



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 1 & 2

General Introduction

- Basics of Computer Programming
- Course Outlines
- Recommendations and references
- Sources
- Elements of Evaluation
 - Assignments/Class Exercises – 10
 - Weekly Quizzes - 10
 - Mid-term Exam - 40
 - Final Exam - 40

Introduction to course

- Why to program?
- Why Program in Object Oriented Language?
- Why to Program with JAVA?
- What are the objectives of Course?
- What is final outcome of Course?

Contents

- Procedural vs. Object Oriented Programming
- History of Programming Languages
- History of JAVA
- Buzzwords

I hear I forget
I see and I remember
I do and I understood

Chinese Proverb

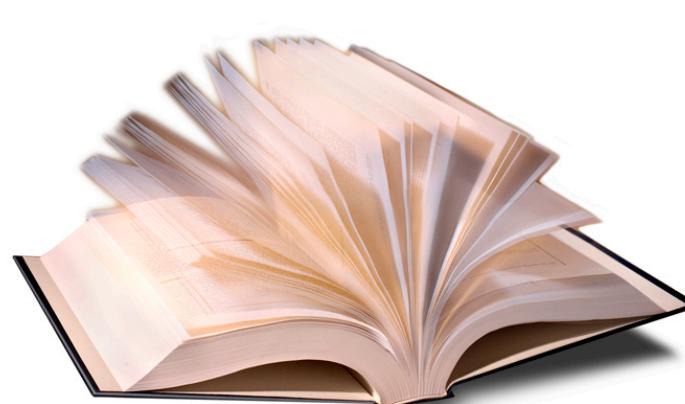
Procedural vs. OOP

- **Procedural Programming (PP):**
 - A series of conceptual steps
 - Step by step approach to achieve desired output
 - Focus: Break down to programming task into variables, methods, data structures.
 - A minor change in any part may become a bigger modification in whole the code
- **Object Oriented Programming (OOP)**
 - A collection of interacting objects
 - Objects act individually but used collectively to achieve desired output
 - Focus: Breakdown to programming task into Objects.
 - Change in any part would not affect other code

Why OOP? Examples



If we have
Thousands or
Millions of LOCs,
Then?



History of Programming Languages

- **FORTRON:** Good for Scientific Applications not good for system code
- **BASIC:** Easy, not v. powerful, lack of structure, not efficient with large programs
- **ASSEMBLY:** Efficient, not very easy, difficult to debug.
- **C:** Procedural
- **C++:** Very popular, Object Oriented

Why JAVA?

- Why Java if C++ is there?
 - Pointers
 - Destructors
 - Networking

History of JAVA

- 1990s' a funded team **Project Green** by sun worked on next age technology.
- Need of an application (Distributed, Heterogeneous and Huge)
- Oak as first name of Java
- In 1995 James Gosling
- Popularity of Internet in 1995 was the reason behind the popularity of Java
- Todays' Java:
- Enterprise applications, Mobile Devices, Mars probes

Buzzwords [1/3]

Simple

Secure

Portable

Object-oriented

Robust

Multithreaded

Architecture-neutral

Interpreted and High performance

Distributed

Dynamic

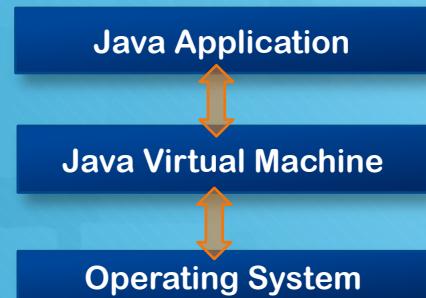
Buzzwords [2/3]

- **Simple:**
 - Easy for professional programmers
 - Java inherits C and C++ syntax and several features
- **Secure**
 - Applets in java doesn't allow the downloaded application to access other computer programs
 - But Active X controllers may harm your computer and security risk
- **Object Oriented:**
 - Based on OOP Concepts 99.9% are objects in Java (Except Primitive datatypes)
- **Robust:**
 - Reliable on variety of Systems
 - Strictly typed language
 - Memory Management
 - Exception Handling

Buzzwords [3/3]

- Portable:

- WORA
 - JVM



- Architecture Neutral

- Supports Different Operating Systems

- WORA

- Multithread

- Allows you to write a program that can perform many things simultaneously.

- Interpreted and High performance

- Java Bytecode

- Distributed

- Internet, with TCP/IP also RMI(Remote Method Invocation)

- Dynamic

- Dynamic updating, small fragments of bytecode maybe dynamically updated on running system.

Questions?

Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 3

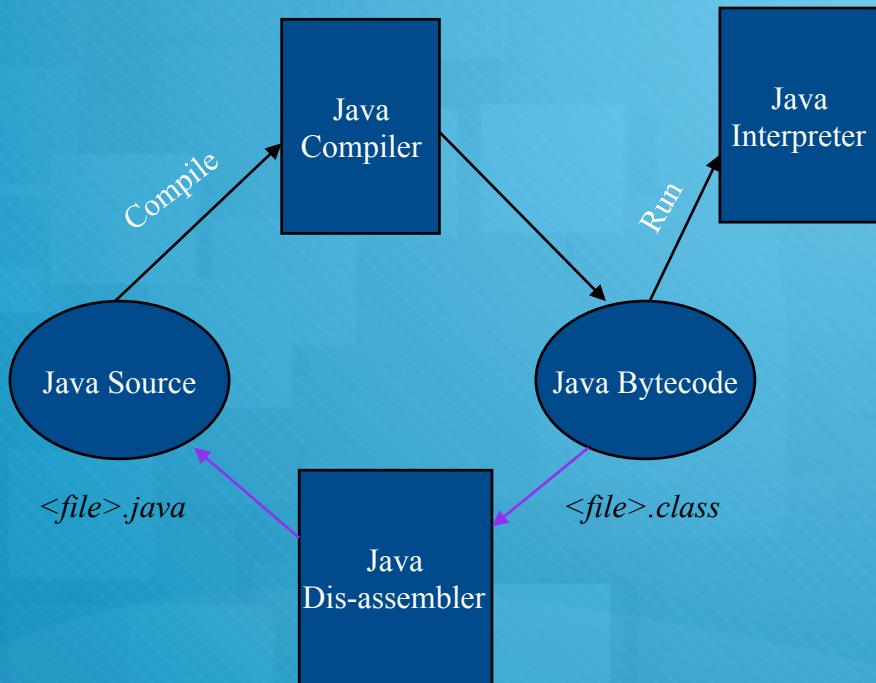
Review

JDK, JRE and JVM

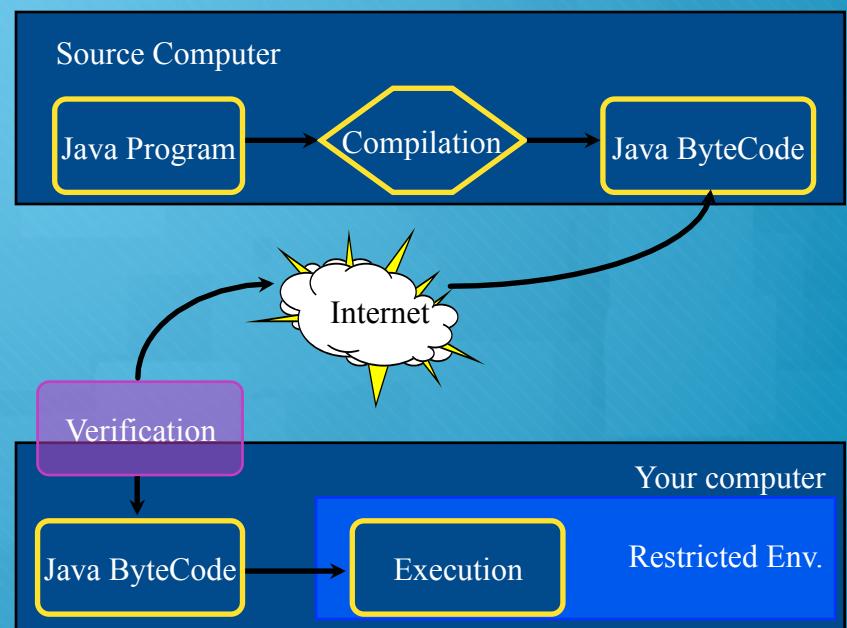
- JVM is used to convert bytecode into machine code, and JVM includes all libraries needed.
- JRE is needed only to run a program. JRE also includes JVM.
- JDK is used by developers and contains both JRE and JVM
- ***JDK contains the JRE which contains JVM.***

Java Development Kit

o Java Development Kit



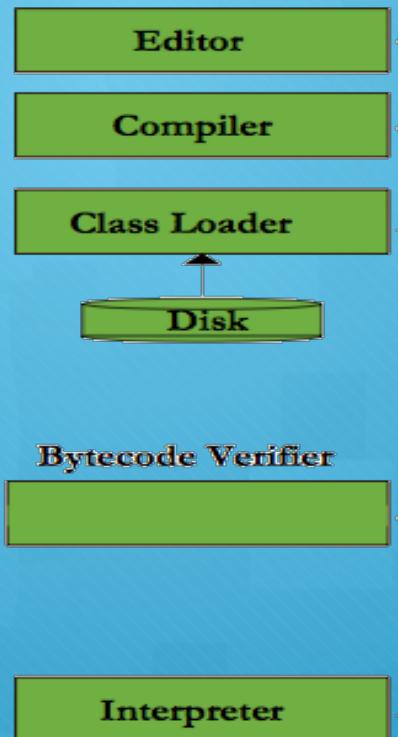
o Prepare and Execute



Development & Execution

PHASES

- Edit
- Compile
- Load
- Verify
- Execute



JDK Editions

- **Java Standard Edition (J2SE)**
 - J2SE can be used to develop client-side standalone applications or applets.
- **Java Enterprise Edition (J2EE)**
 - J2EE can be used to develop server-side applications such as Java servlets and Java ServerPages.
- **Java Micro Edition (J2ME)**
 - J2ME can be used to develop applications for mobile devices such as cell phones.

10/26/14

22

Programming Paradigm

- All computer programs are conceptually organized around *Code or Data*. **What is happening & Who is being affected**
- **Process Oriented Programming:** Series of linear steps (this is, Code). We can say that Code is acting on data
- **Object Oriented Programming:** Organize a Program around its data (that is, object); Data Controlling access to code

Abstraction

- Manage complexity through abstraction
 - A car is composed of 10,000 individual parts!!!
- Breaking into more manageable pieces
- OOP is powerful natural paradigm that once you have a well defined objects, you can gracefully replace or modify the existing parts.

Three OOP Principles

Encapsulation [1/2]

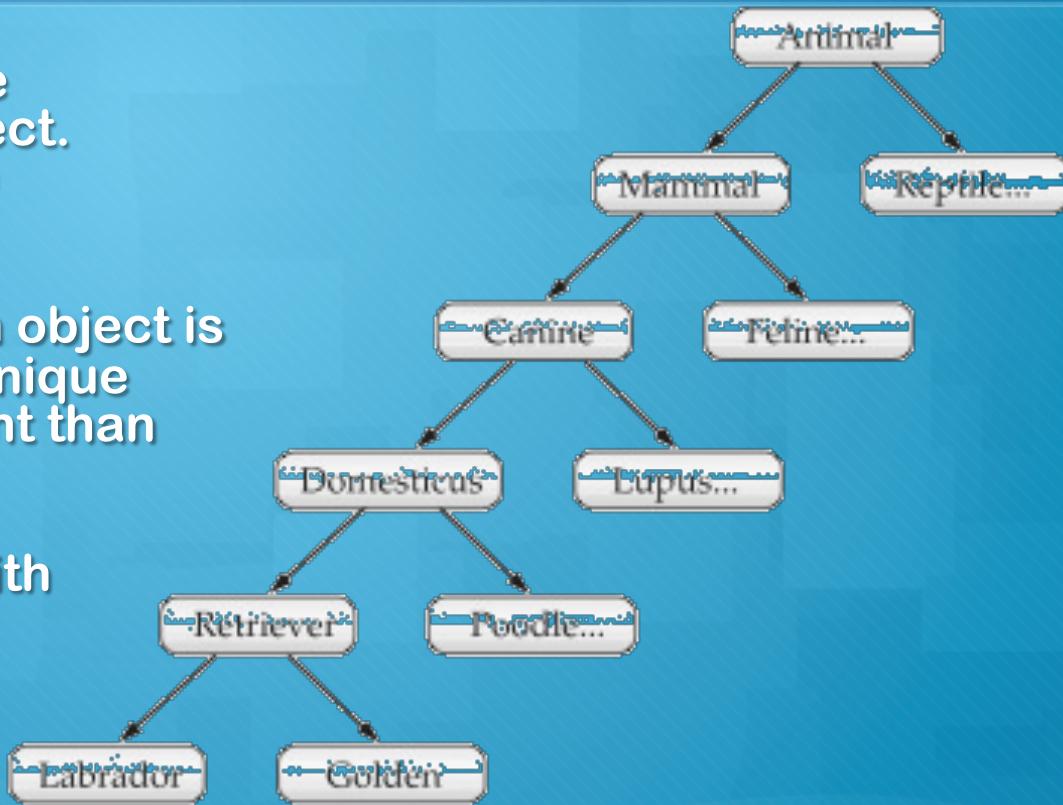
- Mechanism which binds Code & Data and keep them safe (Protective Wrapper)
- Tightly well defined interfaces can be used to access code & data
- Inside transmission doesn't affect outside transmission
- In java basis of encapsulation is class

Encapsulation [1/2]

- Class defines structures and behaviors (Data & Code) shared by the a set of object. So each object contains structures & Behaviors
- An object is also referred as instance of a class. (Class is logical, object is physical) `Rectangle rect = new Rectangle();`
- Collectively data and code are called as members of class.
 - Member variables or variables
 - Member Methods or Methods
- Public: Everything inside the class can be accessed by outside program
- Private: Everything inside the class can only be accessed by the members of that class

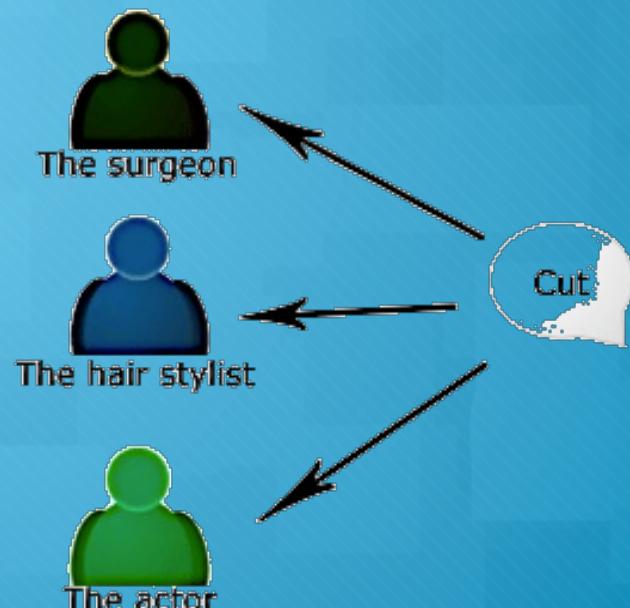
Inheritance

- One object acquires the properties of other object. Hierarchical (top down) classification
- By using Hierarchies an object is needed to define only unique characteristics (different than parents).
- Inheritance interacts with Encapsulation also



Polymorphism

- Poly (many) morph (forms)
- One interface is used for general class of actions, and the action would be determined by the nature of situation
- One interface multiple methods (A generic interface can be defined for group of related activities)
- DOG: Sense of Smell make it to bark if there's cat and same sense make it run to bowl of milk.



Encapsulation Inheritance & Polymorphism Together

- Every java program involves encapsulation, inheritance, and polymorphism.
- All drivers rely on inheritance to drive different types (subclasses) of vehicles.
- Encapsulation is that you are dealing with a car without imagining the complexities it have.
- Polymorphism one manufacture provides several kind of vehicles some have ABS breaks some have normal braking systems. Some have manual transaction some have automatic etc..

Week Review

- Introduction to course
- PP vs. OOP
- History of Languages
- Buzzwords (Simple, Secure
Portable, OO, robust, multithread,
architecture-neutral, interpreted
and High Performance, Distributed,
Dynamic)
- JDK (How it works, Execution steps,
Editions), JRE & JVM
- Programming Paradigms
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Questions?

Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 4 and 5

Recap

First Java application

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
    // Your program begins with a call to main().
    public static void main(String args[]) {
        System.out.println("This is a simple Java program.");
    }
}
```

A second Short Example

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2 {
    public static void main(String args[]) {
        int num; // this declares a variable called num
        num = 100; // this assigns num the value 100
        System.out.println("This is num: " + num);
        num = num * 2;
        System.out.print("The value of num * 2 is ");
        System.out.println(num);
    }
}
```

Two Control Statements

- Chapter 5 is dedicated for detailed discussion about controls

If statement:

```
if(condition) statement;
```

- Condition is a Boolean expression, if true then executes the statement and if false then skip the statement.

For loop:

```
for (initialization; condition; iteration) statement;
```

- Initializer sets starting of the loop
- Condition tests the control variables of loop
- Iteration gives increment or decrement to the loop.

Using Block Code

- 2 or more statements grouped together is called as *block of code* or *code block*

```
if(x < y)    { // begin a block  
    x = y;  
    y = 0;  
} // end of block
```

- Can be targeted for any kind of block; like *if* statement or *for* statement

Lexical Issues [1/5]

1) Whitespace 2) Identifier 3)Literals 4) Comments 5) Separators 6) Keywords

- **Whitespace:** Free-form language, no need to put extra indentations, as long as there's one white space between each token.
- **Identifiers:** Sometimes call it as conventions. Some common conventions to name variables, methods, classes, interfaces and packages.
 - Sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters
 - Java is a case-sensitive language

`MyVariable myVariable _myVariable`

`$myVariable _myVar1 my_variable`

`my-variable My Variable 2MyVariable`

Lexical issues [2/5]

- **Literals:**
 - It's a constant value in Java can be represented by using literals.
100 98.7 'X' "This is a test"
- **Comments:**

There are 3 type of comments in Java

 - Single Line comment
 - Multiple Line Comment
 - Document Comment

Lexical Issues [3/5]

Document Comments

- Document Comments allow you to create documentation for your program.
 - Start of Comments `/**`
 - End of comments `*/`
 - Every line have an `*` in start
 - Several Tags for documentation `@author` `@version` `@deprecated` `@param` `@return` ...

```
/**  
 * This document is Hello World Document  
 * As My first program with OOP Course  
 * @author Student Name  
 * @version 1.0  
 */
```

Lexical Issues [4/5]

Java Separators

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a <code>for</code> statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Lexical Issues [5/5]

Java Keywords

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

Applets

- Client side web base application (Browser)
- Applets run on your browser (A way to run java program through your browser)
- Very small internet or intranet application, and very secure
- Life cycle : Init→start→paint→stop→destroy
- Object→Component→Container→Panel→Applet

Applet Application

```
import java.awt.Graphics;
public class HelloApplet extends
java.applet.Applet
{
    public void init() {
        resize(150,25);
    }
    public void paint(Graphics g) {
        g.drawString("Hello
world!", 50, 25);
    }
}
```

Applet Html

```
<html>
<head>
<title>This is an applet</title>
</head>
<body>
<h1>This is an applet</h1>
<Applet CODE="HelloApplet.class" width="200" height="200">
</body>
</html>
```

Embedding Applet

- The Applet tag in html is used to embed an applet in the web page.
- CODE attribute of Applet tag is used to specify the name of Java applet class name.
- To test your applet open the html file in web browser

Java Documentation

- <http://docs.oracle.com/javase/1.5.0/docs/index.html>

Chapter 3.

Data Types, Variables, and Arrays

Fundamental Elements of Java

- There are 3 fundamental elements of JAVA:
 - Data Types
 - Variables
 - Arrays

Primitive Types

- 8 Primitive types of data, divided in four different categories.
 - Integer: byte, short, int and long
 - Floating-Point: float, double
 - Character: char
 - Boolean: boolean

Integers

- o Java defines four integer types: byte, short, int, and long.

Type	Size (bits)
o byte	-
o short	-
o int	-
o long	-

- o All are signed.

Type	Storage Requirement	Range (Inclusive)
int	4 bytes	-2,147,483,648 to 2,147,483,647 (just over 2 billion)
short	2 bytes	-32,768 to 32,767
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	1 byte	-128 to 127

Example - 1

```
// Compute distance light travels using long variables.  
class Light {  
    public static void main(String args[]) {  
        int lightspeed;    long days;    long seconds;    long  
distance;  
        // approximate speed of light in miles per second  
        lightspeed = 186000;  
        days = 1000; // specify number of days here  
        seconds = days * 24 * 60 * 60; // convert to seconds  
        distance = lightspeed * seconds; // compute distance  
        System.out.print("In " + days);  
        System.out.print(" days light will travel about ");  
        System.out.println(distance + " miles.");  
    }  
}
```

Floating-Point

- o 2 floating point or real number data types: float and double

Type	size	precision
o float	- 64	- 8
o double	- 128	- 16

Type	Storage Requirement	Range
float	4 bytes	approximately $\pm 3.40282347E+38F$ (6-7 significant decimal digits)
double	8 bytes	approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)

Example - 2

```
// Compute the area of a circle.  
class Area {  
    public static void main(String args[]) {  
        double pi, r, a;  
        r = 10.8;                      // radius of circle  
        pi = 3.1416;                   // pi,  
        approximately  
        a = pi * r * r;              // compute area  
        System.out.println("Area of circle is " + a);  
    }  
}
```

Character

- It includes only one type: char

Type	Size	Value
char	16	'A'

- Java is able to use **18** international languages
- UNICODE character set
- ASCII character set

Example - 3

```
// Demonstrate char data type.  
class CharDemo {  
    public static void main(String args[]) {  
        char ch1, ch2;  
        ch1 = 88; // code for X  
        ch2 = 'Y';  
        System.out.print("ch1 and ch2: ");  
        System.out.println(ch1 + " " + ch2);  
    }  
}
```

Scope and lifetime of Variables

- Java allows to use variables within any block
- A block defines the scope, each time new block is a new scope
- Declaring variable within the scope is preventing variable from localizing.
- Scope rules provide for encapsulation
- Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope.

Example - 4

```
// Demonstrate block scope.  
class Scope {  
    public static void main(String args[]) {  
        int x; // known to all code within main  
        x = 10;  
        if(x == 10) { // start new scope  
            int y = 20; // known only to this block  
            // x and y both known here.  
            System.out.println("x and y:  
" + x + " " + y);  
            x = y * 2;  
        }  
        // y = 100; // Error! y not known here  
        // x is still known here.  
        System.out.println("x is " + x);  
    }  
}
```

Type Conversion and Casting [1/2]

- It is fairly common to assign a value of one type to a variable of another type.
- If both types are compatible (Size) then java perform conversion automatically (int to long)
- But conversion from incompatible requires casting for eg. Double to byte (**Explicit Conversion**)
- **Automatic Conversion:** *widening conversion* takes place if
 - The two types are compatible
 - Destination type is large enough to hold source type

Type Conversion and Casting [2/2]

- **Casting Incompatible:** what if you want to assign an int value to a byte variable? This kind of conversion is sometimes called a *narrowing conversion*
(target-type) value

```
int a;  
  
byte b;  
  
// ...  
  
b = (byte) a;
```

- **Truncation:** While assinging floating point values to integer type. Thus if the value 1.23 is assigned to an integer, the resulting value will simply be 1.

Automatic Type Promotion in Expressions

- When conversion occurs in expressions

```
byte a = 40;  
byte b = 50;  
byte c = 100;  
int d = a * b / c;
```

- Result of $a * b$ easily exceeds the range of either of its byte operands. But java performs automatic promotion to each byte short and char to int.

```
byte b = 50;  
b = b * 2; // Error! Cannot assign an int to a byte!
```

- Result is integer value, so need casting

```
byte b = 50;  
b = (byte)(b * 2);
```

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 6

Recap

- First and Second Java Applications
- Introduced two control statements
- Discussed lexical issues
- Applets and an example of applets
- Using Documentations
- Started Chapter 3 where studied 4 categories of Data types, their scope and lifetime, and type conversion.

Arrays

- Group of like-typed variables that are referred to by a common name
- May have one or more dimensions
- Element in an array accessed by index number
- ONE DIMENSIONAL
- MULTI DIMENSIONAL

One dimensional Arrays

[1/4]

- A list of like-typed variables. The syntax for One dimensional Array is:

type var-name[];

`int month_days[];`

- Array here really not exist actually. `month_days` set to *null*. To allocate a memory location you must use *new* keyword

`array-var = new type[size];`

`month_days = new int[12];`

- Array declaration is 2 step process 1) declare variable of desired type 2) assign physical existence using new keyword

One dimensional Arrays

[2/4]

- In Java all arrays are dynamically allocated
- To access an element of array by specifying the index number of that element.
- All array indexes start at zero

```
month_days[1] = 28;
```

OneD Arrays - Example.

[3/4]

```
// Demonstrate a one-dimensional array.  
class Array {  
    public static void main(String args[]) {  
        int month_days[]; month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        //***FILL IN HERE WITH REMAINING MONTHS DAYS***  
        month_days[10] = 30;  
        month_days[11] = 31;  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```

One dimensional Arrays

[4/4]

- Alternative declaration:

```
int month_days[] = new int[12];
```

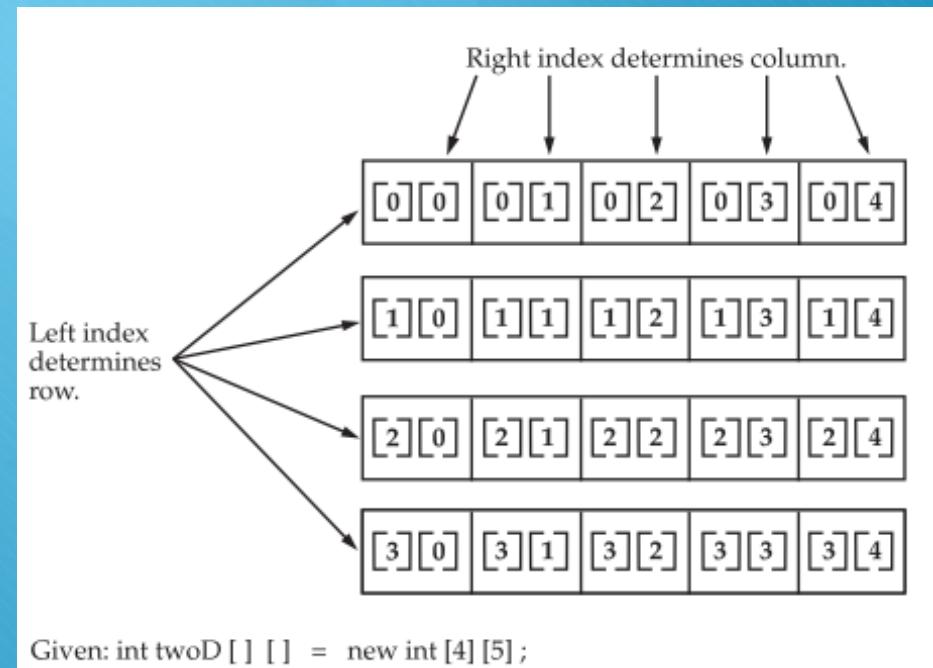
- Alternative declaration & initializations:

```
class AutoArray {  
    public static void main(String args[])  
    {  
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```

Multidimensional Arrays

- This is Arrays of Arrays
- Act like regular multidimensional arrays.

int twoD[][] = new int[4][5];



Multidimensional Arrays

Example

```
public class TwoDArray {  
    public static void main(String args[])  
    {  
        int twoD[][]= new int[4][5];  
        int i, j, k = 0;  
        for(i=0; i<4; i++)  
            for(j=0; j<5; j++) {  
                twoD[i][j] = k;    k++;  
            }  
        for(i=0; i<4; i++) {  
            for(j=0; j<5; j++)  
                System.out.print(twoD[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

Output:

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

Uneven Multidimensional Arrays

- The length of each array is under your control

```
int twoD[][] = new int[4][];
```

```
twoD[0] = new int[1];
```

```
twoD[1] = new int[2];
```

```
twoD[2] = new int[3];
```

```
twoD[3] = new int[4];
```

```
class UnEven{  
    public static void main(String args[]){  
        int twoD[][] = new int[4][];  
        twoD[0] = new int[1];  
        twoD[1] = new int[2];  
        twoD[2] = new int[3];  
        twoD[3] = new int[4];  
        int i, j, k = 0;  
        for(i=0; i<4; i++){  
            for(j=0; j<twoD[i].length; j++){  
                System.out.print(twoD[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

Output:

0	1	2	
3	4	5	
6	7	8	9

Alternative Syntax Array declaration

Syntax: *type[] var-name;*

```
int a1[] = new int[3];
```

OR

```
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
```

```
char[][] twod2 = new char[3][4];
```

Multiple Arrays Declaration

```
int[] nums, nums2, nums3; // create three arrays
```

```
int nums[], nums2[], nums3[]; // create three arrays
```



Strings

- Java string is a sequence of characters.
They are objects of type String.

A String in C is a nul-terminated sequence of chars
The standard C library contains several string functions

S1

T	h	i	s		i	s		a		s	t	r	i	n	g		i	n		c	\0	
---	---	---	---	--	---	---	--	---	--	---	---	---	---	---	---	--	---	---	--	---	----	--

S2

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

A String in Java is an object with several methods

S1

charAt()
compareTo()
length()
...

S2

charAt()
compareTo()
length()
...

Questions?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 7 and 8 – Part 1

Chapter 4 - Operators

Operators

- Arithmetic
- Bitwise
- Relational
- Logical

Arithmetic Operators

Operator	Result
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

Example - 1

```
// Demonstrate the basic arithmetic operators.  
class BasicMath {  
    public static void main(String args[]) { // arithmetic using integers  
        System.out.println("Integer Arithmetic");  
        int a = 1 + 1;    int b = a * 3;    int c = b / 4;    int d = c - a;    int e = -d;  
  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
        System.out.println("e = " + e);  
        // arithmetic using doubles  
        System.out.println("\nFloating Point Arithmetic");  
        double da = 1 + 1; double db = da * 3; double dc = db / 4;  
        double dd = dc - a; double de = -dd;  
        System.out.println("da = " + da);  
        System.out.println("dc = " + dc);  
        System.out.println("de = " + de);  
        System.out.println("db = " + db);  
        System.out.println("dd = " + dd);  
    }  
}
```

Arithmetic Operators

- Modulus operator:
 - `%`, returns the remainder of a division operation
 $X=40.2; X \% 10;$
- Arithmetic Compound Assignment Operators
 - combine an arithmetic operation with an assignment
 $var = var \ op \ expression;$
 $a = a + 4;$
 $a += 4;$
- Increment and Decrement
 $x = x + 1;$ is same as $x++;$ Or $x = x - 1;$ is same as $x--;$

$x = 42;$

$y = ++x;$

Same as

$x = x + 1;$

$y = x;$

However

$x = 42;$

$y = x++;$

Same as:

$y = x;$

$x = x + 1;$

Example - 2

```
// Demonstrate ++.  
class IncDec {  
    public static void main(String args[]) {  
        int a = 1; int b = 2; int c; int d;  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
    }  
}
```

OUTPUT

a= 2
b= 3
c= 4
d= 1

Bitwise Operators

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte.

Operator	Result
<code>~</code>	Bitwise unary NOT
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise exclusive OR
<code>>></code>	Shift right
<code>>>></code>	Shift right zero fill
<code><<</code>	Shift left
<code>&=</code>	Bitwise AND assignment
<code> =</code>	Bitwise OR assignment
<code>^=</code>	Bitwise exclusive OR assignment
<code>>>=</code>	Shift right assignment
<code>>>>=</code>	Shift right zero fill assignment
<code><<=</code>	Shift left assignment

Logical Operations

- Bitwise logical operators are $\&$, $|$, $^$, and \sim .

A	B	A B	A & B	A ^ B	$\sim A$
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```

// Demonstrate the bitwise logical operators.
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
            "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0011 in binary
        int b = 6; // 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;

        System.out.println("    a = " + binary[a]);
        System.out.println("    b = " + binary[b]);
        System.out.println("    a|b = " + binary[c]);
        System.out.println("    a&b = " + binary[d]);
        System.out.println("    a^b = " + binary[e]);
        System.out.println("    ~a&b|a&~b = " + binary[f]);
        System.out.println("    ~a = " + binary[g]);
    }
}

```

OUTPUT

a = 3

b = 6

a|b = 7

a&b = 2

a^b = 5

~a&b|a&~b = 5

~a = 12

OUTPUT

a = 0011

b = 0110

a|b = 0111

a&b = 0010

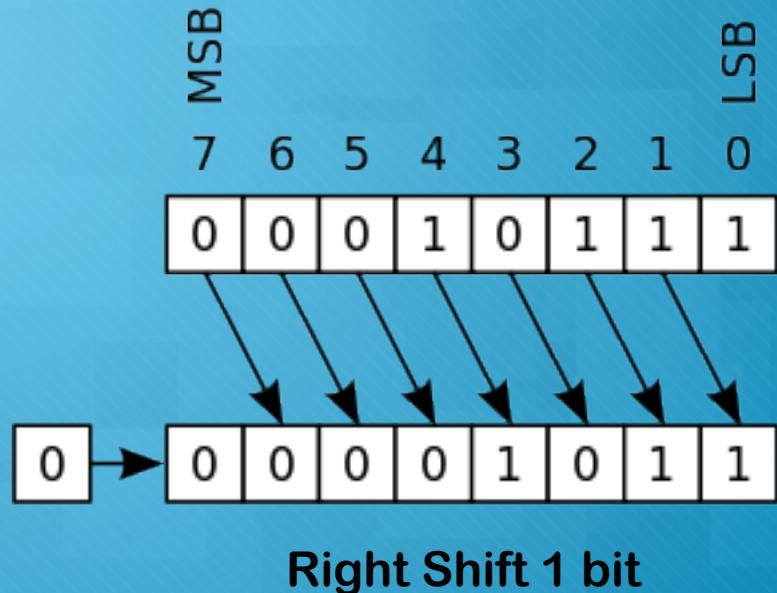
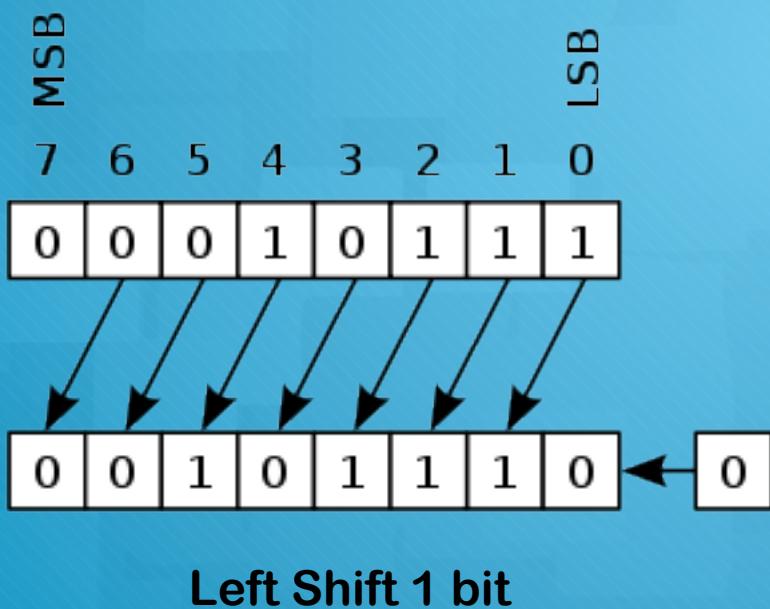
a^b = 0101

~a&b|a&~b = 0101

~a = 1100

SHIFT: Left

and >> Right



Shift << Left

```
// Left shifting a byte value.  
class ByteShift {  
    public static void main(String args[]) {  
        byte a = 64, b;  
        int i;  
  
        i = a << 2;  
        b = (byte) (a << 2);  
  
        System.out.println("Original value of a: " + a);  
        System.out.println("i and b: " + i + " " + b);  
    }  
}
```

OUTPUT

Original value of a: 64
i and b: 256 0

Shift >> Right

```
int a = 35;  
a = a >> 2; // a still contains 8
```

- Looking at the same operation in binary shows more clearly how this happens:
- **00100011 35 >> 2**
- **00001000 8**

Shift Right Fill Zero >>>

```
int a = -1;  
a = a >>> 24;
```

- Here is the same operation in binary form to further illustrate what is happening:
- 11111111 11111111 11111111 11111111 –1 in binary as an int
- >>>24
- 00000000 00000000 00000000 11111111 255 in binary as an int

Relational Operators

- The relational operators determine the relationship that one operand has to the other.

Operator	Result
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

Valid & Not Valid in JAVA

```
int done;  
// ...  
if(!done) ... // Valid in C/C++  
if(done) ...      // but not in Java.
```

- In Java, these statements must be written like this:

```
if(done == 0) ... // This is Java-style.  
if(done != 0) ...
```

Boolean Logical Operators

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Boolean Logical Operators

- The logical Boolean operators, `&`, `|`, and `^`, operate on boolean values in the same way that they operate on the bits of an integer.

A	B	<code>A B</code>	<code>A & B</code>	<code>A ^ B</code>	<code>!A</code>
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Example

```
// Demonstrate the boolean logical operators.  
class BoolLogic {  
    public static void main(String args[]) {  
        boolean a = true;  
        boolean b = false;  
        boolean c = a | b;  
        boolean d = a & b;  
        boolean e = a ^ b;  
        boolean f = (!a & b) | (a & !b);  
        boolean g = !a;  
  
        System.out.println("    a = " + a);  
        System.out.println("    b = " + b);  
        System.out.println("    a|b = " + c);  
        System.out.println("    a&b = " + d);  
        System.out.println("    a^b = " + e);  
        System.out.println("!a&b|a&!b = " + f);  
        System.out.println("    !a = " + g);  
    }  
}
```

OUTPUT

a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false

Short circuit (&&)

- Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

```
if (denom != 0 && num / denom > 10)
```

- There is no risk of causing a run-time exception when denom is zero.

The ? Operator

- ternary (three-way) operator that can replace certain types of **if-then-else** statements.

Expression1 ? Expression2 : Expression3

- Expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated

Example

```
// Demonstrate ?.
class Ternary {
    public static void main(String args[]) {
        int i, k;

        i = 10;
        k = i < 0 ? -i : i; // get absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k);

        i = -10;
        k = i < 0 ? -i : i; // get absolute value of i
        System.out.print("Absolute value of ");
        System.out.println(i + " is " + k);
    }
}
```

OUTPUT

Absolute value of 10 is 10
Absolute value of -10 is 10

Chapter Review

- Arithmetic
- Bitwise
- Relational
- Logical

Questions?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 10 and 11

Chapter – 5

Control Statements

Three Categories

- Java's control statements can be put into 3 categories:
 - Selection
 - Iteration
 - Jump

Selection Statements

- Control the flow of the program only during run time.
 - if
 - switch

If Statement

- Discussed in chapter 2, but here the power is discussed in detail:

```
if (condition) statement1;  
else statement2;
```

Each statement may be a single statement or a compound statement (Block)

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

In no case will both statements be executed.

```
int a, b;  
// ...  
if(a < b) a = 0;  
else b = 0;
```

If with Relational Op. & Boolean variables

- Most often, the expression used to control the if will involve the relational operators. However, this is not technically necessary. It is possible to control the if using a single boolean variable, as shown in this code fragment:

```
boolean dataAvailable;  
// ...  
if (dataAvailable)  
    ProcessData();  
else  
    waitForMoreData();
```

```
int bytesAvailable;  
// ...  
if (bytesAvailable > 0) {  
    ProcessData();  
    bytesAvailable -= n;  
} else  
    waitForMoreData();
```

Nested ifs

- A nested if is an if statement that is the target of another if or else.

```
if(i == 10) {  
    if(j < 20)      a = b;  
    if(k > 100)    c = d; // this if is  
    else a = c;    // associated with this else  
}  
else a = d;          // this else refers to if(i == 10)
```

Ladder of if-else

```
if(condition)  
    statement;  
else if(condition)  
    statement;  
else if(condition)  
    statement;  
.  
.  
. .  
else  
    statement;
```

Example

```
// Demonstrate if-else-if statements.  
class IfElse {  
    public static void main(String args[]) {  
        int month = 4; // April  
        String season;  
  
        if(month == 12 || month == 1 || month == 2)  
            season = "Winter";  
        else if(month == 3 || month == 4 || month == 5)  
            season = "Spring";  
        else if(month == 6 || month == 7 || month == 8)  
            season = "Summer";  
        else if(month == 9 || month == 10 || month == 11)  
            season = "Autumn";  
        else  
            season = "Bogus Month";  
        System.out.println("April is in the " + season + ".");  
    }  
}
```

OUTPUT

April is in the Spring.

switch

- Multiway branch statement.
- A better alternative than a large series of if-else-if statements.

```
switch (expression) {
case value1:
    // statement sequence
    break;
case value2:
    // statement sequence
    break;
.
.
.
case valueN:
    // statement sequence
    break;
default:
    // default statement sequence
}
```

// An improved version of the season program.

```
1 class Switch {
```

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

```
1         int month = 4;
```

```
1         String season;
```

```
1         switch (month) {
```

```
1             case 12:
```

```
1             case 1:
```

```
1             case 2:
```

```
1                 season = "Winter";
```

```
1                 break;
```

```
1             case 3:
```

```
1             case 4:
```

```
1             case 5:
```

```
1                 season = "Spring";
```

```
1                 break;
```

```
1             case 6:
```

```
1             case 7:
```

```
1             case 8:
```

```
1                 season = "Summer";
```

```
1                 break;
```

```
1             case 9:
```

```
1             case 10:
```

```
1             case 11:
```

```
1                 season = "Autumn";
```

```
1                 break;
```

```
1             default:
```

```
1                 season = "Bogus Month";
```

```
}
```

```
1         System.out.println("April is in the " + season + ".");
```

```
}
```

Nested switch Statements

```
switch(count) {  
    case 1:  
        switch(target) { // nested switch  
            case 0:  
                System.out.println("target is zero");  
                break;  
            case 1: // no conflicts with outer switch  
                System.out.println("target is  
one");  
                break;  
        }  
        break;  
    case 2: //
```

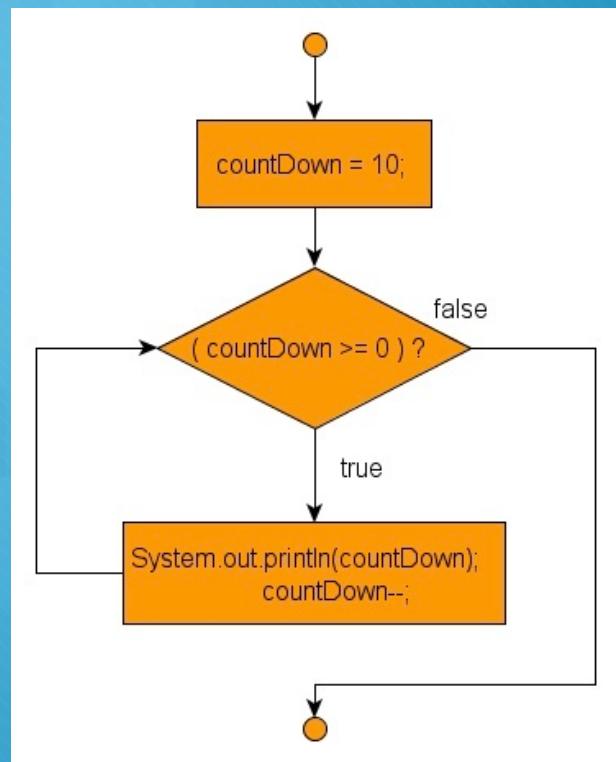
Iteration Statements

- Java's iteration statements are:
 - For
 - While
 - Do-While

While (1/2)

- The while loop is Java's most fundamental loop statement
- It repeats a statement or block while its controlling expression is true.
- The condition can be any Boolean expression.

```
while(condition) {  
    // body of loop  
}
```



While (2/2)

```
// Demonstrate the while loop.  
class While {  
    public static void main(String args[]) {  
        int n = 10;  
  
        while(n > 0) {  
            System.out.println("tick " + n);  
            n--;  
        }  
    }  
}
```

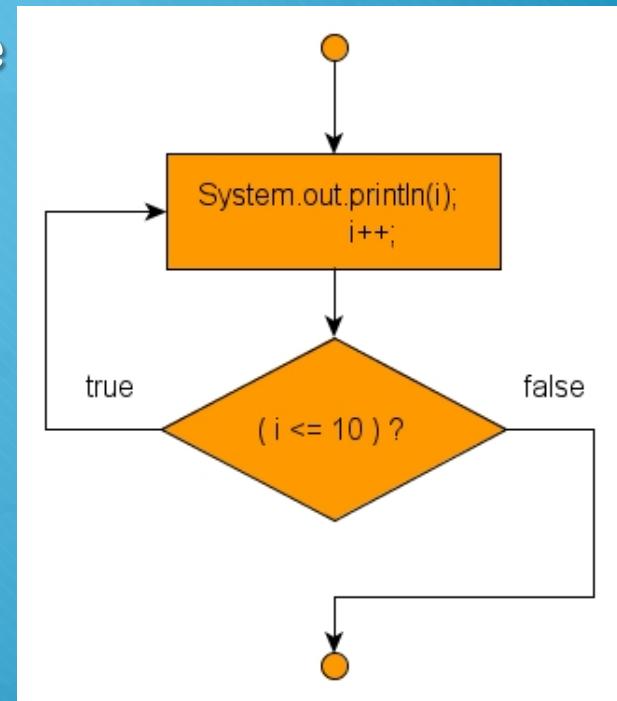
```
// The target of a loop can be empty.  
class NoBody {  
    public static void main(String args[]) {  
        int i, j;  
  
        i = 100;  
        j = 200;  
  
        // find midpoint between i and j  
        while(++i < --j) ; // no body in this loop  
  
        System.out.println("_____ is " + i);  
    }  
}
```

OUTPUT
Midpoint is 150

do-while [1/2]

- Sometimes it is desirable to execute the body of a loop at least once
- This can be achieved because its conditional expression is at the bottom of the loop.

```
do {  
    // body of loop  
} while (condition);
```



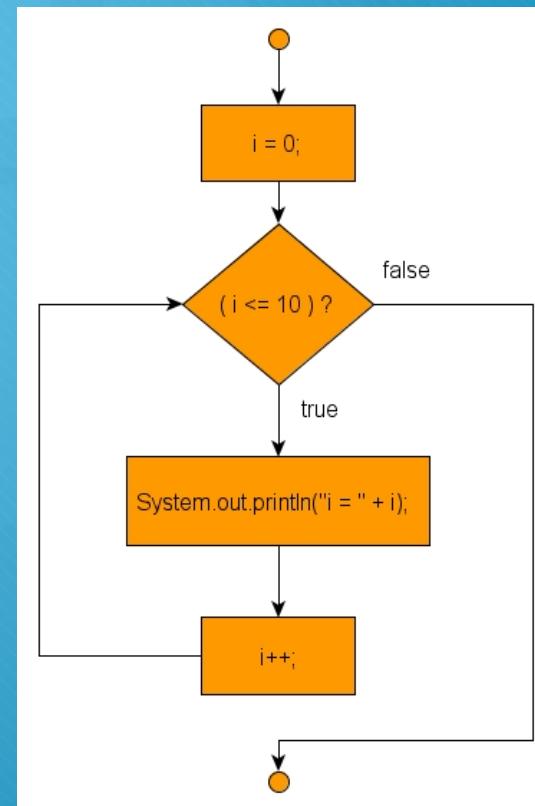
do-while [2/2]

```
class doWhile {  
    public static void main(String[] args){  
        int count = 11;  
        do {  
            System.out.println("Count is: " + count);  
            count++;  
        } while (count < 11);  
    }  
}
```

for

- There are two forms of for loop in java:
 - Traditional for loop
 - for-each

```
for(initialization; condition; iteration) {  
    // body  
}
```



for

- o Declaration of controls is possible inside the loop

for(int n=10; n>0; n--)

- o multiple statements in both the initialization and iteration portions of the for.

for(a=1, b=4; a<b; a++, b--)

for – Some variations

```
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done =
true;
}
```

```
for( ; ; ) {
    // ... Body of loop
}
```

```
// Parts of the for loop can be empty.
class Practice {
    public static void main(String args[]) {
        int i;
        boolean done = false;

        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Question?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 09

Recap

for-each

- designed to cycle through a collection of objects, it is useful for Arrays mostly (in strictly sequential fashion)
- Added after jdk1.5

for(type itr-var : collection) statement-block

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

for-each

```
// Use a for-each style for loop.  
class Practice {  
    public static void main(String args[]) {  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        int sum = 0;  
  
        // use for-each style for to display and sum the values  
        for(int x : nums) {  
            System.out.println("Value is: " + x);  
            sum += x;  
        }  
  
        System.out.println("Summation: " + sum);  
    }  
}
```

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55

for-each with break

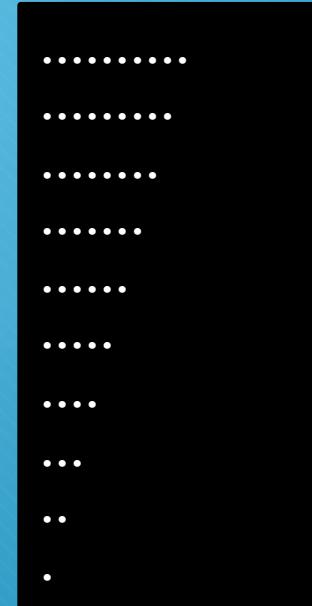
```
// Use break with a for-each style for.  
class ForEach2 {  
    public static void main(String args[]) {  
        int sum = 0;  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
        // use for to display and sum the values  
        for(int x : nums) {  
            System.out.println("Value is: " + x);  
            sum += x;  
            if(x == 5) break; // stop the loop when 5 is obtained  
        }  
        System.out.println("Summation of first 5 elements: " + sum);  
    }  
}
```

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15

Nested Loops

- Like all other programming languages, Java allows loops to be nested.

```
// Loops may be nested.  
class Nested {  
    public static void main(String args[]) {  
        int i, j;  
  
        for(i=0; i<10; i++) {  
            for(j=i; j<10; j++)  
                System.out.print(".");  
            System.out.println();  
        }  
    }  
}
```



Output
Required

Jump Statements [1/4]

- Using break
- Using break to Exit a Loop

```
// Using break to exit a loop.  
class BreakLoop {  
    public static void main(String args[]) {  
        for(int i=0; i<100; i++) {  
            if(i == 10) break; // terminate loop if i is 10  
            System.out.println("i: " + i);  
        }  
        System.out.println("Loop complete.");  
    }  
}
```

OUTPUT

```
i: 0  
i: 1  
i: 2  
i: 3  
i: 4  
i: 5  
i: 6  
i: 7  
i: 8  
i: 9 Loop complete.
```

Jump Statements [2/4]

- Using break as a Form of Goto
- break label;

OUTPUT

Before the break.

This is after second block.

```
// Using break as a civilized form of goto.  
class Break {  
    public static void main(String args[]) {  
        boolean t = true;  
  
        first: {  
            second: {  
                third: {  
                    System.out.println("Before the break.");  
                    if(t) break second; // break out of second block  
                    System.out.println("This won't execute");  
                }  
                System.out.println("This won't execute");  
            }  
            System.out.println("This is after second block.");  
        }  
    }  
}
```

Jump Statements [3/4]

o Using continue

listing 27

```
// Demonstrate continue.  
class Continue {  
    public static void main(String args[]) {  
        for(int i=0; i<10; i++) {  
            System.out.print(i + " ");  
            if (i%2 == 0) continue;  
            System.out.println("");  
        }  
    }  
}
```

OUTPUT

```
0 1  
2 3  
4 5  
6 7  
8 9
```

Jump Statements [4/4]

- The **return** statement is used to explicitly return from a method.

OUTPUT
Before the return.

```
// Demonstrate return.  
class Return {  
    public static void main(String args[]) {  
        boolean t = true;  
  
        System.out.println("Before the return.");  
  
        if(t) return; // return to caller  
  
        System.out.println("This won't execute.");  
    }  
}
```

Math Methods

- Absolute **Math.abs(arg)**
 - Round **Math.round(arg)**
 - Ceiling **Math.ceil(arg)**
 - Floor **Math.floor(arg)**
 - Minimum **Math.min(arg1,arg2)**
 - Maximum **Math.max(arg1,arg2)**

Math Methods

- | | |
|---------------|-------------------|
| ○ Value of PI | Math.PI |
| ○ Cos/Sin | Math.sin(arg) |
| ○ Power | Math.pow(arg,arg) |
| ○ Square root | Math.sqrt(arg) |
| ○ Random | Math.random() |

Week Review

- Arithmetic, Bitwise, Relational, Logical & Operators
- Three Control Statement
 - Selection (IF, SWITCH)
 - Iteration (while, do-while, for and for-each)
 - Jump (break, continue, return)

Questions?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 10 and 11

Introducing Classes

- Class:
 - Class is a blueprint/definition of user defined datatype
 - Think of it as a map of the building on a paper
- Object:
 - Anything we can put a thumb on
 - Objects are instantiated or created from class

a class is a template for an object, and an object is an instance of a class

Class

- Class contains the code that operates on data
- May contain code or data, but most real world classes contains both
- Data/ variables are: instance variables
- Code within class: Method (not defined as public or static)
- No main method in all, only one in main class

Both are members of class

Applet don't have main method at all

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

Simple class [1/2]

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

```
Box mybox = new Box();
```

```
mybox.width = 100;
```

OUTPUT

Volume is 3000.0

```
// This class declares an object of type Box.  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        double vol;  
  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
  
        // compute volume of box  
        vol = mybox.width * mybox.height * mybox.depth;  
  
        System.out.println("Volume is " + vol);  
    }  
}
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

- Objects have its own copy of instance variables.

OUTPUT

Volume is 3000.0
Volume is 162.0

```
class BoxDemo2 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
  
        /* assign different values to mybox2's  
           instance variables */  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
  
        // compute volume of first box  
        vol = mybox1.width * mybox1.height * mybox1.depth;  
        System.out.println("Volume is " + vol);  
  
        // compute volume of second box  
        vol = mybox2.width * mybox2.height * mybox2.depth;  
        System.out.println("Volume is " + vol);  
    }  
}
```

Declaring Objects

- when you create a class, you are creating a new data type.
You can use this type to declare objects of that type.
- obtaining objects of a class:
 - declare a variable of the class type
 - Acquire physical value of object and assign to variable (**new operator**),
 - The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

```
Box mybox; // declare reference to object  
mybox = new Box(); // allocate a Box object
```

```
Box mybox = new Box();
```

new keyword

class-var = new classname();

- Class-var: variable of class type being created
- Classname: class that is being instantiated
- new allocates memory for an object during run time
- *In previous example there was no constructor, so java defines a default constructor*

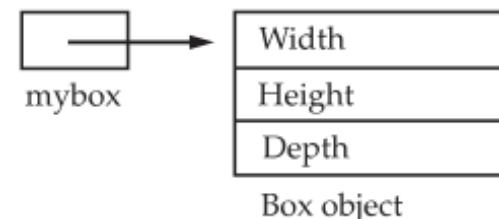
Statement

Box mybox;

Effect

null
mybox

mybox = new Box();



- A class creates a new data type that can be used to create objects
- When you declare an object of a class, you are creating an instance of that class.

Assigning Object Reference Variables

- Object variable act differently than you might thinking

```
Box b1 = new Box();
```

```
Box b2 = b1;
```

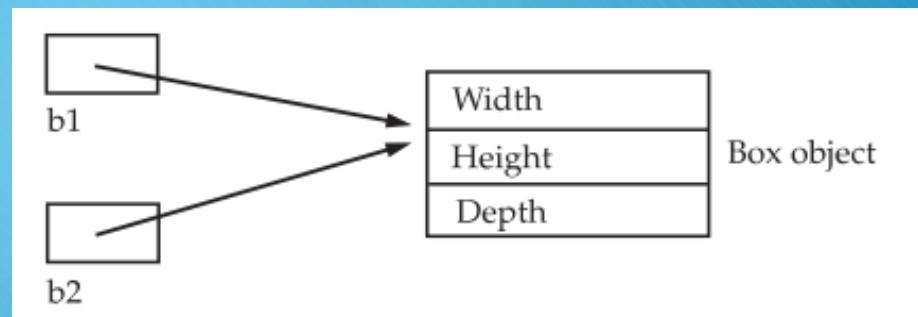
- How about this:

```
Box b1 = new Box();
```

```
Box b2 = b1;
```

```
// ...
```

```
b1 = null;
```



you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

- Classes usually consist of two things: instance variables and methods.
 - Instance Variables
 - Instance Methods
- *Both are members of class*
- Type: is type of data returned (any valid type)
- If the method does not return a value, its return type must be void.
- Name: any identifier but not used as keyword by java
- Parameter-list: Receive the value of the arguments passed to the method when it is called

```
type name(parameter-list) {  
    // body of method  
}
```

```
return value;
```

Adding a Method to the Box Class

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

OUTPUT

Volume is 3000.0
Volume is 162.0

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
  
        /* assign different values to mybox2's  
           instance variables */  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
  
        // display volume of first box  
        mybox1.volume();  
  
        // display volume of second box  
        mybox2.volume();  
    }  
}
```

Returning a Value

151

```
// Now, volume() returns the volume of a box.
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

OUTPUT

```
Volume is 3000.0  
Volume is 162.0
```

```
class BoxDemo4 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        // assign values to mybox1's instance variables  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;  
  
        /* assign different values to mybox2's  
           instance variables */  
        mybox2.width = 3;  
        mybox2.height = 6;  
        mybox2.depth = 9;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

Adding a Method That Takes Parameters [1/2]

- While some methods don't need parameters, most do. Parameters allow a method to be generalized.
- data and/or be used in a number of slightly different situations

```
int square() {  
    return 10 * 10;  
}
```

```
int square(int i) {  
    return i * i;  
}
```

```
int x, y;  
x = square(5); // x equals 25  
x = square(9); // x equals 81  
y = 2;  
x = square(y); // x equals 4
```

Adding a Method That Takes Parameters [2/2]

```
// This program uses a parameterized method.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }

    // sets dimensions of box
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
```

```
class BoxDemo5 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

OUTPUT
Volume is 3000.0
Volume is 162.0

Questions?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 12

Constructor [1/2]

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- So we found a convenient functions like setDim() in previous examples
- But Java allows objects to initialize themselves when they are created. Which can be done by constructors
- It initialize object immediately upon creation

Constructor [2/2]

- Same name as of class
- syntactically similar to a method
- Doesn't have any return type (Not even void), this is because constructor is the class type itself
- called immediately after the object is created, before the **new** operator completes.
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- Default constructor automatically initializes all instance variables to zero.

Example

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // This is the constructor for Box.  
    Box() {  
        System.out.println("Constructing Box");  
        width = 10;  
        height = 10;  
        depth = 10;  
    }  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

OUTPUT:

```
Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0
```

```
class BoxDemo6 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
  
        double vol;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

Re-examine new Operator

- when you allocate an object, you use the following general form:

ClassName class-var = new ClassName();

`Box mybox1 = new Box();`

Parameterized Constructors

- The constructor which contains the parameters.
So that each object can be initialized with a specific parameters.
- Thus every time new object creation creates a new copy of object and one can pass different parameters to it.

Example

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

```
class BoxDemo7 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
  
        double vol;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

OUTPUT

Volume is 3000.0
Volume is 162.0

Questions?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 13 and 14

Recap

The **this** Keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this java have this keyword. It is used inside any method to refer to the current object.

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

- You can use this anywhere a reference to an object of the current class

Instance Variable Hiding [1/2]

- Within the same scope, same name of local variables is illegal in java.
- when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- Because **this** lets you refer directly to the object
- *This is why width, height, and depth were not used as the names*

Example

```
// Use this to resolve name-space collisions. Box  
(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

CAUTION: The use of such names in a context can sometimes be confusing

Garbage Collection

- As new operator allocates memory, then how that memory would reclaim when it is not in use by any program?
- In C++ it is done by calling **delete** operator
- But java handles memory management automatically (GC)
- When there is *no reference* to any object then it means that object is no longer needed and GC reclaim the memory.
- This occurs during the execution of your program.

The System.gc() Method

- Calling this suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse.

Syntax:

```
System.gc();
```

The finalize() Method

- Sometimes an object will need to perform some action when it is destroyed.
- such as a file handle or character font
- You might want to make sure these resources are *freed* before an object is destroyed.

```
protected void finalize()  
{  
    // finalization code here  
}
```

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

Example - A Stack Class

[1/2]

- A stack stores data using first-in, last-out ordering.
That is, a stack is like a stack of plates on a table
- the first plate put down on the table is the last plate
to be used
- Two operations to control:
 - Push
 - Pop

```

class Stack {
    int stck[] = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0)
            System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}

```

Example - A Stack Class

[2/2]

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}

```

Chapter Review

- Classes, objects, new keyword
- Methods
 - No Return Value and Return Value
 - No Parameter and Parameterized
- Constructors, Parameterized Constructors
- The *this* keyword, variable hiding and GC
- Method *System.gc()*
- Method *finalize()*

Questions?



GET READY
Quiz Time

1. What is the difference between a class and an object?
2. How is a class defined?
3. What does each object have its own copy of?
4. Using two separate statements, show how to declare an object called counter of a class called MyCounter.
5. Show how a method called myMeth() is declared if it has a return type of double and has two int parameters called a and b.

6. What name does a constructor have?
7. What does new do?
8. What is garbage collection, and how does it work? What is finalize()?
9. What is this keyword?
10. Can a constructor have one or more parameters?

Solution

1. Class vs. Object

Class

- Class is mechanism of binding data members and associated methods in a single unit
- It is logical existence
- Memory space is not allocated , when it is created
- Definition is created once

Object

- Instance of class or variable of class
- It is physical existence
- Memory space is allocated, when it is created
- it is created many time as you require

2. How class is defined

Class has 2 kind of members:

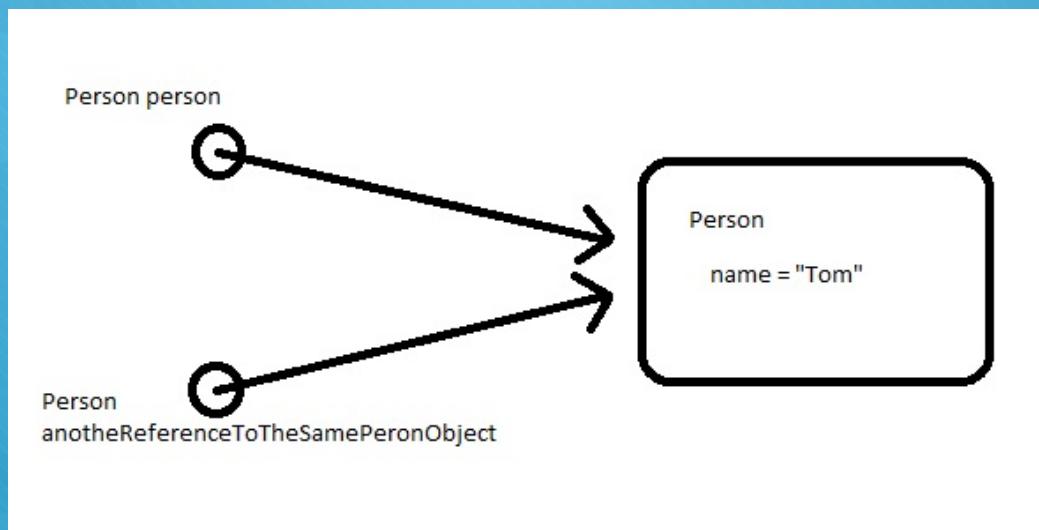
1. Instance variables
2. Instance methods

As many methods/variables can be placed into class definition

```
class name {  
    type variables;  
    ...  
    type methods () {  
        ...  
    }  
    ...  
}
```

3. Object have its own copy of references?

- Each object have its own copy of references



4. 2 Step process for declaring object of a class

- Basically, creating an object is a two step process:
- Step 1:

```
MyCounter counter;
```

- Step 2:

```
counter = new MyCounter();
```

5. myMeth() with 2 int param & double return type

- Specify type with every variable while using it in parameters its wrong to put all together (eg. Int a, b, c this is not allowed while dealing with methods)
- Whatever the return type is, a method must return the same type of data.

```
double myMeth (int a, int b) {  
    double result = a/b;  
    return result;  
}
```

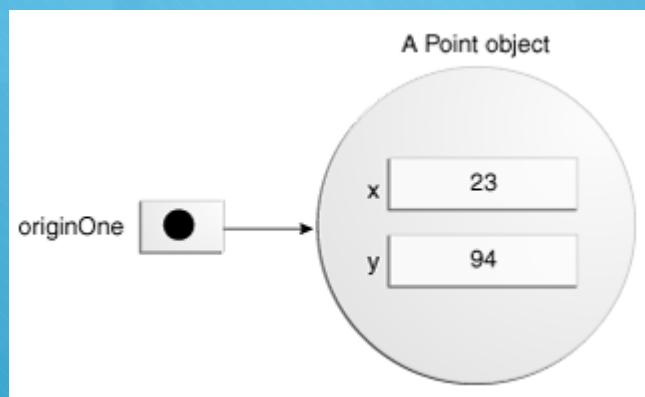
6. Constructor Name?

- A constructor have exactly same name as its class, its syntax is similar to method but don't have any return type

```
public class Person {  
  
    private String firstName;  
    private String lastName;  
    private String address;  
    private String username;  
  
    //The constructor method  
    public Person()  
    {  
  
    }  
}
```

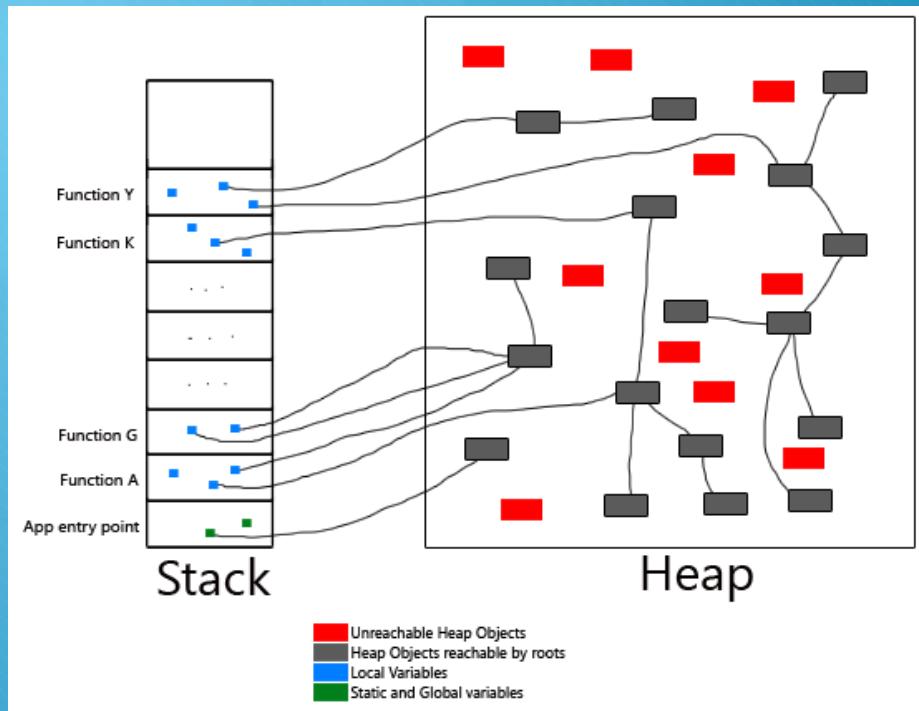
7. What does *new* keyword do?

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.



8. What is GC & how it works in Java?

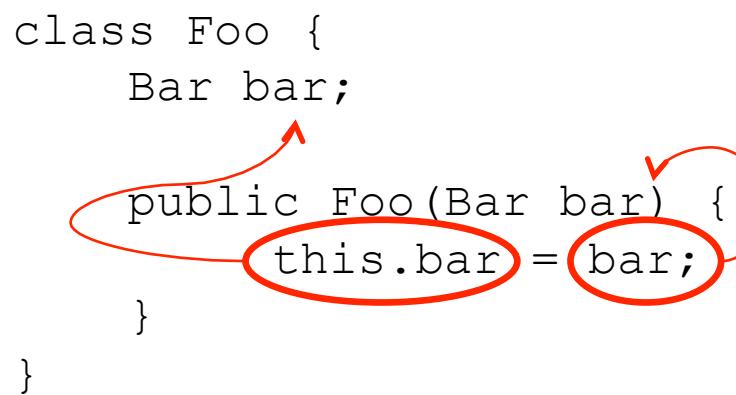
- Automatic garbage collection is the process of looking at heap memory, and perform operations as given below:
 - Identifying which objects are in use and which are not
 - And deleting the unused objects.



9. The this keyword

- The keyword this is a reference variable that refer to the current object
- There are also several other uses of this keyword...

```
class Foo {  
    Bar bar;  
  
    public Foo(Bar bar) {  
        this.bar = bar;  
    }  
}
```



10. Parameterized Constructors

- A constructor is just like a method, it can also have parameters as method have, but it doesn't have any return type

```
class Foo {  
    ...  
    ...  
    public Foo(int fooVar) {  
        ...  
        ...  
    }  
    ...  
}
```



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 16

Chapter # 07

A closer look at methods and classes

Topics covered in this chapter

- Overloading
 - Methods
 - Constructors
- Methods
 - Parameter Passing
- Access Controls
- Recursion
- Keyword static
- String class

Overloading Methods

- It is possible to define two or more methods within the same class that share the same name
- But their declaration must be different
- Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- One of Java's most exciting and useful features
- This is one of the ways that Java supports **Polymorphism**

Simple Example

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25) : 15190.5625

```
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }

    // overload test for a double parameter
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
```

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

Overloading Methods – Automatic Conversion

- o Java's automatic type conversions can play a role in overload resolution

```
void test(int i){  
    // Usage of Integer  
}
```

// to access that Method

```
obTest.test(124.53);
```

Simple Example

```
// Automatic type conversions apply to overloading.
```

```
class OverloadDemo {
```

```
void test() {
```

```
    System.out.println("No parameters");
```

```
}
```

```
// Overload test for two integer parameters.
```

```
void test(int a, int b) {
```

```
    System.out.println("a and b: " + a + " " + b);
```

```
}
```

```
// overload test for a double parameter and return type
```

```
void test(double a) {
```

```
    System.out.println("Inside test(double) a: " + a);
```

```
}
```

```
}
```

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

```
class Overload {
```

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

```
    OverloadDemo ob = new OverloadDemo();
```

```
    int i = 88;
```

```
    ob.test();
```

```
    ob.test(10, 20);
```

```
    ob.test(i); // this will invoke test(double)
```

```
    ob.test(123.2); // this will invoke test(double)
```

```
}
```

```
}
```

One Scenario...

- Since **Box()** requires three arguments, it's an error to call it without them.
- What if you simply wanted a box and did not care (or know) what its initial dimensions were?
- Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?
- *options are not available to you*

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Overloading Constructors

- Overloaded Constructor is solution to that kind of scenarios.
- In addition to overloading normal methods, you can also overload constructor
- A constructor can be with same name as class name.
- But the difference is again number of parameters or type of parameters

Example

```

class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
  
```

OUTPUT

Volume of mybox1 is 3000.0
 Volume of mybox2 is -1.0
 Volume of mycube is 343.0

```

class OverloadCons {
    public static void main(String args[]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
    }
}
  
```

Using Objects as Parameters

- Until now, we have only gave data types in parameters
- However it is possible to pass objects to a method

Object to Method Example

```
// Objects may be passed to methods.  
class Test {  
    int a, b;  
  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
  
    // return true if o is equal to the invoking object  
    boolean equals(Test o) {  
        if(o.a == a && o.b == b) return true;  
        else return false;  
    }  
}
```

OUTPUT
ob1 == ob2: true
ob1 == ob3: false

```
class PassOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(100, 22);  
        Test ob2 = new Test(100, 22);  
        Test ob3 = new Test(-1, -1);  
  
        System.out.println("ob1 == ob2: " +  
ob1.equals(ob2));  
  
        System.out.println("ob1 == ob3: " +  
ob1.equals(ob3));  
    }  
}
```

Any Question?



Thank you...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 17 and 18

Review

Object to its Constructor as Parameter

- One of the most common uses of object parameters involves constructors.
- Frequently, you will want to construct a new object so that it is initially the same as some existing object.

```

class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

```

Example

OUTPUT

Volume of mybox1 is 3000.0
 Volume of mybox2 is -1.0
 Volume of cube is 343.0
 Volume of clone is 3000.0

```

class OverloadCons2 {
    public static void main(String args[]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        Box myclone = new Box(mybox1);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of cube is " + vol);

        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}

```

A Closer Look at Argument Passing

- There are two ways that a computer language can pass an argument to a subroutine.
- **Call by Value:** Copy values of parameter to a subroutine, therefore change made in parameters of subroutine have no effect on arguments
- **Call by Reference:** A reference to an argument (not the value of the argument) is passed to the parameter. Access actual specified arguments. Therefore changes made to the parameter will affect the argument used to call the subroutine.

Arguments in JAVA

- When you pass a primitive type to a method, it is passed by value. And have no effect on arguments, this is call by value
- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference

Example – Call by value

```
// Simple Types are passed by value.  
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}
```

```
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
  
        System.out.println("a and b before call: " + a + " " + b);  
  
        ob.meth(a, b);  
  
        System.out.println("a and b after call: " + a + " " + b);  
    }  
}
```

OUTPUT

a and b before call: 15 20
a and b after call: 15 20

Example – Call by Reference

```
class Test {  
    int a, b;  
  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
  
    // pass an object  
    void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2;  
    }  
}
```

```
class CallByRef {  
    public static void main(String args[]) {  
        Test ob = new Test(15, 20);  
  
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);  
  
        ob.meth(ob);  
  
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);  
    }  
}
```

OUTPUT

ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

REMEMBER

When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

Returning Objects

- A method can return any type of data
- Even class type that you have created

Example

OUTPUT

ob1.a: 2
 ob2.a: 12
 ob2.a after second increase: 22

```
// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}
```

```
class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: " + ob2.a);
    }
}
```

Recursion [1/5]

- Recursion is the process of defining something in terms of itself
- In Java it is the attribute that allows a method to call itself
- A method that calls itself is said to be *recursive*.

Example – 1

OUTPUT

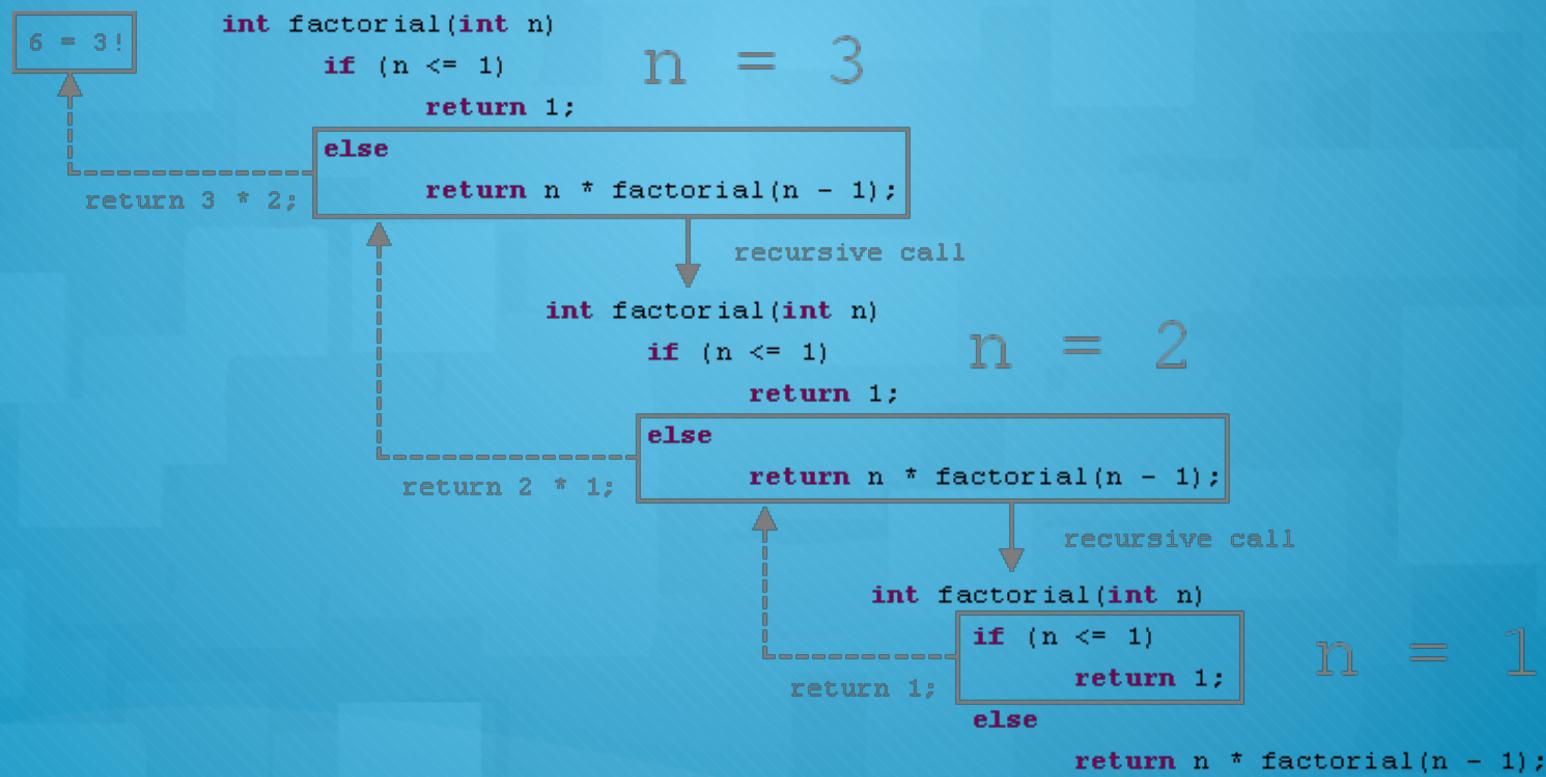
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

- The classic example of recursion is the computation of the factorial of a number

```
// A simple example of recursion.  
class Factorial {  
    // this is a recursive function  
    int fact(int n) {  
        int result;  
  
        if(n==1) return 1;  
        result = fact(n-1) * n;  
        return result;  
    }  
}
```

```
class Recursion {  
    public static void main(String args[]) {  
        Factorial f = new Factorial();  
  
        System.out.println("Factorial of 3 is " + f.fact(3));  
        System.out.println("Factorial of 4 is " + f.fact(4));  
        System.out.println("Factorial of 5 is " + f.fact(5));  
    }  
}
```

Elaboration of Example 1



Recursion [3/5]

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.
- For example, the
 - *QuickSort* sorting algorithm is quite difficult to implement in an iterative way
 - Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

Recursion [4/5]

- When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed.
- *If you don't do this, once you call the method, it will never return.*

Any Question?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 19

Review

Access Control

- Encapsulation provides another feature of access control
- By using controlling access you can prevent misuse
- For eg. Allowing access to only a few defined methods you can prevent from misuse
- Access specifiers determine that how a method can be accessed
- **NOTE: Classes already created in course were not meeting this goal**

Access Specifiers

- **public, private, and protected**

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
<i>no modifier*</i>	✓	✓	✗	✗
private	✓	✗	✗	✗

- Java also defines a default access level
- *Protected* applies only when inheritance is involved.
- The other access specifiers are

Public and Private

- Public: that member can be accessed by any other code.
- Private: that member can only be accessed by other members of its class
- Now you can understand why main is always public
- When no access specifier is defined then it have default access

Example:

```
public int i;  
private double j;  
private int myMethod(int a, char b) { // ...}
```

Example – Public and Private

```
/* This program demonstrates the  
difference between  
public and private.  
  
class Test {  
    int a; // default access  
    public int b; // public access  
    private int c; // private access  
  
    // methods to access c  
    void setc(int i) { // set c's value  
        c = i;  
    }  
    int getc() { // get c's value  
        return c;  
    }  
}
```

```
class AccessTest {  
    public static void main(String args[]) {  
        Test ob = new Test();  
  
        // These are OK, a and b may be accessed directly  
        ob.a = 10;  
        ob.b = 20;  
  
        // This is not OK and will cause an error  
        // ob.c = 100; // Error!  
  
        // You must access c through its methods  
        ob.setc(100); // OK  
  
        System.out.println("a, b, and c: " + ob.a + " " +  
                           ob.b + " " + ob.getc());  
    }  
}
```

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in **conjunction** with an object of its class. However, it is possible to create a member that can be used by **itself**, without reference to a specific instance.
- most common example of a static member is **main()**.

static

- Methods declared as static have several restrictions:
 - They can only call other static methods.
 - They must only access static data.
 - They cannot refer to this or super in any way.

static

- If you need to do computation in order to initialize your static variables, you can declare a **static block** that gets executed exactly once, when the class is first loaded.

Example

OUTPUT

Static block initialized. x = 42
a= 3
b = 12

listing 15

```
// Demonstrate static variables, methods, and
blocks.

class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String args[]) {
        meth(42);
    }
}
```

static

- A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

static

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}
```

```
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " +  
        StaticDemo.b);  
    }  
}
```

OUTPUT
a = 42
b = 99

Introducing **final**

- Using Final prevents its contents from being modified.
- These can behave like as if they were constants, without fear that a value has been changed
- Common coding convention is, to choose all uppercase identifiers for final variables.
- The keyword final can also be applied to methods, but it works differently with methods and variables.

Example

```
final int FILE_NEW = 1;  
final int FILE_OPEN = 2;  
final int FILE_SAVE = 3;  
final int FILE_SAVEAS = 4;  
final int FILE_QUIT = 5;
```

Any Question?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 20

- Classes and Objects
- Methods and Constructor

Basics revisited

Problem

1. Do something.
2. Do something else.
3. Check this.
4. Draw that.
5. Did something happen?
6. If so, do this.

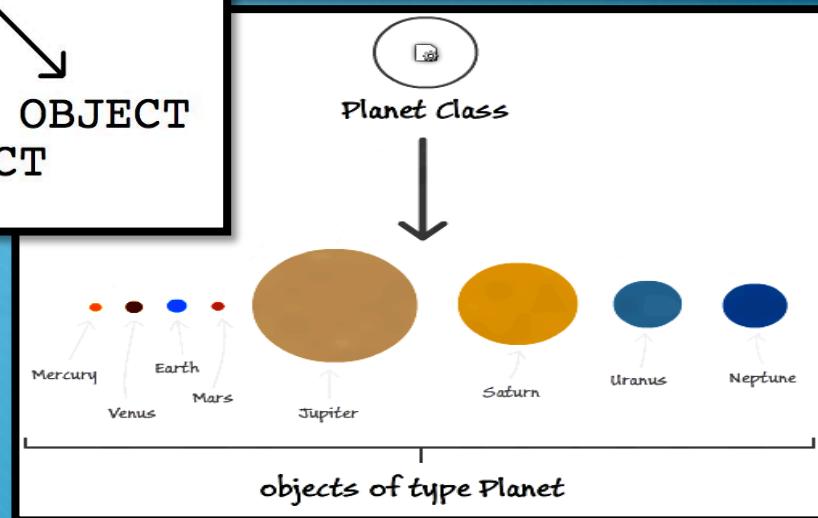
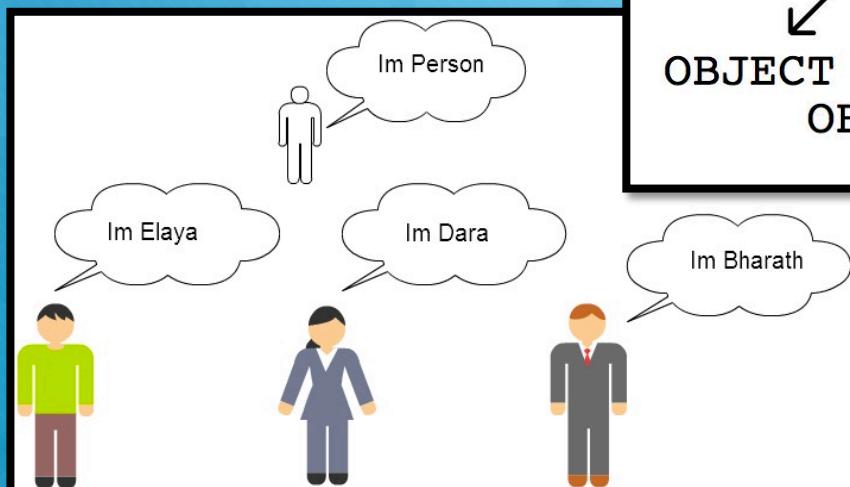
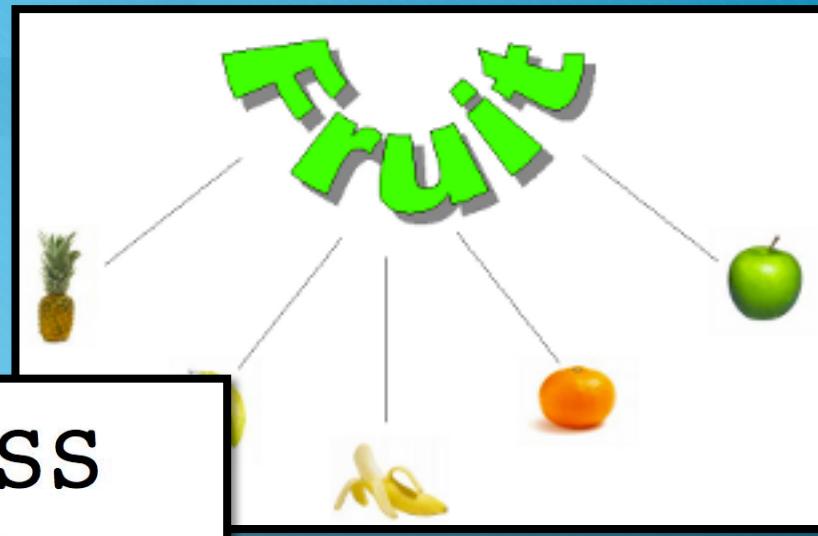
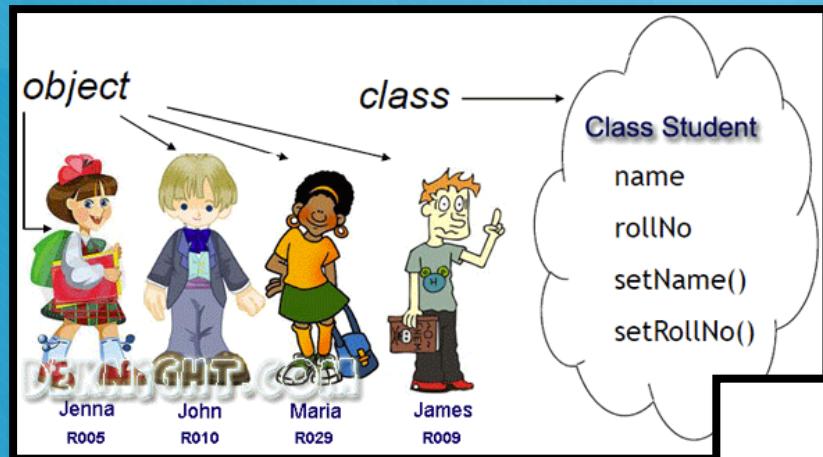
...

Procedural Programming

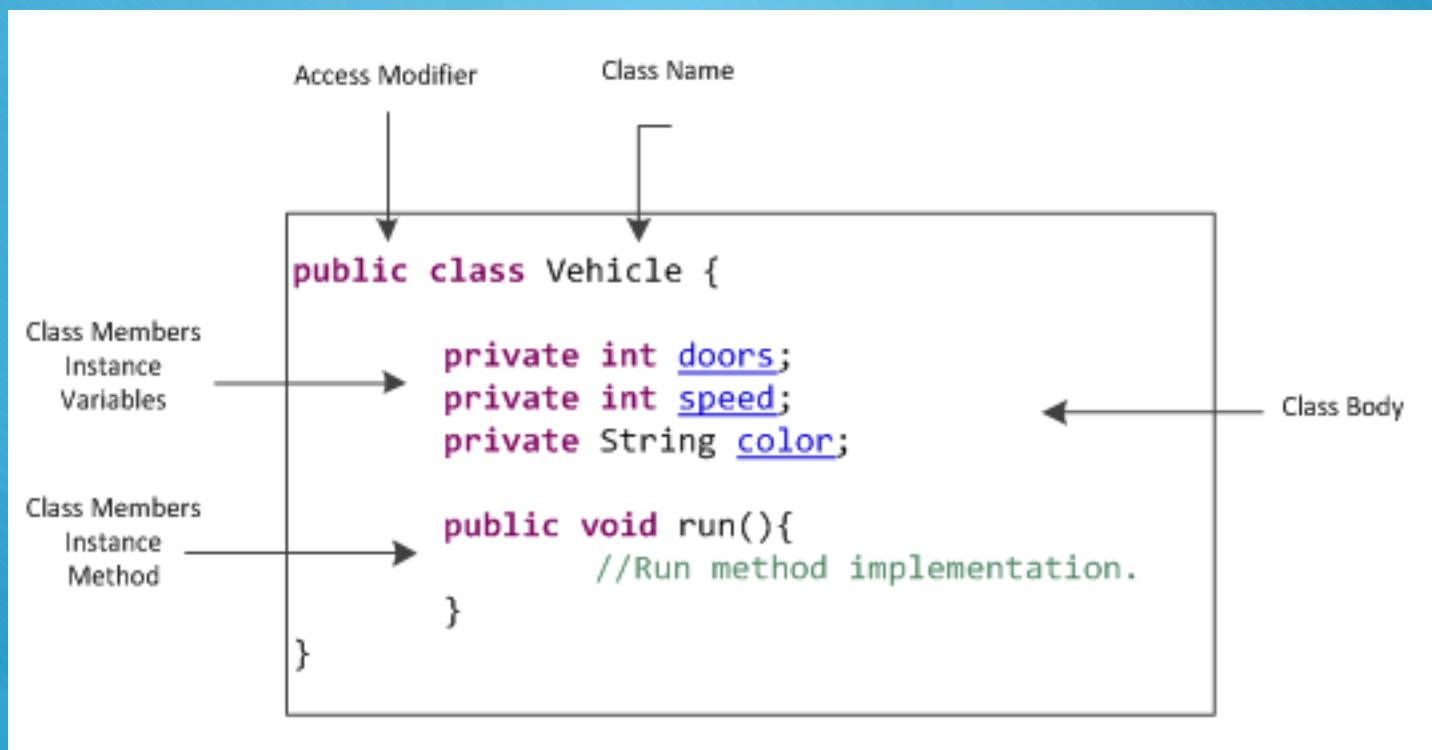
Class and objects in Java

- In the real world, you'll often find many individual objects all of the same kind.
- There may be thousands of other bicycles in existence, all of the same make and model.
- Each bicycle was built from the same set of blueprints and therefore contains the same components.
- In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles.
- A class is the blueprint from which individual objects are created.

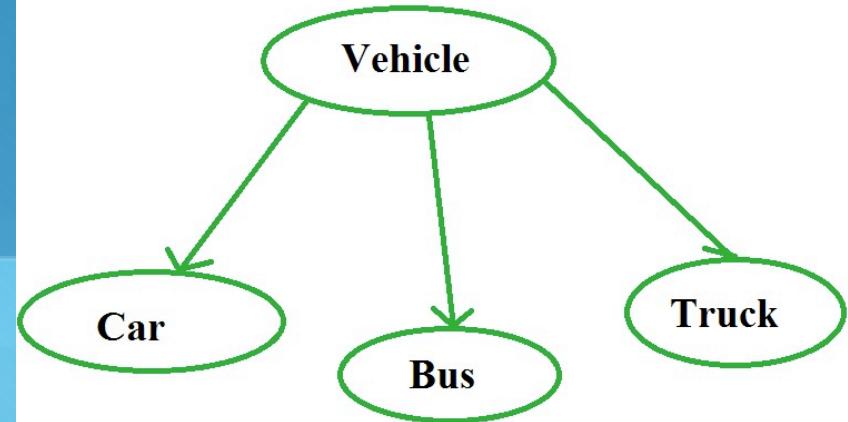
Class and Objects



Class



Class Vehicle



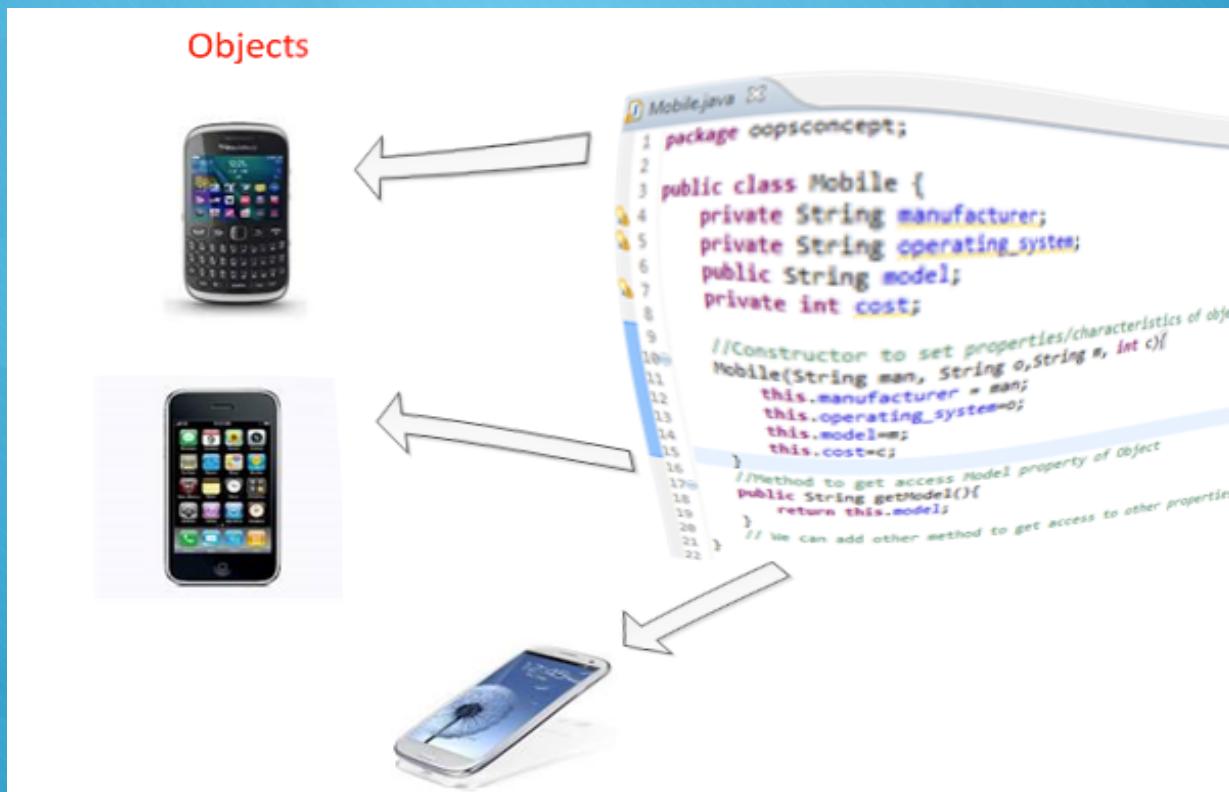
```
Vehicle car = new Vehicle();
```

```
Vehicle Bus = new Vehicle();
```

```
Vehicle Truck = new Vehicle();
```

Another Example

Objects



The diagram illustrates the concept of objects. Three different mobile phones (a feature phone, an iPhone, and a smartphone) are shown, each with an arrow pointing towards a Java code snippet. The code defines a class named 'Mobile' with properties for manufacturer, operating system, model, and cost, along with a constructor and a method to get the model.

```
1 package oopsconcept;
2
3 public class Mobile {
4     private String manufacturer;
5     private String operating_system;
6     public String model;
7     private int cost;
8
9     //Constructor to set properties/characteristics of object
10    Mobile(String man, String os, String m, int c){
11        this.manufacturer = man;
12        this.operating_system = os;
13        this.model = m;
14        this.cost = c;
15    }
16
17    //Method to get access Model property of object
18    public String getModel(){
19        return this.model;
20    }
21
22    // We can add other method to get access to other properties
}
```

Methods

- A Java method is a collection of statements that are grouped together to perform an operation.
- When you call the `System.out.println` method, for example, the system actually executes several statements in order to display a message on the console.

Opening Problem

Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

Problem

```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
    sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
    sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Problem

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;  
  
System.out.println("Sum from 1 to 10 is " + sum);
```

```
sum = 0;  
for (int i = 20; i <= 30; i++)  
    sum += i;  
  
System.out.println("Sum from 20 to 30 is " + sum);
```

```
sum = 0;  
for (int i = 35; i <= 45; i++)  
    sum += i;  
  
System.out.println("Sum from 35 to 45 is " + sum);
```

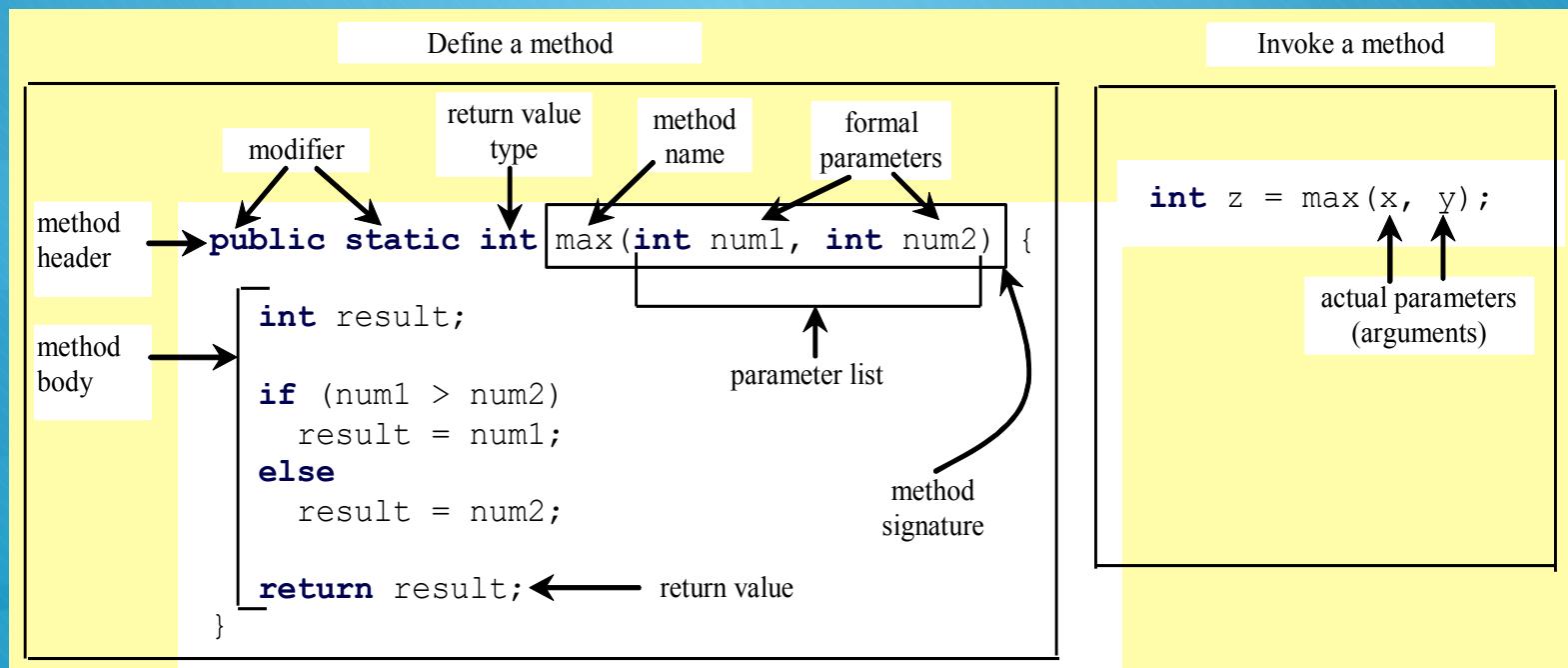
Solution

```
public static int sum(int i1, int i2) {  
    int sum = 0;  
    for (int i = i1; i <= i2; i++)  
        sum += i;  
    return sum;  
}
```

```
public static void main(String[] args) {  
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));  
    System.out.println("Sum from 20 to 30 is " + sum(20, 30));  
    System.out.println("Sum from 35 to 45 is " + sum(35, 45));  
}
```

Method and its anatomy

A method is a collection of statements that are grouped together to perform an operation.



Calling Methods

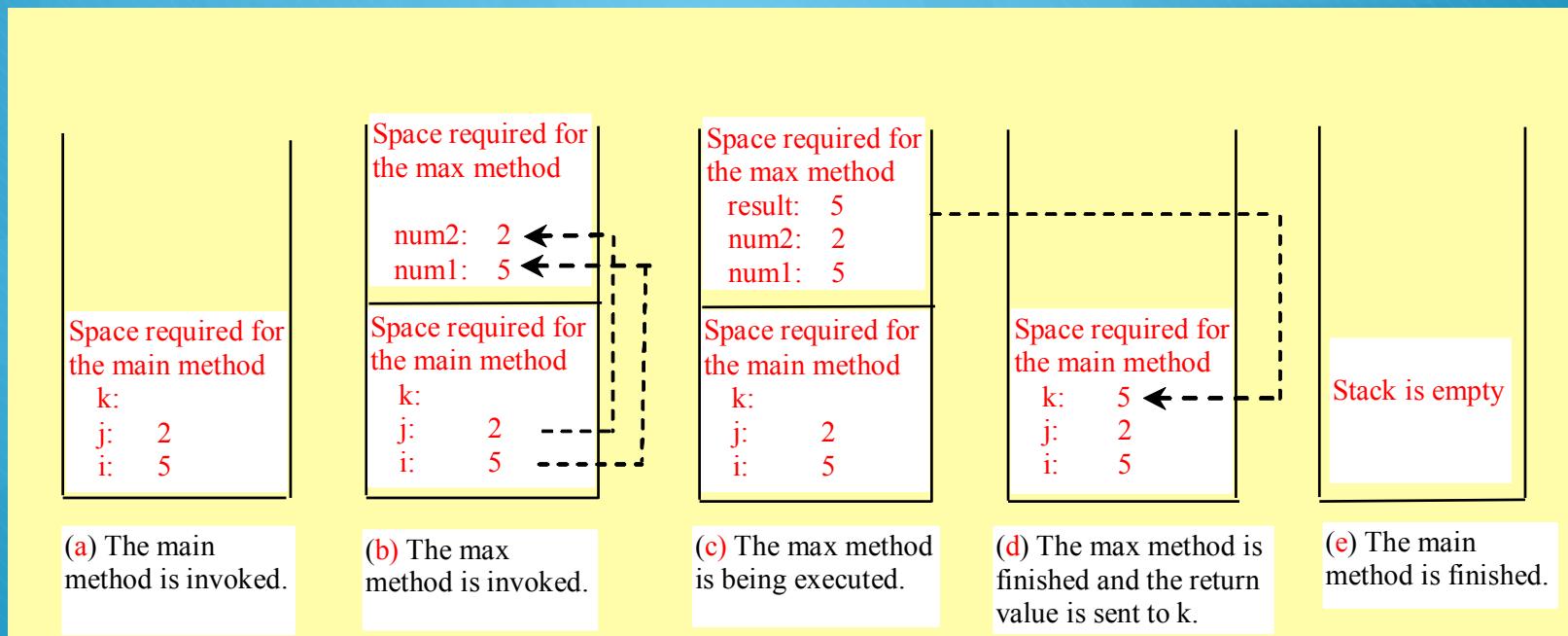
```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

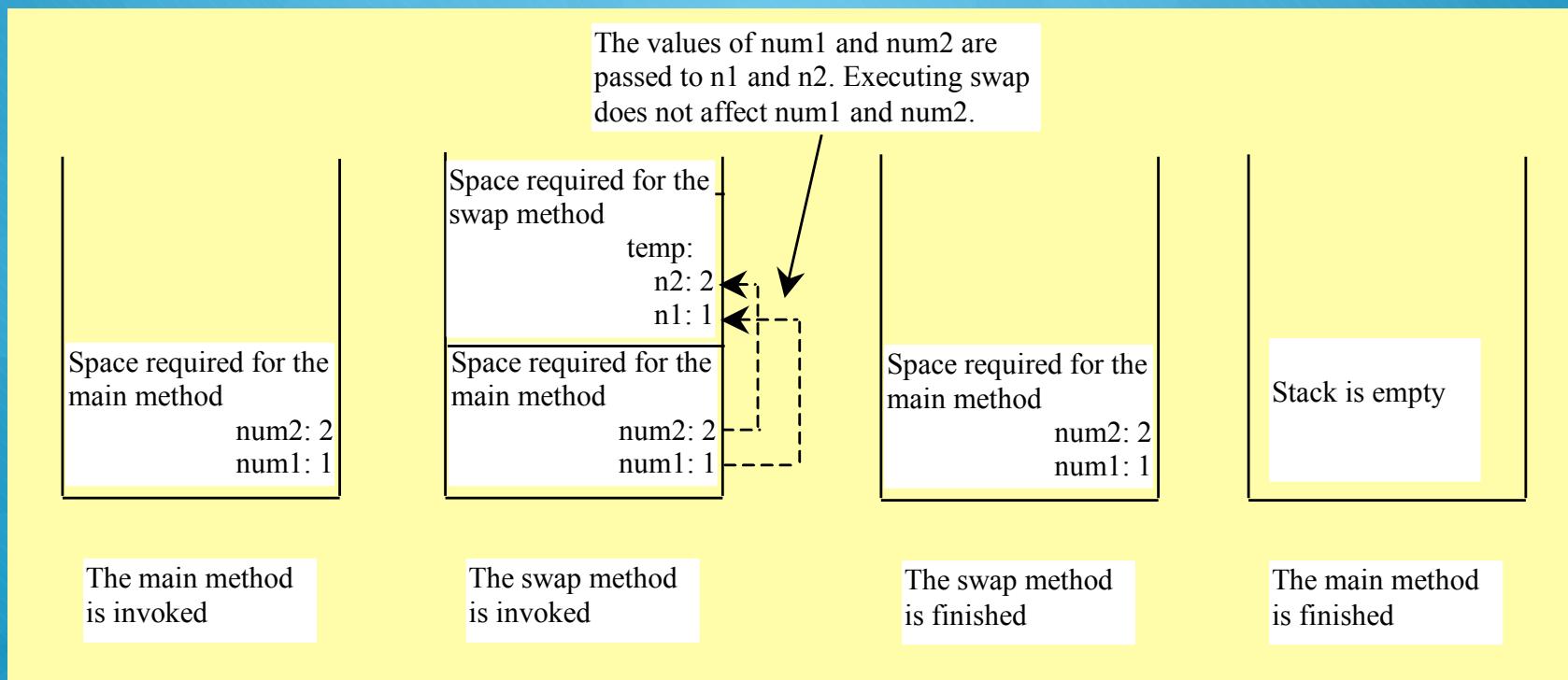
pass the value of i

pass the value of j

Call Stacks



Method, return type void



What is the Output?

```
public static void nPrintln(String message, int n) {  
    for (int i = 0; i < n; i++)  
        System.out.println(message);  
}
```

**Suppose you invoke the method using
nPrintln("Welcome to Java", 5);**

**Suppose you invoke the method using
nPrintln("Computer Science", 15);**

Constructor

- Constructor in Java is block of code which is executed at the time of Object creation.
- But other than getting called, Constructor is entirely different than methods and has some specific properties like: name of constructor must be same as name of Class.
- Constructor also can not have any return type
- constructor's are automatically chained by using this keyword and super.
- Since Constructor is used to create object, object initialization code is normally hosted in Constructor.

Problem

```
Class Temprature{  
    double myDegree;  
    char myScale;  
}
```

Constructors

- Initial values for any class variable(s) are **0/NUL**
 - *Better to give them valid values*
- Constructor method used to give these values
 - default-value constructor
 - explicit value constructor

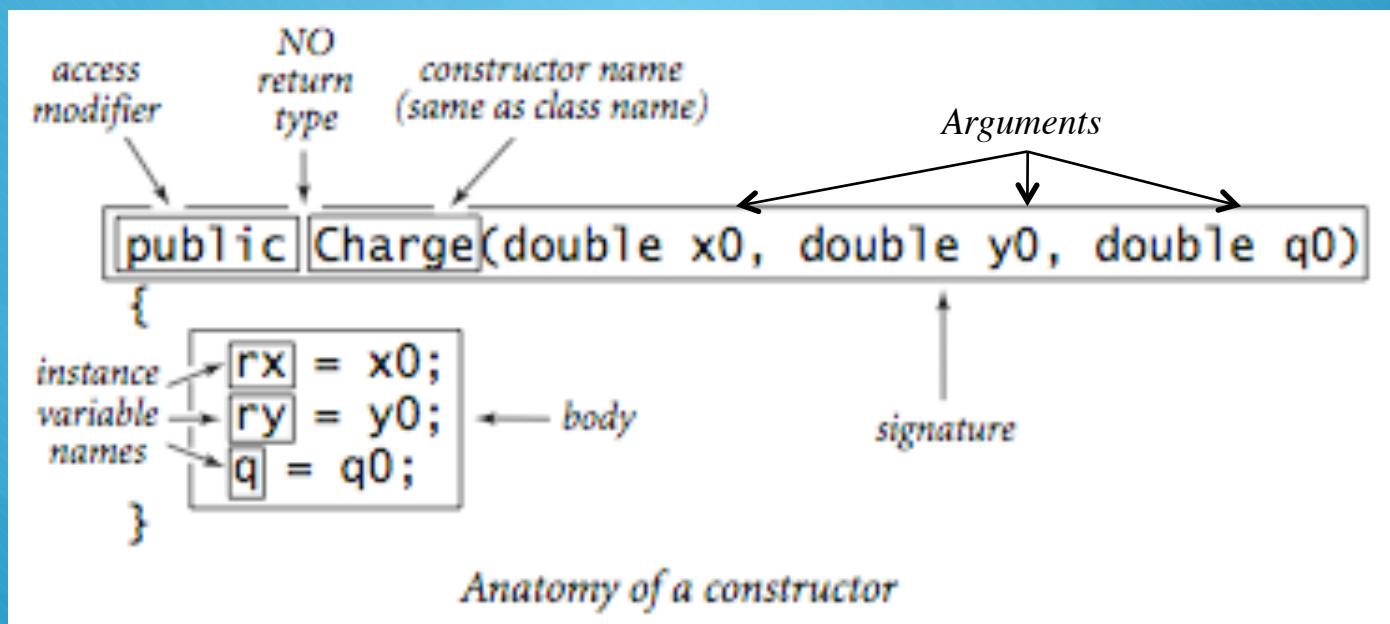
Default Value Constructor

- Whenever a **Temperature** object is declared, this specifies initial values

```
public    Temperature()
{ myDegrees = 0.0;
  myScale = 'C'; }
```

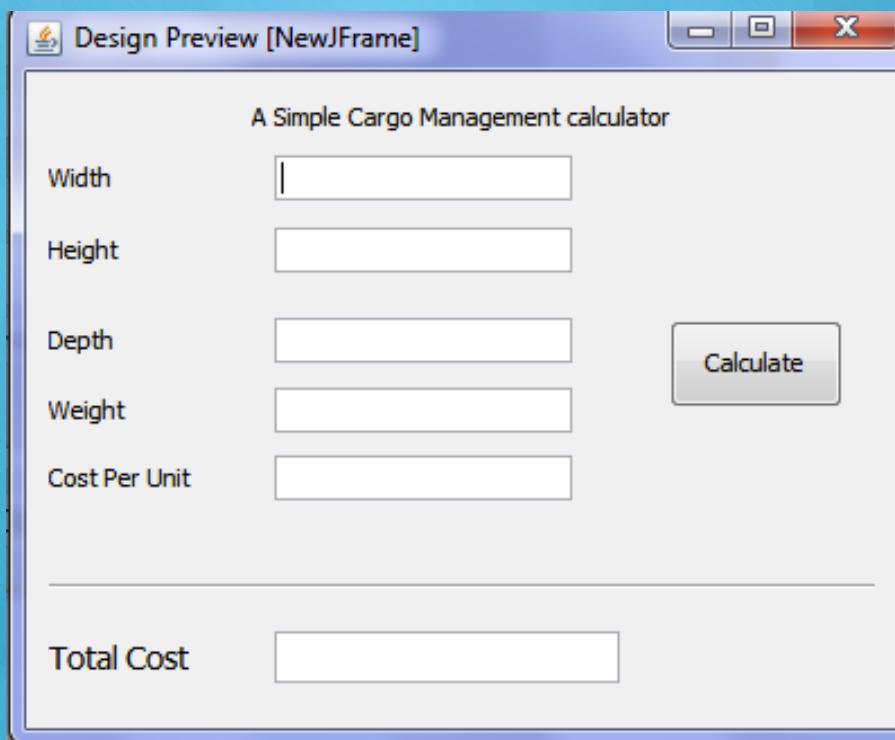
- Note:
 - no return type (not even void)
 - name of constructor method must be same as name of class

Constructor and its anatomy

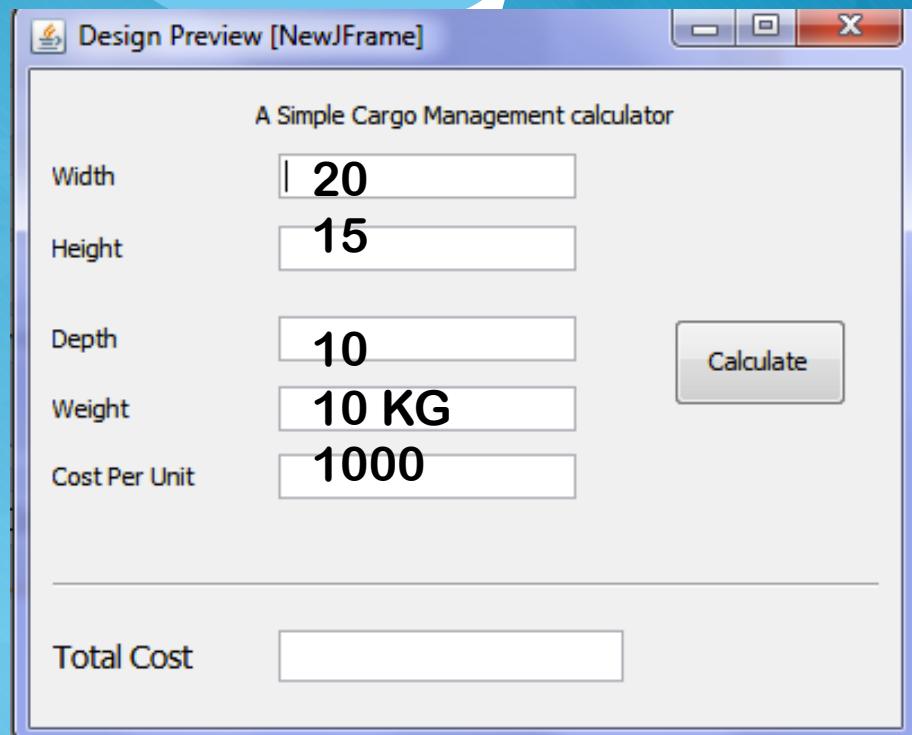


Real life scenario

Why we need constructors?



This is why you need constructors...



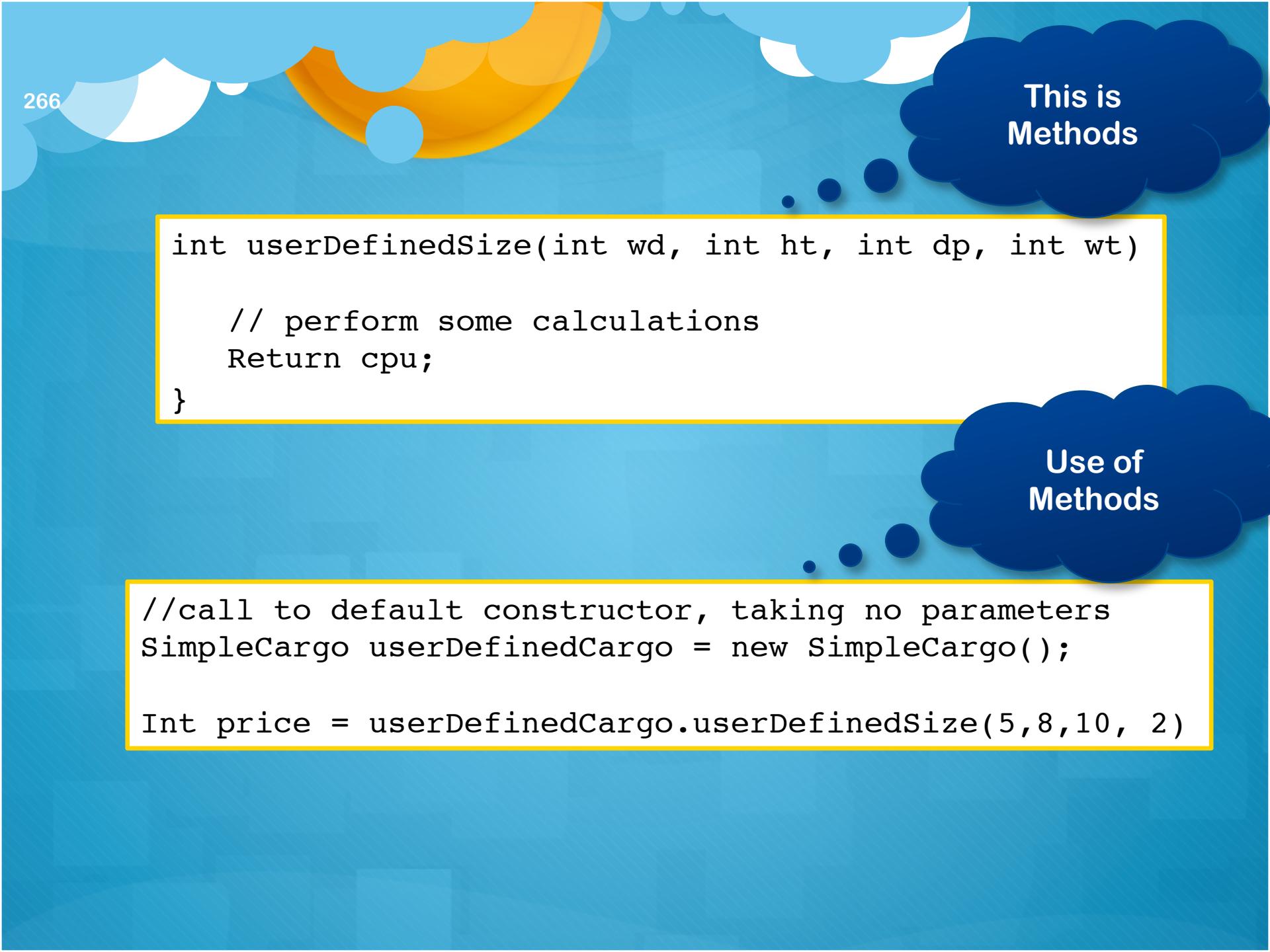
```
class SimpleCargo {  
    int width, height, depth;  
    int weight, costPerUnit;  
  
    SimpleCargo(int wd, int ht, int dp, int wt, int cpu)  
        width = wd; height = ht; depth=dp;  
        weight = wt; costPerUnit = cpu;  
}
```

```
class SimpleCargo {  
    int width, height, depth;  
    int weight, costPerUnit;  
  
    SimpleCargo(int wd, int ht, int dp, int wt, int cpu)  
        width = wd; height = ht; depth=dp;  
        weight = wt; costPerUnit = cpu;  
}
```

This is
Constructor

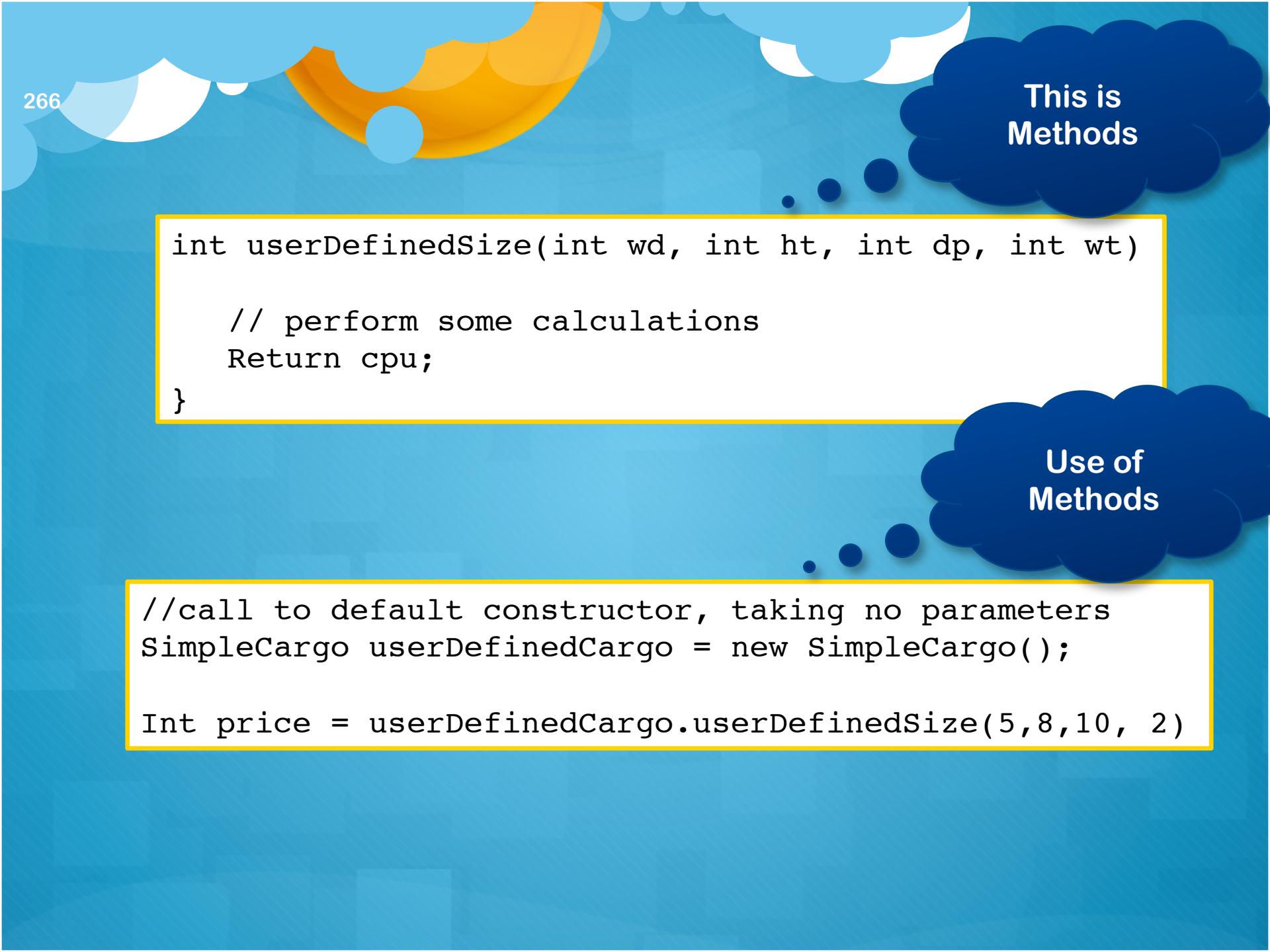
Use of
constructor

```
SimpleCargo Eid = new SimpleCargo(20,15,10,10,1000);  
SimpleCargo DatesSeason = new SimpleCargo (20, 20, 20, 2000);  
SimpleCargo MangoesSeason = new SimpleCargo(24,10,30, 2800);
```



This is
Methods

```
int userDefinedSize(int wd, int ht, int dp, int wt)  
  
    // perform some calculations  
    Return cpu;  
}
```



Use of
Methods

```
//call to default constructor, taking no parameters  
SimpleCargo userDefinedCargo = new SimpleCargo();  
  
Int price = userDefinedCargo.userDefinedSize(5,8,10, 2)
```

Assignments # 6

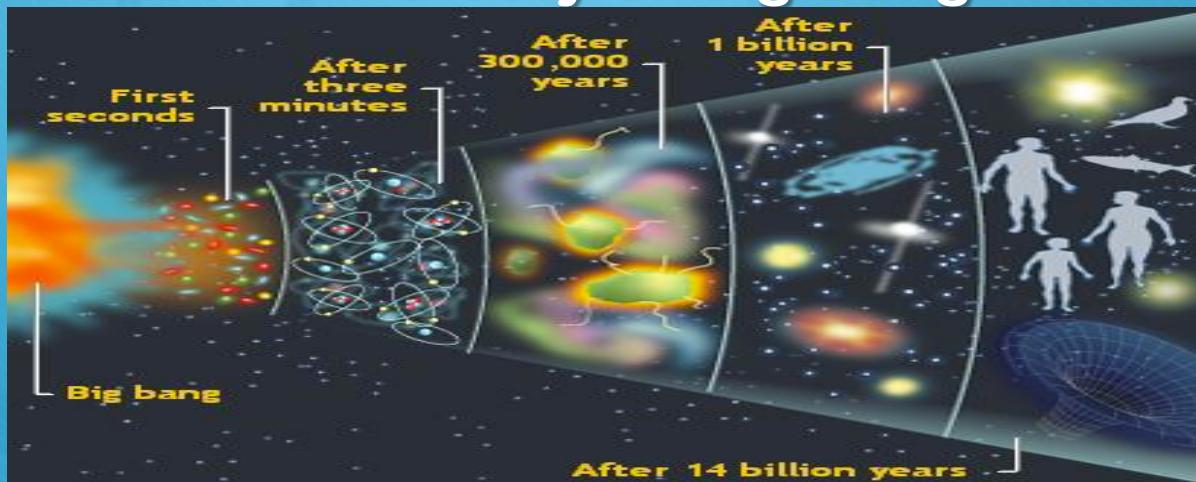
Last date of submission: 5 March, 2014

Part I – Exercise

- Create a mobile class, and create 3 instances of that, create method to change its color
- Create Vehicle class and create 3 instances of that, create methods to change its size and color.
- Create an Architect class, and it must have a default design (provide by constructor), and then create methods which can help architect to design upon user defined designs
- Make greeting application program, which prints “Welcome to my application” for first time, if you want to iterate the greetings it displays “Welcome again to my application” (Use constructor and methods)

Part I – Exercise... cont.

- Create an Earth planet history and add methods to describe its behaviors time to time. You can take an start from theory of Big Bang!



Part II – Theory

- Write in your own words, you own ideas, you own perceptions...
- What is class
- What is Object
- What is Method
- What is Constructor
- And a Scenario fits with all definitions

Any Question?



Thanks...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 22

Nested Classes

- It is possible to define a class within another class, *nested classes*.
- Thus, if class B is defined within class A, then B does not exist independently of A
- An inner class is a non-static nested class.
- An inner class has access to all of the members of its enclosing class, but the reverse is not true.

Nested Classes - Example

```
// Demonstrate an inner class.  
class Outer {  
    int outer_x = 100;  
  
    void test() {  
        Inner inner = new Inner();  
        inner.display();  
    }  
  
    // this is an inner class  
    class Inner {  
        void display() {  
            System.out.println("display: outer_x = " + outer_x);  
        }  
    }  
}
```

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

Another Example

```
class Outer {  
    int outer_x = 100;  
  
    void test() {  
        Inner inner = new Inner();  
        inner.display();  
    }  
  
    // this is an innner class  
    class Inner {  
        int y = 10; // y is local to Inner  
        void display() {  
            System.out.println("display: outer_x = " +  
outer_x);  
        }  
    }  
  
    void showy() {  
        System.out.println(y); ← // error, y not known here!  
    }  
}
```

This Program
will not work

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

// error, y not known here!

Inner Class within any block scope

- it is possible to define inner classes within any block scope
- For example, you can define a nested class within the block defined by a method or even within the body of a for loop,

Example

```
// Define an inner class within a for loop.  
class Outer {  
    int outer_x = 100;  
  
    void test() {  
        for(int i=0; i<10; i++) {  
            class Inner {  
                void display() {  
                    System.out.println("display: outer_x = " + outer_x);  
                }  
            }  
            Inner inner = new Inner();  
            inner.display();  
        }  
    }  
}
```

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

OUTPUT:

Exploring String Class

[1/3]

- **The first thing** to understand about strings is:
 - that every string you create is actually an object of type String
 - Even string constants are actually String objects

For eg.

```
System.out.println("This is a String, too");
```

Exploring String Class

[2/3]

- **The second** thing to understand is:
- once a String object is created, its contents cannot be altered
- If you need to change a string, you can always create a new one that contains the modifications
- Java defines a peer class of String, called **StringBuffer**, which allows strings to be altered

String usage

```
String myString = "this is a test";
```

- Once created a string can be used in variety of ways.

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

- Concatenation in strings

```
String myString = "I" + " like " + "Java.;"
```

- results in myString containing “I like Java.”

Example

OUTPUT

First String

Second String

First String and Second String

```
class StringDemo {  
    public static void main(String args[]) {  
        String strOb1 = "First String";  
        String strOb2 = "Second String";  
        String strOb3 = strOb1 + " and " + strOb2;  
        System.out.println(strOb1);  
        System.out.println(strOb2);  
        System.out.println(strOb3);  
    }  
}
```

Useful methods of String class

- The String class contains several methods that you can use.
- equals()
- length()
- charAt()

General forms for all:

boolean equals(String object)

int length()

char charAt(int index)

Example

```
class StringDemo2 {  
    public static void main(String args[]) {  
        String strOb1 = "First String";  
        String strOb2 = "Second String";  
        String strOb3 = strOb1;  
  
        System.out.println("Length of strOb1: " + strOb1.length());  
  
        System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));  
  
        if(strOb1.equals(strOb2))  
            System.out.println("strOb1 == strOb2");  
        else  
            System.out.println("strOb1 != strOb2");  
  
        if(strOb1.equals(strOb3))  
            System.out.println("strOb1 == strOb3");  
        else  
            System.out.println("strOb1 != strOb3");  
    }  
}
```

OUTPUT

Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3

Example

```
// Demonstrate String arrays.  
class StringDemo3 {  
    public static void main(String args[]) {  
        String str[] = { "one", "two", "three" };  
  
        for(int i=0; i<str.length; i++)  
            System.out.println("str[" + i + "]: " +  
                               str[i]);  
    }  
}
```

OUTPUT
str[0]: one
str[1]: two
str[2]: three

Varargs: Variable-Length Arguments

- A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.
- A method that require variable number of arguments is not unusual.
 - Example: A method that interact with internet connection may ask for username, password, authentication file and protocol...
 - Another example is the printf()

Old approach

- Previously variable-length arguments could be handled two ways:
 - First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method
 - Second approach, used when number of arguments is unknown, then the array was passed to the method.

```
// Use an array to pass a variable number of
// arguments to a method.
class PassArray {
    static void vaTest(int v[]) {
        System.out.print("Number of args: " + v.length + " Contents: ");

        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        // Notice how an array must be created to
        // hold the arguments.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };

        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}
```

Example

OUTPUT

Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:

Questions?



Thank you...



Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 23 and 24

New Approach – vararg

- A variable-length argument is specified by three periods (...). For example, here is how vaTest() is written using a vararg:

```
static void vaTest(int ... v) {
```

- This syntax tells the compiler that vaTest() can be called with zero or more arguments
- v is implicitly declared as an array of type int[]
- Thus, inside vaTest(), v is accessed using the normal array syntax

```
class VarArgs {  
  
    // vaTest() now uses a vararg.  
    static void vaTest(int ... v) {  
        System.out.print("Number of args: " + v.length + " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    public static void main(String args[]) {  
  
        // Notice how vaTest() can be called with a  
        // variable number of arguments.  
        vaTest(10);    // 1 arg  
        vaTest(1, 2, 3); // 3 args  
        vaTest();      // no args  
    }  
}
```

Example

OUTPUT

Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:

vararg variations

- A method can have “normal” parameters along with a variable-length parameter.
- However, the variable-length parameter must be the last parameter declared by the method.

```
int doIt(int a, int b, double c, int ... vals) {
```

- *Remember, the varargs parameter must be last. And must not be more than one vararg parameter*

```
int doIt(int a, int b, int ... vals, boolean stopFlag){ // Error!
```

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

```
// Use varargs with standard arguments.  
class VarArgs2 {  
  
    // Here, msg is a normal parameter and v is a  
    // varargs parameter.  
    static void vaTest(String msg, int ... v) {  
        System.out.print(msg + v.length + " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
    public static void main(String args[])  
    {  
        vaTest("One vararg: ", 10);  
        vaTest("Three varargs: ", 1, 2, 3);  
        vaTest("No varargs: ");  
    }  
}
```

Example

OUTPUT

One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:

Overloading Vararg Methods

- You can also overload vararg methods, just as like other methods

3/ i) Varargs and overloading.

```
class VarArgs3 {  
  
    static void vaTest(int ... v) {  
        System.out.print("vaTest(int ...): " +  
            "Number of args: " + v.length +  
            " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    static void vaTest(boolean ... v) {  
        System.out.print("vaTest(boolean ...) " + "Number of args: " + v.length + " Contents: ");  
  
        for(boolean x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    static void vaTest(String msg, int ... v) {  
        System.out.print("vaTest(String, int ...): " +  
            msg + v.length +  
            " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
}
```

OUTPUT

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3  
vaTest(String, int ...): Testing: 2 Contents: 10 20  
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

Example

```
public static void main(String args[])  
{  
    vaTest(1, 2, 3);  
    vaTest("Testing: ", 10, 20);  
    vaTest(true, false, false);  
}
```

Overloading vararg method

- It is possible to overload vararg with non-vararg methods:

`vaTest(int x)` is a valid `vaTest()`

- This version is invoked only when one int argument is present.
- When two or more int arguments are passed following is used:

`vaTest(int...v)`

Varargs and Ambiguity

- Careful crafting is needed, when two overloaded methods have parameters of varargs, and both fall into ambiguous condition that, which one to call?

vararg Ambiguity Example

```
class VarArgs4 {  
  
    static void vaTest(int ... v) {  
        System.out.print("vaTest(Integer ...): " +  
            "Number of args: " + v.length +  
            " Contents: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    static void vaTest(boolean ... v) {  
        System.out.print("vaTest(boolean ...) " +  
            "Number of args: " + v.length +  
            " Contents: ");  
  
        for(boolean x : v)  
            System.out.print(x + " ");  
  
        System.out.println();  
    }  
  
    public static void main(String args[]) {  
        vaTest(1, 2, 3); // OK  
        vaTest(true, false, false); // OK  
  
        vaTest(); // Error: Ambiguous!  
    }  
}
```

ERROR

Chapter 8 – A Review

- Overloading Methods
- Overloading Constructors
- Using Objects as Parameters
- A Closer Look at Argument Passing
- Returning Objects
- Recursion
- Introducing Access Control
- Understanding static
- Introducing final
- Arrays Revisited
- Introducing Nested and Inner Classes
- Exploring the String Class
- Using Command-Line Arguments
- Varargs: Variable-Length Arguments
- Overloading Vararg Methods
- Varargs and Ambiguity

SIBA Programming contest

Something you must need to know before you appear in contest

File I/O

- As we have learned so far making different logics using data from internal source.
- But most of the programs couldn't accomplish their goal without accessing external data.
 - A network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
 - Although physically different, these devices are all handled by the same abstraction: the **stream**.

Streams [1/2]

- A **stream** is an object that allows for the flow of data between your program and some I/O device or some file.
- If flow is into your program, the stream is called an **input stream**.
- If flow is out of your program, the stream is called **output stream**.
- If input stream flows from keyboard, then your program will take *input from the keyboard*.
- If input stream flows from a file, then your program will take *its input from that file*.
- Similarly, an output stream can go to the screen or to a file.

Streams [2/2]

- Although you may not realize it, you have already been using streams in your programs when you have output something to the screen.
- **System.out** (used in `System.out.println`) is an output stream connected to the screen.
- **System.in** is an input stream connected to the keyboard

```
Scanner keyboard = new Scanner(System.in);
```

Simple file I/O

File i/o – Example 1

Check an existing file

```
import java.io.File;  
  
public class FileIOTest1{  
  
    public static void main(String[] args) {  
        File x = new File("C:\\test\\text.txt");  
        if(x.exists())  
            System.out.println(x.getName()  
+"Exist");  
        else  
            System.out.println("Missing");  
    }  
}
```

Creating file in current folder

```
import java.io.*;
public class FileExample {
    public static void main(String[] args) {
        boolean flag = false;
        // create File object
        File stockFile = new File("d://Stock/stockFile.txt");

        try {
            flag = stockFile.createNewFile();
        } catch (IOException ioe) {
            System.out.println("Error while Creating File in Java" + ioe);
        }

        System.out.println("stock file" + stockFile.getPath() + " created ");
    }
}
```

Create Folder at a given location

```
import java.io.*;
public class DirectoryExample {
    public static void main(String[] args) {
        boolean dirFlag = false;
        // create File object
        File stockDir = new File("d://Stock/ stockDir ");
        try {
            dirFlag = stockDir.mkdir();
        } catch (SecurityException Se) {
            System.out.println("Error while creating directory in Java:" + Se);
        }
        if (dirFlag)
            System.out.println("Directory created successfully");
        else
            System.out.println("Directory was not created successfully");
    }
}
```

File i/o – Example 2

Create a file

```
import java.util.*;  
  
public class FileIOTest2{  
  
    public static void main(String[] args) {  
        final Formatter x;  
        try{  
            x = new Formatter ("File.txt");  
            System.out.println("You have  
created a file");  
        }  
        catch(Exception e){  
            System.out.println("You got an error");  
        }  
    }  
}
```



```
import java.io.*;
import java.lang.*;
import java.util.*;

public class FileIOTest3{

    Formatter x;

    public void openFile(){
        try{
            x = new Formatter ("File.txt");
            System.out.println("You have created a file");
        }
        catch(Exception e){
            System.out.println("You got an error");
        }
    }

    public void addRecords(){
        x.format("%s %d %d\n", "Raheel ",1234,343);
        System.out.println("Record Added Successfully.");
    }

    public void closeFile(){
        x.close();
        System.out.println("File Closed Successfully");
    }
}
```

```
public class testFile{
    public static void main(String[] args) {
        FileIOTest f3 = new GUI();
        f3.openFile();
        f3.addRecords();
        f3.addRecords();
        f3.closeFile();
    }
}
```

Add the
record
to the
file

Read from existing file

```
import java.io.*;
import java.lang.*;
import java.util.*;

public class TestFileIO4{
    Scanner x;
    void openFile(){
        try {
            x = new Scanner(new File("File.txt"));
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }
    void readFile(){
        while(x.hasNext()){
            String name = x.next();
            int account=x.nextInt();
            int balance=x.nextInt();
            System.out.printf("%s %d %d\n",name,
account, balance);
        }
    }
    public void closeFile(){
        x.close();
    }
}
```

```
public static void main(String[] args) {
    TestFileIO4 t4= new GUI();
    t4.openFile();
    t4.readFile();
    t4.closeFile();
}
```

Some Streams...

//Reading from console

```
BufferedReader stdin = new BufferedReader(new  
InputStreamReader(System.in));
```

// Reading from file

```
BufferedReader br=new BufferedReader(new FileReader  
("input.txt"));
```

//Reading from network

```
BufferedReader br = new BufferedReader(new  
InputStreamReader (s.getInputStream()));
```

Assignment - 6

- Read about Streams in Java
- And make programs to read and write using streams

Questions?



Thank you...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 25



Chapter # 08

Inheritance

Inheritance

- Cornerstone of OOP
- Allows the creation of hierarchical classification
- The class that's inherited is called Superclass
- And the class that does the inheritance is called subclass.
- Contains all of instance variables and methods of superclass plus its own unique elements

extends

- Word **extends** is used to inherit the class

SYNTAX:

```
-----  
class superclass-name {  
    //body of class;  
}  
-----
```

```
class subclass-name extends superclass-name {  
    //body of class;  
}  
-----
```

Example

OUTPUT

Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

// Create a superclass.

class A {

 int i, j;

 void showij() {

 System.out.println("i and j: " + i + " " + j);

}

}

// Create a subclass by extending class A.

class B extends A {

 int k;

 void showk() {

 System.out.println("k: " + k);

}

 void sum() {

 System.out.println("i+j+k: " + (i+j+k));

}

}

class SimpleInheritance {

 public static void main(String args[]) {

 A superOb = new A();

 B subOb = new B();

 // The superclass may be used by itself.

 superOb.i = 10;

 superOb.j = 20;

 System.out.println("Contents of superOb: ");

 superOb.showij();

 System.out.println();

 /* The subclass has access to all public members of its superclass. */

 subOb.i = 7;

 subOb.j = 8;

 subOb.k = 9;

 System.out.println("Contents of subOb: ");

 subOb.showij();

 subOb.showk();

 System.out.println();

 System.out.println("Sum of i, j and k in subOb:");

 subOb.sum();

}

Supper & Subclass

- Only single inheritance is supported with java.
- A superclass can be inherited in more than one java programs.
- Superclass is still an independent working class
- Can't inherit class itself

Member Access

- It can't access those methods/functions which are **private**

```
class Access {  
public static void main(String args[]) {  
    B subOb = new B();  
  
    subOb.setij(10, 12);  
  
    subOb.sum();  
    System.out.println("Total is " + subOb.total);  
}  
}
```

```
class A {  
    int i; // public by default  
    private int j; // private to A  
  
    void setij(int x, int y) {  
        i = x;  
        j = y;  
    }  
  
    // A's j is not accessible here.  
    class B extends A {  
        int total;  
  
        void sum() {  
            total = i + j; // ERROR, j is not accessible here  
        }  
    }  
}
```

```

class Box {
    double width;
    double height;
    double depth;

    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    Box(double len) {
        width = height = depth = len;
    }

    double volume() {
        return width * height * depth;
    }
}

```

```

class BoxWeight extends Box {
    double weight; // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m)
    {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

class DemoBoxWeight {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}

```

OUTPUT

Volume of mybox1 is 3000.0
 Weight of mybox1 is 34.3
 Volume of mybox2 is 24.0
 Weight of mybox2 is 0.076

Advantages

- It is not necessary for BoxWeight to re-create all of the features found in Box. It can simply extend Box to meet its own purposes.
- Once you have created a superclass that defines common set of attributes, these can be used to create more specific subclasses

Extension...

```
// Here, Box is extended to include color.  
class ColorBox extends Box {  
    int color; // color of box  
  
    ColorBox(double w, double h, double d, int c) {  
        width = w;  
        height = h;  
        depth = d;  
        color = c;  
    }
```

Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

Extending previous Example

```
class RefDemo {  
    public static void main(String args[]) {  
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);  
        Box plainbox = new Box();  
        double vol;  
  
        vol = weightbox.volume();  
        System.out.println("Volume of weightbox is " + vol);  
        System.out.println("Weight of weightbox is " + weightbox.weight);  
        System.out.println();  
  
        // assign BoxWeight reference to Box reference  
        plainbox = weightbox;  
  
        vol = plainbox.volume(); // OK, volume() defined in Box  
        System.out.println("Volume of plainbox is " + vol);  
  
        /* The following statement is invalid because plainbox  
         does not define a weight member. */  
        // System.out.println("Weight of plainbox is " + plainbox.weight);  
    }  
}
```

Using *super* [1/4]

- Subclass can call a constructor defined by its superclass by use of the following form of super:

`super(arg-list);`

- improved version of the BoxWeight() class:

```
// BoxWeight now uses super to initialize its Box attributes.  
class BoxWeight extends Box {  
    double weight; // weight of box  
  
    // initialize width, height, and depth using super()  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); // call superclass constructor  
        weight = m;  
    }  
}
```

1 USAGE

Using *super*[2/4]

- In the preceding example, `super()` was called with three arguments. Since constructors can be overloaded
- `super()` can be called using any form defined by the superclass.
- In each case, `super()` is called using the appropriate arguments.

```

class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

```

```

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }

    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }

    // default constructor
    BoxWeight() {
        super();
        weight = -1;
    }

    // constructor used when cube is created
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
}

```

```
class DemoSuper {  
public static void main(String args[]) {  
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);  
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);  
    BoxWeight mybox3 = new BoxWeight(); // default  
    BoxWeight mycube = new BoxWeight(3, 2);  
    BoxWeight myclone = new BoxWeight(mybox1);  
    double vol;  
  
    vol = mybox1.volume();  
    System.out.println("Volume of mybox1 is " + vol);  
    System.out.println("Weight of mybox1 is " +  
mybox1.weight);  
    System.out.println();  
  
    vol = mybox2.volume();  
    System.out.println("Volume of mybox2 is " + vol);  
    System.out.println("Weight of mybox2 is " +  
mybox2.weight);  
    System.out.println();
```

```
    vol = mybox3.volume();  
    System.out.println("Volume of mybox3 is " + vol);  
    System.out.println("Weight of mybox3 is " +  
mybox3.weight);  
    System.out.println();  
  
    vol = myclone.volume();  
    System.out.println("Volume of myclone is " + vol);  
    System.out.println("Weight of myclone is " +  
myclone.weight);  
    System.out.println();  
  
    vol = mycube.volume();  
    System.out.println("Volume of mycube is " + vol);  
    System.out.println("Weight of mycube is " +  
mycube.weight);  
    System.out.println();  
}
```

Using *super*[3/4]

- The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.

super.member

- Here, member can be either a method or an instance variable.

```
// Using super to overcome name hiding.  
class A {  
    int i;  
}  
  
// Create a subclass by extending class A.  
class B extends A {  
    int i; // this i hides the i in A  
  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
  
}  
  
class UseSuper {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2);  
  
        subOb.show();  
    }  
}
```

2 USAGE

OUTPUT

i in superclass: 1
i in subclass: 2

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 26 & 27

Recap

- What is Inheritance
- Keyword *extends*
- Super and subclass
- Member access
- Extension as an advantage

Multilevel Hierarchy

- You can build hierarchies that contain as many layers of inheritance
- It is perfectly acceptable to use a subclass as a superclass of another

A, B, and C

Where,

- **C** can be a subclass of **B**,
- And **B** is a subclass of **A**

```

class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

```

```

// Add weight.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }

    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }

    // default constructor
    BoxWeight() {
        super();
        weight = -1;
    }

    // constructor used when cube is created
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
}

```

```

// Add shipping costs
class Shipment extends BoxWeight {
    double cost;

    // construct clone of an object
    Shipment(Shipment ob) { // pass object to constructor
        super(ob);
        cost = ob.cost;
    }

    // constructor when all parameters are specified
    Shipment(double w, double h, double d,
             double m, double c) {
        super(w, h, d, m); // call superclass constructor
        cost = c;
    }

    // default constructor
    Shipment() {
        super();
        cost = -1;
    }

    // constructor used when cube is created
    Shipment(double len, double m, double c) {
        super(len, m);
        cost = c;
    }
}

```

OUTPUT

Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: \$3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: \$1.28

```

class DemoShipment {
    public static void main(String args[]) {
        Shipment shipment1 =
            new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =
            new Shipment(2, 3, 4, 0.76, 1.28);

        double vol;

        vol = shipment1.volume();
        System.out.println("Volume of shipment1 is " + vol);
        System.out.println("Weight of shipment1 is "
                           + shipment1.weight());
        System.out.println("Shipping cost: $" + shipment1.cost);
        System.out.println();

        vol = shipment2.volume();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "
                           + shipment2.weight());
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}

```

Constructors Call

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?
- For example, given a subclass called B and a superclass called A, is A's constructor called before B's, or vice versa?
- The answer is that in a class hierarchy, constructors are called in order of derivation.

When Constructors are called-Example

```
// Create a super class.  
class A {  
    A() {  
        System.out.println("Inside A's  
constructor.");  
    }  
}
```

```
//Create a subclass by extending class A.  
class B extends A {  
    B() {  
        System.out.println("Inside B's  
constructor.");  
    }  
}
```

```
// Create another subclass by extending B.  
class C extends B {  
    C() {  
        System.out.println("Inside C's  
constructor.");  
    }  
}
```

```
class CallingCons {  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```

OUTPUT

Inside A's constructor
Inside B's constructor
Inside C's constructor

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass that is called method overriding
- So, when an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden

Method Overriding

OUTPUT
k: 3

```
class A {  
    int i, j;  
  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
  
    // display i and j  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    // display k -- this overrides show() in A  
    void show() {  
        System.out.println("k: " + k);  
    }  
}
```

```
class Override {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2, 3);  
  
        subOb.show(); // this calls show() in B  
    }  
}
```

so...

How to call superclass
OVERRIDEN method



Here is the answer...

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        super.show(); // this calls A's show()  
        System.out.println("k: " + k);  
    }  
}
```

OUTPUT
i and j: 1 2
k: 3

Method Overloading

- Method overriding occurs only when the names and the type signatures of the two methods are identical.
- If they are not, then the two methods are simply overloaded.

Method Overloading

```
/* Methods with differing type signatures are
overloaded -- not overridden.*/
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

```
// Create a subclass by extending class A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}
```

```
class OverLoading{
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

OUTPUT
This is k: 3
i and j: 1 2

Using Abstract Classes

- If you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses
- leaving it to each subclass to fill in the details.
- class determines the nature of the methods that the subclasses must implement.
- **abstract** keyword

Using Abstract Classes

- Any class that contains one or more abstract methods must also be declared abstract.

```
abstract class A{ /* Statements */ }
```

- The methods in abstract class must be abstract also

```
abstract type name(parameter-list);
```

Using Abstract Classes

```
// A Simple demonstration of abstract.  
abstract class A {  
    abstract void callme();
```

```
// concrete methods are still allowed in abstract classes  
void callmetoo() {  
    System.out.println("This is a concrete method.");  
}
```

```
class B extends A {  
    void callme() {  
        System.out.println("B's implementation of callme.");  
    }  
}
```

```
class AbstractDemo {  
    public static void main(String args[]) {  
        B b = new B();  
  
        b.callme();  
        b.callmetoo();  
    }  
}
```

OUTPUT:

This is a concrete method.
B's implementation of callme.

Things to remember

- As mentioned, it is not possible to instantiate an abstract class by creating its object
- Class A implements a concrete method called callmetoo(). It is perfectly fine.

3 usage of final

- First: already defined
- Second: it disallow a method from being overrideng.

```
final void meth() { /*Statements*/ }
```

```
void meth() { /*Statements*/ } // Not allowed
```

- Third: Sometimes you also want to prevent your class from being inherited

```
final class A { /* Statements */ }
class B extends A { //... } // Not allowed
```

Chapter Review

- Inheritance basics
- Using super
- Creating Multilevel Hierarchy
- When Constructors are called
- Method overriding
- Using Abstract Classes
- Using Final with inheritance

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 28



Packages and Interfaces

Chapter # 9

Packages

- Directories in our filesystem
- They contains set of related classes
- Mainly used to avoid namespace collisions
- It is both *naming* and a *visibility* control mechanism

Packages

- To create package– include a **package** command as the first statement in the Java source file.
- General form – package pkg;
- Java uses file system directories to store packages.
- Directory name must match with package name.

Packages

- You can create a hierarchy of packages by separating them with a period.

```
package pkg1[ .pkg2 [ .pkg3 ] ]
```

- A package hierarchy must be reflected in the file system of Java development system

```
package java.awt.Image ;
```

- Need to be saved in:

"java \ awt \ image" in a Windows environment.

Example

OUTPUT

K. J. Fielding: \$123.23
Will Tell: \$157.02
-->> Tom Jackson: \$-12.33

```
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}
```

Compilation & Executing

- Call this file AccountBalance.java and put it in a directory called MyPack.
- Next, compile the file, and make sure that the resulting .class file is also in the MyPack directory.
- Then, come out of that directory and try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Importing Packages

- Java includes the import statement to bring classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- In a Java source file, import statements occur *immediately* following the package statement

```
import pkg1[ .pkg2 ] . (classname | * ) ;
```

Importing Packages

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the java package called **java.lang**.

```
import java.lang.*;
```

- it is implicitly imported by the compiler for all programs.

Creating & Importing pkgs

```
package MyPack;

public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
```

```
/*Save this file in another folder*/
import MyPack.*;

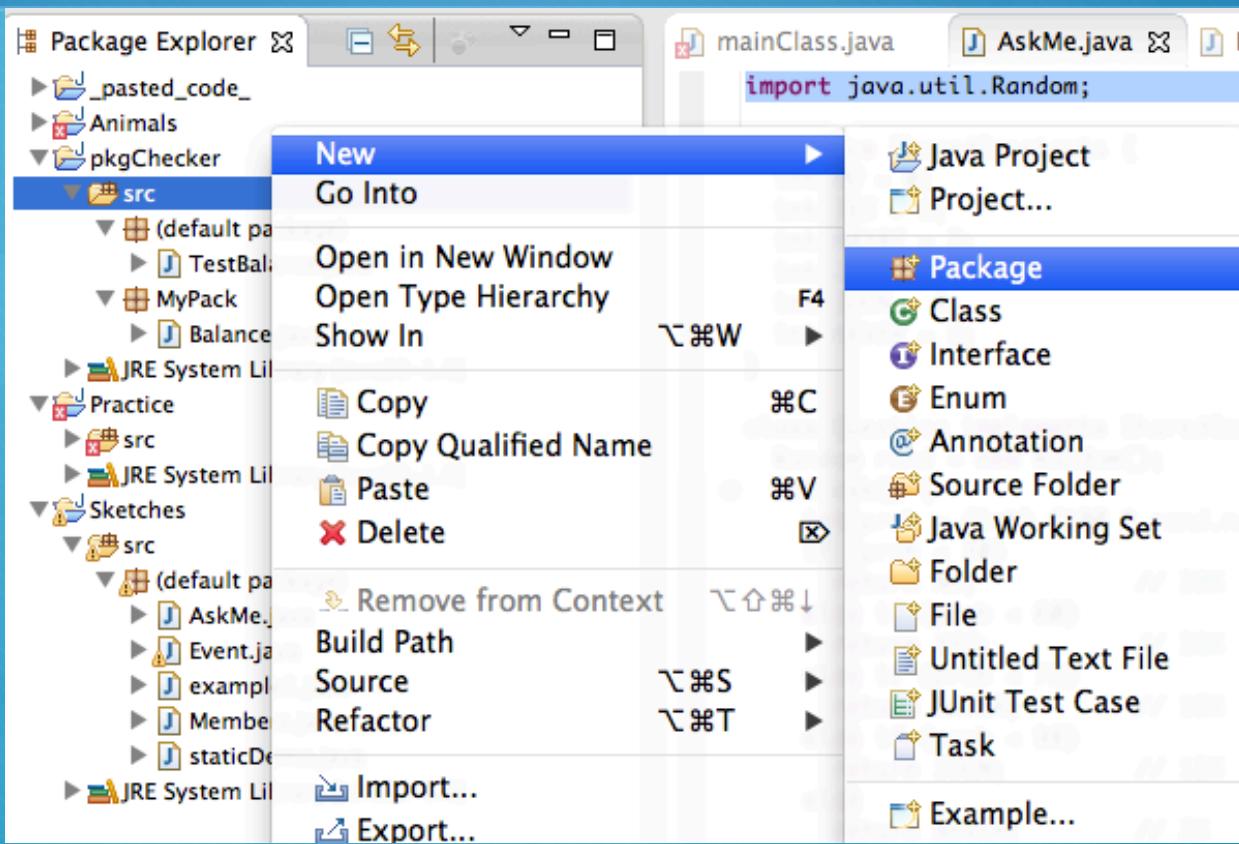
class TestBalance {
    public static void main(String args[]) {

        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show();
    }
}
```

OUTPUT
J. J. Jaspers: \$99.88

Making package in Eclipse



Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 29 and 30



Packages and Interfaces

Chapter # 9

Packages

- Directories in our filesystem
- They contains set of related classes
- Mainly used to avoid namespace collisions
- It is both *naming* and a *visibility* control mechanism

Packages

- To create package– include a **package** command as the first statement in the Java source file.
- General form – package pkg;
- Java uses file system directories to store packages.
- Directory name must match with package name.

Packages

- You can create a hierarchy of packages by separating them with a period.

```
package pkg1[ .pkg2 [ .pkg3 ] ]
```

- A package hierarchy must be reflected in the file system of Java development system

```
package java.awt.Image ;
```

- Need to be saved in:

"java \ awt \ image" in a Windows environment.

Example

OUTPUT

K. J. Fielding: \$123.23
Will Tell: \$157.02
-->> Tom Jackson: \$-12.33

```
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}
```

Compilation & Executing

- Call this file AccountBalance.java and put it in a directory called MyPack.
- Next, compile the file, and make sure that the resulting .class file is also in the MyPack directory.
- Then, come out of that directory and try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Importing Packages

- Java includes the import statement to bring classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- In a Java source file, import statements occur *immediately* following the package statement

```
import pkg1[ .pkg2 ] . (classname | * ) ;
```

Importing Packages

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the java package called **java.lang**.

```
import java.lang.*;
```

- it is implicitly imported by the compiler for all programs.

Creating & Importing pkgs

```
package MyPack;

public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
```

```
/*Save this file in another folder*/
import MyPack.*;

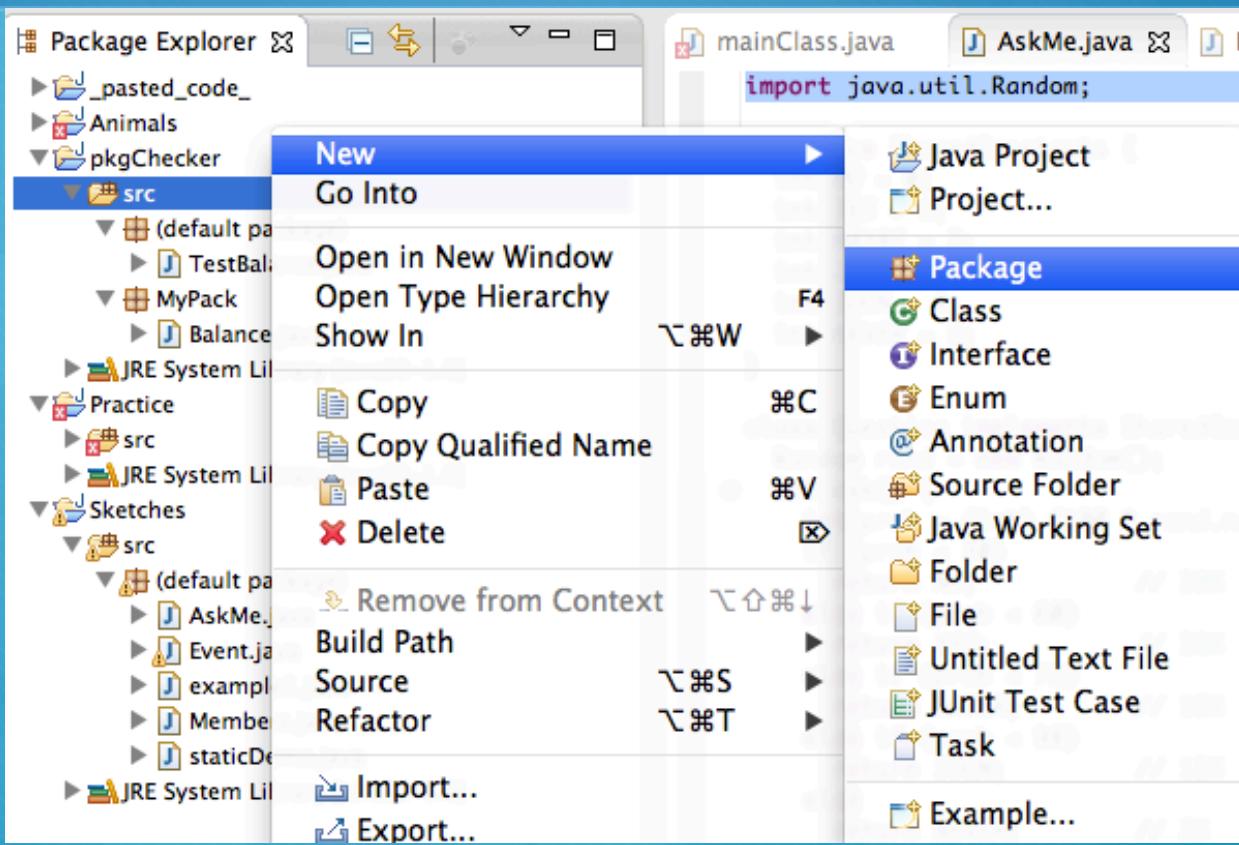
class TestBalance {
    public static void main(String args[]) {

        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show();
    }
}
```

OUTPUT
J. J. Jaspers: \$99.88

Making package in Eclipse



Interfaces

- A Java **interface** specifies a set of methods that any class that **implements** the interface must have.
- An interface is itself a type, which allows you to define methods with parameters of an interface type
- What a class must do, not how it does it.
- They are syntactically similar to classes, but it has:
 - Lack of instance variables, though they may declare constants
 - Methods are declared without body

Interfaces

- An interface is something like the extreme case of an abstract class.
- One class can implement any number of interfaces
- Each class is free to determine the details of its own implementation
- By using interface one can do multiple inheritance in java.

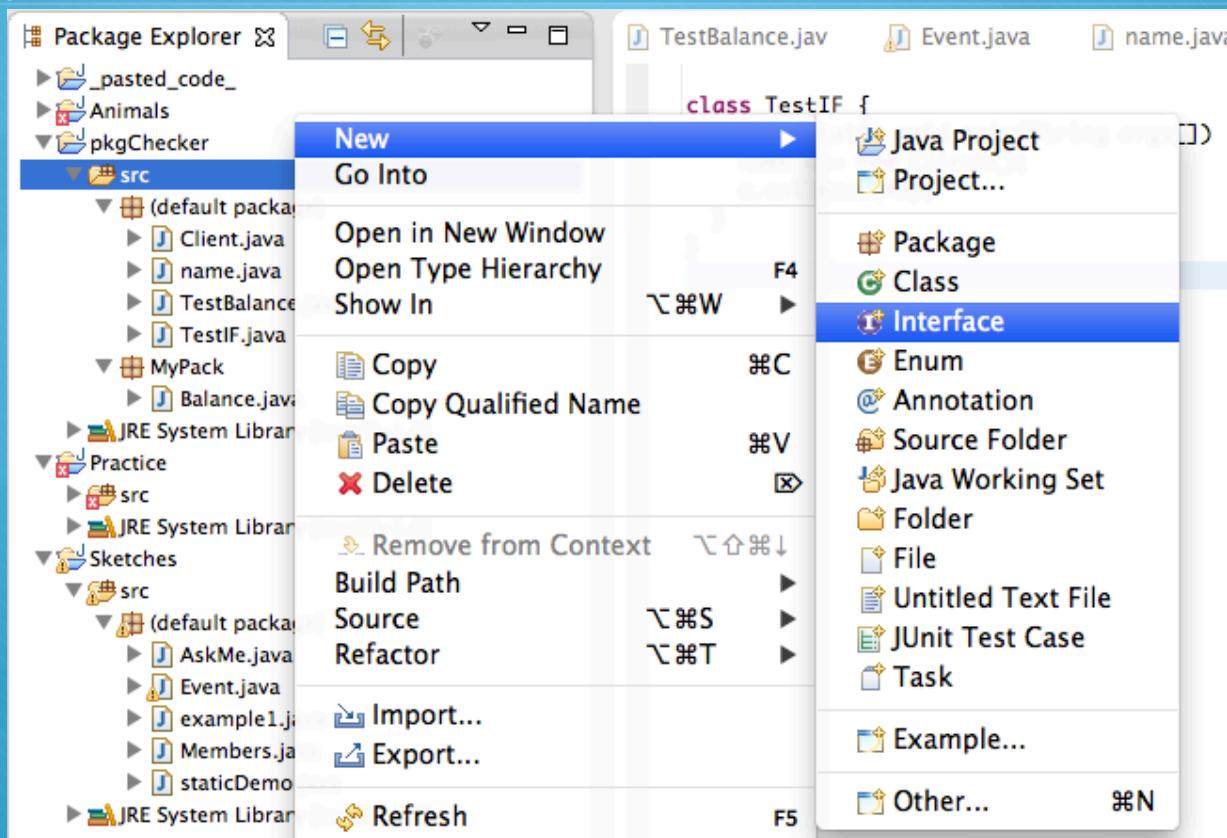
Defining an interface

- An interface is defined much like a class. This is the general form of an interface:
- Keyword : **interface**

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    // ...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

EXAMPLE
interface Callback {
 void callback(int param);
}

Creating Interface in Eclipse



Implementing Interfaces

```
class classname [extends superclass] [implements interface [,interface...]] {  
    // class-body  
}
```

- To implement an interface, a concrete class (that is, a class other than an abstract class) must do two things:
 - It must include the phrase

implements Interface_Name

At the start of the class definition. To implement more than one interface, you list

- All the interface names, separated by commas, as in
 - **implements SomeInterface, AnotherInterface**

Notice that callback() is declared using the public access specifier.

EXAMPLE

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

Permissions

- It is both permissible and common for classes that implement interfaces to define additional members of their own

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
  
    void nonifaceMeth() {  
        System.out.println("Classes may also define other members, too.");  
    }  
}
```

Accessing through references

- The following example calls the `callback()` method via an interface reference variable:

```
class Testiface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
    }  
}
```

NOTE: c could not be used to access nonifaceMeth() since it is defined by Client but not Callback.

Polymorphic Methods

```
// Another implementation of Callback.  
class AnotherClient implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("Another version of callback");  
        System.out.println("p squared is " + (p*p));  
    }  
}  
  
class TestInterface2 {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        AnotherClient ob = new AnotherClient();  
  
        c.callback(42);  
  
        c = ob; // c now refers to AnotherClient object  
        c.callback(42);  
    }  
}
```

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract. For example:

```
abstract class Incomplete implements Callback {  
    int a, b;  
    void show() {  
        System.out.println(a + " " + b);  
    }  
    // ...  
}
```

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;          // 30%
        else if (prob < 60)
            return YES;         // 30%
        else if (prob < 75)
            return LATER;        // 15%
        else if (prob < 98)
            return SOON;        // 13%
        else
            return NEVER;       // 2%
    }
}
```

```
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

Interfaces Can Be Extended

```
// One interface can extend another.  
interface A {  
    void meth1();  
    void meth2();  
}
```

```
interface B extends A {  
    void meth3();  
}
```

```
class IFExtend {  
    public static void main(String arg[]) {  
        MyClass ob = new MyClass();  
  
        ob.meth1();  
        ob.meth2();  
        ob.meth3();  
    }  
}
```

```
// This class must implement all of A and B  
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```

Things to remember

- Use the Keyword **interface** to define an interface.
- If you define an interface with name **myInterface** the java file should be named as **myInterface.java** (Similar to public class definition rules).
- A class implementing an interface should use the keyword **implements**.
- No objects can be created from an interface.
- Interfaces don't have constructors as they can't be initiated
- An Interface can extends one or more interfaces.
- You can define a reference of type **interface** but you should assign to it an object instance of **class type** which implements that interface.

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 31 and 32



Exception Handling

Chapter # 10 Recommended Reading

Contents

- Exception
- Exception Handling
- Errors
- Try, catch and finally blocks
- Things to remember
- Exception propagation
- Throws (Way 1 and 2)
- Re-throw
- Creating own Exceptions
- Checked and Unchecked Exceptions

Exception

- Exception is an abnormal condition or an event that happens during execution of code, which interrupts the normal flow of code.
- Exception can occur for many reasons
 - User has entered invalid data
 - File that need to be opened couldn't found
 - A network connection has been lost in the middle of communication
 - Suddenly JVM has ran out of memory etc...

Example

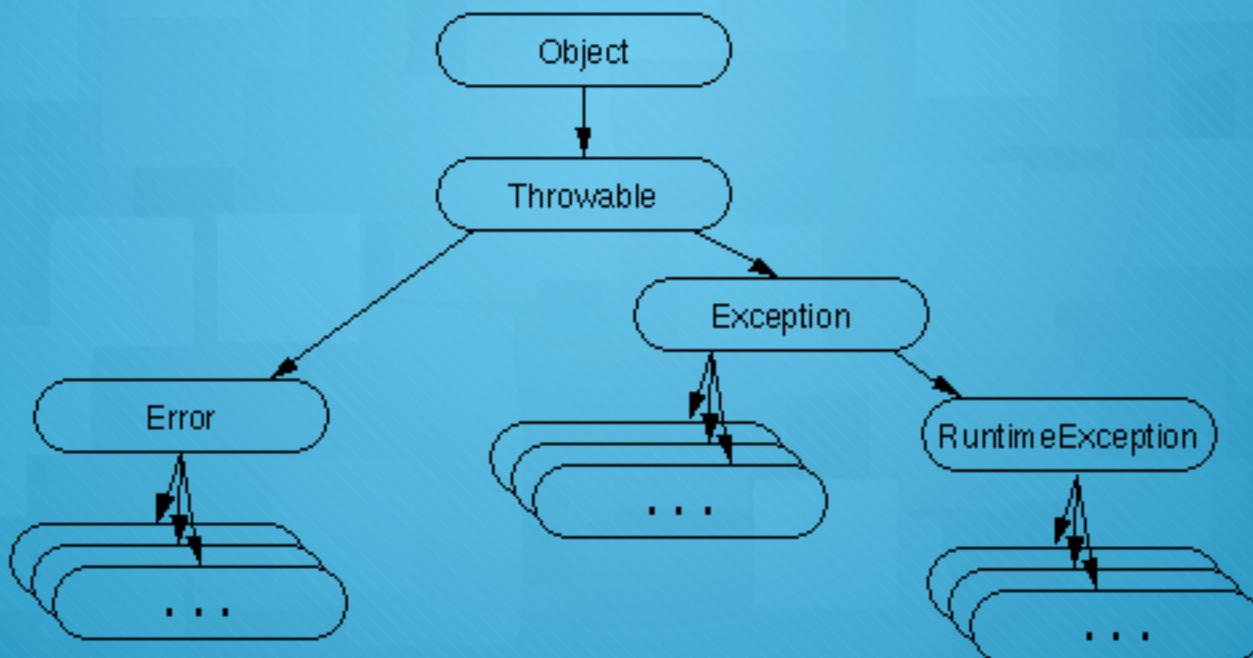
```
double division (double a, double b) {  
    double c = a/b;  
    return c;  
}
```

Java Provides a way to handle such exceptions.
Called it as Exception Handling

Exception Handling

- Java's exception handling enables your Java applications to handle exceptions sensibly.
- When an exception occurs in a Java program it usually results in an exception being thrown.
- Exception occurs only in run time during program execution
- Some keywords:
 - **try, catch, finally, throws and throw**

Exception Hierarchy



Errors

- Errors are irrecoverable situations that occurs during program execution
- Once occurs the application can't be recovered from and come to halt.
- Errors need not to be handled
- Examples:
 - `java.lang.OutOfMemoryError`
 - `java.lang.StackOverFlow`

Example – Error

```
public class Test{  
    public void method1(){  
        this.method2();  
    }  
    public void method2(){  
        this.method1();  
    }  
  
    public static void main (String args[]){  
        Test errorDemo = new Test ();  
        ed.method1();  
    }  
}
```

```
Exception in thread "main" java.lang.StackOverflowError  
at Test.method2(Test.java:6)  
at Test.method1(Test.java:3)  
at Test.method2(Test.java:6)  
at Test.method1(Test.java:3)  
at Test.method2(Test.java:6)  
at Test.method1(Test.java:3)  
at Test.method2(Test.java:6)  
at Test.method1(Test.java:3)  
.  
.  
.  
at Test.method1(Test.java:3)
```

Exceptions (try, catch and finally)

Unlike the errors, exceptions can be handled

```
try{  
    //Some code, might throw an exception  
}  
catch (<ExceptionType> <name>){  
    //Control comes here  
}  
finally{  
    // release some resources  
}
```

try, catch and finally – Example

```
import java.io.*;
public class ExcEg{
    public void FileIOOperation(){
        try{
            FileReader fr = new FileReader("MyFile.txt");
        }
        catch(FileNotFoundException e){
            System.out.println(e.getMessage());
        }
        finally{
            System.out.println("I will execute always");
        }
    }
    public static void main (String args[]){
        ExcEg eeg = new ExcEg();
        eeg.FileIOOperation();
    }
}
```

OUTPUT

MyFile.txt (No such file or directory)
I will execute always

Things to remember [1/2]

- o At least, catch or finally block should be there accompanying with try block, both catch and finally can also be there
- o No statement is allowed in between try, catch and finally.
- o There can be multiple catch blocks
- o One finally block for a try block – and its optional
- o Catch block will execute only when exception is thrown

Things to remember [2/2]

- When Child and Parent relation is there in exceptions, the catch block of parent always comes after child. (Narrow exceptions comes first and broader exceptions in last)
- Finally always runs even when there's no exception.
- In a try block after occurrence of exception, no remaining LOCs will execute.

Questions?

References

- <http://docs.oracle.com/javase/tutorial/essential/exceptions/>



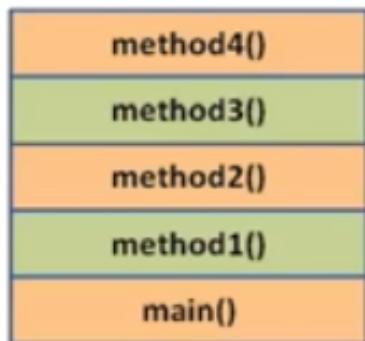
Thanks...

Raheel Ahmed Memon

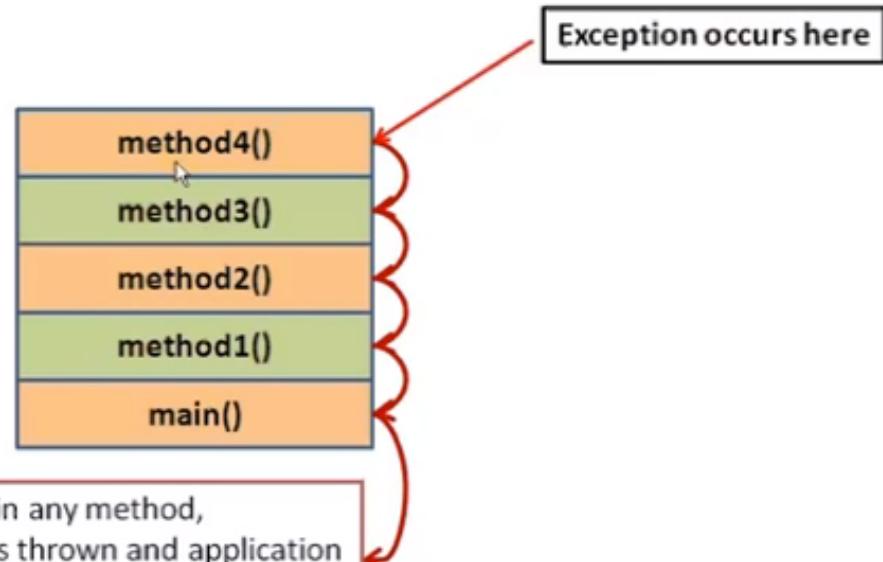
Object Oriented Programming (JAVA)

Lecture 33

Exception Propagation



First IN and Last OUT



If not handled in any method,
the exception is thrown and application
will stop.

Exception Propagation-Example

```
import java.io.*;
public class Test{
    public void method4(){
        int a = 7; int b = 0; int c = a/b;
    }
    public void method3(){
        this.method4();
        System.out.println("After Method 4 completion");
    }
    public void method2(){
        this.method3();
        System.out.println("After Method 3 completion");
    }
    public void method1(){
        this.method2();
        System.out.println("After Method 2 completion");
    }
    public static void main (String args[]){
        Test eHD = new Test();
        eHD.method1();
        System.out.println("After method1 completion");
    }
}
```

Exception in thread "main"
java.lang.ArithmaticException: / by zero
at Test.method4(Test.java:8)
at Test.method3(Test.java:12)
at Test.method2(Test.java:16)
at Test.method1(Test.java:20)
at Test.main(Test.java:26)

Exception Propagation-Example

```
import java.io.*;
public class Test{
    public void method4(){
        int a = 7; int b = 0; int c = a/b;
    }
    public void method3(){
        this.method4();
        System.out.println("After Method 4 completion");
    }
    public void method2(){
        this.method3();
        System.out.println("After Method 3 completion");
    }
    public void method1(){
        try{
            this.method2();
            System.out.println("After Method 2 completion");
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
    public static void main (String args[]){
        Test eHD = new Test();
        eHD.method1();
        System.out.println("After method1 completion");
    }
}
```

/ by zero
After method1 completion

throws

- When an exception is not handled, a method can declare the exception with throws clause.
- When marked as thrown, there are 2 choices to handle it:
 1. Calling method handle the exception
 2. Or declare the throw to the same

throws – Example (choice 1)

```
import java.io.FileNotFoundException;
import java.io.FileReader;
public class ThrowsDemo{
    public void readFile() throws FileNotFoundException{
        FileReader fr = new FileReader("Test");
    }
    public static void main (String args[]){
        ThrowsDemo td = new ThrowsDemo ();
        try{
            td.readFile();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

throws – Example (choice 2)

```
import java.io.FileNotFoundException;
import java.io.FileReader;
public class Test{
    public void readFile() throws FileNotFoundException{
        FileReader fr = new FileReader("Test");
    }
    public void readFile2() throws FileNotFoundException{
        readFile();
    }
    public static void main (String args []){
        Test td = new Test ();
        try{
            td.readFile();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

Re-throw the exceptions

- You can also rethrow the exception that captured in a try and catch block. This can be used for log purpose, or information maintained for the applications status.

```
public void readFile() throws FileNotFoundException{
    try{
        FileReader fr = new FileReader("Test.txt");
    }
    catch (FileNotFoundException e){
        log.info(e.getMessage());
        throw e;
    }
}
```

throws and throw

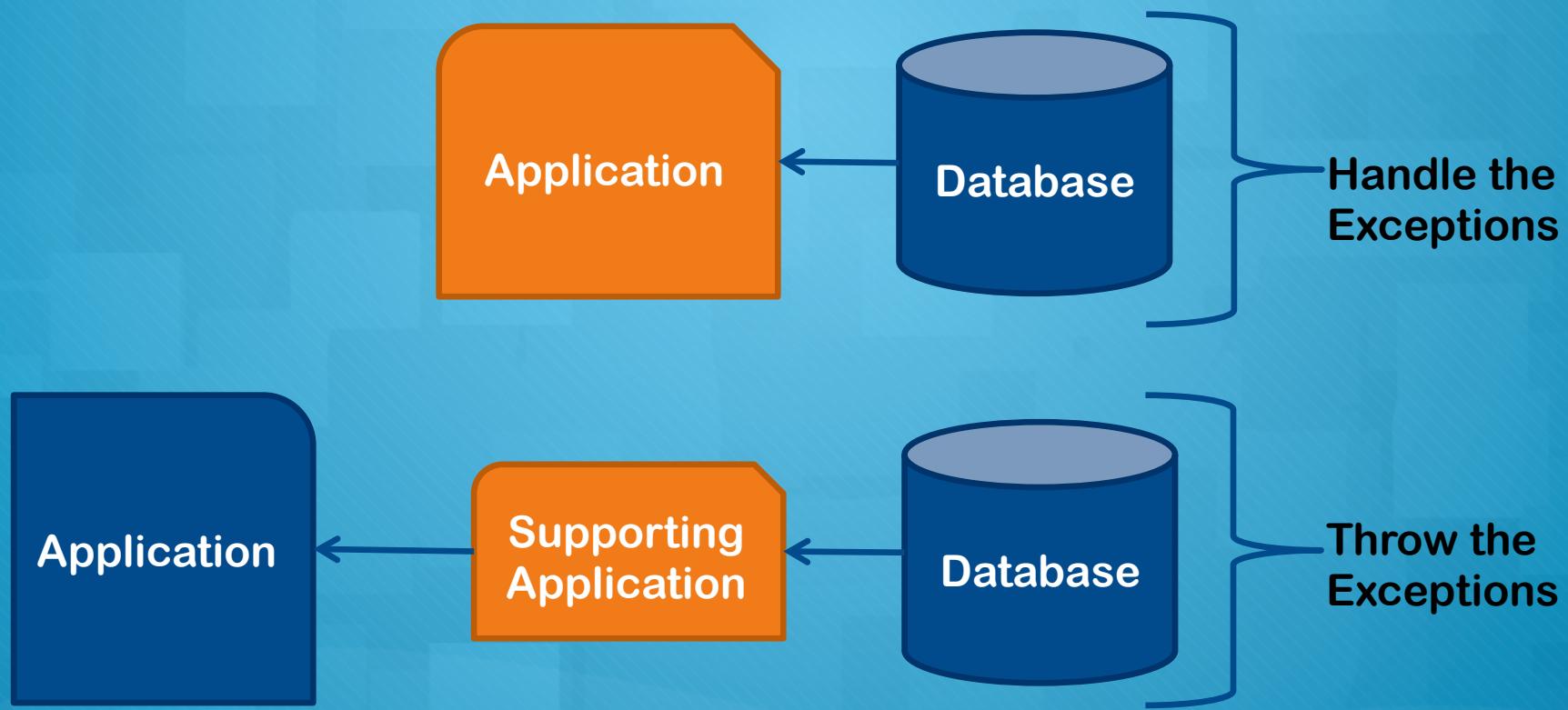
- Syntax for throw:

```
throw ThrowableInstance;
```

- Syntax for throws:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

When to use throwing and when to handle exception?



```
import java.io.IOException;
public class Test {
    // a code after long process
    // returns 0 means success
    // otherwise there's an error

    public void run () throws IOException{

        int code = 0;

        if (code != 0){
            //do something

            throw new IOException("Couldn't
connect to server..");
        }
        System.out.println("Connected Successfully...");

    }

    public static void main (String args[]){
        Test test = new Test ();
        try{
            test.run();
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
    }
}
```

OUTPUT:
Connected Successfully...

```
import java.lang.Exception;
public class Test {
    // a code after long process
    // returns 0 means success
    // otherwise there's an error

    public void run () throws ServerException{
        int code = 1;

        if (code != 0){
            //do something
            throw new ServerException("Couldn't connect
to server.");
        }
        System.out.println("Connected Successfully...");
    }

    public static void main (String args[]){
        Test test = new Test ();
        try{
            test.run();
        }
        catch(ServerException e){
            System.out.println(e.get
e.printStackTrace());
        }
    }
}
```

OUTPUT:
Couldn't Connect to server..

OUTPUT:
Couldn't connect to server..
ServerException: Couldn't connect to
server..
at Test.run(Test.java:16)
at Test.main(Test.java:23)

• • •

Creating Own Exceptions

- What if you couldn't find appropriate Exception for your Program??
Like, we are using IOException for not reaching to server!!!
- **Luckily we can create our own Exceptions**

Creating Own Exceptions and Errors

- A new Exception can be created by extending: **java.lang.Exception** class
-

```
Public MyException extends Exception {  
}
```

```
Public MyError extends Error{  
}
```

Checked & Unchecked Exceptions

- **Checked:**
 - Any Exception that extends `java.lang.Exception`
 - Should be handled by try-catch block or throws clause
- **Unchecked:**
 - Any Exception that extends `java.lang.RuntimeException`
 - Also called as runtime exceptions
 - No need to handle with try-catch or use of throws clause as well

Summary

- Exception
- Exception Handling
- Errors
- Try, catch and finally blocks
- Things to remember
- Exception propagation
- Throws (Way 1 and 2)
- Re-throw
- Creating own Exceptions
- Checked and Unchecked Exceptions

Questions?

References

- <http://docs.oracle.com/javase/tutorial/essential/exceptions/>



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 34 and 35

Introduction to GUI

- Graphical User Interface ("Goo-ee")
 - Pictorial interface to a program
 - Distinctive "look" and "feel"
- GUIs built from components
 - Component: object with which user interacts
 - Examples: Labels, Text fields, Buttons, Checkboxes

Awt & Swing

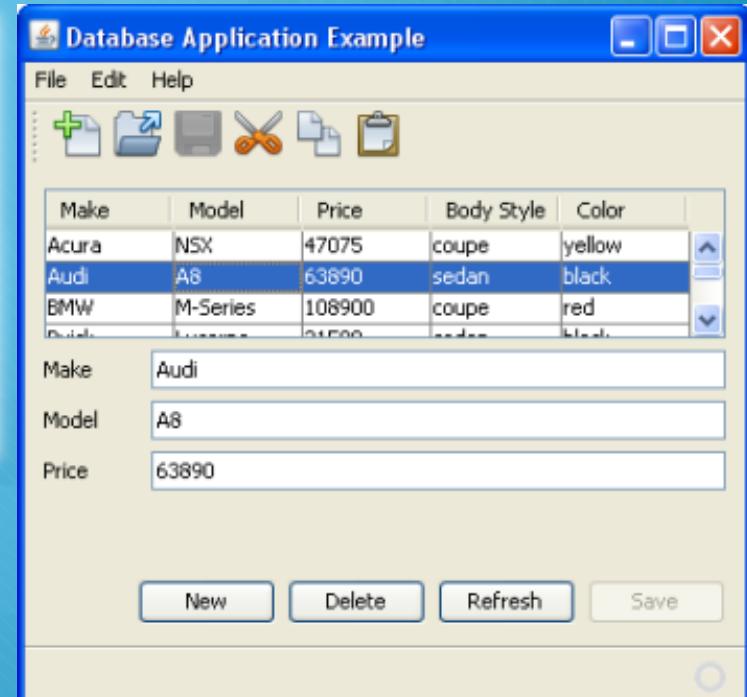
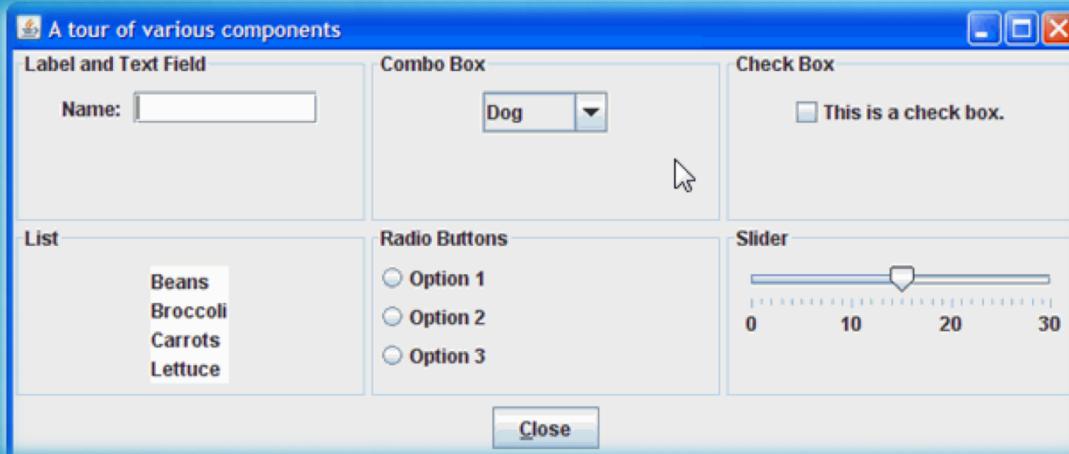
- Sun's initial idea: create a set of classes/methods that can be used to write a multi-platform GUI (Abstract Windowing Toolkit, or AWT)
 - problem: not powerful enough; limited; a bit clunky to use
- Second edition (JDK v1.2): Swing
 - a newer library written from the ground up that allows much more powerful graphics and GUI construction
- **Drawback:** Both exist in Java now; easy to get them mixed up; still have to use both sometimes!

Awt & Swing

```
java.lang.Object
+--java.awt.Component
    +--java.awt.Container
        |
        +--javax.swing.JComponent
            +--javax.swing.JButton
            +--javax.swing.JLabel
            +--javax.swing.JMenuBar
            +--javax.swing.JOptionPane
            +--javax.swing.JPanel
            +--javax.swing.JTextArea
            +--javax.swing.JTextField
        |
        +--java.awt.Window
            +--java.awt.Frame
                +--javax.swing.JFrame
```

```
import java.awt.*;
import javax.swing.*;
```

GUI Applications



Steps for GUI Creation

1. Import required packages, eg: swing or awt

2. Setup the top level container

```
JFrame myFrame = new JFrame();
```

3. Get component area of the top level container

```
Container c = myFrame.getContentPane();
```

- System Area
- Component Area

4. Apply Layout to that Area

```
c.setLayout(new FlowLayout());
```

Steps for GUI Creation

5. Create and add components:

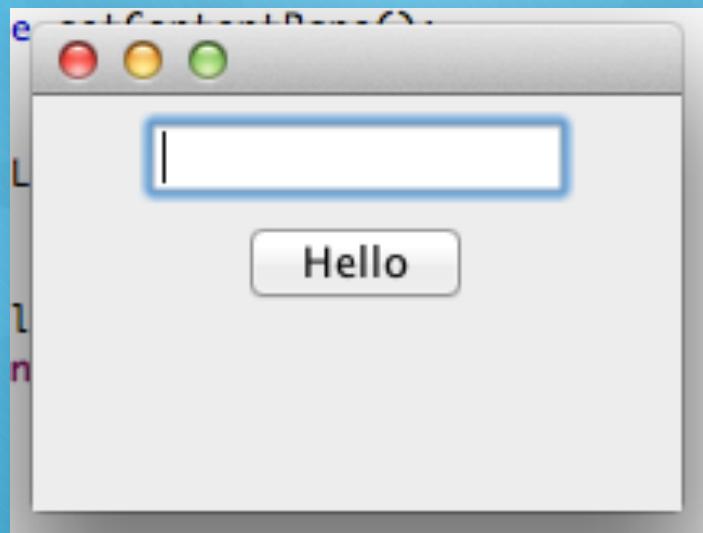
```
JButton b1 = new JButton("Hello");  
c.add(b1);
```

6. Set size of Frame and make it visible

```
myFrame.setSize(200,200);  
myFrame.setVisible(true);
```

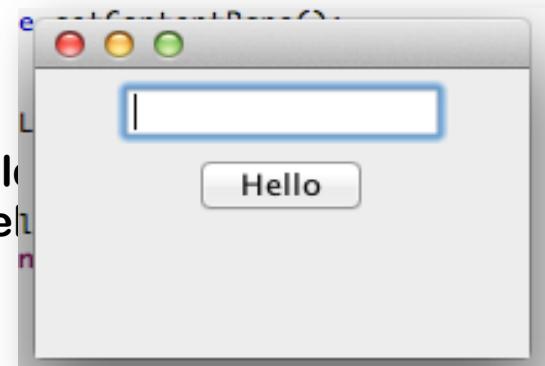
NOTE: When you are using FlowLayout you can't set the size of components

First Java GUI Application



```
//Step1:  
import java.awt.*;  
import javax.swing.*;  
  
class guiTest{  
    JFrame myFrame; // Step 2  
    JTextField myTextField;  
    JButton myButton1;  
  
    void initGUI(){  
        myFrame = new JFrame("Hello");  
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        myFrame.setSize(200,150);  
        myFrame.setVisible(true);  
    }  
}  
  
//Step3:  
Container c = myFrame.getContentPane();  
//Step4:  
c.setLayout(new FlowLayout());  
//Step5:  
JTextField myTextField = new JTextField("Hello");  
JButton myButton1 = new JButton("Hello");  
c.add(myTextField);  
c.add(myButton1);  
  
//Step6:  
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
myFrame.setSize(200,150);  
myFrame.setVisible(true);
```

```
public class GUI{  
    public static void main(String[] args) {  
        guiTest gt = new guiTest();  
        gt.initGUI();  
    }  
}
```



2 different ways you may find

Composition

```
class GUITest{  
    JFrame frame;  
    Container c;  
    public GUITest(){  
        frame = new JFrame();  
        c = frame.getContentPane();  
        ...  
        frame.setVisible(true);  
    }  
    ...  
}
```

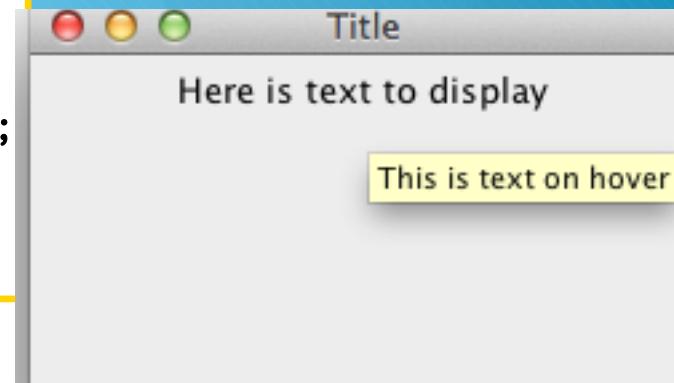
Inheritance

```
class GUITest extends JFrame{  
    Container c;  
    public GUITest(){  
        c = getContentPane();  
        ...  
        setVisible(true);  
    }  
    ...  
}
```

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class GUI extends JFrame{
    private JLabel item1;
    GUI(){
        super("Title");
        setLayout(new FlowLayout());
        item1 = new JLabel("Here is
text to display");
        is text on hover
        item1.setToolTipText("This
import javax.swing.JFrame;
public class GUICTest {
    public static void main(String args[]){
        GUI gt = new GUI();
        gt.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
        gt.setSize(250,150);
        gt.setVisible(true);
    }
}
```

Example 1



Layout Managers

- Java provides many layout managers, common layout managers are:
 - FlowLayout
 - GridLayout
 - BorderLayout

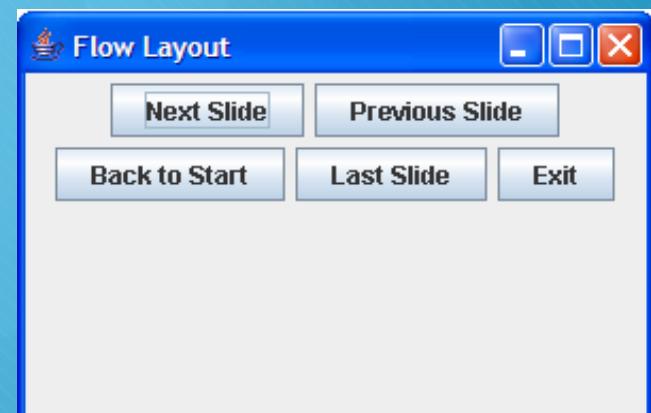
FlowLayout

- With **flow layout**, the components arrange themselves from left to right in the order they were added
- You can't set the size of components here, size is automatic, for eg. Label of a button



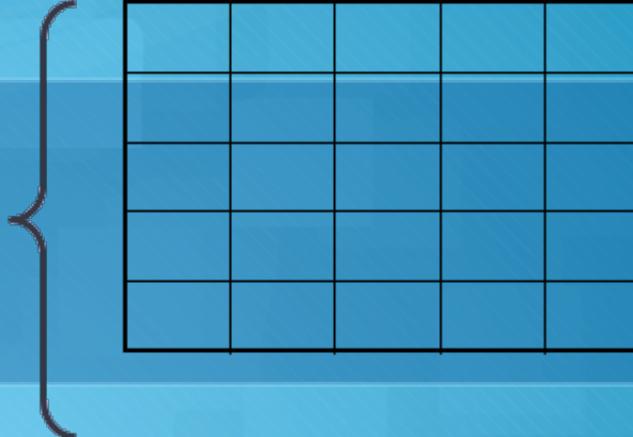
Demo of FlowLayout

```
c.setLayout( new FlowLayout( ) );  
  
b1 = new JButton("Next Slide");  
b2 = new JButton("Previous Slide");  
b3 = new JButton("Back to Start");  
b4 = new JButton("Last Slide");  
b5 = new JButton("Exit");  
  
c.add(b1);  
c.add(b2);  
c.add(b3);  
c.add(b4);  
c.add(b5);
```



GridLayout

rows



- With **grid layout**, the components arrange themselves in a matrix formation (rows, columns)
- First it fills the rows and then columns
- Forces the component to occupy whole size of cell

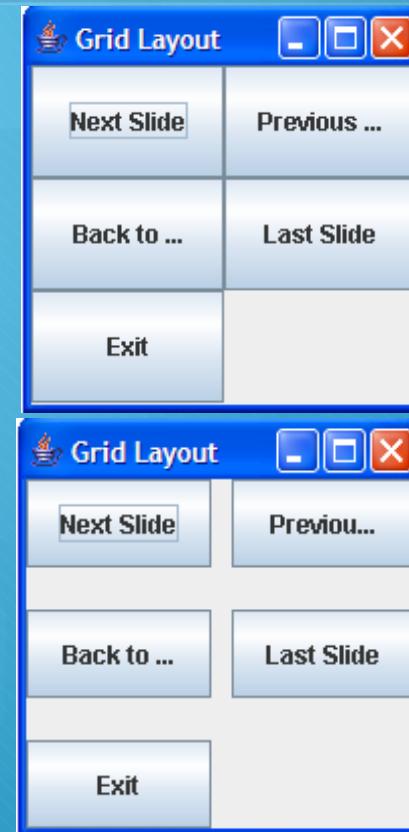


Demo of GridLayout

```
c.setLayout( new GridLayout( 3 , 2 ) );
b1 = new JButton("Next Slide");
b2 = new JButton("Previous Slide");
b3 = new JButton("Back to Start");
b4 = new JButton("Last Slide");
b5 = new JButton("Exit");

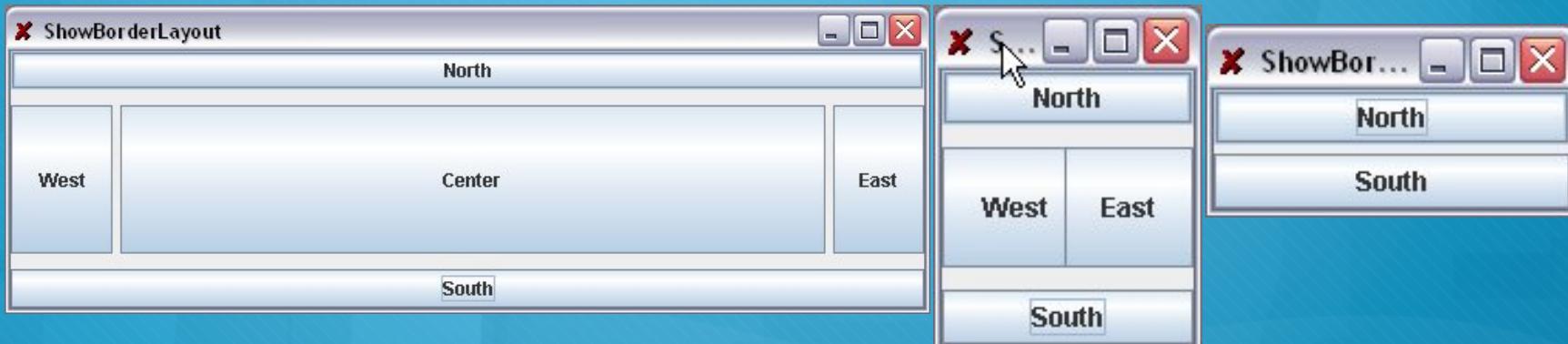
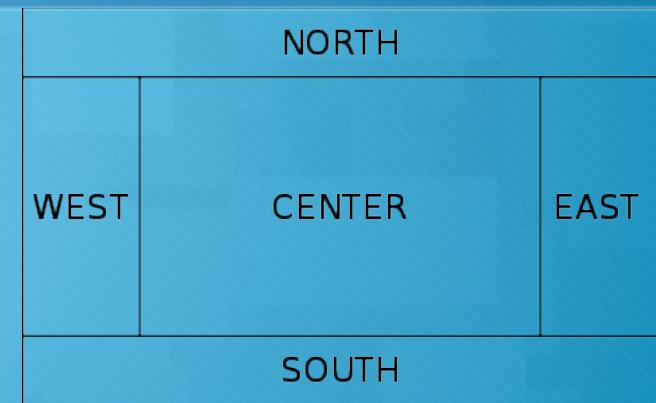
c.add(b1);
c.add(b2);
c.add(b3);
c.add(b4);
c.add(b5);
```

```
//Space Horizontal and Vertical
c.setLayout( new GridLayout( 3 , 2 , 10 , 20 ) );
```



BorderLayout

- o Divide the area into 5 regions
- o Add the component to specific region
- o Forces the size of each component to occupy whole the region



Demo of BorderLayout

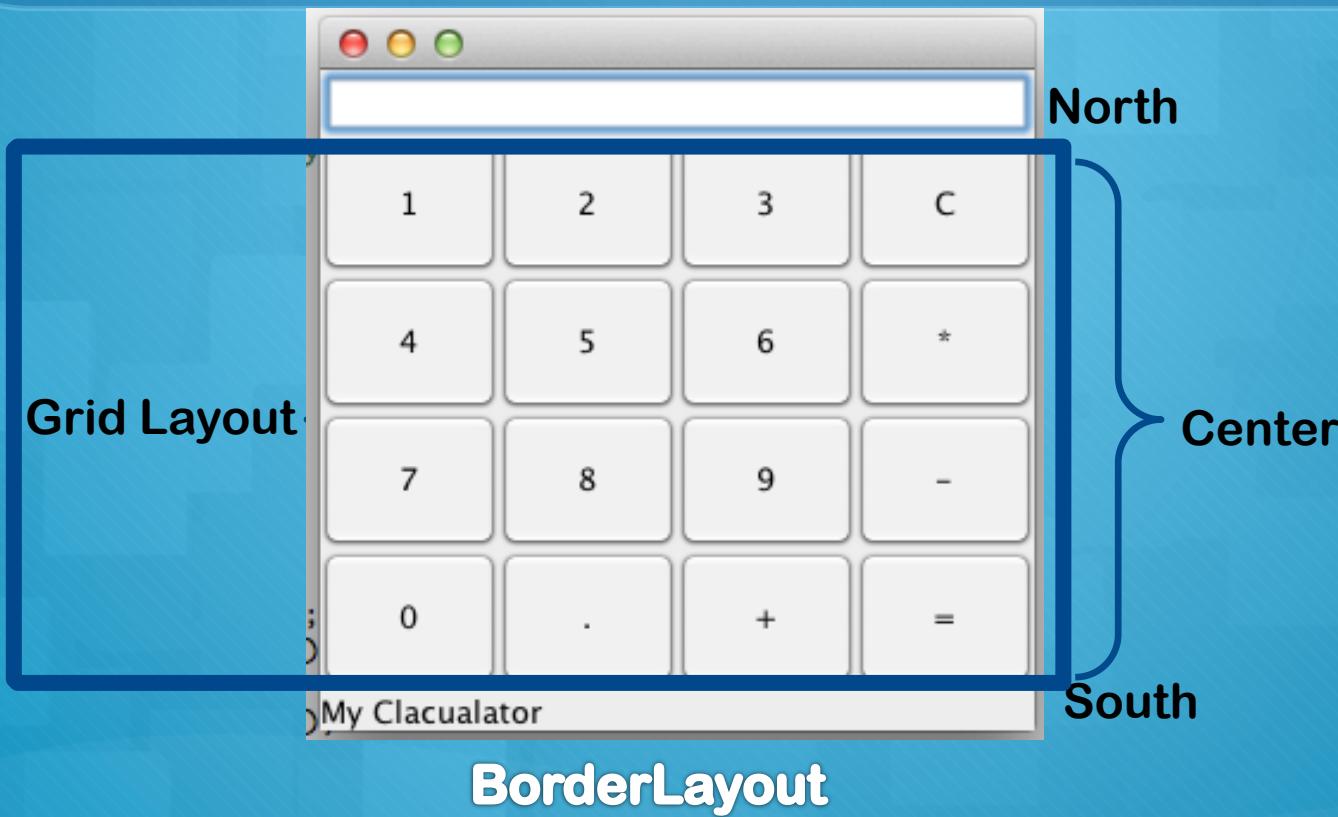
```
c.setLayout( new BorderLayout( );
b1 = new JButton("Next Slide");
b2 = new JButton("Previous Slide");
b3 = new JButton("Back to Start");
b4 = new JButton("Last Slide");
b5 = new JButton("Exit");

c.add( b1 , BorderLayout.NORTH );
c.add( b2 , BorderLayout.SOUTH );
c.add( b3 , BorderLayout.EAST );
c.add( b4 , BorderLayout.WEST );
c.add( b5 , BorderLayout.CENTER);
```



- Container default layout is BorderLayout
- JPanel default layout is FlowLayout
- Speakers in car example

GUI Calculator



```
// File CalculatorGUI.java
import java.awt.*;
import javax.swing.*;
public class CalculatorGUI {
    JFrame fCalc;
    JButton b1, b2, b3, b4, b5, b6, b7, b8, b9, b0;
    JButton bPlus, bMinus, bMul, bPoint, bEqual, bClear;
    JPanel pButtons;
    JTextField tfAnswer;
    JLabel lMyCalc;

    //method used for setting layout of GUI public
    void initGUI () {
        fCalc = new JFrame();
        b0 = new JButton("0"); b1 = new JButton("1");
        b2 = new JButton("2"); b3 = new JButton("3");
        b4 = new JButton("4"); b5 = new JButton("5");
        b6 = new JButton("6"); b7 = new JButton("7");
        b8 = new JButton("8"); b9 = new JButton("9");
        bPlus = new JButton("+"); bMinus = new
        JButton("-");
        bMul = new JButton("*"); bPoint = new JButton(".");
        bEqual = new JButton("="); bClear = new
        JButton("C");
        tfAnswer = new JTextField();
        lMyCalc = new JLabel("My Clacualator");
    }
}
```


Continue
Continue

```
//creating panel object and setting its layout
pButtons = new JPanel (new GridLayout(4,4));
//adding components (buttons) to panel
pButtons.add(b1); pButtons.add(b2);
pButtons.add(b3); pButtons.add(bClear);
pButtons.add(b4); pButtons.add(b5);
pButtons.add(b6); pButtons.add(bMul);
pButtons.add(b7); pButtons.add(b8);
pButtons.add(b9); pButtons.add(bMinus);
pButtons.add(b0); pButtons.add(bPoint);
pButtons.add(bPlus); pButtons.add(bEqual);
// getting component area of JFrame
Container con = fCalc.getContentPane();
con.setLayout(new BorderLayout());

//adding components to container
con.add(tfAnswer, BorderLayout.NORTH);
con.add(IMyCalc, BorderLayout.SOUTH);
con.add(pButtons, BorderLayout.CENTER);
fCalc.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fCalc.setSize(300, 300);
fCalc.setVisible(true);

} // end of GUI Method

public CalculatorGUI () { // default constructor
    initGUI ();
}

public static void main (String args[ ]){
    CalculatorGUI calGUI = new CalculatorGUI ();
}

} // end of class
```

Events

- GUI Generates events when the user interact with interface; clicking button, moving the mouse, closing window etc..
- In java, events are represented by objects
 - ActionEvent (When you click on a button)
 - WindowEvent (When you do something with window like; close, minimize or maximize a window)
 - KeyEvent, MouseEvent, InputEvent...
- So there are several events for every action we perform on a GUI interface

Events

- Both awt and swing provides events

- **java.awt.event.*;**
- **java.swing.event.*;**

```
java.lang.Object
    +--java.util.EventObject
        +--java.awt.awt.event
            +--java.awt.event.ActionEvent
            +--java.awt.event.TextEvent
            +--java.awt.event.ComponentEvent
                +--java.awt.event.FocusEvent
                +--java.awt.event.WindowEvent
                +--java.awt.event.InputEvent
                    +--java.awt.event.KeyEvent
                    +--java.awt.event.MouseEvent
```

Event Handling

- **Event Delegation Model**
 - Processing of an Event is delegated to a particular object (Handler) in program
 - Publish-subscribe model
 - Separate GUI from logic source code

Steps in Event Handling

1. Create a component which can generate events
2. Build class that can handle the events (Event Handler)
3. Register the handler with generators

Step 1

- We have learned about buttons, text fields, window etc..

```
JButton b1 = new JButton ("Hello");
```

- Now, b1 can generate events

Step 2

○ Implement Listener interfaces

- If a class needs to handle an event it needs to implement corresponding listener interface
- ActionListener for ActionEvent, MouseListener for MouseEvent, KeyListener for KeyEvent...

```
public interface ActionListener {  
    public void actionPerformed  
        (ActionEvent e);  
}
```

```
public class Test implements ActionListener{  
    public void actionPerformed(ActionEvent ae) {  
        // do something  
    }  
}
```

CONTRACT:
By implementing an interface a class agrees to implement all of the methods of interface

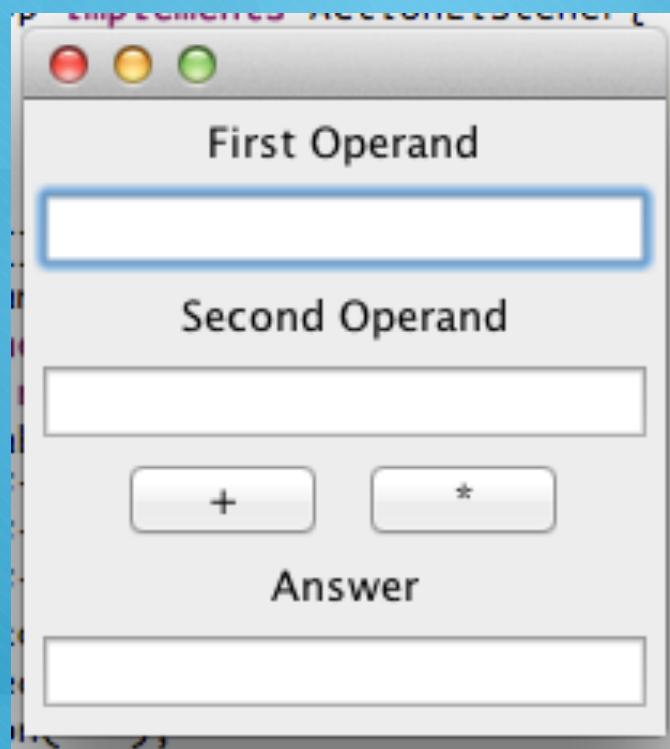
Step 3

- The event generator is told about the object which can handle its events
- Event generator have a method

add _____ Listener(_____);

b1.addActionListner(ObjectOfClass);

GUI Example – Small Calculator



```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class SmallCalcApp implements ActionListener{
// There can be multiple listeners separating by comma.
    JFrame frame;
    JLabel firstOperand, secondOperand, answer; JTextField op1, op2, ans;
    JButton plus, mul;
    // setting layout
    public void initGUI() {
        frame = new JFrame();
        firstOperand = new JLabel("First Operand");
        secondOperand = new JLabel("Second Operand");
        answer = new JLabel("Answer");
        op1 = new JTextField(15);
        op2 = new JTextField(15);
        ans = new JTextField(15);
        plus = new JButton("+");
        plus.setPreferredSize(new Dimension(70,25));
        mul = new JButton("*");
        mul.setPreferredSize(new Dimension(70,25));

        Container cont = frame.getContentPane();
        cont.setLayout(new FlowLayout());
        cont.add(firstOperand); cont.add(op1);
        cont.add(secondOperand); cont.add(op2);
        cont.add(plus); cont.add(mul);
        cont.add(answer); cont.add(ans);

        plus.addActionListener(this);
        mul.addActionListener(this);
```

Step 1. Create Components

Step 3. Registration

Continue

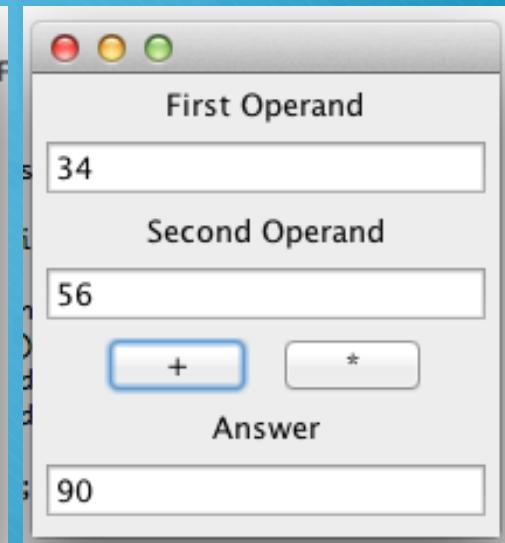
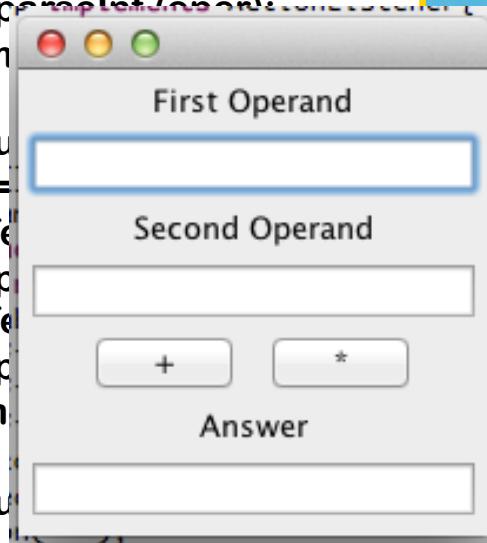
Continue

```

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 220);
        frame.setVisible(true);
    }
    public SmallCalcApp () { // Constructor
        initGUI();
    }
    public void actionPerformed(ActionEvent event){
        String oper, result; int num1, num2, res;
        if (event.getSource() == plus) {
            oper = op1.getText();
            num1 = Integer.parseInt(oper);
            oper = op2.getText();
            num2 = Integer.parseInt(oper);
            res = num1+num2;
            result = res+"";
            ans.setText(result);
        }
        else if (event.getSource() == minus) {
            oper = op1.getText();
            num1 = Integer.parseInt(oper);
            oper = op2.getText();
            num2 = Integer.parseInt(oper);
            res = num1-num2;
            result = res+"";
            ans.setText(result);
        }
        else if (event.getSource() == multiply) {
            oper = op1.getText();
            num1 = Integer.parseInt(oper);
            oper = op2.getText();
            num2 = Integer.parseInt(oper);
            res = num1*num2;
            result = res+"";
            ans.setText(result);
        }
        else if (event.getSource() == divide) {
            oper = op1.getText();
            num1 = Integer.parseInt(oper);
            oper = op2.getText();
            num2 = Integer.parseInt(oper);
            res = num1/num2;
            result = res+"";
            ans.setText(result);
        }
    }
    public static void main(String args[]) {
        SmallCalcApp scApp = new SmallCalcApp();
    }
}// end class

```

**Step 2.
methods of interfaces**



Incomplete implementation of interface?

- You will find interfaces which may have more than one methods, so what to do if you don't need all of those?
- 2 ways:
 1. Keep **empty body** for all methods you don't need
 2. OR add **abstract** as prefix of class name

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 36

Contents

- What is JDBC
- Why JDBC
- How JDBC Works
- Database Basics
- RDBMS
- SQL Basics
- What SQL can do
- Some Useful Commands
- Examples
- Recommendations

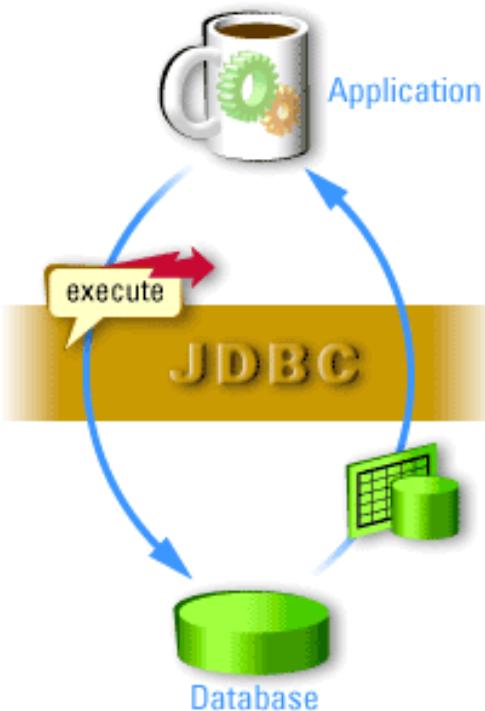
What is JDBC

- JDBC is a Java-based data access technology (Java Standard Edition platform) from Oracle Corporation.
- This technology is an API for the Java programming language that defines how a client may access a database.
- It provides methods for querying and updating data in a database.
- Introduced with JDK 1.1 in 1997
- Related classes are contained in `java.sql` package
- The latest version of JDBC is 1.4, and included in JDK1.7

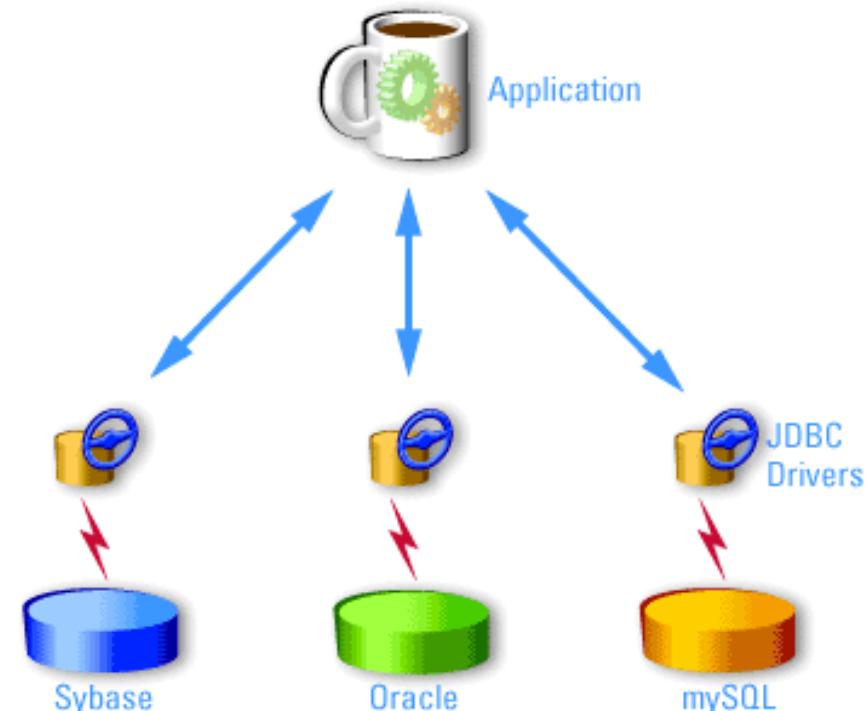
Why JDBC?

- Write Once & Run Anywhere
- Object-related Mapping
 - Objects are developed independently and interact with each other
 - Database optimized for searching indexes
- Network Independent
 - Work across all internet protocols
- Database Independent
 - Java can access any database vendor

How JDBC Works? [1/3]

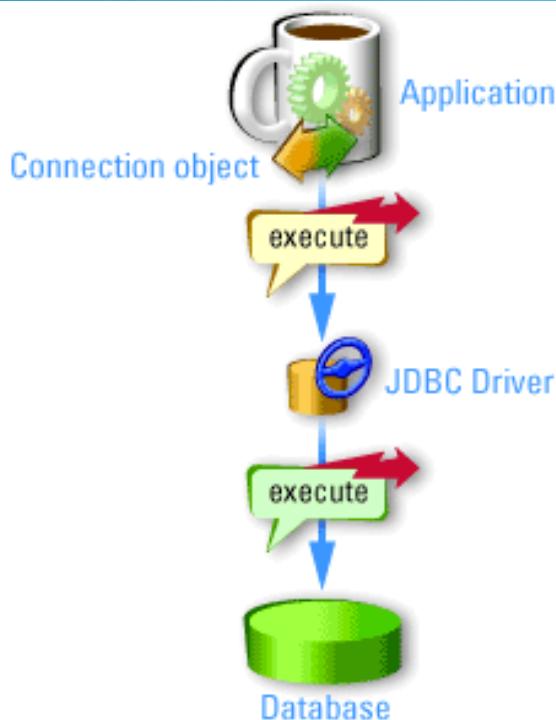


1 of 5
JDBC allows an application to send SQL statements to a database and receive the results.



2 of 5
JDBC interfaces for specific database engines are implemented by a set of classes called JDBC drivers. Since the JDBC driver handles the low-level connection and translation issues, you can focus on the database application development without worrying about the specifics of each database.

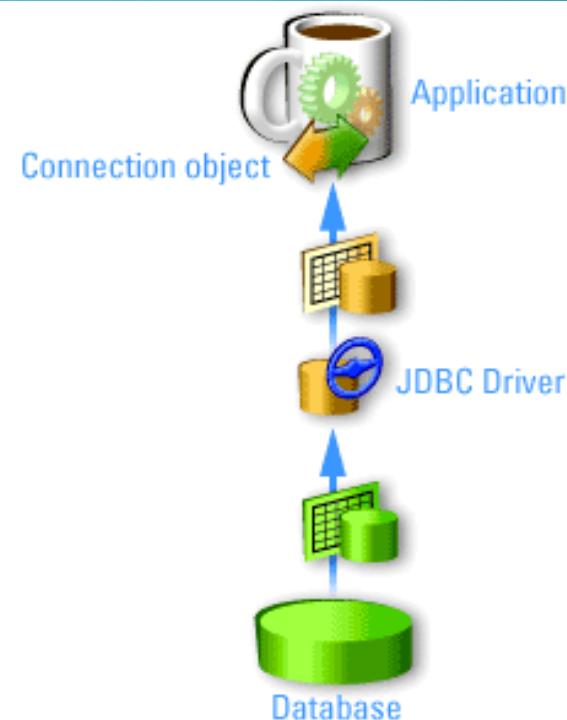
How JDBC Works? [2/3]



3 of 5

The basic sequence of events in a database query is a three-step approach:

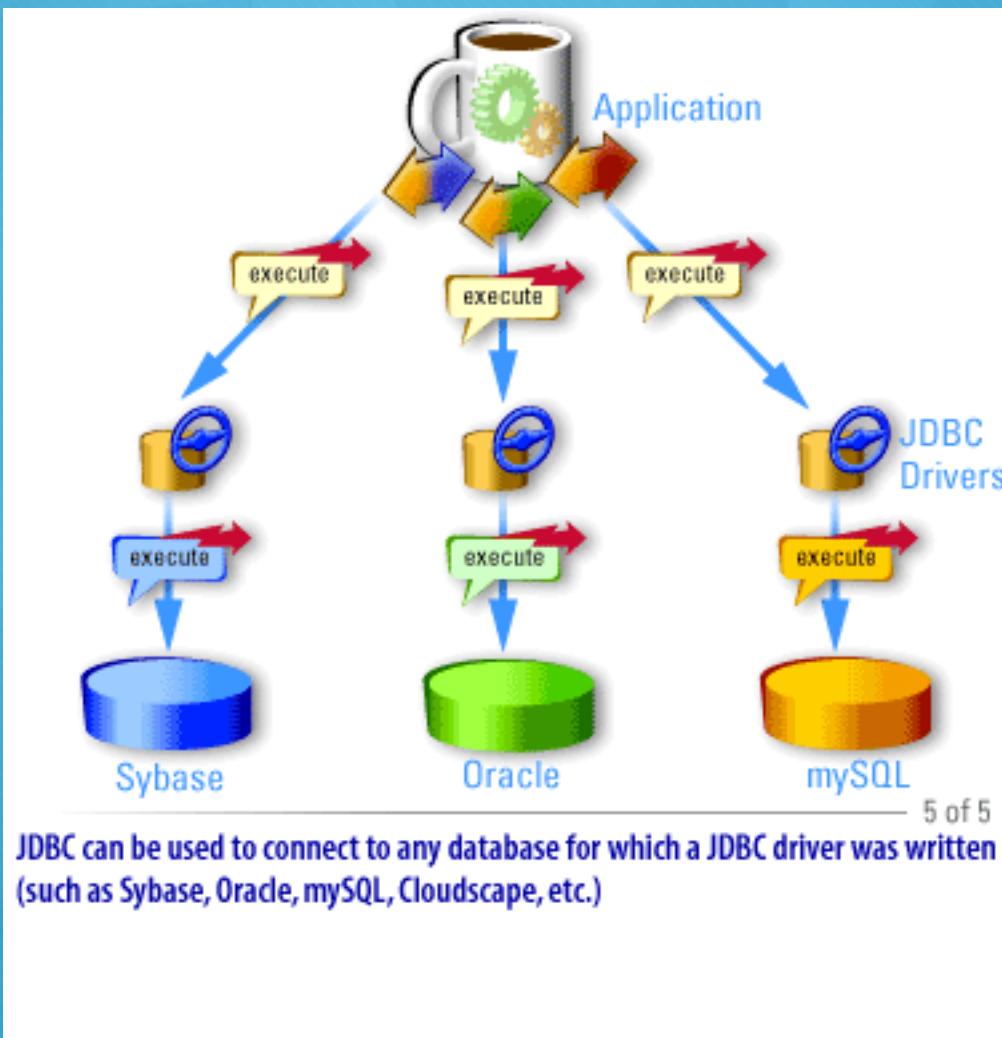
- The client application connects to the database.
- The client application issues an SQL statement (e.g., SELECT).
- The driver translates the JDBC SELECT statement into the database's proprietary format.



4 of 5

The driver translates the `ResultSet` returned by the database, which is then returned to the application.

How JDBC Works? [3/3]



Database Basics

- Relational Database Concepts
 - Relational Table
 - Record/row Field/column.
 - Relationship
 - Keys
 - Primary
 - Foreign

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables.
- A table is a collection of related data entries and it consists of columns and rows.

SQL Basics

- Let you access and manipulate with databases
- SQL is an ANSI (American National Standards Institute) standard
-

What SQL can do?

- **SQL can execute queries against a database**
- **SQL can retrieve data from a database**
- **SQL can insert records in a database**
- **SQL can update records in a database**
- **SQL can delete records from a database**
- **SQL can create new databases**
- **SQL can create new tables in a database**
- **SQL can create stored procedures in a database**
- **SQL can create views in a database**
- **SQL can set permissions on tables, procedures, and views**

Some Important Commands of SQL [1/2]

- **SELECT** - extracts data from a database
- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database
- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database

Some Important Commands of SQL [2/2]

- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

SELECT Statement

```
SELECT column_name,column_name  
FROM table_name;
```

OR

```
SELECT * FROM table_name;
```

CustomerName	City
Alfreds Futterkiste	Berlin
Ana Trujillo Emparedados y helados	México D.F.
Antonio Moreno Taquería	México D.F.
Around the Horn	London
Berglunds snabbköp	Luleå

SELECT CustomerName,City FROM Customers;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

SELECT * FROM Customers;

WHERE Clause

SELECT column_name,column_name

FROM table_name

WHERE column_name operator value;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
13	Centro comercial Moctezuma	Francisco Chang	Sierras de Granada 9993	México D.F.	05022	Mexico
58	Pericles Comidas clásicas	Guillermo Fernández	Calle Dr. Jorge Cash 321	México D.F.	05033	Mexico
80	Tortuga Restaurante	Miguel Angel Paolino	Avda. Azteca 123	México D.F.	05033	Mexico

**SELECT * FROM Customers
WHERE Country='Mexico';**

UPDATE Statement

UPDATE table_name

SET column1=value1,column2=value2,...

WHERE some_column=some_value;



```
UPDATE Customers
SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName='Alfreds Futterkiste';
```

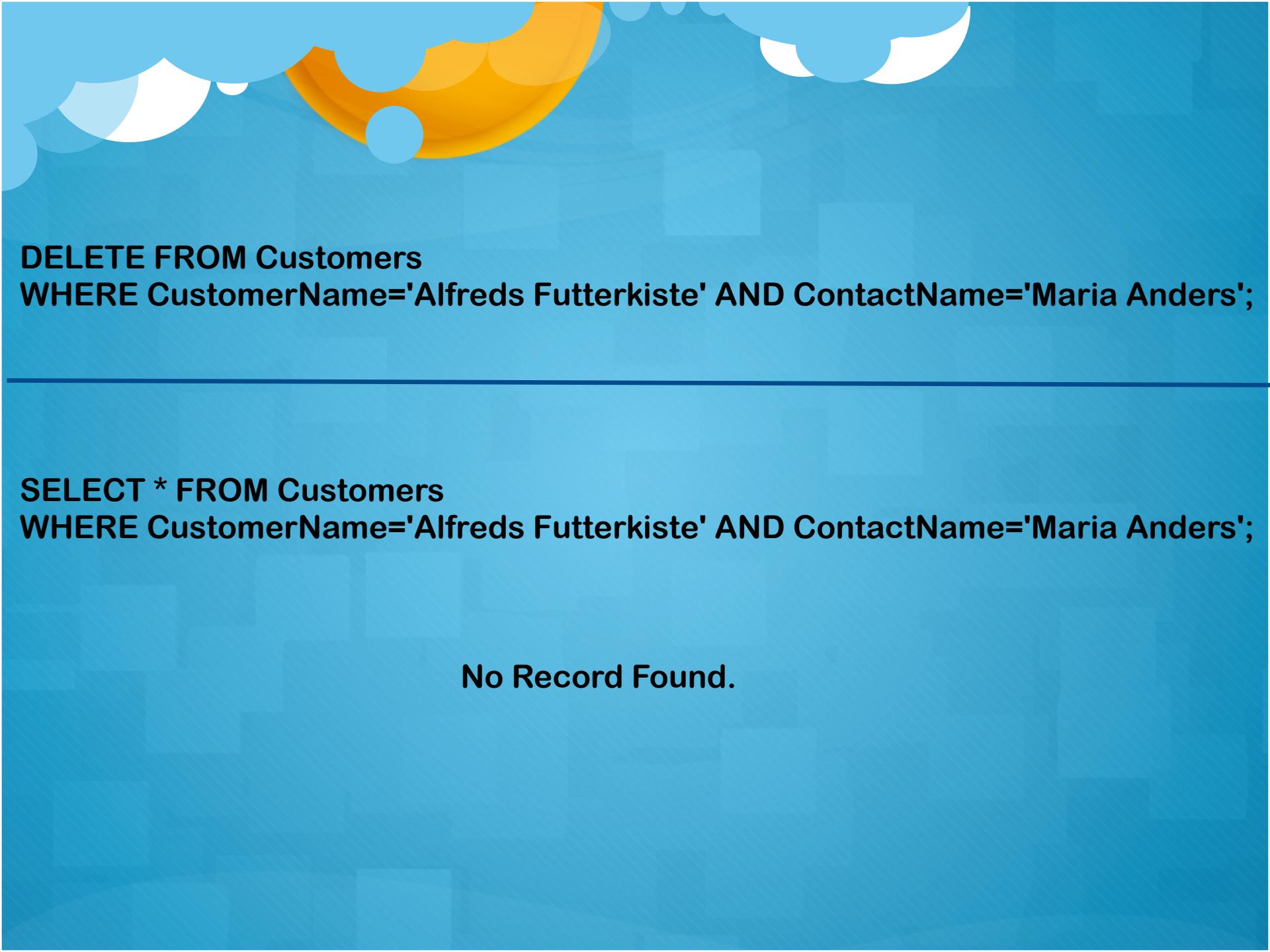
```
SELECT * FROM customers
WHERE CustomerName='Alfreds Futterkiste';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Hamburg	12209	Germany

DELETE Statement

DELETE FROM table_name

WHERE some_column=some_value;



```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';
```

```
SELECT * FROM Customers
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';
```

No Record Found.

Recommendation

- 1. It is recommended that you must visit following website and take the tutorial provided for SQL Commands before moving to JDBC:**
<http://www.w3schools.com/>

- 2. Must know how to create a database using Microsoft Access**
<http://office.microsoft.com/>

Questions?



Thanks...

Raheel Ahmed Memon

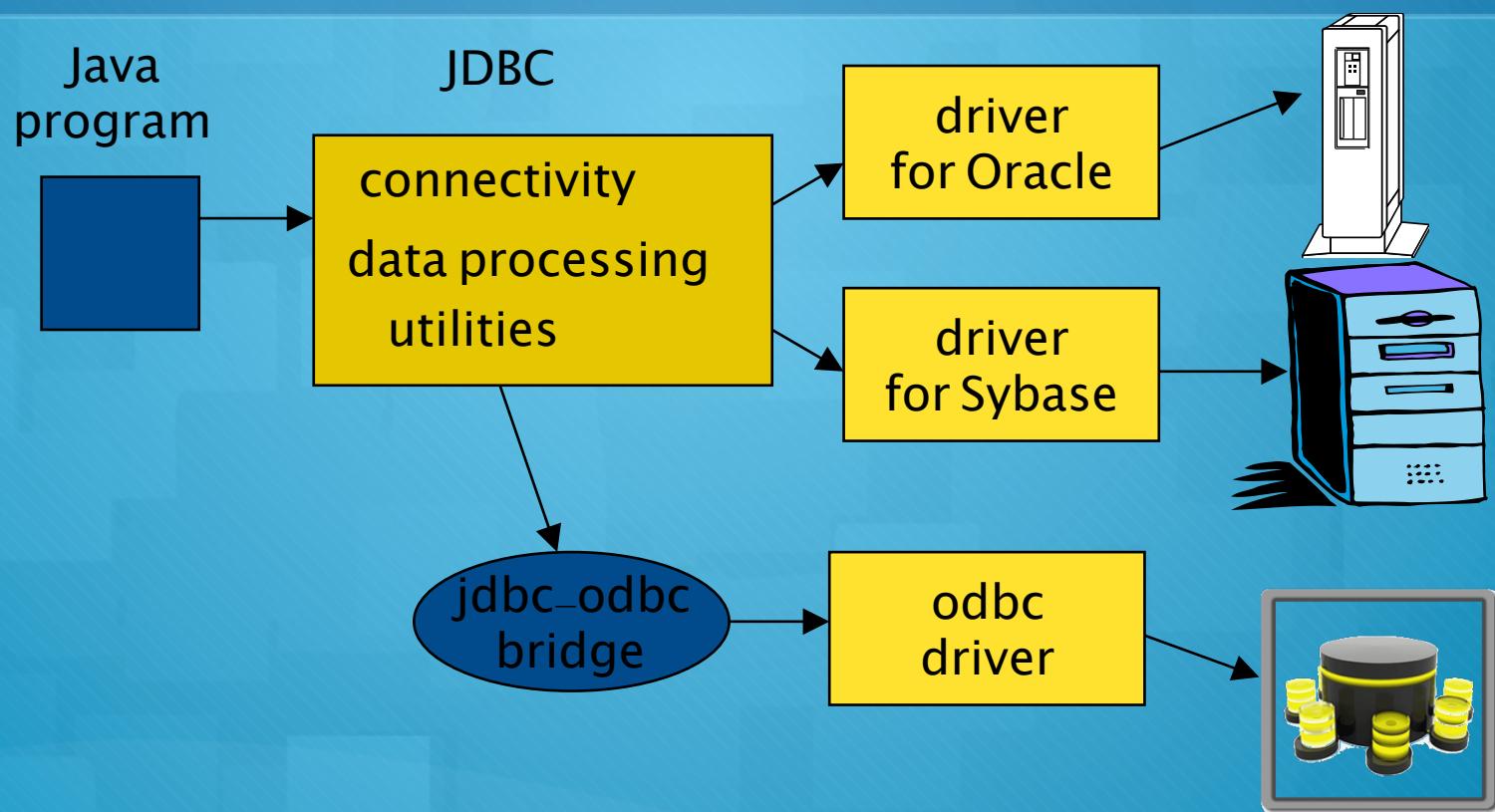
Object Oriented Programming (JAVA)

Lecture 37 and 38

JDBC/ODBC

- Open Database Connectivity as a standard software API for connecting to DBMS.
- Java Database Connectivity is a API which provides classes and methods to interact with all kind of databases
- JDBC is an interface which allows Java code to execute SQL statements inside relational databases
 - the databases must follow the ANSI SQL-2 standard

JDBC in use



The JDBC-ODBC Bridge

- ODBC (Open Database Connectivity) is a Microsoft standard from the mid 1990's.
- It is an API that allows C/C++ programs to execute SQL inside databases
- ODBC is supported by many products.
- The JDBC-ODBC bridge allows Java code to use the C/C++ interface of ODBC
- it means that JDBC can access many different database products
- The layers of translation (Java --> C --> SQL) can slow down execution.

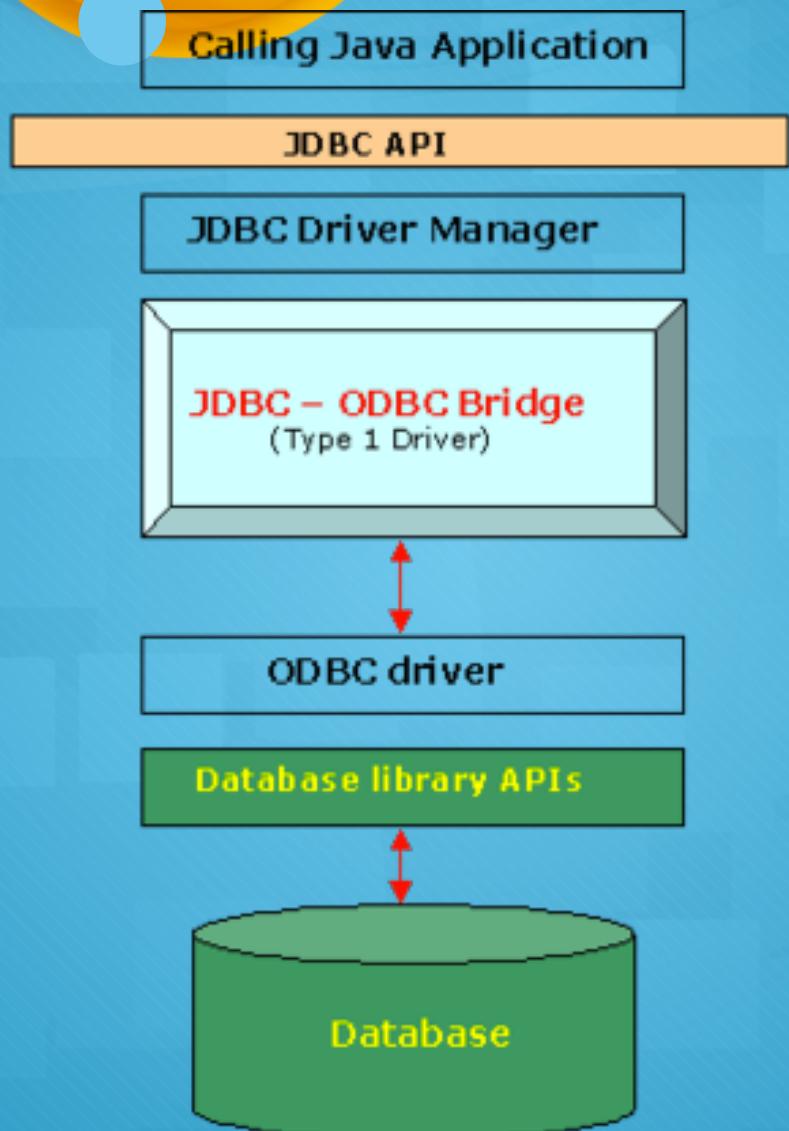
The JDBC-ODBC Bridge

- The JDBC-ODBC bridge comes free with the JDK:
 - called sun.jdbc.odbc.JdbcOdbcDriver
- The ODBC driver for Microsoft Access comes with MS Office
- so it is easy to connect Java and Access

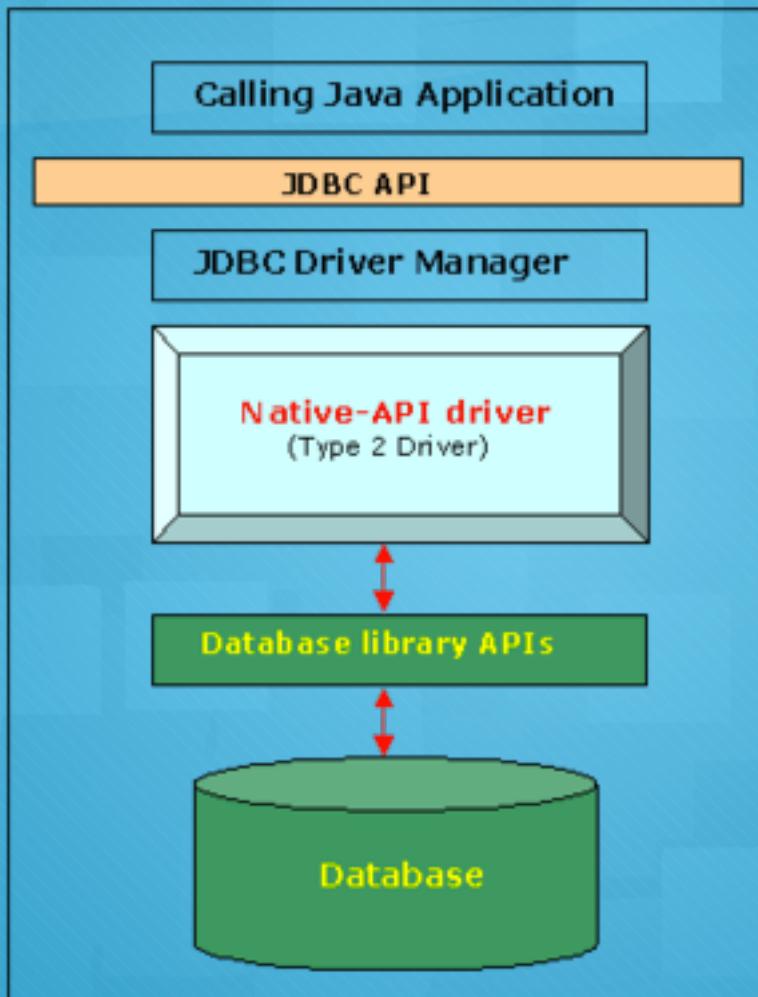
Four Kinds of JDBC Driver

- 1. JDBC-ODBC Bridge:** translate Java to the ODBC API
- 2. Native API:** translate Java to the database's own API
- 3. Native Protocol:** use Java to access the database more directly using its low level protocols
- 4. Net Protocol:** use Java to access the database via networking middleware (usually TCP/IP) required for networked applications

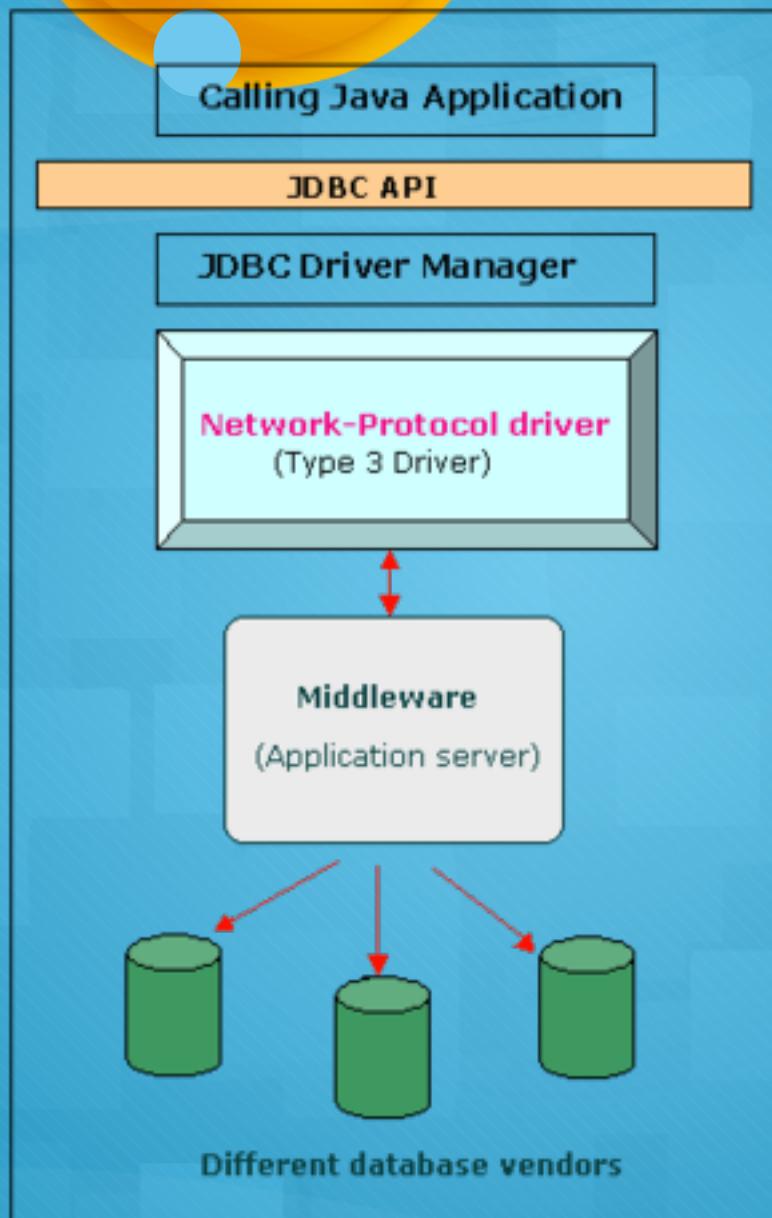
JDBC-ODBC Drivers



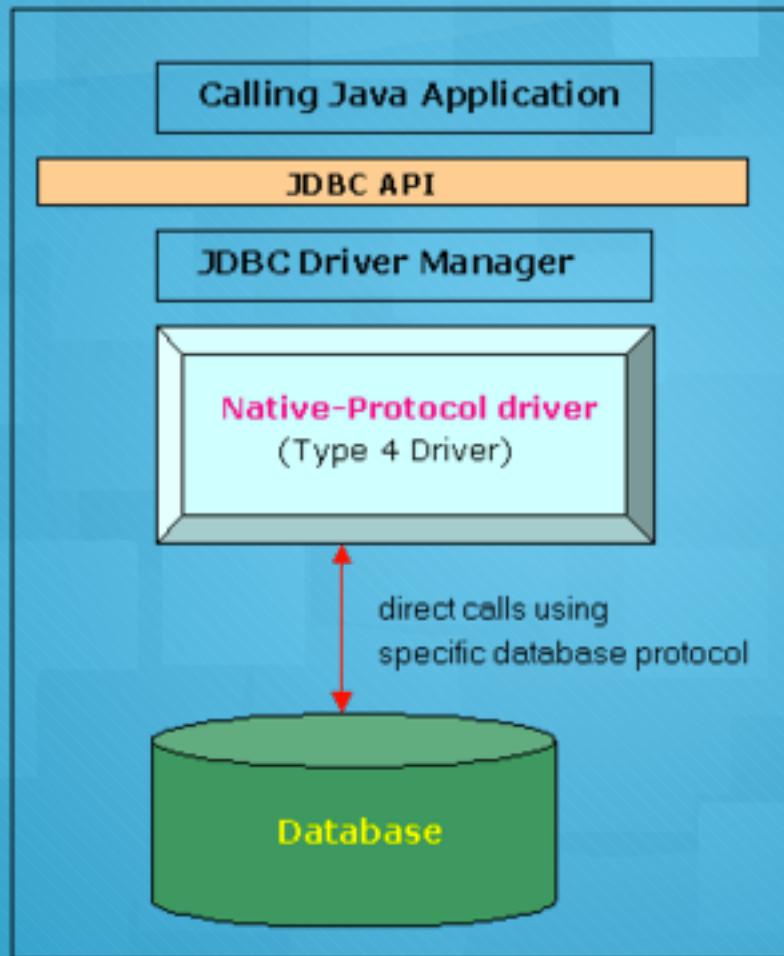
Native-API Driver



Network-Protocol Driver



Native-Protocol Driver



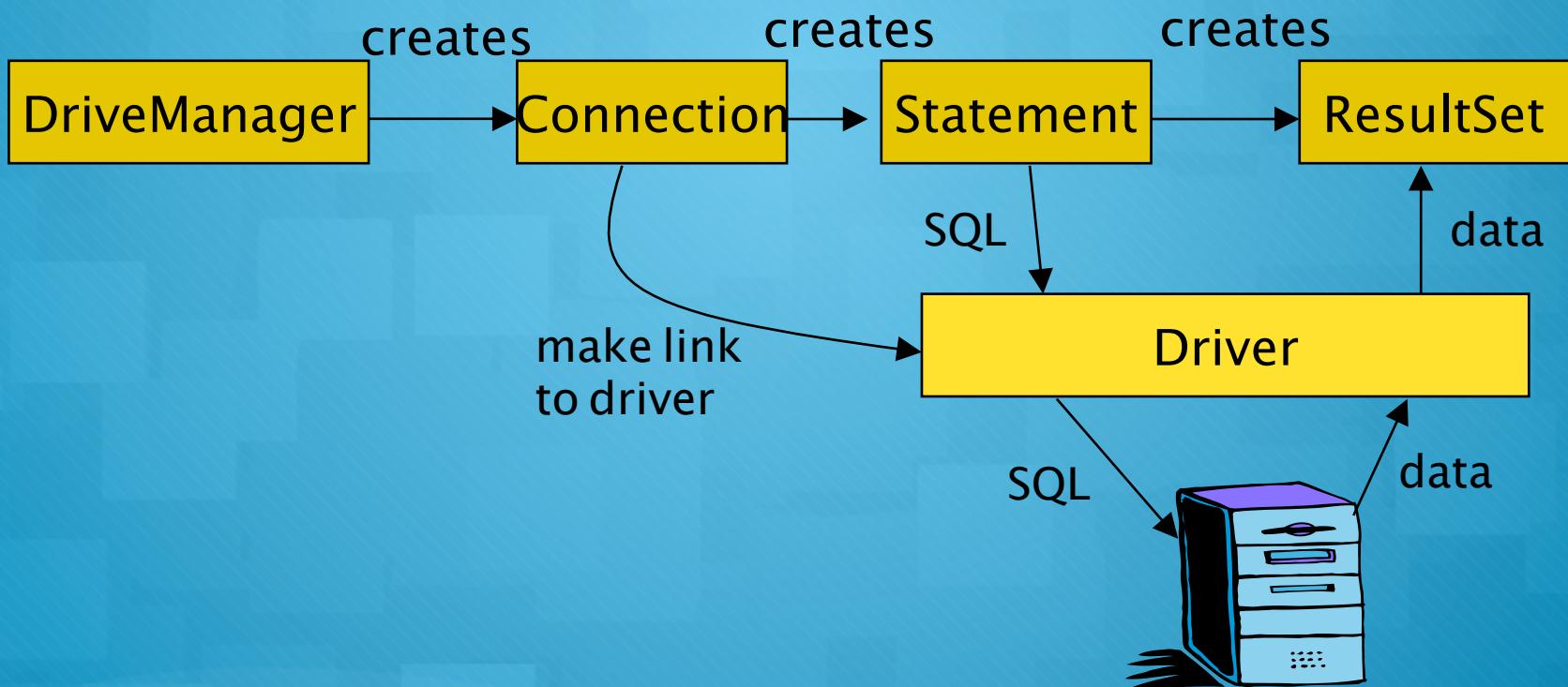
Process of Using JDBC

1. Load JDBC Driver
2. Connection to database
 - Talk to the database (Name, User Name & Password)
3. Create SQL Statement
4. Execute the statement
5. Process the Results
6. Close the Connection

NOTE: Surround these statements with *try* and *catch* blocks, many of these statements require exception handling.

Exceptions:
ClassNotFoundException
SQLException

Process of Using JDBC



JDBC Object class

- **Driver Manager**
 - Load chosen drivers
- **Driver**
 - Connect to actual database
- **Connection**
 - A series of SQL statements to and from the DB
- **Statement**
 - A Single SQL Statement
- **ResultSet**
 - The records returned from a statement

Database Connection

- o Load/register JDBC-ODBC driver
- o Class.forName(sun.jdbc.odbc.jdbcodbcdriver)
- o Create a Connection (Object)

```
Connection con = DriverManager.getConnection(dsn,  
id, password)
```

Dsn: “ jdbc.odbc.nameOfDatabase”

ID: ODBC login id

Password: ODBC login Password

Preparing & Executing SQL Statements

- Use the connection object to create statement object
- Prepare a SQL query string
- Execute the Statement
 - ReadSet is read only object, used to read a complete record
 - Use next() method for moving the next rows

```
Statement stmt = con.createStatement();
```

```
String query = "SELECT * FROM product";
```

```
ResultSet rSet = stmt.executeQuery(query);
```

Getting data from ResultSet

- Get row number (rows starts from 1)

```
Int rowNum=rSet.getRow();
```

- Get field values using getter methods

- Getters can work for only once for a field/column.

```
rSet.getString(String columnName) OR
```

```
rSet.getString(int columnNumber)
```

Last Step (Never Forget)

- Close the connection

```
con.close();
```

Example

```
import java.sql.*;

class JDBCQuery
{
public static void main( String args[] )
{
try
{
// Load the database driver
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );

// Get a connection to the database
Connection conn = DriverManager.getConnection( "jdbc:odbc:Database" );

// Print all warnings
for( SQLWarning warn = conn.getWarnings(); warn != null; warn = warn.getNextWarning() )
{
System.out.println( "SQL Warning:" );
System.out.println( "State : " + warn.getSQLState() );
System.out.println( "Message: " + warn.getMessage() );
System.out.println( "Error : " + warn.getErrorCode() );
}

// Get a statement from the connection
Statement stmt = conn.createStatement();
```

```
// Execute the query
ResultSet rs = stmt.executeQuery( "SELECT * FROM Cust" );

// Loop through the result set
while( rs.next() )
    System.out.println( rs.getString(1) );
// Close the result set, statement and the connection
rs.close();
stmt.close();
conn.close();
}
catch( SQLException se )
{
    System.out.println( "SQL Exception:" );
// Loop through the SQL Exceptions
while( se != null )
{
    System.out.println( "State : " + se.getSQLState() );
    System.out.println( "Message: " + se.getMessage() );
    System.out.println( "Error : " + se.getErrorCode() );

    se = se.getNextException();
}
}
catch( Exception e )
{
    System.out.println( e );
}
}
```

Example

```
import java.sql.*;
public class MySql{
    public static void main (String args[]){
        int i;
        Connection con = null;
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection("jdbc:odbc:MyDB");
            System.out.println("Connected Successfully");
            String sqlStatement = "select stID, StName from student";
            Statement stmt = con.createStatement();
            ResultSet rSet = stmt.executeQuery(sqlStatement);

            while(rSet.next())
                System.out.println("ID: "+rSet.getString
                    "+rSet.getString(2));
            stmt.close();
            con.close();

        } catch(Exception e){System.out.println(e.getMessage());}
    }
}
```

ResultSet Object [1/2]

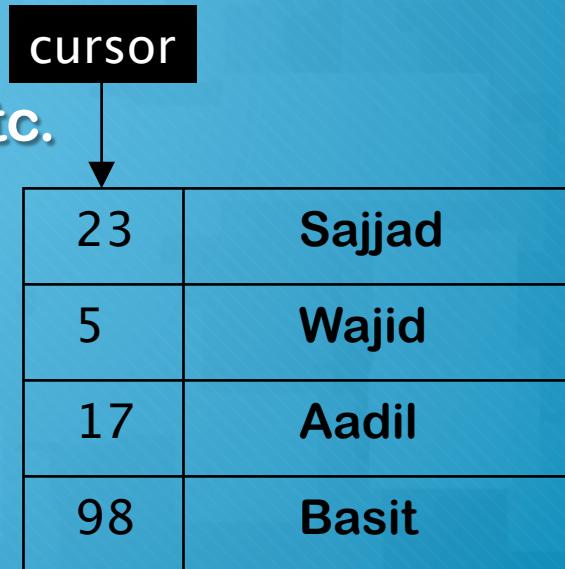
- Stores the results of a SQL query.
- A ResultSet object is similar to a ‘table’ of answers, which can be examined by moving a ‘pointer’ (cursor).

ResultSet Object [2/2]

- Cursor operations:
 - `first()`, `last()`, `next()`, `previous()`, etc.

- Typical code:

```
while( rs.next() ) {  
    // process the row;  
}
```



A diagram illustrating a cursor operation. A black rectangular box labeled "cursor" contains a downward-pointing arrow. This arrow points to the top-left cell of a 5x2 grid table. The table has five rows and two columns. The data in the table is as follows:

23	Sajjad
5	Wajid
17	Aadil
98	Basit

Update in JDBC

- The following JDBC statement can be used to perform update or insert to the database:

```
stmt.executeUpdate("the insert or update SQL statement");
```

```
stmt.executeUpdate("update Motorcycle  
set Maker='Honda_H' where Model=' + id + ');
```

```
String sqlStatement = "select stID, StName from student";
String sqlStatementInsert = "INSERT INTO student VALUES (4,'Irshad', 'Sukkur', 0333)";
String sqlStatementUpdate = "UPDATE student SET StName='Irshad Nazir' WHERE StName='Irshad'";
String sqlStatementDelete = "DELETE FROM student WHERE StName='Irshad Nazir'";
String sqlStatementSelective = "select * from student";
```

Transactions

- Transaction – more than one statement which must all succeed or all fail together
- Also can't leave DB in inconsistent state halfway through a transaction
- COMMIT: Complete Transaction
- ROLLBACK: abort

Questions?



Thanks...

Raheel Ahmed Memon

Object Oriented Programming (JAVA)

Lecture 39



Java Network Programming

Recommended Reading Ch # 20: Networking

Network Programming

- All network computers have an IP Address
 - Unique
 - In the form of xxx. xxx. xxx. xxx (eg. 192.168.0.10)
 - 32-bit = ~4billion possibilities (IPv4)
- Network Interface Card, AKA interfaces, Ethernet adapter, VPN.
- *Some computers also can have multiple NICs*

Network Programming

- **Socket:** A socket is an endpoint of two way communication link between two programs running on the network
- Socket is a bidirectional communication channel,
- Socket is bound to a port number
- Socket is an abstraction of the network similar to the way a file is an abstraction of your hard drive. So that you store and retrieve data from hard drive without knowing the complexities.

IP Address

- IN Windows:
- Open Command Prompt and type: **ipconfig**



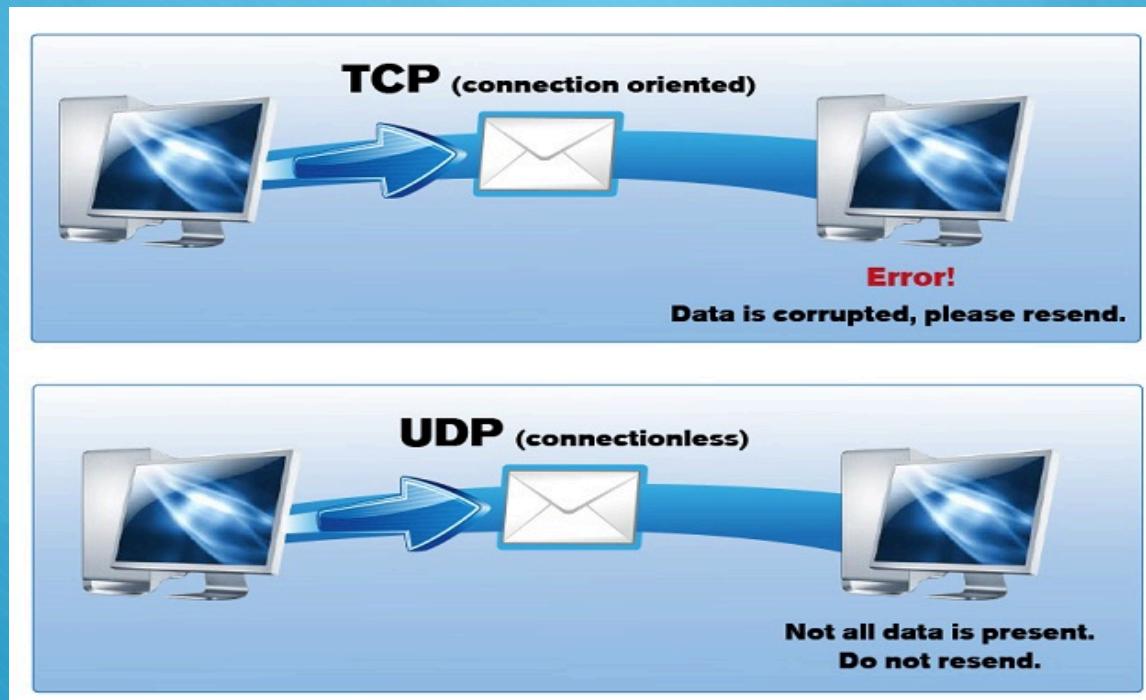
```
Command Prompt  
C:\Users\SBSAdmin>ipconfig  
Windows IP Configuration  
  
Ethernet adapter Local Area Connection:  
  
Connection-specific DNS Suffix . :  
Link-local IPv6 Address . . . . . : fe80::b8d9:c753:a9c3:b41bx11  
IPv4 Address . . . . . : 192.168.15.2  
Subnet Mask . . . . . : 255.255.255.0  
Default Gateway . . . . . : 192.168.15.1
```

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the command "ipconfig" being run, followed by the output for the "Ethernet adapter Local Area Connection". The output includes the connection-specific DNS suffix, link-local and IPv4 addresses, subnet mask, and default gateway.

Port

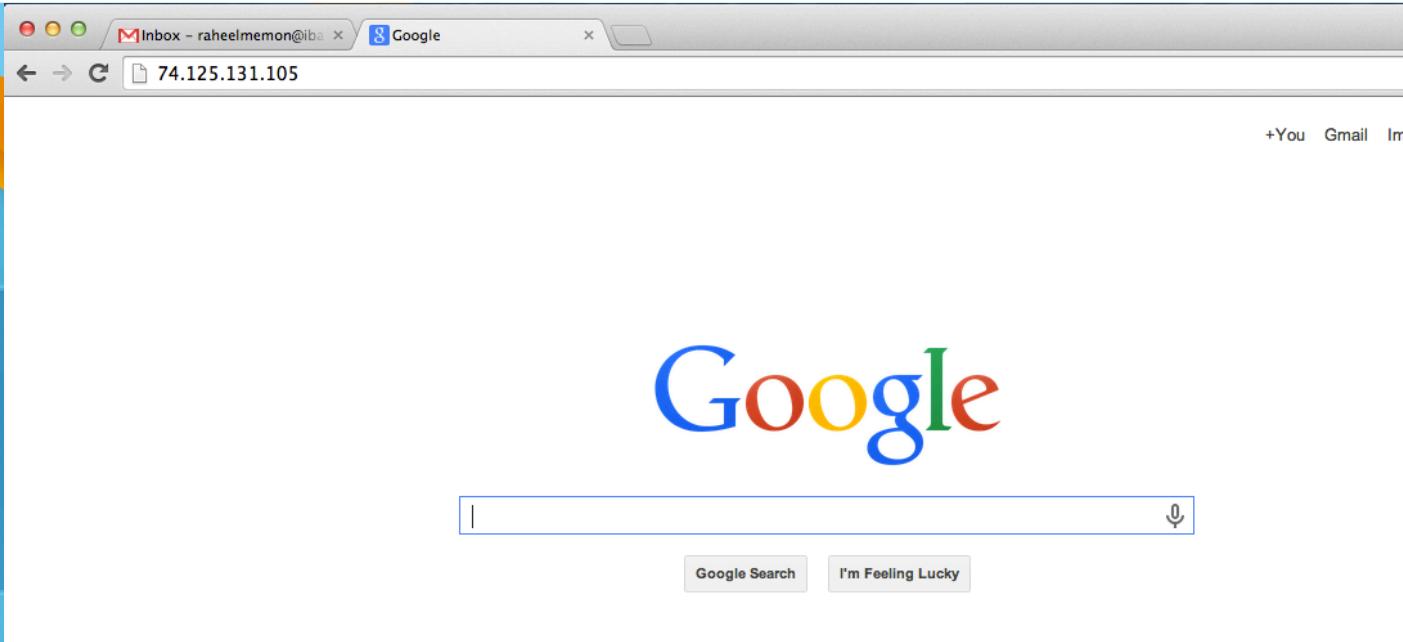
- Transport address to which processes can listen for connection requests
- Machine can specify 65000 local host (TCP & UDP) Ports
 - 1024 are reserved for OS
- Common Ports:
 - 21 FTP (File Transfer Protocol)
 - 22 SSH (Secure Shell)
 - 23 Telnet
 - 25 SMTP (Simple Mail Transfer Protocol)
 - 80 HTTP (Hypertext Transfer Protocol)
 - 110 POP3 (Post Office Protocol)

TCP/UDP



There are reserved ports for both

DNS

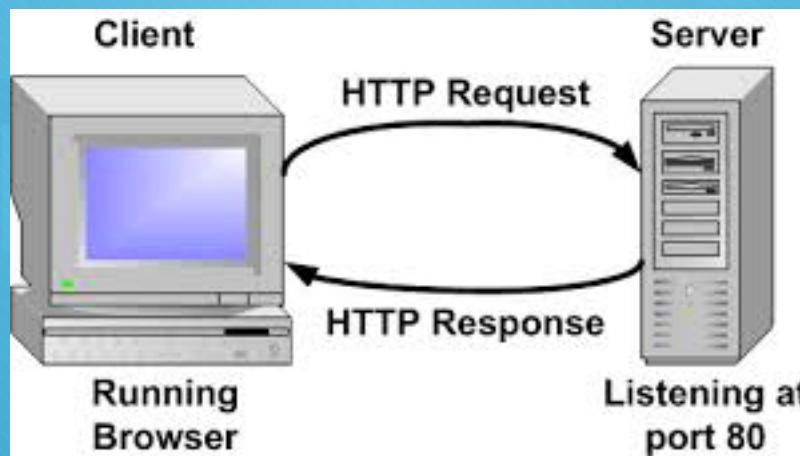


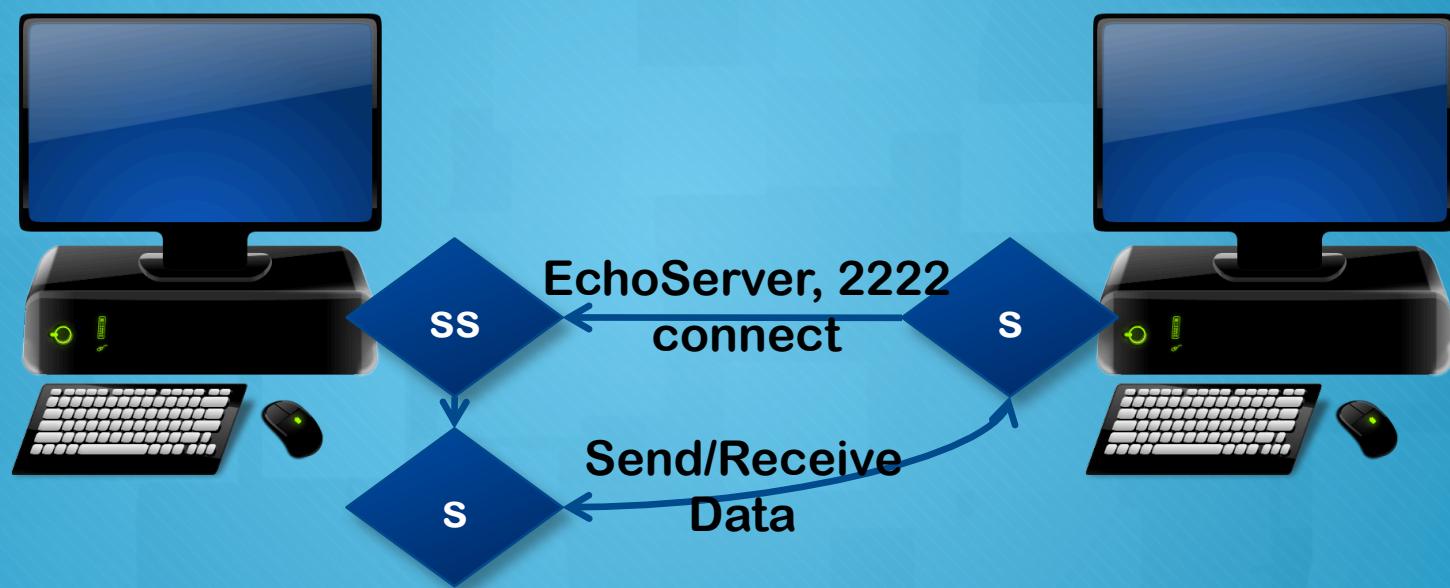
- Domain Name Service
 - Translate Human readable into IP Address
- Example:
 - www.google.com
 - IP Address: 74.125.131.105
- Type nslookup www.google.com in command prompt

```
raheelmemon — bash — 80x24
Raheels-MacBook-Pro:~ raheelmemon$ nslookup www.google.com
Server:      198.6.1.1
Address:     198.6.1.1#53

Non-authoritative answer:
Name:  www.google.com
Address: 74.125.131.105
Name:  www.google.com
Address: 74.125.131.104
Name:  www.google.com
Address: 74.125.131.106
Name:  www.google.com
Address: 74.125.131.105
```

Request Response Model





Steps to Make a Client

[1/2]

1. Import required packages

- `import java.net.*;`
- `import java.io.*;`

2. Connect/Open a socket with Server

- `Socket s = new Socket("sAdd", sPort);`

3. Get I/O Streams of Socket

INPUT

- `InputStream is = s.getInputStream(); // Byte Reader`
- `InputStreamReader isr = new InputStreamReader(is); // Character Reader`
- `BufferedReader br = new BufferedReader(isr); // Character Reader`

Steps to Make a Client

[2/2]

3. Get I/O Streams of Socket (*continue*)

OUTPUT

- `OutputStream os = s.getOutputStream(); // Byte Writer`
- `PrintWriter pw = new PrintWriter(os, true); // Char Writer`

4. Send/Receive Message

SEND:

- `pw.println("Hello Networking");`

RECEIVE:

- `String recMsg = br.readLine();`

5. Close Socket

- `s.close();`

Steps to Make a Simple Server [1/3]

1. Import required package

- import java.net.*;
- import java.io.*;

2. Create a Server Socket

- When Connection established, communication socket is returned.

3. Wait for incoming connection(s)

- `ServerSocket ss = new ServerSocket(portNo);`
- `Socket s = ss.accept();`

Steps to Make a Simple Server [1/3]

4. Get I/O Streams of communication Socket
 - *Same as of Client*
5. Send/Receive Message
 - *Same as of Client*
6. Close Socket
 - *Same as of Client*

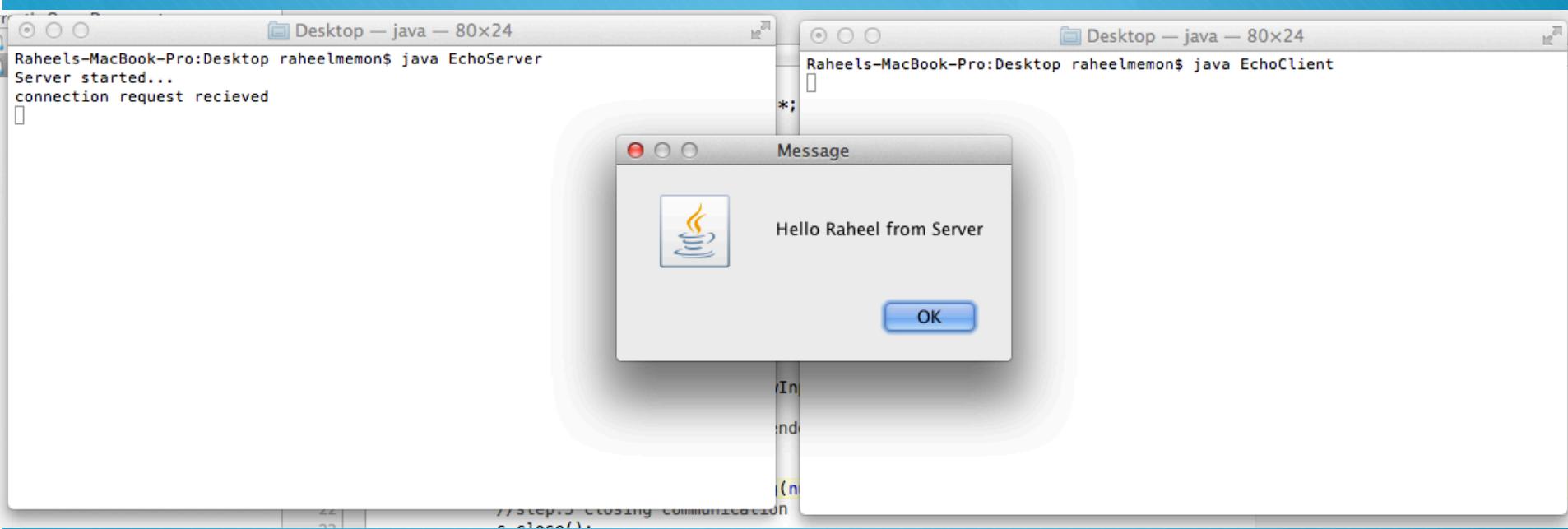
Example: EchoServer

```
// step 1: importing required package
import java.io.*;
import java.net.*;
public class EchoServer {
    public static void main(String args[]) {
        try {
            //step 2: create a server socket
            ServerSocket ss = new ServerSocket(2222);
            System.out.println("Server started...");
            while(true) {
                Socket s = ss.accept();                                // step 3: wait for
                System.out.println("connection request received"); // step 4:
                InputStream is = s.getInputStream();InputStreamReader isr= new
                InputStreamReader(is);BufferedReader
                br = new BufferedReader(isr);
                OutputStream os = s.getOutputStream();PrintWriter pw = new PrintWriter
                (os,true); // step 5: Send / Receive msg
                String name = br.readLine();
                String msg = "Hello " + name + " from
                Server"; // sending back to client
                pw.println(msg);
                // step 6: closing communication
                s.close(); } // end while
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
}
```

```
// step 1: importing required package
import java.net.*; import java.io.*; import javax.swing.*;
public class EchoClient{
    public static void main(String args[]){
        try {
            //step 2: create a communication socket
            Socket s = new Socket("localhost", 2222);
            // step 3: Get I/O streams
            InputStream is = s.getInputStream();
            InputStreamReader isr= new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            OutputStream os = s.getOutputStream();
            PrintWriter pw = new PrintWriter(os,true);
            // step 4: Send / Receive message
            // sending name to server
            String msg = JOptionPane.showInputDialog("Enter your name");
            pw.println(msg);
            // reading message (name appended with hello) from// server
            msg = br.readLine();
            // displaying received
            JOptionPane.showMessageDialog(null , msg);
            //step:5 closing communication
            s.close();
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
} // end class
```

Example: EchoClient

How to run this program?



Questions?



Thanks...