


# Programming with Java

# Sujata Batra - sujatabatra@hotmail.com

- IT Trainer Since 2000
- Conducting Java Training Since 2011
- More than 50+ Corporate Clients

# Objective

- Describe the key features of Java technology
- Write, compile, and run a simple Java application
- Describe the function of the Java Virtual Machine
- Define garbage collection

# History of Java

- Conceived as Programming language for embedded systems like microwave ovens, televisions etc
- One of the first projects developed using Java
    - personal hand-held remote control named Star 7.
- The original language was called Oak
- Java was developed in the year 1991 at Sun Microsystems
    - Java is a simple, object oriented, interpreted distributed, robust, secure, architecture neutral, portable high-performance, multithreaded, and dynamic language

# Primary Goals of Java

- **Provides an easy-to-use language by:**
  - Avoiding many pitfalls of other languages
  - Being object-oriented
  - Enabling users to create streamlined and clear code
- **Provides an interpreted environment for**:
  - Improved speed of development
  - Code portability
  - Enables users to run more than one thread of activity
  - Loads classes dynamically; that is, at the time they are actually needed
  - Supports changing programs dynamically during runtime by loading classes from disparate sources

# Features of Java

- The following features fulfill these goals:
  - The Java Virtual Machine (JVM)
  - Garbage collection
  - The Java Runtime Environment (JRE)
  - JVM tool interface

# The Java Virtual Machine

- Executes instructions that a Java compiler generates.
- A  runtime environment, is embedded in various products, such as web browsers, servers, and operating systems
- Its  Imaginary machine that is implemented by emulating software on a real machine
- Reads compiled byte codes that are platform-independent
- **Bytecode**
  - a special machine language that can be understood by the JVM independent of any particular computer hardware,

# JVM Tasks

1. Loads code
   - Loads all classes necessary for the execution of a program
   - Maintains classes of the local file system in separate *namespaces*
2. Verifies code (Bytecode Verifier)
   - The code adheres to the JVM specification.
   - The code does not violate system integrity.
   - The parameter types for all operational code are correct.
   - No illegal data conversions have occurred.

# Class Loader

- Bootstrap Class Loader
  - loads java's core classes like java.lang, java.util etc.
- Extensions Class Loader
  - JAVA_HOME/jre/lib/ext contains jar packages that are extensions of standard core java classes.
  - Extensions class loader loads classes from this ext folder.
  - Using the system environment propery java.ext.dirs you can add 'ext' folders and jar files to be loaded using extensions class loader.
- System Class Loader
  - Java classes that are available in the java classpath are loaded using System class loader.

# Garbage Collection

- Allocated memory that is no longer needed should be deallocated.
- In other languages, deallocation is the programmer's responsibility.
- The Java programming language provides a system-level thread to track memory allocation.
- Garbage collection has the following characteristics:
  - Checks for and frees memory no longer needed
  - Is done automatically
  - Can vary dramatically across JVM implementations

# Application and Runtime Environment

- The JRE of the SDK contains the complete set of class files for all the Java technology packages, which includes basic language classes, GUI component classes, and so on.
- The Java Application Programming Interface (API) is prewritten code, organized into packages of similar topics.
- Applet and AWT packages include classes for creating fonts, menus, and buttons. The full Java API is included in the Java 2 Standard Edition

# Application and Runtime  Environment

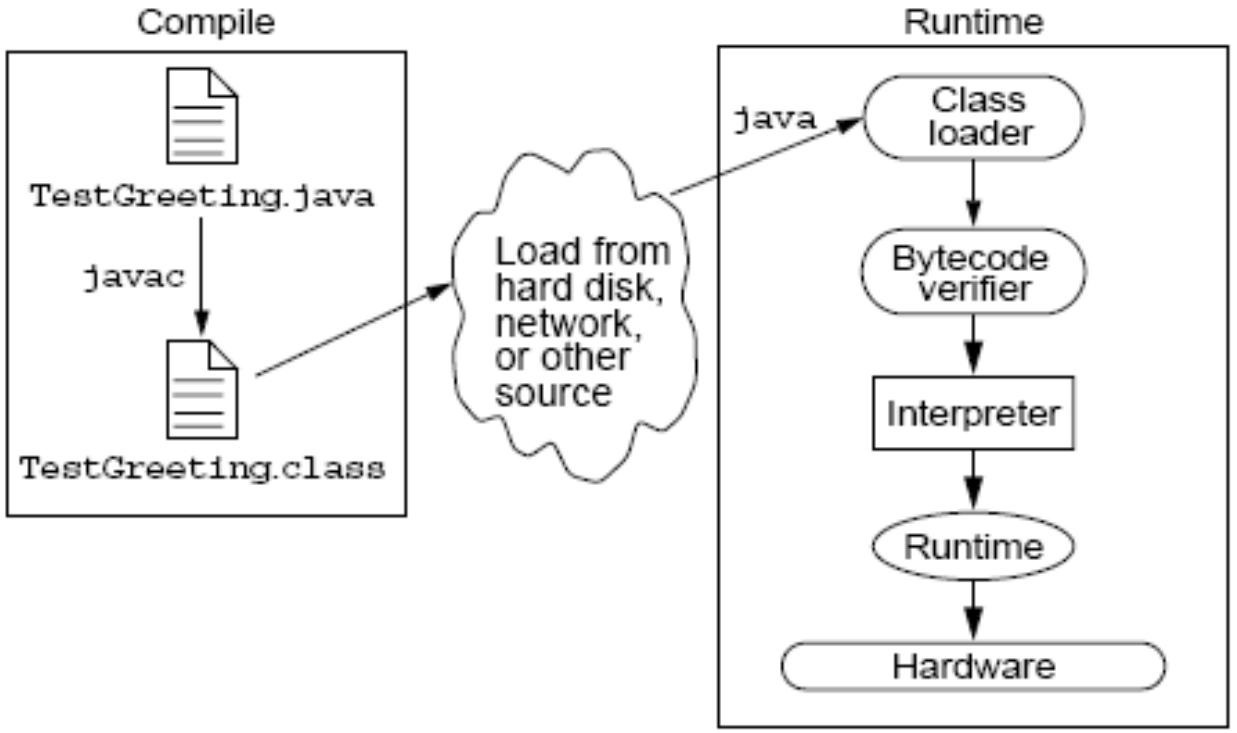
- J2SE includes the essential compiler, tools, runtimes, and APIs for writing, deploying, and running applets and applications in the Java programming language.

- JDK is the short-cut name for the set of Java development tools, consisting of the API classes, a Java compiler, and the Java virtual machine interpreter, regardless of which version.

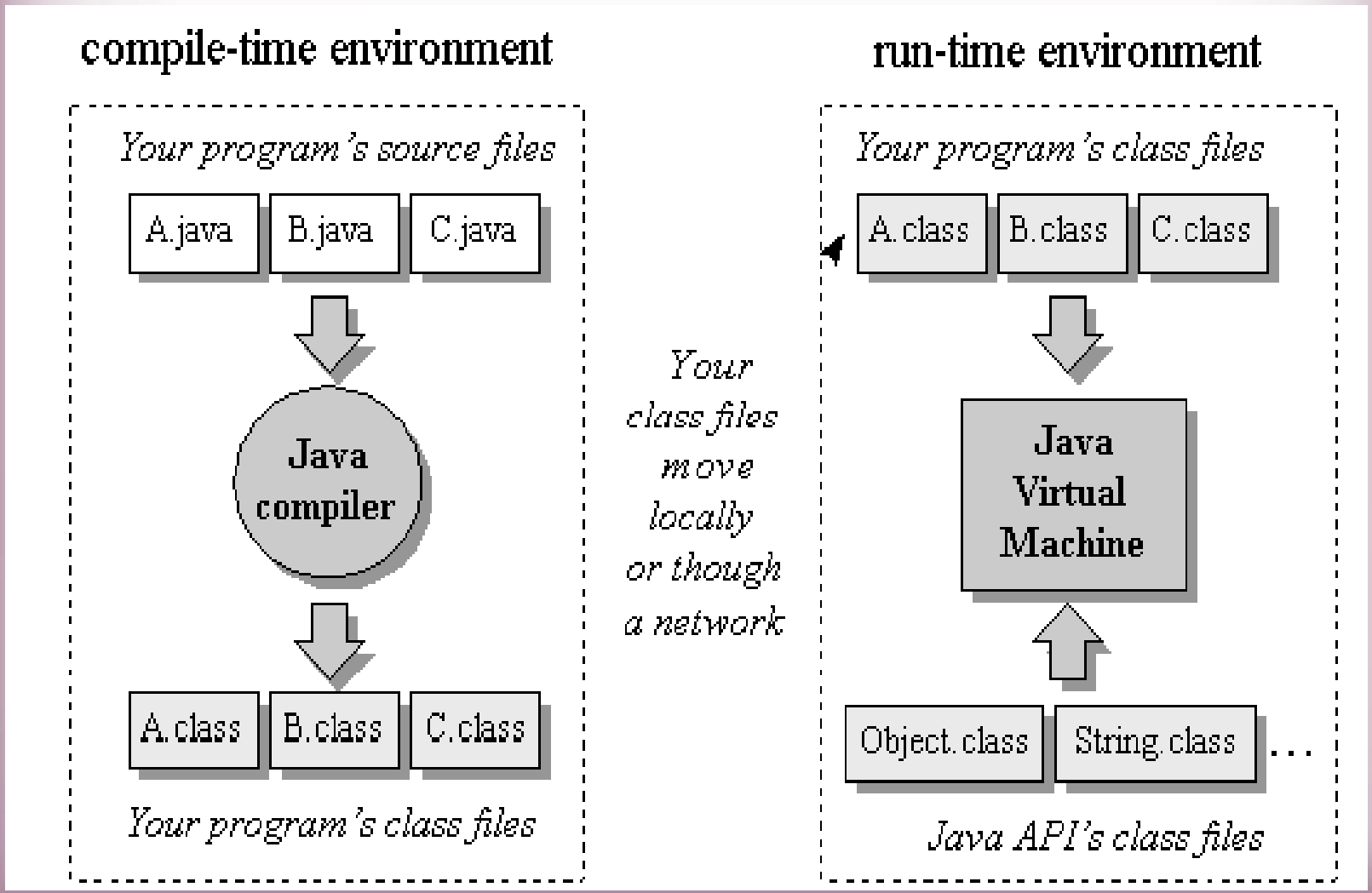
- The JDK is used to compile Java applications and applets. The most current version is the J2SE is 6.0

# Java Run Time Environment

The Java application environment performs as follows:

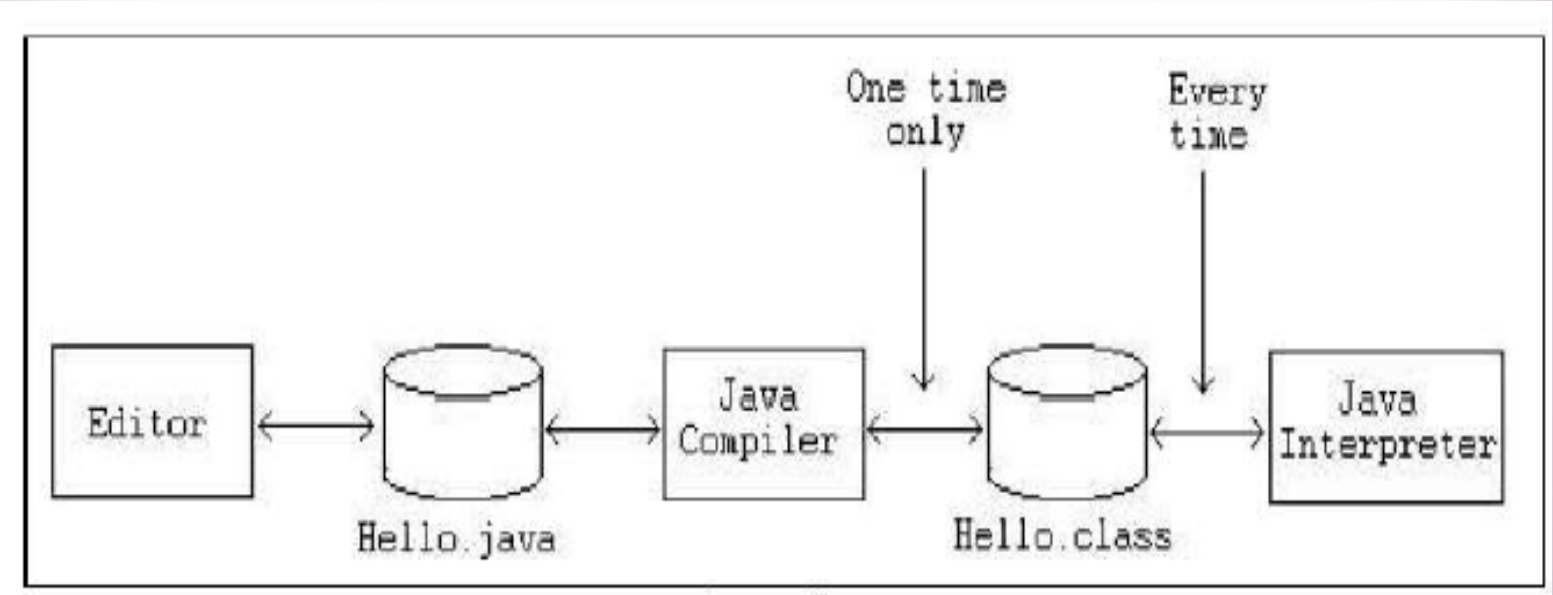


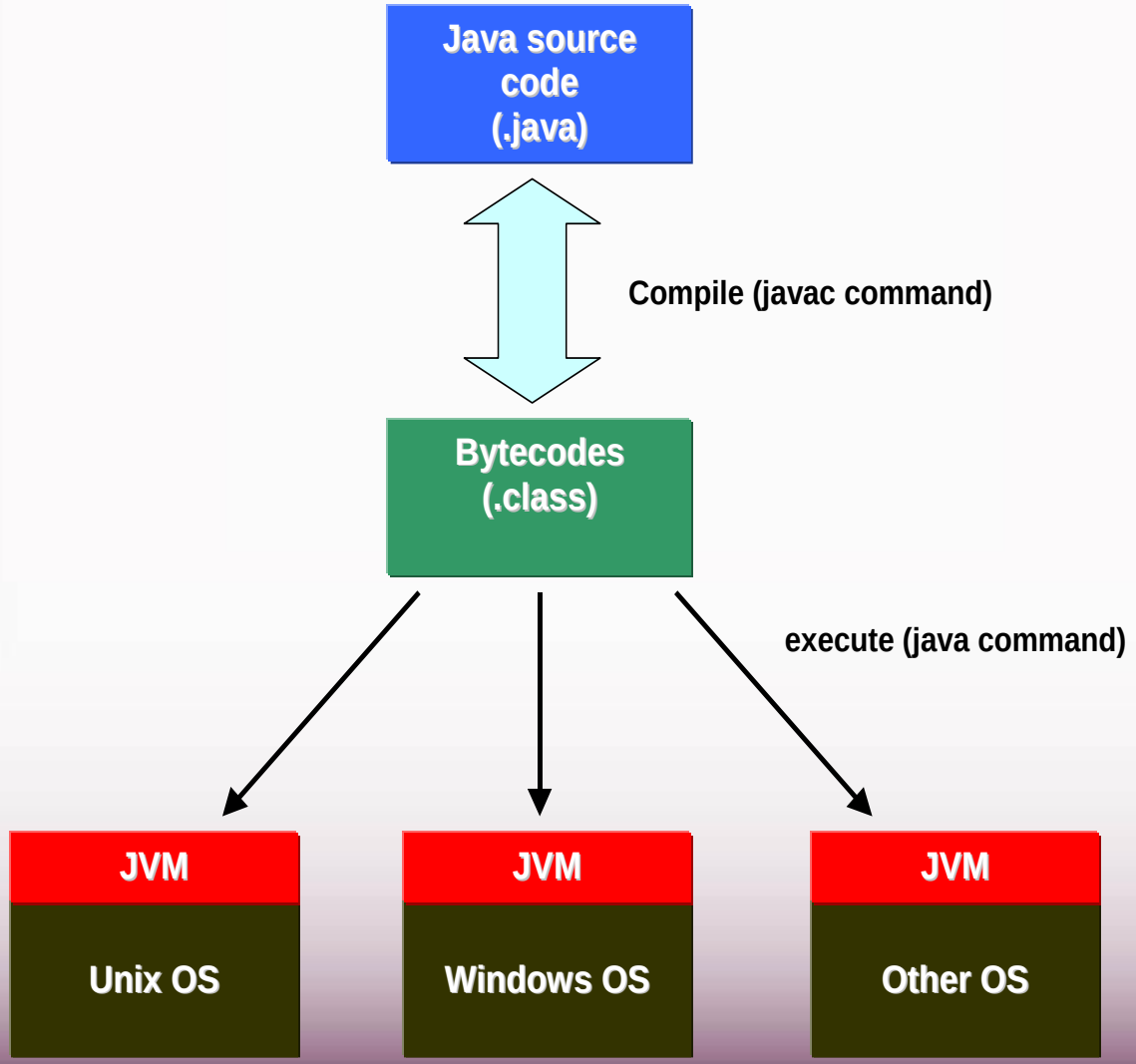
# Compile time and Runtime Environment



# Phases of a Java Program

- The following figure describes the process of compiling and executing a Java program




Java source code (.java)
Compile (javac command)
Bytecodes (.class)
execute (java command)
JVM
Unix OS
JVM
Windows OS
JVM
Other OS

# Integrated Development Environments

- An IDE is the high-productivity tool
- Used to edit, compile, and test programs, manage projects, debugging, building GUI interfaces, etc.
- IDE provides extensive programming support for editing and project management,
- The Popular IDE’s
  - Eclipse
  - NetBeans
  - IntelliJ

# Java language syntax

# Objectives

- Identify the basic parts of a Java program
- Differentiate among Java literals, primitive data types,
- variable types ,identifiers and operators
- Develop a simple valid Java program using the concepts learned in this chapter

# Program structure

- A program in Java consists of one or more class definitions
- One of these classes must define a method *main(),* which is where the program starts running
- Java programs should always end with the .java extension.
- There can be More than one Class Definition in a class ,but only one public class
- Source Filenames should match the name of the  public class.

# Source File Layout

- Basic syntax of a Java source file is:
- ***[<package_declaration>]***
- ***<import_declaration>****
- ***<class_declaration>+***

# Software Packages

- Packages help manage large software systems.
- Packages can contain classes and sub-packages.

- **package <*top_pkg_name*>[.<*sub_pkg_name*>]*;**

- Specify the package declaration at the beginning of the source file.

- Only one package declaration per source file.

- If no package is declared, then the class is placed into the default package.

- Package names must be hierarchical and separated by dots.

# The import Statement

- **import *<pkg_name>[.<sub_pkg_name>]*.<class_name>;***

- **import *<pkg_name>[.<sub_pkg_name>]*.*;***

- import java.util.List;
- import java.io.*;
- import shipping.gui.reportscreens.*;

- The import statement does the following:
  - Precedes all class declarations
  - Tells the compiler where to find classes

# Example 1.1

```
package com.training;

public class Greetings {


    public String getMessage() {

       return "Welcome to Java  Programming";

    }

}
```

# Example 1.1 (contd)

```
package com.training;

public class TestGreetings {

    public static void main(String[] args) {

      Greetings grtObj = new Greetings();


      System.out.println(grtObj.getMessage());

    }

}
```

# The System Class

- Its part of the java.lang package
- The classes in this package are available without the need for an import statement
- This  class contains several useful class fields and methods.
- It can’t be Instantiated
- It also  Provides facilities for
  - Standard Input
  - Standard Output
  - Error Output Streams
  - Access to externally defined properties

# Objectives

- Define modeling concepts: *abstraction, encapsulation,*
- Discuss why you can reuse Java technology application code
- Define *class, member, attribute, method, constructor, and package*
- Use the access modifiers private and public as appropriate for the guidelines of encapsulation
- Invoke a method on a particular object
- Use the Java technology application programming interface (API) online documentation

# The Analysis and Design Phase

- Analysis describes *what the system needs to do:*
- Modeling the real-world, including actors and activities, objects, and behaviors
- Design describes *how the system does it:*
- Modeling the relationships and interactions between objects and actors in the system
- Finding useful abstractions to help simplify the problem or solution

# Abstraction

- Functions – Write an algorithm once to be used in many situations
- Objects – Group a related set of attributes and behaviors into a class
- Frameworks and APIs – Large groups of objects that support a complex activity;
- Frameworks can be used *as is or be modified to extend the basic behavior*

# Classes as Blueprints for Objects

- **In manufacturing**, a blueprint describes a device from which many physical devices are constructed.
- **In software**, a class is a description of an object:
- A class describes the data that each object includes.
- A class describes the behaviors that each object exhibits.

- In Java technology, classes support three key features OOP
  - Encapsulation
  - Inheritance
  - Polymorphism

# Declaring Java Technology Classes

- Basic syntax of a Java class:

***<modifier>* class <class_name> {***
***<attribute_declaration>****
***<constructor_declaration>****
***<method_declaration>****
**}**

# Declaring Attributes

- Basic syntax of an attribute:

- ***<modifier>* <type> <name> [ = <initial_value>];***

- public class Foo {
- private int x;
- private float y = 10000.0F;
- private String name = "Bates Motel";
- }

# Declaring Methods

- Basic syntax of a method:
- ***<modifier>* <return_type> <name> ( <argument>* ) {***
- ***<statement>****
- **}**

```
public int getWeight() {
return weight;
}

public void setWeight(int newWeight) {
      if ( newWeight > 0 ) {
            weight = newWeight;
      }
   }
```

# Accessing Object Members

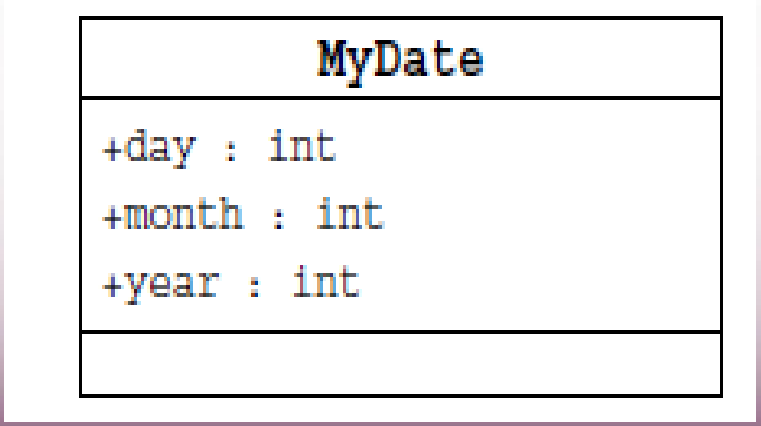
- The *dot notation is:* ***<object>.<member>***
  - used to access object members, including attributes and methods.

- d.setWeight(42);
- d.weight = 42; // only permissible if weight is public

# Information Hiding

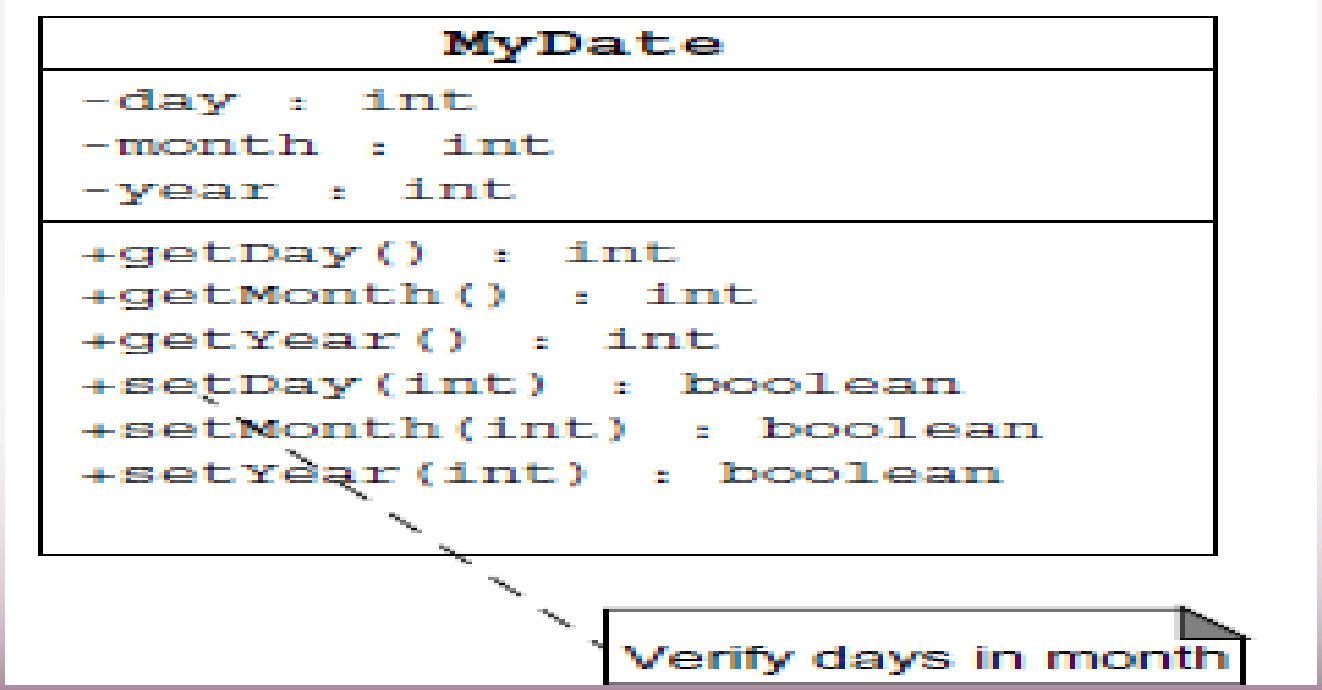
- Client code has direct access to internal data
- (d refers to a MyDate object):

- d.day = 32; // invalid day
- d.month = 2; d.day = 30; // plausible but wrong
- d.day = d.day + 1; // no check for wrap around



# Encapsulation

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code more maintainable



# Objects and Data Abstraction

- Consider the data
  - In many applications, data is more complicated than just a simple value
  - An Employee
    - The data here are actually:
      - int empId – an integer of Employee Id
      - double[] phoeNumber – an array of Phone Numbers
      - double salary – the salary of employee
    - Note that individually the data are just int or a double
      - However, together they make up a Employee
    - This is fundamental to object-oriented programming (OOP)

# Objects and Data Abstraction

- Consider the operations
  - Now consider operations that an Employee can do
    - Note how that is stated – we are seeing what a Employee CAN DO rather than WHAT CAN BE DONE to it
    - This is another fundamental idea of OOP – objects are **ACTIVE** rather than PASSIVE
- Objects enable us to combine the data and operations of a type together into a single entity

# Encapsulation and Data Abstraction

- We do not need to know the implementation details of a data type in order to use it
  - This includes the methods AND the actual data representation of the object
- This concept is exemplified through objects
  - We can think of an object as a container with data and operations inside
    - We can see some of the data and some of the operations, but others are kept hidden from us
    - The ones we can see give us the functionality of the objects

# Declaring Constructors

- Basic syntax of a constructor
  - ***[<modifier>] <class_name> ( <argument>* ) {***
  - ***<statement>****
  - **}**

```
public class Dog {
private int weight;
  public Dog() {
    weight = 42;
  }
}
```

# The Default Constructor

- There is always at least one constructor in every class.
- If the writer does not supply any constructors, the default constructor is present automatically:
- The default constructor takes no arguments
- The default constructor body is empty
- The default enables you to create object instances with new *Xxx()without having to write a constructor.*

# Comments

- Java supports three forms of comments
  - // single line comments
  - /* multi
    line
    comment
    */
  - /** a
    * Javadoc
    * comment
    */

# Variables and Methods

- A **variable**, which corresponds to an attribute, is a named memory location that can store a certain type of value.
- Variable is a kind of  special container that can only hold objects of a certain type.
- Primitive type Variable
  - Basic, built-in types that are part of the Java language
  - Two basic categories
    - boolean
    - Numeric
      - » Integral - byte, short, int, long, char
      - » Floating point - float, double

# Instance Variables and Methods

- Variables and methods can be associated either with objects or their classes
- An **instance variable and instance method** belongs to an object.
- They can have any one of the four access levels
  - Three access modifies – private, public , protected
  - Can be Marked final, transient
- They have a default value
- **A class variable** (or **class method**) is a variable (or method) that is associated with the class itself.

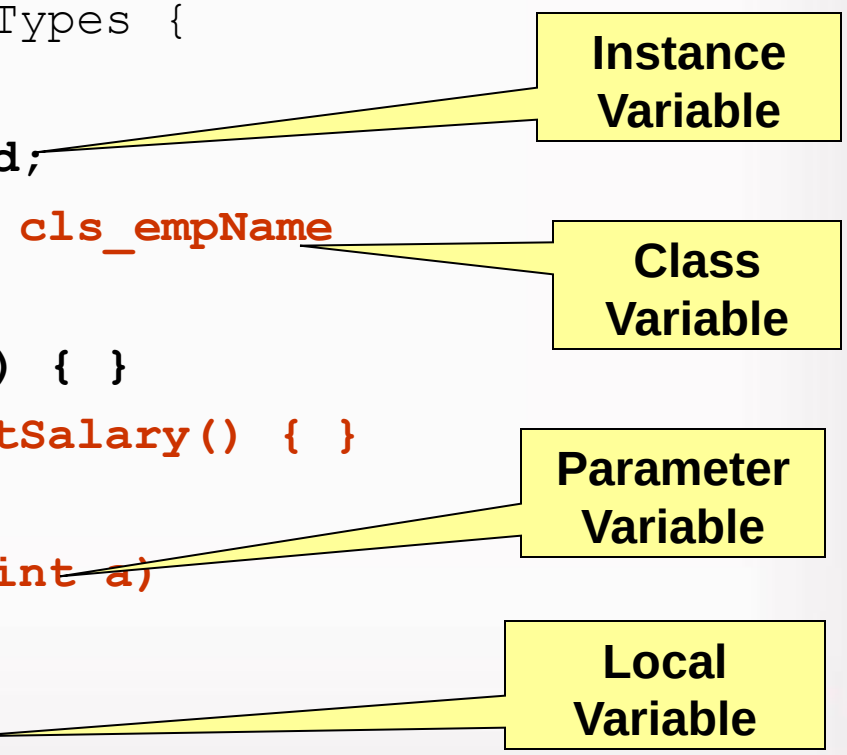
# Example for Variables

```
public class VariableTypes {

private int inst_empid;
private static String cls_empName

public  void getData() { }
public static void getSalary() { }

public void showData(int a)
{
  int localVariable ;
}
}
```



# Identifiers

- Identifiers are used to name **variables, methods, classes and interfaces**

- Identifiers
  - start with an alphabetic character
  - can contain letters, digits, or “_”
  - are unlimited in length

- Examples
  - *answer*, *total*, *last_total*, *relativePoint*, *gridElement*, *person*, *place*, *stack*, *queue*

# Initialization

- Local variables
  - must be *explicitly* initialized before use
- Parameter variables
  - Pass by value
- Class and instance variables
  - Instance variables Have a Default Value

# Local Variable needs to Be Initialized

```
public class LocalVariable {
    private String name;
public void display()
{
 int age;
 System.out.println("Age"+age);
 System.out.println("Name"+name);


}
```

**Age Should be Initialized before Use**

# Instance Variable have Default Values

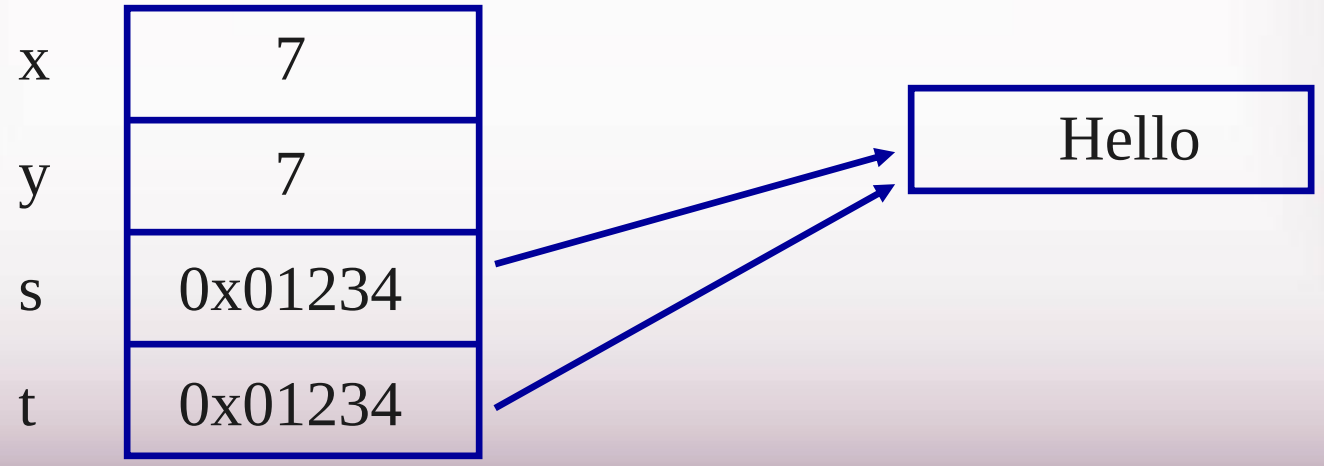
```
class Values
{
   private int a;
   private float b;
   private String c;
   public void display()
   {
        System.out.println("integer"+a);
        System.out.println("float"+b);
        System.out.println("String"+c);
}
public class DefaultVales {

   public static void main(String[] args) {

        Values v = new Values();
        v.display();
   }
}
```

# Assignment of reference variables

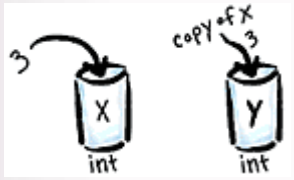
```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

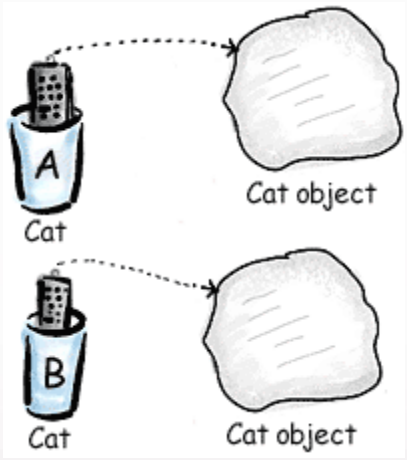


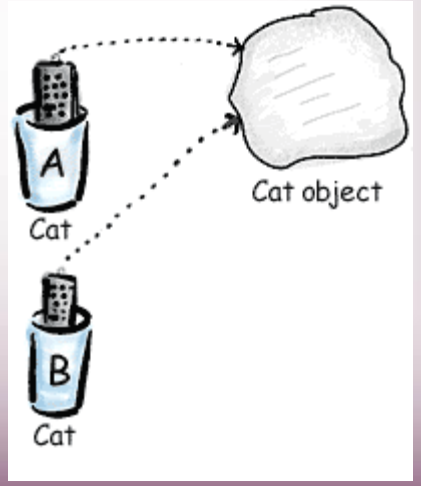
# Pass-by-Value

- The Java programming language only passes arguments by value for primitive data types.
- When an object instance is passed as an argument to a method, the value of the argument is a *reference* to the object
- The *contents* of the object can be changed in the called method, but the object reference is never changed
- **For primitives, you pass a copy of the actual value.**
- **For references to objects, you pass a copy of the reference**
- You never pass the object. All objects are stored on the heap.

# Pass By Value



# Casting Primitive Types

- Casting creates a new value and allows it to be treated as a different type than its source
- JVM can implicitly promote from a narrower type to a wider type
- To change to a narrow type explicit casting is required
- Byte -> short -> int -> long -> float -> double

# Casting Primitive Types

```
public class PrimCasting {

      public static void main (String [] args){
                int x = 99;
                double y = 5.77;
                x = (int)y;  //Casting
                System.out.println("x = "+ x);

                        double y1 = x; //No Casting
          int i =42;
          byte bt;
          bt= (byte)i;

        System.out.println("The Short number"+ bt);
     }
   }
```

# Wrapper Classes

- Primitives  have no associated methods
- Wrapper Classes are used encapsulate primitives
- They also provide some static utility methods to work on them

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

# Wrapping Primitives

- Wrapping a value
  - int i = 288
  - Integer iwrap = new Integer(i);
- unWrapping a value
  - int unwrapped = iwrap.intValue();
- **Methods In Wrapper Class**
  - parseXxx()
  - xxxValue()
  - valueOf()

# Wrapper Class Method
# Convert String to Numbers

```
public class ParsingStrings
 {
      public static void main(String args[])
        {
   int ino=Integer.parseInt(args[0]);
    floatfno = Float.parseFloat(args[1]);
    double dno = Double.parseDouble(args[2]);
    Long lno = Long.parseLong(args[3]);

     System.out.println("Integer value" +ino );
    _String strIno = Integer.toString(ino);
     System.out.Println("String Value"+strIno);
     }
}
```

# Auto Boxing

- Java 5.0 provided autoboxing

| | |
|---|---|
| Integer n = new Integer(123)<br>Int m = n.intValue()<br>m++;<br>n=new Integer(m);<br>System.out.println(n); | Integer n = new Integer(123);<br>n++;<br>System.out.println(n); |

# Auto Boxing

```
public class ABoxing {

      public void show(int a, float b)
      {
        System.out.println("Integer"+a*2);
        System.out.println("Float"+b*2);
      }

      public static void main(String[] args) {

        ABoxing abObj =new  ABoxing ();

        Integer a = 10;
        Float b =20f;
        abObj.show(a,b);

      }
    }
```

# Command Line Arguments

```
class CommandLineArgs
{
   public static void main(String args[])
  {
   System.out.println("hello, welcome to the java world
  "+args[0]+" to all "+args[1]);

   System.out.println("it is "+args[2]+" a high level
  language"  +args[0]);
  }
}
```

# java.util.Scanner Class

- A simple text scanner which can parse primitive types and strings using regular expressions.
- A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.
- The resulting tokens may then be converted into values of different types using the various next methods.

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

# java.util.Scanner Class

```
public void show()
   {
    Scanner sc = new Scanner(System.in);

    System.out.println("Enter The Number");
    int number = sc.nextInt();
    System.out.println("Enter the Name");
    String name = sc.next();
    System.out.println(number + ":"+name);

   }
```

# java.util.Scanner Class

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


        String line="Java,is,in,OOP,Language";

        Scanner sc1 = new Scanner(line);

        sc1.useDelimiter(",");

       while (sc1.hasNext())
       {
         System.out.println(sc1.next());
       }

  }
```

# Flow Control

# Objectives

- Decision control structures (if, else, switch)
- Repetition control structures (while, do-while, for)
- Branching statements (break, continue, return)

# Control Structures

- To change the ordering of the statements in a program
- Two types of Control Structures
- decision control structures ,
  - allows us to select specific sections of code to be executed
    - **if -- else , if – else if**
    - **switch -- case**
- repetition control structures
  - allows us to execute specific sections of the code a number of times
    - **while**
    - **do -- while**
    - **for**

# Decision Control Structures

- Types:
  - if-statement
  - if-else-statement
  - If-else if-statement

- If Specifies that a statement or block of code will be executed if and only if a certain **boolean** statement is true.

```
if( boolean_expression )
statement;
```

or

```
if( boolean_expression ){
statement1;
statement2;
}
```

- **boolean_expression** : can be an expression or variable.

# if-else statement

```
if( boolean_exp ){
Statement(s)
}
else {
Statement(s)
}
```

```
if(boolean_exp1 )
statement1;
else if(boolean_exp2)
statement2;
else
statement3;
```

- For Comparison **==** used instead =
- **=** being an assignment operator
- **equals** Method Should Be Used for **Objects** comparison

# switch-statement

- Allows branching on multiple outcomes.
- switch expression  is an integer ,character expression or variable
- case_selector1, case_selector2 and unique integer or character constants.
- If none of the cases are satisfied, the optional default block if present is executed.

```
switch( switch_expression ){
case case_selector1:
Statement(s);
break;
case case_selector2:
Statement(s);
break;
default:
statement1;
}
```

# switch-statement

- When a switch is encountered,
  - evaluates the switch_expression,
  - jumps to the case whose selector matches the value of the expression.
  - executes the statements in order from that point on until a break statement is encountered
  - break statement stops executing statements in the subsequent cases, it will be  last statement in a case.
  - Used to make decisions based only on a single integer or character value.
  - The value provided to each case statement must be unique.

# switch-statement

```
public double CalculateDiscount(int pCode)
{
double discountPercentage=0.0d;

switch(pCode)
{
case 1:
  discountPercentage=0.10d;
  break;
case 2:
  discountPercentage=0.15d;
  break;
case 3:
  discountPercentage=0.20d;
  break;
default:
  discountPercentage=0.0;
}
return discountPercentage;
}
```

# Switch-Case in a Method

- Can have Either Return or Break if present inside a Method, but should provide  a default Value

```
public String switchExample(int key)       {
        switch (key) {
        case 1:
                return "Good Morning";
        case 2:
                return "Good AfterNoon";
        case 3:
           return "Good Evening";
        default:
                return "Good Night";
        }
```

# Repetition Control Structures

- **while-loop**
  - The statements inside the while loop are executed as long as the Boolean expression evaluates to true
- **do-while loop**
  - statements inside a do-while loop are executed several times as long as the condition is satisfied, the statements inside a do-while loop are executed at least once

```
while(boolean_expression){
statement1;
statement2;
}
```

```
do{
statement1;
statement2;
}while(boolean_expression);
```

**Watch this**

# for-loop

- Same code a number of times.

- ```
  for(Initialexpr;LoopCondition;StepExpr) {
   statement1;
   }
  ```

  - Declaration parts are left out so the `for` loop will act like an endless loop.

  ```
  for( ; ; ) {
      System.out.println("Inside an endless loop");
  }
  ```
  ```

# Enhanced For Loop

- The enhanced for loop, simplifies looping through an array or a collection.
- Instead of having *three* components, the enhanced for has *two*.

- **declaration**
  - The *newly declared* block variable, of a type compatible with the elements of the array being accessed.
- **expression**
  - This must evaluate to the array you want to loop through. This could be an array variable or a method call that returns an array

- int [] a = {1,2,3,4};

- **for(int n : a)**
- **System.out.print(n);**.

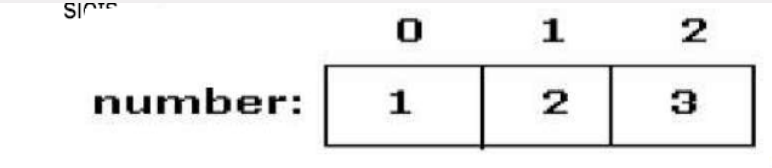
# Branching Statements

- Branching statements allows to redirect the flow of program execution.
  - break
  - continue
  - return.
- `System.exit()` All program execution stops; the VM shuts down.

# Arrays

# Introduction to Arrays

- Array is one variable is used to store a list of data and manipulate
- This type of variable is called an array.
- It stores multiple data items of the same data type, in a contiguous block of memory, divided into a number of slots.
- The *new* keyword to create an array object.
- Need to tell the compiler how many elements will be stored in it.



# Creating Arrays

- There are three steps to creating an array,
- **declaring**
  - int[] k;
  - float[] yt;
  - String[] names;
- **allocating**
  - k = new int[3];
  - yt = new float[7];
  - names = new String[50];
- **initializing**
  - int[ ] k = {1, 2, 3};
  - float[ ] yt = {0.0f, 1.2f, 3.4f, -9.87f, 65.4f, 0.0f, 567.9f};

# Array Bounds

- All array subscripts begin with 0 and ends with n-1
- In order to get the number of elements in an array, can use the length field of an array.

- The length field of an array returns the size of the array.

```
int list [] = new int[10];

for (int i = 0; i< list.length; i++)
{
        System.out.println(list[i]);
}
```

- There is no array element arr[n]! This will result in an array-index-out-of bounds exception.

# Array of Objects

```
public class Book
{
   private int bookno;
   private String bookname;
   public Book(int bookno,String bookname)
   {
        this.bookno=bookno;
        this.bookname=bookname;
   }
   public int getBookno()
   {
   return this.bookno;
   }
   public String getBookname()
   {
   return this.bookname;
   }
 }
```

# Array of Objects

```
public class ArrayofObject {

public void displaybooks(Book[] bks)
{
for(int i=0;i<bks.length;i++)
{
    System.out.println("Book Number :="+bks[i].getBookno());
    System.out.println("Book Name :="+bks[i].getBookname());
}
}
   public static void main(String args[])
   {
   Book[]  bk = new Book[2];
   Book b1 = new Book(100,"java");
   Book b2= new Book(101,"j2ee");
   bk[0]=b1;
   bk[1]=b2;
         ArrayofObject ab= new ArrayofObject();
         ab.displaybooks(bk);
}
}
```

# Classes

# Classes

- The class declaration introduces a new class
  - A class describes the structure and behavior of its instance objects in terms of instance variables and methods
  - Like variables, classes may be declared at different scopes. The scope of a class directly affects certain properties of the class

*Class declaration syntax*

```
modifier class identifier {
    constructorDeclarations
    methodDeclarations
    staticMemberDeclarations
    instanceVariableDeclarations
    staticVariableDeclarations
}
```

# Classes

Top-level classes can be declared as

- public
  - a public class is globally accessible.
  - a single source file can have only *one* public class or interface
- abstract
  - an abstract class cannot be instantiated
- final
  - a final class cannot be subclassed
- Default
  - With any Modifier

- They can't be declared as protected and private

# Constructors

- Have no return type
- Have the same name as the class
- If we don't' put a constructor the compiler puts a default one
  - The default constructor has the *same access modifier as the class*.
  - The default constructor has *no arguments*.
  - The default constructor is *always* a no-arg constructor, but a no-arg constructor is not necessarily the *default* constructor
  - The default constructor includes a *no-arg call to the super constructor* (super()).
- They are not inherited and hence they are not overridden
- It can be Overloaded
- It can have any of  the Four Access Modifies
- It cannot be synchronized
- It can have throws clause

# Instantiation with new

- It is the process by which individual objects are created.
  - **Class objectReference = new Constructor();**
- Declaration
  - Employee empObj;
- Instantiation
  - `empObj = new Employee()`
- Declaration and Instantiation
  - `Employee empObj = new Employee()`
  - **new** operator allocates memory for an object.

# Constructor Overloading

- One constructor can call another overloaded constructor of the same class by using ***this*** keyword.
- this() is used to call a constructor from another overloaded constructor in the same class
- The call to this() can be used only in a constructor ,and must be the first statement in a constructor
- A constructor can call its super class constructor by using the ***super*** keyword.
- A constructor can have a call to **super() or this() but never both**

# Overloaded Constructor

Time.java

```
class Time
{
  private int hour,min,sec;

  // Constructor
  Time()
  {
   hour = 0;
   min = 0;
   sec = 0;
  }

  //Overloaded constructor
  Time(int h, int m, int s)
  {
    hour = h;
    min = m;
    sec = s;
  }
}
  // Code continues …
```

# *this* keyword

- Is a reference to the object from which the method was invoked

*this* keyword

```
Time(int hour, int min, int sec)
  {
    this.hour = hour;
    this.min = min;
    this.sec = sec;
  }
```

# Modifiers for declarations

- There are Four Access Level and 3 Modifiers
- Any declaration can be preceded by
  - public
    - a declaration is accessible by any class
  - protected
    - Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
    - Works just like default , Except it also allows subclasses outside the package to inherit the protected thing.
  - default(no modifier)
    - Only accessible by classes, including subclasses, in the same package as its class(package accessibility).
  - private
    - a declaration is only accessible within the class in which it is declared

# Method Overloading

- If two (or more) methods of a class have the same name but different signatures, then the method name is said to be overloaded.
- The signature of a method consists of the name of the method and the number and types of formal parameters in particular order.
- Method overloading is method name overloading.
- Several methods that implement similar behavior but for different data types.
- They are independent methods of a class and can call each other just like any other method.
- A method *can* be overloaded in the same class or in a subclass.

# Overloading Methods

- Overloaded methods MUST change the argument list.
- Overloaded methods CAN change the return type.
- Overloaded methods CAN change the access modifier.
- Overloaded methods CAN declare new or broader checked exceptions.

# Overloading and AutoBoxing

```
public class Overloading {


    public Integer add(Integer a , Integer b)
    {
            Integer c = a+b;

            return c+100;
    }

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

# Main Method

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

    Overloading olObj = new Overloading();

    System.out.println(olObj.add(45, 55));



}
```

**Output will be 100 and Not 200**

# Static Variables and Methods

- A static method belongs to a class and not to its instance objects and hence they are shared between Objects
- Static Methods can not be overridden
- They can only call other *static* methods
- They can  access only static variables and not instance Variables
- They cannot refer to *this* or *super*
- **Instance variable : 1 per instance** and  **Static variable : 1 per class**
- Static final variables are constants
- *main( )* is defined to be a static method so as to be invoked directly

# Static Method access only static

```
public class StatClass {

   private int id;
   private static String name;

   private void instMethod()
   {
      staticMethod();
      System.out.println(id);
      System.out.println(name);
   }
   private static void staticMethod()
   {
   System.out.println(id);
   System.out.println(name);
   instMethod();
   }
}
```

Can Access Static from Instance Method

Cannot Access Instance Variable From Static

Cannot Access Instance Method From Static

# Static block

- Used to initialize static variables
  - Gets executed only once when the class is first loaded

StaticExample.java

```
class StaticExample{

    static int a,b;

    static{
        a = 9;
        b = 5;
    }

 // Other statements
}
```

# Static Import

```
import java.util.*;
import static java.lang.System.out;
import static java.lang.System.in;

public class StaticImport {

public static void main(String[] args) {

Scanner kb = new Scanner(in);
out.print("Enter an integer ");

int x = kb.nextInt();
out.print("Enter a double ");

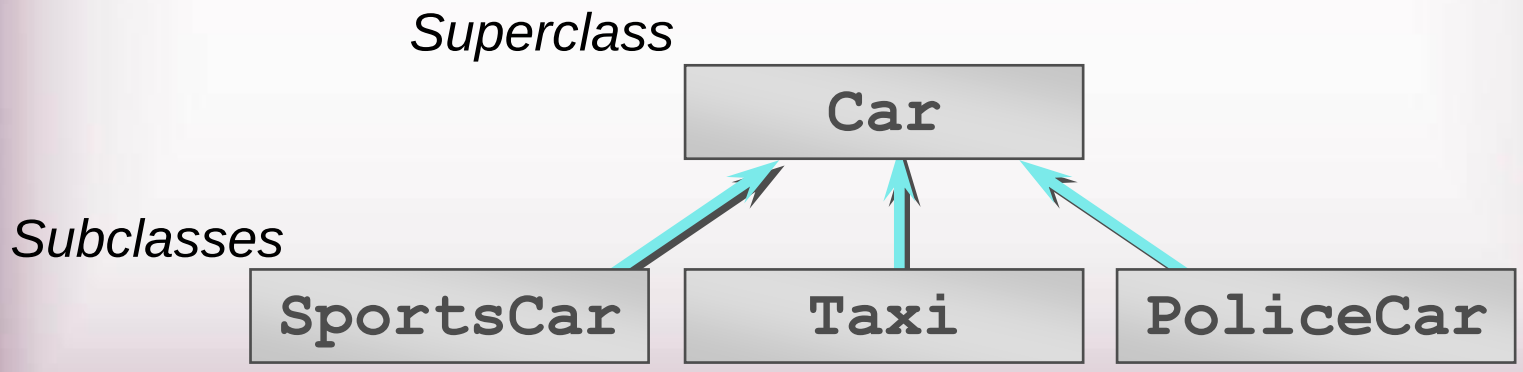
double d = kb.nextDouble();
out.println("The sum is " + (x+d));
}
}
```

# INHERITANCE

# Overview

- A class can inherit from another class
  - Original class is the “superclass”
  - New class is called the “subclass”

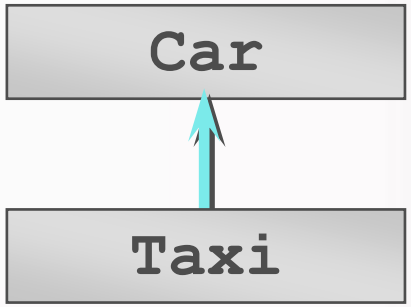
- Inheritance is a fundamental OO concept



# Example of Inheritance

– The Car class defines certain methods and variables

– `Taxi` extends `Car`, and can:
  - Add new variables
  - Add new methods
  - Override methods of the `Car` class



# Specifying Inheritance in Java

– Inheritance is achieved by specifying which superclass the subclass
"extends"

•Taxi inherits all the variables and methods of Car

```
public class Car {
   …
}
```

```
public class Taxi extends Car {
   …
}
```

# Aspects of Inheritance

- Objects
  - What does a subclass object look like?
- Construction
  - How do constructors work now?
- Methods
  - What methods can be called?
  - How can methods be overridden?

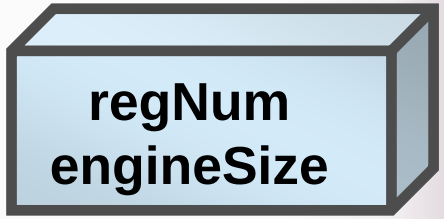
# What Does an Object Look Like?

A subclass inherits all the instance variables of its superclass

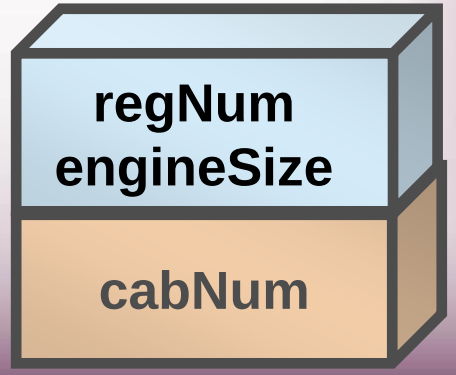
```
public class Car {
   String regNum;
    int engineSize; …
}
```

```
public class Taxi extends Car {
   private int cabNum; …
}
```

***Car* object**



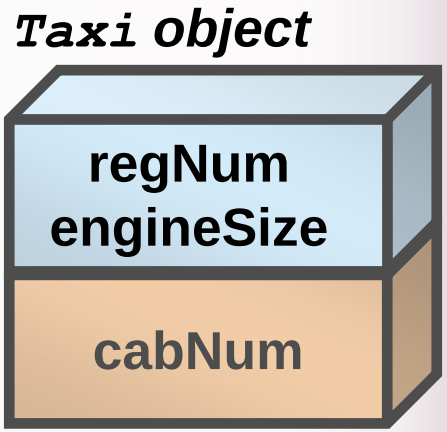

***Taxi* object**



# Default Initialization

– What happens when a subclass object is created?

```
Taxi taxi1 = new Taxi();
```




– If no constructors are defined:

– No-arg constructor is called in superclass

  - No-arg constructor called in subclass

# Nondefault Initialization

Specific initialization can be performed as follows:

```
public class Car {
  Car(String r, int e) {
    regNum = r;
    engineSize = e;
  } …
```

```
public class Taxi extends Car {
  Taxi(String r, int e, int c) {
    super(r, e);
    cabNum = c;
  } …
```

Use `super()` to call superclass constructor

# Specifying Additional Methods

- The superclass defines methods that are applicable for all kinds of Car
- The subclass can specify additional methods that are specific to Taxis

```
public class Car {
   public int getReg()…
   public void changeOwner()…
   …
```

```
public class Taxi extends Car {
   public void renewCabLicense()…
   public boolean isBooked()…
   …
```

# Overriding Superclass Methods

- A subclass inherits all the methods of its superclass
- The subclass can override a method with a specialized version, if it needs to:

```
public class Car {
  public String details() {
    return "Reg:" + getReg();
  }
```

```
public class Taxi extends Car {
  public String details() {
    return "Reg:" + getReg() + cabNum;
  }
```

# Overriding

- When a sub-class defines a
  - “method with ***same method name, argument types, argument order and return type as a method in its super class***, its called method overriding. “
- Methods declared as **final, static and private  cannot be overridden**.
- An overriding method can be declared as final
- The method can’t be less accessible
  - **Public  =>  only public**
  - **Protected => Both Public and Protected**
  - **Default => default, public and protected**

# Overriding

*extends* keyword

```
class Base
{
      protected int a;
      base()
      {
        a = 20;
      }
      void display()
      {
        System.out.print("a = "+a);
      }
}
class Derived extends Base
{
      private int b;
      derived()
      {
       b = 25;
      }
      void display()
      {
        System.out.print("b = "+b);
      }
}
```

# Overriding with Compatible Return type

- Arguments must be same and return type must be compatible

```
class Base
{
   private int i = 5;

   public Number  getNumber()
   {
     return i;
   }
}

class Derived extends Base
{
   @Override
   public Integer getNumber()
   {
     return new Integer(10);
   }
}
```

# Invoking Superclass Methods

- If a subclass overrides a method, it can still call the original superclass method
- Use super.method()to call a superclass method from the subclass
- Though keyword super is used to refer super class, method call super.super.method() is invalid.

```
public class Car {
  public String details() {
    return "Reg:" + getReg();
  }
```

```
public class Taxi extends Car {
  public String details() {
    return super.details() + cabNum;
  }
```

# *super* keyword

- The keyword *super* refers to the base class
  - *super( )*
    - invokes the base class constructor
    - base class constructors are automatically invoked
  - *super.method( )*
    - invokes the specified method in the base class
  - *super.variable*
    - to access the specified variable in the base class
- ***super*** `must be the first statement in a constructor`

# Overriding –When Method Has Exceptions

- Any exceptions declared in overriding method must be of the same type as those thrown by the super class, or a subclass of that type.

```
class MyBase {

  public void method1() throws Exception
    {  }
  public void method2() throws Runtim
    {  }
}
class Sub extends MyBase
{
  public void method1() throws Throwable {
    }
  public void method2() throws
                        ArithmeticExc
  }
}
```

This is compile Time Exception

This is Allowed

# Hiding-Fields and Static Members

- A Sub Class can hide the fields but cannot override that of the super class, its done by defining fields with the same name as in the super class.
- Code in the subclass can use the keyword super to access such members
- A static method can hide a static method from the Super Class
- A hidden super class static method is not inherited., will result in compile time Exception

# Hiding-Fields and Static Members

Variable Hiding

```
class MyBase
{
   int var1=100 ;

   public static void method1()
   {
      System.out.println("Super
   Class        Method1");

   }
   public void method2()
   {
     System.out.println("Super
   Class method2 ");
   }
}
```

```
class Mysub extends MyBase
{
    int var1 =200;

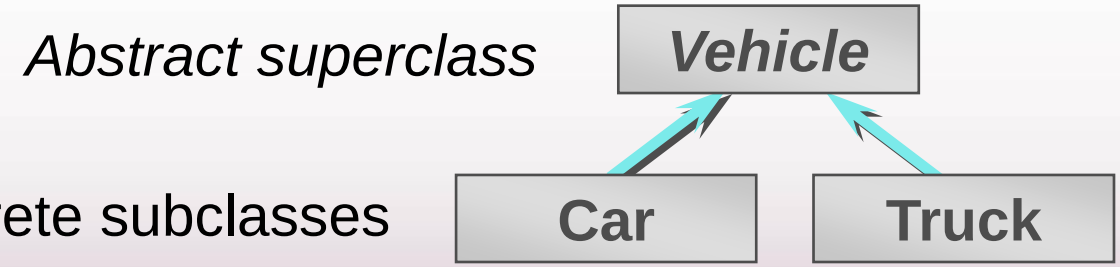
   public static void method1()
    {
System.out.println("Sub Class
  Method1");
    }
 public void method2()
    {
System.out.println("Sub Class
  Method2");

System.out.println("Var"+var1
  +"superclass
  Var"+super.var1);
    } }
```

Metho Hiding

# Abstract Class

- An abstract class is a shared superclass
  - Provides a high-level partial implementation of some concept
  - Cannot be instantiated
- Use the abstract keyword to declare a class as abstract



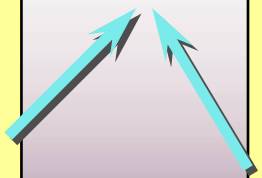
# Defining Abstract Classes in Java

- Use the `abstract` keyword to declare a method as abstract
  - Only provide the method signature
  - Class must also be abstract
- Must be implemented by a concrete subclass
  - Each concrete subclass can implement the method differently

```
public abstract class Vehicle {
   private String regNum;
   public String getRegNum()…
   public abstract boolean
                        isRoadWorthy();
}
```

```
public class Car
        extends Vehicle{
 private int numSeats;
 public void fitRack()…
```

```
public class Truck
        extends Vehicle{
 private int axles;
 public void setLoad()…
```



# The Abstract Class-Super Class

Note this

```
public abstract class BankAccount {

   public abstract void deposit(float amount);

   public abstract void withdraw(float amount);

   public void sayHello() {

            System.out.println("Thanks-Come Again");
      }

}
```

# Child Class –Its alsoAbstract

```
public abstract class Savingaccount extends BankAccount
   {

   public abstract void getClientdetails();
}
```

# The Concrete Class

```
public class Supersavings extends Savingaccount {
  float balance,amount ;
  String custname;
  int accno;
    public void deposit(float amt)
   {
     balance+=amt;
    }
   public void withdraw(float amt)
   {
    balance-=amt;
 }
   public void getClientdetails()
   {
    System.out.println(custname);
    System.out.println(accno);
    System.out.println(amount);
    System.out.println(balance);
   }
}
```

# Interfaces & Polymorphism

# Interfaces

- An interface is like a fully abstract class
  - All of its methods are public and abstract
  - No instance variables if present they are public static final
- An interface defines a set of methods that other classes can implement
  - A class can *implement* many interfaces, but can *extend* only one class
- Can extend other interfaces and extend multiple interfaces
- Extends another sub interface  inherits everything from super interface

# Interface

- Interfaces define types in an abstract form
  - An interface declaration allows the specification of a reference type without providing an implementation
  - The class that implements the interface decides how to implement

*implements* keyword

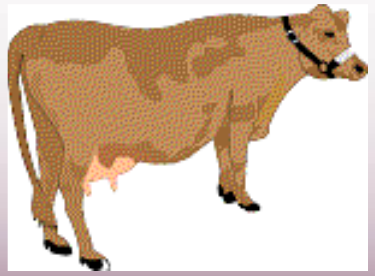
```
// the interface
interface Sort
{
    void do_sort();
}

// the implementor
class BubbleSort implements Sort
{
    public void do_sort()
    {
        // The sorting method implementation
    }
}
```

# Example of Interfaces

- Interfaces describe an aspect of behavior that different classes can support
- For example, any class that supports the “Steerable” interface can be steered:

***Steerable***



***Not Steerable***

# Defining an Interface in Java

– Use `interface` keyword instead of `class`

```
public interface Steerable {

  int maxTurn = 135;
  void turnLeft(int deg);
  void turnRight(int deg);
}
```

# Implementing an Interface

```
public interface Steerable {
   int maxTurn = 135;
   void turnLeft(int deg);
   void turnRight(int deg);
}
```

```
public class Car
        extends Vehicle
        implements Steerable {

   public void turnLeft(int deg)
   {…}

   public void turnRight(int deg)
   {…}
```

# Partial Implementation of an Interface

- Declare the class as abstract if the class does not implement all the methods of an interface

```
Public abstract class Car
                  extends Vehicle
                     implements Steerable {

  public void turnLeft(int deg)
  {…}

}
```

# Interface

- A single class can implement many interfaces

*implements* keyword

```
interface Sort{

    void doSort();
}

interface List{

    void createList();
}

class DataUtility implements Sort, List{

    public void doSort(){
        // Statements
    }

    public void createList(){
        // Statements
    }
}
```

# Using `instanceof` with Interfaces

- The `instanceof` operator can be used to check if an object implements an interface
- Downcasting can be used if necessary, to call methods defined in the interface

```
public void aMethod(Object obj) {
   …
   if (obj instanceof Steerable)
      ((Steerable)obj).turnLeft();
}
```

# Extending an Interface

- One Interface can extend another Interface

```
public interface Steerable {
   int maxTurn = 135;
   void turnLeft(int deg);
   void turnRight(int deg);
}
```

```
public interface Navigator
                    extends Steerable {
   void calcSpeed(int distance);

}
```

# Polymorphism

- This OO Principle allows the programmer to program abstractly
- Objects of a subclass can be treated as objects of its superclass
- The base class of a hierarchy is usually an abstract class or an Interface
  - They define the common behavior (functionality)
- Classes and Subclasses inherit this behavior and extend it according to their own properties by overriding the methods of the super class
- **Subclass objects can be treated as super class** objects through references,
  - But, **super class objects are not subclass objects**

# Polymorphism

- Allows for hierarchies to be highly extensible
- New classes can be introduced and can still be processed without changing other parts of the program
- Same method can do different things, depending on the class that implements it.
- Method invoked is associated with OBJECT and not with reference.
- Types of Polymorphism -
  - Method Overloading.
  - Method Overriding.

# Polymorphism

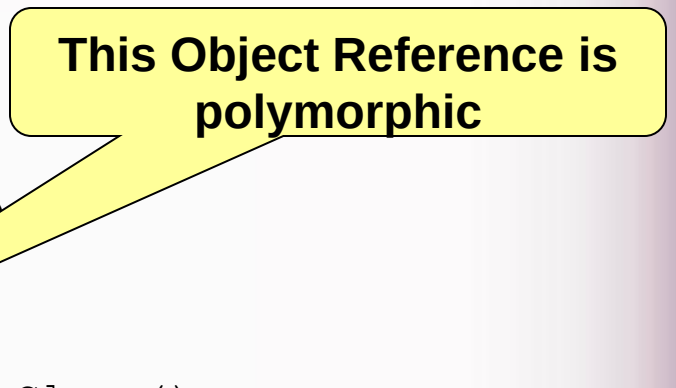
```
class SuperClass
{
 int value = 100;

void callMe()
{
   System.out.println("I am in Super Class");
  }
}

class SubClass extends SuperClass
{
int value = 10;

void callMe()
{
System.out.println("I am in Sub Class");
  }
}
```

# Polymorphism




```
public class TestDMD
{
public static void main(String s[])
{

 SuperClass objSuperClass = new SubClass();

  objSuperClass.callMe();

  System.out.println(objSuperClass.value);

  }

}
```

- This means object variables are polymorphic , A variable of type Super Class  can refer to an object of type Super class as well as subclass.

# Substitution of Object References in Method Calls

- A subclass object can be passed into any method that expects a superclass object

- The method that got invoked, is the version that's present in the object type and NOT the reference type.

```
public static void main(String[] args) {
  Taxi t = new Taxi("ABC123", 2000, 79);
  displayDetails(t);

public static void displayDetails(Car c) {
  System.out.println(c.details());
}
```

# Dynamic Method Dispatch

- **Related Through Inheritance**

- *superclass = subclass*
  - always valid
- *subclass=(subclass)superclass*
  - valid at compile time, checked at runtime. if is invalid then the exception ClassCastException is thrown.
- *subclass = superclass*
  - not valid at compile time, needs a cast

- **Unrelated Classes – Not Allowed**

- someClass = someUnrelatedClass
  - not valid, won't compile
- somcClass = (someClass)someUnrelatedClass
  - not valid, won't compile

# Dynamic Method Dispatch

```
class First
{
  public void show()
  {
      System.out.println("ShowFirst");
  }
}

class Second extends First
{
    public void show()
    {
System.out.println("Show Second");
    }
}
```

```
First fst = new Second();

Second sec = (Second)fst;

sec.show();

First fst2 = new First();

Second sec2 = (Second)fst2;

sec2.show();
```

***subclass= (subclass)superclass***

***Valid at both Compile ,RunTime***

***subclass= (subclass) erclass***

***Valid at Compile exception ,RunTime***

# Packages

- Packages are containers of classes
- An organized collection of classes that are logically related
- Mechanism for partitioning the class name space
- Helps Distributed Computing
- Some of Packages in Java are
  - java.lang
  - java.io
  - java.util
  - java.net
- User defined classes can also be organized into packages

# Package

- In the following example, the class *class1* is part of the package *pack1*
- Use the *javac -d* option to compile

*package* keyword

```
package utility;
// indicates that class Sort is part of Utility package

class Sort
{
   public void do_sort()
   {
      // Statements
   }
}
```

# Package

- Use the *import* keyword to access the classes in a package

*import* keyword

```
import utility.*;

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

    Sort obj1;

    // Code continues …
```

# toString( )

- The objectS can be passed to
  - System.out.println( )
  - Methods
  - Or Can be used string concatenation,
- Java automatically calls its toString( ) method.

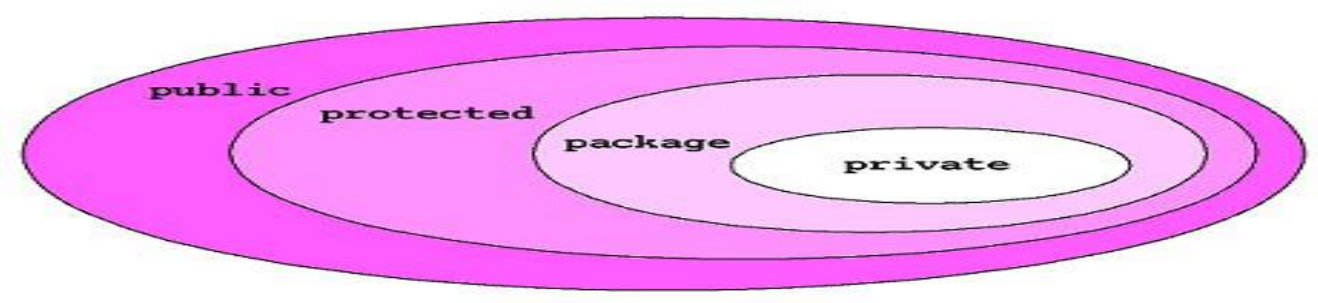
- The toString() just prints the class name, an @ sign, and the object's memory address -hashCode) value

- toString( ) implementation prints the  state of the object  which will result in formatting control in println( ) and anywhere  objects get referred to in a String context.

# CLASSPATH

- Is an environment variable which specifies the location of the class files
- For compiling the *Sort.java* (listed earlier)
  - javac -d c:\packages Sort.java
  - this would create a sub-directory *Utility* under c:\packages and store *Sort.class* over there
- While compiling *test.java* (listed earlier), set
  - CLASSPATH = . ; c:\packages
- Any package can be included in the working package
- Include all packages in CLASSPATH
    - import package_name.class_name;
    - import package_name.subpackage_name.class_name;
    - import package_name.*;

# Package - Access modifiers




| | *Private* | *No modifier* | *Protected* | *Public* |
|---|---|---|---|---|
| *Same class* | Yes | Yes | Yes | Yes |
| *Same package - subclass* | No | Yes | Yes | Yes |
| *Same package - nonsubclass* | No | Yes | Yes | Yes |
| *Different package - subclass* | No | No | Yes | Yes |
| *Different package - nonsubclass* | No | No | No | Yes |

# *final* keyword

- A final method or class is Java's way of prohibiting inheritance and/or overriding
- Final methods cannot be overridden by subclasses
- All private methods are implicitly final
- Final classes Cannot be extended and their methods are implicitly final
- Can be applied to variables too
  - The value of a *final* variable cannot be altered
  - *final int max_limit = 23*
  - *final void method( )*
  - *final class last_class*

# Interface vs Abstract Class

- A class may implement several interface
- An interface cannot provide any code at all
- Interfaces are often used to describe the peripheral abilities of a class, not its central identity,
- A class may extend only one class
- An abstract class can provide complete or partial code
- An abstract class defines the core identity of its descendants.

# Interface, Abstract Class,Subclass,Class

- A plain class is designed when in doesn't pass the “is-a” test.
- A Plain class is designed when its not extending any other class other than object class

- Make a subclass , when it needs to extend a super class
- Make a subclass, when it needs to override or add new behaviors

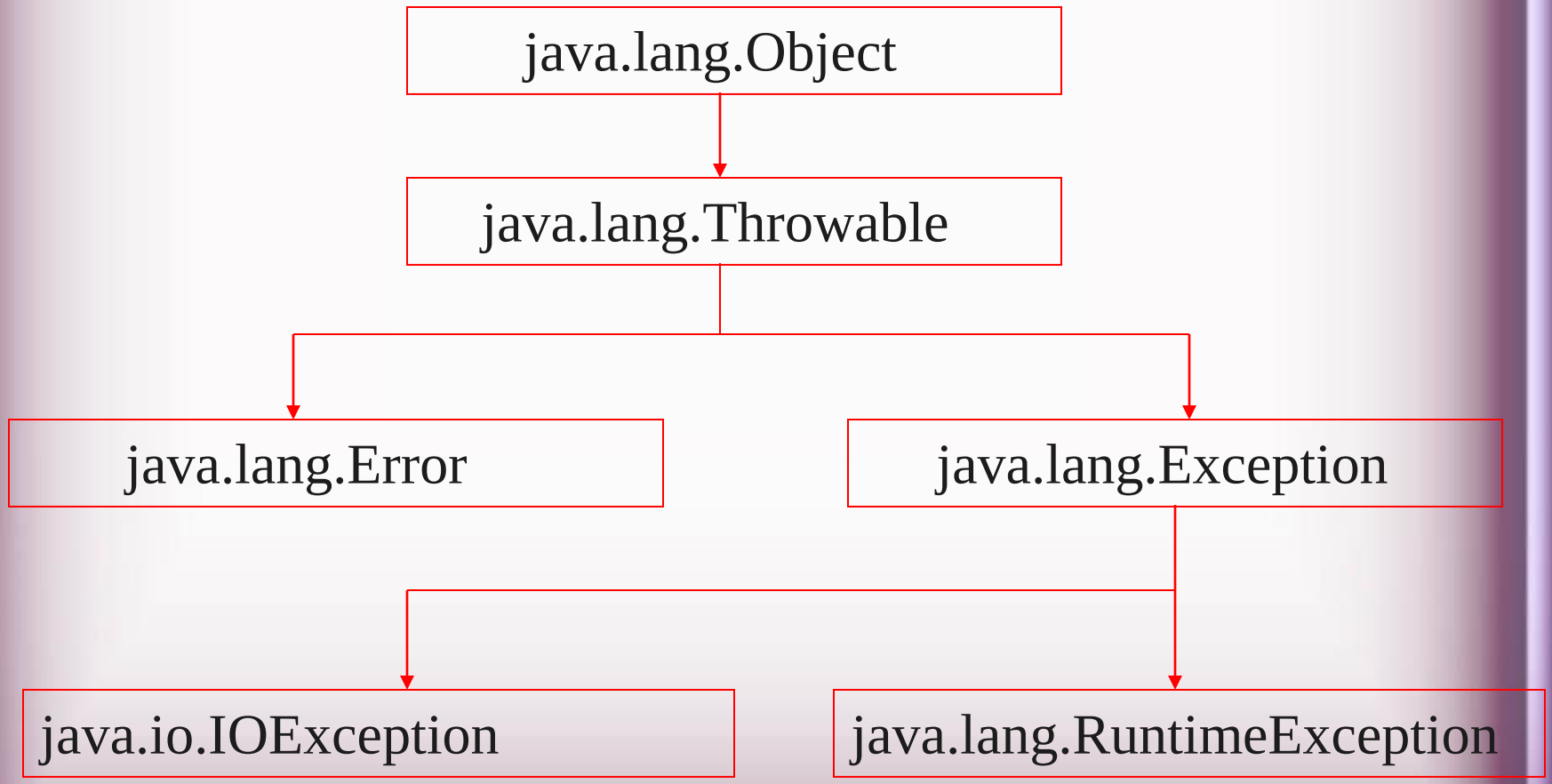
- Make an abstract class , when there is a requirement for a kind of “template” for a group of subclasses, and there are some implementation in the class which the subclass can use .
- Make an abstract class , to guarantee that nobody can make objects of that type.
- Design an interface , to define a role that other classes can play, regardless of the inheritance tree,

# Exception handling

# Topics

- Define exceptions
- Use try, catch, and finally statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions
- Distinguish appropriate and inappropriate uses of assertions
- Enable assertions at runtime

# Exception Handling



# Exceptions

- Conditions that can readily occur in a correct program are *checked exceptions.*
- These are represented by the Exception class.
- Problems that  reflect program bugs are *unchecked exceptions.*
- Fatal situations are represented by the Error class.
- Probable bugs are represented by the RuntimeException class.
- The API documentation shows checked exceptions that can be thrown from a method

# Handle or Declare Rule

- Handle the exception by using the try-catch-finally block.
- Declare that the code causes an exception by using the throws clause.
- void trouble() **throws IOException { ... }**
- void trouble() **throws IOException, MyException { ... }**
- Other Principles
- You do not need to declare runtime exceptions or errors.
- You can choose to handle runtime exceptions

# Example of Exception

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

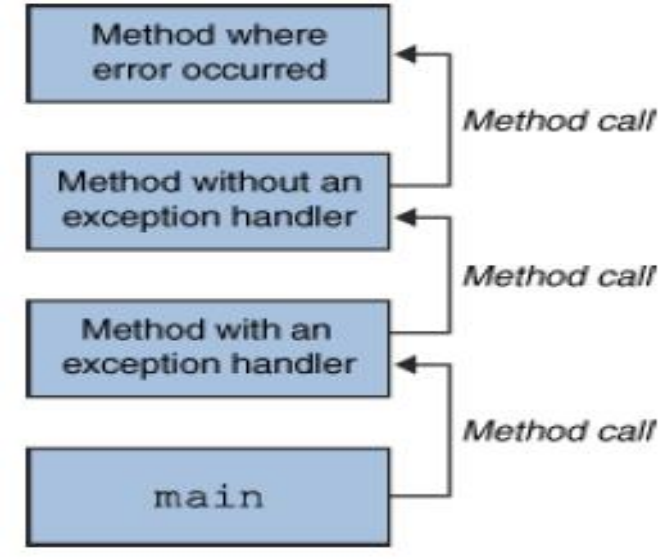
     System.out.println(3/0);
     System.out.println(“Hi”);

        }
   }
}
```

- Default exception handler
  - Provided by Java runtime ,Prints out exception description
  - Prints the stack trace
- Hierarchy of methods where the exception occurred
  - Causes the program to terminate

# What Happens When an Exception Occurs

- When an exception occurs within a method,
  - the method creates an exception object and hands it off to the runtime system - “throwing an exception”
  - These Exception object contains information about the error, including its type and the state of the program when the error occurred
  - The runtime system searches the call stack for a method that contains an exception handler



# What Happens When an Exception Occurs

- When an appropriate handler is found, the runtime system passes the exception to the handler
  - An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
  - The exception handler chosen is said to catch the exception.

- If appropriate exception handler is not found by the runtime system the application terminates and uses the default exception handler

# Checked and Unchecked

- Checked exceptions include all subtypes of Exception, *excluding* classes that extend Runtime Exception.

- Checked exceptions are subject to the *handle or declare* rule; any method that *might* throw a checked exception
  - must either declare the exception using the throws keyword, or handle the exception with an appropriate *try/catch*.

- Subtypes of Error or Runtime Exception are unchecked, so the compiler doesn’t enforce the handle or declare rule.

# Catching Exceptions

```
class DivByZero {
 public static void main(String args[]) {
 try {
  System.out.println(3/0);
  System.out.println("Please print me.");
  } catch (ArithmeticException exc) {
    //Division by zero is an ArithmeticException
          System.out.println(exc);
    }
    System.out.println("After exception.");
  }
 }
```

# Method Overriding and Exceptions

- **The overriding method can throw:**

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

- **The overriding method cannot throw:**

- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method

# Catching Exception –Multiple Catch

```
class MultipleCatch {
public static void main(String args[]) {
try {
int den = Integer.parseInt(args[0]);
   System.out.println(3/den);
}
catch (ArithmeticException exc)
{
   System.out.println("Divisor was 0.");
}
catch (ArrayIndexOutOfBoundsException exc2)
{
   System.out.println("Missing argument.");
  }
   System.out.println("After exception.");
  }
}
}
```

# Catching Exception –Nested Try's

```
    public static void main(String args[]){
try {
       int a = Integer.parseInt(args[0]);
    try {
          int b = Integer.parseInt(args[1]);
          System.out.println(a/b);
        }
     catch (ArithmeticException e1)
      {
         System.out.println("Div by zero error!");
      }
     }
catch(ArrayIndexOutOfBoundsException e2)
     {
      System.out.println("Need 2 parameters!");
      }
    }
}
```

# Finally block

- Its optional and will always be invoked,
  - whether an exception in the corresponding *try* is thrown or not,
  - whether a thrown exception is caught or not.

- *finally*-will-always-be-called except if the JVM shuts down. When the try or catch blocks may call System.exit();

- Block of code is always executed despite of different scenarios:
  - Forced exit occurs using a *return*, a *continue* or a *break* statement
  - Normal completion
  - Caught exception thrown
  - Exception was thrown and caught in the method
  - Uncaught exception thrown
  - Exception thrown was not specified in any catch block in the method

# Throwing Exceptions

- **The *throw* Keyword**
- Java allows you to throw exceptions (generate exceptions)
  - **throw <exception object>**
- An exception you throw is an object
  - You have to create an exception object in the same way you create any other object
- Example:
  - **throw new ArithmeticException(“testing...”);**

# Throws Clause

- A method is required to either catch or list all exceptions it might throw
  - Except for *Error* or *RuntimeException*, or their subclasses

- If a method may cause an exception to occur but does not catch it, then it must say so using the *throws* keyword

- Applies to checked exceptions only

```
<type> <methodName> (<parameterList>)  throws <exceptionList>
{
    <methodBody>
}
```

# Uncaught Exception

- Uncaught exceptions propagate back through the call stack,
- Starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown

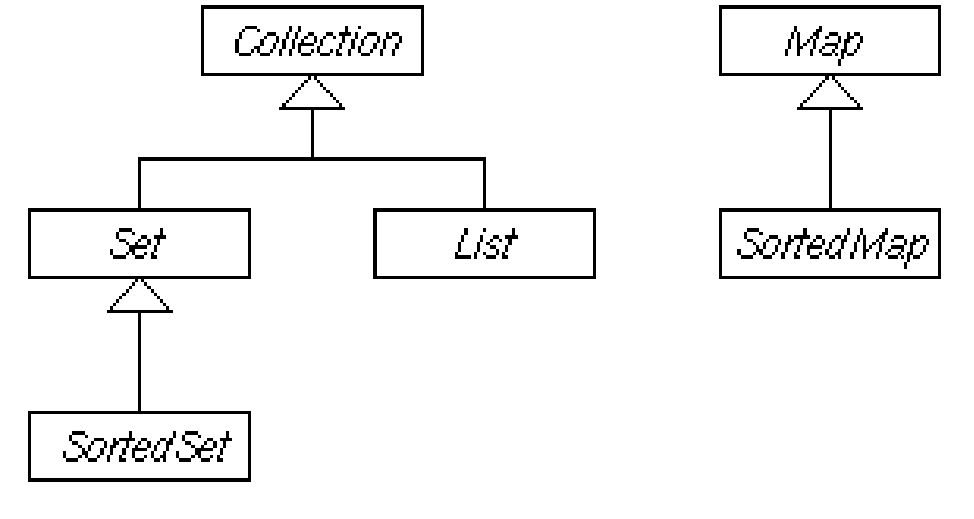
# Effective Exception Hierarchy

- The exception classes are arranged in a hierarchy.
- Catches for specific exceptions should always be written prior to the generic handler.
- FileNotFoundException is a child of IOException, a catch for FileNotFoundException should occur before the one for IOException.
- A generic handler for Exception would cover any missing situation.

# Java Collections Framework

# Java Collection Framework

- Provides tools to maintain data container of object
- Is an alternative to the creation of custom data structures
- Consists of interfaces and classes that implement inerfaces within the java.util package
- Collection is the root interface for most of the Java collections hierarchy



# Collection categories

- **Ordered Collection**
  - The objects within the collection are maintained in a particular order or not
  - When an object is added or removed from an ordered collection , the order is automatically maintained
- **Duplicate Elements**
  - Whether or not the collection allows duplicate objects
  - Collection will automatically reject an attempt to add an object
- **Mapping of Key & Value**
  - Whether or not the collection maps a key to a particular object .
  - This provides a means of quickly locating the object.
  - Such collections do not allow duplicate objects

# Collection Interface

- The root interface in the *collection hierarchy*.
- Represents a group of objects, known as its *elements*.
- There is no *direct* implementation the more specific sub-interfaces are Set and List.
- This interface is typically used to pass collections around and manipulate them

# List  Interface

- can contain only Objects
- An ordered collection
- Elements can be accessed by their integer index (position in the list)
- Lists typically allow duplicate and null elements.
- Iterating over the elements in a list is typically preferable to indexing

# Methods in List Interface

- void **add**(object o)
- void **addAll**(Collection c)
- void **clear()**
- boolean **contains**(object)
- Object **get**(int index)
- int **indexOf**(object o)
- ListIterator **listIterator**()
- Object **set** (int index, Object element)

# List Implementations

- Array List
  - a resizable-array implementation like Vector
    - unsynchronized
- Linked List
  - a doubly-linked list implementation
  - if elements frequently inserted/deleted within the List
    - May provide better performance than ArrayList
  - For queues and double-ended queues (deques)
- Vector
  - a synchronized resizable-array implementation of a List with additional "legacy" methods.

# Class ArrayList

- Resizable-array implementation of the List interface.
- permits all elements, including null.
- Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. , its capacity grows automatically.

# Class ArrayList

- **ArrayList**()
  Constructs an empty list with an initial capacity of ten.

- **ArrayList**(Collection c)
  Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

- **ArrayList**(int initialCapacity)
  Constructs an empty list with the specified initial capacity.

# Generics

- Introduced in Java 5.0
- Can put **any** Object in to Collections, because they hold Just Objects
- Type-safe Collections are created ,ensuring  type-safety at compile time rather than run-time
- Creating an Instance of Classes of Generic Type
  - `new ArrayList<Employee>`
  - `ArrayList<Employee> mylist = new ArrayList<Employee>();`

# ArrayList – Creating ArrayList

```
ArrayList<Employee> alist=new ArrayList<Employee>();
```

```
Employee e1 = new Employee(101,”Ramesh”);
```

```
    alist.add(e1);
```

# Two Schemes of Traversing Collections

- **for-each**
  - The for-each construct allows you to concisely traverse a collection or array using a for loop
- **for (Object o: collection)**
  - System.out.println(o);
- **Iterator**
  - An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

# ArrayList – Enhanced For Loop

```
public void ehForLoop()
   {
     for(Employee e :alist)
     {
      System.out.println(e.getEmpname());
         System.out.println(e.getEmpno());
     }
   }
```

# The Iterator Interface

- Provides for the one-way traversal of a Set or SortedSet collection (forward only)
- Has several required methods but the most frequently used are:

| Method | Usage |
| --- | --- |
| `hasNext()` | **Returns `true` if the iteration has more elements** |
| `next()` | **Returns the next element in the iteration (as a generic `Object`)** |

# Array List –with Iterator

```
public void dispalyArrayList()
   {
        Iterator<Employee> itr = alist.iterator();
        while(itr.hasNext())
        {
               Employee empObj = itr.next();
               System.out.println(empObj.getEmpname());
               System.out.println(empObj.getEmpno());

        }
   }
```

# The ListIterator Interface

- Extends the Iterator interface to provide for the two-way traversal of a List collection (forward and backward)
- The most frequently used are:

| Method | Usage |
|---|---|
| haxNext() | Returns `true` if this list iterator has more elements when traversing the list in the forward direction |
| hasPrevious() | Returns `true` if this list iterator has more elements when traversing the list in the backward direction |
| next() | Returns the next element in the list (as a generic `Object`) |
| previous() | Returns the previous element in the list (as a generic `Object`) |