The Solution:
 - We create another class that wraps the underlying class/type and provides an appropriate interface for the client.
 - e.g. we create an Integer class that subclasses Object (as all classes do), allowing us to store wrapped int's in Vectors.

Example that will *not* work

```
import java.util.Vector;
```

```
public class MyApp {
```

```
    public static void main(String[] args) {
```

```
        int myValue = 2;
```

```
        Vector myVector = new Vector();
```

```
        myVector.addElement(myValue);
```

```
        for (int x=0; x < myVector.size(); x++) {
```

```
            System.out.println("Value: " +  
myVector.get(x));
```

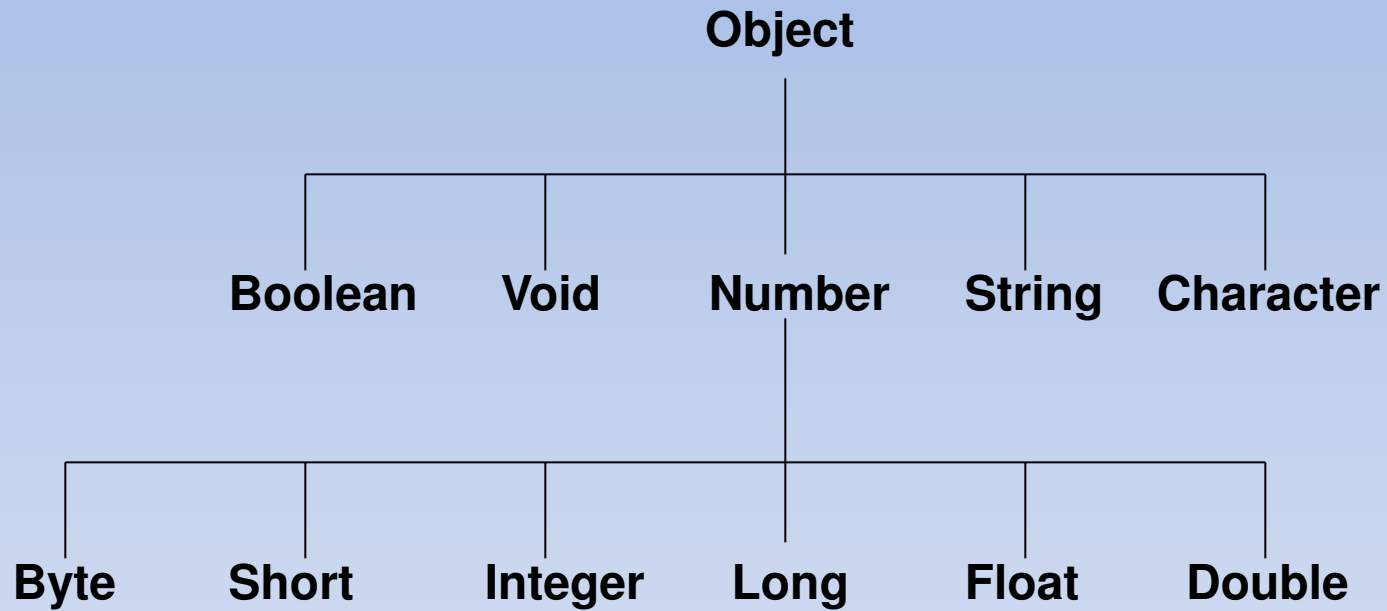
```
        }
```

```
    }
```

```
}
```

The compiler detects
an error here

Java Wrapper Classes



Primitives & Wrappers

- Java has a *wrapper* class for each of the eight primitive data types:

Primitive Type	Wrapper Class	Primitive Type	Wrapper Class
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short

Converting Primitive to Object

- Constructor calling Conversion Action
- Integer IntVal = new Integer(i);
Primitive integer to Integer object
- Float FloatVal = new Float(f);
Primitive float to Float object
- Double DoubleVal = new Double(d);
Primitive double to Double object
- Long LongVal new Long(l);
Primitive long to Long object

Converting Numeric Strings to Primitive

- `int i = Integer.parseInt(str);`
Converts String `str` into primitive integer `i`
- `long l = Long.parseLong(str);`
Converts String `str` into primitive long `l`
- `Double d = Double.parseDouble(str);`
Converting String to primitive double

Note:

`parseInt()` and `parseLong()` ... methods throw a `NumberFormatException` if the value of the `str` does not represent an integer.

Some Useful Methods

- The Java Wrapper Classes include various useful methods:
 - Getting a value from a String

e.g. `int value = Integer.parseInt("33");`

sets value to 33.

- Converting a value to a String:

e.g. `String str = Double.toString(33.34);`

sets str to the String "33.34".

- Getting a wrapper class instance for a value:

e.g. `Integer obj = Integer.getInteger("12");`

creates a new Integer object with value 12 and makes obj refer to that object.

Reading a Double

```
import java.io.*;

class MyProgram {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(
                                new
        InputStreamReader(System.in));
        String line = null;

        System.out.println("Input Something:" );
        try {
            line = in.readLine();
        } catch (IOException ie) {
            System.out.println("Exception caught: " + ie);
        }

        try {
            double value = Double.parseDouble(line);
            System.out.println("Value: " + value);
        } catch (NumberFormatException nfe) {
            System.out.println("You didn't enter a double number");
        }
    }
}
```

Use of the Wrapper Classes

- Java's *primitive* data types (boolean, int, etc.) are not classes.
- Wrapper classes are used in situations where objects are required, such as for elements of a Collection:

```
List<Integer> a = new ArrayList<Integer>();  
methodRequiringListOfIntegers(a);
```

Value => Object: Wrapper Object Creation

- `Wrapper.valueOf()` takes a value (or string) and returns an object of that class:

```
Integer i1 = Integer.valueOf(42);  
Integer i2 = Integer.valueOf("42");
```

```
Boolean b1 = Boolean.valueOf(true);  
Boolean b2 = Boolean.valueOf("true");
```

```
Long n1 = Long.valueOf(42000000L);  
Long n1 = Long.valueOf("42000000L");
```

Object => Value

- Each wrapper class Type has a method `intValue` to obtain the object's value:

```
Integer i1 = Integer.valueOf(42);  
Boolean b1 = Boolean.valueOf("false");  
System.out.println(i1.intValue());  
System.out.println(b1.intValue());
```

=>

42

false

String => value

- The Wrapper class for each primitive *type* has a method `parseType()` to parse a string representation & return the literal value.

```
Integer.parseInt("42")           => 42  
Boolean.parseBoolean("true")     => true  
Double.parseDouble("2.71")       => 2.71  
//...
```

- Common use: Parsing the arguments to a program:

Parsing argument lists

```
// Parse int and float program args.  
public parseArgs(String[] args) {  
    for (int i = 0; i < args.length; i++) {  
        try {  
            ...println(Integer.parseInt(args[i]));  
        } catch (Exception e) {  
  
        }  
    }  
}
```

Sample values:

```
boolObj new Boolean(Boolean.TRUE);  
charObj = new Character('a');  
byteObj = new Byte("100");  
shortObj = new Short("32000");  
intObj = new Integer(2000000);  
longObj = new Long(5000000000000000000L);  
floatObj = new Float(1.42);  
doubleObj = new Double(1.42);  
  
printWrapperInfo(); //method to print objects above
```

Sample values (output from previous slide):

=>

For Boolean & Character Wrappers:

Boolean:true

Character:a

For Number wrappers:

Byte:100

Short:32000

Integer:2000000

Long:5000000000000000000

Float:1.42

Double:1.42

Each Number Wrapper has a **MAX_VALUE** constant:

```
byteObj = new Byte(Byte.MAX_VALUE);  
shortObj = new Short(Short.MAX_VALUE);  
intObj = new Integer(Integer.MAX_VALUE);  
longObj = new Long(Long.MAX_VALUE);  
floatObj = new Float(Float.MAX_VALUE);  
doubleObj = new Double(Double.MAX_VALUE);  
  
printNumValues("MAXIMUM NUMBER VALUES:");
```

MAX values (output from previous slide):

=>

Byte:127

Short:32767

Integer:2147483647

Long:9223372036854775807

Float:3.4028235E38

Double:1.7976931348623157E308

Byte Example

- The Byte class encapsulates a byte value. It defines the constants `MAX_VALUE` and `MIN_VALUE` and provides these constructors:
 - `Byte(byte b)`
`Byte(String str)`
- Here, `b` is a byte value and `str` is the string equivalent of a byte value.

Byte Example

```
import java.util.*;
public class Byte_Demo
{
    public static void main(String args[])
    {
        Byte b1 = new Byte((byte)120);
        for(int i = 125; i<=135; i++)
        {
            Byte b2 = new Byte((byte)i);
            System.out.println("b2 = " + b2);
        }
        System.out.println("b1 Object = " + b1);
        System.out.println("Minimum Value of Byte = " + Byte.MIN_VALUE);
        System.out.println("Maximum Value of Byte = " + Byte.MAX_VALUE);
        System.out.println("b1* 2 = " + b1*2);
        System.out.println("b1* 2 = " + b1.byteValue()*2);
        Byte b3 = new Byte("120");
        System.out.println("b3 Object = " + b3);
        System.out.println("(b1==b3)? " + b1.equals(b3));
        System.out.println("(b1.compareTo(b3)? " + b1.compareTo(b3));
        /*Returns 0 if equal. Returns -1 if b1 is less than b3 and 1 if b1 is
        greater than 1*/
    }
}
```

Short Example

- The Short class encapsulates a short value. It defines the constants MAX_VALUE and MIN_VALUE and provides the following constructors:
Short(short s)
Short(String str)

Short Example

```
import java.util.*;
public class Short_Demo
{
    public static void main(String args[])
    {
        Short s1 = new Short((short)2345);
        for(int i = 32765; i<=32775; i++)
        {
            Short s2 = new Short((short)i);
            System.out.println("s2 = " + s2);
        }
        System.out.println("s1 Object = " + s1);
        System.out.println("Minimum Value of Short = " + Short.MIN_VALUE);
        System.out.println("Maximum Value of Short = " + Short.MAX_VALUE);
        System.out.println("s1* 2 = " + s1.shortValue()*2);
        Short s3 = new Short("2345");
        System.out.println("s3 Object = " + s3);
        System.out.println("(s1==s3)? " + s1.equals(s3));
        Short s4 = Short.valueOf("10", 16);
        System.out.println("s4 Object = " + s4);
    }
}
```

Integer Example

- The Integer class encapsulates an integer value. This class provides following constructors:
 - Integer(int i)
 - Integer(String str)
- Here, i is a simple int value and str is a String object.

Integer Example

```
import java.util.*;
public class Int_Demo
{
    public static void main(String args[])
    {
        Integer i1 = new Integer(12);
        System.out.println("I1 = " + i1);
        System.out.println("Binary Equivalent = " + Integer.toBinaryString(i1));
        System.out.println("Hexadecimal Equivalent = " + Integer.toHexString(i1));
        System.out.println("Minimum Value of Integer = " + Integer.MIN_VALUE);
        System.out.println("Maximum Value of Integer = " + Integer.MAX_VALUE);
        System.out.println("Byte Value of Integer = " + i1.byteValue());
        System.out.println("Double Value of Integer = " + i1.doubleValue());
        Integer i2 = new Integer(12);
        System.out.println("i1==i2 " + i1.equals(i2));
        System.out.println("i1.compareTo(i2) = " + i2.compareTo(i1));
        // Compareto - if it is less than it returns -1 else 1, if equal it return 0.
        Integer i3 = Integer.valueOf("11", 16);
        System.out.println("i3 = " + i3);
    }
}
```


Character Example

- The Character class encapsulates a char value. This class provides the following constructor.
- `Character(char ch)`
- Here, `c` is a char value. `charValue()` method returns the char value that is encapsulated by a Character object and has the following form:
- `char charValue()`

Character Example

```
import java.util.*;

public class Char_Demo
{
    public static void main(String args[])
    {
        Character c1 = new Character('m');
        char c2 = 'O';
        System.out.println("Object C1 = " + c1);
        System.out.println("char value of Character Object = " + c1.charValue());
        System.out.println(c2 + " is defined character set ? " + Character.isDefined(c2));
        System.out.println("c2 is digit = " + Character.isDigit(c2));
        System.out.println("c2 is lowercase character = " + Character.isLowerCase(c2));
        System.out.println("c2 is uppercase character = " + Character.isUpperCase(c2));
    }
}
```

Boolean Example

- The Boolean class encapsulates a Boolean value. It defines FALSE and TRUE constants.
- This class provides following constructors:
 - Boolean(Boolean b)
 - Boolean(String str)
- Here, b is a Boolean value and str is the string equivalent of a Boolean value.
- The methods associated with Boolean Class are as follows:
 1. Boolean booleanValue()
 2. Boolean equals(Boolean b)
 3. String toString(Boolean b)

Boolean Example

```
import java.util.*;
```

```
public class Boolean_Demo
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        Boolean b1 = new Boolean(true);
```

```
        Boolean b2 = new Boolean(false);
```

```
        System.out.println("Object B1 = " + b1);
```

```
        System.out.println("Object B2 = " + b2);
```

```
        Boolean b3 = new Boolean("true");
```

```
        Boolean b4 = new Boolean("false");
```

```
        System.out.println("Object B3 = " + b3);
```

```
        System.out.println("Object B4 = " + b4);
```

```
        System.out.println("Boolean Value = " + b1.booleanValue());
```

```
        System.out.println("(b1==b2)? " + b1.equals(b2));
```

```
        String S1 = b1.toString();
```

```
        System.out.println("String S1 " + S1);
```

```
    }
```

```
}
```

The Object Class

- Every java class has Object as its superclass and thus inherits the Object methods.
- Object is a non-abstract class
- Many Object methods, however, have implementations that aren't particularly useful in general
- In most cases it is a good idea to override these methods with more useful versions.
- In other cases it is **required** if you want your objects to correctly work with other class libraries.

Clone Method

- Recall that the “=” operator simply copies Object **references**. e.g.,
 - >> Student s1 = new Student(“Smith”, Jim, 3.13);
 - >> Student s2 = s1;
 - >> s1.setName(“Sahil”);
 - >> System.out.println(s2.getName());OP:- Sahil
- What if we want to actually make a copy of an Object?
- Most elegant way is to use the clone() method inherited from Object.
 - Student s2 = (Student) s1.clone();

About clone() method

- First, note that the clone method is *protected* in the Object class.
- This means that it is *protected* for subclasses as well.
- Hence, it cannot be called from within an Object of another class and package.
- To use the clone method, you must override in your subclass and upgrade visibility to *public*.
- Also, any class that uses clone must implement the *Cloneable* interface.
- This is a bit different from other interfaces that we've seen.
- There are no methods; rather, it is used just as a marker of your intent.
- The method that needs to be implemented is inherited from Object.

Issue With clone() method

- Finally, clone throws a CloneNotSupportedException.
- This is thrown if your class is not marked Cloneable.
- This is all a little odd but you must handle this in subclass.

Steps For Cloning

- To reiterate, if you would like objects of class C to support cloning, do the following:
 - implement the Cloneable interface
 - override the clone method with public access privileges
 - call `super.clone()`
 - Handle `CloneNotSupportedException` Exception.
- This will get you default cloning means shallow copy.

Shallow Copies With Cloning

- We haven't yet said what the default clone() method does.
- By default, clone makes a *shallow copy* of all iv's in a class.
- *Shallow copy* means that all native datatype iv's are copied in regular way, but iv's that are objects are not recursed upon – that is, references are copied.
- This is not what you typically want.
- Must override clone explicitly for Deep Copying.

Deep Copies

- For *deep copies* that recurse through the object iv's, you have to do some more work.
- `super.clone()` is first called to clone the first level of iv's.
- Returned cloned object's object fields are then accessed one by one and clone method is called for each.
- See `DeepClone.java` example

Additional clone() properties

- Note that the following are typical, but not strictly required:
 - `x.clone() != x;`
 - `x.clone().getClass() == x.getClass();`
 - `x.clone().equals(x);`
- Finally, though no one really cares, `Object` does not support `clone()`;

toString() method

- The Object method

```
String toString();
```

is intended to return a readable textual representation of the object upon which it is called. This is great for debugging!

- Best way to think of this is using a print statement. If we execute:

```
System.out.println(someObject);
```

we would like to see some meaningful info about someObject, such as values of iv's, etc.

default toString()

- By default toString() prints total garbage that no one is interested in
`getClass().getName() + '@' + Integer.toHexString(hashCode())`
- By convention, print simple formatted list of field names and values (or some important subset).
- The intent is not to overformat.
- Typically used for debugging.
- Always override toString()!

equals() method

- Recall that boolean == method compares when applied to object compares references.
- That is, two object are the same if the point to the same memory.
- Since java does not support operator overloading, you cannot change this operator.
- However, the equals method of the Object class gives you a chance to more meaningful compare objects of a given class.

equals method, cont

- By default, equals(Object o) does exactly what the == operator does – compare object references.
- To override, simply override method with version that does more meaningful test, ie compares iv's and returns true if equal, false otherwise.
- See Equals.java example in course notes.

equals subtleties

- As with any method that you override, to do so properly you must obey contracts that go beyond interface matching.
- With equals, the extra conditions that must be met are discussed on the next slide:

equals contract

- ◆ It is *reflexive*: for any reference value x , $x.equals(x)$ should return true.
- ◆ It is *symmetric*: for any reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.
- ◆ It is *transitive*: for any reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true.
- ◆ It is *consistent*: for any reference values x and y , multiple invocations of $x.equals(y)$ consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- ◆ For any non-null reference value x , $x.equals(null)$ should return false.

hashCode() method

- Java provides all objects with the ability to generate a hash code.
- By default, the hashing algorithm is typically based on an integer representation of the java address.
- This method is supported for use with `java.util.Hashtable`
- Will discuss `Hashtable` in detail during Collections discussion.

Rules for overriding hashCode

- Whenever invoked on the same object more than once, the hashCode method must return the same integer, provided no information used in equals comparisons on the object is modified.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the [equals\(java.lang.Object\)](#) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

finalize() method

- Called as final step when Object is no longer used, just before garbage collection
- Object version does nothing
- Since java has automatic garbage collection, finalize() does not need to be overridden reclaim memory.
- Can be used to reclaim other resources – close streams, database connections, threads.
- However, it is strongly recommended *not* to rely on this for scarce resources.
- Be explicit and create own dispose method.

The Object Class

- Every java class has Object as its superclass and thus inherits the Object methods.
- Object is a non-abstract class
- Many Object methods, however, have implementations that aren't particularly useful in general
- In most cases it is a good idea to override these methods with more useful versions.
- In other cases it is **required** if you want your objects to correctly work with other class libraries.

Clone Method

- Recall that the “=” operator simply copies Object **references**. e.g.,
 - >> Student s1 = new Student(“Smith”, Jim, 3.13);
 - >> Student s2 = s1;
 - >> s1.setName(“Sahil”);
 - >> System.out.println(s2.getName());OP:- Sahil
- What if we want to actually make a copy of an Object?
- Most elegant way is to use the clone() method inherited from Object.
 - Student s2 = (Student) s1.clone();

About clone() method

- First, note that the clone method is *protected* in the Object class.
- This means that it is *protected* for subclasses as well.
- Hence, it cannot be called from within an Object of another class and package.
- To use the clone method, you must override in your subclass and upgrade visibility to *public*.
- Also, any class that uses clone must implement the *Cloneable* interface.
- This is a bit different from other interfaces that we've seen.
- There are no methods; rather, it is used just as a marker of your intent.
- The method that needs to be implemented is inherited from Object.

Issue With clone() method

- Finally, clone throws a CloneNotSupportedException.
- This is thrown if your class is not marked Cloneable.
- This is all a little odd but you must handle this in subclass.

Steps For Cloning

- To reiterate, if you would like objects of class C to support cloning, do the following:
 - implement the Cloneable interface
 - override the clone method with public access privileges
 - call `super.clone()`
 - Handle `CloneNotSupportedException` Exception.
- This will get you default cloning means shallow copy.

Shallow Copies With Cloning

- We haven't yet said what the default clone() method does.
- By default, clone makes a *shallow copy* of all iv's in a class.
- *Shallow copy* means that all native datatype iv's are copied in regular way, but iv's that are objects are not recursed upon – that is, references are copied.
- This is not what you typically want.
- Must override clone explicitly for Deep Copying.

Deep Copies

- For *deep copies* that recurse through the object iv's, you have to do some more work.
- `super.clone()` is first called to clone the first level of iv's.
- Returned cloned object's object fields are then accessed one by one and clone method is called for each.
- See `DeepClone.java` example

Additional clone() properties

- Note that the following are typical, but not strictly required:
 - `x.clone() != x;`
 - `x.clone().getClass() == x.getClass();`
 - `x.clone().equals(x);`
- Finally, though no one really cares, `Object` does not support `clone()`;

toString() method

- The Object method

```
String toString();
```

is intended to return a readable textual representation of the object upon which it is called. This is great for debugging!

- Best way to think of this is using a print statement. If we execute:

```
System.out.println(someObject);
```

we would like to see some meaningful info about someObject, such as values of iv's, etc.

default toString()

- By default toString() prints total garbage that no one is interested in
`getClass().getName() + '@' + Integer.toHexString(hashCode())`
- By convention, print simple formatted list of field names and values (or some important subset).
- The intent is not to overformat.
- Typically used for debugging.
- Always override toString()!

equals() method

- Recall that boolean == method compares when applied to object compares references.
- That is, two object are the same if the point to the same memory.
- Since java does not support operator overloading, you cannot change this operator.
- However, the equals method of the Object class gives you a chance to more meaningful compare objects of a given class.

equals method, cont

- By default, equals(Object o) does exactly what the == operator does – compare object references.
- To override, simply override method with version that does more meaningful test, ie compares iv's and returns true if equal, false otherwise.
- See Equals.java example in course notes.

equals subtleties

- As with any method that you override, to do so properly you must obey contracts that go beyond interface matching.
- With equals, the extra conditions that must be met are discussed on the next slide:

equals contract

- ◆ It is *reflexive*: for any reference value `x`, `x.equals(x)` should return `true`.
- ◆ It is *symmetric*: for any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- ◆ It is *transitive*: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- ◆ It is *consistent*: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the object is modified.
- ◆ For any non-null reference value `x`, `x.equals(null)` should return `false`.

hashCode() method

- Java provides all objects with the ability to generate a hash code.
- By default, the hashing algorithm is typically based on an integer representation of the java address.
- This method is supported for use with `java.util.Hashtable`
- Will discuss `Hashtable` in detail during `Collections` discussion.

Rules for overriding hashCode

- Whenever invoked on the same object more than once, the hashCode method must return the same integer, provided no information used in equals comparisons on the object is modified.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the [equals\(java.lang.Object\)](#) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

finalize() method

- Called as final step when Object is no longer used, just before garbage collection
- Object version does nothing
- Since java has automatic garbage collection, finalize() does not need to be overridden reclaim memory.
- Can be used to reclaim other resources – close streams, database connections, threads.
- However, it is strongly recommended *not* to rely on this for scarce resources.
- Be explicit and create own dispose method.