


# Programming with Java

# Sujata Batra - sujatabatra@hotmail.com

- IT Trainer Since 2000
- Conducting Java Training Since 2013
- 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

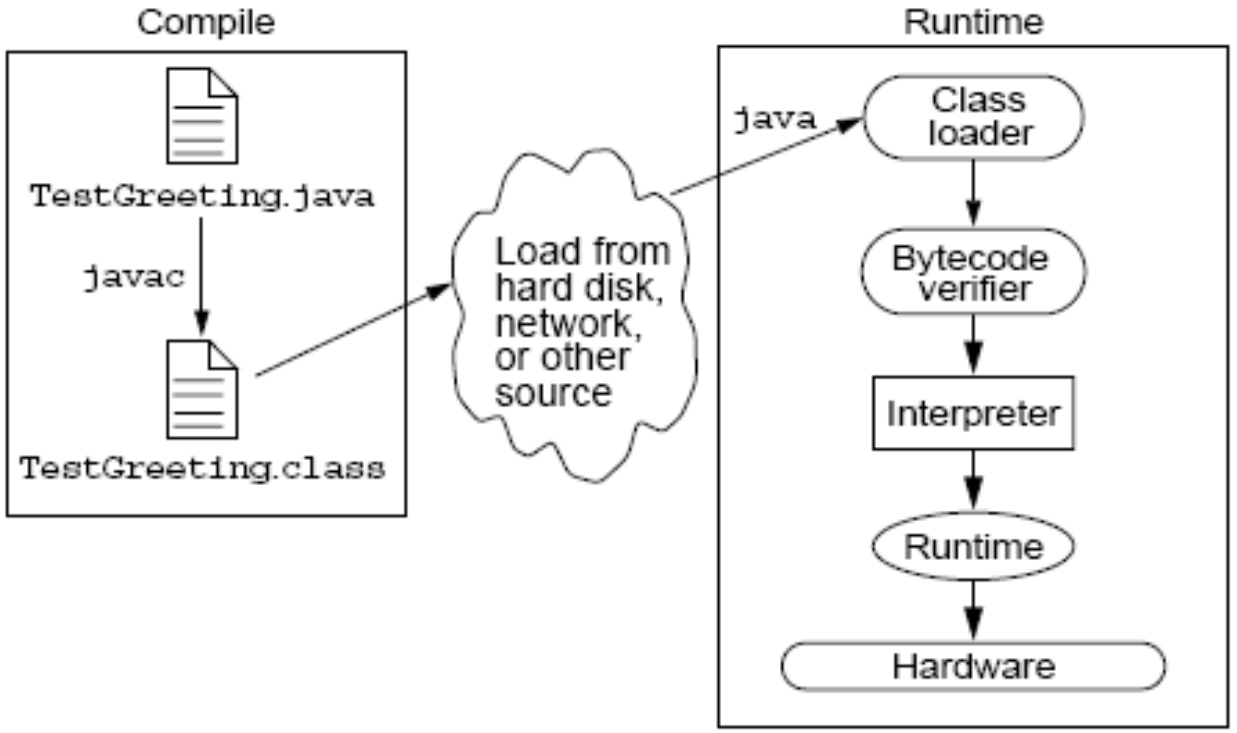
- Conceived as Programming language for embedded systems like microwave ovens, televisions etc
- One of the first projects developed using Java
    - personal hand-held remote control named Star 7.
- The original language was called Oak
- Java was developed in the year 1991 at Sun Microsystems
    -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

# Primary Goals of Java

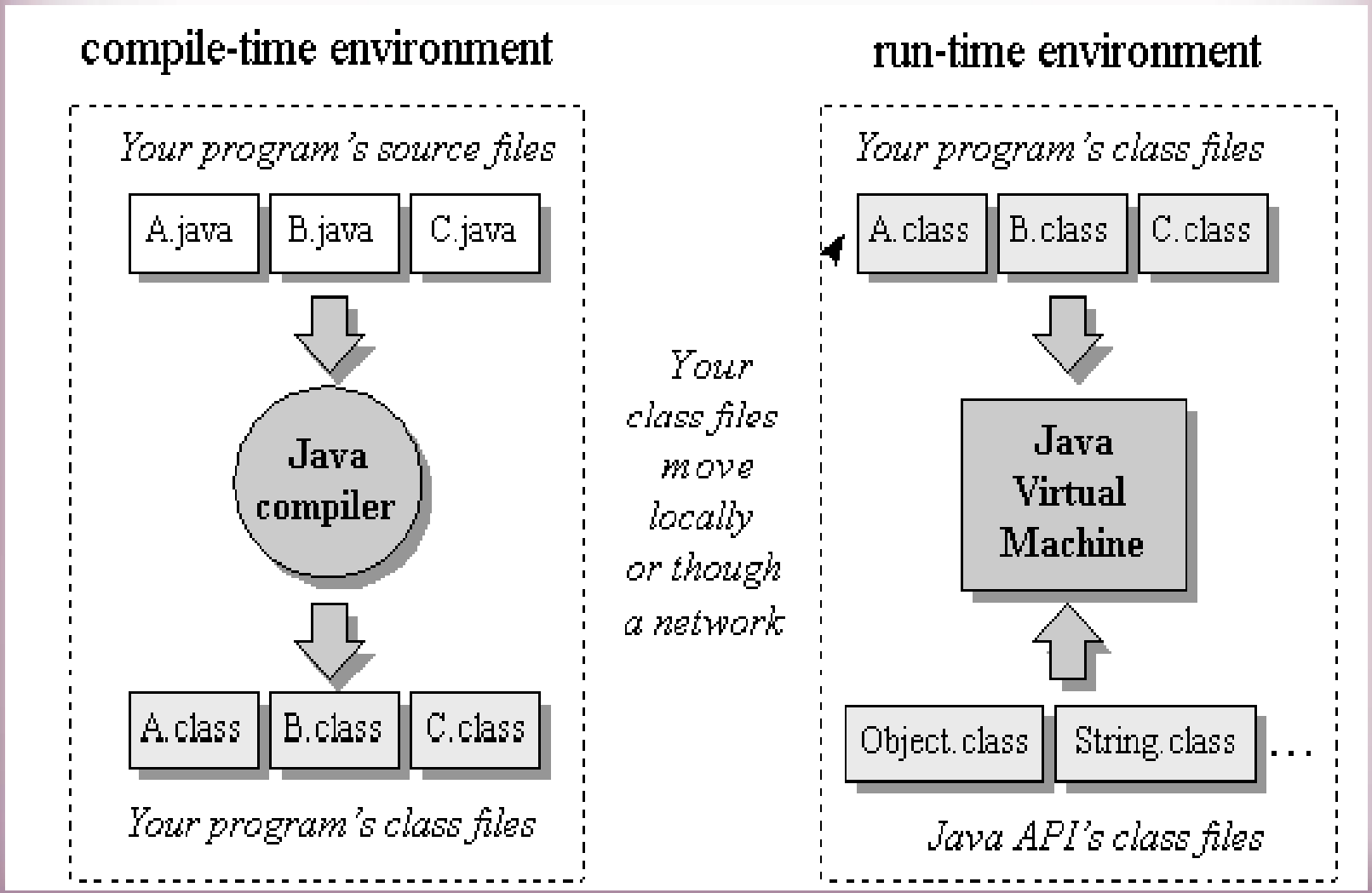
- **Provides an easy-to-use language by:**
  - Avoiding many pitfalls of other languages
  - Being object-oriented
  - Enabling users to create streamlined and clear code
- **Provides an interpreted environment for**:
  - Improved speed of development
  - Code portability
  - Enables users to run more 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

# Java Run Time Environment

The Java application environment performs as follows:

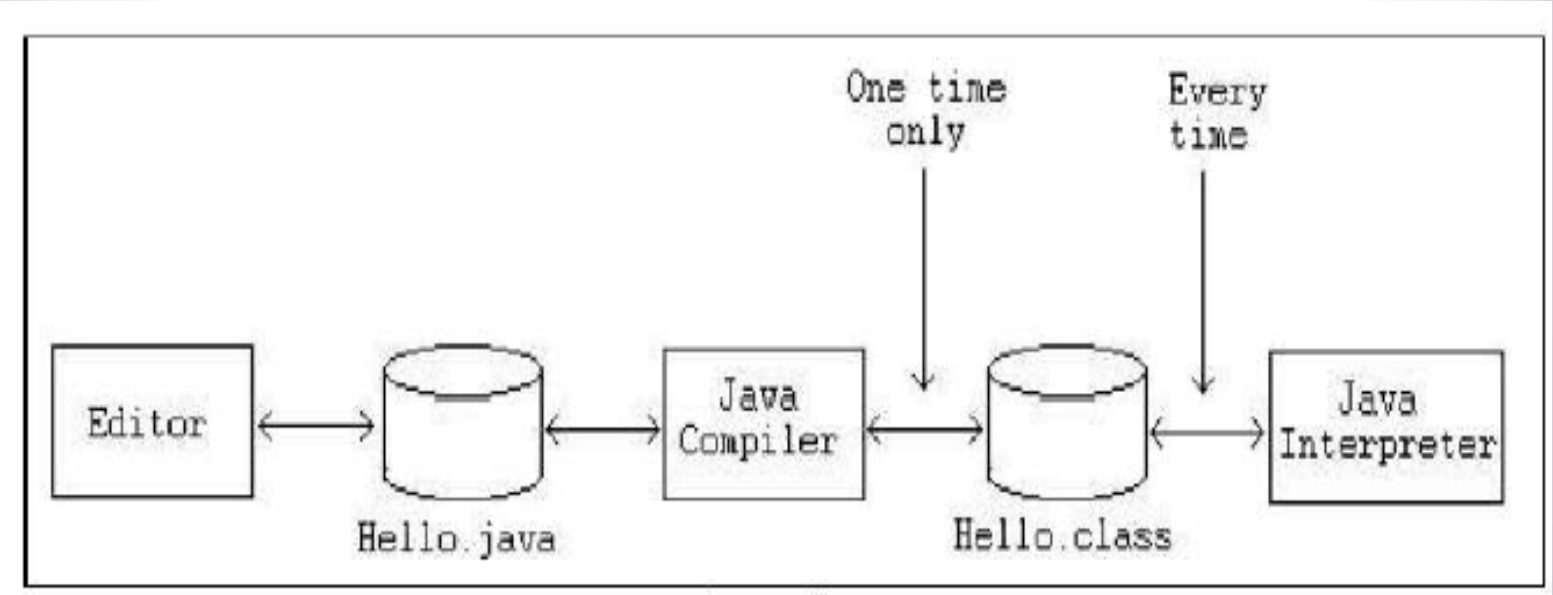


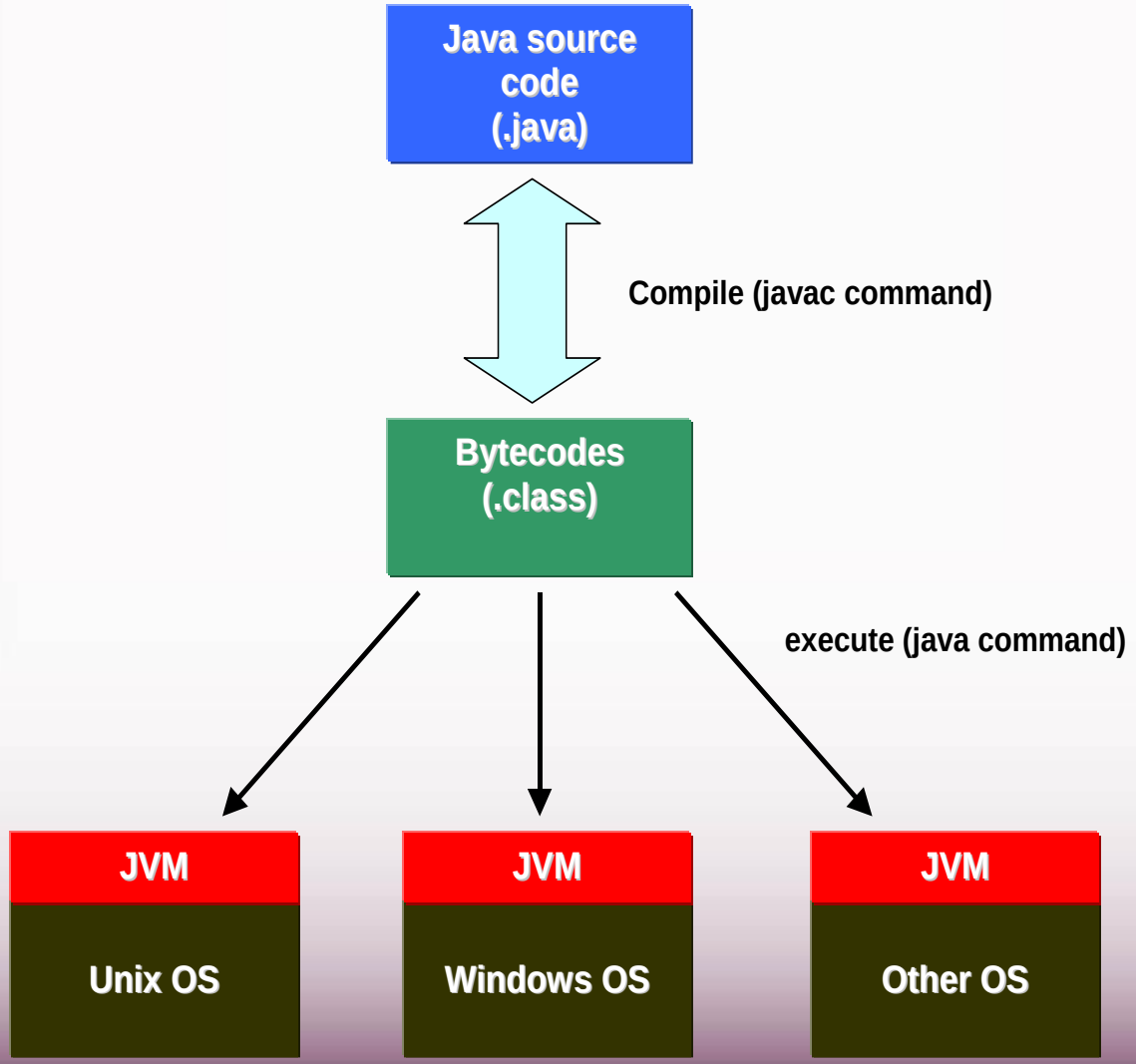
# Compile time and Runtime Environment



# 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

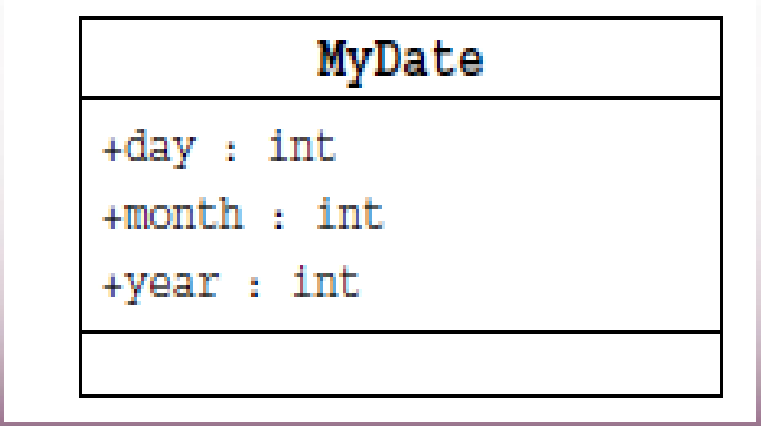
- The *dot notation is:* ***<object>.<member>***
  - used to access object members, including attributes and methods.

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

# Information Hiding

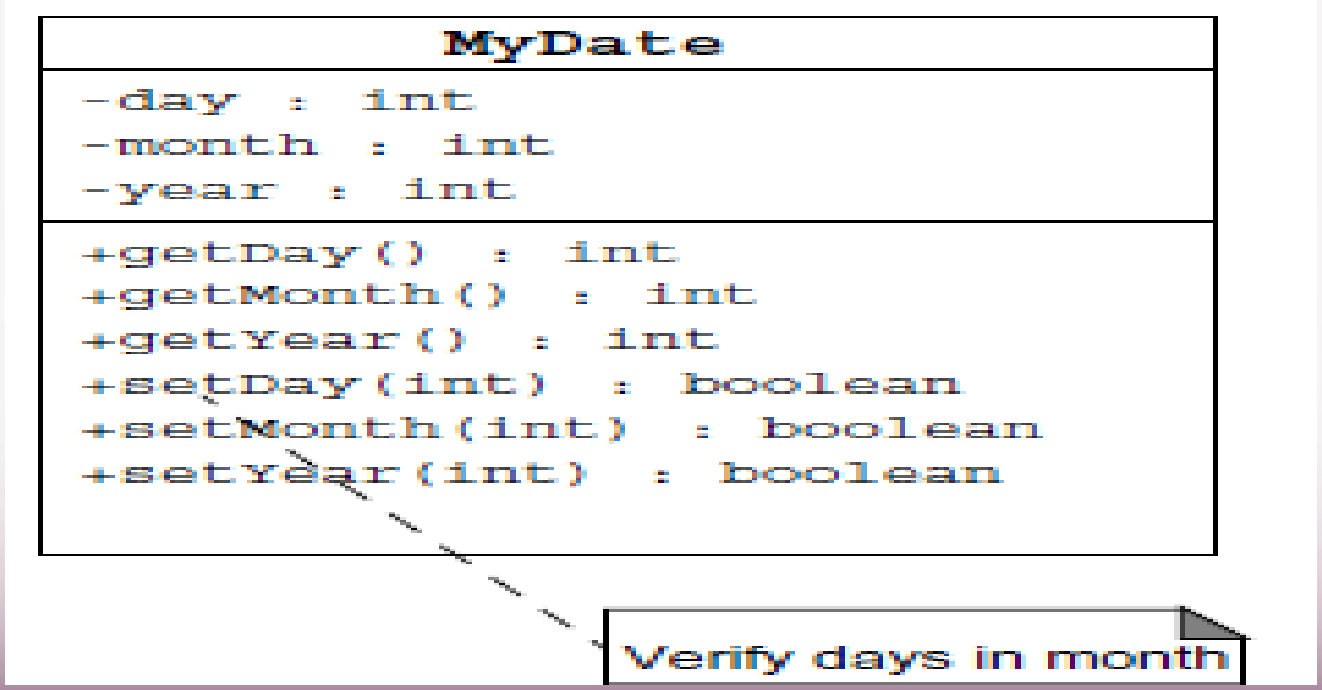
- Client code has direct access to internal data
- (d refers to a MyDate object):

- 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.

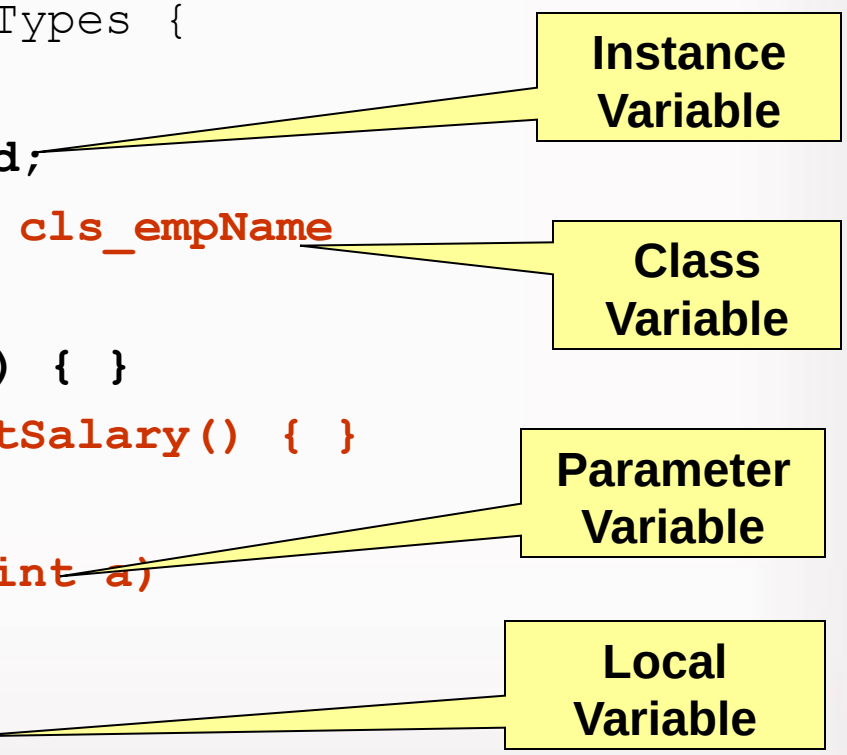
# Example for Variables

```
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

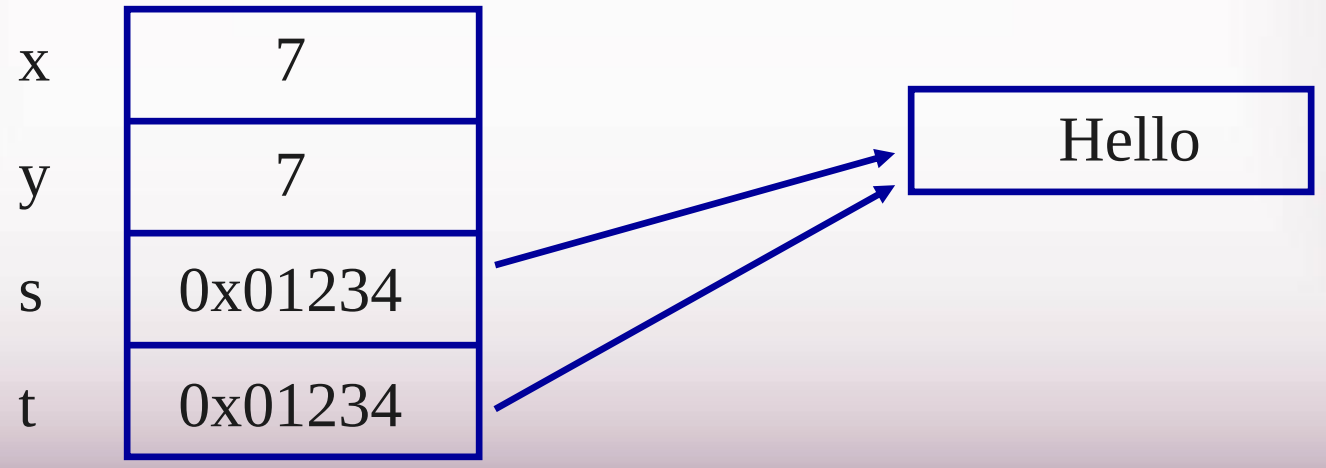
```
class Values
{
   private int a;
   private float b;
   private String c;
   public void display()
   {
        System.out.println("integer"+a);
        System.out.println("float"+b);
        System.out.println("String"+c);
}
public class DefaultVales {

   public static void main(String[] args) {

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

# Assignment of reference variables

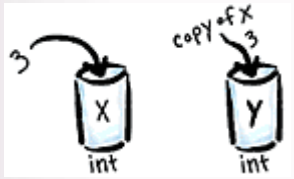
```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

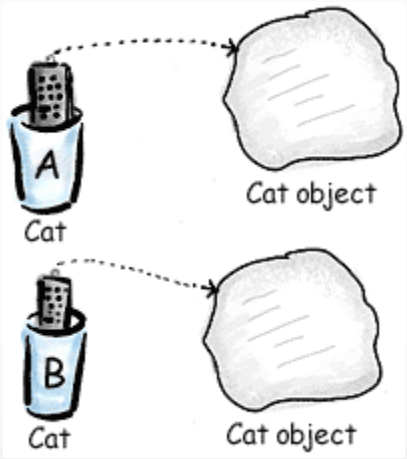


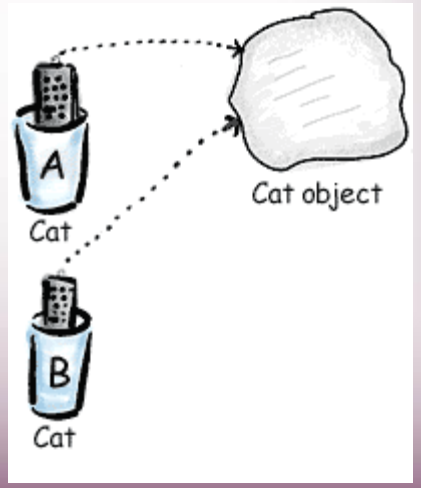
# Pass-by-Value

- 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);

      }
    }
```

# 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.

# 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));



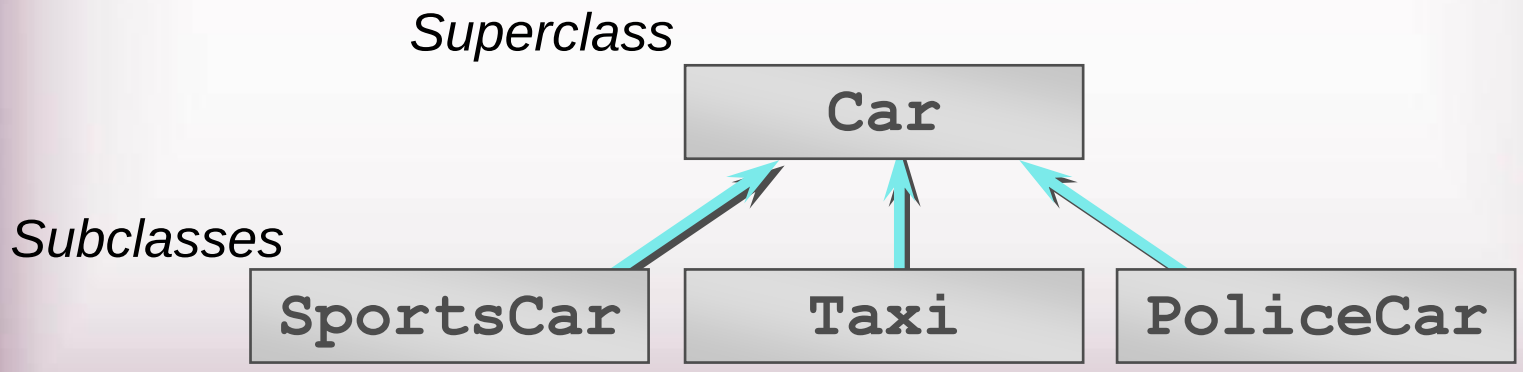
}
```

**Output will be 100 and Not 200**

# Overview

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

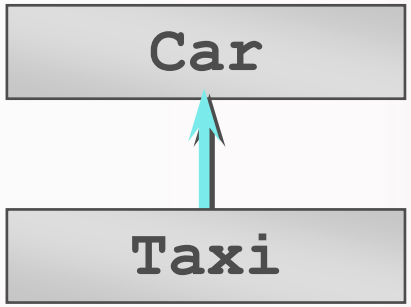
- Inheritance is a fundamental OO concept



# 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?

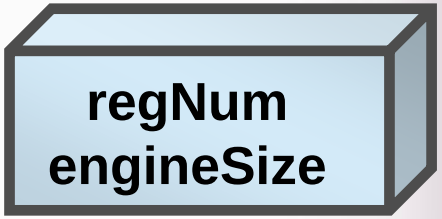
# What Does an Object Look Like?

A subclass inherits all the instance variables of its superclass

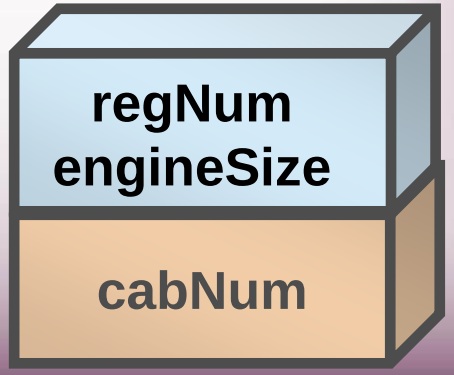
```
public class Car {
   String regNum;
    int engineSize; …
}
```

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

***Car* object**



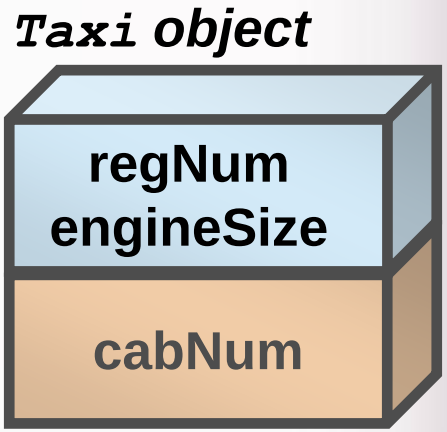

***Taxi* object**



# 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;
  }
```