
Core Java

Onkar Deshpande



Module 1. Getting Started

▶ Overview

- ▶ Introduction to Java
- ▶ Writing, compiling and running a program
- ▶ Platform independency in Java
- ▶ Integrated Development Environment
- ▶ Some important terms in Java

Introduction to Java

- ▶ Developed by : James Gosling
- ▶ Released and Controlled by Sun Microsystems, USA
- ▶ Some important features :
 - ▶ Simplicity
 - ▶ Syntax borrowed from C++, eliminating the complex pointer concept.
 - ▶ Object Oriented Programming
 - ▶ Supports all features of OOP's.
 - ▶ Secure
 - ▶ Type-checked language.
 - ▶ Platform independence.
 - ▶ Write Once, Compile Once, Run Anywhere feature.

Writing a program

- Create a directory, such as C:\JavaTraining for your code.
- Using Notepad, create a file called "HelloWorld.java" in your source directory, with the following code:

```
package module1.helloworld;

public class HelloWorld {

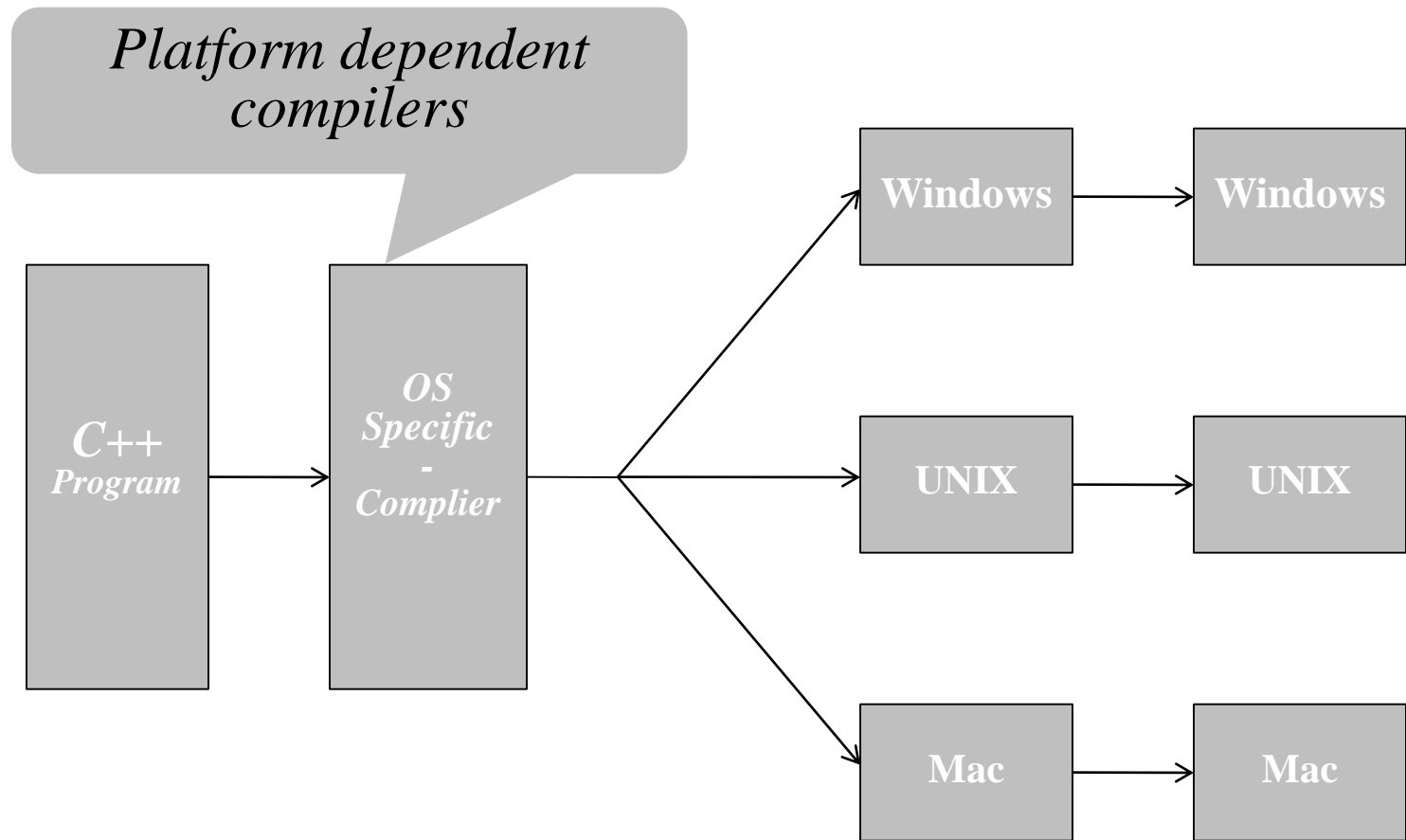
    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

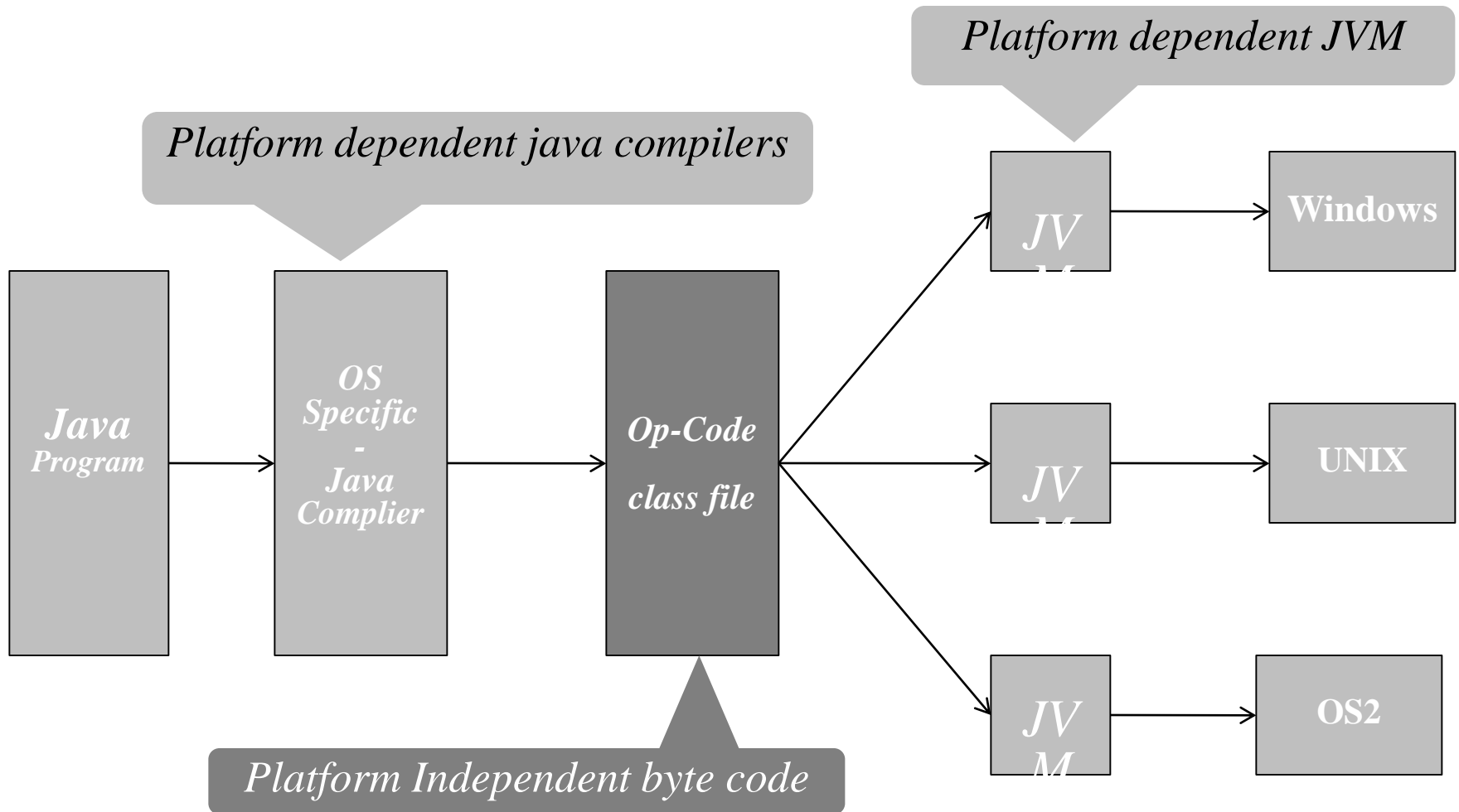
Compiling and running a program

- ▶ Set the environment variable PATH to Java's bin directory.
- ▶ To compile, at the command prompt, type:
javac HelloWorld.java
- ▶ If there are no errors, there should be a file called **HelloWorld.class** in your working directory.
- ▶ To run the program, at the command prompt, type:
java HelloWorld

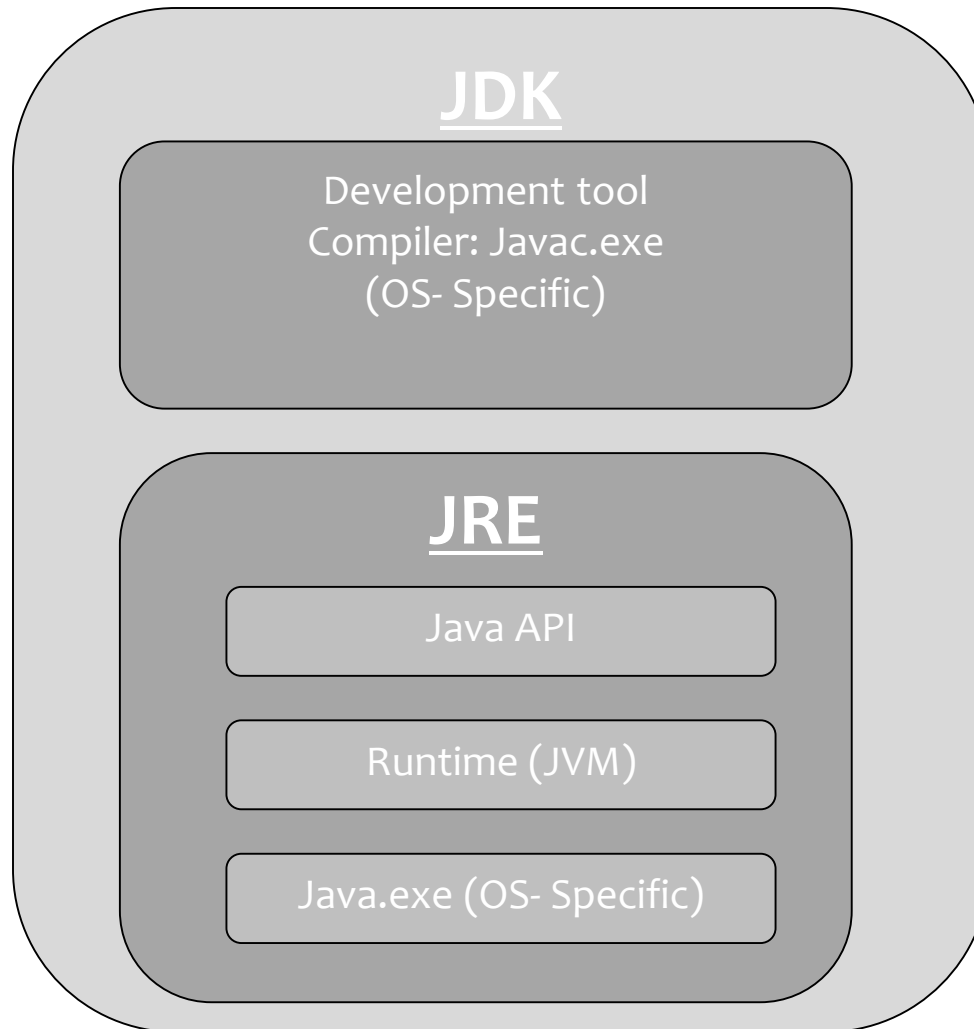
Platform Dependency



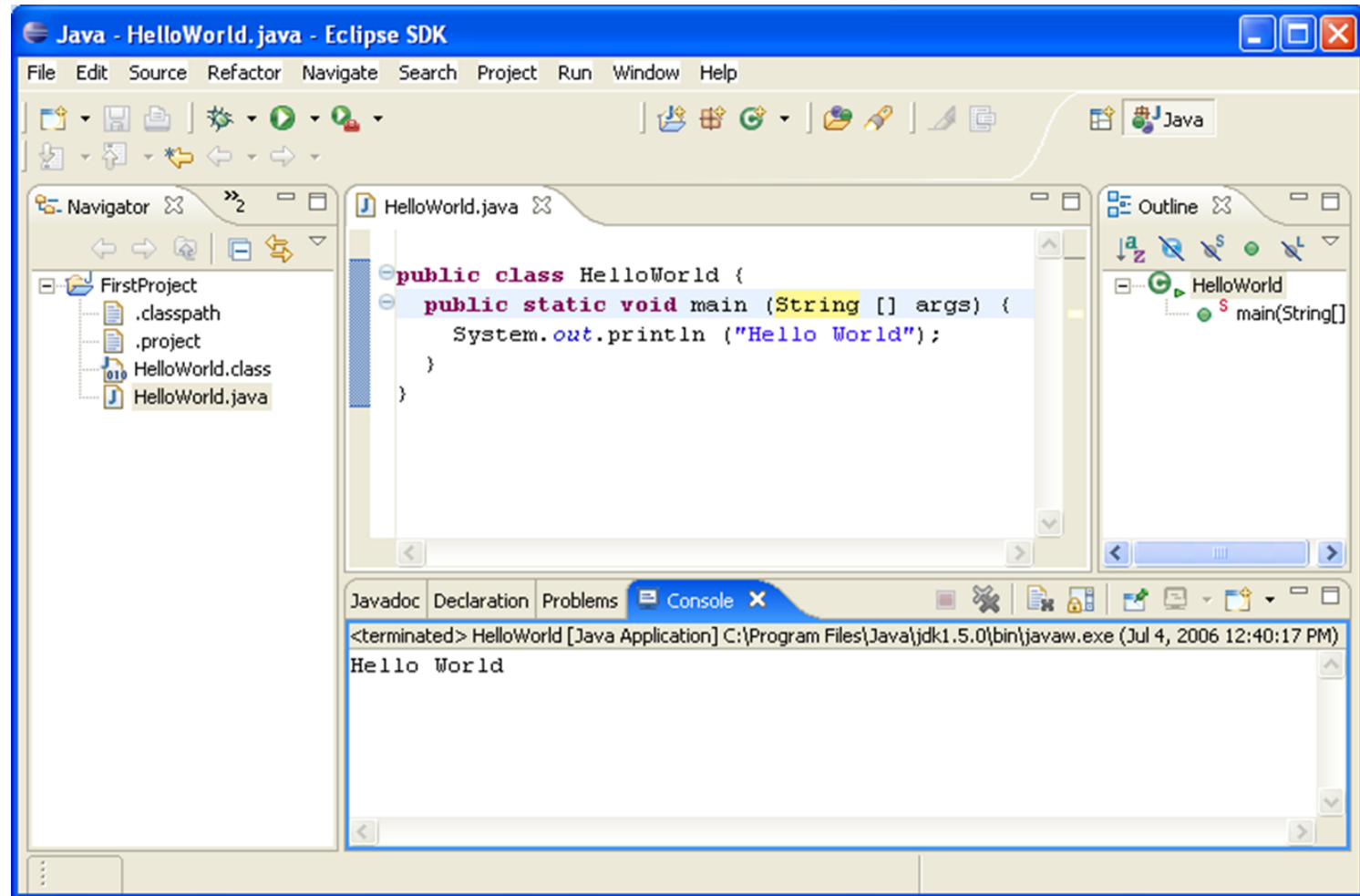
Platform Independency



Some Important Terms in Java



Integrated Development Environment



Popular IDE

- ▶ NetBeans
- ▶ Eclipse
- ▶ IntelliJ
- ▶ jDeveloper

Module 2. Basic Language Constructs

▶ Overview

- ▶ Naming conventions in Java
- ▶ Data types, Variables and Named Constants
- ▶ Writing Comments
- ▶ Operators (arithmetic, assignment, relational, logical and bitwise)

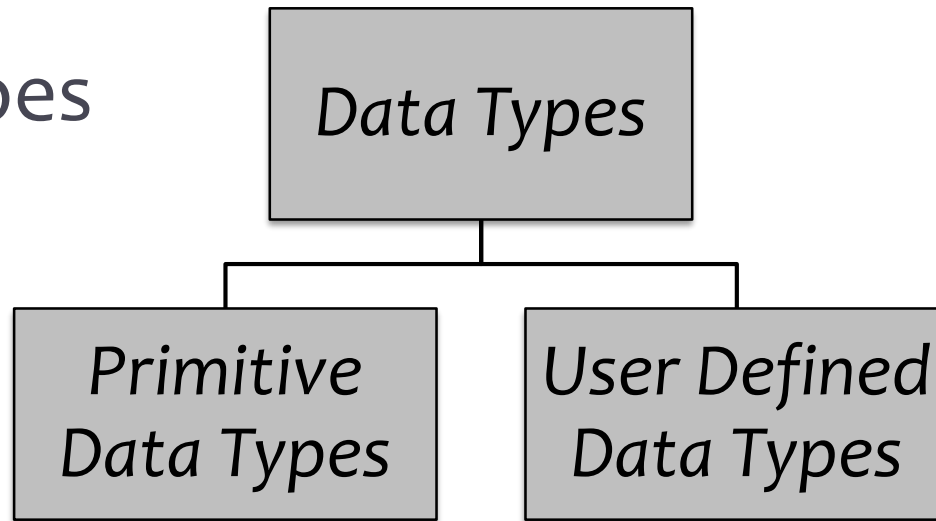
Naming Conventions in Java

- ▶ ClassNames
- ▶ variableNames
- ▶ methodNames
- ▶ packagenames
- ▶ CONSTANTS

Data Types

- ▶ Primitive data types
- ▶ User defined data types

Data Types



Types
<i>byte</i>
<i>short</i>
<i>int</i>
<i>long</i>
<i>float</i>
<i>double</i>
<i>char</i>
<i>boolean</i>

Types
<i>class</i>
<i>enum</i>
<i>Array</i>

Data Types (Cont...)

byte	8-bit integer (signed) Size (-2^7 to 2^7-1)
short	16-bit integer (signed) Size (-2^{15} to $2^{15}-1$)
int	32-bit integer (signed) Size (-2^{31} to $2^{31}-1$)
Long	64-bit integer (signed) Size (-2^{63} to $2^{63}-1$)
Float	32-bit floating-point. Precision to 7-8th digit
Double	64-bit floating-point. Precision to 16-18th digit
boolean	either true or false
char	16-bit Unicode

Variables

```
public class Temperature {  
    public static void main (String [ ] args) {  
        double centigrade;  
        double fahrenheit;  
        centigrade = 33.33;  
        fahrenheit = (centigrade * 9 / 5) + 32;  
        System.out.println(centigrade+" Centigrade ="  
        +fahrenheit + " Fahrenheit.");  
    }  
}
```


Variables

```
public class Temperature {  
  
    public static void main (String [ ] args) {  
        double centigrade;  
        double fahrenheit;  
        centigrade = 33.33;  
        fahrenheit = (centigrade * 9 / 5) + 32;  
        System.out.println(centigrade+" Centigrade ="  
+fahrenheit + " Fahrenheit.");  
    }  
}
```

Notice that we have changed the data type from "double" to "float". Now the program gives an error on compiling. Why? And how to correct this problem?

Writing Comments

```
public class Temperature {
```

```
    /** The program is written by...  
    @ Author Onkar Deshpande  
    */
```

documentation comment

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

```
        //Variable declarations:
```

Single line comment

Multiple line comment

```
        /* double centigrade;  
        double fahrenheit;  
        */
```

```
        double centigrade;
```

```
        double fahrenheit;
```

```
        centigrade = 33.33;
```

```
        fahrenheit = (centigrade * 9 / 5) + 32;
```

```
        System.out.println(centigrade + " Centigrade = "  
        + fahrenheit + " Fahrenheit.");
```

```
    }
```

```
}
```

Named Constants

```
import java.util.Scanner;
```

```
public class Area {
```

```
    static final float PI = 3.1416f;
```

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

```
        float radius;
```

```
        float area;
```

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

```
        radius = sc.nextFloat();
```

```
        area = PI*radius * radius ;
```

```
        System.out.println ("Area is " + area);
```

```
    }
```

```
}
```

Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
%	remainder

Unary minus (-) for negation

Unary plus (+).

Increment and Decrement Operators

```
public class IncOp {  
    public static void main(String[] args) {  
        int i = 1, j=1;  
        System.out.println (++i + " " + i++ + " " + i);  
    }  
}
```

What will the output of this program?

Assignment Operators

Let initial value of j be 15,

`i = j;`

`i += 5;`

`i = i + 5;`

`i -= 5;`

`i *= 5;`

`i /= 5;`

`i %= 5;`

What would be the final value for 'i'?

Conditional and Comparison Operators

== Comparison Operator

!= Not Equal To Operator

< , > Greater than and Less than operators

<= , >= Greater than or equal to / Less than or equal to

Logical and Boolean Operators

```
public class Test {  
    public static void main (String [ ] args) {  
        if( getConnection() && openFile()){  
            System.out.println("true ");  
        }  
        else{  
            System.out.println("false ");  
        }  
    }  
    public static boolean getConnection() {  
        System.out.println("connecting...");  
        return false;  
    }  
    public static boolean openFile(){  
        System.out.println("opening file");  
        return true;  
    }  
}
```


Arithmetic Operations using Bitwise operators

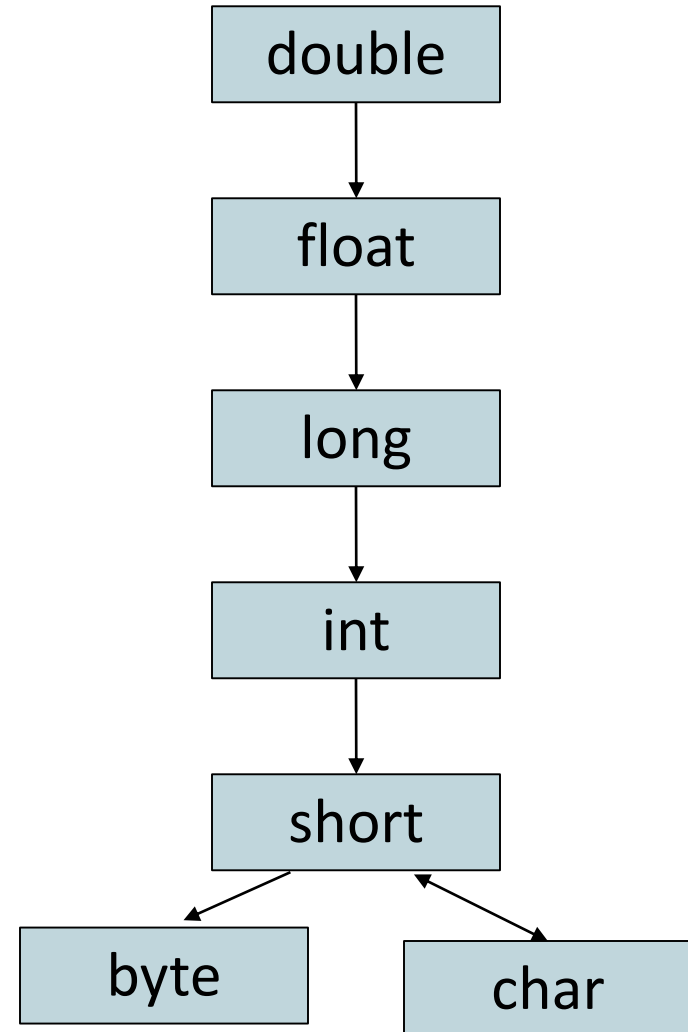
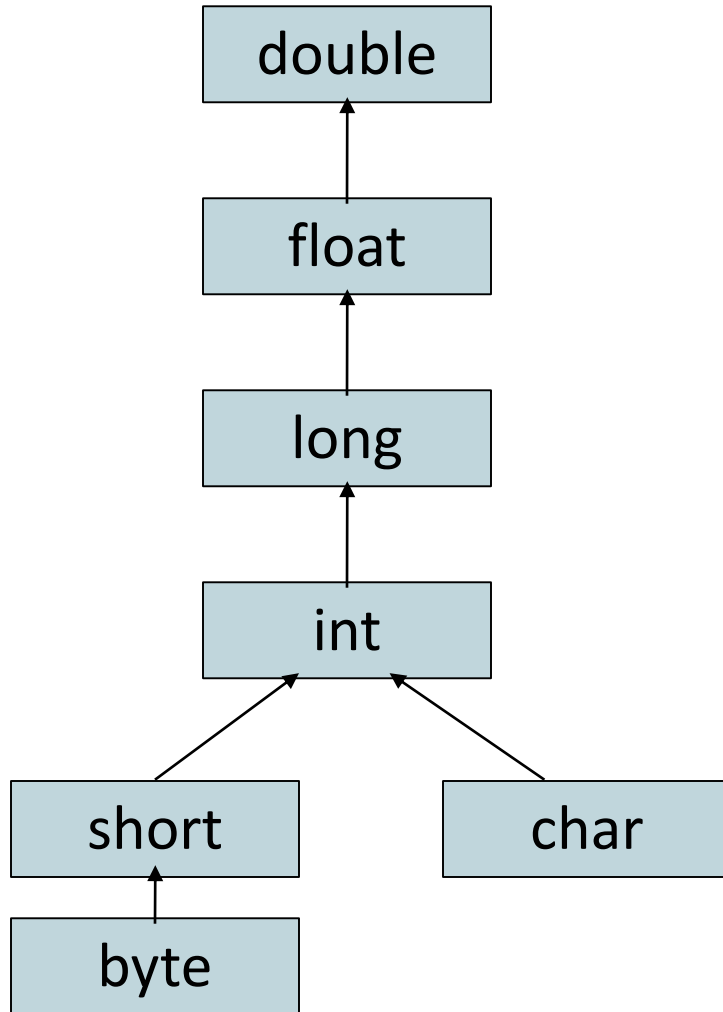
```
public class BitWiseOperators {  
    public static void main(String[] args){  
        int i = 10;  
        // A single left shift will multiply the no by 2.  
        int j = i<<2;  
        System.out.println("The value of j : "+j);  
  
        // A single right shift will divide the no by 2.  
        int k = i>>1;  
        System.out.println("The value of k : "+k);  
    }  
}
```

Module 3. Programing Constructs

▶ Overview

- ▶ Promotion and demotion rules for operators
- ▶ Looping (while, do... while, for loops)
- ▶ Conditional statements (if... else..., switch case)
- ▶ break and continue statements
- ▶ Reference Variables
- ▶ Arrays
- ▶ Arrays of Arrays
- ▶ Object Vs. Local Variables
- ▶ Enhanced for loop
- ▶ **New features of Java 7**
 - ▶ Switch with String
 - ▶ Underscore operator

Promotion and Demotion Rules



The if...else Structure

```
if (expression){  
    statement1;  
}  
else {  
    statement2;  
}
```

The if...else Structure (Cont...)

```
class Larger {  
    public static void main (String [ ] args) {  
        int a = 10;  
        int b = 15;  
        if (a > b){  
            System.out.println ("A is larger.");  
        }  
        else{  
            System.out.println ("B is larger.");  
        }  
    }  
}
```

Nested if...else

```
class LargestOutOfThree {  
    public static void main(String[] args) {  
        int a = 25, b = 15, c = 10, largest;  
        if (a > b) {  
            if (a > c)  
                largest = a;  
            else  
                largest = c;  
        } else {  
            if (b > c)  
                largest = b;  
            else  
                largest = c;  
        }  
        System.out.println("Largest : " + largest);  
    }  
}
```

Another Example For if...else if... Blocks

```
public class DayOfWeek {  
    public static void main(String[] args) {  
        int dayOfWeek = Integer.parseInt(args[0]);  
        if (dayOfWeek == 0)  
            System.out.println("Sunday");  
        else if (dayOfWeek == 1)  
            System.out.println("Monday");  
        else if (dayOfWeek == 2)  
            System.out.println("Tuesday");  
        else if (dayOfWeek == 3)  
            System.out.println("Wednesday");  
        else if (dayOfWeek == 4)  
            System.out.println("Thursday");  
        else if (dayOfWeek == 5)  
            System.out.println("Friday");  
        else if (dayOfWeek == 6)  
            System.out.println("Saturday");  
    }  
}
```

The switch Statement (Contd..)

```
switch (dayOfWeek) {  
    case 0: System.out.println ("Sunday");  
    case 1: System.out.println ("Monday");  
    case 2: System.out.println ("Tuesday");  
    case 3: System.out.println ("Wednesday");  
    case 4: System.out.println ("Thursday");  
    case 5: System.out.println ("Friday");  
    case 6: System.out.println ("Saturday");  
  
    default : System.out.println ("ERROR!");  
}
```


The switch Statement

```
switch (dayOfWeek) {  
    case 0: System.out.println ("Sunday"); break;  
    case 1: System.out.println ("Monday"); break;  
    case 2: System.out.println ("Tuesday"); break;  
    case 3: System.out.println ("Wednesday"); break;  
    case 4: System.out.println ("Thursday"); break;  
    case 5: System.out.println ("Friday"); break;  
    case 6: System.out.println ("Saturday"); break;  
  
    default : System.out.println ("ERROR!");  
}
```

The while Loop

```
public class WhileLoop {  
    public static void main(String[] args) {  
        int i = 1;  
        while (i <= 19) {  
            System.out.println (i);  
            i += 2;  
        }  
    }  
}
```

The do...while Loop

```
public class DoWhileLoop {  
    public static void main(String[] args) {  
        int i = 0;  
        do {  
            System.out.println (i);  
            i += 2;  
        } while (i <= 19);  
    }  
}
```

The for Loop

```
public class ForLoop {  
    public static void main(String[] args) {  
        int i;  
        for (i = 1; i<=10; i++){  
            System.out.println (i);  
        }  
    }  
}
```

break and continue Statements

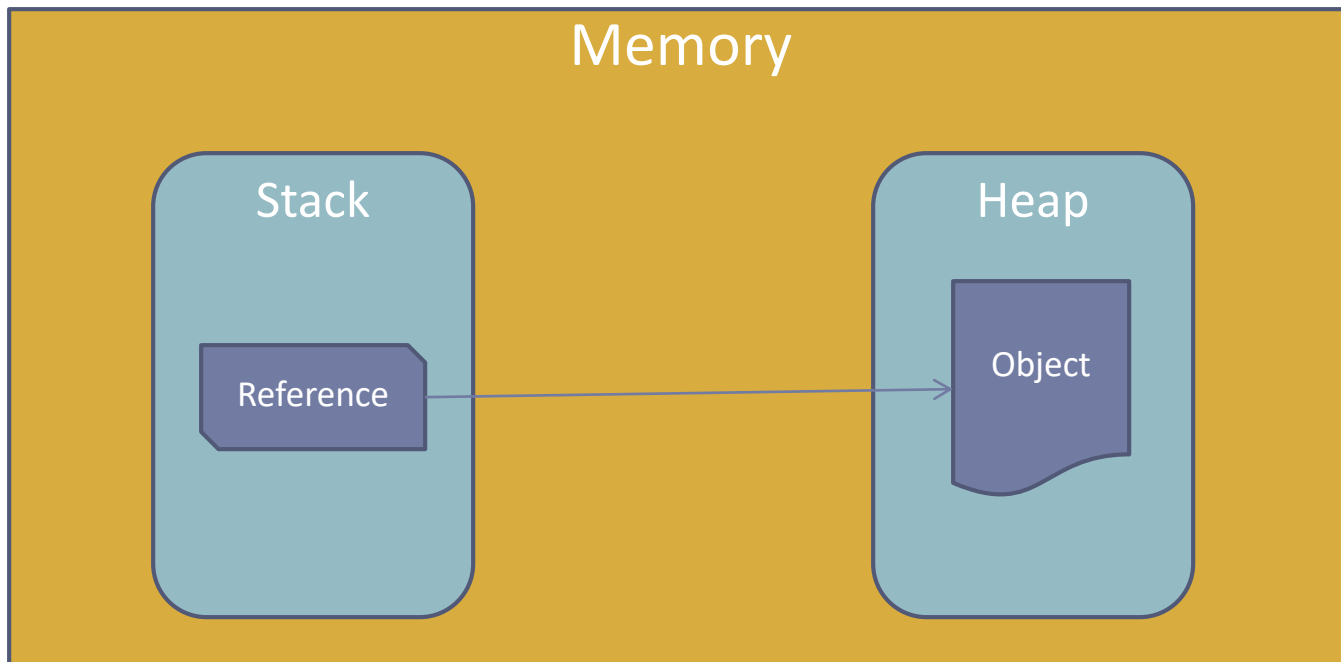
```
public class TestBreakContinue {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 70; i++) {  
            if (i % 7 == 0)  
                continue;  
            System.out.print(" " + i);  
  
            if (i % 40 == 0) {  
                System.out.println("Terminating Loop");  
                break;  
            }  
        }  
    }  
}
```

Labeled break and continue Statements

```
public class LabeledBreak {  
  
    public static void main(String[] args) {  
        outer:  
        for (int i = 1; i <= 10; i++) {  
            for (int j = 1; j <= 5; j++) {  
                System.out.print("\t" + (i * j));  
                if ((i * j) == 18) {  
                    break outer;  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

Reference Variables

- ▶ Reference and Object creation
 - ▶ `int [] intArray = new int [5];`
 - ▶ `String s = " Pragati Software Pvt. Ltd";`



Default Values in Array Initialization

Type	Initial value
boolean	false
char	'\u0000'
byte, short, int, long	0
float	0.0f
Double	0.0
Object Reference	null

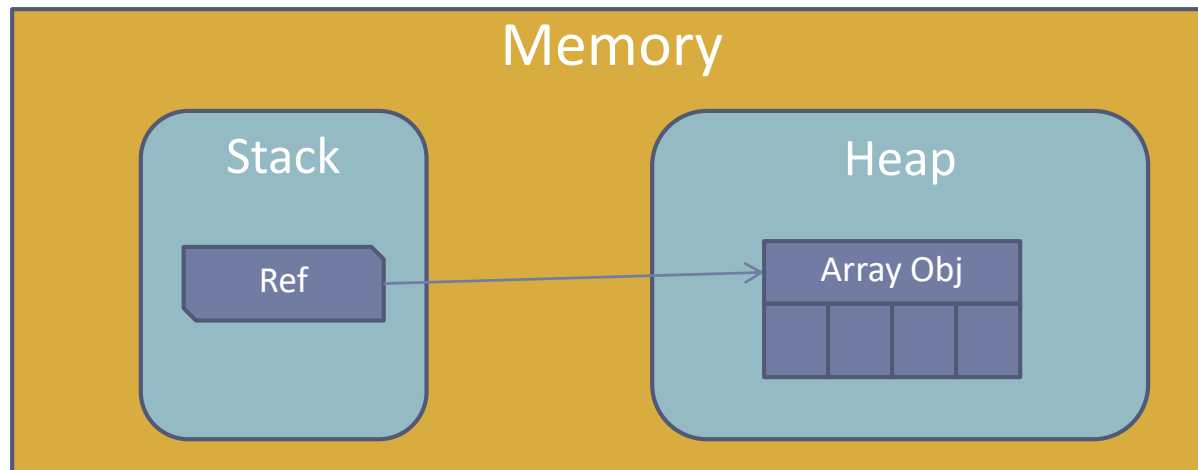
Arrays

▶ Array Declaration

- ▶ `int [] intArray = new int [5];`
- ▶ `int [][] array2D = new int [4][4];`

▶ Length of an array

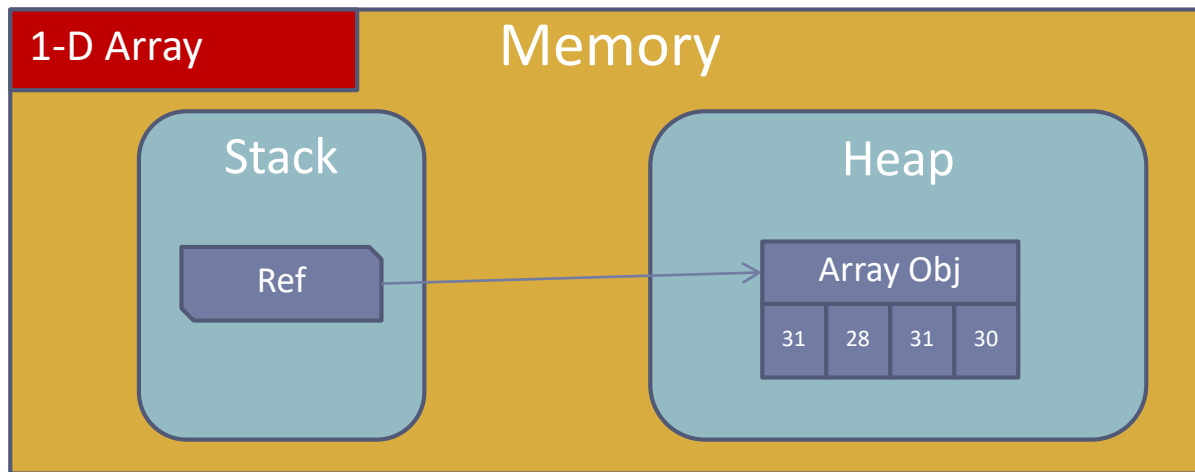
- ▶ `for (int i = 0; i < intArray.length; i++)`
- ▶ `System.out.println (intArray [i]);`



Arrays (Cont...)

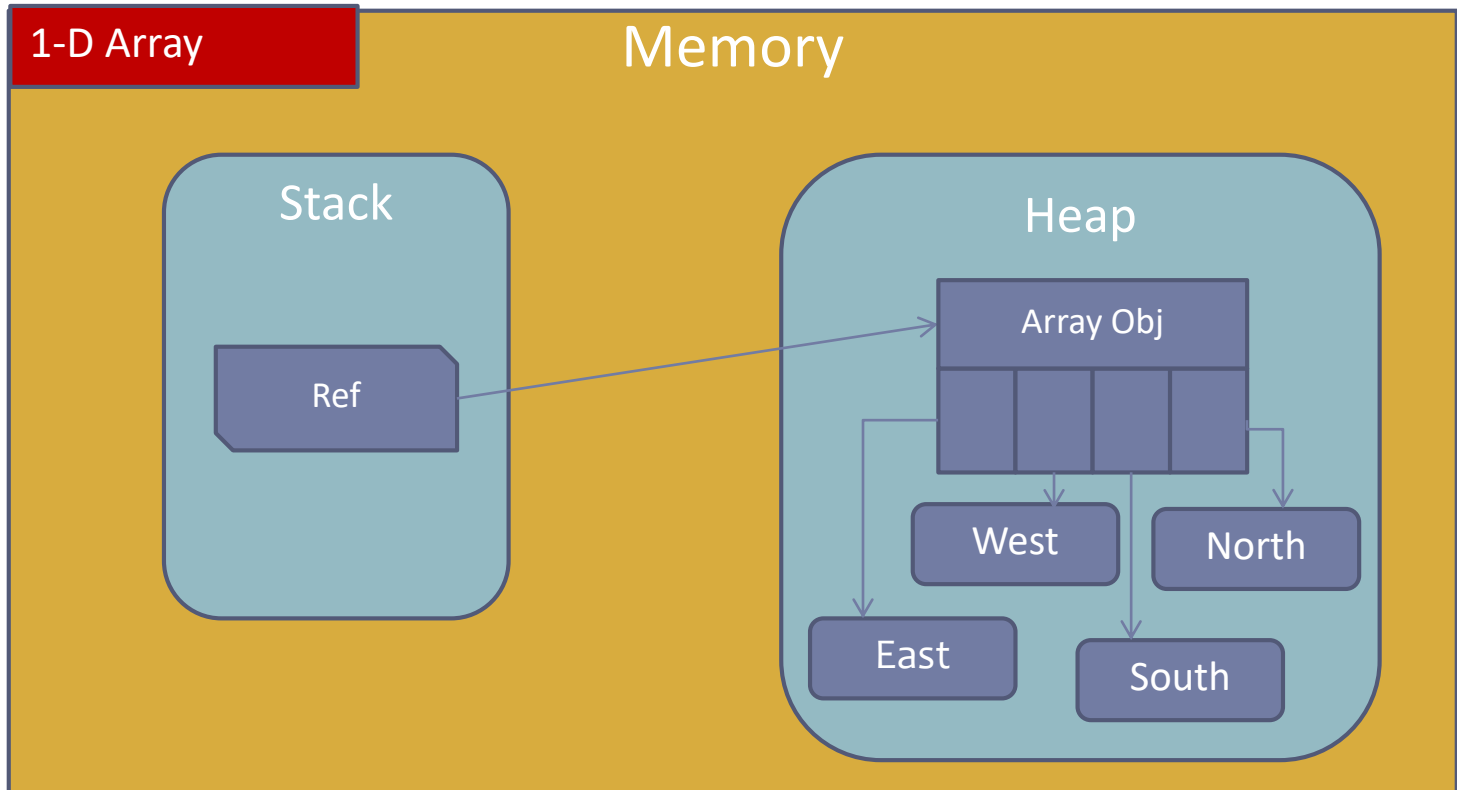
► Array Initialization

► `int [] daysInMonths = { 31, 28, 31, 30, };`



Memory Map of array (contd...)

► `String [] zones = { "East", "West", "North", "South" } ;`



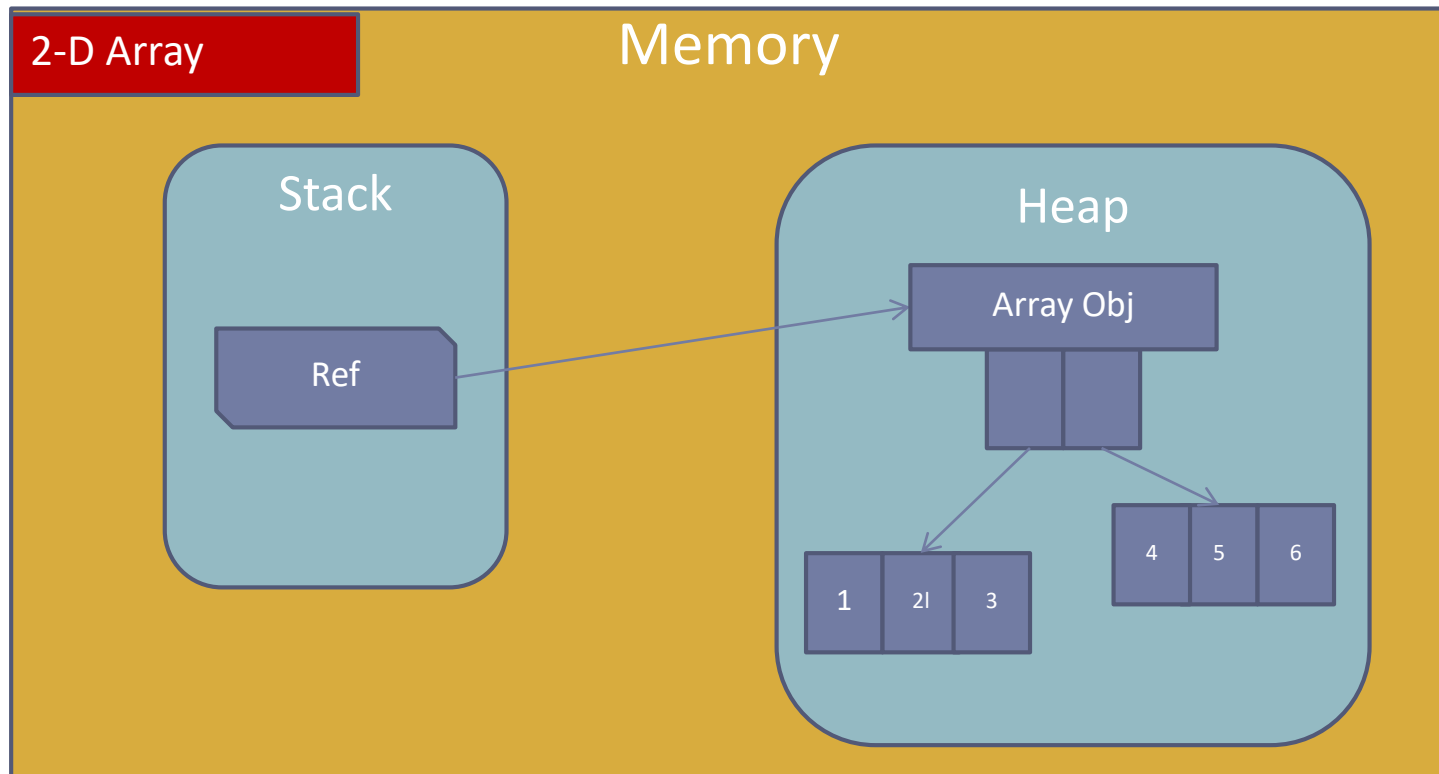
Arrays of Array

```
int [ ][ ] pascalsTriangle = {  
    {1},  
    {1, 1},  
    {1, 2, 1},  
    {1, 3, 3, 1},  
    {1, 4, 6, 4, 1}  
};
```

```
int[ ][ ] array2D = new array2D [ 5 ][ ]; // First subscript is mandatory.
```

Memory Map of array

```
int [ ][ ] mat = {  
    { 1, 2, 3 },  
    { 4, 5, 6 }  
};
```



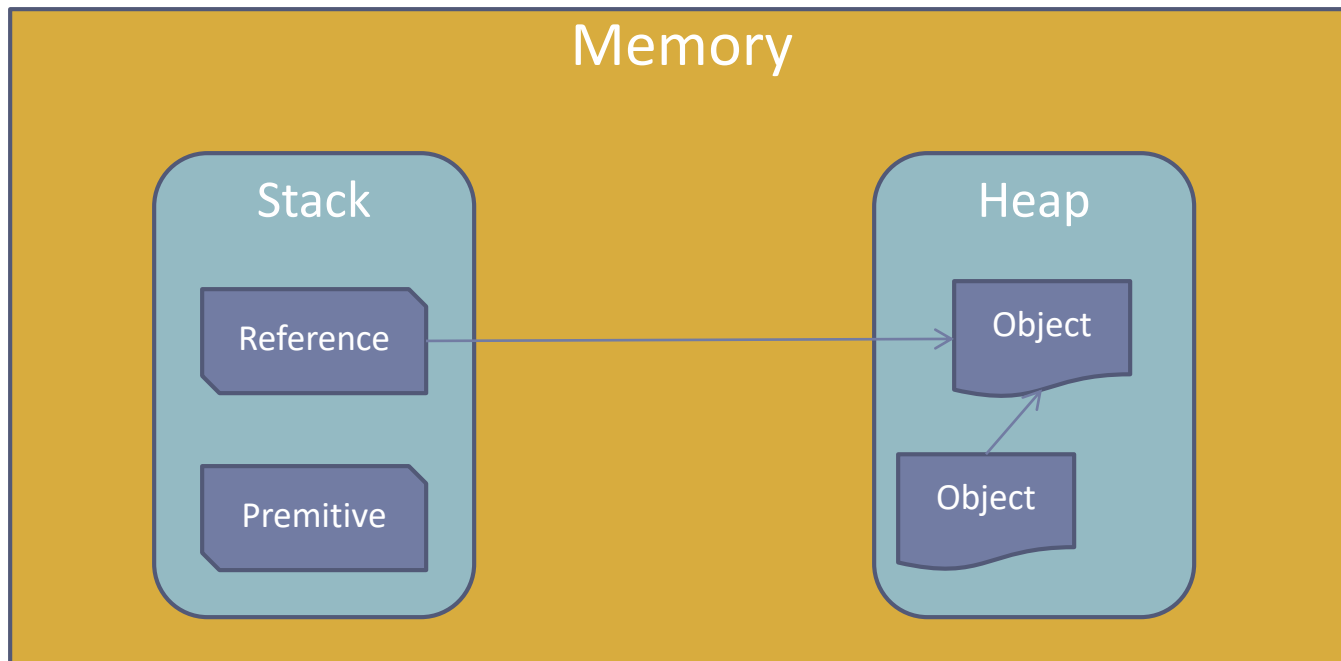
Arrays of Arrays

```
import java.util.Scanner;

public class TwoDimArray {
    public static void main(String[] args) {
        int [][] marks = new int [4][4];
        Scanner sc = new Scanner(System.in);
        for (int i = 1; i < marks.length ; i ++ ) {
            System.out.println ("Enter marks of student "+i);
            for (int j = 1; j < marks[i].length ; j++) {
                System.out.println ("Subject "+j);
                marks [i][j] = sc.nextInt();
            }
        }
        for (int i = 1; i < marks.length ; i++) {
            for (int j = 1; j < marks[i].length ; j++)
                System.out.print (marks[i][j]+" ");
        }
    }
}
```

Objects Vs. Local Variables

```
int [ ] intArr = new int[ 15 ];  
int x;
```



An Enhanced Version of For Loop

```
public class EnhancedForLoop {  
  
    public static void main(String[] args) {  
        int numbers[]={40,50,60,70,80};  
        for (int x : numbers) {  
            System.out.println("Value is : " + x);  
        }  
    }  
}
```


Module 4. Classes and Objects

▶ Overview

- ▶ OOP and POP
- ▶ Classes and objects
- ▶ Access Specifiers
- ▶ Method Overloading
- ▶ Constructors and Init blocks
- ▶ Static methods and fields
- ▶ Var-args
- ▶ Garbage collection -finalize() method
- ▶ The 'toString()' method
- ▶ The 'this' reference

Procedural Vs. Object Oriented Programming

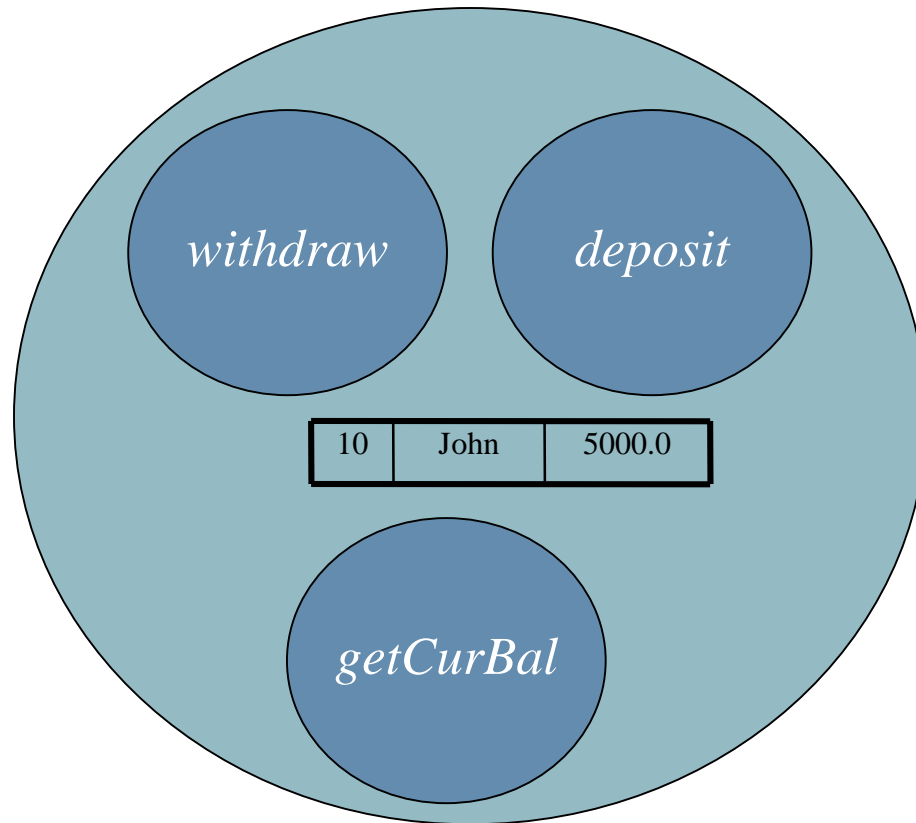
Procedural	Object Oriented
Breakdown programming tasks into variables, subroutines.	Breakdown programming tasks into classes where classes encapsulates own methods.
Uses procedures to operate on data structure. Procedures and data structures are loosely coupled	Bundles both into object structure and thus both are associated with object structure.
Does not control dependency among code part naturally.	Reduces dependency among code part by associating subroutines and fields to object structure.
Nomenclature: Procedure, module, Procedure call, variable	Nomenclature: Method, Object, message, attributes

Procedural Vs. Object Oriented Programming

```
public class Calculator {  
    public static void main(String[] args) {  
        int a, b;  
  
        // Take inputs  
        Scanner sc = new Scanner(System.in);  
  
        a = sc.nextInt();  
        b = sc.nextInt();  
  
        int c = a + b; // Addition  
        System.out.println("a+b=" + c);  
  
        int d = a - b; // Subtract  
        System.out.println("a-b=" + d);  
    }  
}
```

```
public class Calculator {  
    private int a;  
    private int b;  
  
    public void setValues(int a, int b)  
    {  
        this.a = a;  
        this.b = b;  
    }  
  
    public int getAddition() {  
        return a + b;  
    }  
  
    public int getSubtraction() {  
        return a - b;  
    }  
}  
  
// Main method code...  
Calculator c = new Calculator();  
// Read values of a and b  
System.out.println  
(c.getAddition(a, b));
```

Classes and Objects



Creating a Class

```
public class BankAccount {  
    int id;  
    float curBal=0;  
    String name;  
  
    void deposit (float amt) {  
        curBal += amt;  
    }  
    void withdraw (float amt) {  
        curBal -= amt;  
    }  
}
```

Using the BankAccount Class

```
class TestBankAccount {  
  
    public static void main(String[] args) {  
        BankAccount ba = new BankAccount ();  
        System.out.println ("Previous Balance :"  
+ ba.curBal);  
        ba.deposit (5000);  
        System.out.println ("Balance after  
depositing Rs.5000/-      ":" + ba.curBal);  
        ba.withdraw(1000);  
        System.out.println ("Balance after  
withdrawing Rs.1000/-    ":" + ba.curBal);  
    }  
}
```

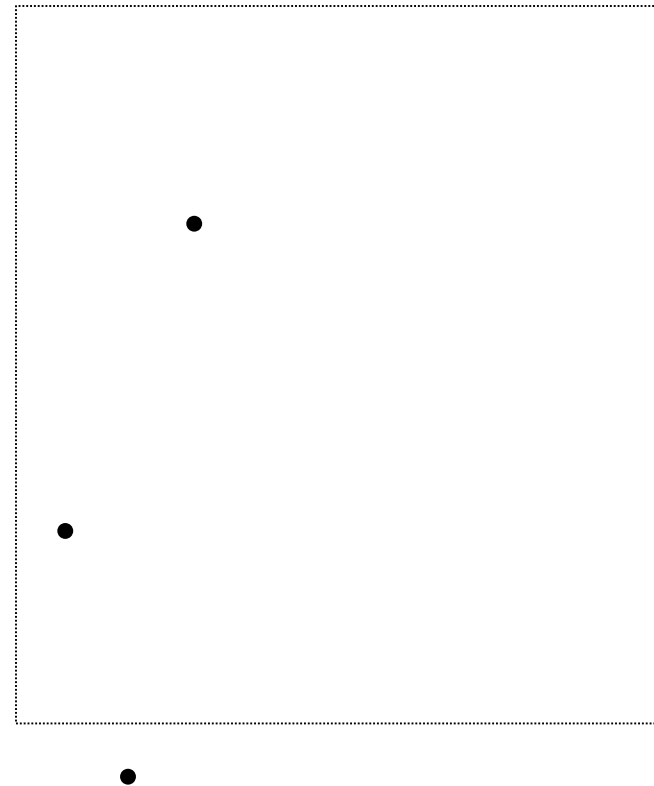


Access Control

- ▶ **private** : Visible to the class only.
- ▶ **package/default** : Visible to the package. the default.
No modifiers are needed.
- ▶ **protected** : Visible to the package and all subclasses.
- ▶ **public** : Visible to the world.

Fields and Methods

```
public class BankAccount {  
    private int id;  
    private float curBal;  
    private String name;  
  
    public void deposit (float amt) {  
        curBal += amt;  
    }  
    public void withdraw (float amt) {  
        curBal -= amt;  
    }  
    public float getCurBal () {  
        return curBal;  
    }  
}
```



Overloading

```
public class Addition {  
    public void add(int a ,int b){  
        System.out.println("Adding 2 integers...");  
        System.out.println("sum : "+(a+b));  
    }  
    public void add(int a ,int b,int c){  
        System.out.println("Adding 3 integers...");  
        System.out.println("sum : "+(a+b+c));  
    }  
    public void add(float a ,float b){  
        System.out.println("Adding 2 float values...");  
        System.out.println("sum : "+(a+b));  
    }  
    public static void main(String[] args){  
        Addition a = new Addition();  
        a.add(10,20);  
        a.add(10,20,30);  
        a.add(10.0f,20.0f);  
    }  
}
```

Constructors

➤ Declaration and Overloading.

```
class BankAccount {  
    private float curBal;  
    public BankAccount ( ) {  
        curBal = 0;  
    }  
    public BankAccount (float amt) {  
        curBal = amt;  
    }  
}
```

➤ Invoking Constructors

```
BankAccount acc1 = new BankAccount ( );  
BankAccount acc2 = new BankAccount (5000);
```

Static Methods and Fields

```
public class BankAccount {
    private int id;
    private float curBal;
    private String name;
    private static int nextId;

    public BankAccount() {
        id = ++nextId;
        curBal = 0;
    }

    public BankAccount(String name, float curBal) {
        id = ++nextId;
        this.name = name;
        this.curBal = curBal;
    }

    public static int getNextId() {
        return nextId;
    }
}
```

Init Blocks

```
public class BankAccount {  
    private int id;  
    private float curBal;  
    private String name;  
    private static int nextId;  
  
    {  
        id=++nextId;  
    }  
  
    public BankAccount () {  
        curBal = 0;  
    }  
    public BankAccount(String name, float curBal) {  
        this.name=name;  
        this.curBal=curBal;  
    }  
}
```

Static initialization block

```
class Primes {  
    static int[] knownPrimes = new int[4];  
  
    static {  
        knownPrimes[0] = 2;  
        for (int i = 1; i < knownPrimes.Length; i++) {  
            knownPrimes[i] = nextPrime();  
        }  
    }  
}  
  
// declaration of nextPrime...  
}
```

Var-Args

```
public class VarArgs {  
    static void test(int ... v) {  
        System.out.println("Number of arguments :  
        "+v.length+" Contains : ");  
        for(int x : v){  
            System.out.println(x+" ");  
        }  
    }  
  
    public static void main(String[] args){  
        test(10);  
        test(1,435,78);  
        test();  
    }  
}
```

Garbage Collection – finalize () Method

```
class Employee {
    Employee(){
        System.out.println ("Employee created...");
    }
    protected void finalize () throws Throwable {
        System.out.println ("\t\tFinalizing...");
    }
}

class GarbageCollectionTest{
    public static void main (String [] args) {
        for (int i = 1; i < 15000; i++){
            Employee x = new Employee();
        }
    }
}
```

Explicitly Destroying an Object

```
BankAccount b = new BankAccount ();  
// ...  
b = null;
```


The 'toString()' Method

```
class BankAccount {
    int accNo;
    String name;
    float curBal;

    BankAccount (int ano, String nm, float bal) {
        accNo = ano;
        name = nm;
        curBal = bal;
    }

    public String toString () {
        String str = name + " has balance of :: " + curBal;
        return str;
    }
}
```

The 'toString()' Method (Cont...)

```
public class ToStringTest {  
    public static void main (String [] args) {  
        BankAccount objectB = null;  
        objectB = new BankAccount (10, "John",  
5000.0f);  
        System.out.println ("Details of objectB  
:" + objectB);  
    }  
}
```

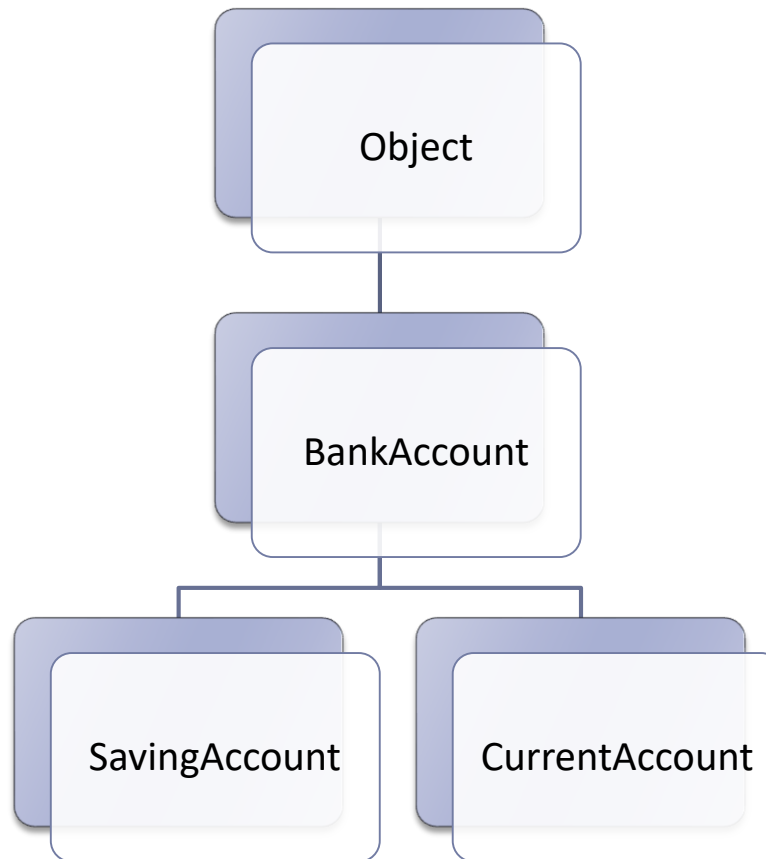
The 'this' Reference

```
public class BankAccount {  
    private int id;  
    private float curBal;  
    private String name;  
  
    public BankAccount () {  
        curBal = 0;  
    }  
    public BankAccount(int id, String name, float curBal) {  
        this.id = id;  
        this.name=name;  
        this.curBal=curBal;  
    }  
}
```

Module 5. Extending Classes

- ▶ Overview
 - ▶ Inheritance
 - ▶ Protected keyword
 - ▶ Constructors in extended classes
 - ▶ Overriding methods
 - ▶ Polymorphism
 - ▶ Hiding Base Class Fields
 - ▶ Making Methods and Classes Final

Extending Classes



The BankAccount Class – The Super class

```
public class BankAccount {  
    private int accNo;  
    public String name;  
    private float curBal;  
    private static int idNum = 1;  
  
    public BankAccount () {  
        accNo = idNum++;  
        curBal = 0;  
    }  
  
    public void deposit (float amt) {  
        curBal += amt;  
    }  
  
    public void withdraw (float amt) {  
        curBal -= amt;  
    }  
  
    public float getCurBal () {  
        return curBal;  
    }  
}
```

The SavingsAccount Class – The Sub Class

```
public class SavingsAccount extends BankAccount {  
    private boolean isSalaryAcc;  
  
    public void setSalaried (boolean is) {  
        isSalaryAcc = is;  
    }  
    public boolean isSalaryAccount() {  
        return isSalaryAcc;  
    }  
}
```

Using a Derived Class Object

```
public class Banking {  
    public static void main (String [] args) {  
        SavingsAccount sa = new SavingsAccount();  
        sa.deposit (5000);  
        System.out.println ("Balance : " + sa.getCurBal  
            ());  
        System.out.println ("Annual interest : "  
            +sa.isSalaryAccount());  
    }  
}
```


The protected Keyword

```
class BankAccount {  
    protected float curBal;  
    protected String name;  
    ...  
    ...  
    ...  
}
```

Constructors in Inheritance

```
class Object {  
    Object( ){  
    }  
}  
class BankAccount{  
    BankAccount( ){  
        ...  
    }  
}  
class SavingsAccount() extends BankAccount{  
    SavingsAccount( ){  
    }  
}  
public static void main(String[ ] args){  
    SavingsAccount sa = new SavingsAccount( );  
}
```

The diagram illustrates the flow of constructor calls in a Java inheritance hierarchy. Red arrows indicate the sequence of constructor calls: from `main` to `SavingsAccount`, then to `BankAccount`, and finally to `Object`. Blue arrows show the inheritance chain: from `SavingsAccount` to `BankAccount`, and from `BankAccount` to `Object`. The blue arrows originate from the `extends` keyword and the `new` keyword, respectively.

Constructors in Extended Classes

```
public class BankAccount {  
    protected float curBal;  
    protected String name;
```

```
    public BankAccount () {  
        curBal = 0;  
    }  
}
```

```
class SavingsAccount extends BankAccount {  
    private boolean isSalaried;  
    SavingsAccount (boolean isSal) {  
        isSalaried = isSal;  
    }  
}
```

Constructors in Extended Classes (contd...)

```
public class BankAccount {
    protected float curBal;
    protected String name;

    public BankAccount (float amt,String name) {
        curBal = amt;
        this.name = name;
    }
}

public class SavingsAccount extends BankAccount {
    private boolean isSalaried;

    SavingsAccount (float amount,String name, boolean isSal) {
        super (amount,name);
        isSalaried = isSal;
    }
}
```

Invoking constructor using 'this' keyword

```
class BankAccount {  
    protected float curBal;  
    protected String name;  
  
    public BankAccount () {  
        curBal = 0;  
    }  
}  
  
class SavingsAccount extends BankAccount {  
    private boolean isSalaried;  
  
    SavingsAccount (float curBal) {  
        this(curBal,true);  
    }  
  
    SavingsAccount (float curBal,boolean sal) {  
        this.curBal = curBal;  
        isSalaried = sal;  
    }  
}
```

Overriding Methods

```
class BankAccount {  
    protected float curBal;  
    protected String name;  
  
    BankAccount(String n, float bal){  
        name = n;  
        curBal = bal;  
    }  
  
    public void print () {  
        System.out.println ("customer : " + name);  
        System.out.println ("Balance : " + curBal);  
    }  
}
```

Overriding Methods (Cont...)

```
class SavingsAccount extends BankAccount {  
    private boolean isSalaried;  
  
    SavingsAccount(String name, float bal, boolean sal){  
        super(name, bal);  
        isSalaried = sal;  
    }  
    public void print (){  
        //super.print();  
        System.out.println ("Is Salaried : " + isSalaried);  
    }  
}
```

The CurrentAccount class

```
class CurrentAccount extends BankAccount{
    private float overDraftLimit;

    CurrentAccount( String name, float bal, float odl){
        super( name, bal);
        overDraftLimit = odl;
    }
    public void print ( ){
        super.print( );
        System.out.println ("OverDraftLimit : " +
overDraftLimit);
    }
}
```


Polymorphism

```
class PolymorphismTest {  
    public static void main (String [] args) {  
        BankAccount ba[] = {  
            new SavingsAccount ("Amar",10000,true),  
            new CurrentAccount ("Akbar",20000,5000)  
        };  
        System.out.println ("Printing polymorphically");  
        for(BankAccount account:ba){  
            account.print();  
            System.out.println ("-----");  
        }  
    }  
}
```

Output

Printing polymorphically

customer : Amar

Balance : 10000.0

Is Salaried : true

customer : Akbar

Balance : 20000.0

OverDraftLimit : 5000.0

Making Methods and Classes as final

```
class LinkedList {  
    public final int count () {  
        ...  
    }  
}
```

```
final class Stack {  
    ...  
}
```

Module 6. Abstract Classes and Interfaces

▶ Overview

- ▶ Abstract classes and methods
- ▶ Extending abstract class
- ▶ Abstract class and Polymorphism
- ▶ Declaring interfaces
- ▶ Implementing interfaces
- ▶ Extending interfaces

Abstract Classes

```
abstract class BankAccount {  
    // ...  
}
```

Abstract Methods

```
public abstract class BankAccount {  
    private int id;  
    protected float balance;  
  
    public BankAccount(int id, float balance) {  
        this.id = id;  
        this.balance = balance;  
    }  
  
    abstract float calculateInterest();  
}
```

Extending an Abstract Class

```
public class SavingsAccount extends BankAccount{  
    public SavingsAccount (int id, float balance) {  
        super (id, balance);  
    }  
    public float calculateInterest() {  
        return balance * 0.10f;  
    }  
}
```

Extending an Abstract Class (Cont...)

```
public class LoanAccount extends BankAccount{  
    public LoanAccount (int id, float balance) {  
        super (id, balance);  
    }  
  
    public float calculateInterest() {  
        return balance * 0.13f;  
    }  
}
```


Extending an Abstract Class (Cont...)

```
class CurrentAccount extends BankAccount{  
    public CurrentAccount(int id, float balance) {  
        super (id, balance);  
    }  
    public float calculateInterest() {  
        return balance * 0.11f;  
    }  
}
```

Abstract class and Polymorphism

```
public class AbstractTest {  
    public static void main (String [] args ) {  
        showInterest (new SavingsAccount (3,5000));  
        showInterest(new LoanAccount (4,6000));  
        showInterest(new CurrentAccount (5,7000));  
    }  
    public static void showInterest(BankAccount e) {  
        System.out.println ("Interest :" +  
            e.calculateInterest ());  
    }  
}
```

Module 7. Interfaces and Lambda Expressions

▶ Overview

- ▶ Declaring interfaces
- ▶ Implementing interfaces
- ▶ Loose coupling using interfaces
- ▶ Extending interfaces
- ▶ Choosing between interface inheritance and class inheritance
- ▶ **New Features of Java 8**
 - ▶ **Defining a Lambda Expression**

Declaring interfaces

```
public interface Graphic{  
    float PI = 3.14f;  
    float area ();  
    float periphery ();  
}
```

Implementing interfaces

```
class Circle implements Graphic {  
    private float radius;  
  
    public Circle (float r) {  
        radius = r;  
    }  
    public float area () {  
        return PI * radius * radius;  
    }  
    public float periphery () {  
        return 2 * PI * radius;  
    }  
}
```

Implementing interfaces

```
class Rectangle implements Graphic {  
    private float width, height;  
  
    public Rectangle (float w, float h) {  
        width = w; height = h;  
    }  
    public float area () {  
        return width * height;  
    }  
    public float periphery () {  
        return 2 * (width + height);  
    }  
}
```

Loose coupling using interfaces

```
class GraphicTest {  
    public static void main (String [] args) {  
        Graphic g1 = new Circle (10);  
        Graphic g2 = new Rectangle (5,4);  
        System.out.println  
        ("Area of circle is " + g1.area ());  
        System.out.println  
        ("Periphery of circle is " + g1.periphery ());  
        System.out.println  
        ("Area of rectangle is " + g2.area ());  
        System.out.println  
        ("Periphery of rectangle is " + g2.periphery ());  
    }  
}
```

Extending Interfaces

```
interface DrawableGraphic extends Graphic {  
    void draw (int x, int y);  
}  
  
class DrawableCircle implements DrawableGraphic {  
    void draw (int x, int y) {  
        //... some code here  
    }  
    // ... other functions  
}
```


Module 8. Nested Classes

- ▶ Overview
 - ▶ Static nested classes
 - ▶ Inner classes
 - ▶ Anonymous inner classes

Static Nested Classes

```
class BankAccount {  
    private float curBal;  
    private String name;  
  
    public static class Permission {  
        public static boolean canDeposit;  
        public boolean canWithdraw;  
    }  
    BankAccount () {  
        name = "Aditi";  
        curBal = 5000;  
        Permission.canDeposit = true;  
    }  
    void show(){  
        System.out.println("Fields from outer class....");  
        System.out.println  
        ("The currentBalance : "+curBal+"\tName : "+name);  
        System.out.println  
        ("Fields from Static Nested Class : ");  
        System.out.println  
        ("CanDeposit ?"+Permission.canDeposit);  
    }  
}
```

Static Nested Classes (Cont...)

```
class StaticNestedClassTest {  
    public static void main (String [] args) {  
        BankAccount b = new BankAccount();  
  
        BankAccount.Permission.canDeposit = true;  
        BankAccount.Permission p = new  
        BankAccount.Permission ();  
        p.canWithdraw = true;  
  
        b.show();  
        System.out.println("Instance field of nested  
        class : "+p.canWithdraw);  
    }  
}
```

Inner Classes

```
class LinkedListTest {
    private Item top = null;
    private Item bottom = null;

    class Item {
        public int next;

        public Item(int next) {
            this.next = next++;
        }
    }

    public void insert(int val) {
        Item item = new Item(val);
        top = item;
        if (bottom == null)
            bottom = item;
    }

    public void show() {
        System.out.println("Top :" + top);
    }
}
```

Inner Classes (contd...)

```
class LinkedList {  
    public static void main(String[] args) {  
        LinkedListTest ll = new LinkedListTest();  
        LinkedListTest.Item li = ll.new  
        Item(100);  
        ll.insert(50);  
        ll.show();  
    }  
}
```

Anonymous Inner Classes

```
class Employee {  
    int empno;  
    float basic;  
    Employee ( int empno, float basic) {  
        this.empno = empno;  
        this.basic = basic;  
    }  
    void showEmployee () {  
        System.out.println ("Number    : " + empno);  
        System.out.println ("Basic      : " + basic);  
    }  
}
```

Anonymous Inner Classes (Cont...)

```
public class AnonymousTest {  
    public static void main (String [] args) {  
        Employee e1 = new Employee (10,5000.0f);  
        e1.showEmployee ();  
        Employee e2 = new Employee (11,6000.0f) {  
            float bonus = 500;  
            void showEmployee () {  
                super.showEmployee ();  
                System.out.println ("Bonus : " + bonus);  
            }  
        };           //end of anonymous inner class  
        e2.showEmployee () ;  
    }  
}
```

Module 7. Packages

▶ Overview

- ▶ Creating packages
- ▶ Naming packages
- ▶ Package Access
- ▶ Packages and class path
- ▶ Importing packages
- ▶ Static imports

Java in-built packages

java.lang

System
Thread
Exception
String
StringBuffer

java.io

Reader
Writer
InputStream
OutputStream
PrintWriter

java.net

Socket
ServerSocket
URLEncoder
InetAddress
SocketImpl

java.util

HashMap
Set
Vector
List
ArrayList

java.awt

Applet
Frame
Button
TextField
Checkbox

Creating Packages

```
package graphics;  
class Circle extends Shape implements  
Draggable {  
    ...  
}
```

Naming Packages

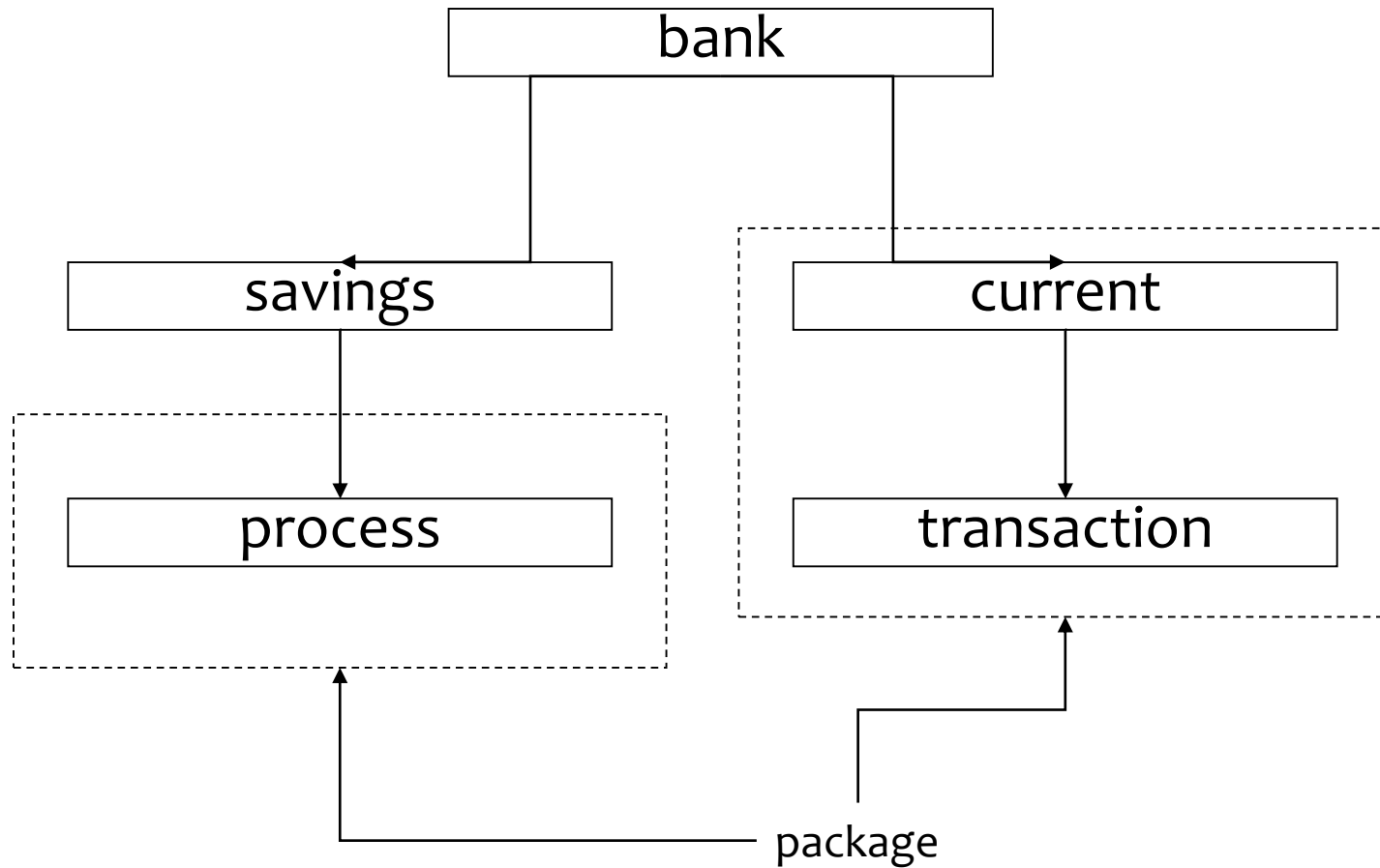
`com.onkarjava.graphics`

`com.onkarjava.sql.graphics`

Access Specification

Access	Private	Default	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-Subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes (But only through Inheritance)	Yes
Different package Non-Subclass	Yes	Yes	Yes	Yes

Classpath



Using a class without importing the package

```
public class ProgramWithoutImport {  
    public static void main(String[] args){  
        module09.packages.mypackage.AccessSpecifierT  
est ast = new  
module09.packages.mypackage.AccessSpecifierT  
est();  
ast.print();  
}  
}
```

Package import

- ▶ Whole name of every component of a package
- ▶ **mypackage.TestAccSpecifiers**
- ▶ Once you import a package...
- ▶ **import mypackage.TestAccSpecifiers;**
- ▶ now, package components could be named as...
- ▶ **TestAccSpecifiers**
- ▶ Importing all components except subpackage from the package...
- ▶ **import mypackage.*;**
- ▶ Importing components of subpackage of package...
- ▶ **import mypackage.subpackage.*;**

Using a class by importing the package

```
import
module09.packages.mypackage.AccessSpecifierTes
t;

public class ProgramWithImport {
    public static void main(String[] args){
        AccessSpecifierTest ast = new
        AccessSpecifierTest();
        ast.print();
    }
}
```


Static Import

```
public class MathImportTest {  
    public static void main (String args [ ]) {  
        double x = Math.sqrt(Math.pow(4,2) + Math.pow(5,2));  
        System.out.println("x : " + x);  
    }  
}
```

```
import static java.lang.Math.sqrt;  
import static java.lang.Math.pow;  
  
public class StaticImportTest {  
    public static void main (String args []) {  
        double x = sqrt(pow(4,2) + pow(5,2));  
        System.out.println("x : " + x);  
    }  
}
```

Module 8. Exceptions

▶ Overview

- ▶ Introduction to Exceptions
- ▶ Unchecked exceptions
- ▶ Checked exceptions
- ▶ The “try-catch” structure
- ▶ The “finally” clause
- ▶ The “throws” clause
- ▶ Custom Exception
- ▶ Exception chaining
- ▶ New Features of Java 7
 - ▶ try with Resources
 - ▶ AutoCloseable
 - ▶ Handling Multiple Exceptions

Exceptions

```
public class TestExceptions{  
    public static void main(String[] argv){  
        int a = 10, b=0, c;  
        c = a/b;  
        System.out.println("C : "+c);  
    }  
}
```

Conventional Way of Exception Handling

```
int readFile ( ) {  
    int errorCode = 0;                // error code to be returned.  
    open the file;  
    if (the file is opened) {  
        determine the file length;  
        if (got the file length)  
            allocate the required memory;  
            if (memory allocated)  
                read the file into memory;  
                if (read failed)  
                    errorCode = -1;  
                else {    errorCode = -2;    }  
            else {    errorCode = -3;    }  
        }  
    else {  
        errorCode = -4 ;  
    }  
}
```

Java-Style Exception Handling Approach

```
readFile( ) {  
    try {  
        open the file;  
        determine the file length;  
        allocate the required memory;  
        read the file into memory;  
        close the file;  
    } catch (file open failed) {  
        do something here;  
    } catch (size determination failed) {  
        do something here;  
    } catch (memory allocation failed) {  
        do something here;  
    } catch (reading the file failed) {  
        do something here;  
    } catch {file close failed) {  
        do something here;  
    }  
}
```

Handling Exceptions

```
public class HandlingExceptions {  
    public static void main (String [] args ) {  
        try {  
            int array [] = {10,20,30};  
            int a = 10, b = 0, c = 0;  
            System.out.println ("array [2] " + array [2] );  
            c = a / b ;  
            System.out.println ("Division : " + c);  
        } catch (ArithmeticException e) {  
            System.out.println ("Math error : " + e);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println ("Array index error : " + e);  
        } catch (Exception e) {  
            System.out.println ("Error : " + e);  
        }  
    }  
}
```

Try-with-resources

```
private static void printFileJava7() throws IOException {  
    try(FileInputStream input = new FileInputStream("file.txt")) {  
        int data = input.read();  
        while(data != -1){  
            System.out.print((char) data);  
            data = input.read();  
        }  
    }  
}
```

Using Multiple Resources

- ▶ You can use multiple resources inside a try-with-resources block and have them all automatically closed.

```
private static void printFileJava7() throws IOException {  
  
    try( FileInputStream input = new FileInputStream("file.txt");  
        BufferedInputStream bufferedInput =  
            new BufferedInputStream(input)) {  
  
        int data = bufferedInput.read();  
        while(data != -1){  
            System.out.print((char) data);  
            data = bufferedInput.read();  
        }  
    }  
}
```


AutoClosable

- ▶ The try-with-resources construct does not just work with Java's built-in classes. You can also implement the `java.lang.AutoCloseable` interface in your own classes, and use them with the try-with-resources construct.

```
public interface AutoClosable {  
    public void close() throws Exception;  
}
```

AutoClosable Example

```
public class MyAutoClosable implements AutoCloseable {  
    public void doIt() {  
        System.out.println("MyAutoClosable doing it!");  
    }  
  
    @Override  
    public void close() throws Exception {  
        System.out.println("MyAutoClosable closed!");  
    }  
}
```

The doIt() method is not part of the AutoClosable interface. It is there because we want to be able to do something more than just closing the object.

AutoClosable Example Continue...

```
private static void myAutoClosable() throws Exception {  
    try(MyAutoClosable myAutoClosable = new  
MyAutoClosable()){  
        myAutoClosable.doIt();  
    }  
}
```

Output:

```
MyAutoClosable doing it!  
MyAutoClosable closed!
```

Catching Multiple Exceptions

- ▶ Before Java 7 you would write something like this:

```
try {  
    // execute code that may throw 1 of the 3 exceptions below.  
} catch(SQLException e) {  
    logger.log(e);  
}  
catch(IOException e) {  
    logger.log(e);  
}  
catch(Exception e) {  
    logger.severe(e);  
}
```

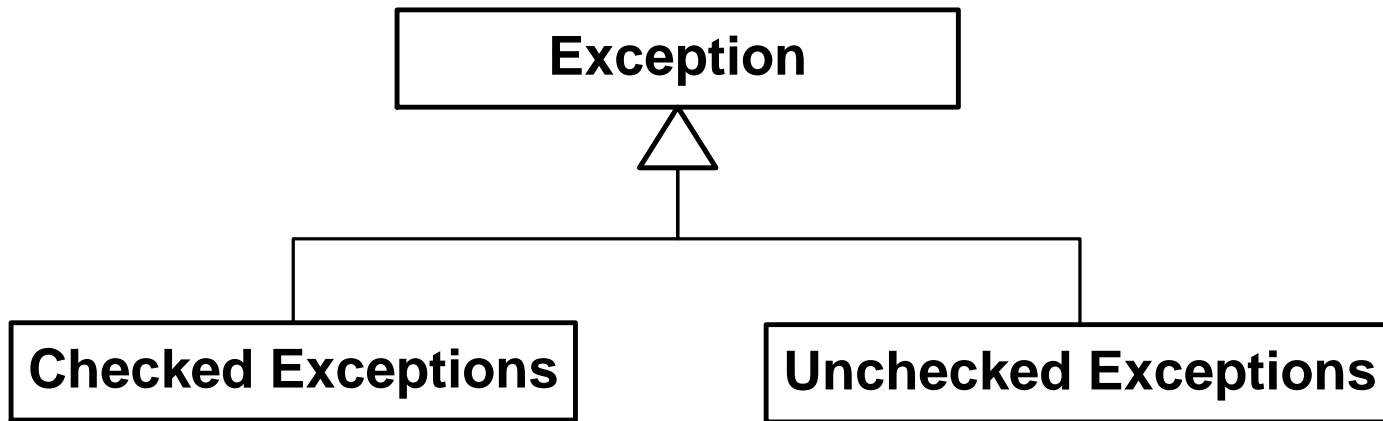
As you can see, the two exceptions `SQLException` and `IOException` are handled in the same way, but you still have to write two individual catch blocks for them.

Catching Multiple Exceptions Continue...

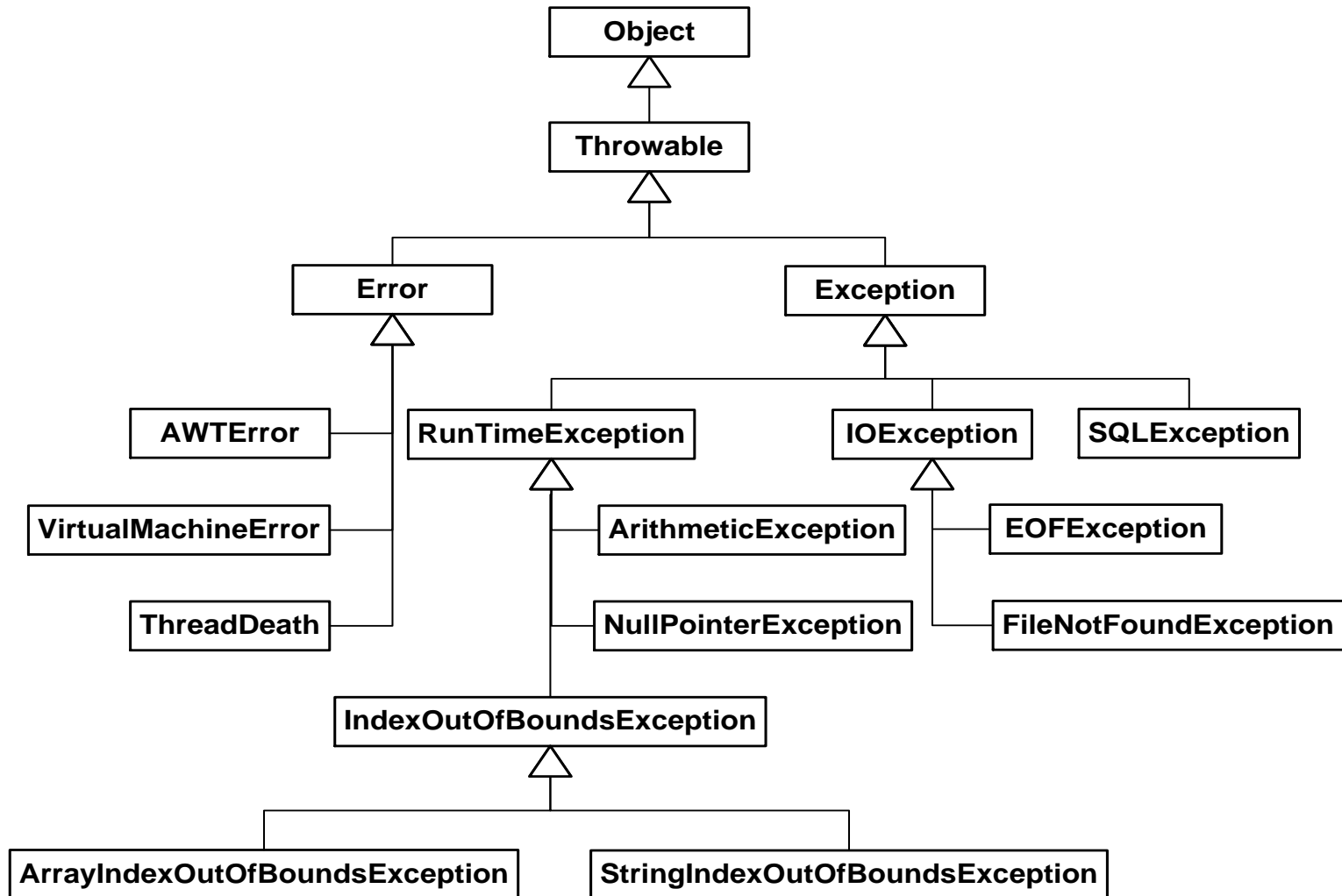
```
try {  
    // execute code that may throw 1 of the 3 exceptions  
    below.  
}  
catch(SQLException | IOException e) {  
    logger.log(e);  
}  
catch(Exception e) {  
    logger.severe(e);  
}
```

Notice how the two exception class names in the first catch block are separated by the pipe character |. The pipe character between exception class names is how you declare multiple exceptions to be caught by the same catch clause.

Types of Exceptions



Partial Exception Class Hierarchy



The try and catch Clauses

Following types are **not** allowed :

```
try {  
    // ...  
} catch (InvalidIndex e) {  
    // ...  
} catch (InvalidIndex f) {  
    // ...  
}
```

Multiple catch blocks should always have different types of exception.

```
try{  
    //...  
} catch(IOException ioe){  
    //...  
} catch(FileNotFoundException fne){  
    //...  
}
```

While writing multiple catch blocks, the super type of the exception class should always be at the last.

Using finally Clause

```
public class FinallyDemo {  
    static float dividingNos(float i, float j) {  
        try {  
            if (j == 0)  
                throw new ArithmeticException();  
            else  
                System.out.println("Printing value....");  
            return i / j;  
        } catch (ArithmeticException ae) {  
            System.out.println("in catch");  
        } finally {  
            System.out.println("This will be executed");  
        }  
        return 0;  
    }  
  
    public static void main(String[] args) {  
        // Accept two values for variables i and j .  
        System.out.println(dividingNos(i, j));  
    }  
}
```

Using finally Clause

```
import java.io.*;

class FinallyClauseDemo {

    public static void main (String [ ] args) throws
    IOException {
        InputStream in = null;
        try {
            in = new FileInputStream (args [0]);
            int total = 0;
            while (in.read ( ) != -1)
                total ++;
            System.out.println (total + " bytes.");
        }
        catch (FileNotFoundException e) {
            System.out.println ("File not found.");
        }
        finally {
            if (in != null) in.close ( );
        }
    }
}
```

The throws Clauses

```
public class ThrowsDemo {
    static float dividingNos(float i, float j)throws
    ArithmeticException{
        try{
            if(j==0)
                throw new ArithmeticException( );
            else
                System.out.println("Printing value....");
            return i/j;
        }
        finally{ System.out.println("This will be executed");}
    }
    public static void main(String[ ] args){
        try{
            // Accept two values for variables i and j .
            System.out.println(dividingNos(i,j));
        }
        catch(ArithmeticException ae){
            System.out.println("Enter number other than zero");
        }
    }
}
```

Custom Exceptions

```
class Banking {
    int balance = 10000;

    public void withdraw(float amt) throws BankException {
        if ((balance - amt) < 5000)
            throw new BankException();
        else {
            balance -= amt;
            System.out.println("Sucessfully withdrawn");
        }
    }

    public static void main(String[] args) {
        Banking b = new Banking();
        try {
            b.withdraw(6000);
        } catch (BankException e) {
            System.out.println(" Not Enough Balance..");
        }
    }
}
```

BankException Class

```
class BankException extends Exception{  
    public String toString( ){  
        return "NotEnoughBalance";  
    }  
    //some more methods  
}
```

Exception Chaining

```
import java.io.*;

public class ExceptionChainingTest {
    public static void main(String[] args){
        try{
            FileReader fr = new
FileReader("Pragati.txt");
            int i = fr.read();
        }
        catch(IOException ioe){
            NullPointerException npe =
new NullPointerException("caught");
            npe.initCause(ioe);
            throw npe;
        }
    }
}
```

Module 9. Some Useful Built-In Classes

▶ Overview


- ▶ The Object class
- ▶ The String class
- ▶ The StringBuffer class
- ▶ The StringBuilder class
- ▶ Utility classes

The java.lang.Object Class

- ▶ Some useful methods of Object class
 - ▶ public boolean equals (Object obj)
 - ▶ public int hashCode ()
 - ▶ protected Object clone () throws CloneNotSupportedException
 - ▶ public final Class getClass ()
 - ▶ public String toString ()
 - ▶ protected void finalize () throws Throwable

The equals Method

```
class MyObject{  
  
};  
class EqualsMethodVersion01{  
    public static void main(String args[]){  
        MyObject o1=new MyObject();  
        MyObject o2=o1;  
        MyObject o3=new MyObject();  
  
        if(o1.equals(o2))  
            System.out.println("o1 equals to o2");  
        if(o3.equals(o2))  
            System.out.println("o3 equals to o2");  
        else  
            System.out.println("o3 is not equals to  
o2");  
        if(o1==o3)  
            System.out.println("o1 equals to o3");  
        else
```



The equals Method

```
class SavingsAccount extends BankAccount {  
    private float interestRate;  
  
    SavingsAccount (String name, float curBal,  
        float interestRate) {  
        this.interestRate = interestRate;  
    }  
    float getInterestRate(){  
        return interestRate;  
    }  
  
    public boolean equals (SavingsAccount s) {  
        if (getInterestRate() ==  
s.getInterestRate()){  
            return true;  
        }  
        else{  
            return false;  
        }  
    }  
}
```

The equals Method (Cont...)

```
public class EqualsMethodVersion02 {  
    public static void main (String [] args ) {  
        SavingsAccount sa1 = new  
        SavingsAccount ("jack", 1000.0f, 8.75f);  
        SavingsAccount sa2 = new  
        SavingsAccount ("jack", 1000.0f, 8.75f);  
        if (sa1.equals (sa2)) {  
            System.out.println ("sa1 and sa2 are  
equal");  
        }  
        else {  
            System.out.println ("sa1 and sa2 are  
not equal");  
        }  
    }  
}
```

Strings

- ▶ To construct a new String with the value "".
- ▶ **String ()**
- ▶ To construct a new String that is a copy of the specified String object value
- ▶ **String (String value)**
- ▶ To return the length of the string.
- ▶ **int length ()**
- ▶ To return the char at the specified position.
- ▶ **char charAt (int position)**

Strings (Cont...)

```
class StringDemo1 {  
    public static void main(String[] args) {  
        String str = "abc";  
        String str1 = "abc";  
        String str2 = new String("abc");  
        String str3 = "xyz";  
        // immutability  
        System.out.println(str.concat("def"));  
        System.out.println(str);  
  
        // using == operator  
        System.out.println(" str == str1 : " +  
            (str == str1));  
        System.out.println("str == str2 : " +  
            (str == str2));  
        System.out.println(" str == str3 : " +  
            (str == str3));
```

```
        // using equals() method
```

```
        System.out.println(" str equals(str1) : "
```

Searching in a String

<u>Method</u>	<u>Returns index of...</u>
<code>int indexOf (char ch)</code>	first position of ch
<code>int indexOf (int ch, int start)</code>	first position of ch \geq start
<code>int indexOf (String str)</code>	first position of str
<code>int indexOf (String str, int start)</code>	first position of str \geq start
<code>int lastIndexOf (char ch)</code>	last position of ch
<code>int lastIndexOf (char ch, int start)</code>	last position of ch \leq start
<code>int lastIndexOf (String str)</code>	last position of str
<code>int lastIndexOf (String str, int start)</code>	last position of str \leq start

Methods in a String

<u>Method</u>	<u>Returns index of...</u>
boolean equalsIgnoreCase()	Compares this String to another String, ignoring case considerations.
int compareTo(String str)	Compares two strings lexicographically.
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String split(String regex)	Splits this string around matches of the given regular expression.
String substring(int beginIndex)	Returns a new string that is a substring of this string.



String Conversions

<u>Method</u>	<u>Returns index of...</u>
String toUpperCase()	Converts all of the characters in this String to upper case using the rules of the default.
String toLowerCase()	Converts all of the characters in this String to lower case using the rules of the default locale.
String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
String valueOf(boolean b)	Returns the string representation of the boolean argument.
String valueOf(char c)	Returns the string representation of the char argument.
int hashCode ()	Returns a hash code for this string.

StringBuffer Class

```
class StringBufferAppend {  
    public static void main (String [] args){  
        int i = 16;  
        StringBuffer str = new StringBuffer ();  
        str.append ("Square root of ").append  
(i).append (" is ");  
        str.append(Math.sqrt (i));  
        System.out.println (str);  
    }  
}
```

StringBuffer Class

Method	Returns index of...
StringBuffer append(String str)	Appends the specified string to this character sequence.
int capacity()	Returns the current capacity.
Char charAt(int index)	Returns the char value in this sequence at the specified index.
StringBuffer delete(int start, int end)	Removes the characters in a substring of this sequence.
StringBuffer deleteCharAt(int index)	Removes the char at the specified position in this sequence.
StringBuffer reverse()	Causes this character sequence to be replaced by the reverse of the sequence.
StringBuffer trimToSize()	Attempts to reduce storage used for the character sequence.

String Builder

```
class StringBuilderDemo {  
    public static void main (String [] args) {  
        int i = 16;  
        StringBuilder str = new StringBuilder ();  
        str.append ("Square root of ").append  
(i).append (" is ");  
        str.append(Math.sqrt (i));  
        System.out.println (str);  
    }  
}
```

Module 10. Collections and Generics Framework

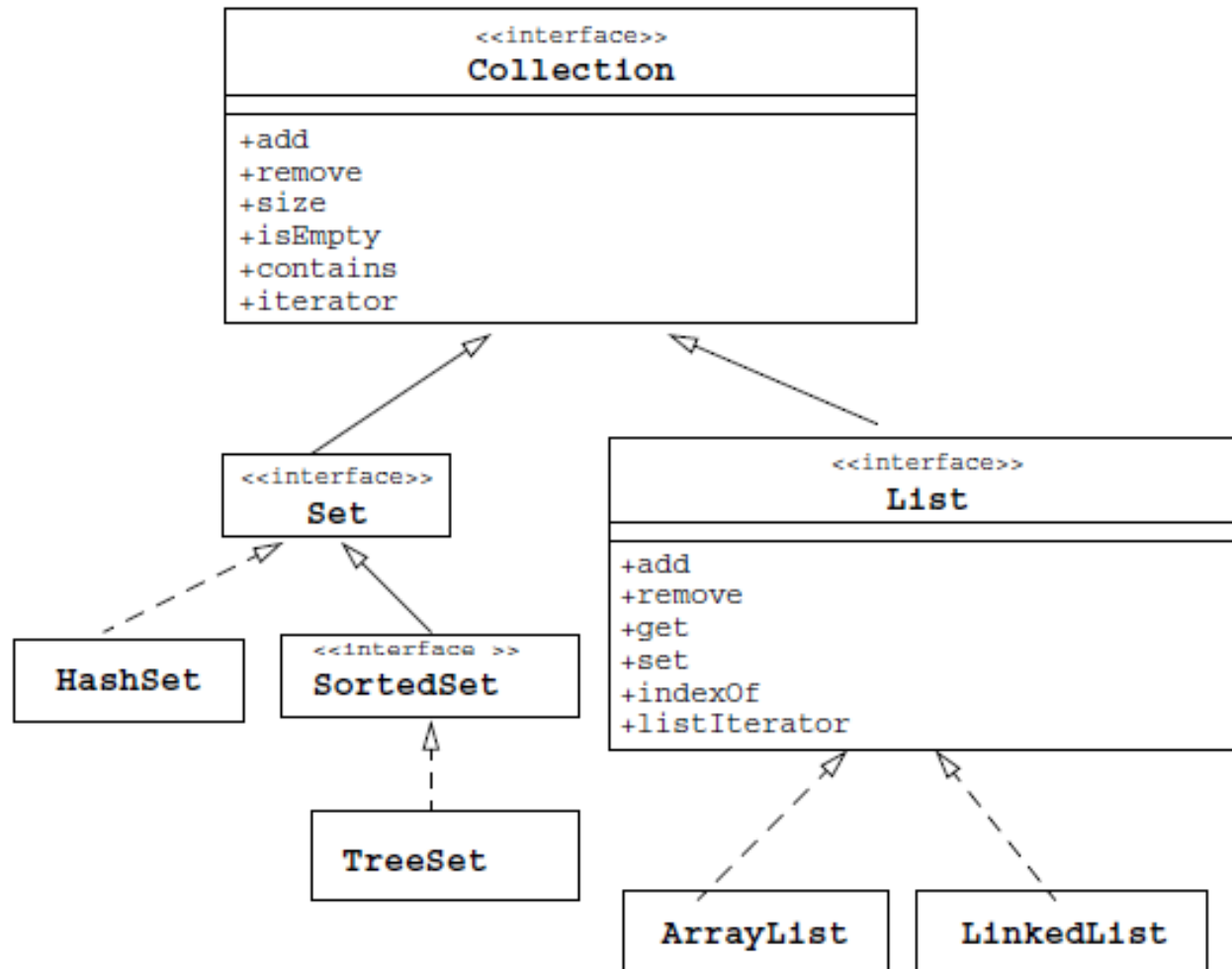
▶ Objectives

- ▶ Describe the Collections
- ▶ Describe the general purpose implementations of the core interfaces in the Collections framework
- ▶ Examine the Map interface
- ▶ Use generic collections
- ▶ Use type parameters in generic classes
- ▶ Write a program to iterate over a collection

The Collections API

- ▶ A collection is a single object managing a group of objects known as its elements.
- ▶ The Collections API contains interfaces that group objects as one of the following:
 - ▶ **Collection** – A group of objects called elements; implementations determine whether there is specific ordering and whether duplicates are permitted.
 - ▶ **Set** – An unordered collection; no duplicates are permitted.
 - ▶ **List** – An ordered collection; duplicates are permitted.

The Collections API



Collection Implementations

- ▶ There are several general purpose implementations of the core interfaces (Set, List, Deque and Map)

	HashTable	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList			
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

A Set Example

```
1  import java.util.*;
2  public class SetExample {
3      public static void main(String[] args) {
4          Set set = new HashSet();
5          set.add("one");
6          set.add("second");
7          set.add("3rd");
8          set.add(new Integer(4));
9          set.add(new Float(5.0F));
10         set.add("second");           // duplicate, not added
11         set.add(new Integer(4));     // duplicate, not added
12         System.out.println(set);
13     }
14 }
```

[one, second, 5.0, 3rd, 4]

A List Example

```
1  import java.util.*
2  public class ListExample {
3      public static void main(String[] args) {
4          List list = new ArrayList();
5          list.add("one");
6          list.add("second");
7          list.add("3rd");
8          list.add(new Integer(4));
9          list.add(new Float(5.0F));
10         list.add("second");           // duplicate, is added
11         list.add(new Integer(4));     // duplicate, is added
12         System.out.println(list);
13     }
14 }
```

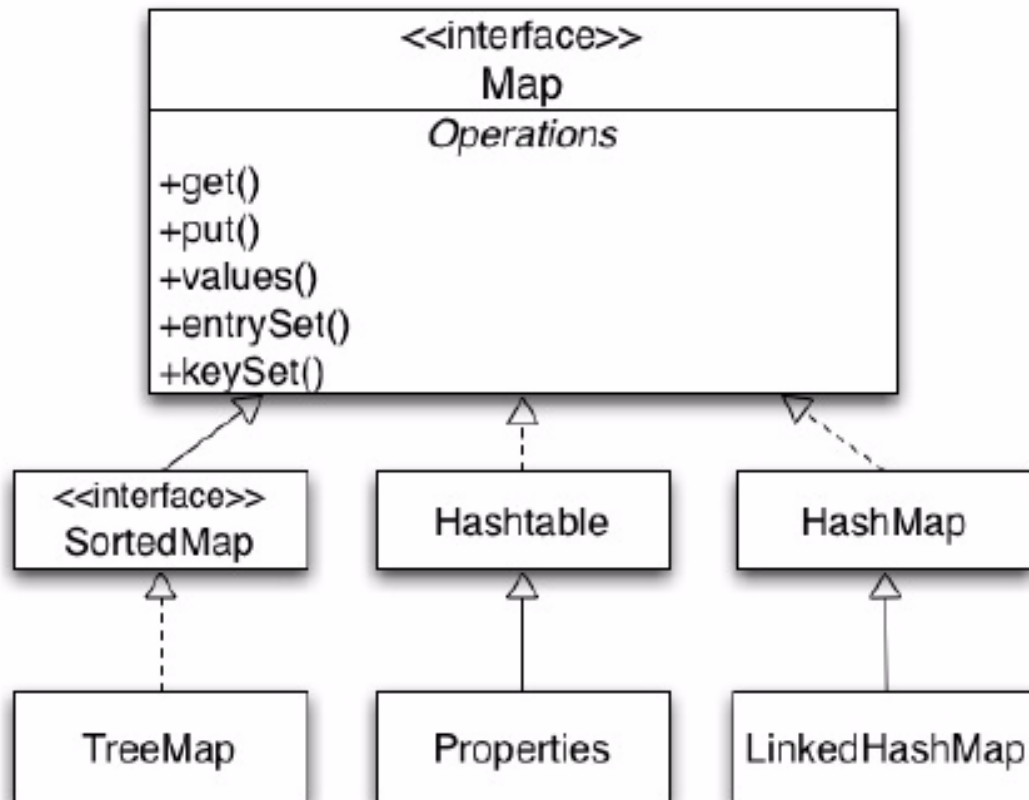
The output generated from this program is:

[one, second, 3rd, 4, 5.0, second, 4]

The Map Interface

- ▶ Maps are sometimes called associative arrays
- ▶ A Map object describes mappings from keys to values:
 - ▶ Duplicate keys are not allowed
 - ▶ One-to-many mappings from keys to values is not permitted
- ▶ The contents of the Map interface can be viewed and manipulated as collections
 - ▶ **entrySet** – Returns a Set of all the key-value pairs.
 - ▶ **keySet** – Returns a Set of all the keys in the map.
 - ▶ **values** – Returns a Collection of all values in the map.

The Map Interface API



A Map Example

```
1  import java.util.*;
2  public class MapExample {
3      public static void main(String args[]) {
4          Map map = new HashMap();
5          map.put("one", "1st");
6          map.put("second", new Integer(2));
7          map.put("third", "3rd");
8          // Overwrites the previous assignment
9          map.put("third", "III");
10         // Returns set view of keys
11         Set set1 = map.keySet();
12         // Returns Collection view of values
13         Collection collection = map.values();
14         // Returns set view of key value mappings
15         Set set2 = map.entrySet();
16         System.out.println(set1 + "\n" + collection + "\n" + set2);
17     }
18 }
```

[second, one, third] [2, 1st, III] [second=2, one=1st,
third=III]

Generics

- ▶ Generics are described as follows:
 - ▶ Provide compile-time type safety
 - ▶ Eliminate the need for casts
 - ▶ Provide the ability to create compiler-checked homogeneous collections
- ▶ Using non-generic collections:

```
ArrayList list = new ArrayList();  
list.add(0, new Integer(42));  
int total = ((Integer)list.get(0)).intValue();
```
- ▶ Using generic collections:

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = list.get(0).intValue();
```

Generic Set Example

```
1  import java.util.*;
2  public class GenSetExample {
3      public static void main(String[] args) {
4          Set<String> set = new HashSet<String>();
5          set.add("one");
6          set.add("second");
7          set.add("3rd");
8          // This line generates compile error
9          set.add(new Integer(4));
10         set.add("second");
11         // Duplicate, not added
12         System.out.println(set);
13     }
14 }
```

Generic Map Example

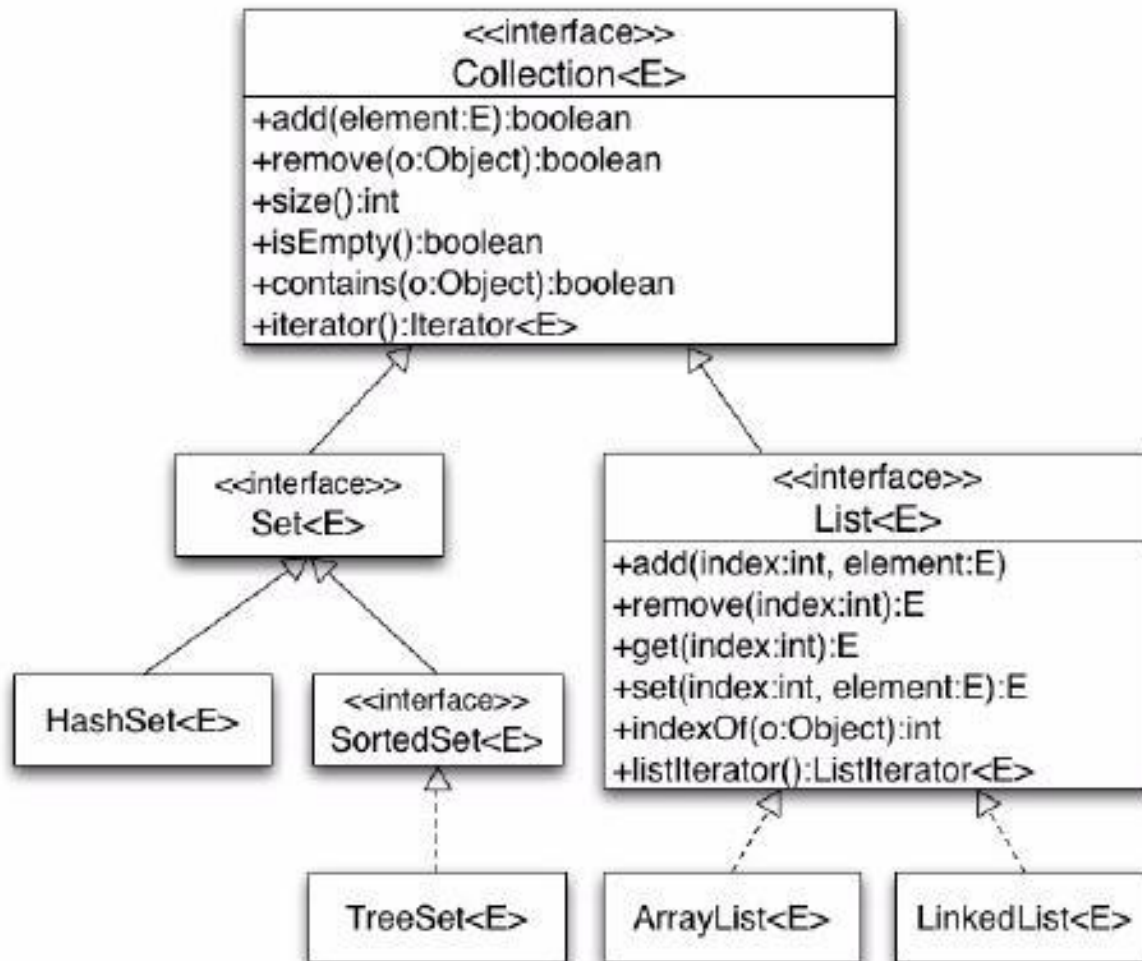
```
1  import java.util.*;
2
3  public class MapPlayerRepository {
4      HashMap<String, String> players;
5
6      public MapPlayerRepository() {
7          players = new HashMap<String, String> ();
8      }
9
10     public String get(String position) {
11         String player = players.get(position);
12         return player;
13     }
14
15     public void put(String position, String name) {
16         players.put(position, name);
17     }
```

Generics: Examining Type Parameters

- Shows how to use type parameters

Category	Non Generic Class	Generic Class
Class declaration	<code>public class ArrayList extends AbstractList implements List</code>	<code>public class ArrayList<E> extends AbstractList<E> implements List <E></code>
Constructor declaration	<code>public ArrayList (int capacity);</code>	<code>public ArrayList<E> (int capacity);</code>
Method declaration	<code>public void add((Object o) public Object get(int index)</code>	<code>public void add(E o) public E get(int index)</code>
Variable declaration examples	<code>ArrayList list1; ArrayList list2;</code>	<code>ArrayList <String> list1; ArrayList <Date> list2;</code>
Instance declaration examples	<code>list1 = new ArrayList(10); list2 = new ArrayList(10);</code>	<code>list1= new ArrayList<String> (10); list2= new ArrayList<Date> (10);</code>

Generic Collections API



Wild Card

- ▶ Consider the problem of writing a routine that prints out all the elements in a collection.
- ▶ Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

Wild Card

- ▶ And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- ▶ The problem is that this new version is much less useful than the old one.
- ▶ Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`, which, as we've just demonstrated, is **not** a supertype of all kinds of collections!

Wild Card

- ▶ What is the supertype of all kinds of collections?
 - ▶ It's written `Collection<?>` (pronounced "collection of unknown"), that is, a collection whose element type matches anything

```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

The Type-Safety Guarantee

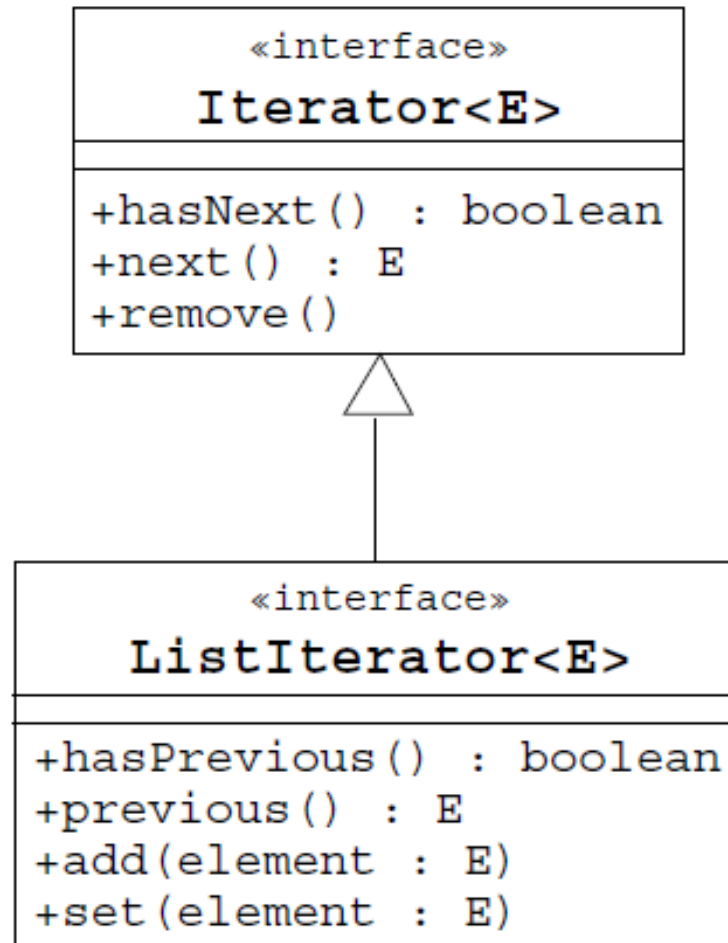
```
1  public class TestTypeSafety {
2
3      public static void main(String[] args) {
4          List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
5
6          lc.add(new CheckingAccount("Fred")); // OK
7          lc.add(new SavingsAccount("Fred")); // Compile error!
8
9          // therefore...
10         CheckingAccount ca = lc.get(0);      // Safe, no cast required
11     }
12 }
```

Iterators

- ▶ Iteration is the process of retrieving every element in a collection.
- ▶ The basic Iterator interface allows you to scan forward through any collection.
- ▶ A List object supports the ListIterator, which allows you to scan the list backwards and insert or modify elements.

```
1  List<Student> list = new ArrayList<Student>();  
2  // add some elements  
3  Iterator<Student> elements = list.iterator();  
4  while (elements.hasNext()) {  
5      System.out.println(elements.next());  
6  }
```

Generic Iterator Interfaces



Module 11. Threads

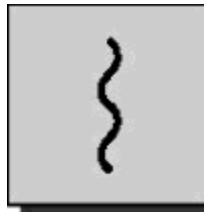
▶ Objective

- ▶ What is a Thread?
- ▶ Multitasking & Multithreading
- ▶ Different ways of creating a Thread
 - ▶ Thread class
 - ▶ Runnable Interface
 - ▶ Difference between Thread & Runnable
- ▶ Thread Attributes
- ▶ Thread Priority
- ▶ Executor Framework
- ▶ Semaphores
- ▶ Latches
- ▶ FutureTask

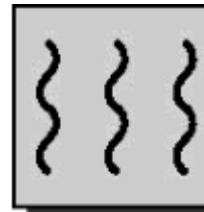


What is a Thread?

- ▶ **Threading** is a technique that allows multiple activities to coexist within a single program.
- ▶ Each Activity that can run concurrently within a program is referred to as a thread.



One process one thread



One process multiple threads

Process & Thread

- ▶ Process is an instance of a program that has its own execution environment and memory space
- ▶ Threads are independent, concurrent paths of execution through a program, and each thread has its own stack, program counter and local variables.
- ▶ A thread must be a part of the process for it to exist and execute
- ▶ A process can support multiple threads, which appear to execute simultaneously and asynchronously to each other

Process & Thread

- ▶ Multiple threads within a process share the same memory address space, thereby sharing the same variables and objects of a process within which they exist.
- ▶ A process remains running until all of the non-daemon threads are done executing.
- ▶ Threads are referred to as light-weight process since it requires fewer resources to create a thread than creating a new process.

Importance of Thread

- ▶ Every java program uses at least one thread called the **main** thread
 - ▶ Upon execution of a java program, JVM creates a main thread and executes the main() within that thread.
- ▶ Java Programs like AWT, Swing, Applets, Servlets, RMI, EJB etc use the concept of threads.
- ▶ Threads help us do the following in an easier and elegant way.
 - ▶ Make the User Interface more responsive
 - ▶ Simplify Modeling
 - ▶ Perform Asynchronous and background Processing

Importance of Thread

- ▶ **Make the User Interface more responsive**
 - ▶ Applet responds to user events at the same time draws random shapes on the display
- ▶ **Modularize the composite task**
 - ▶ The timer that displays the last updated time of a document can be put in a thread instead of having the main() to do it in a loop. This helps the main thread to focus on other important tasks to be done.
- ▶ **Perform Asynchronous and background Processing**
 - ▶ Word processor has to format the user text at the same time spell check the contents. The spell checker can be put in a thread that does the spell check at the background rather than the main thread doing it by intervening the format job and make the user wait till spell check is complete.

Different ways to create threads

- ▶ A thread can be created using
 - ▶ Extending Thread class
 - ▶ Implementing Runnable interface

Multitasking & Multithreading

- ▶ **Multitasking** is the ability to run multiple application programs concurrently.
 - ▶ In Multitasking the operating system can switch between tasks by resuming the task from the suspended state.
- ▶ **Multithreading** is the ability to run multiple parts within the same program concurrently.
 - ▶ Multithreading allows servicing more than one request to the same program by having one instance of the program in memory but spawning multiple threads to service each request.

Different ways of creating a Thread

- ▶ Extending `java.lang.Thread` class
- ▶ Implementing `java.lang.Runnable` interface

Using Thread class

```
public class Thread1 extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++)  
            System.out.println("Thread1 doing its Job!!!");  
    }  
}
```

Thread 1

Thread 1

```
public class Thread2 extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++)  
            System.out.println("Thread2 doing its Job!!!");  
    }  
}
```

Using Thread class

```
public class ThreadExecutor {  
    public static void main(String[] args) {  
        Thread1 t1=new Thread1();  
        Thread2 t2=new Thread2();  
        t1.start();  
        t2.start();  
    }  
}
```

```
Thread1 doing its Job!!!  
Thread1 doing its Job!!!  
Thread1 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread1 doing its Job!!!  
Thread1 doing its Job!!!
```



Order of output is
non-deterministic in
a thread program

Difference between Thread & Runnable

- ▶ Extending Thread class enforces is-a-type of relationship making the class Thread1 itself as a Thread.
 - ▶ Thread1 is a type of Thread that should be more focused towards threading related tasks
- ▶ Implementing Runnable interface enforces Thread2 to bind to a contract that Thread2 should also work like thread
 - ▶ Purpose of the Thread2 class can be different apart from that it should also behave like thread

Thread Attributes

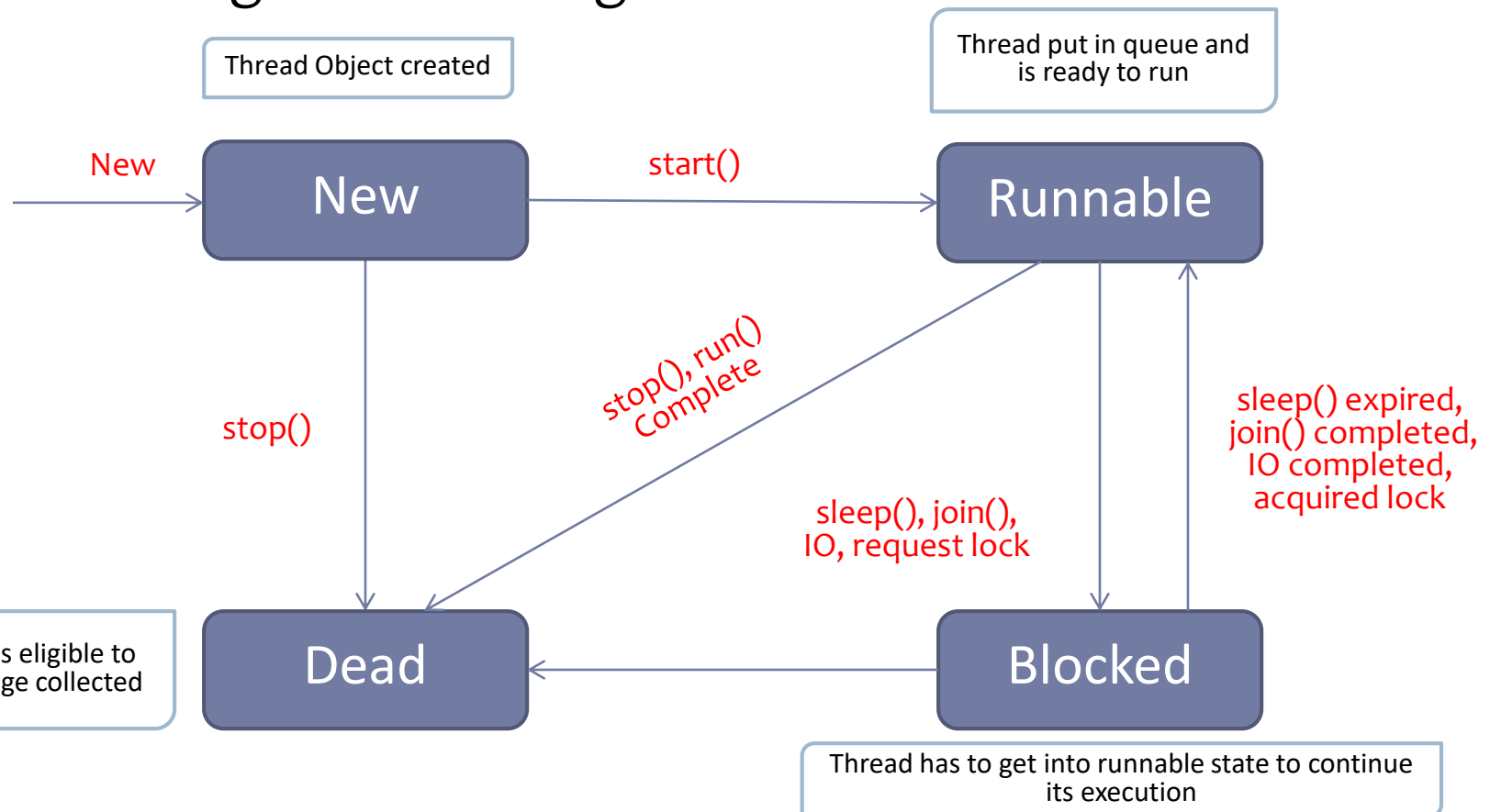
- ▶ Aspects of Threading
 - ▶ Thread Body
 - ▶ Thread State
 - ▶ Thread Priority

Thread Body

- ▶ The core part of the thread is the Thread body that is defined via the run() method
- ▶ All actions the thread is expected to do has to be provided in the run() method
- ▶ The thread body can be defined in 2 ways
 - ▶ Extending Thread class and override the Thread's run() method
 - ▶ Implementing the Runnable interface and implementing the run() method

Thread State

- ▶ A java thread can be possibly in any one of the following states during its life time



Thread Priority

- ▶ Multiple thread executing on a single CPU in an order is called as **Scheduling**.
- ▶ Java runtime system schedules a thread to run based on its priority.
- ▶ A java thread acquires the priority of a thread from the thread that created it although we can modify this priority any time after creation.
- ▶ Thread priorities range from low to high as indicated by `MIN_PRIORITY-1` and `MAX_PRIORITY-10` (public static final constants declared in Thread class). Higher the priority, greater the chance for it to run.

Thread Priority

- ▶ We can access and modify the thread priorities by `getPriority()` and `setPriority(int)` methods in `Thread` class
- ▶ A thread runs until
 - ▶ A higher priority thread becomes `Runnable`
 - ▶ Running thread issues a `yield` method() or its `run()` is complete
 - ▶ On time-slicing systems, the time allotted for that thread expires
- ▶ The thread scheduler in certain situations can choose lower priority thread over a higher priority thread for execution(e.g to avoid starvation).Therefore program logic should not rely on thread priority.

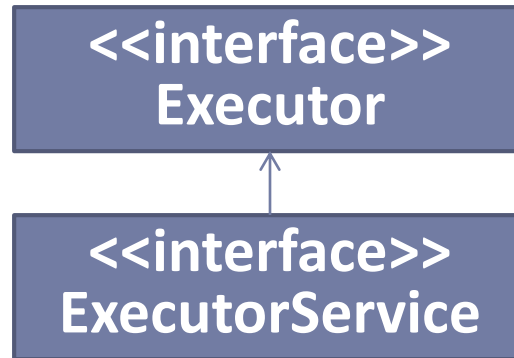
Executor Framework

- ▶ **java.util.concurrent** package was added in java 5. This package provides API for developing multithreaded applications in Java in an easier way.
- ▶ **java.util.concurrent.Executor** is an object that executes Runnable threads.
- ▶ Executor framework *decouples* the task of thread submission, executing its run() method ,thread scheduling, etc.

Executor Framework

- ▶ Executor framework has to be used as follows
 - ▶ Obtain the executor object via a factory method.
 - ▶ Factory is the standard name given for method that returns an object of a class instead of we creating the object explicitly.
 - ▶ `java.util.concurrent.Executors` class provides many static factory methods for this purpose.
- ▶ Pass the target thread to its execute method

Executor Framework



```
ExecutorService  
executor=java.util.concurrent.Executors.newFixedThreadPool(5);  
executor.execute(new Thread1());  
executor.execute(new Thread2());
```

```
Thread1 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread2 doing its Job!!!  
Thread1 doing its Job!!!  
Thread1 doing its Job!!!  
Thread1 doing its Job!!!  
Thread1 doing its Job!!!
```

Order of output is
non-deterministic
in a thread
program

pool with fixed
number of
threads sharing
an unbounded
queue

Semaphores

- ▶ Semaphores are often used to restrict the number of threads that can access a resource at a time.
- ▶ Semaphore is a class in `java.util.concurrent` package. Semaphore maintains a set of permits it is initialized with.
 - ▶ Permits are nothing but a count indicating the maximum number of threads that can access a particular resource
- ▶ Semaphore has 2 methods
 - ▶ `acquire()`
 - ▶ The thread calling the `acquire()` method will obtain a permit to access a resource, till the maximum number of permits are exhausted
 - ▶ `release()`
 - ▶ Thread calling the `release()` will release the permit for that resource, so that other threads in the queue can issue the `acquire()` call and get the permit

Semaphores

- ▶ Semaphores are normally used to guard a critical section of code. Therefore the thread acquiring the lock has to release it

```
Semaphore semaphore = new Semaphore(3);  
    semaphore.acquire();  
    //access critical section of code  
    semaphore.release();
```

Latches

- ▶ A latch is a synchronizer that can delay the progress of threads until it reaches its terminal state
- ▶ A latch acts as a gate: until the latch reaches the terminal state the gate is closed and no thread can pass, and in the terminal state the gate opens, allowing all threads to pass.
- ▶ Once the latch reaches the terminal state, it can not change state again, so it remains open forever

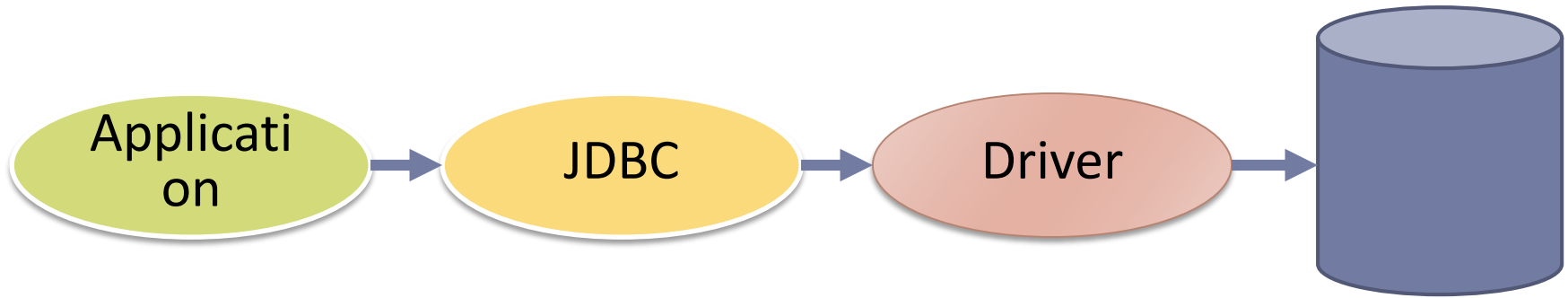
FutureTask

- ▶ FutureTask also acts like a latch.
- ▶ FutureTask implements Future, which describes an abstract result bearing computation
- ▶ A computation represented by a FutureTask is implemented with a Callable, the result bearing equivalent of Runnable, and can be in one of three states: waiting to run, running ,or completed.
- ▶ Completion subsumes all the ways a computation can complete, including normal completion, cancellation, and exception.
- ▶ Once a FutureTask enters the completed state, it stays in that state forever.

Module 12. Building Database Applications with JDBC

- ▶ Objectives
- ▶ Understand JDBC Architecture
- ▶ Understand Drivers
 - ▶ What are they used for
 - ▶ Different types
- ▶ Define the layout of the JDBC API
- ▶ Connect to a database by using a JDBC driver
- ▶ Submit queries and get results from the database
- ▶ Use the JDBC 4.1 RowSetProvider and RowSetFactory
- ▶ Use a Data Access Object Pattern to decouple data and business methods

JDBC Architecture



1. Java code calls JDBC library
2. JDBC loads a driver
3. Driver talks to a particular database

Can have more than one driver -> more than one database
Ideal: can change database engines without changing any application code

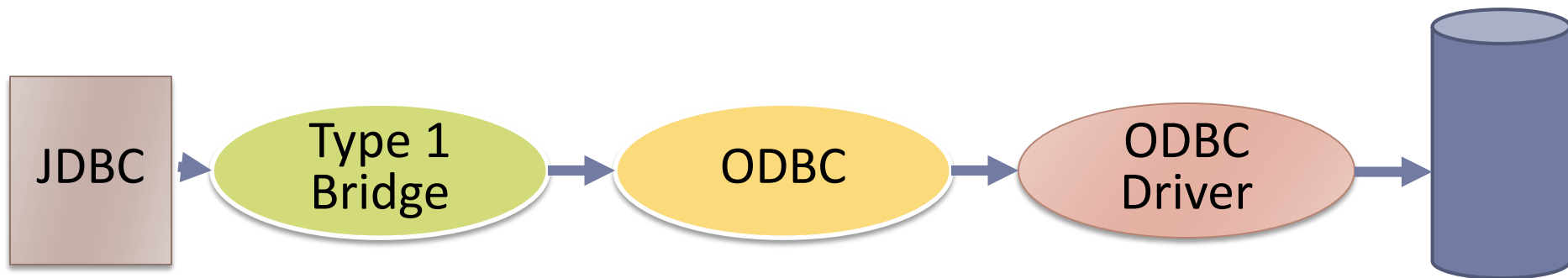
JDBC Drivers

- ▶ Type I: “Bridge”
- ▶ Type II: “Native”
- ▶ Type III: “Middleware”
- ▶ Type IV: “Pure”

Type I Drivers

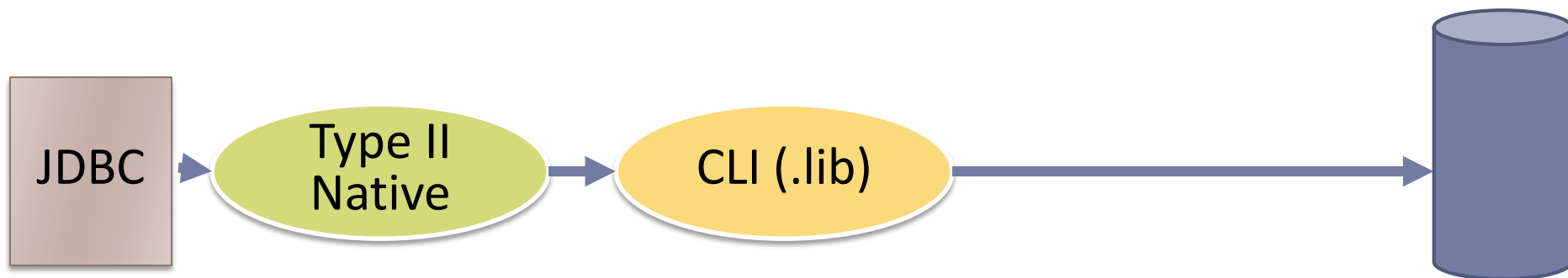
▶ **JDBC-ODBC bridge :**

- ▶ Provides JDBC API access via one or more ODBC drivers.
- ▶ ODBC native code and native database client code must be loaded on each client machine.
- ▶ Appropriate when automatic installation and downloading of a Java technology application is not important.



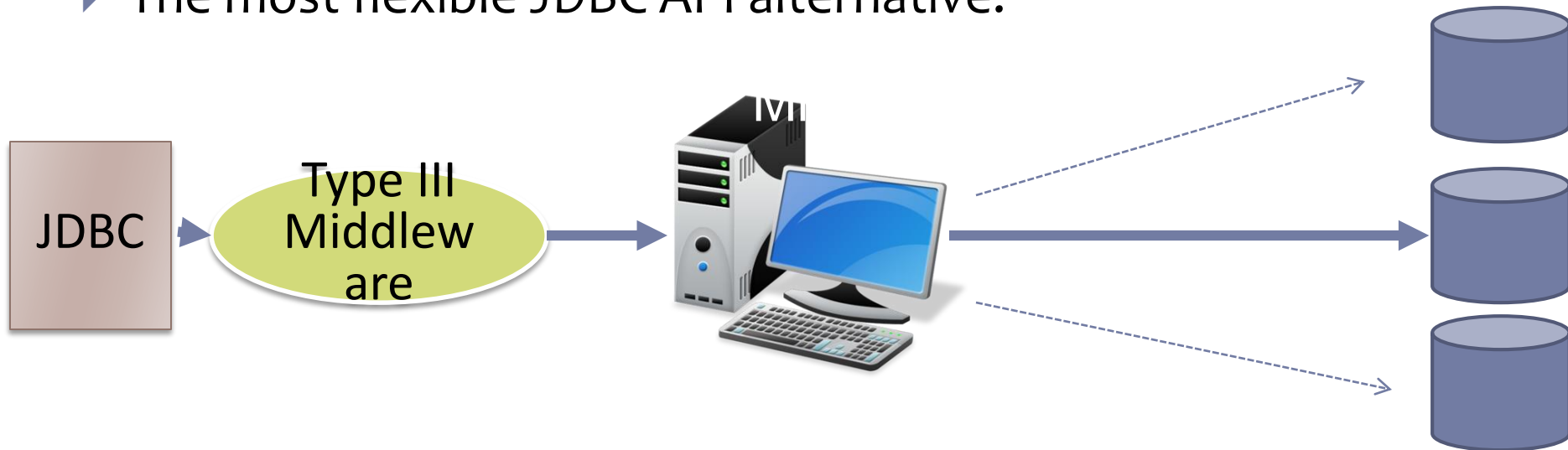
Type II Drivers

- ▶ *Native-API partly Java technology-enabled driver.*
- ▶ • Converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS.
- ▶ • Requires that some binary code be loaded on each client machine.



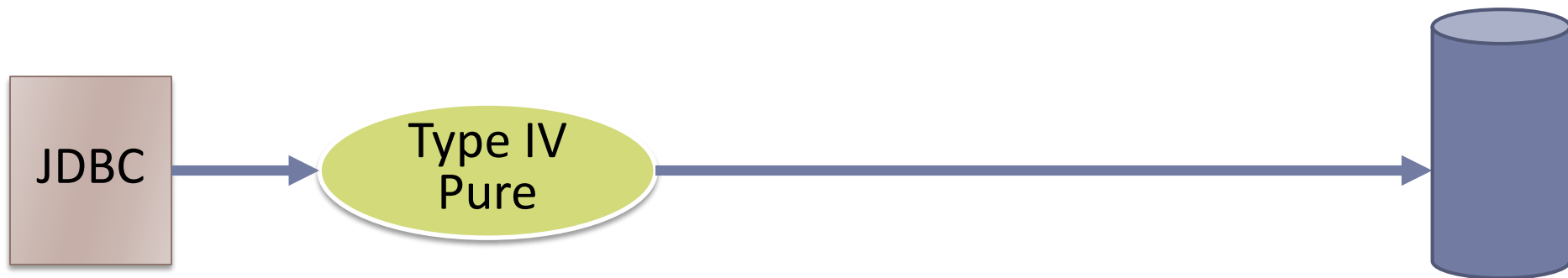
Type III Drivers

- ▶ Net-protocol fully Java technology-enabled driver .
- ▶ Translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server.
- ▶ The server middleware is able to connect all of its Java technology-based clients to many different databases.
- ▶ The specific protocol used depends on the vendor.
- ▶ The most flexible JDBC API alternative.

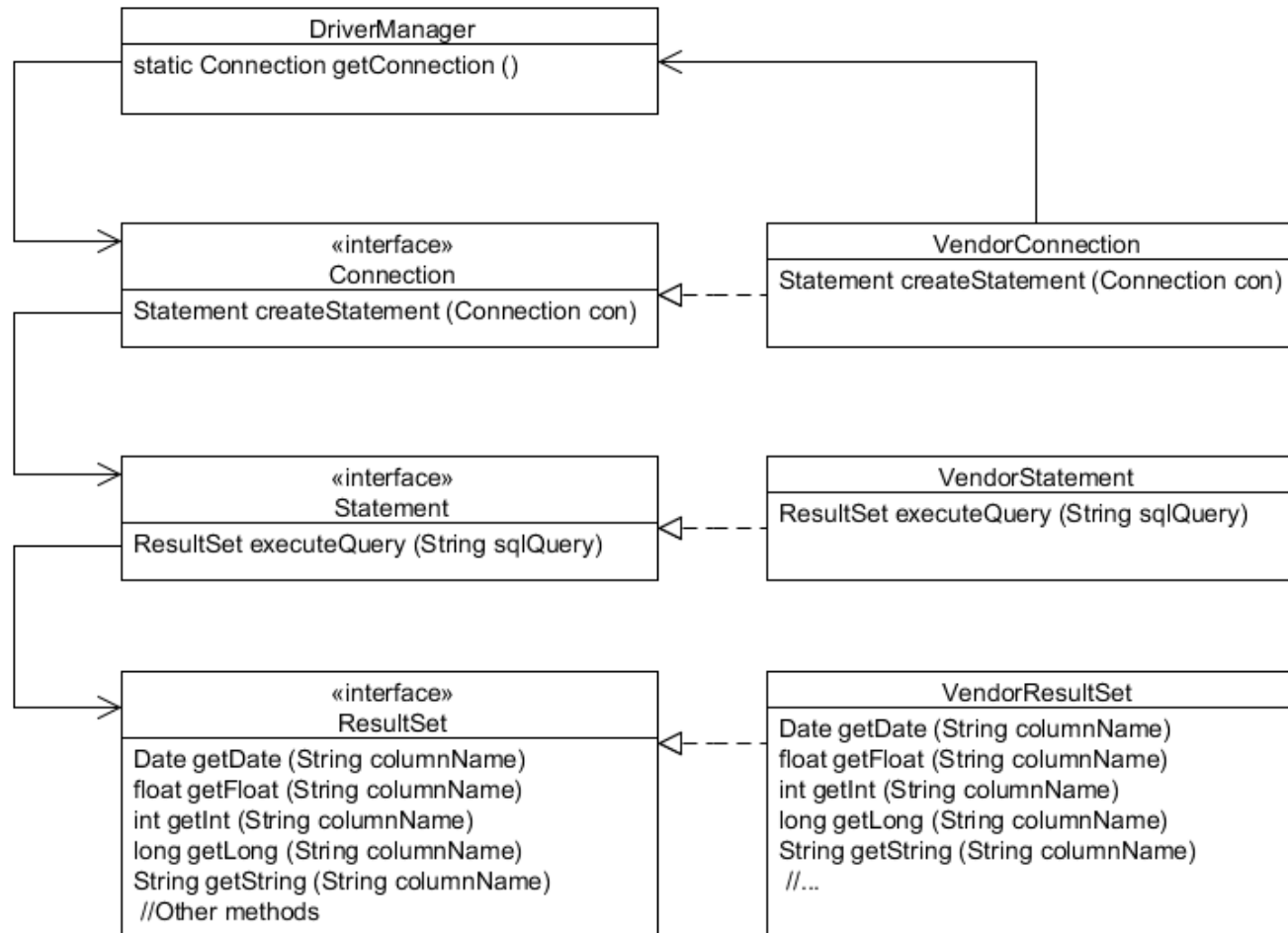


Type IV Drivers

- ▶ *Native-protocol fully Java technology-enabled driver.*
- ▶ Converts JDBC technology calls into the network protocol used by DBMSs directly.
- ▶ Allows a direct call from the client machine to the DBMS.
- ▶ Several database vendors have these in progress.



Using JDBC API



Using a Vendor's Driver Class

- ▶ **The DriverManager class is used to get an instance of a Connection object, using the JDBC driver named in the JDBC URL:**

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";  
Connection con = DriverManager.getConnection(url);
```

- ▶ **The URL syntax for a JDBC driver is:**

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]  
Each vendor can implement their own subprotocol.
```

- ▶ **The URL syntax for an Oracle Thin driver is:**

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```


Key JDBC API Components

- ▶ **Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:**
 - ▶ **java.sql.Connection** – A connection that represents the session between your Java application and the database
Connection con = DriverManager.getConnection(url, username, password);
 - ▶ **java.sql.Statement** – An object used to execute a static SQL statement and return the result
Statement stmt = con.createStatement();
 - ▶ **java.sql.ResultSet** – An object representing a database result set
*String query = "SELECT * FROM Employee";*
ResultSet rs = stmt.executeQuery(query);

Using a ResultSet Object

- ▶ The result of executing a query statement is a set of rows returned in a `java.sql.ResultSet` object.

```
String query = "SELECT * FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        int empID = rs.getInt("ID");
        String first = rs.getString("FIRSTNAME");
        String last = rs.getString("LASTNAME");
        Date birthDate = rs.getDate("BIRTHDATE");
        float salary = rs.getFloat("SALARY");
        // other code....
    }
```



Writing Portable JDBC Code

- ▶ The JDBC driver provides a programmatic "insulating" layer between your Java application and the database. However, you also need to consider SQL syntax and semantics when writing database applications.
- ▶ Most databases support a standard set of SQL syntax and semantics described by the American National Standards Institute (ANSI) SQL-92 Entry-level specification.
- ▶ You can programmatically check for support for this specification from your driver:

```
        Connection con =  
        DriverManager.getConnection(url, username,  
                                    password);  
        DatabaseMetaData dbm = con.getMetaData();  
        if (dbm.supportsANSI92EntrySQL()) {  
            // Support for Entry-level SQL-92 standard  
        }
```

Putting It All Together

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;

public class SimpleJDBCTest {
public static void main(String[] args) {
    String url =
"jdbc:derby://localhost:1521/EmployeeDB";
    String username = "Scott";
    String password = "tiger";
    String query = "SELECT * FROM Employee";
    try (Connection con =
        DriverManager.getConnection(url,
            username,
            password)) {
        Statement stmt = con.createStatement();206
```

Putting It All Together

```
        while (rs.next()) {
            int empID = rs.getInt("ID");
            String first =
                rs.getString("FirstName");
            String last =
                rs.getString("LastName");
            Date birthDate =
                rs.getDate("BirthDate");
            float salary =
                rs.getFloat("Salary");
            System.out.println("Employee ID: "
                + empID + "\n"
                + "Employee Name: " + first + " "
                + last + "\n"
                + "Birth Date: " + birthDate +
```

The *SQLException* Class

- ▶ An `SQLException` is thrown from errors that occur in one of the following types of actions: driver methods, methods that access the database, or attempts to get a connection to the database.
- ▶ The `SQLException` class also implements `Iterable`—exceptions can be chained together and returned as a single object.
- ▶ `SQLException` is thrown if the database connection cannot be made due to incorrect username or password information, or simply the database is offline.
- ▶ `SQLException` can also result by attempting to access a column name that is not part of the SQL query.
- ▶ `SQLException` is also subclassed, providing granularity of the actual exception thrown.

Closing JDBC Objects

- ▶ Closing a Connection object will automatically close any Statement objects created with this Connection.
- ▶ Closing a Statement object will close and invalidate any instances of ResultSet created by the Statement object.
- ▶ Resources held by the ResultSet, may not be released until garbage is collected, so it is a good practice to explicitly close ResultSet objects when they are no longer needed.
- ▶ When the close() method on ResultSet is executed, external resources are released.
- ▶ ResultSet objects are also implicitly closed when an associated Statement object is re-executed.

The *try*-with-resources Construct

- ▶ Given the following try-with-resources statement:

```
try (  
    Connection con =  
    DriverManager.getConnection(url, username,  
                                password);  
    Statement stmt = con.createStatement();  
    ResultSet rs = stmt.executeQuery (query))}
```

- ▶ The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`
- ▶ This interface includes one method: `void close()`.
- ▶ The close method is automatically called at the end of the try block in the proper order (last declaration to first).
- ▶ Multiple closeable resources can be included in the try block, separated by semicolons.

try-with-resources: Bad Practice

- ▶ It might be tempting to write try-with-resources more compactly:

```
try (ResultSet rs =  
    DriverManager.getConnection(url, username,  
password).createStatement().executeQuery(query  
))
```

- ▶ However, only the close method of ResultSet is called, which is not a good practice.
- ▶ Always keep in mind which resources you need to close when using try-with-resources.

Writing Queries and Getting Results

- ▶ To execute SQL queries with JDBC, you need to create a SQL query wrapper object, an instance of the Statement object.

```
Statement stmt = con.createStatement();
```

Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery(query);
```

Note that there are three Statement execute methods:

Method	Returns	Used for
executeQuery(sqlString)	ResultSet	SELECT statement
executeUpdate(sqlString)	int (rows affected)	INSERT, UPDATE, DELETE, or a DDL
execute(sqlString)	boolean (true if there was a ResultSet)	Any SQL command or commands

Working with the Derby Database and JDBC

- ▶ This practice covers the following topics:
- ▶ Populating the database with data (the Employee table).
- ▶ Running SQL queries to look at the data.
- ▶ Compiling and running the sample JDBC application.

Using *PreparedStatement*

- ▶ `PreparedStatement` is a subclass of `Statement` that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000;  
PreparedStatement pStmt =  
con.prepareStatement("SELECT * FROM Employee  
WHERE Salary > ?");  
pStmt.setDouble(1, value);
```

```
ResultSet rs = pStmt.executeQuery();
```

- ▶ In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than 100,000.
- ▶ `PreparedStatement` is useful when you have SQL statements that you are going to execute multiple times.

Using CallableStatement

- ▶ A CallableStatement allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt =  
con.prepareCall("{CALL EmpAgeCount (?, ?)}");  
int age = 50; cStmt.setInt (1, age);  
ResultSet rs = cStmt.executeQuery();  
cStmt.registerOutParameter(2, Types.INTEGER);  
boolean result = cStmt.execute();  
int count = cStmt.getInt(2);  
System.out.println("There are " + count + "  
Employees over the age of " + age);
```

-

RowSet 1.1: RowSetProvider and RowSetFactory

- ▶ The JDK 7 API specification introduces the new RowSet 1.1 API. One of the new features of this API is RowSetProvider.
- ▶ `javax.sql.rowset.RowSetProvider` is used to create a RowSetFactory object:
`myRowSetFactory = RowSetProvider.newFactory();`
- ▶ The default RowSetFactory implementation is:
`com.sun.rowset.RowSetFactoryImpl`
- ▶ RowSetFactory is used to create one of the RowSet 1.1 RowSet object types.

Using RowSet 1.1 RowSetFactory

- ▶ RowSetFactory is used to create instances of RowSet implementations:

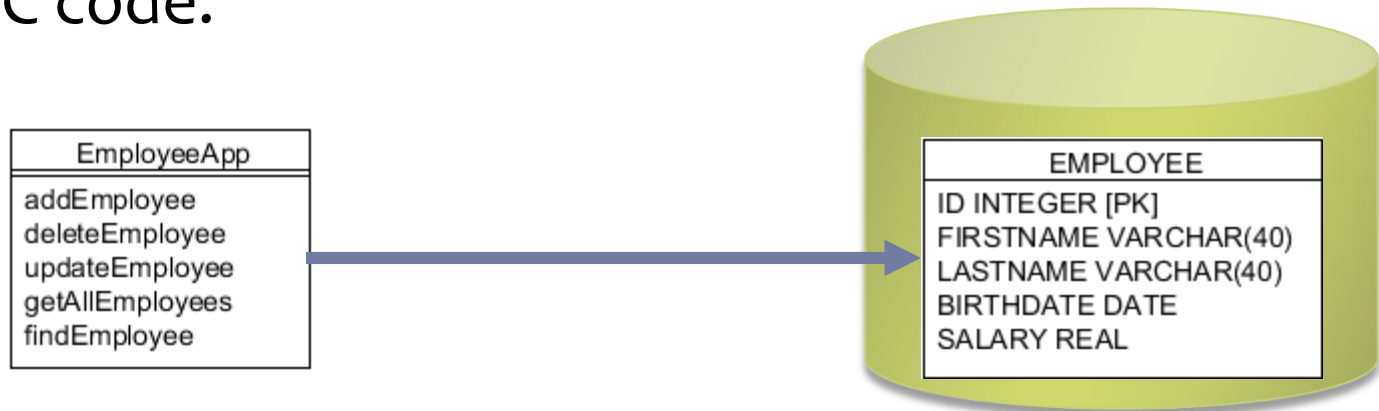
RowSet type	Provides
CachedRowSet	A container for rows of data that caches its rows in memory
FilteredRowSet	A RowSet object that provides methods for filtering support
JdbcRowSet	A wrapper around ResultSet to treat a result set as a JavaBeans component
JoinRowSet	A RowSet object that provides mechanisms for combining related data from different RowSet objects
WebRowSet	A RowSet object that supports the standard XML document format required when describing a RowSet object in XML

Example: Using JdbcRowSet

```
try (JdbcRowSet jdbcRs =
RowSetProvider.newFactory().createJdbcRowSet
    ()) {
    jdbcRs.setUrl(url);
    jdbcRs.setUsername(username);
    jdbcRs.setPassword(password);
    jdbcRs.setCommand("SELECT * FROM Employee");
    jdbcRs.execute();
    // Now just treat JDBC Row Set like a
    // ResultSet object
    while (jdbcRs.next()) {
        int empID = jdbcRs.getInt("ID");
        String first =
            jdbcRs.getString("FIRST_NAME");
        String last =
            jdbcRs.getString("LAST_NAME");
        Date birthDate =
            jdbcRs.getDate("BIRTH_DATE");
        float salary =
```

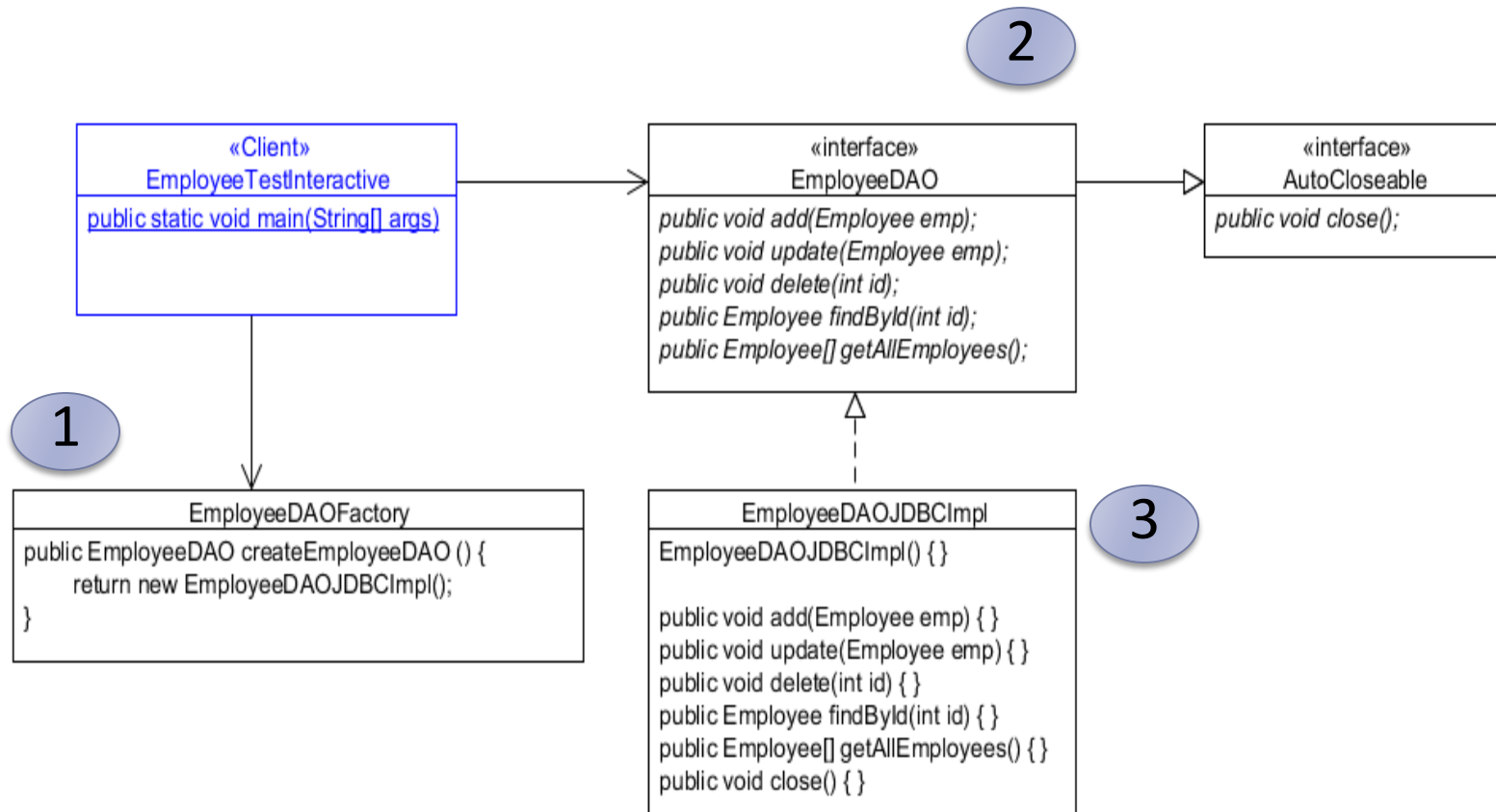

Going Further: Data Access Objects

- ▶ Consider an employee table like the one in the sample JDBC code.



- ▶ By combining the code that accesses the database with the "business" logic, the data access methods and the Employee table are tightly coupled.
- ▶ Any changes to the table (such as adding a field) will require a complete change to the application.
- ▶ Employee data is not encapsulated within the example application.

The Data Access Object Pattern



Module 13. Lambda Expressions

▶ Objective

- ▶ Introduction to Lambda
- ▶ Method References
- ▶ Functional Interfaces
- ▶ Default Methods

Introduction to Lambda

- ▶ Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming and simplifies the development a lot.

`parameter -> expression body`

Important characteristics of a lambda expression.

- ▶ **Optional type declaration** - No need to declare type of the parameter. Compiler can inference the same from value of the parameter.
- ▶ **Optional parenthesis around parameter** - No need to declare a single parameter in parenthesis. For multiple parameters, parenthesis are required.
- ▶ **Optional curly braces** - No need to use curly braces in expression body if body contains a single statement.
- ▶ **Optional return keyword** - Compiler automatically returns value if body has a single expression to return the value. Curly Braces are required to indicate that expression returns a value.

Example

- ▶ With type declaration

```
MathOperation addition = (int a, int b) -> a  
                        + b;
```

- ▶ With out type declaration

```
MathOperation subtraction = (a, b) -> a - b;
```

- ▶ With return statement along with curly braces

```
MathOperation multiplication = (int a, int b) -> { return a * b; };
```

- ▶ Without return statement and without curly braces

```
MathOperation division = (int a, int b) -> a / b;
```

Examples

▶ With parenthesis

```
GreetingService greetService1 =  
    message ->
```

```
        System.out.println("Hello " +  
▶ With parenthesis message);  
GreetingService greetService2 = (message) ->  
    System.out.println("Hello " + message);
```

Method References

- ▶ Method references helps to point to method by their name. A method reference is described using "::" symbol. A method reference can be used to point following types of methods.
 - ▶ Static methods.
 - ▶ Instance methods.
 - ▶ Constructors using new operator(`TreeSet::new`)

Method Reference Example

```
import java.util.List;
import java.util.ArrayList;
public class Java8Tester {

    public static void main(String args[]){

        List names = new ArrayList();
        names.add("Onkar");
        names.add("Sachin");
        names.add("Kiran");
        names.add("Rahul");
        names.add("Kalpesh");

        names.forEach(System.out::println);
    }
}
```

Functional Interfaces

- ▶ Functional interfaces are those interfaces which has a single functionality to exhibit. For example, a Comparable interface with a single method compareTo and is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions. Following is the list of functional interfaces defined in java.util.Function package.

Functional Interfaces Continue...

S.N.	Interface & Description
1	<code>BiConsumer<T,U></code> Represents an operation that accepts two input arguments and returns no result.
2	<code>BiFunction<T,U,R></code> Represents a function that accepts two arguments and produces a result.
3	<code>BinaryOperator<T></code> Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
4	<code>BiPredicate<T,U></code> Represents a predicate (boolean-valued function) of two arguments.
5	<code>BooleanSupplier</code> Represents a supplier of boolean-valued results.
6	<code>Consumer<T></code> Represents an operation that accepts a single input argument and returns no result.

Functional Interfaces Continue...

7	<code>DoubleBinaryOperator</code> Represents an operation upon two double-valued operands and producing a double-valued result.
8	<code>DoubleConsumer</code> Represents an operation that accepts a single double-valued argument and returns no result.
9	<code>DoubleFunction<R></code> Represents a function that accepts a double-valued argument and produces a result.
10	<code>DoublePredicate</code> Represents a predicate (boolean-valued function) of one double-valued argument.
11	<code>DoubleSupplier</code> Represents a supplier of double-valued results.
12	<code>DoubleToIntFunction</code> Represents a function that accepts a double-valued argument and produces an int-valued result.

Functional Interfaces Continue...

13	DoubleToLongFunction Represents a function that accepts a double-valued argument and produces a long-valued result.
14	DoubleUnaryOperator Represents an operation on a single double-valued operand that produces a double-valued result.
15	Function<T,R> Represents a function that accepts one argument and produces a result.
16	IntBinaryOperator Represents an operation upon two int-valued operands and producing an int-valued result.
17	IntConsumer Represents an operation that accepts a single int-valued argument and returns no result.

Functional Interfaces Continue...

18	<code>IntFunction<R></code> Represents a function that accepts an int-valued argument and produces a result.
19	<code>IntPredicate</code> Represents a predicate (boolean-valued function) of one int-valued argument.
20	<code>IntSupplier</code> Represents a supplier of int-valued results.
21	<code>IntToDoubleFunction</code> Represents a function that accepts an int-valued argument and produces a double-valued result.
22	<code>IntToLongFunction</code> Represents a function that accepts an int-valued argument and produces a long-valued result.

Example

```
public static void eval(List<Integer> list,
Predicate<Integer> predicate) {
    for(Integer n: list) {
        if(predicate.test(n)) {
            System.out.println(n + " ");
        }
    }
}
```

```
List<Integer> list = Arrays.asList(1, 2, 3, 4,
5, 6, 7, 8, 9);
```

```
// Predicate<Integer> predicate = n -> true
// n is passed as parameter to test method of
Predicate interface
// test method will always return true no
matter what value n has.
```

```
System.out.println("Print all numbers:");
```

```
//pass n as parameter
```

```
eval(list, n->true);
```

Example Continue

```
//Predicate<Integer> predicate1 = n -> n%2 == 0
// n is passed as parameter to test method of
Predicate interface
// test method will return true if n%2 comes to
be zero
System.out.println("Print even numbers:");
eval(list, n-> n%2 == 0 );
```

```
// Predicate<Integer> predicate2 = n -> n > 3
// n is passed as parameter to test method of
Predicate interface
// test method will return true if n is greater
than 3.
```

```
System.out.println("Print numbers greater than
3:");
eval(list, n-> n > 3 );
```


Default Methods

- ▶ Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage lambda expression capability of JAVA 8.

```
public interface vehicle {  
    default void print(){  
        System.out.println("I am a vehicle!");  
    }  
}
```

Multiple Defaults

- ▶ With default functions in interfaces, there is a quite possibility that a class implementing two interfaces with same default methods then how to resolve that ambiguity.

```
public interface vehicle {  
    default void print(){  
        System.out.println("I am a vehicle!");  
    }  
}  
public interface fourWheeler {  
    default void print(){  
        System.out.println("I am a four wheeler!");  
    }  
}
```

Solution

► 1st Solution

```
public class car implements vehicle,
fourWheeler {
    default void print(){
        System.out.println("I am a four wheeler
car vehicle!");
    }
}
```

► 2nd Solution

```
public class car implements vehicle,
fourWheeler {
    default void print(){
        vehicle.super.print();
    }
}
```

Static default methods

```
public interface vehicle {  
    default void print(){  
        System.out.println("I am a vehicle!");  
    }  
    static void blowHorn(){  
        System.out.println("Blowing horn!!!");  
    }  
}
```

Module 14. Streams

- ▶ Objective
 - ▶ What is Stream?
 - ▶ Generating Streams

Introduction to Stream

- ▶ Stream is a new abstract layer introduced in JAVA 8. Using stream, you can process data in a declarative way similar to SQL statements

```
SELECT max(salary),employee_id,employee_name  
FROM Employee
```

- ▶ Above SQL expression automatically returns the maximum salaried employee's details without doing any computation on developer's end. Using collections framework in java, developer has to use loops and make checks repeatedly. Another concern is efficiency, as multi-core processors are available at ease, java developer has to write parallel code processing which can be pretty error prone.

What is Stream?

- ▶ Stream represents a sequence of objects from a source which supports aggregate operations.
 - ▶ **Sequence of elements** - A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
 - ▶ **Source** - Stream takes Collections, Arrays or I/O resources as input source.
 - ▶ **Aggregate Operations** - Stream supports aggregate operations like filter, map, limit, reduce, find, match and so on.

What is Stream? Continue...

- ▶ Stream represents a sequence of objects from a source which supports aggregate operations.
- ▶ **Pipelining** - Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process input and returns output to target. `collect()` method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- ▶ **Automatic Iterations** - stream operations do the iterations internally over the source elements provided in contrast to Collections where explicit iteration is required.
- ▶ **Automatic Iterations**

Generating Streams

- ▶ With Java 8, Collection interface has two methods to generate stream.
 - ▶ **stream()** - Returns a sequential stream considering collection as its source.
 - ▶ **parallelStream()** - Returns a parallel Stream considering collection as its source.

Generating Streams continue...

▶ ForEach

- ▶ Stream has provided a new method `forEach` to iterate each element of the stream. Consider below example to print 10 random numbers.

- ▶

```
Random random = new Random();  
random.ints().limit(10).forEach(System.out::println  
);
```

▶ map

- ▶ `map` method is used to map each element to corresponding result. Consider below example to print unique squares of numbers.

- ▶

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3,  
7, 3, 5);
```

```
//get list of unique squares
```

```
List<Integer> squaresList =  
numbers.stream().map( i ->  
i*i).distinct().collect(Collectors.toList());
```

Generating Streams continue...

▶ Filter

- ▶ filter method is used to eliminate element based on criteria. Consider below example to print count of empty strings.

```
List<String>strings = Arrays.asList("abc", "",  
    "bc", "efg", "abcd","", "jkl");  
    //get count of empty string  
    int count = strings.stream().filter(string ->  
    string.isEmpty()).count();
```

▶ Limit

- ▶ limit method is used to reduce the size of the stream. Consider below example to print 10 random numbers.

```
Random random = new Random();  
    random.ints().limit(10).forEach(System.out::prin  
tln);
```

Generating Streams continue...

▶ Sorted

- ▶ sorted method is used to sort the stream. Consider below example to print 10 random numbers in sorted order.

- ▶

```
Random random = new Random();  
    random.ints().limit(10).sorted().forEach(System.  
out::println);
```

▶ Parallel Processing

- ▶ parallelStream is alternative of stream for parallel processing. Consider below example to print count of empty strings.

- ▶

```
List<String> strings = Arrays.asList("abc", "",  
    "bc", "efg", "abcd", "", "jkl");  
    //get count of empty string  
    int count = strings.parallelStream().filter(string ->  
    string.isEmpty()).count();
```

Generating Streams continue...

► Collectors

- Collectors are used to combined the result of processing on elements of a stream. Collectors can be used to return a list or a string.

```
List<String>strings = Arrays.asList("abc", "",  
"bc", "efg", "abcd","", "jkl");  
List<String> filtered =  
strings.stream().filter(string ->  
!string.isEmpty()).collect(Collectors.toList()  
);
```

► System.out.println("Filtered List: " +

Generating Streams continue...

► Statistics

- With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
List<Integer> numbers = Arrays.asList(3, 2, 2,
3, 7, 3, 5);
IntSummaryStatistics stats =
integers.stream().mapToInt((x) ->
x).summaryStatistics();
System.out.println("Highest number in List : "
+ stats.getMax());
System.out.println("Lowest number in List : "
+ stats.getMin());
System.out.println("Sum of all numbers : " +
stats.getSum());
```