Core Java

Table of Contents

Module 1. Getting Started

- **Overview**
  - Introduction to Java
  - Writing, compiling and running a program
  - Platform independency in Java
  - Integrated Development Environment
  - Java Application Development Life Cycle
  - JDK Packaging
  - Just Minute

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:

```
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:

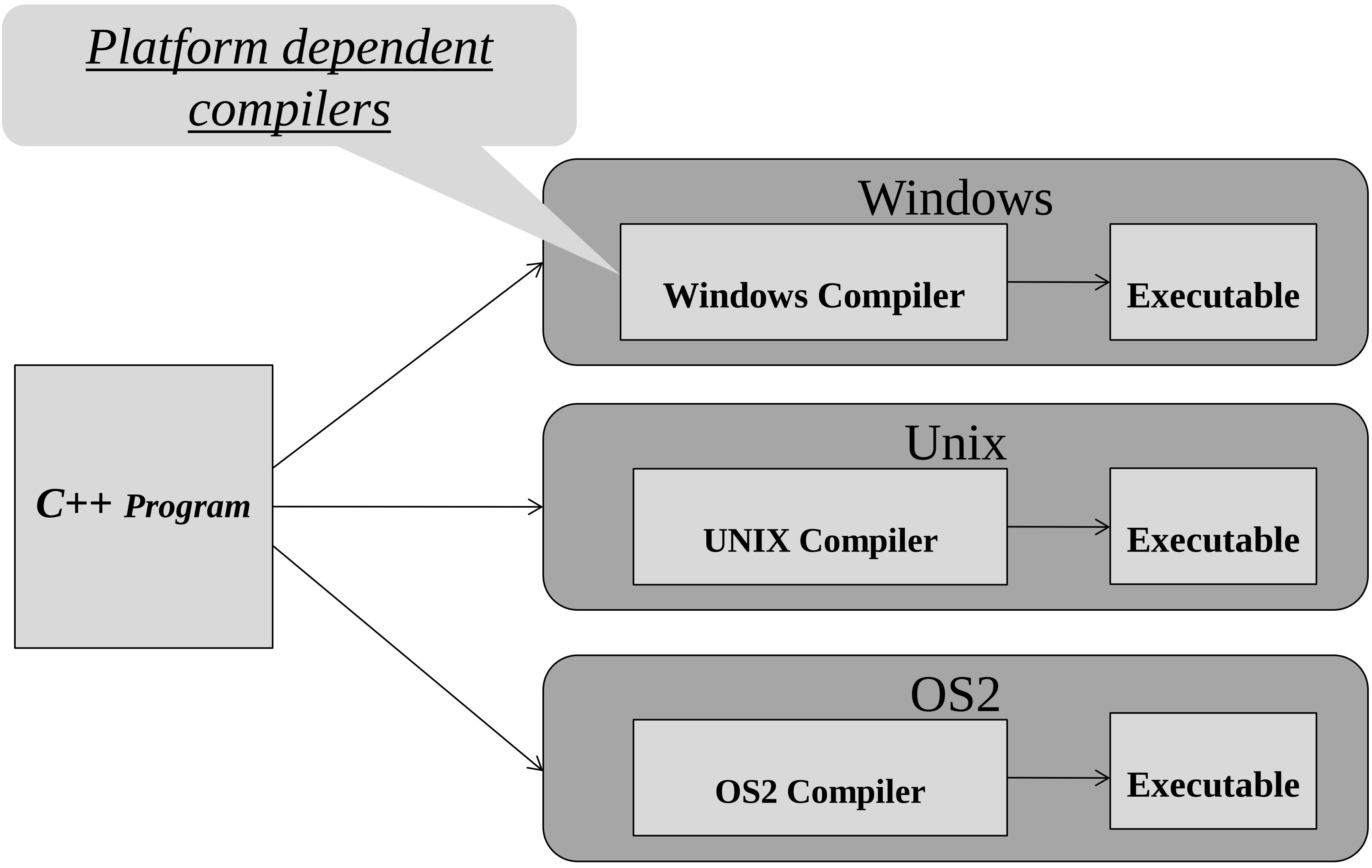
  javac HelloWorld.java

- If there are no errors, there should be a file called **HelloWorld.class** in your working directory.

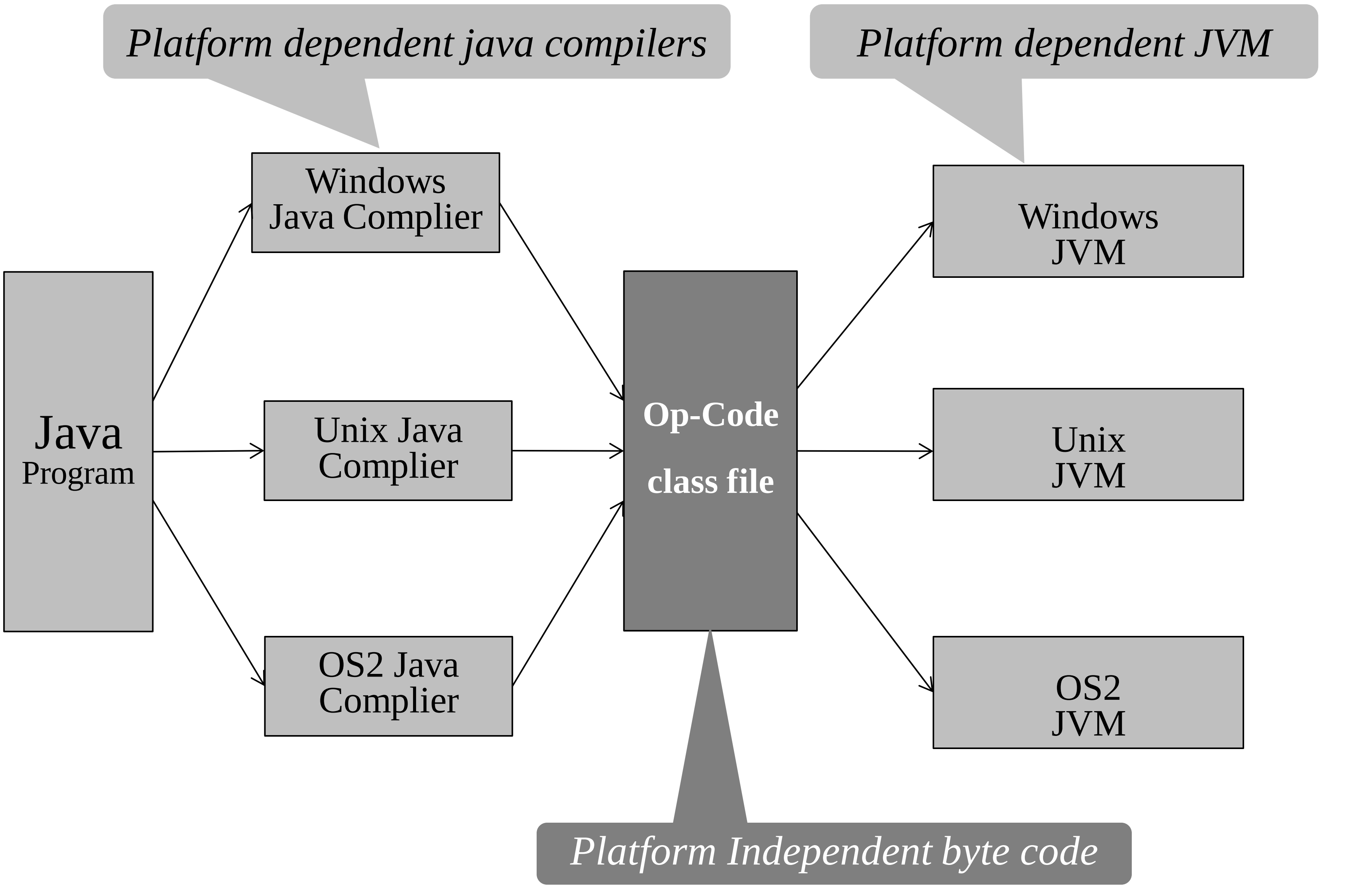
- To run the program, at the command prompt, type:

  java HelloWorld

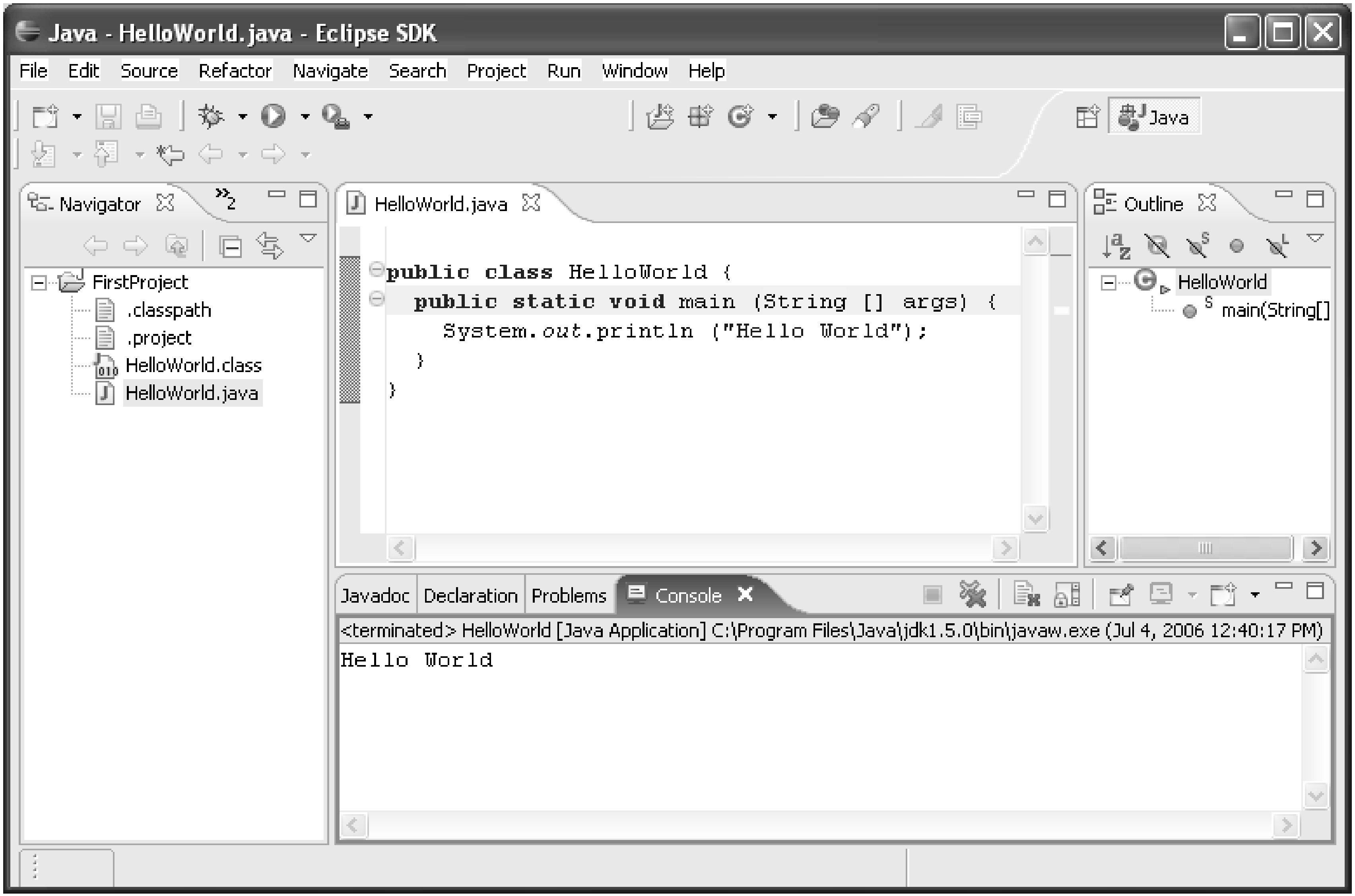
Platform Dependency



Platform Independency



Integrated Development Environment

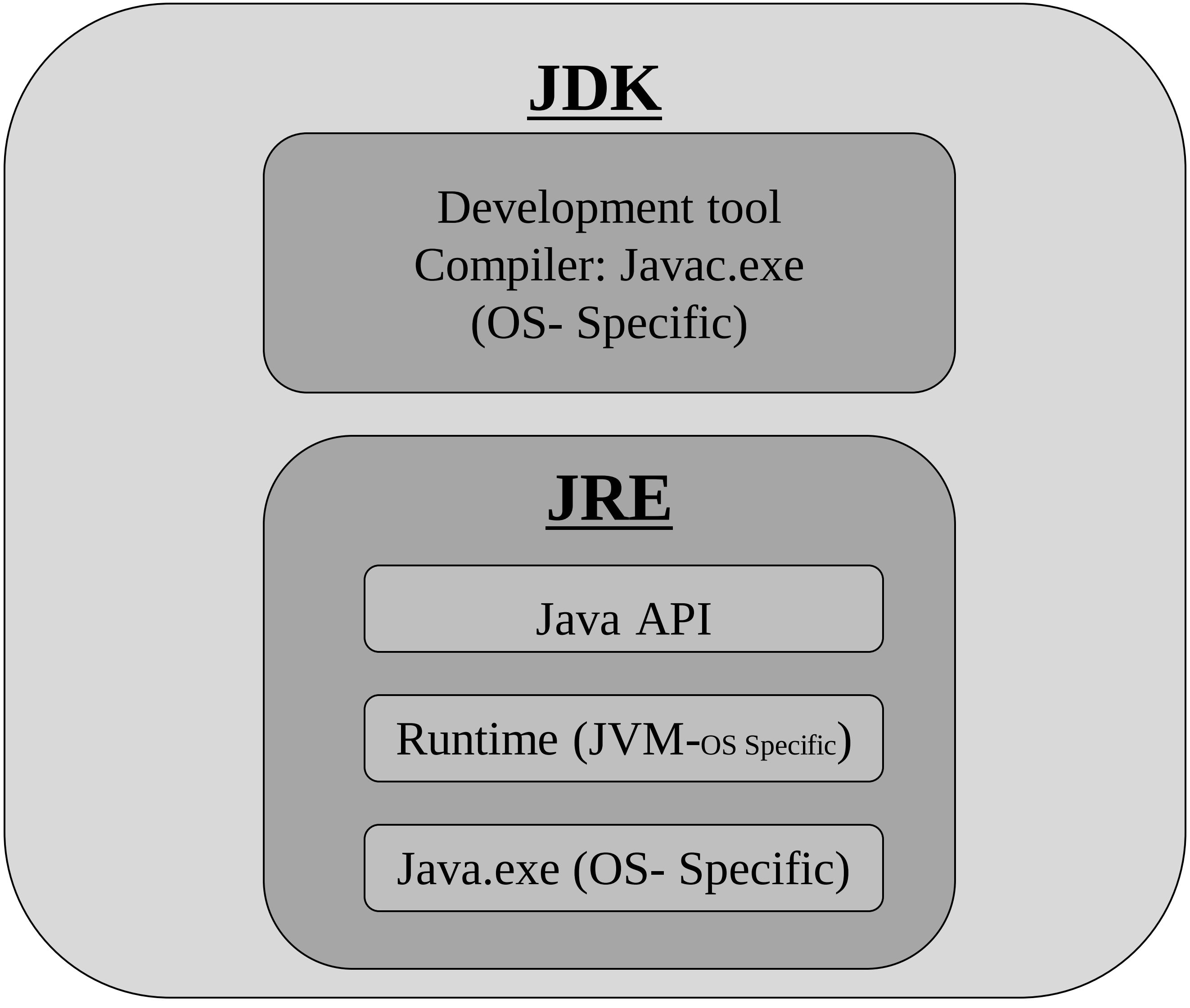


Java Application Development Life Cycle

Design Approach → Development Approach → Testing → Zip → Deploy on client Side

Design Approach	Development Approach	Testing	Zip	Deploy on client Side
Tools	**Tools**	**Tools**	**Utility**	**Tools**
Enterprise Architect	Eclipse	JUnit	Jar	ANT
Visio	Netbeans	SpryTest		Mevan
Argo UML	IBM WebSphere Studio	Jtest		
	Borland Jbuilder			
	Oracle JDeveloper			

JDK Development Platform



Just Minute…

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)
  - 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

Naming Conventions in Java

//package name should be in smallcase

package com.pragatisoftware.calculator

//class name should be in PascalCase

class ScientificCalculator {

  //Contansts shoule be in UPPERCASE

  final double PIE = 3.14;

  //variable names should be in camelCase

  int valueOne;

  //method name should be in camelCase

  public double add(int num1, int num2) {...}

}

Data Types

Data Types

Primitive Type

Reference Type

Types	Range
byte	8-bit integer (signed) Size (-27 to 27-1)
short	16-bit integer (signed) Size (-215 to 215 -1)
int	32-bit integer (signed) Size (-231 to 231 – 1)
long	64-bit integer (signed) Size (-263 to 263 – 1)
float	32-bit floating-point. Precision to 7-8th digit
double	64-bit floating-point. Precision to16-18th digit
char	16-bit Unicode
boolean	8 -bit

Types
class
enum
array

Variables

```
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

```
class Temperature {
        public static void main (String [ ] args) {
                float centigrade;
                float fahrenheit;
                centigrade =33.33 F;
                fahrenheit = (centigrade * 9 / 5) + 32;
                System.out.println ("33.3 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?

Post fixing a constant with letter F tells Java Virtual Machine to treat it as a float value rather then default double

Language Enhancements

Using underscore in numbers to identify places.

```
int  thousand = 1_000;   // Convention: 1,000.
int  cror=1_00_00_000; // Convention:1,00,00,000
int  thousand1 = 1_0_0______0;// Consecutive _ allowed.
float real1 = 1_234.56f;     // Reals allowed.

if (thousand==thousand1){
        System.out.println(true);
}
```

- Allowed with int, long, short, byte, float, double etc.
- Does not change value of a number.

.

Writing Comments

```
class Temperature {
    /**  The program is written by…                    documentation comment
    @ Author Pragati Software Pvt. Ltd
    */
    public static void main (String [] args) {
      //Variable declarations:                         single line comment

      /* double centigrade;
      double fahrenheit;                              multiple line comment
      */
      float centigrade;
      float fahrenheit;
      centigrade = 33.3f;
      fahrenheit = (centigrade * 9 / 5) + 32; // Conversion Formula
      System.out.println ("33.3 Centigrade = “ + fahrenheit + " Fahrenheit.");
    }
}
```

Named Constants

```
public class CircleArea {
    static final float PI = 3.1416f;
    final float BORDER_THICKNESS = 2.4F;
    float circleRadius;

    public float getArea() {
      // Code for calculating area of circle goes here
    }

    public static void main(String[] args) {
      System.out.println(CircleArea.PI);
      CircleArea circle = new CircleArea();
      circle.circleRadius = 20;
      float area = circle.getArea();
      System.out.println("Area is :- “ + area);
    }
}
```

Arithmetic Operators

+ addition
- subtraction
* multiplication
/ division
% remainder

Unary minus (-) for negation

Unary plus (+).

Increment and Decrement Operators

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

What will the output of this program?

Assignment Operators

Let initial value of num1 be 15,

```
num = num1;
num+= 5;
num = num + 5;
num -= 5;
num *= 5;
num /= 5;
num %= 5;
```

What would be the final value for ‘num’?

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

Lets initialize values

int num = 10, num2 = 20, num3 =10, num4 =15;

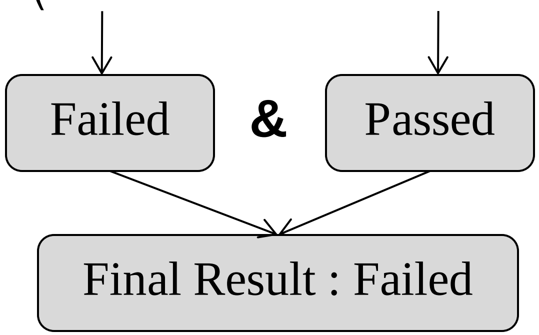
num	**==**	Num4	**?**
num2	**!=**	num3	**?**
num3	**>**	num2	**?**
num	**<**	num4	**?**
num4	**>=**	num2	**?**
num2	**<=**	num	**?**

Logical and Boolean Operators

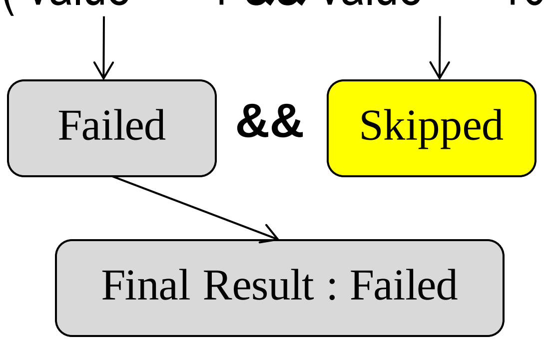
Check value is in range of 1 - 100

int value = -2

if (value >= 1 **&** value <= 100)



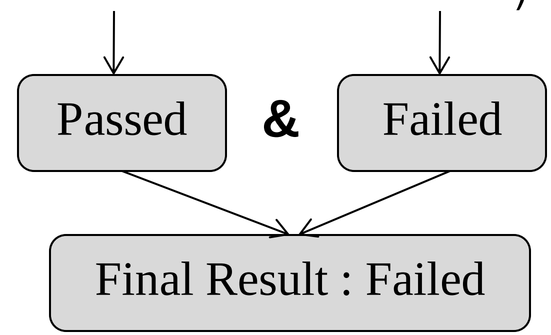
if (value >= 1 **&&** value <= 100)



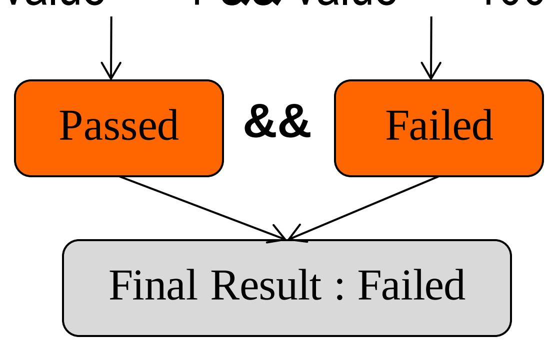
Check value is in range of 1 - 100

int value = 125

if (value >= 1 **&** value <= 100)



if (value >= 1 **&&** value <= 100)

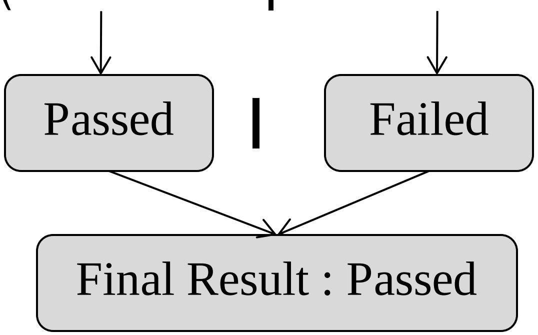


Logical and Boolean Operators Continue...

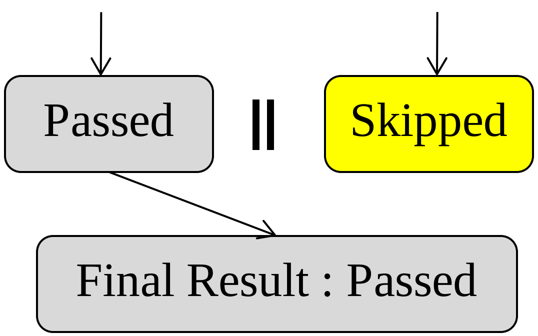
Check value is not in range of 50 - 100

int value = 102

if (value >= 50 | value <= 100)



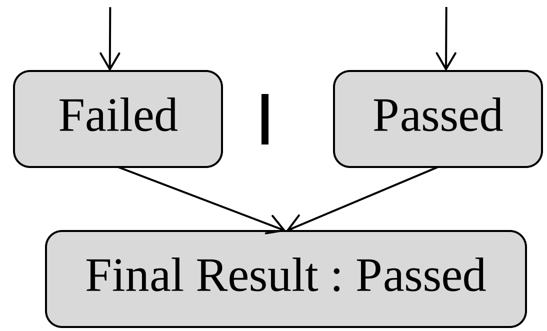
if (value >= 50 || value <= 100)



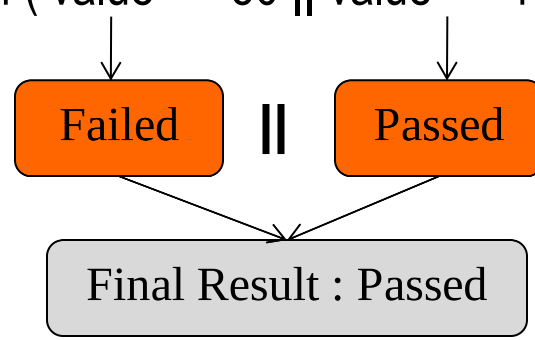
Check value is not in range of 50 - 100

int value = 30

if (value >= 50 | value <= 100)



if (value >= 50 || value <= 100)



Bit-wise Operators

- Bitwise AND(&)
- Bitwise OR(|)
- Bitwise Exclusive OR(^)
- Bitwise Shift Left(<<)
- Bitwise Shift Right with Sign bit(>>)
- Bitwise Shift Right with zero bit(>>>)
- Bitwise Complement(~)

Arithmetic Operations using Bitwise operators

```
public class BitWiseOperators {
    public static void main(String[ ] args){
            int i = 10;
    // A single left shift will multiply the number by 2.
            int j = i<<1;
            System.out.println("The value of j : "+j);
    // A single right shift will divide the number by 2.
            int k = i>>1;
            System.out.println("The value of k : "+k);

    }
}
```

Language Enhancements (Contd…)

Integers in binary form: They make relationships among data more apparent.

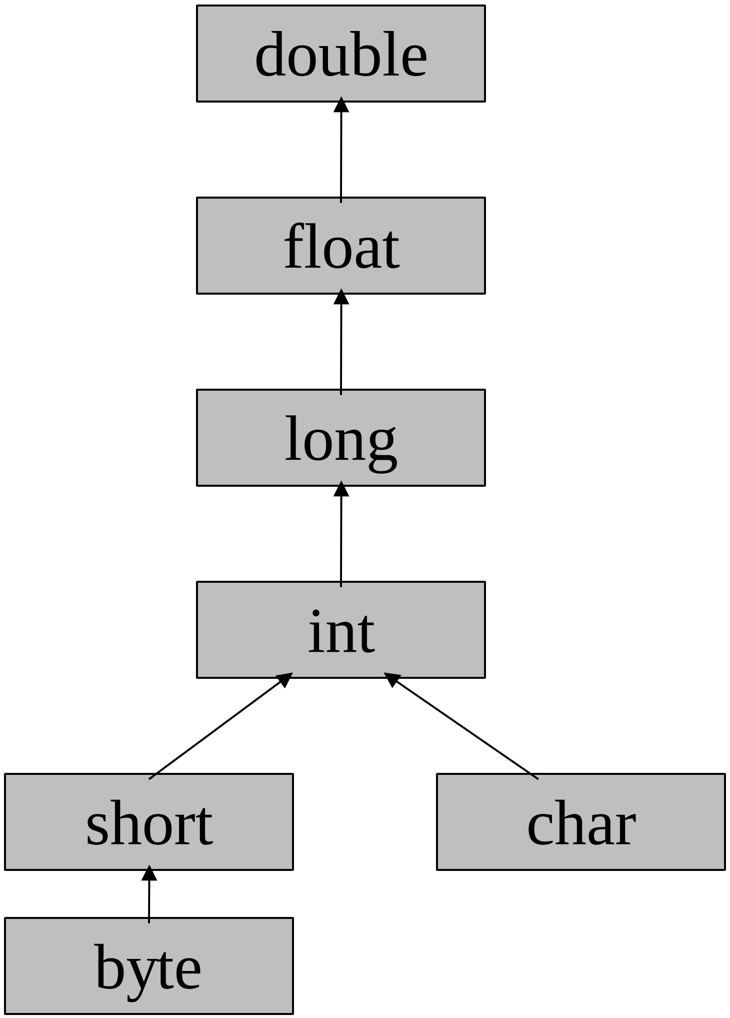
Binary representation- **int anInt2 = 0b101;**

rotated bitwise:
int[] phases = { **0x31, 0x62**, 0xC4, 0x89, 0x13, 0x26, 0x4C, 0x98}; // Hex representation
int[] phases = { **0b00110001, 0b01100010**, 0b11000100, 0b10001001, 0b00010011, 0b00100110, 0b01001100, 0b10011000}; // Binary represent. Clarity of left shift by 1 position.

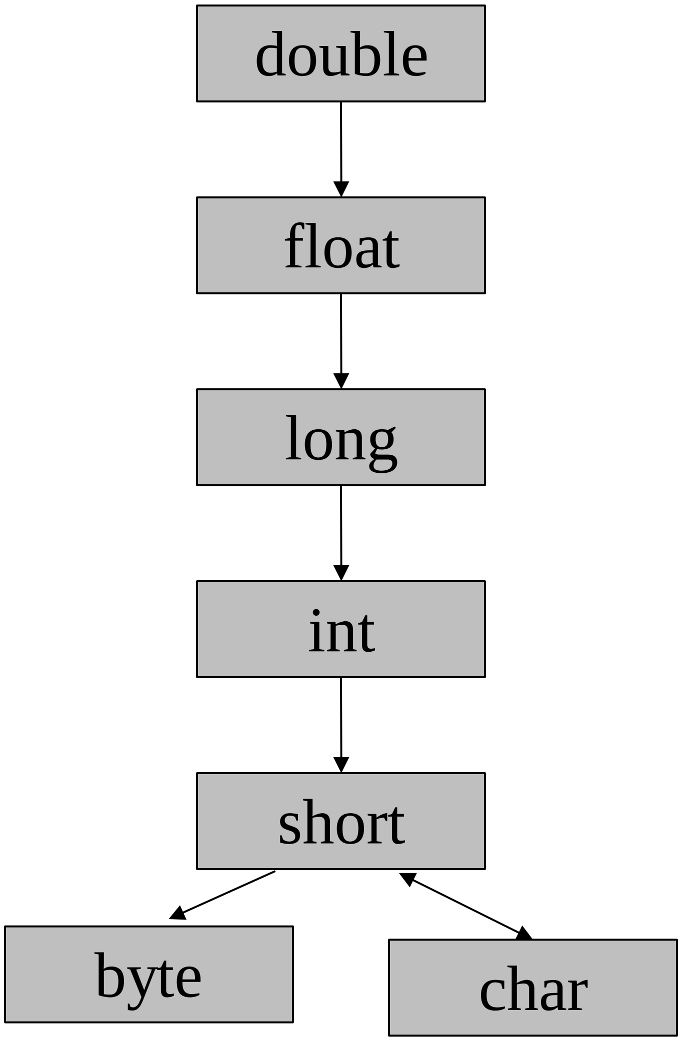
int binaryPattern = 0B1000_0011;
System.*out.println("Binary Pattern:"*+*binaryPattern);*
int mask = 0B0000_1000;
if ((**binaryPattern & mask**)>0)

Promotion and Demotion Rules

Promotions



Demotions



Points to remember

short = short + short

//Gives compile time error. Because + operator returns int value

byte = byte + byte

//Gives compile time error. Because + operator returns int value

int = char

//works. Java implicitly converts char to int (ascii equivalent of char)

char = int

//Gives compile time error. Implicit cast from int to char is not allowed.

while & do while Loop

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

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

The for Loop

Structure of the for loop

```
  for (exp1; exp2; exp3)
    statement;
```

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

for loop vs. while loop

For loop	While loop

int num;
for (num = 1; num <= 10; num++)
  System.out.println (num);
	int num;
	num = 1;
	while (num <= 10)
	  System.out.println (num++);

Ideal for predictable number of Iterations	Not ideal for predictable number of Iterations
Suitable for Generating numbers due to its nature	Not suitable for Generating numbers

The if...else Structure

```
class Larger {
    public static void main (String [ ] args) {
        int num1 = 10;
        int num2 = 15;
        if ( num1 > num2 ){
            System.out.println ("Num1 is larger.");
        }
        else{
          System.out.println ("Num2 is larger.");
        }
    }
}
```

Nested if...else

```
class LargestOutOfThree {

        public static void main (String [ ] args) {
                int num1 = 25, num2 = 15, num3 = 10, largest;

          if ( num1 > num2 ) {
                        if ( num1 > num3 )
                                largest = num1;
                        else
                                largest = num3;
          }
          else {
                        if ( num2 > num3 )
                        largest = num2;
                else
                        largest = num3;
                }
                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)
        …
        …
        else if (dayOfWeek == 6)
                System.out.println ("Saturday");
    }
}
```

The switch Statement

```
switch ( dayOfWeek ) {
    case 0: System.out.println ( "Sunday“ );
    case 1: System.out.println ( "Monday“ );
    .
    ......
    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!");       break;
}
```

break and continue Statements

```
int count = 0;


for( count = 0 ; count < 40 ; count++) {
	if( count % 7 == 0 )
		continue;  //Skip below code but continue loop
	System.out.println( count );
}
```

```
for( count = 0 ; count < 10 ; count )  {
	if( count % 30 == 0 )
		break; //Terminate the current loop
	 System.out.println( count );
}
```

Language Enhancements (Contd…)

Switch with Strings:

```
String actionCommand;
actionCommand = // Assign a command string.

switch(actionCommand){
	case "Insert": {
		System.out.println("Join new user");
		// Steps to insert new user.
		break;
	}
	case "Delete": {
		System.out.println("Retire user.");
		// Steps to retire a user.
		break;
	}
}
```

Using if-else

```
actionCommand = // Assign a command string.

If (actionCommand.equals(“Insert”)){
	System.out.println("Join new user");
	// Steps to insert new user
} else if (actionCommand.equals(“Insert”)) {
	System.out.println(“Retireuser");
	// Steps to retire a user
}
```

Labelled break and continue Statements

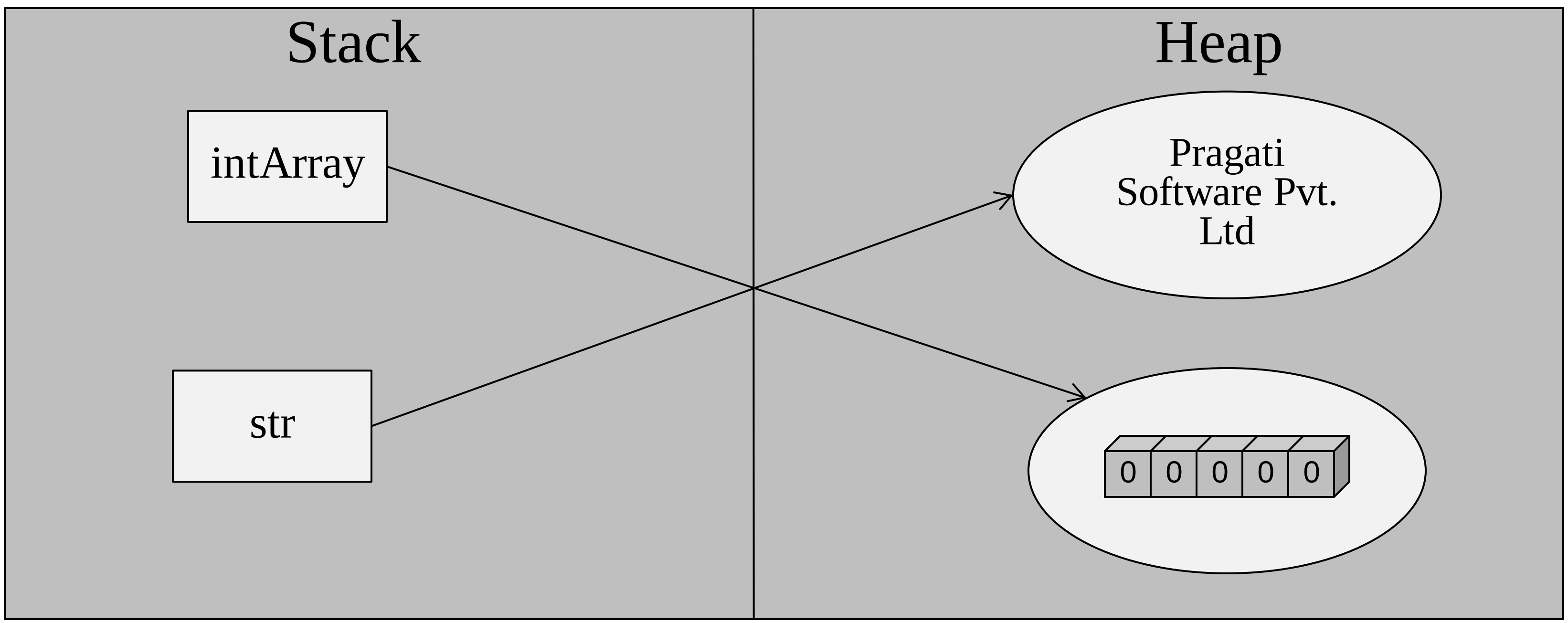
```
public class LabelledBreak {
    public static void main (String [ ] args) {
        outer:
        for ( int num = 1; num <= 10; num++ )  {
            for ( int num1 = 1; num1 <= 5; num1++ )  {
                System.out.print ( "\t" + (num * num1 ));

                if ( (num * num1) == 18 ) {
                        break outer;
                }
            }
            System.out.println ( );
        }
    }
}
```

Reference Variables

- **Reference and Object creation**

int [] intArray = new int [5];

String str = “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

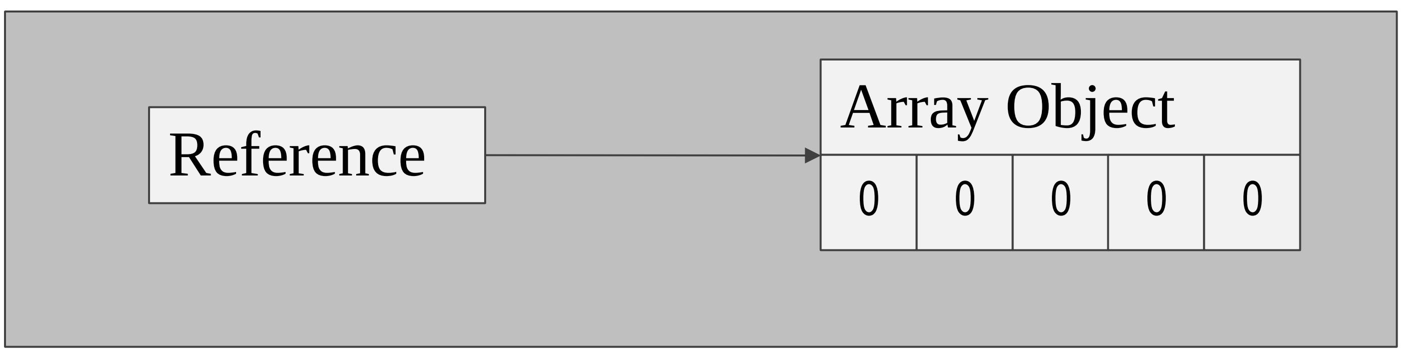
Arrays

- **Array Declaration**

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

- **Length of an array**

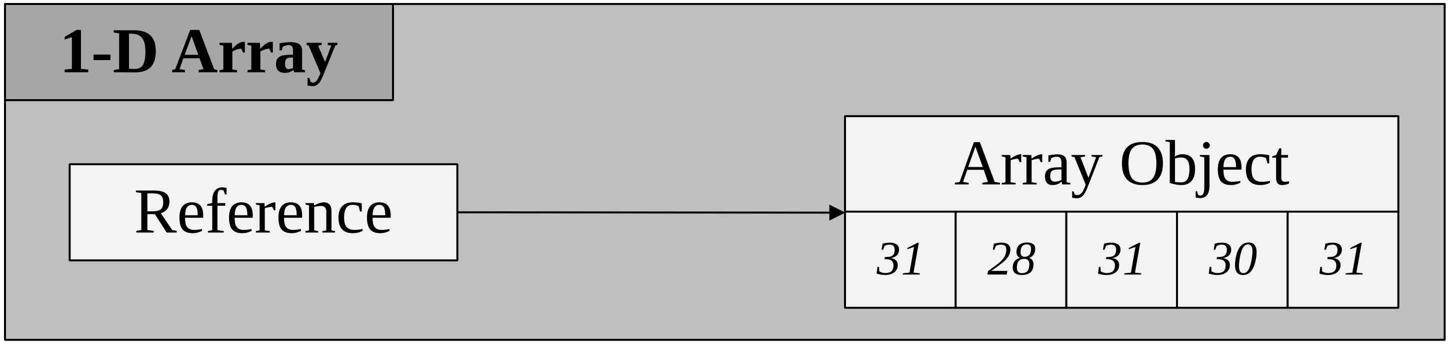
```
for ( int index = 0; index < intArray.length; index++)
    System.out.println (intArray [index]);
```



Arrays (Cont…)

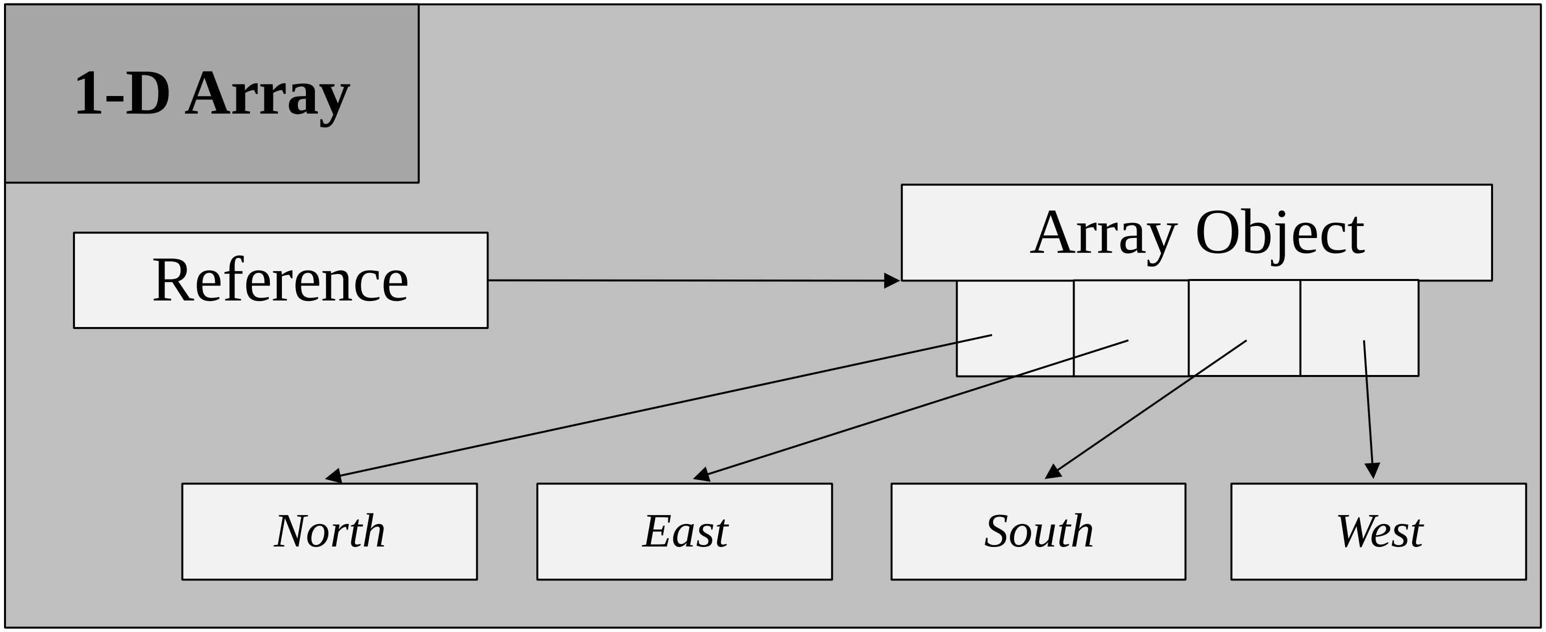
Array Initialization

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



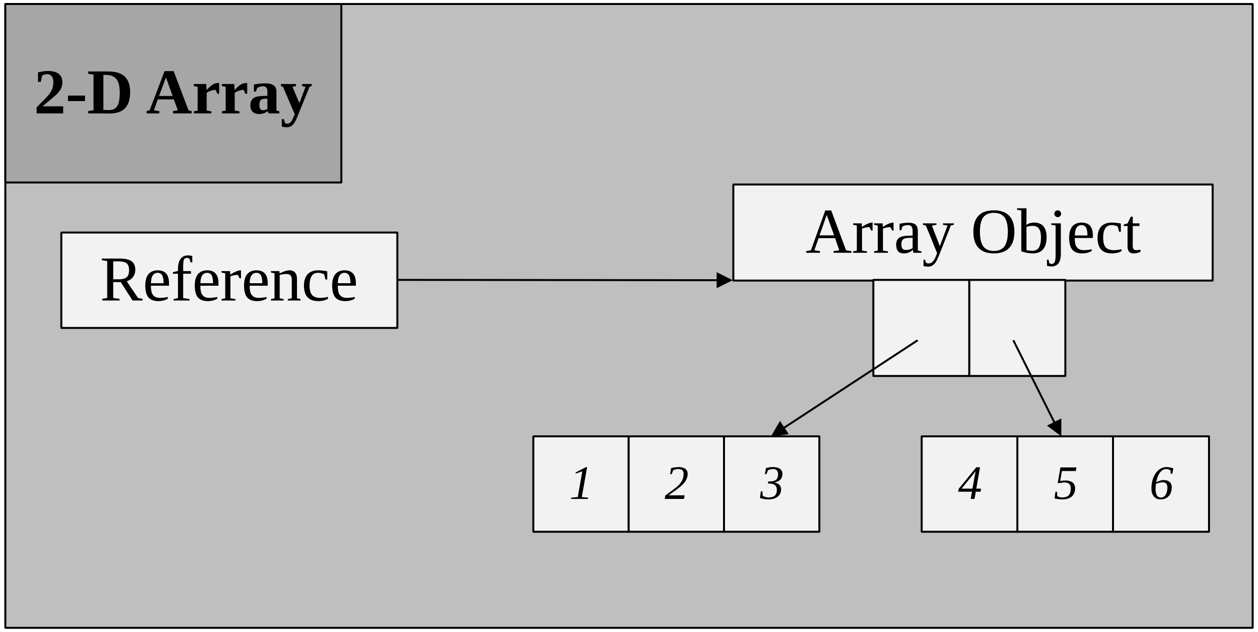
Memory Map of array (contd…)

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



Memory Map of array

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



Arrays of Array

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

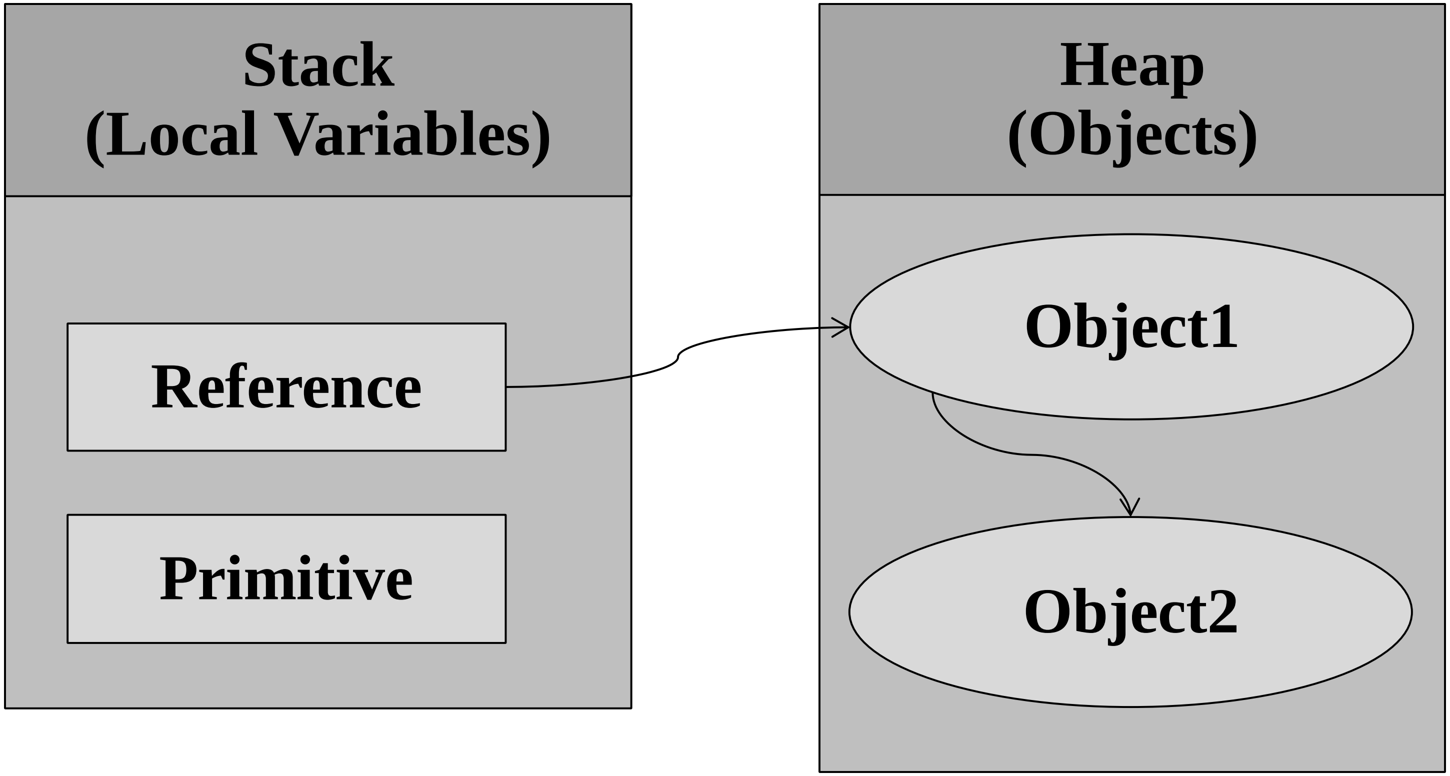
```
int[ ] [ ] array2D = new array2D [ 5 ][ ];  // First subscript is mandatory.
```

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 rows = 1; rows < marks.length ; rows ++) {
                        System.out.println ("Enter marks of student "+rows);

                        for (int cols = 1; cols < marks[rows].length ; cols++) {
                                System.out.println ("Subject "+cols);
                                marks [rows][cols] = sc.nextInt();
                        }
                }  for (int rows = 1; rows < marks.length ; rows++) {
                                for (int cols = 1; cols < marks[rows].length ; cols++)
                                System.out.print (marks[rows][cols]+" ");
                }

        }
}
```

Objects Vs Local Variables

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



An Enhanced Version of For Loop

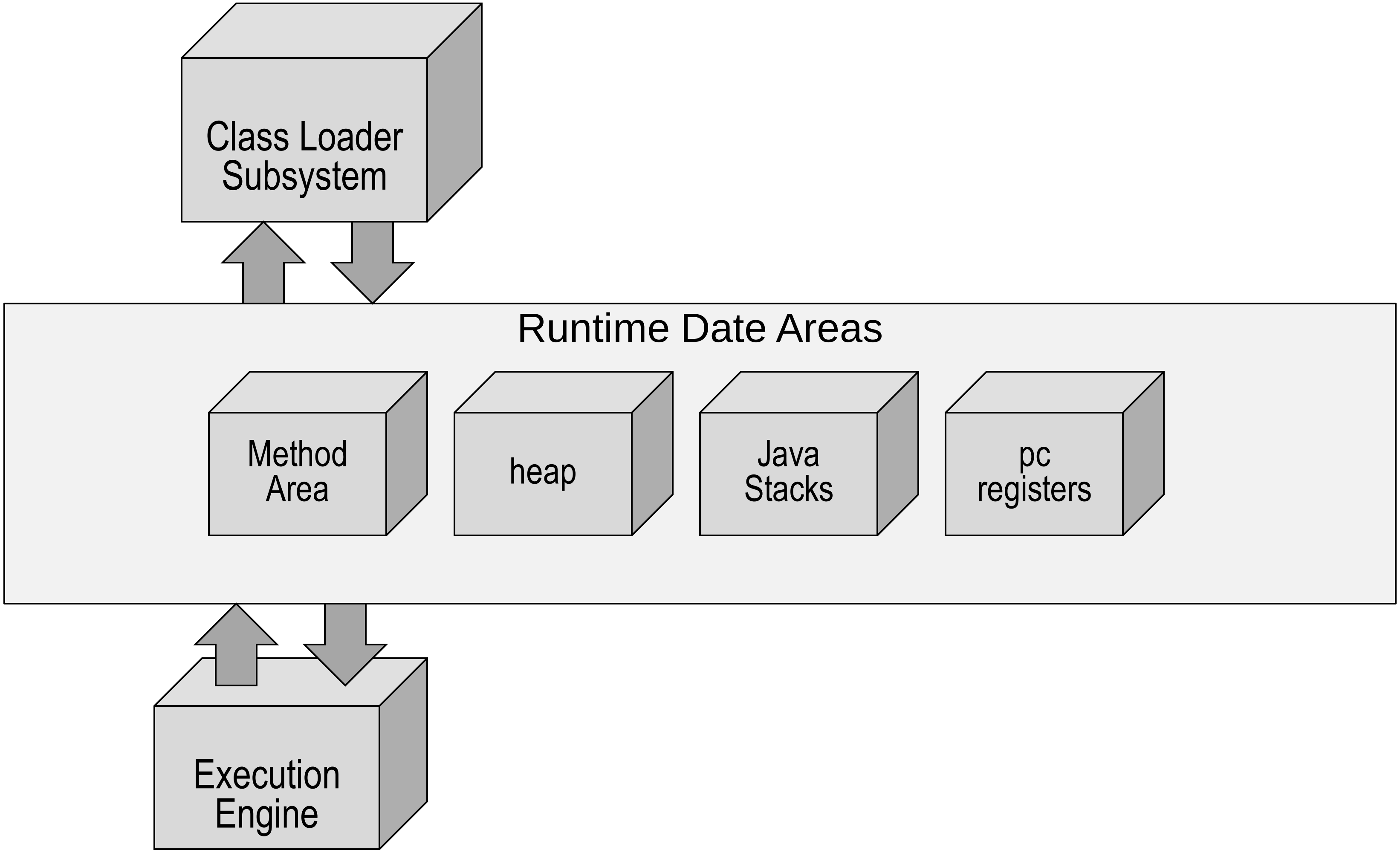
```
class EnhancedForLoop {
	public static void main (String [ ] args) {
		int numbers[ ]={40,50,60,70,80};
		for (int number : numbers) {
			System.out.println("Value is : " + number);
		}
	}
}
```

Advantages	Disadvantages
Convenience from the programmer's point of view for iterating over the Collection	Step-value cannot be incremented
Performs a strict sequential iteration of all elements from the given container	Backward traversal is not possible

Multidimensional Arrays using For-each loop

```
import java.io.IOException; import java.util.Scanner;
class TwoDimArray {
	public static void main (String [ ] args) throws IOException {
			int [ ][ ] marks = new int [4][4];
			Scanner sc = new Scanner(System.in);

			for (int rows = 0; rows < marks.length ; rows ++) {
				System.out.println ("Enter marks of student : " + rows);

				for (int cols = 0; cols < marks[rows].length ; cols++) {
					System.out.println ("Subject " + cols);
					marks [rows][cols] = sc.nextInt();
				}
			}
			for (int[ ] rows : marks) {
				for (int cols : rows){
					System.out.print (cols + " "); }
				System.out.println();
			}
	}
}
```

JVM Architecture Specification



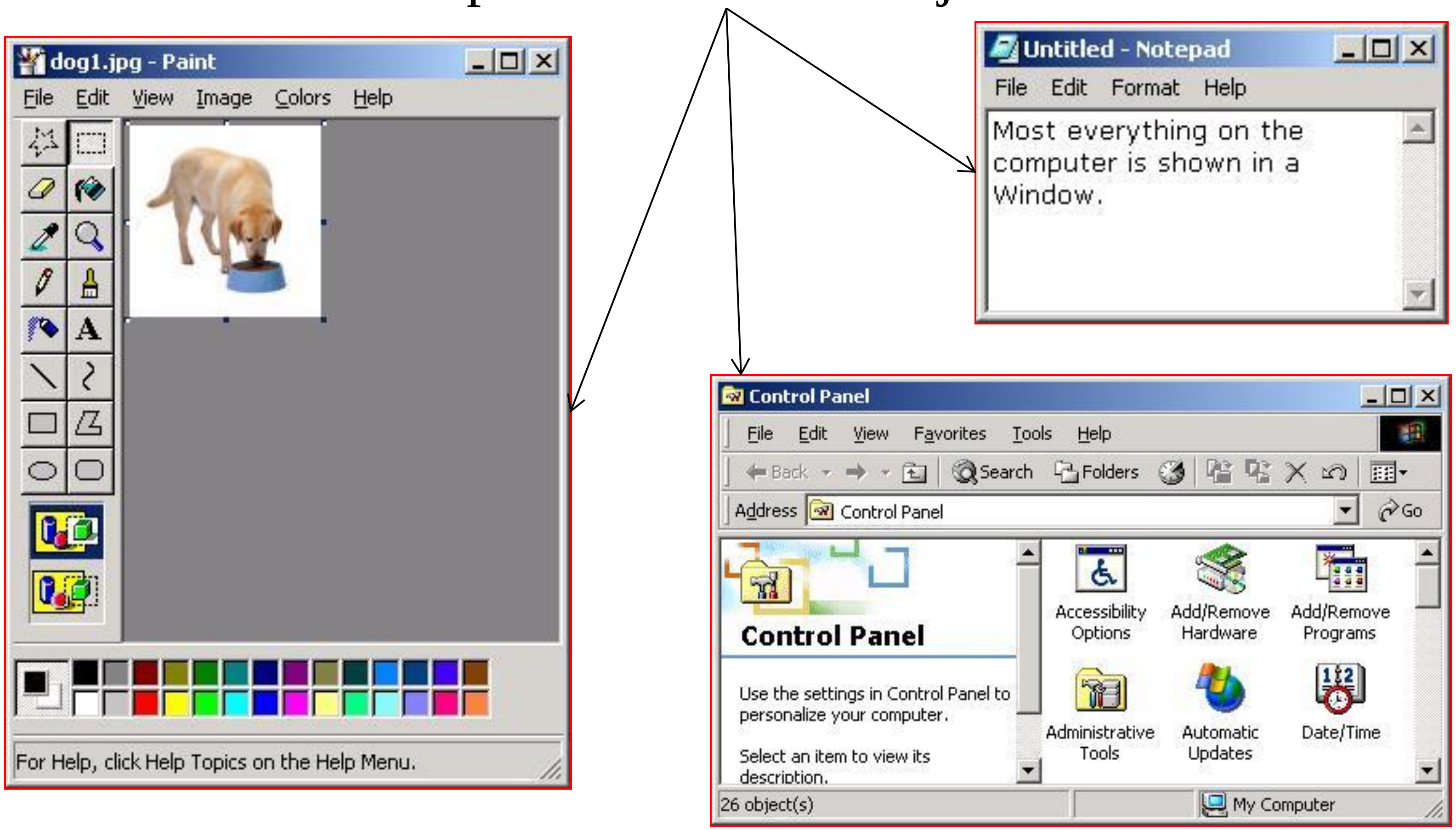
Just Minute…

Module 3. Classes and Objects

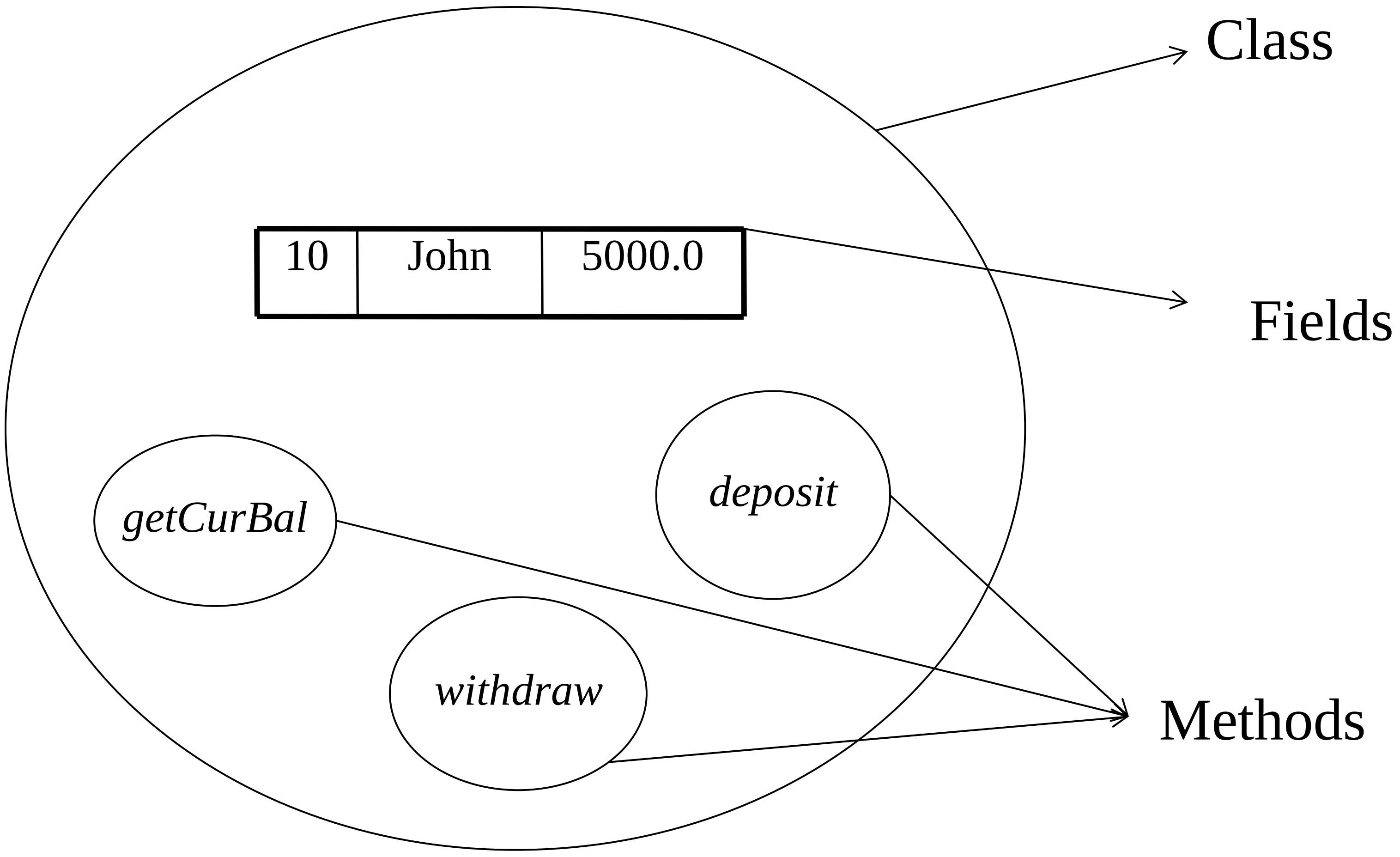
- **Overview**
  - Classes and objects
  - Access Specifiers
  - Method Overloading
  - Constructors and Init blocks
  - Static methods and fields
  - Var-args
  - Garbage collection -finalize() method
  - The ‘toString()’ method
  - The ‘this’ reference

Classes and Objects

- A class is an abstract description of objects
  - class Window { ... }
- Here are some examples of Windows objects:



Classes and Objects



Creating a Class

```
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

Access Control	Description
Public	Accessible anywhere the class is accessible. They are also inherited by all subclasses
Package/Default	Accessible in the class itself, and accessible to, and inherited by, code in the same package.
Protected	Accessible in the class itself, & accessible to,& inherited by, class in the same package & in subclasses.
Private	Accessible anywhere the class is accessible. They are also inherited by all subclasses.

Fields and Methods

```
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;
    }
}
```

Parameter Values

```
public class PassByValue {
    public static void main(String[ ] args)
    {
        int value = 15;      int[ ] array = {10, 20, 30, 40};
        BankAccount account = new BankAccount(101, "abc", 5000);

        //print : value , Array Data,  account.getName();

        changeThem(value, array, account);

        //print : value , Array Data,  account.getName();
    }

    public static void changeThem(int value, int[ ] array, BankAccount account)
    {
        value = 60;
        for(int index=0; index<array.length; index++) {
            array[index]++;
            account.setName("xyz");
        }
    }
}
```

Overloading

```
public class Addition {
        public void add(int num1 ,int num2) {
          System.out.println("Adding 2 integers...");
          System.out.println("sum : "+(num1+num2));
        }
        public void add(int num1 ,int num2,int num3) {
          System.out.println("Adding 3 integers...");
          System.out.println("sum : "+(num1+num2+num3));
        }
        public void add(float num1 ,float num2) {
          System.out.println("Adding 2 float values...");
          System.out.println("sum : "+(num1+num2));
        }
          public static void main(String[ ] args){
          Addition obj = new Addition( );
          obj.add(10,20);      obj.add(10,20,30);     obj.add(10.0f,20.0f);
        }
}
```

Overloading Constructors

```
class BankAccount  {
    private float curBal;

    public BankAccount ( ) {
      curBal = 0;
    }

    public BankAccount (float amt)  {      //Overloaded Constructor
      curBal = amt;
    }
}

BankAccount acc1 = new BankAccount ( );         // Constructor Invocation
BankAccount acc2 = new BankAccount (5000);    // Constructor Invocation
```

Multiple Constructor

```
public class User {
    String firstName;
    String lastName;
    String alias;

    public User(String firstName) {
      this.firstName = firstName; this.lastName = "Unknown";
      this.alias = "Unknown";
    }

    public User(String firstName, String lastName) {
      this.firstName = firstName;  this.lastName = lastName;
      this.alias = "Unknown";
    }

    public User(String firstName, String lastName, String alias) {
      this.firstName = firstName;  this.lastName = lastName;
      this.alias = alias;
    }
}
```

Can this code be Improved?

Init Blocks

```
class User {
    String firstName;  String lastName; String alias;
    {
      this.firstName = "Unknown"; this.lastName = "Unknown";
      this.alias = "Unknown"; // Init block
    }

    public User(String firstName) {
      this.firstName = firstName;
    }

    public User(String firstName, String lastName) {
      this.firstName=firstName; this.lastName = lastName;
    }

    public User(String firstName, String lastName, String alias) {
      this.firstName=firstName; this.lastName=lastName; this.alias = alias;
    }
}
```

Can this code be Improved?

The ‘this’ Reference

```
class User {
      String firstName; String lastName; String alias;
      {
        this.firstName = "Unknown";
        this.lastName = "Unknown";
        this.alias = "Unknown"; // Init block
      }

      public User(String firstName) {
        this.firstName = firstName;
      }

      public User(String firstName, String lastName) {
        this(firstName);        this.lastName = lastName;
      }

      public User(String firstName, String lastName, String alias) {
        this(firstName, lastName);        this.alias = alias;
      }
}
```

Static Methods and Fields

```
class BankAccount {
        private int accNo;
        private float curBal;
        private static int idNum = 1000;

        public BankAccount () {
                accNo = idNum++; curBal = 0;
        }
        public static int getIdNum () {
                return idNum;
        }
        public static void main (String [ ] args) {
                BankAccount ba = new BankAccount ();
                System.out.println (BankAccount.getIdNum ());
        }
}
```

The main Method

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

    }
}
```

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

```
class Calculator {
    public static int add(int... parameters) {
        int total = 0;
        for (int number : parameters)
            total += number;
        return total;
    }
}

public class TestCalculator {
    public static void main(String[] args) {
        System.out.println(Calculator.add(10, 89));
        System.out.println(Calculator.add(657, 34));
        System.out.println(Calculator.add(56, 787, 565, 5655, 354, 786, 435));
    }
}
```

Garbage Collection – finalize () Method

```
class Employee {
    Employee (){
        System.out.println ("Employee created...");
    }
    protected void finalize () {
        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;
```

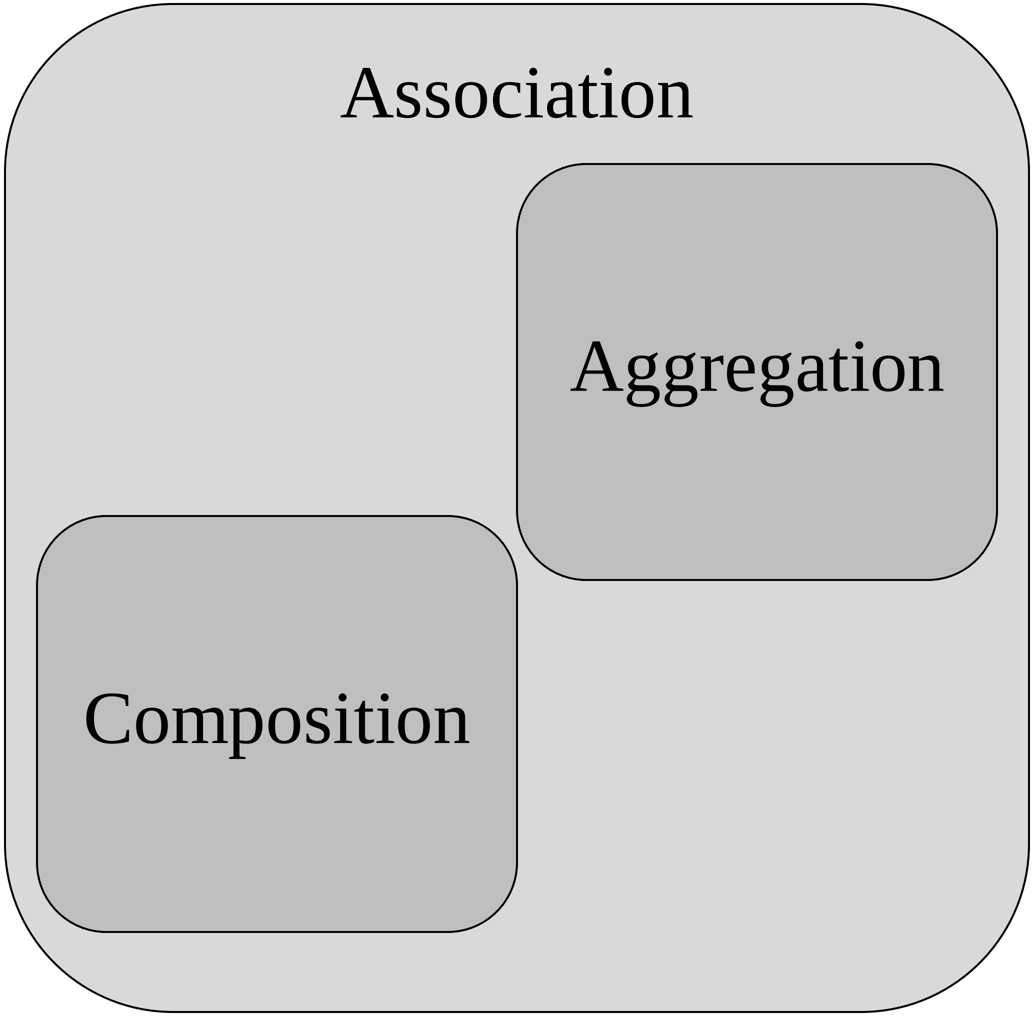
The ‘toString()’ Method

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

	public BankAccount(int accNo, String name, float curBal) {
	  super();
	  this.accNo = accNo;
	  this.name = name;
	  this.curBal = curBal;
	}

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

```
//Test code
BankAccount objectB = new BankAccount (10, "John", 5000.0f);
System.out.println ("Details of objectB : " + objectB);
```

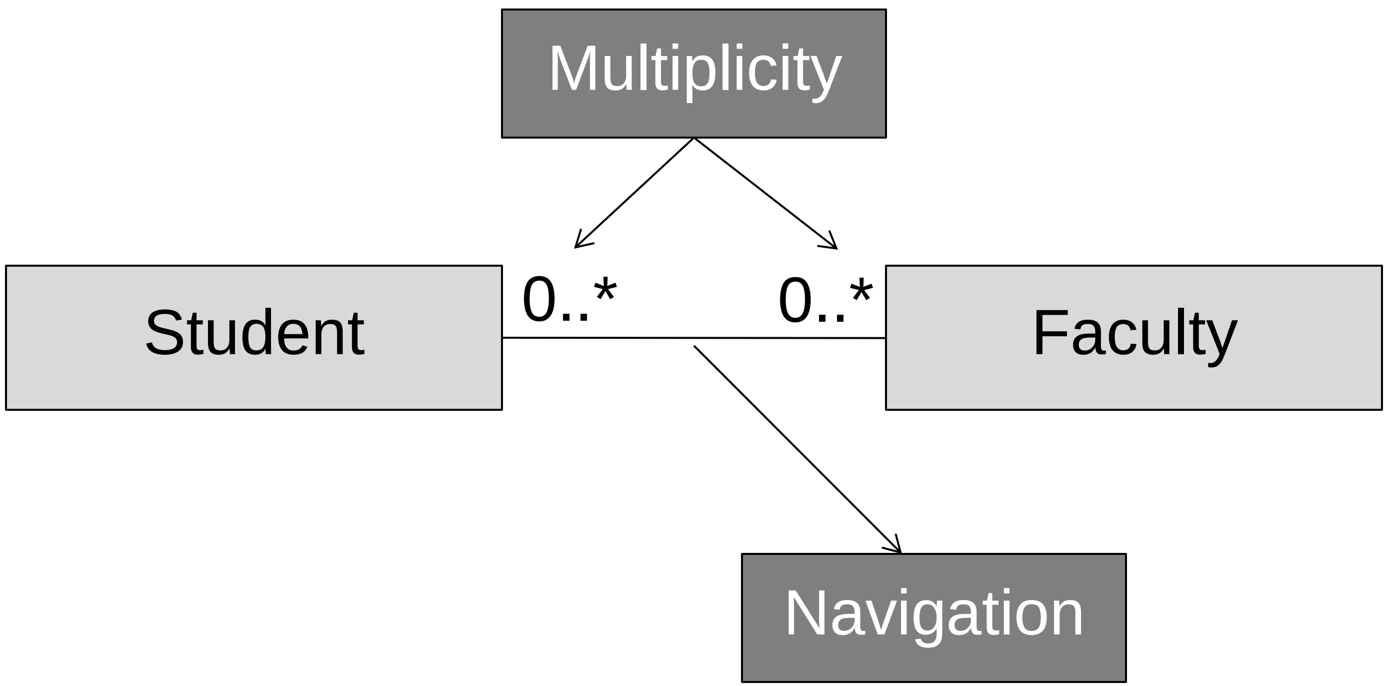
Just Minute…

Module 4. Relation among Classes



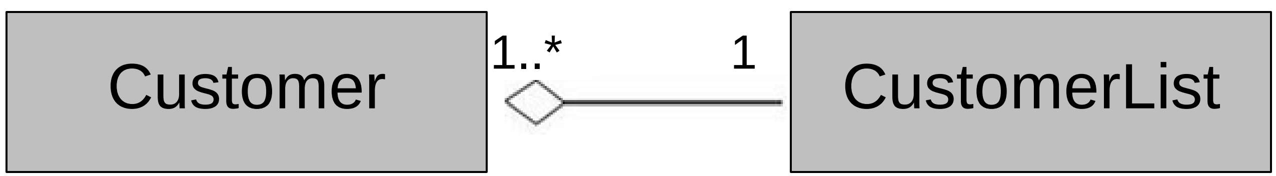
Association

- Association is a relationship between two objects.
- Association defines the **multiplicity** between objects.
- One-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects.



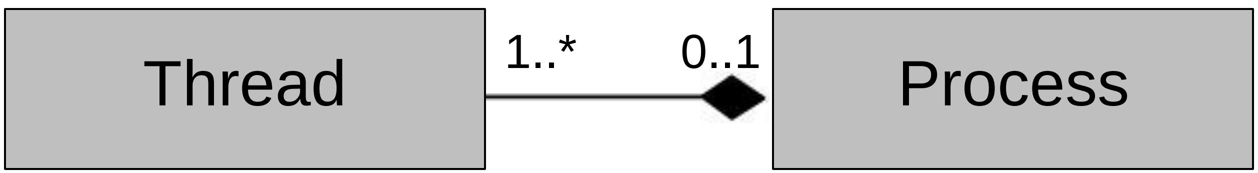
Aggregation

- Aggregation is a special case of association.
- A directional association between objects.
- When an object refers to group of objects, then you have got an aggregation between them.



Composition

- Composition is a special case of aggregation.
- In a more specific manner, a restricted aggregation is called composition.
- When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.



Difference between Aggregation & Composition

Aggregation	Composition
Aggregation is less restrictive	Composition is more restrictive
Existence of two objects is not depended on each other	the composed object cannot exits without other object

Example:

- A Library contains students and books.
- Relationship between library and student is aggregation.
- Relationship between library and book is composition.
- A student can exist without a library and therefore it is Aggregation
- A book cannot exist without a library & therefore its a composition.

Just Minute…

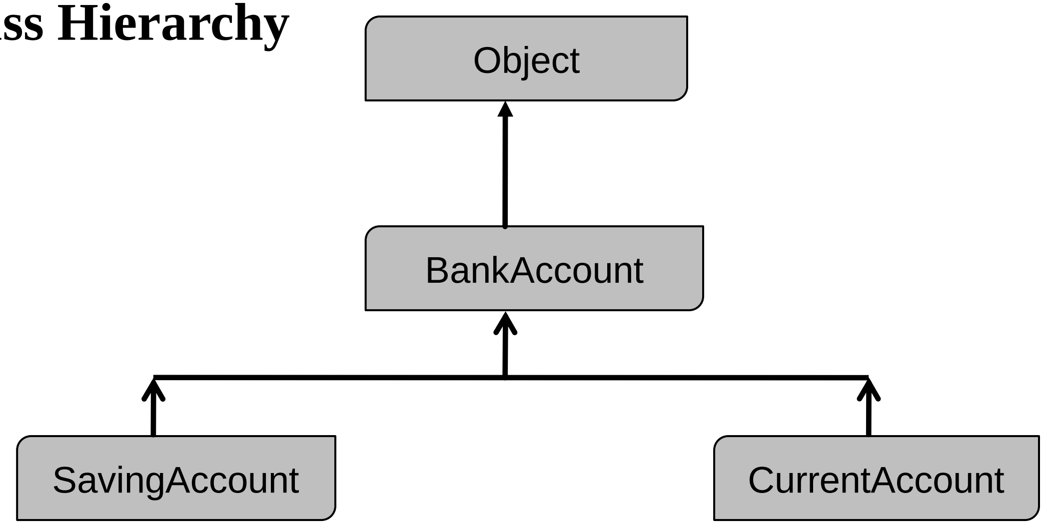
Module 5. Extending Classes

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

Generalization

- Generalization uses a “is-a” relationship from a specialization to the generalization class.
- Common structure and behavior are used from the specialization to the generalized class.
- At a very broader level you can understand this as inheritance.

Class Hierarchy



The BankAccount Class – The Super class

```
class BankAccount {
    private int accNo;
    private String name;
    private float curBal;
    private static int idNum = 1;

    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

```
class SavingsAccount extends BankAccount {
        private boolean isSalaryAcc;

          public void setSalaryAcc (boolean isSalaryAcc) {
                this.isSalaryAcc = isSalaryAcc;
         }
         public boolean isSalaryAcc ( ) {
                return isSalaryAcc;
        }
}
```

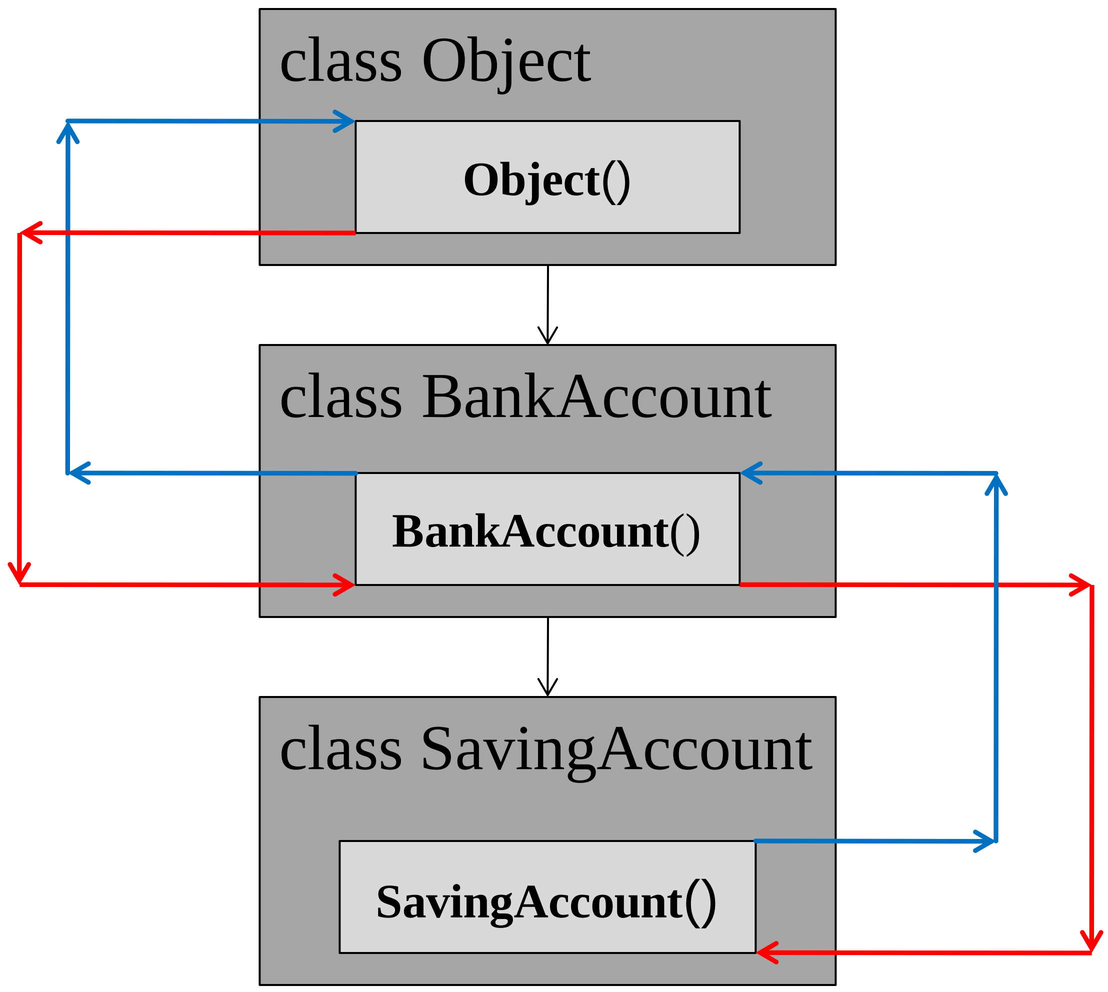
Using a Derived Class Object

```
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.isSalaryAcc( ));
	}
}
```

The protected Keyword

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

Constructors in Inheritance



Constructors in Extended Classes

```
class BankAccount {
        protected float curBal;
         protected String name;
         public BankAccount ( ) {
                curBal = 0;
        }
}

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

Constructors in Extended Classes (contd…)

```
class BankAccount {
        protected float curBal;
        protected String name;
        public BankAccount (float amt) {
                curBal = amt;
        }
}
public class SavingsAccount extends BankAccount {
        private boolean isSalaryAcc;
        SavingsAccount (float amount, boolean isSal) {
                super (amount);
                 isSalaryAcc = isSal;
        }
}
```

Invoking constructor using ‘this’ keyword

```
class BankAccount {
        protected float curBal;
        protected String name;
        public BankAccount () {
                curBal = 0;
        }
}
class SavingsAccount extends BankAccount {
        private boolean isSalaryAcc;
        SavingsAccount (float amount) {
                this (amount, false);
        }
        SavingsAccount (float amount, boolean isSal) {
                curBal = amount;
                isSalaryAcc = isSal;
        }
}
```

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 isSalaryAcc;
          SavingsAccount(String name, float bal,boolean sal){
                super(name,bal);
                isSalaryAcc = sal;
        }
        @Override
          public void print ( ) {
        //super.print( );
          System.out.println ("Is it a Salary Account : " + isSalaryAcc);
        }
}

public class OverridingTest {
        public static void main( String[ ] args){
          SavingsAccount sa = new SavingsAccount("Dinesh",10000,true);
          sa.print( );
        }
}
```

The CurrentAccount class

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

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

Polymorphism

```
class PolymorphismTest {
    public static void main (String [ ] args) {
        BankAccount ba[ ] = {  new SavingsAccount ("Amar",1000,true),
                    new CurrentAccount ("Akbar",2000,5000)
                        };
        System.out.println ("Printing polymorphically");

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

```
Printing polymorphically
customer : Amar
Balance  : 10000.0
Is Salaried : true
----------------------------------------
customer : Akbar
Balance  : 20000.0
OverDraftLimit : 5000.0
----------------------------------------
```

BankList.java

```
public class BankList {
    BankAccount[ ]  accArray;
    int top;

    BankList(int size){
            top= -1;
            accArray = new BankAccount[size];
    }
    public void printAll( ){
            for(BankAccount b:accArray){
                    System.out.println(b);
            }
    }
    public void addNewAccount(BankAccount ba){
            if(top<accArray.length)
                    accArray[++top] = ba;
            }

      }
}
```

Hiding Base Class Fields

```
class InterestCalculation {
        public float interestRate = 8.5f;
        double balance = 10000;

        public void calcInterest( ) {
                System.out.println("Annual Interest1 : "+(balance*interestRate)/100);
        }
}
class NewCalculation extends InterestCalculation{
        public float interestRate = 7.0f;
        double balance = 10000;

        public void calcInterest( ) {
                System.out.println("Annual Interest2 : "+(balance*interestRate)/100);
        }
}
```

Hiding Base Class Fields (Cont…)

```
public class HidingFieldsTest {
    public static void main (String [ ] args) {
        NewCalculation nc = new NewCalculation ( );
        System.out.println ("Calculating with interest Rate : "+nc.interestRate);
        nc.calcInterest( );

        InterestCalculation ic = new NewCalculation( );
        System.out.println ("Calculating with interest Rate : "+ic.interestRate);
        ic.calcInterest( );

    }
}
```

```
Calculating with interest Rate : 7.0
Annual Interest2 : 700.0
Calculating with interest Rate : 8.5
Annual Interest2 : 700.0
```

Making Methods and Classes as final

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

final class Stack {
    ...
}
```

Just Minute…

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

```
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

```
class SavingAccount extends BankAccount {
    private boolean isSalaryAcc;
    public SavingAccount (int id, float balance, boolean isSal) {
        super (id, balance);
        this.isSalaryAcc = isSal;
    }
    @Override
    public float calculateInterest( ) {
        return balance * 0.10f;
    }
}
```

Extending an Abstract Class (Cont…)

```
class LoanAccount extends BankAccount{
    private float loanAmt;
    public LoanAccount (int id, float balance, float loanAmt) {
        super (id, balance);
        this.loanAmt = loanAmt;
    }
    @Override
    public float calculateInterest( ) {
        return balance * 0.13f;
    }
}
```

Extending an Abstract Class (Cont…)

```java
class CurrentAccount extends BankAccount{
        private float overDraft;
        public CurrentAccount(int id, float balance, float overDraft) {
                super (id, balance);
                this.overDraft = overDraft;
         }
        @Override
        public float calculateInterest( ) {
                return balance * 0.11f;
        }
}
```

Abstract class and Polymorphism

```
public class AbstractTest {
        public static void main (String [ ] args ) {
                showInterest (new SavingAccount (3,5000, true));
                showInterest(new LoanAccount (4,6000, 100000));
                showInterest(new CurrentAccount (5,7000, 200000));
        }
        public static void showInterest(BankAccount account) {
                System.out.println (“Interest :" + account.calculateInterest ( ));
        }
}
```

Interfaces

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

Interfaces (Cont…)

```
class Circle implements Shape{
        private float radius;

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

Interfaces (Cont…)

```
class Rectangle implements Shape{
        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);
        }
}
```

Interfaces (Cont…)

```
class ShapeTest {
        public static void main (String [ ] args) {
                Shape circle = new Circle (10);
                 Shape rect = new Rectangle (5, 4); //Loss coupling
                System.out.println ("Area of circle is " +             circle.area ( ));
                System.out.println ("Periphery of circle is " +        circle.periphery ( ));
                System.out.println ("Area of rectangle is " +          rect.area ( ));
                System.out.println ("Periphery of rectangle is " +  rect.periphery ( ));
        }
}
```

Extending Interfaces

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

Just Minute…

Module 7. Nested Classes

- **Overview**
  - Inner classes
  - Anonymous inner classes

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);
        }
 }
```

```
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...)

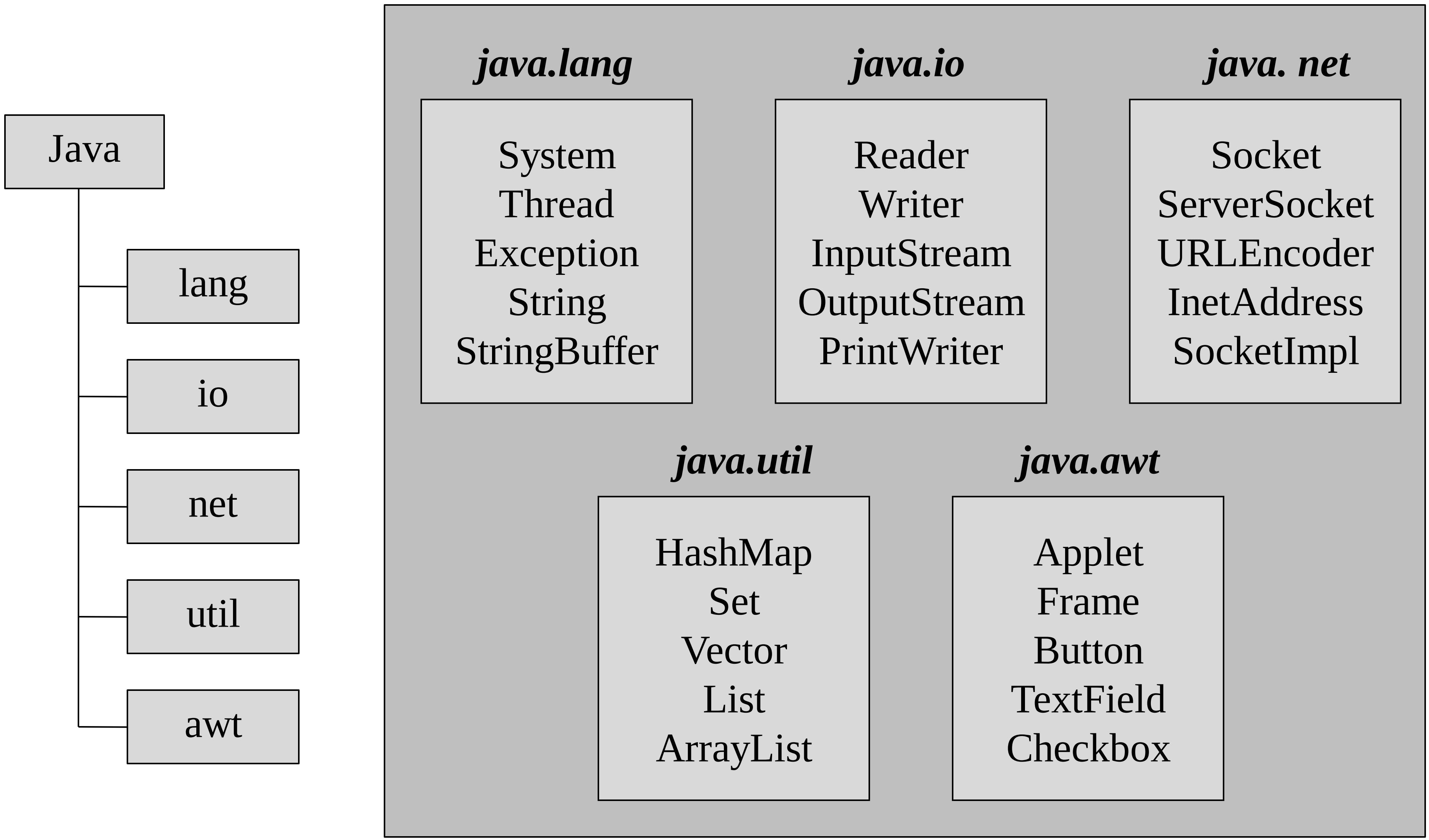
```
public class AnonymousTest {
    public static void main (String [ ] args) {
        Employee e1 = new Employee (10,5000.0f);
        e1.showEmployee ( );
        Employee e2 = new Employee (11,6000.0f) {
            float bonus = 500;
            void showEmployee ( ) {
                super.showEmployee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

Just Minute…

Module 8. Packages

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

Java in-built packages



Creating Packages

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

Naming Packages

com.pragatisoftware.graphics

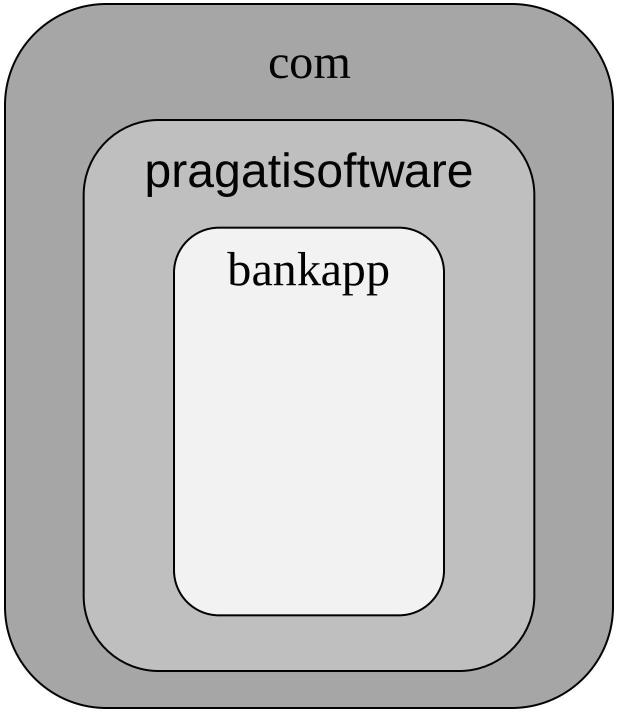
com.pragatisoftware.sql.graphics

Access Specification

Scenarios	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package Subclass	Yes	Yes	Yes	Yes
Same package Non-subclass	Yes	Yes	Yes	Yes
Different package Subclass	Yes	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

Yes (But only through inheritance)

Classpath



Deploy place	Claspath
C:\BankFolder	Set classpath=C:\BankFolder
C:\com\pragatisoftware\BankFolder	Set classpath= C:\com\pragatisoftware\BankFolder

Using a class without importing the package

```
public class ProgramWithoutImport {
    public static void main(String[ ] args){
            Module07.packages.mypackage.AccessSpecifierTest ast = new
      Module07.packages.mypackage.AccessSpecifierTest( );
            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 Module07.packages.mypackage.AccessSpecifierTest;
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 result = Math.sqrt( Math.pow( 4, 2 ) + Math.pow( 5, 2 ) );
                System.out.println(" Result : " + result );
        }
}
```

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

Just Minute…

Module 9. Exceptions

- **Overview**
  - Introduction to Exceptions
  - Unchecked exceptions
  - Checked exceptions
  - The “try-catch” structure
  - The “finally” clause
  - The “throws” clause
  - Custom Exception
  - Exception chaining

Exceptions

```
public class TestExceptions{
        public static void main( String[ ] args ){
         int num1 = 10, num2=0, num3;
           num3 = num1 / num2;
          System.out.println( “Num3:” + num3 );
        }
}
```

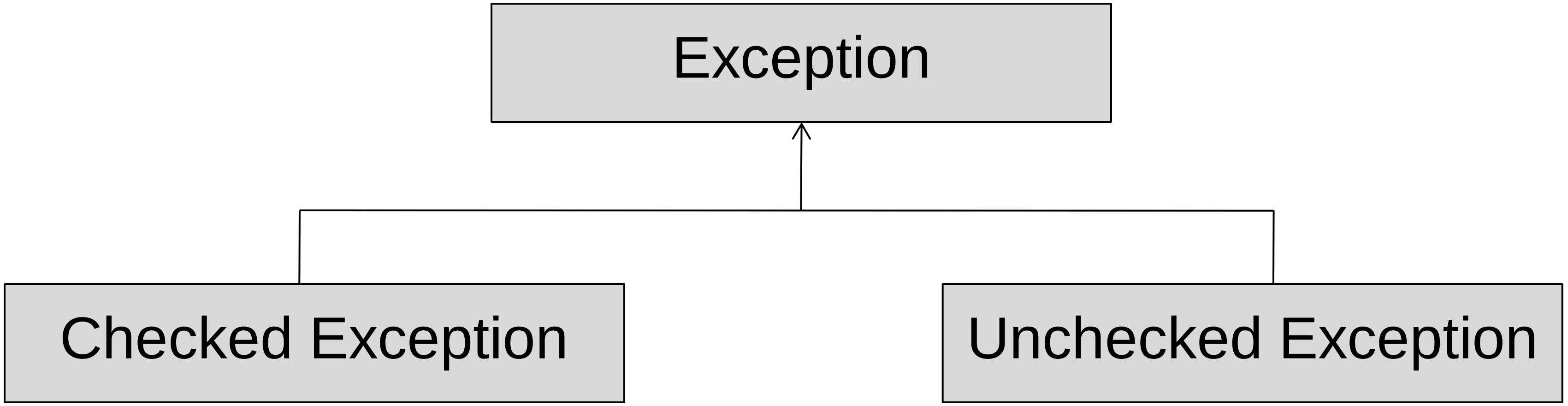
Java-Style Exception Handling Approach

```
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;
    }
}
```

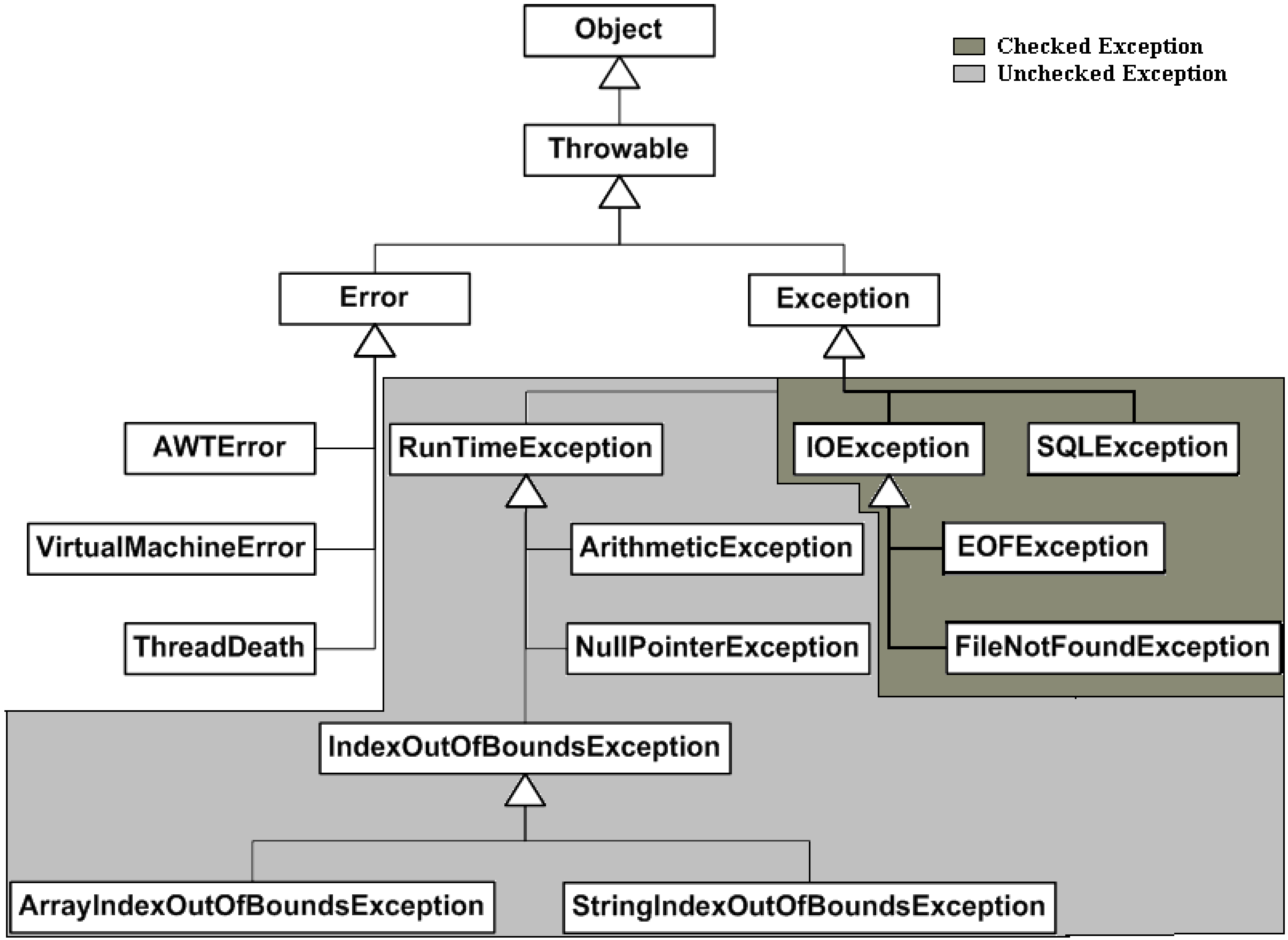
Handling Exceptions

```
public class HandlingExceptions {
        public static void main (String [ ] args ) {
                try {
                        int array [ ] = {10,20,30}, num1 = 10, num2 = 0, num3 = 0;

                        System.out.println ("array [2] " + array [2] );
                        num3 = num1 / num2 ;
                        System.out.println ("Division : " + num3);

                } 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);
                }
        }
}
```

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

```
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 Clause -Unchecked Exception

```
public class ThrowsDemo {
    static float dividingNos(float num1, float num2)throws ArithmeticException{
            try{
                    if( num2 == 0 )
                            throw new ArithmeticException( );
                    else
                            System.out.println( "Printing value....“ );
                    return num1 / num2;
            } finally{ System.out.println( "This will be executed“ );
             }
    }
    public static void main( String[ ] args ){
      try{
            // Accept two values for variables num1 and num2 .
            System.out.println( dividingNos ( num1,num2 ) );
      }
      catch( ArithmeticException ae ){  System.out.println( "Enter number other than zero“ );    }
    }
}
```

Custom Exceptions - Checked exception

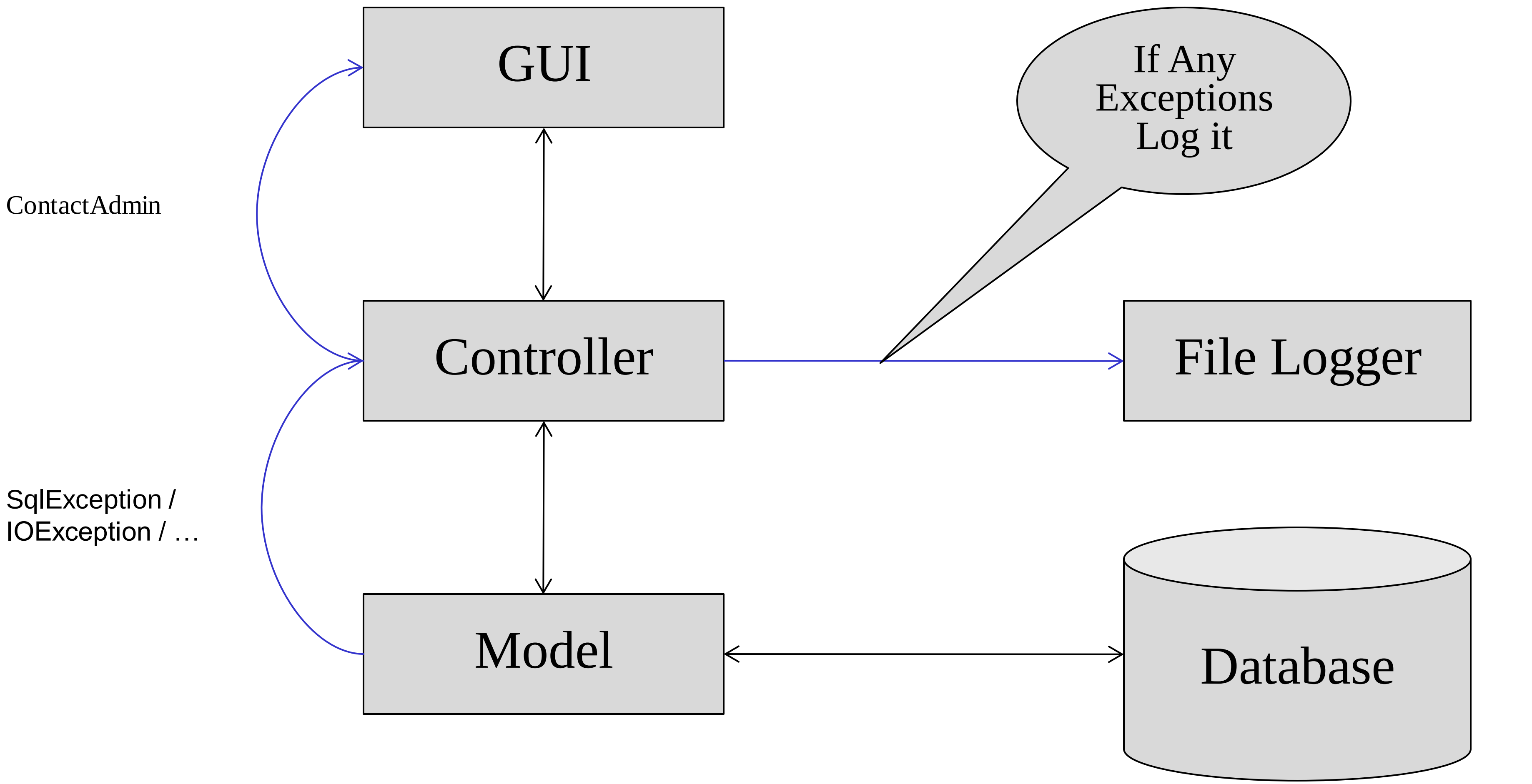
```
class Banking {
    int balance = 10000;
    public void withdraw( float amt ) throws BankException {
        if( ( balance - amount ) < 5000 )
            throw new BankException( );
        else{
            balance -= amount;
            System.out.println( "Sucessfully withdrawn“ );
        }
    }
    public static void main( String[ ] args ){
      Banking banking = new Banking( );
      try {
            banking.withdraw( 6000 );
      } catch (BankException e) { System.out.println( “Not Enough Balance..“ ); }
    }
}
```

```
class BankException extends Exception{

    public String toString( ){
        return "NotEnoughBalance";
    }
}
```

Exception Handling In Inheritance

```
class BankAccount{
	public void withdraw( int amount ){
	…
	}
}

class  CurrentAccount extends BankAccount{
	@Override
	public void withdraw( int amount ) throws  InsufficientBalance{      //compile time error
	….
	}
}

class TestCurrentAccount {
	public static void main( String [ ] args ){
		CurrentAccount currentAccount = new CurrentAccount( );
		currentAccount .withdraw( 2000);
	}
}
```

Custom Exceptions With Layered Diagram



Exception Chaining

```
import java.io.*;

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

Just Minute…

Module 10. Some Useful Built-In Classes

- **Overview**
  - The Object class
  - The String class
  - The StringBuffer class
  - The StringBuilder class

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 EqualsMethodDemo{
        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
                  System.out.println("o1 not equals to o3");
        }
}
```

The equals Method

```
Class BankAccount {
	private int id;
	private float curBal;
	private String name;

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

	public int getId(){
		return id;
	}
```

```
public boolean equals (BankAccount s)
{
	if (getId()== s.getId( )){
		return true;
	else
		return false;
	}
}
```

The equals Method (Cont…)

```
public class EqualsMethodVersion02 {
    public static void main (String [ ] args ) {
        BankAccount sa1 = new BankAccount (100, "jack", 1000.0f);
        BankAccount sa2 = new BankAccount (200, "jill", 3000.0f);

        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) : " +( str.equals(str1) ) );
                System.out.println(" str.equals(str2) : " +( str.equals(str2) ) );
                System.out.println(" str.equals(str2) : " +( str.equals(str3) ) );
        }
}
```

Searching in a String

Rtn Type	Method	Returns index of
int	indexOf (char ch)	first position of ch
int	indexOf (int ch, int start)	first position of ch >= start
int	indexOf (String str)	first position of str
int	indexOf (String str, int start)	first position of str >= start
int	lastIndexOf (char ch)	last position of ch
int	lastIndexOf (char ch, int start)	last position of ch <= start
int	lastIndexOf (String str)	last position of str
int	lastIndexOf (String str, int start)	last position of str <= start

Methods in a String

Rtn Type	Method	Returns index of...
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

Rtn type	Method	Returns index of...
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 num = 16;
		StringBuffer str = new StringBuffer ();
		str.append ("Square root of ").append (num).append (" is");
		str.append(Math.sqrt (num));
		System.out.println (str);
	}
}
```

StringBuffer Class

Rtn Type	Method	Returns Index of…
StringBuffer	append(String str)	Appends the specified string to this character sequence
int	capacity()	Returns the current capacity
char	chatAt(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 num = 16;
          StringBuilder str = new StringBuilder ();
          str.append ("Square root of ").append (num).append (“  is ");
          str.append(Math.sqrt (num));
          System.out.println (str);
        }
}
```

Class String Vs StringBuffer

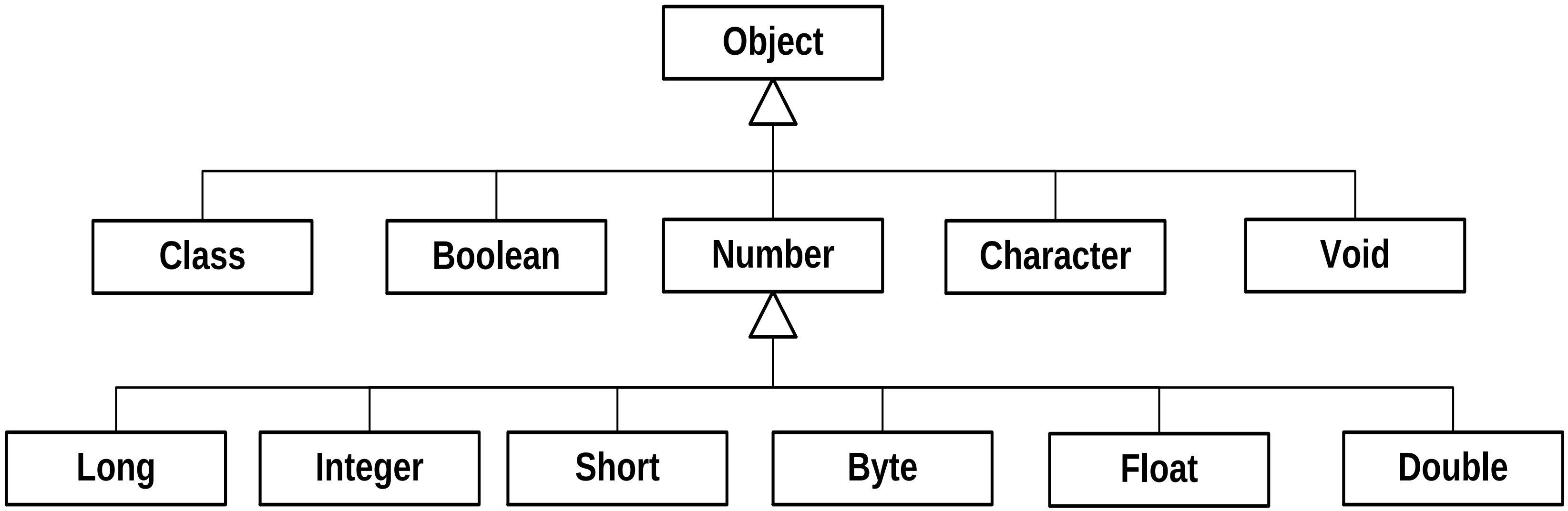
String	StringBuffer
String Class is immutable	StringBuffer class is not immutable
Overrides default equals() method	Does not overrides default equals() method
Is not thread safe	Is thread Safe
does not have a append() method to append more data	does have a append() method to append more data

Just Minute…

Module 11. Wrapper Classes and Autoboxing

- **Overview**
  - Wrapper Classes
  - Auto-Boxing and Un-Boxing
  - Utility Classes

Wrapper Classes



Wrapper Classes (Cont…)

```
public class WrapperTest
{
    public static void main (String args[ ]) {
        int num=10;

        fun(num);
        Integer numRef = new Integer (num);  //wrap the num into Integer class
        fun (num);                           //valid call to fun
    }
    static void fun (Integer numRef) {
        System.out.println ("The value is : " + numRef.intValue ());
    }
}
```

Invalid call to the argument
JDK 1.4

Wrapper Classes (Cont…)

```
java.util.ArrayList employeeIds = new java.util.ArrayList();

for (int counter = 1; counter <= 10; counter++ ) {
        employeeIds.add( counter ) ;
}
```

Autoboxing

```
class AutoBoxTest {
  public static void main (String args[ ]) {
      Integer numRef  = 100;              // auto-boxing an int into an Integer object
      int numPrim         =  numRef;        // auto-unboxing an Integer object ;
      System.out.println(numPrim + " " + numRef);
  }
}
```

Autoboxing in Method Calls

```
class  AutoBoxUsingMethods
{
        public static int getValue(Integer numRef) {
        return numRef;
        }

        public static void main (String args[ ]) {
                int numPrim = getValue(300);
                System.out.println(numPrim);
        }
}
```

Utility Classes – Calendar Class

```
Calendar date1  = Calendar.getInstance();
Calendar date2  = Calendar.getInstance();

date1.set(2012, 1, 1);
date2.set(2012, 3, 29);

Date startDate   = date1.getTime();
Date endDate     = date2.getTime();

long startTime   = startDate.getTime();
long endTime     = endDate.getTime();
long diffTime    = endTime - startTime;
long diffDays    = diffTime / (1000 * 60 * 60 * 24);

System.out.println("Date difference between date " +
 startDate + " end date " + endDate + " = " + diffDays);
```

Utility Classes – Arrays class

Rtn Type	Method	Description
Static int	binarySearch(byte[] a, byte key)	Searches the specified array of bytes for the specified value using the binary search algorithm.
Static int	binarySearch(double[] a, double key)	Searches the specified array of doubles for the specified value using the binary search algorithm.
static double	copyOf(double[] original, int newLength)	Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static char	copyOfRange(char[] original, int from, int to)	Copies the specified range of the specified array into a new array.
static boolean	equals(double[] a, double[] a2)	Returns true if the two specified arrays of doubles are equal to one another.
static void	sort(double[] a)	Sorts the specified array into ascending numerical order.
static void	sort(double[] a, int fromIndex, int toIndex)	Sorts the specified range of the array into ascending order.

Just Minute…

Module 12. Collections

- **Overview**
  - Introduction to Collection
  - Java Collection API
  - Iterators
  - List Interface
    - ArrayList
    - LinkedList
    - ArrayList vs LinkedList
  - Set Interface
    - HashSet
    - SortedSet Interface
    - TreeSet
  - Maps
  - Collections utility class

Collections

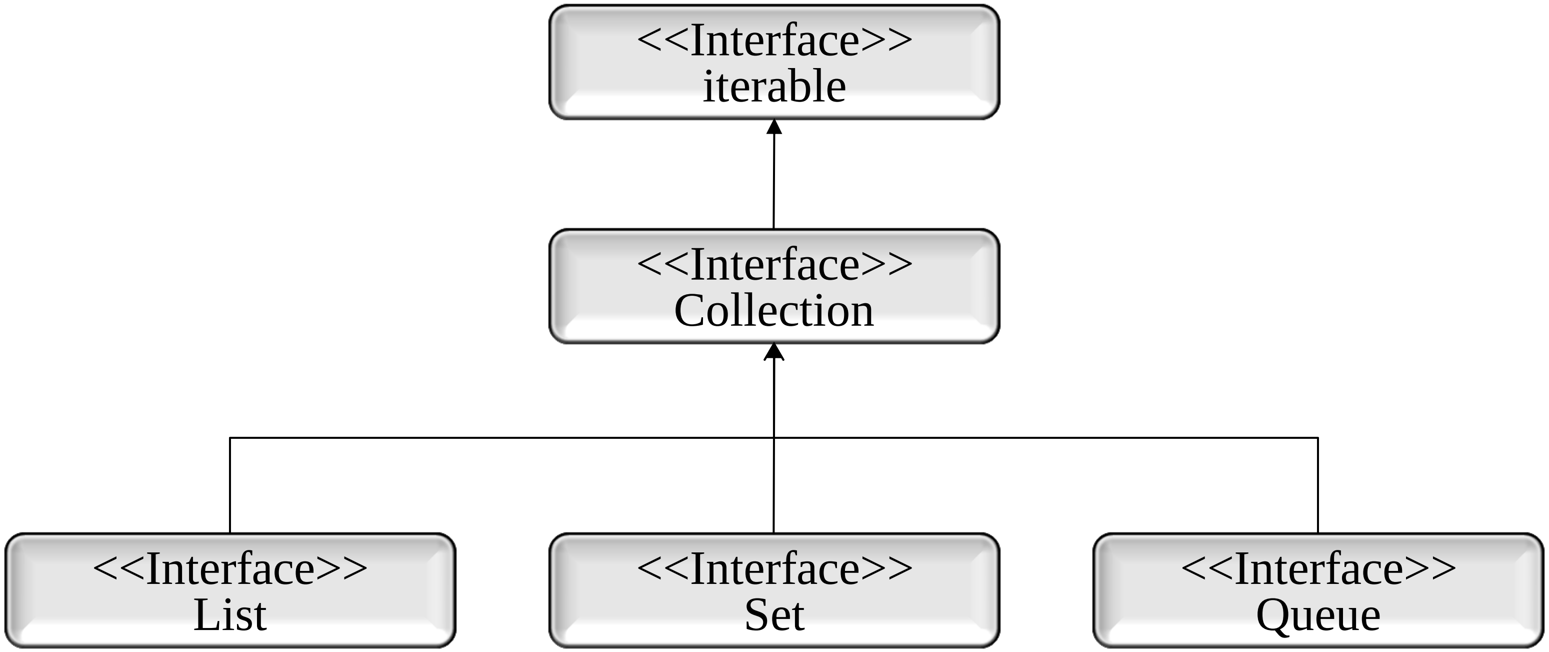
Suppose a Company ABC has many branches spread across different states. These branches send their daily transaction information to a centralized database. Here the data is send over a network and we do not know how much data is going to come at the common location. Where do we hold all the information whose size is unknown?

What is Collection

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.

- They typically represent data items that form a natural group, e.g.
  - poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).

Collection API

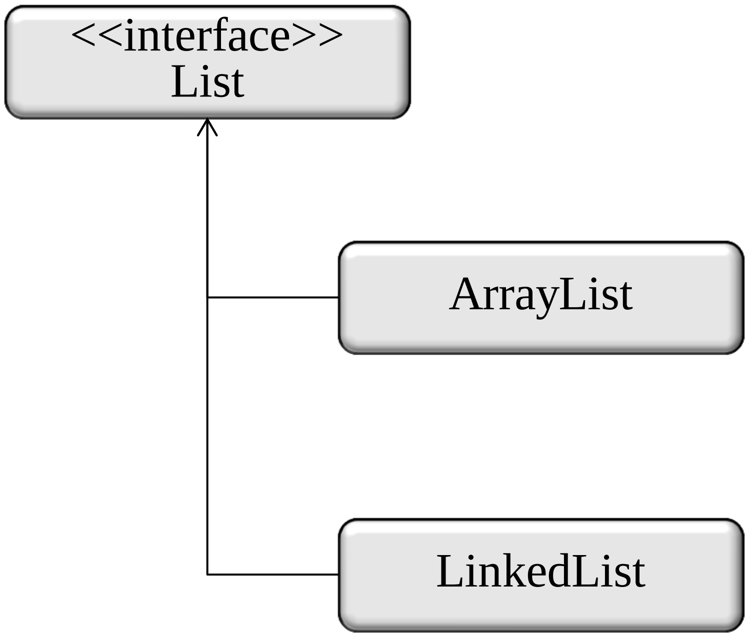


List
An ordered collection
allow duplicate elements

Set
Does not allow duplicate elements

Queue
Order elements in a FIFO (first-in-first-out) manner.

List Interface

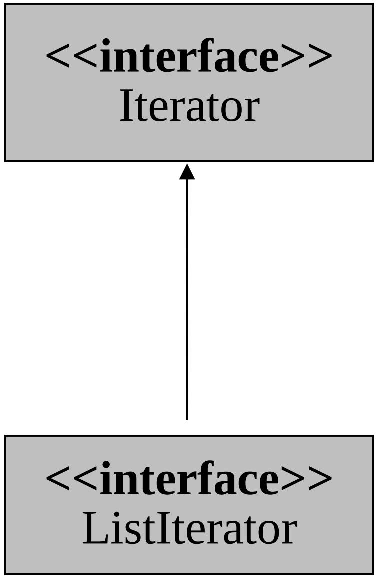


- An ordered collection (also known as a *sequence*).
- The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
- lists typically allow **duplicate elements**.

Methods in List Interface

Rtn Type	Method	Description
boolean	add(Object o)	Appends the specified element to the end of this list
boolean	addAll (Collection c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator
Object	get(int index)	Returns the element at the specified position in this list.
Object	remove(int index)	Removes the element at the specified position in this list
Object	set(int index, Object element)	Replaces the element at the specified position in this list with the specified element
List	subList (int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
ListIterator	listIterator()	Returns a list iterator of the elements in this list (in proper sequence).

Iterators



Class ArrayList

- Implementation of the List interface
- Also know as "Resizable-array"
- Implements all optional list operations, and permits all elements, including null.
- Provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

ArrayList - Demo

```
import java.util.ArrayList;
class BankAccount
{
    double balance; long accountId; String name;

    public BankAccount(double balance, long accountId, String name) {...}

    @Override
    public boolean equals(Object obj){
      return accountId == ((BankAccount)obj).accountId;
    }

    //Since we are overriding equals method it is recommended to override the
    //hashcode too.
    @Override
    public int hashCode(){
      return (int)accountId;
    }
    @Override
    public String toString() {...}
}
```

ArrayList - Demo

```
public class TestArrayList {

    public static void main(String[] args) {
      ArrayList arrayList = new ArrayList();

      arrayList.add(new BankAccount(20000, 12345, "ABCD"));
      arrayList.add(new BankAccount(23000, 56754, "PQRS"));
      arrayList.add(new BankAccount(20000, 87943, "WXYZ"));

      printCollection(arrayList);
      arrayList.add(2, new BankAccount(54000, 23433, "JURH"));

      printCollection(arrayList);

      arrayList.remove(1); // index

      printCollection(arrayList);
    }

    private static void printCollection(ArrayList arrayList) {
      System.out.println(arrayList);
    }
}
```

Class LinkedList

```
package collection;

import java.util.LinkedList;
import java.util.ListIterator;

public class TestLinkedList {
    public static void main(String[] args)     {
      LinkedList linkedList = new LinkedList();

      linkedList.add(new BankAccount(20000, 123456, "AAA"));
      linkedList.addFirst(new BankAccount(23000, 654645, "CCC"));
      linkedList.addLast(new BankAccount(16000, 565656, "PPP"));
      linkedList.add(new BankAccount(76000, 324563, "DDD"));

      //gets the reference of the object without removeing it
      BankAccount ba = (BankAccount)linkedList.peek();
```

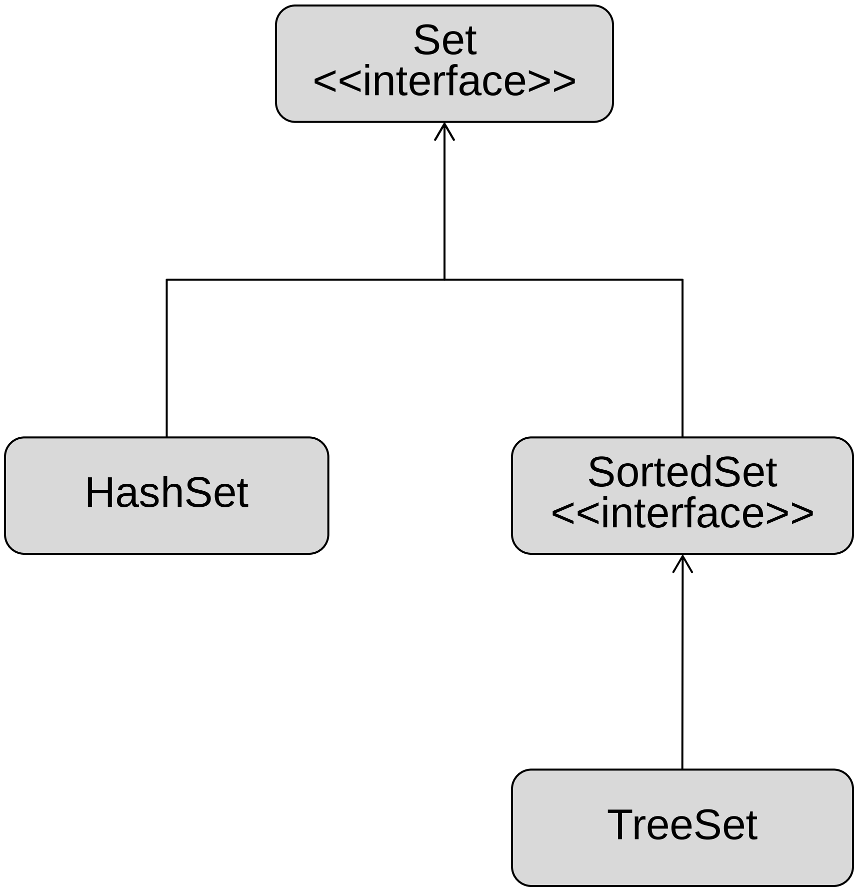
Class LinkedList

```
        //gets the reference of the object and removes from collection
        ba = (BankAccount)linkedList.poll();

        //pops the element and returns the first element on top of the stack
        ba = (BankAccount)linkedList.pop();

        ListIterator listIterator = linkedList.listIterator();

        //read in forward order
        while(listIterator.hasNext())
            System.out.println(listIterator.next());

        //read in reverse order
        while(listIterator.hasPrevious())
            System.out.println(listIterator.previous());
    }
}
```

LinkedList vs. ArrayList

ArrayList	LinkedList
ArrayList is implementing RandomAccess	LinkdList implementing Queue
Fast Random Access You can perform random access without fearing for performance. Calling get(int) will just access the underlying array.	**No random access** Even though the get(int) is still there, it now just iterates the list until it reaches the index you specified. It has some optimizations in order to do that, but that’s basically it.
Slow manipulation When you’ll want to add a value randomly inside the array, between two already existing values, the array will have to start moving all the values one spot to the right in order to let that happen.	**Fast manipulation** As you’d expect, adding and removing new data *anywhere* in the list is instantaneous. Change two links, and you have a new value anywhere you want it.

Set



- A Set is a Collection that **cannot contain duplicate** elements. It models the mathematical set abstraction.
- The Set interface contains **only methods inherited from Collection** and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the **equals and hashCode** operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Methods Of Set

Rtn Type	Method	Description
boolean	add(Object o)	Adds the specified element to this set if it is not already present (optional operation).
void	clear()	Removes all of the elements from this set (optional operation).
boolean	contains(Object o)	Returns true if this set contains the specified element.
boolean	isEmpty( )	Returns true if the collection has no elements
iterator( )	iterator( )	Returns an iterator over the elements in this set.
int	size()	Returns the number of elements in this set (its cardinality).

Class HashSet

- Implements the Set interface
- Does not guarantee that the order will remain constant over time.
- Permits the null element.
- Offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.
- Note that this implementation is not synchronized.

HashSet - Demo

```
class SavingAccount{
      double balance; long accountId; String name;
      public SavingAccount(double balance, long accountId, String name) {...}

      @Override
      public boolean equals(Object obj)     {
        return accountId == ((SavingAccount)obj).accountId;
      }

      @Override
      public int hashCode() {
        return (int)accountId;
      }

      @Override
      public String toString() {...}
}
```

HashSet – Demo Continued

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

        SavingAccount sa1 = new SavingAccount(20000, 12345, "Rohit");
        SavingAccount sa2 = new SavingAccount(20000, 12345, "Rohit");
        SavingAccount sa3 = new SavingAccount(12000, 56456, "Mohit");

        java.util.Set accounts = new java.util.HashSet();
            accounts.add(sa1);
            accounts.add(sa2);
            accounts.add(sa3);

        java.util.Iterator it = accounts.iterator();

        while(iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

```
Output :
[balance=12000.0, accountId=56456, name=Mohit]
[balance=20000.0, accountId=12345, name=Rohit]
```

TreeSet

- implements the SortedSet interface.
- guarantees that the sorted set will be in ascending element order.
- sorts according to the natural order of the elements if non comparator args constructor is used
- provides way of control sorting by accepting a comparator object in one of the constructor

TreeSet Methods

Rtn Type	Method	Description
Comparator	comparator()	Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.
Object	first()	Returns the first (lowest) element currently in this sorted set.
Object	last()	Returns the last (highest) element currently in this sorted set.
SortedSet	subSet(Object fromElement, Object toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet	tailSet	Returns a view of the portion of this set whose elements are greater than or equal to fromElement

TreeSet Demo

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

        java.util.TreeSet treeSet = new java.util.TreeSet();

        treeSet.add("Sunday");
        treeSet.add("Tuesday");
        treeSet.add("Wednesday");
        treeSet.add("Thrusday");
        treeSet.add("friday");
        treeSet.add("Saturday");

        java.util.Iterator iterator = treeSet.iterator();

        while(iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

Output :

```
Saturday
Sunday
Thrusday
Tuesday
Wednesday
friday
```

Comparable Interface

- Imposes a total ordering on the objects of each class that implements it.
  - This ordering is referred to as the class's natural ordering
  - The class's compareTo() method is referred to as its natural comparison method.

- Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort)

- Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals with the exception of java.math.BigDecimal.

Comparable - Demo

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

        //Since String class implements comparable interface it provides natural ordering
        String[] names = new String[]{"Java", "C#", "C", "JavaScript", "Php"};

        //This method tries to sort the elements according to the logic provided while
    implementing comparable interface

    Arrays.sort(names);

        for (String string : names) {
            System.out.println(string);
        }
    }
}
```

Output:
C
C#
Java
JavaScript
Php

Comparator Interface

- Provides a comparison function, which imposes a total ordering on some collection of objects.
- Comparators can be passed to a sort method (such as Collections.sort) to allow precise control over the sort order.
- Comparators can also be used to control the order of certain data structures (such as TreeSet or TreeMap).

Comparator - Demo

```
import java.util.Comparator;

class Employee{
    String name;

    public Employee(String name) {…}

    public String getName() {…}

    public void setName(String name) {…}

    @Override
    public String toString() {
      return "\nname=" + name;
    }
}
```

```
class SortNameInAscendingOrder implements
Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        return ((Employee)o1).getName().
        compareTo(((Employee)o2).getName());
    }
}

class SortNameInDescendingOrder implements
Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        return ((Employee)o2).getName().
        compareTo(((Employee)o1).getName());
    }
}
```

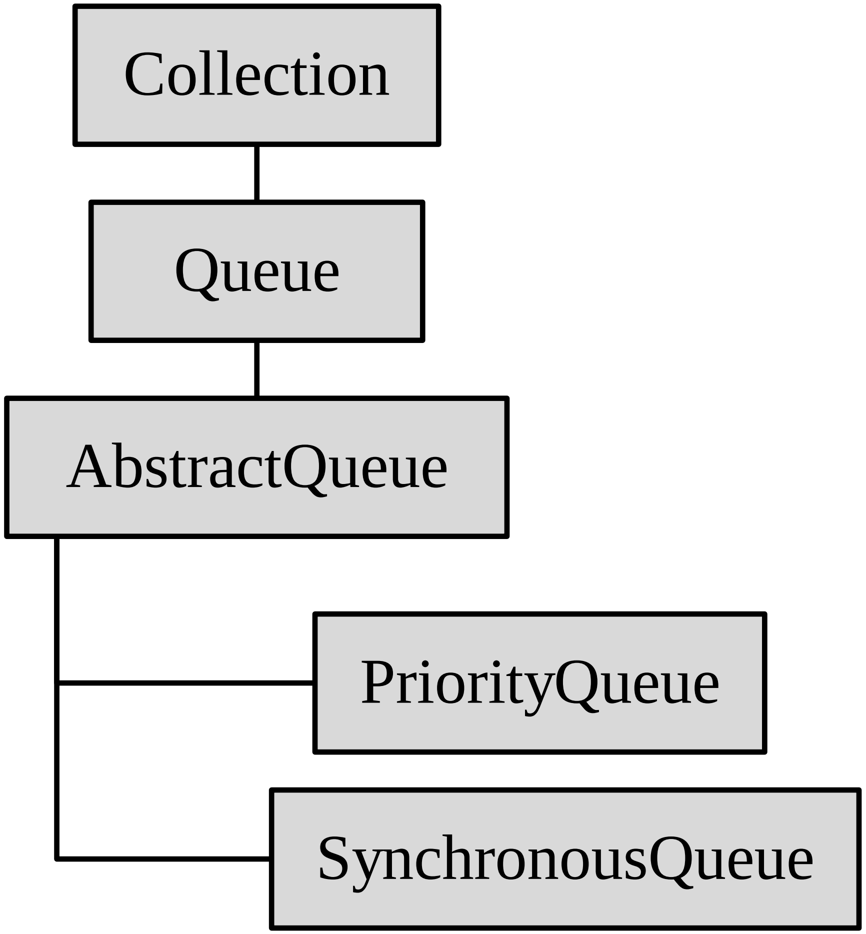
Comparator – Demo Continued

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

        ArrayList employees = new ArrayList();
            employees.add(new Employee("BBB"));
            employees.add(new Employee("MMM"));
            employees.add(new Employee("AAA"));

        //First parameter is collection to be sorted
        //Second parameter is a comparator Object knowing how to sort

        Collections.sort(employees, new SortNameInDescendingOrder());

        System.out.println(employees);
    }
}
```

Output :

[name=MMM,
name=BBB,
name=AAA]

Queue



- A collection designed for holding elements prior to processing.
- Queues also provides additional insertion, extraction, and inspection operations.
- Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner

Methods Of Queue Set

Rtn Type	Method	Description
E	element()	Retrieves, but does not remove, the head of this queue.
boolean	offer(E o)	Inserts the specified element into this queue, if possible.
E	peek()	Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
E	poll()	Retrieves and removes the head of this queue, or null if this queue is empty.
E	remove()	Retrieves and removes the head of this queue.

Note :- E - the type of elements held in this collection

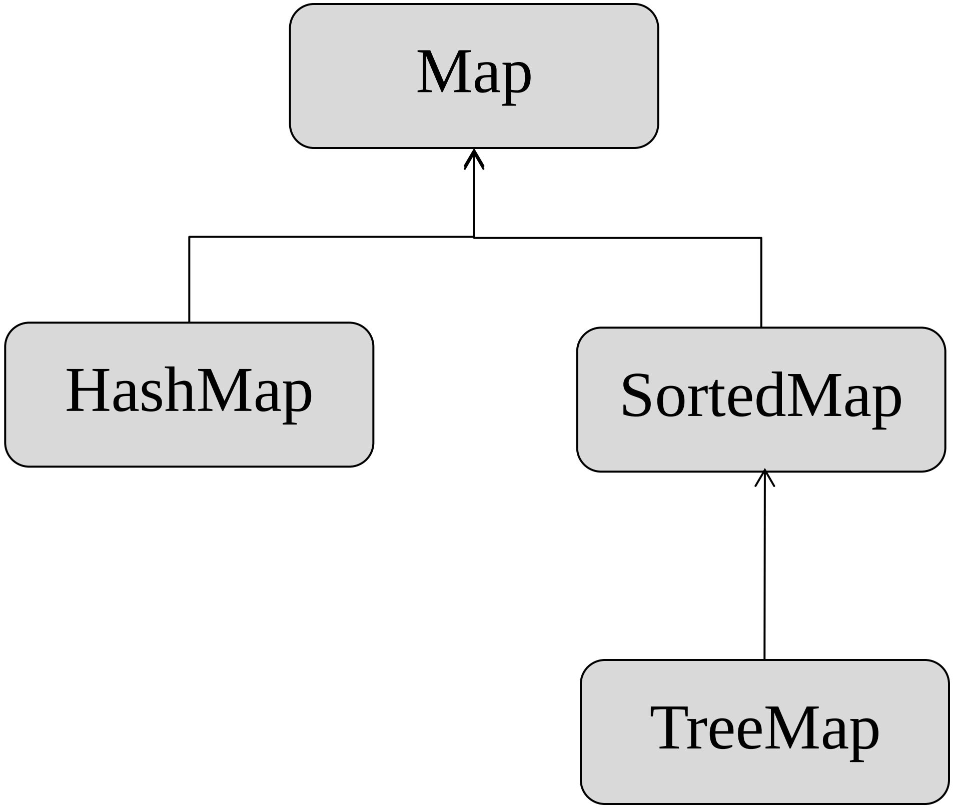
Priority Queue

```
import java.util.PriorityQueue;

public class TestPriorityQueue
{
    public static void main(String[] args)
    {
      PriorityQueue stringQueue = new PriorityQueue();

      stringQueue.add("abc");
      stringQueue.add("ab");
      stringQueue.add("abcd");
      stringQueue.add("a");

/*    The PriorityQueue adds and removes based on
      Comparable; however, if you iterate of the
      PriorityQueue you may not get the results that
      you expect. The iterator does not necessarilly
      go through the elements in the order of their
      Priority. */
```

Output :
a
ab
abc
abcd

```
while(stringQueue.size() > 0)
System.out.println(stringQueue.remove());
        }
}
```

Map



- An object that maps keys to values.
- A map cannot contain duplicate keys;
- each key can map to at most one value.
- The Map interface provides three *collection views*
  - set of keys
  - collection of values
  - set of key-value mappings
- map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Map Methods

Rtn Type	Method	Description
boolean	containsKey	Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
Set	entrySet()	Returns a set view of the mappings contained in this map.
Object	get(Object key)	Returns the value to which this map maps the specified key.
Set	keySet()	Returns a set view of the keys contained in this map.
Object	put(Object key, Object value)	Associates the specified value with the specified key in this map.
Object	remove(Object key)	Copies all of the mappings from the specified map to this map.
Collection	values()	Returns a collection view of the values contained in this map.

Extracting data from HashMap

Map

Keys	**Values**
101	Cake
102	Bread
103	Butter
104	Milk
105	Jam

keyset( )

101	102	103	104	105

values( )

Cake	Bread	Butter	Milk	Jam

getKey( ) **getValue( )**

101	Cake
102	Bread
103	Butter
104	Milk
105	Jam

entrySet( )

HashMap

- Hash table based implementation of the Map interface.
- Permits null values and the null key.
- Equivalent to Hashtable, except that it is unsynchronized and permits nulls.
- It does not guarantee that the order will remain constant over time.
- An instance of HashMap has two parameters that affect its performance
  - Initial capacity
  - Load factor.

HashMapTest.java

```
class maptest{
        int i =20;String s = "xyz";
        /*public boolean equals(Object o){return (this.hashCode()== o.hashCode());}
        public int hashCode(){return i;}      */
}
class HashMapTest {
        public static void main (String [ ] args) {
                Map m =new HashMap ();
                maptest a = new maptest();
                maptest a1 = new maptest();
                m.put(1,a);
                m.put(1,a1);
                m.put ("A","1");
                m.put ("C","2");
                m.put ("B","3");
                System.out.println(m);
                System.out.println ("\nm.size ()= " + m.size ());
```

HashMapTest.java (Cont…)

```
    System.out.println ("Displaying keys: ");
    System.out.println ("\nDisplaying key-values: ");
    Iterator it = m.entrySet ().iterator ();
   while (it.hasNext ()) {
    Map.Entry me = (Map.Entry) it.next ();
    System.out.println (me.getKey () + " " + me.getValue ());
    }
  Set s = m.keySet ();
  it = s.iterator ();
  while (it.hasNext ()) {
            System.out.println (it.next ());
    }
    System.out.println ("\nDisplaying only values: ");
    it = m.values().iterator ();
    while (it.hasNext ())
    System.out.println (it.next ());
  }

}
```

Just Minute…

Module 13. Generics

- **Overview**
  - Generics
  - Generics and Type Safety
  - Generic with two type parameters

ManageAccounts.java

```
class ManageAccounts {
	BankAccount BankList[ ];
	int size, top;

	ManageAccounts(int s) {
		size = s;
		BankList = new BankAccount [size];
		top = -1;
	}

	void addNewAccount(BankAccount account) {
		BankList [ ++top ] = account;
	}

	BankAccount showAccount() {
		return BankList [ top-- ];
	}
}
```

ManageAccounts.java

```
public class TestManageAccounts {
    public static void main (String args[ ]) {
        ManageAccounts ma = new ManageAccounts(2);
        ma.addNewAccount(new BankAccount());
        ma.addNewAccount(new SavingsAccount());

        for(int i=0; i<ma.size; i++) {
            System.out.println(ma.showAccount());
        }
    }
}
```

A Simple Generic Class

```
import java.util.*;

class GenericStack<AnyType> {
	ArrayList<AnyType> buffer;
	int size, top;
	GenericStack(int s) {
		size = s;
		buffer = new ArrayList<AnyType>(size);
		top=-1;
	}
	void addNewAccount(AnyType item) {
		buffer.add(++top,item);
	}
	AnyType getAccount() {
		return buffer.get(top--);
	}
}
```

A Simple Generic Class (Contd…)

```
public class GenericStackTest {
        public static void main (String args[ ]) {
                GenericStack<BankAccount> si = new GenericStack<BankAccount>(2);
                si.addNewAccount(new BankAccount());
                si.addNewAccount(new SavingsAccount());
                for (int i=0; i<si.size; i++) {
                        System.out.println (si.getAccount());
                }
                System.out.println();
               GenericStack<SavingsAccount> sf = new GenericStack<SavingsAccount>(2);
                sf.addNewAccount(new SavingsAccount());
                // sf.addNewAccount(new BankAccount());
                for (int i=0; i<sf.size; i++)
                        System.out.println (sf.getAccount());
        }
}
```

Generics and Type Safety

```
public class GenericTypeSafety {
    public static void main (String args[ ]) {
        GenericStack<Integer> si = new GenericStack<Integer>(3);
        GenericStack<Float> sf = new GenericStack<Float>(3);
        si = sf; // error!
    }
}
```

A Generic Class with Two Type Parameters

```
class Employee{
        public String toString(){
                return "I am an employee";
        }
}
class TwoGeneric<AnyType1, AnyType2> {
        AnyType1  a;      AnyType2   b;
        TwoGeneric(AnyType1 x, AnyType2 y) {
                a = x; b =y;
        }
        void showTypes() {
        System.out.println("Type of AnyType1 : "+a.getClass().getName());
        System.out.println("Type of AnyType2 : "+b.getClass().getName());
        }
        AnyType1 getA() {
        return a;
        }
        AnyType2 getB() {
        return b;
        }    }
```

A Generic Class with Two Type Parameters (Cont..)

```
public class TwoGenericTest {
    public static void main (String args[ ]) {
        BankAccount ba = new BankAccount();
        Employee e = new Employee();
        TwoGeneric<BankAccount, Employee> tg =
                    new TwoGeneric<BankAccount,Employee>(ba,e);
        tg.showTypes();
        BankAccount x = tg.getA();
        System.out.println("Value of x : " +x);
        Employee y = tg.getB();
        System.out.println("Value of y :" +y);
    }
}
```

Language Enhancements (Contd…)

Simplified Generic Handling

Simplified Diamond Operator
Map<String, List<Integer>> trades2 = **new TreeMap<>();**

Old way…
Map<String, List<Integer>> trades1 = **new TreeMap<String, List<Integer>> ();**

Not allowed…
Map<String, List<Integer>> tr = **new TreeMap<String, ArrayList<Integer>> ();**

- The old syntax has redundant generic type. Although compiler can infer right side generic types by referring to the left side.

Just Minute…

Module 14: JUnit & Logging API

- JUnit
  - Introduction to JUnit
  - Test Driven Development
  - Test Suit
  - Terminology
  - Structure of JUnit Class
  - JUnit Testing In Eclipse

- Logging
  - Why Logging
  - When to use Logging
  - Logging Concepts
  - java.util.logging
  - Logging: Best practices
  - Logging: Worst practices

Introduction to JUnit

JUnit :

It is a regression Testing Framework for Java Programming Language.

The JUnit Testing Framework was designed and developed by Kent Beck and Erich Gamma.

Kent Beck

Kent Beck is an American software engineer and the creator of the Extreme Programming

Erich Gamma

Is Swiss computer scientist and co-author of the influential Software engineering textbook, Design Patterns: Elements of Reusable Object-Oriented Software.

Test Driven Development

Test Driven Development (TDD) :

- A software development process that relies on repetition of a very short development life cycle.

- It is related to the *test first* programing of Extreme Programming.

- Requires developers to write testing code (test cases) before writing the actual the application code.

- Frameworks available- Flexunit for Flex, JUnit for java, AspUnit, C++ Test…

Test Driven Development

Test Driven Development Life Cycle :

- **Add a test.**
- **Run all the test and see if new one fails**
- **Write/modify some code**
- **Run the automated test.**
- **Refactor code**
- **Repeat**

Terminology

- **Unit test:**
  - A unit test is a test of a single class
- **Test case:**
  - A test case tests the response of a single method to a particular set of inputs
- **Test suite:**
  - A test suite is a collection of test cases
- **Test runner:**
  - A test runner is software that runs tests and reports results
- **Integration test:**
  - An integration test is a test of how well classes work together reports results

Test Suit

- Test Suit is a collection of test cases
- Sometimes it is also known as Validation suit.

Test Suit

Advantages of Test Suit:

- Finds problem early in development cycle.
- It guarantees that the code tested works now and in the future.
- Developers will be confident if they need to change any old code.
- Test suit reduces the need of manual testing.

Extreme Programming

In the Extreme Programming approach:

- Tests are written before the code itself
- If code has no automated test case, it is assumed not to work
- A test framework is used so that automated testing can be done after every small change to the code
- If a bug is found after development, a test is created to keep the bug from coming back

Structure Of JUnit TestCase Class.

JUnit TestCase Class Constructors And Methods:

Return Type	Method	Detail
	TestCase()	No-arg constructor to enable serialization.
	TestCase(String name)	Constructs a test case with the given name.
Int	countTestCases()	Counts the number of test cases executed by run(TestResult result).
TestResult	createResult()	Creates a default TestResult object
String	getName()	Gets the name of a TestCase
TestResult	run()	A convenience method to run this test, collecting the results with a default
Void	run(TestResult result)	Runs the test case and collects the results in TestResult.

Structure Of JUnit TestCase Class.

JUnit TestCase Class Constructors And Methods:

Return Type	Method	Detail
void	runTest()	Override to run the test and assert its state.
void	setName(String name)	sets the name of a TestCase
Void	tearDown()	Tears down the fixture, for example, close a network connection.
String	toString	Returns a String representation of the TestCase Class.

Class Calculator

```
public class Calculator
{
	public int addition(int num1, int num2){
		return num1 + num2;
	}

	public int substract(int num1, int num2){
		return num1 - num2;
	}

	public int division(int num1, int num2){
		return num1 / num2;
	}

	public int multiplication(int num1, int num2){
		return num1 * num2;
	}
}
```

JUnit Test for Calculator class (JUnit 3)

```
public class TestCalculator extends junit.framework.TestCase
{
    Calculator calculator;

    TestCalculator(){}      //default Constructor

    protected void setUp(){
        calculator = new Calculator();
    }

    protected void tearDown(){
        calculator = null;
    }

    public void testAdd(){
        assertTrue(calculator.addition(2, 2) == 4);
    }
}
```

JUnit Test for Calculator class (JUnit 4)

```
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestCalculatorByJUnit4
{
	Calculator calculator;

	public TestCalculatorByJUnit4(){
	}
```

JUnit Test for Calculator class (JUnit 4)

```
@BeforeClass
public static void setUpBeforeClass() throws Exception {}

@AfterClass
public static void tearDownAfterClass() throws Exception {}

@Before
public void setUp() throws Exception {
        calculator = new Calculator();
}

@After
public void tearDown() throws Exception {
        calculator = null;
}
```

JUnit Test for Calculator class (JUnit 4)

```
    @Test
    public void testAddition() {
        assertEquals(24, calculator.addition(12, 12));
    }

    @Test
    public void testSubstract() {
        assertEquals(10, calculator.substract(12, 2));
    }
}
```

JUnit TestSuit

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

	public static Test suite() {
		TestSuite suite = new TestSuite(AllTests.class.getName());

		//$JUnit-BEGIN$

		suite.addTestSuite(TestCase2.class);

		suite.addTestSuite(TestCase1.class);

		suite.addTestSuite(TestCalculator.class);

		//$JUnit-END$
		return suite;
	}
}
```

Class Calculator (Modified To Throw Exception)

```
public class Calculator
{
    public int addition(int num1, int num2){
        return num1 + num2;
    }
    public int substract(int num1, int num2){
        return num1 - num2;
    }
    public int multiplication(int num1, int num2){
        return num1 * num2;
    }
    public int division(int num1, int num2)throws IllegalArgumentException{
        if(num2 < 1)
            throw new IllegalArgumentException();

        return num1 / num2;
    }
}
```

JUnit Exception Handling - JUnit 3

```
package util;

import junit.framework.TestCase;

public class TestCalculatorException_1 extends TestCase
{
    Calculator calculator;

    protected void setUp() throws Exception {

        calculator = new Calculator();
    }


    protected void tearDown() throws Exception {

        calculator = null;
    }
```

JUnit Exception Handling - JUnit 3

```
    public void testPassingZeroAsSecondArgumentToDivisiong()
    {
        try {
            calculator.division(12, 0);
            fail();
        }
        catch (IllegalArgumentException e){
            //code working fine
        }
        catch(Exception e){
            fail();
        }
    }
}
```

JUnit Exception Handling - JUnit 4

```
package util;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TestCalculatorException_2
{
    Calculator calculator;

    @Before
    public void setUp() throws Exception {
        calculator = new Calculator();
    }

    @After
    public void tearDown() throws Exception
    {
        calculator = null;
    }
```

JUnit Exception Handling - JUnit 4

```
        @Test(expected=IllegalArgumentException.class)
        public final void testDivision()
        {
                calculator.division(12, 0);
        }
}
```

JUnit Timeout

```
package util;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class TestCalculatorTimeout
{
	Calculator calculator;

	@Before
	public void setUp() throws Exception {
		calculator = new Calculator();
	}

	@After
	public void tearDown() throws Exception {
		calculator = null;
	}
```

JUnit Timeout

```
        @Test(timeout=1000)
        public void testDivision()
        {
              calculator.division(123123, 343434);
        }
}
```

Assert Statement:

There are different types of assert statements to choose from.

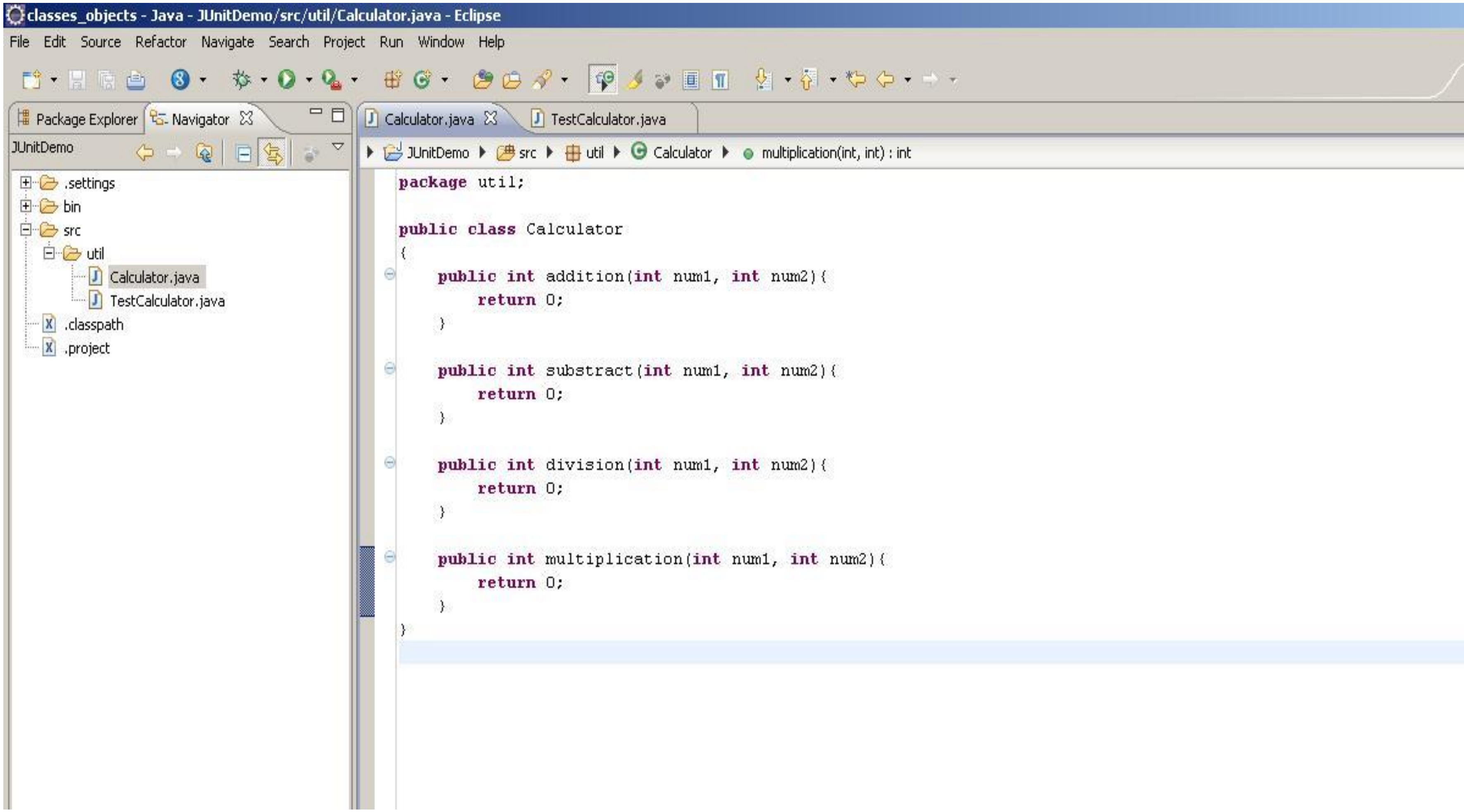
Assert Methods
assertEquals(expected, actual)
assertEquals(message, expected, actual)
assertFalse(condition)
assertFalse(message, condition)
assertNotNull(object)
assertNotNull(message, object)
assertNotSame(expected, actual)
assertNotSame(message, expected, actual)
assertNull(object)
assertNull(message, object)

Assert Statement:

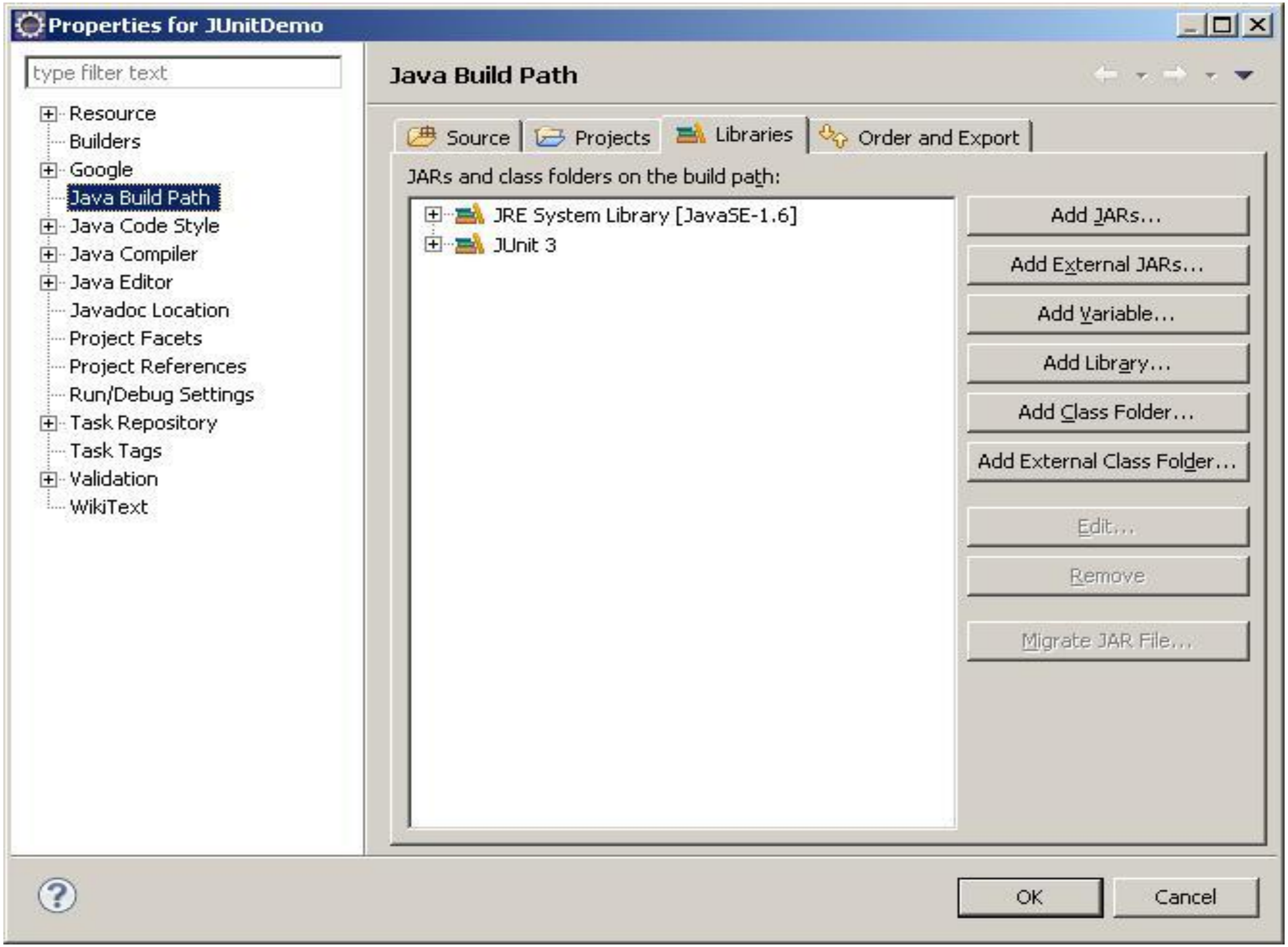
There are different types of assert statements to choose from.

Assert Methods
assertSame(expected, actual)
assertSame(message, expected, actual)
assertTrue(condition)
assertTrue(message, condition)
fail()
fail(message)

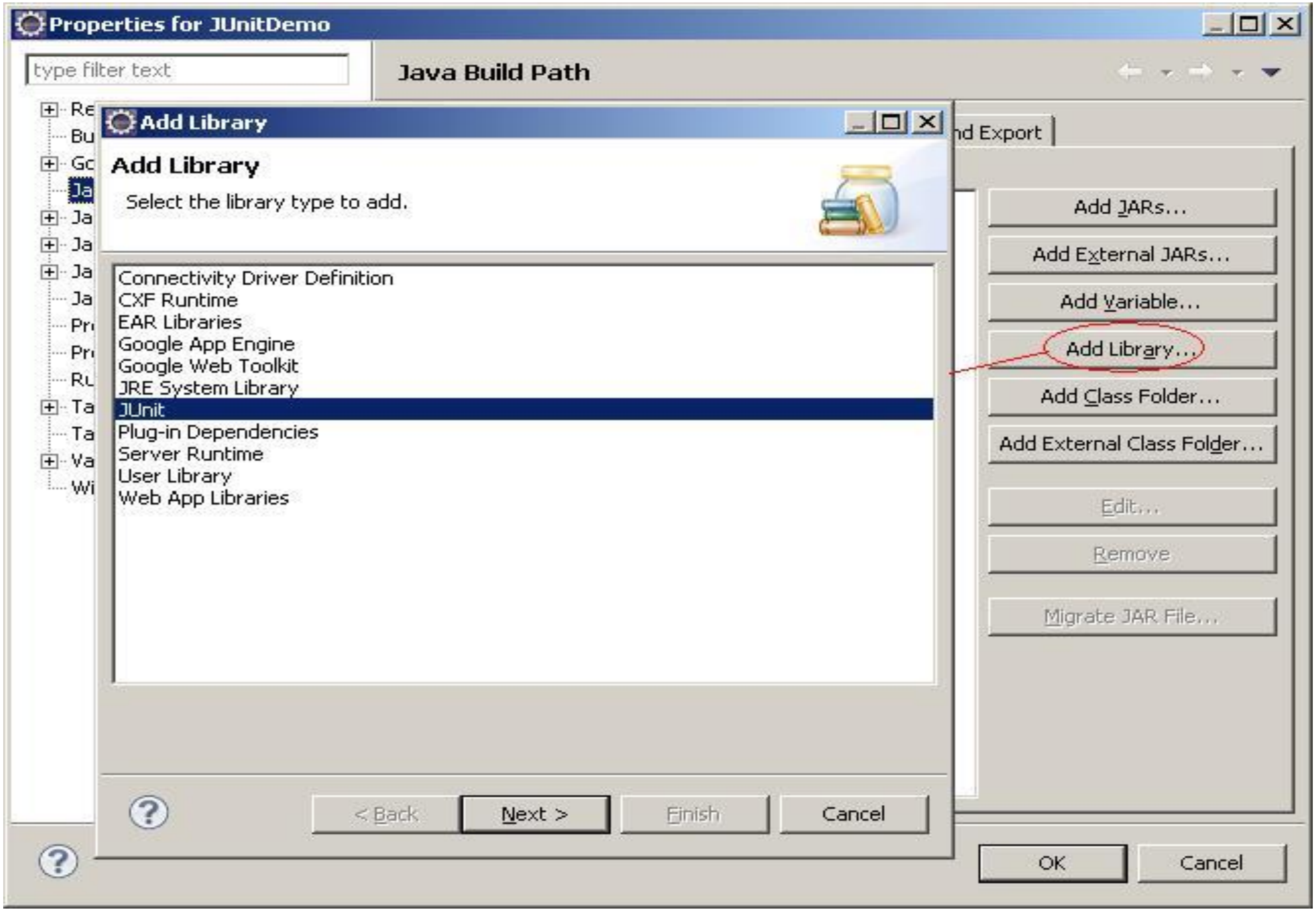
JUnit Testing In Eclipse



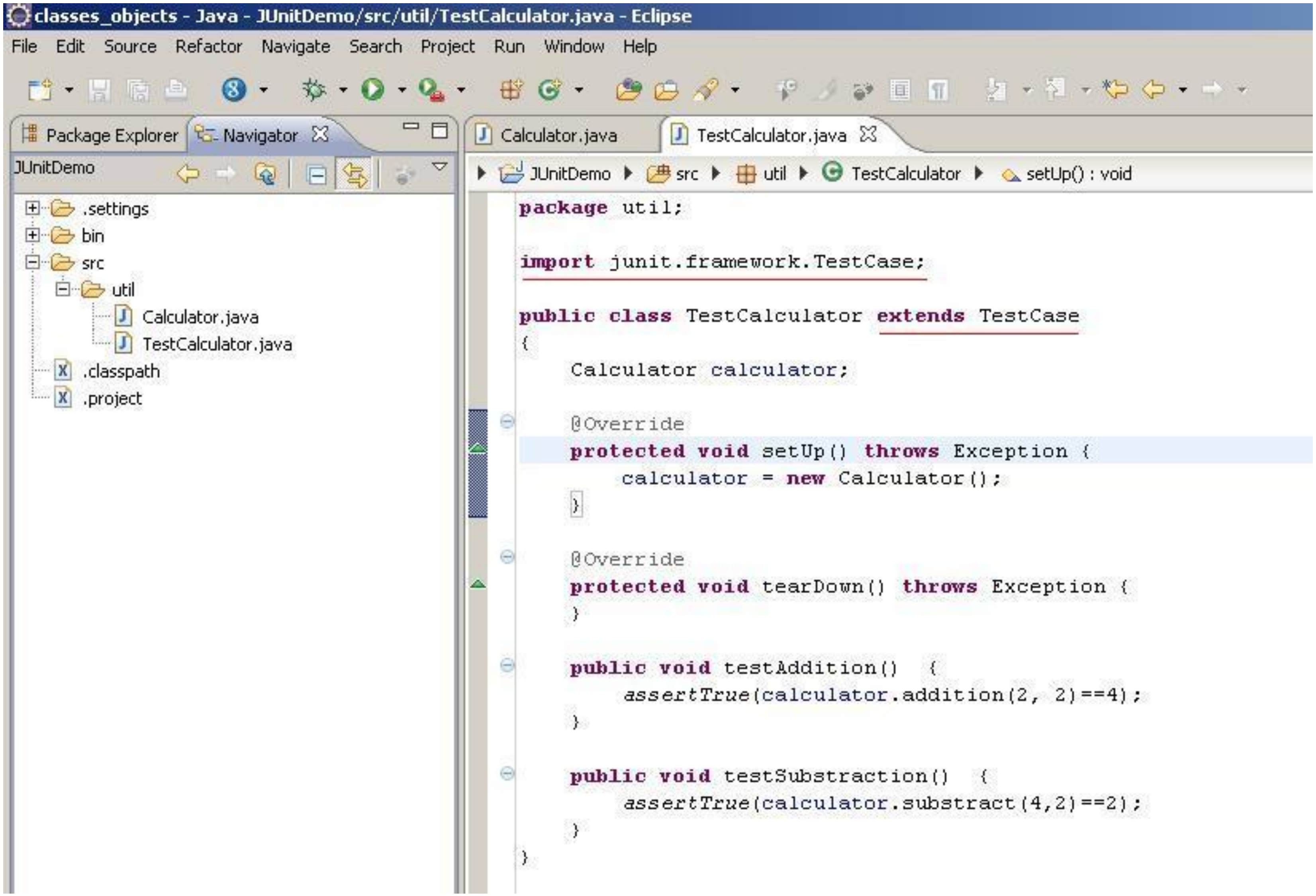
JUnit Testing In Eclipse



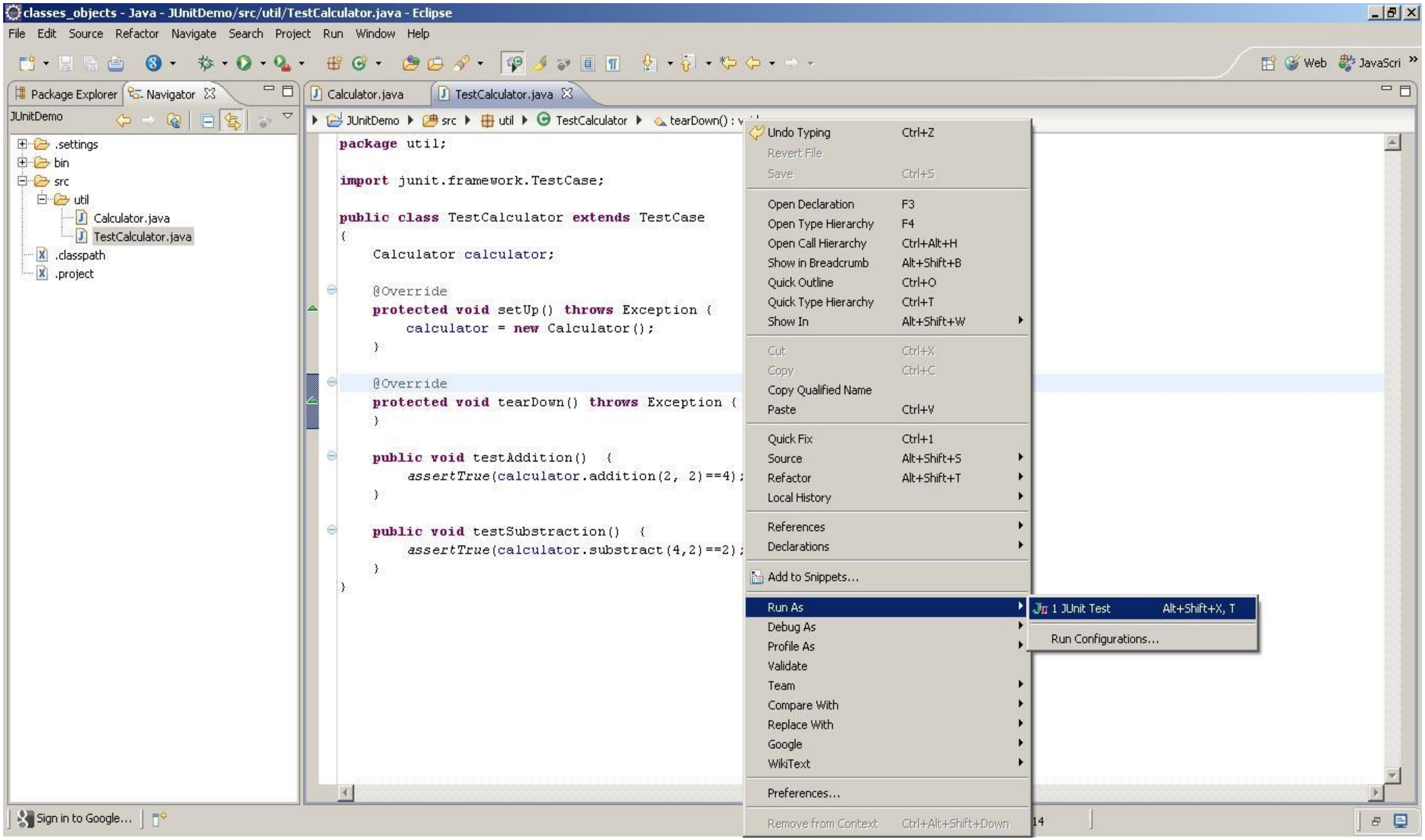
JUnit Testing In Eclipse



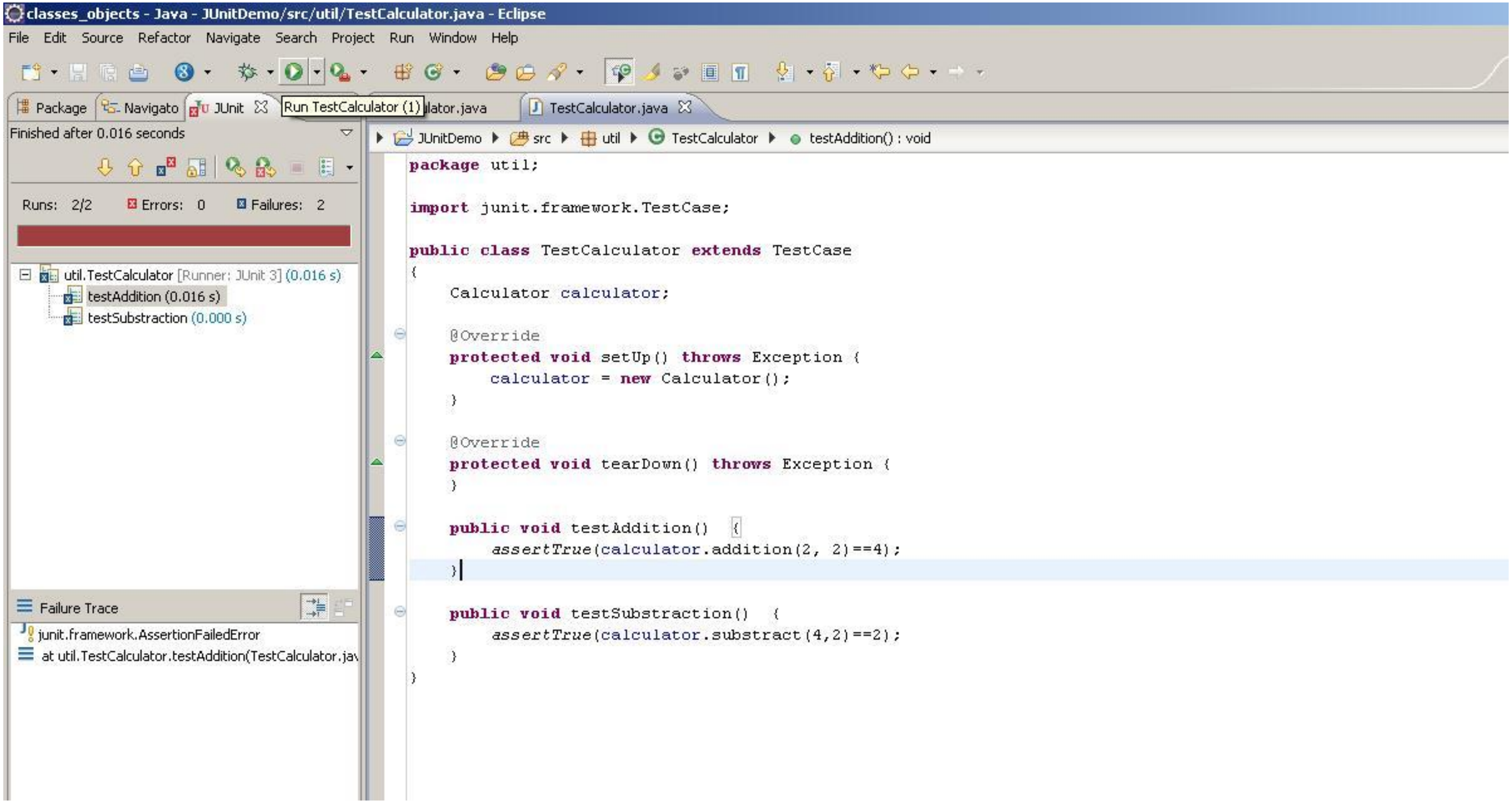
JUnit Testing In Eclipse



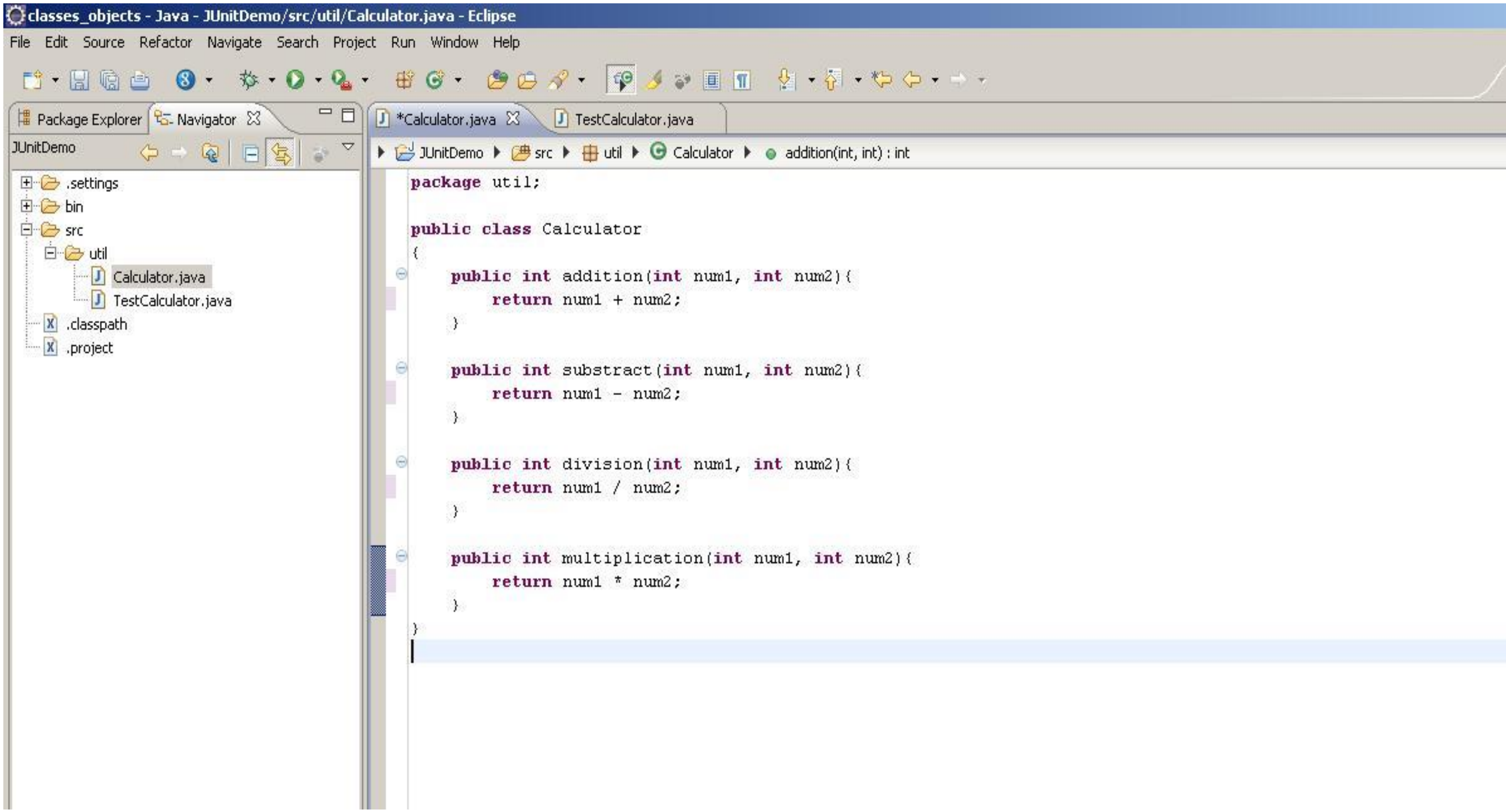
JUnit Testing In Eclipse



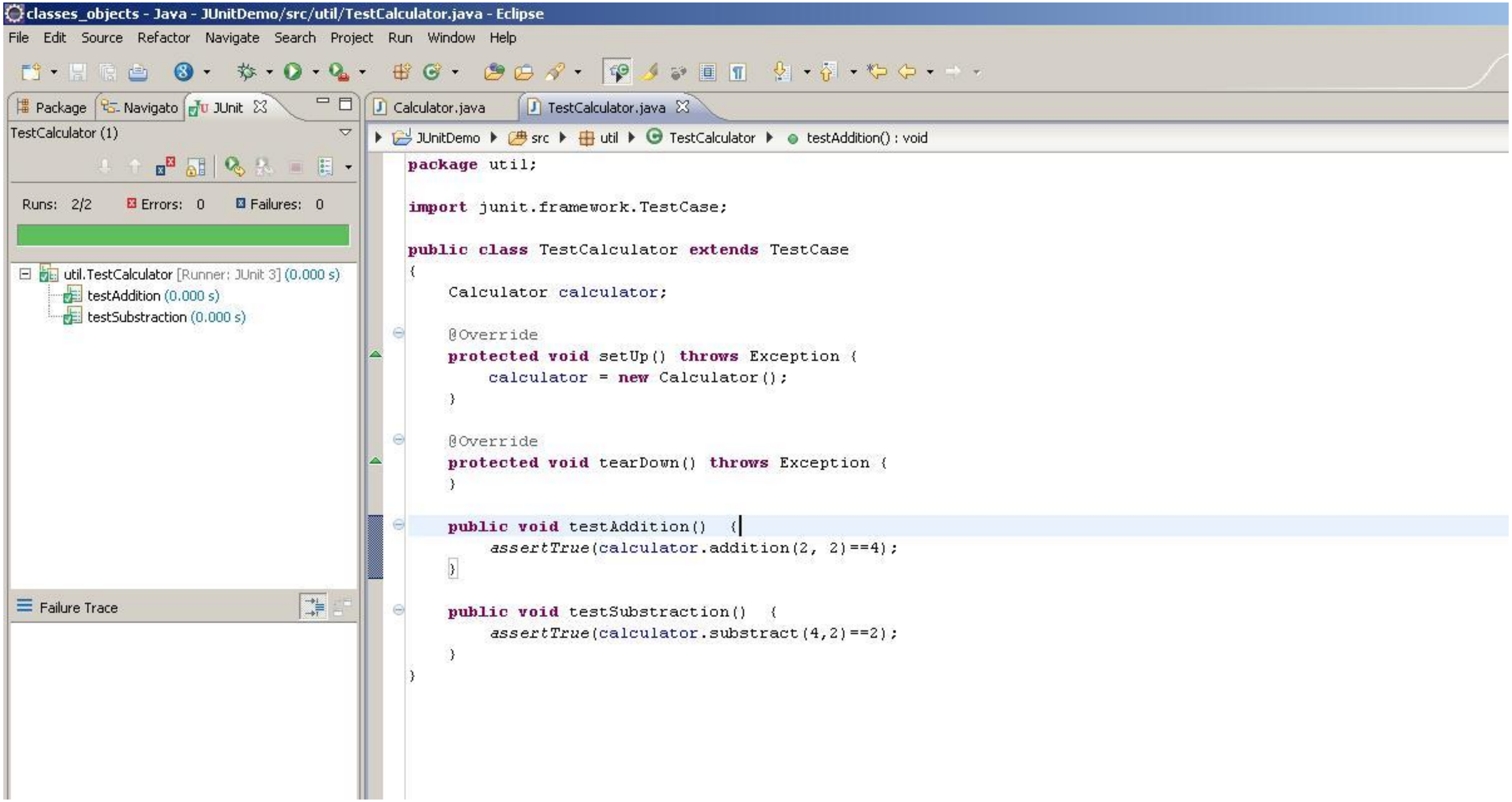
JUnit Testing In Eclipse



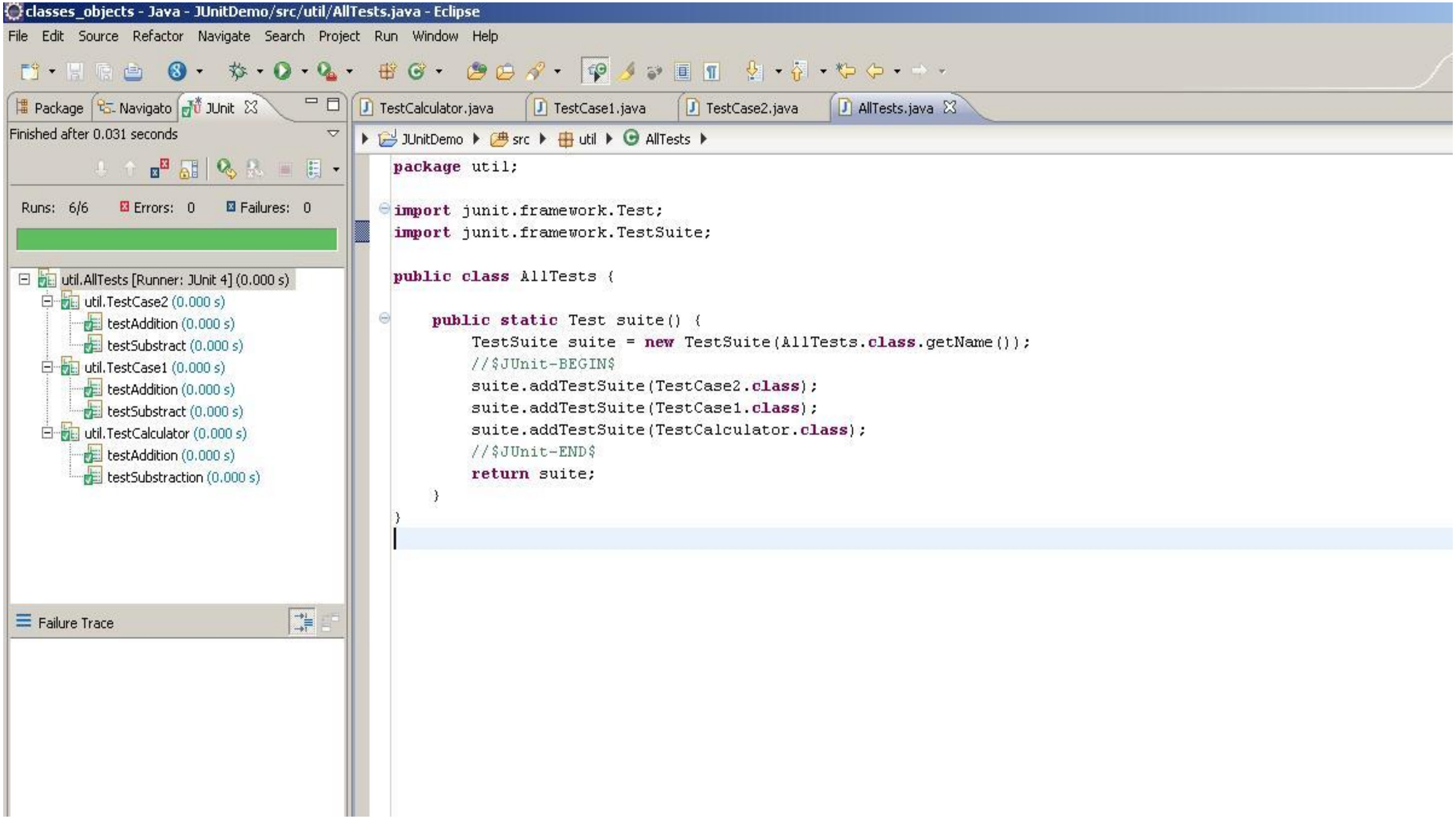
JUnit Testing In Eclipse



JUnit Testing In Eclipse



JUnit Testing In Eclipse



Real-world applications are complex

- Applications are multi-threaded and multi-user.
- Multiple web applications per application server.
- Each web application may communicate with one-or-more backend systems.

Why logging?

- Logs provide precise context about a run of the application.

- Logs can be saved to a persistent medium to be studied at a later time

When to use logging

- In your development phase:
    - logging can help you debug the code
- In your production environment:
    - helps you troubleshoot problems

Hello java.util.logging

```
package packLogging;

import java.util.logging.Level;
import java.util.logging.Logger;

public class HelloUserLogging {
        private static Logger logger = Logger.getLogger("packLogging.HelloUserLogging");

        public static void main(String argv[]) {
          logger.setLevel(Level.ALL);
          logger.fine("Program started");
          logger.info("This app uses java.util.logging");
          logger.warning("Oops, I did it again");
        }
}
```

Output from HelloUserLogging

```
Dec 16, 2011 11:43:24 AM packLogging.HelloUserLogging main
INFO: This app uses java.util.logging
Dec 16, 2011 11:43:24 AM packLogging.HelloUserLogging main
WARNING: Oops, I did it again
```

Logging concepts

- Named Loggers
  - Logger logger = Logger.getLogger("packLogging.HelloUserLogging“);
- Levels
- Destination for log messages
- Message log format

java.util.logging

- **java.util.logging**
  - LogManager class
  - Logger class
  - Named loggers
  - Level class
  - Filter interface

java.util.Logging.Level

- Order:
  - SEVERE (highest value)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (lowest value)

- Other levels:
  - Level.ALL
  - Level.OFF

Logging with Exception

```
package packLogger;
import java.util.logging.*;

public class LogException{
        public static void main(String[] args) {
          Logger log = Logger.getLogger("packLogger.LogException");
          try{
                    DivByZero();
          } catch (Exception e){ log.log(Level.WARNING, "Can not divede by zero", e.getMessage()); }
          try{
                    ArrayBound();
          } catch(Exception ex){ log.log(Level.INFO, "Array is blank", ex.getMessage()); }
        }
        public static void DivByZero() {
          System.out.println(1/0);
        }
        static int arr[];
        public static void ArrayBound() {
          System.out.println(arr[0]);
        }
}
```

Output

```
Dec 16, 2011 1:06:18 PM packLogger.LogException main
WARNING: Can not divide by zero
Dec 16, 2011 1:06:18 PM packLogger.LogException main
INFO: Array is blank
```

java.util.logging Handlers

- StreamHandler
- ConsoleHandler
- FileHandler
- SocketHandler
- MemoryHandler

java.util.logging Formatters

- SimpleFormatter
- XMLFormatter

MyLogger java.util.logging

```
package packLogger;
import java.io.IOException;
import java.util.logging.*;

public class MyLogger {
      public static void main(String[] args) {
        Logger logger = Logger.getLogger("packLogger.MyLogger");
        FileHandler fh;
        try {
                fh = new FileHandler("MyLogFile.log", true);
                logger.addHandler(fh);
                logger.setLevel(Level.ALL);
                //SimpleFormatter formatter = new SimpleFormatter();
                XMLFormatter formatterXML= new XMLFormatter();
                //fh.setFormatter(formatter);
                fh.setFormatter(formatterXML);
                logger.log(Level.WARNING, "My first log");
        } catch (SecurityException e) { e.printStackTrace(); }
          catch (IOException e) {  e.printStackTrace(); }
      }
}
```

Logging: Best practices

- Use the appropriate message level
- Roll your log files daily / weekly
- Review your error log on a daily basis

Logging: Worst practices

- System.out.println / System.err.println
- logging passwords to a log file
- logging informational messages to STDERR
- logging a message for every single HTTP request
- multiple applications sending log messages to a single log file
- ignoring error messages that appear in your application error log
- misleading log messages

Just Minute…

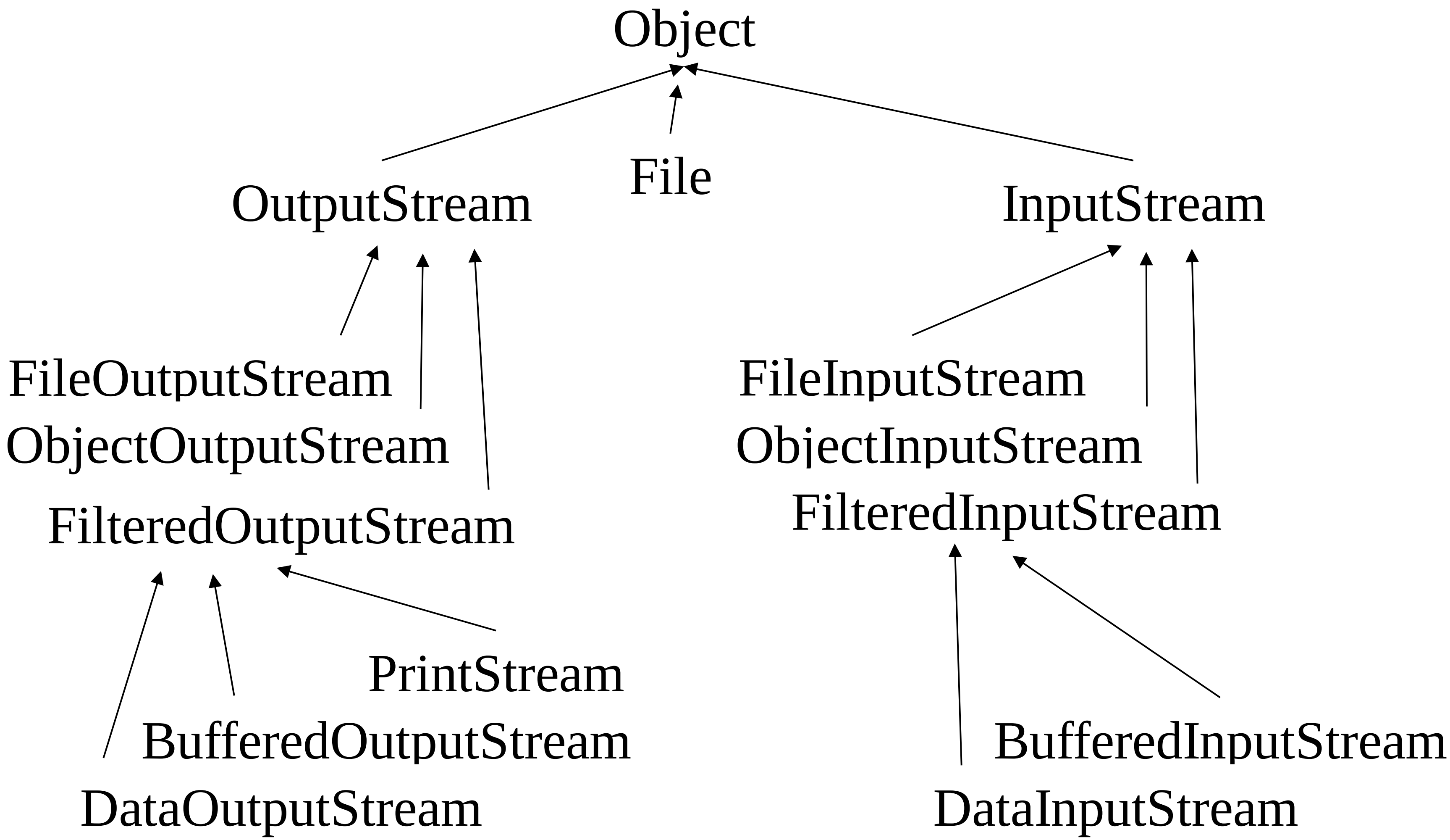
Module 15. File IO

- **Overview**
  - Types of Input and Output Streams
    - Byte-based stream
    - Character-based stream
  - File Class
  - SequenceInputStream
  - Reader and Writers
  - PrintWriter
  - Random Access Files

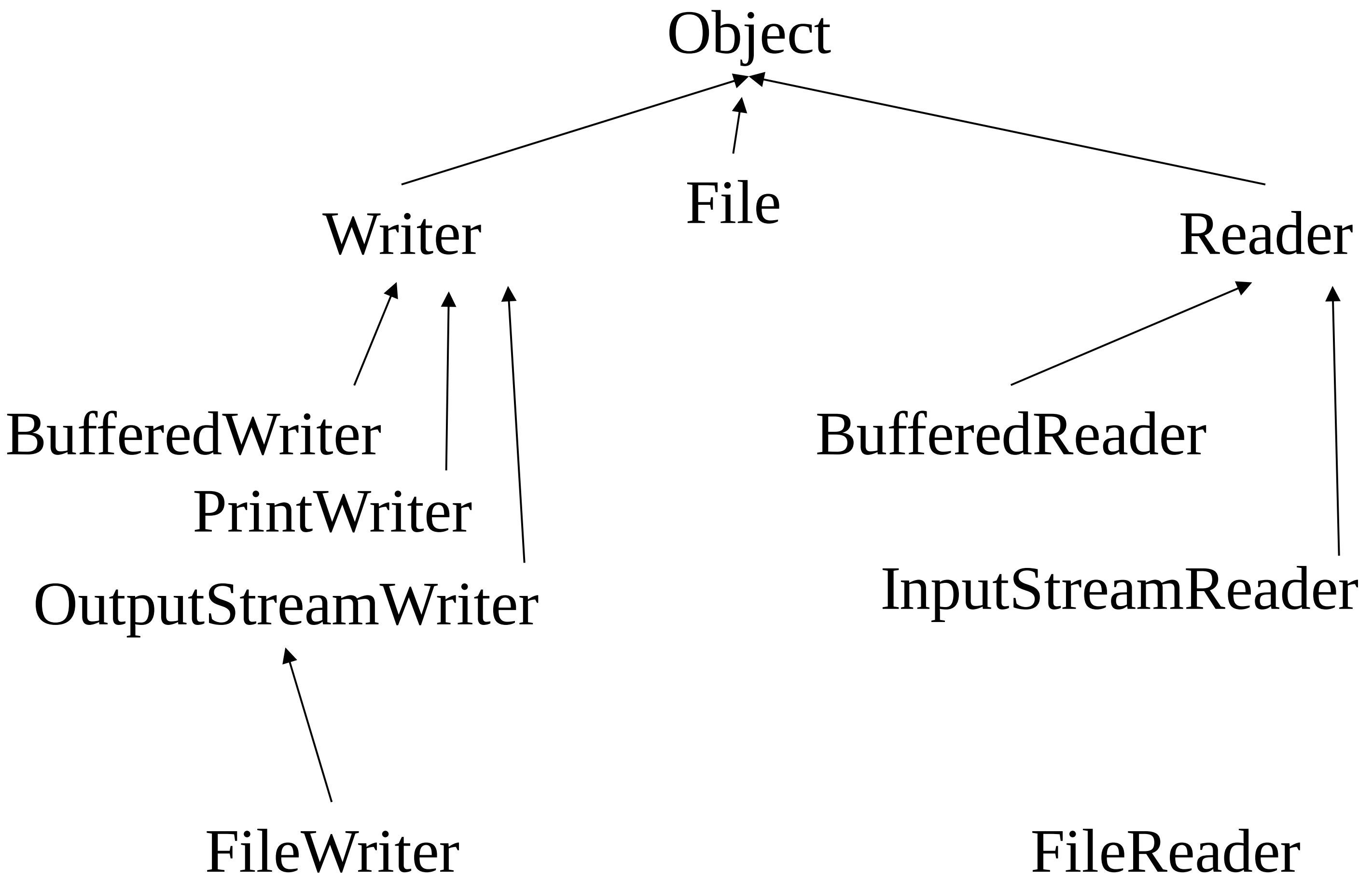
Types of Input and Output Streams

Byte Based Stream	Character Based Stream
Representation unit = 1 **byte** data	Representation unit = 2 **bytes** data (Unicode character)
For Binary data I/O like images, wave files, executable code.	For character based I/O like text files
Also known as input stream & output stream	Also known as Reader and Writer

Partial List for Byte Based Streams



Partial List for Character Based Streams



A File class

```
import java.io.*;  import java.sql.Date;
class FileDemo {
      public static void main(String[ ] args) {
        File f=new File("abc.txt");
        try {   f.createNewFile() ; }
        catch (IOException e) {   e.printStackTrace();    }
        System.out.println("Does the file exists ? "+f.exists());
        System.out.println("Is this a file ? "+f.isFile());
        System.out.println("Is this a directory ? "+f.isDirectory());
        System.out.println("The AbsolutePath is : "+f.getAbsolutePath());
        System.out.println("The file separator : "+f.separator);
        System.out.println("The size of file : "+f.length());
        System.out.println("The file was last modified at : "+new Date(f.lastModified()));
        System.out.println("Is the file Readable ? "+f.canRead());
        System.out.println("Is the file Writable ? "+f.canWrite());
        System.out.println("Is the file Hidden ? "+f.isHidden());
      }
}
```

Reading File Using Byte Stream

```
class ReadImage {
    public static void main(String[ ] args) {
    try {
            FileInputStream fis=new FileInputStream("roses.jpg");
            FileOutputStream fos=new FileOutputStream("NewRoses.jpg");
            int i=fis.read();
            while(i!= -1) {
                fos.write(i);
                i = fis.read();
            }
    }
    catch (FileNotFoundException e) { e.printStackTrace();  }
    catch(IOException e) {  e.printStackTrace();   }
    }
}
```

Reading File Using a Character Stream

```
public class ReadText {
    public static void main (String [ ] args) {
            int i=0;
            try {
                FileReader fr = new FileReader ("File1.txt");
                FileWriter fw= new  FileWriter("NewFile.txt");
                while ((i=fr.read())!=-1) {
                    fw.write(i);
                }
                fr.close();
                fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
}
```

SequenceInputStream

```
public class  SequenceInputStreamTest {
  public static void main (String [ ] args) {
    try {
      FileInputStream f1 = new FileInputStream ("abc.txt");
      FileInputStream f2 = new FileInputStream ("xyz.txt");
      SequenceInputStream seq = new SequenceInputStream (f1,f2);
      int b = seq.read ();
      while (b != -1) {
                System.out.print ( (char) b);
                 b = seq.read ();
        }
      seq.close (); f1.close (); f2.close ();
    } catch (IOException e) { System.out.println ("Error in IO is : " + e);     }
  }
}
```

BufferedReader

```
public class LineReader {
      public static void main(String[ ] args){
        try{
                FileReader fr = new FileReader("File1.txt");
                BufferedReader br = new BufferedReader(fr);

                String s = br.readLine();

                while(s!= null){
                        System.out.println(s);
                        s = br.readLine();
                }
        }
        catch(IOException ioe){
                ioe.printStackTrace();
        }

      }
}
```

BufferedWriter

```
import java.io.*;

public class LineWriter {
    public static void main(String args[ ]){
            try{
                FileWriter fw = new FileWriter("File1.txt",true);
                BufferedWriter br=new BufferedWriter(fw);
                br.newLine();
                br.write("using BufferedWriter");

                br.flush();
                br.close();
                fw.close();

            }
            catch(IOException ex){
                ex.printStackTrace();
            }
     }
}
```

PrintWriter

```
import java.io.*;
public class  PrintWriterDemo{
        public static void main (String [ ] args) {
                PrintWriter pw = new PrintWriter (System.out);
                pw.println ("hello");
                pw.println ("Hi");
                pw.flush ();
        }
}
```

Random Access Files

```
class RandomAccessFileDemo {
    RandomAccessFile raf;

    RandomAccessFileDemo(){
        try{
            raf=new RandomAccessFile("empTable.txt","r");
        }
        catch(FileNotFoundException e ){
            e.printStackTrace();
        }
    }

    public void display(String empno){
      try {
        int i=0;
        String[ ] emp=new String[15];
        String empNumber;
```

Random Access Files (Contd…)

```
            while(raf.getFilePointer()<raf.length()){
                emp[i]=raf.readLine();
                empNumber=emp[i].split(":")[0];
                if(empNumber.equals(empno)){
                    System.out.println("employee found...\n"+emp[i]);
                    break;
                }
                i++;
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        RandomAccessFileDemo rafd=new RandomAccessFileDemo();
        rafd.display("1006");
    }
}
```

Language Enhancements (Contd…)

Automatic resource management:

```
try (FileOutputStream fos = new FileOutputStream("movies.txt"); )
{

        // Work with file using 'fos'.

} catch (IOException e) {
        // Handle exception
}
```

- The resource 'try' is managing is implementing 'AutoCloseable'.

- The 'try' invokes close() method on resource through 'AutoCloseable' automatically and guranteedly if both cases of successful execution or if exception is raised.

- This 'try' can manage even more than one resources this way.

The Old try-catch

```
FileOutputStream createFile = null;

try {
        crearteFile = new FileOutputStream("movie.txt");

        // Work with file
} catch (IOException io) {
        // Handle Exception
} finally {
        try {
                createFile.close();
        } catch (IOException ie) {
                // Handle exception for failed file closing.
        }
} // End of finally.
```

Language Enhancements (Contd…)

Handling custom resources:

```
public class MyResource implements AutoCloseable {

@Override
public void close(){
        System.out.println("Resource closed.");
}
```

Resource Management

```
try(MyResource mrr = new MyResource();){
        System.out.println(mrr.getMessage());
}
```

- The interface 'AutoCloseable' imposes implementation of close() method.

- The Files, Stream and connection classes implement similar Closeable interface and the proper close() method. In Java 7, Closeable interface extends AutoCloseable.
- The new getSuppressed() method within Exception class.
- The new constructor in Throwable class to enable/disable suppression.
- Order of closing is latest first.

Language Enhancements (Contd…)

Improved Exception Handling: One Catch block for multiple exceptions.

```
try {
      cn.checkExceptions("Exception2");

} catch (CustException1 |CustException2 e) {
      e.printStackTrace();
} catch (CustException3 e) {
      e.printStackTrace();
}
```

• Catch blocks for different exceptions but with same implementation may be combined together.

• Catch blocks with different implementations can be written separately.

• Using operator ‘|’, numerous exceptions can be joined this way.

Old type of catch

```
NewCatch cn = new NewCatch();

try {
      cn.checkExceptions("Exception3")
      ;
} catch (CustException1 e) {
      e.printStackTrace();
} catch (CustException2 e) {
      e.printStackTrace();
} catch (CustException3 e) {
      e.printStackTrace();
}
```

Module 7. Serialization

- **Overview**
  - What is Serialization
  - Object Serialization
  - Serialization on Collection
  - Serialization in Inheritance

Just Minute…

Object Serialization

- Converting an object’s representation into stream of bytes.
- To serialize a class, it must implement java.io.Serializable interface.
- Reading the state from a serialized object into the memory is known as object de-serialization

ObjectOutputStream, ObjectInputStream

- java.io.ObjectOutputStream
  - This class has functionality to serialize the object.
- java.io.ObjectInputStream
  - This class has functionality to de-serialize the object.

Serializing an Object

```
import java.io.*;

class Employee implements Serializable {
        int eno;
        String ename;
        int sal;

        public Employee (int eno, String ename, int sal) {
          this.eno=eno;
          this.ename=ename;
          this.sal=sal;
        }
        public void show () {
          System.out.println ("Emp number   : " + eno);
          System.out.println ("Emp name      : " + ename);
            System.out.println ("Emp sal          : " + sal);
        }
}
```

Serialization.java

```
class Serialization {
        public static void main (String [ ] args) {
          Employee e = new Employee(101,"jack",10000);
          try {
                  FileOutputStream fout=new FileOutputStream ("employee.ser");
                  ObjectOutputStream oos=new ObjectOutputStream (fout);
                  System.out.println ("Trying to serialize object....");
                  oos.writeObject (e);
                  System.out.println ("Object serialized...");
                  oos.close ();
                  fout.close ();
          }
          catch (IOException ioe) {
                  System.out.println ("Error: "+e);
          }
        }
}
```

DeSerialization.java

```
public class DeSerialization {
    public static void main (String [ ] args) {
        try {
            FileInputStream fin = new FileInputStream ("employee.ser");
            ObjectInputStream ois = new ObjectInputStream (fin);
            System.out.println ("De-serializing object....");
            Employee x = (Employee) ois.readObject ();
            System.out.println ("Object de-serialized...");
            System.out.println ("Printing values\n");
            x.show ();
        }catch (IOException ioe) {
            System.out.println ("Error is : " + ioe);
        }catch (ClassNotFoundException e) {
             System.out.println ("Class does not exist : " + e);
        }
    }
}
```

serialVersionUID

```
class employee implements serializable{
    static final long serialVersionUID=-7964162467535886628L;

}
```

SerializingCollectionOfObjects

```
public class SerialMultipleObjects {
	public static void main (String [ ] args)  {
		int eno = 1001, eno1 = 1002;
		String ename = "Ramesh", ename1 = "Umesh";
		int sal = 10000,sal1 = 15000;
		Employee eobj1 = new Employee (eno,ename,sal);
		Employee eobj2 = new Employee (eno1,ename1,sal1);
		try {
			FileOutputStream fout=new FileOutputStream ("emp.ser");
			ObjectOutputStream oos=new ObjectOutputStream (fout);
			System.out.println ("Serializing object....");
			HashSet hs = new HashSet();
			hs.add (eobj1);
			hs.add (eobj2);
			oos.writeObject (hs);
			System.out.println ("Objects serialized...");
			oos.close ();
			fout.close ();
		}catch (Exception e)  { System.out.println ("Error is : "+e);     }
	}
}
```

DeSerialMultipleObjects.java

```
public class  DeSerialMultObj {
  public static void main (String [ ] args) {
    try {
      FileInputStream fin = new FileInputStream ("emp.ser");
      ObjectInputStream ois = new ObjectInputStream (fin);
      System.out.println ("De-serializing objects....");
      Vector v = (Vector) ois.readObject ();
      Enumeration e = v.elements ();
      System.out.println ("Object de-serialized...");
      while (e.hasMoreElements ()) {
        Emp x = (Emp) e.nextElement ();
        x.show ();
      }
    } catch (IOException e) { System.out.println (e);}
      catch (ClassNotFoundException e) { System.out.println (e); }
  }
}
```

Serialization in Inheritance

```
import java.io.*;
class BankAccount{
        private double balance;
        public double getBalance(){
          return balance;
        }
        protected void setBalance(double balance){
          this.balance=balance;
        }
}
class SavingAccount extends BankAccount implements Serializable {
        private boolean isSalaryAcc;

        public boolean isSalaryAcc(){
          return isSalaryAcc;
        }
```

Serialization in Inheritance (contd…)

```
public void setSalaried(boolean isSalaryAcc){
        this.isSalaryAcc=isSalaryAcc;
}

private void readObject(ObjectInputStream is) throws IOException {
        setBalance(is.readDouble());
        this.isSalaryAcc=is.readBoolean();
}

private void writeObject(ObjectOutputStream os) throws IOException{
        os.writeDouble(getBalance());
        os.writeBoolean(isSalaryAcc);
}
}
```

Serialization in Inheritance (contd…)

```
import java.io.*;
class BankDemo1 {
        public static void main(String [ ] args) {
          SavingAccount sa1=new SavingAccount();
          try {
                 FileOutputStream fos=new FileOutputStream("bank.ser");
                 ObjectOutputStream oos=new ObjectOutputStream(fos);
                 sa1.setBalance(1000);
                 sa1.setSalaried(true);
                 oos.writeObject(sa1);
                 fos.close();
          } catch (FileNotFoundException e) {
                 e.printStackTrace();
          }catch(IOException e){
                 e.printStackTrace();
          }
        }
}
```

Serialization in Inheritance (contd…)

```
class BankDemo2 {
        public static void main(String[ ] args) {
          try {
                FileInputStream fis=new FileInputStream("bank.ser");
                ObjectInputStream ois=new ObjectInputStream(fis);
                SavingAccount sa1=(SavingAccount)ois.readObject();
                System.out.println("Balance :"+sa1.getBalance());
                System.out.println("isSalaryAcc"+sa1.isSalaryAcc());
          }catch (FileNotFoundException e) {
                e.printStackTrace();
          }catch(ClassNotFoundException e){
                e.printStackTrace();
          }catch (IOException e) {
                e.printStackTrace();
          }
        }
}
```

Just Minute…

Java NIO

- NIO was created to allow Java programmers to implement high-speed I/O without having to write custom native code.
- NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.

NIO Features

- Channel and Buffers
- File locking
- Memory Mapped Files
- Scatter and Gather
- Channel to Channel Transfer
- Non-Blocking Sockets

Buffers

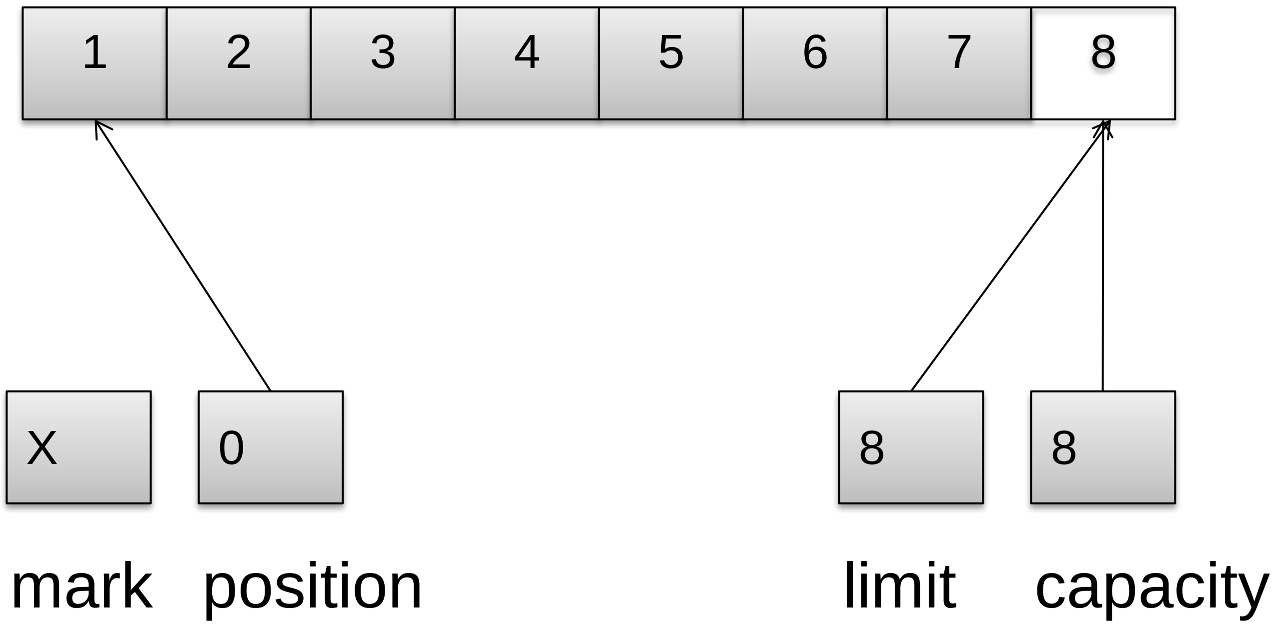
A Buffer is an object, which holds some data, that is to be written to or that has just been read from.

The addition of the Buffer object in NIO marks one of the most significant differences between the new library and original I/O.

Buffer Properties

Properties	Information
Position	The index of the next element to be read or written.
Capacity	The maximum number of elements a Buffer can contain.
Limit:	Index of the first element that should not be read or written (anything beyond the limit is a forbidden access)
Mark	Marks stores the current buffer position when you call mark()

Buffer Properties



Buffer Types

- ByteBuffer
- CharBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FoatBuffer
- DoubleBuffer

Buffer Basic Operations

Creating
Filling and Draining
Flipping and Rewinding
Marketing
Comparing
Duplicating

Creating Buffers

```
public static XXXBuffer allocate(int capacity)
public static XXXBuffer wrap(XXX[] array)
public static XXXBuffer wrap(XXX[] array,    int offset,   int length)

CharBuffer charBuffer = CharBuffer.allocate( 10 );
char[] charArray = new char[10];
CharBuffer charBuffer = CharBuffer.wrap( charArray );
CharBuffer charBufre = CharBuffer.wrap( charArray, 3, 6 );
```

Filling and Draining Buffers

```
XXXBufferput (XXXb );
XXXBufferput( intindex, XXXb );
XXXget( );
XXXget( intindex );

XXXBufferput( XXX[]   src );
XXXBufferput( XXX[]   src, int offset, int length );
XXXBufferget( XXX[]  dest );
XXXBufferget( XXX[]  dest, int  offset,  int  length) ;
```

Filling and Draining Buffers Cont…

```
FileInputStream fis = new FileInputStream("c:\\somedata.txt");
FileChannel fileChannel = fis.getChannel();
ByteBuffer byteBuffer = ByteBuffer.allocate(10);

    //filling
    for(int counter = 0 , counter < 10 ; counter++)
    {
      byteBuffer.put( ( byte ) counter );
    }

    byteBuffer.position(0);

    //Draining
    byteBuffer.get(4);
```

Flipping and Rewinding Buffers

Manually Flipping a buffer

```
buffer.limit(buffer.position()).position(0);
```

Flip and Rewind methods Provided byAPI.

```
Buffer flip();
Buffer rewind();
```

Flipping and Rewinding Buffers Cont…

```
FileInputStream fis = new FileInputStream(“c:\\someFile”);
FileChannel fChannel = fis.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(10);

for(int counter = 0; counter < 10; counter++ )
{
    buffer.put( ( byte) counter );
}

//flipping the buffer
buffer.flip();;
//buffer is ready to be written to the file
```

Marking Buffers

Buffer mark();
Buffer reset();

Comparing Buffers

```
boolean equals(Object obj);
int compareInt(Object obj);
```

Duplicating

CharBuffer duplicate();
CharBuffer asReadOnlyBuffer();
CharBuffer slice();

Demo

CharBuffer duplicate();
CharBuffer asReadOnlyBuffer();
CharBuffer slice();

Direct Buffer

ByteBuffer allocateDirect (int capacity);
boolean isDirect ( ) ;

A byte buffer is either direct or non-direct. *Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations.*

Channel

A Channel is an object from which you can read data and to which you can write data.

Comparing NIO with original I/O, a channel is like a stream.

Channels differ from streams in that they are bi-directional.

Channel Basic

Creating
Reading / Writing
Scattering / Gathering
Closing

Creating Channel

```
SocketChannel sc = SocketChannel.open();
ServerSocketChannel ssc = ServerSocketChannel.open();
DatagramChannel dc = datagramChannel.open();
RandomAccessFile raf = new RandomAccessFile("filepath", "rw");
    FileChannel fc = raf.getChannel();
```

Reading / Writing

- ReadableByteChannel
  - .read (ByteBuffer dst)
- WritableByteChannel
  - .write(ByteBuffer src)

Eg.

```
FileInputStream fis = new FileInputStream( fileName );
FileChannel channel = fis..getChannel();
channel.reaed ( buffer ):
```

Reading / Writing (Copy Demo)

```
FileChannel fChannel1 = new FileInputStream("C:\\Source.txt").getChannel();
FileChannel fChannel2 = new FileOutputStream("C:\\Destination.txt") .getChannel();

//allocates an underlying array of the specified size and wraps it in a buffer object
ByteBuffer buffer = ByteBuffer.allocate(1024);

//sets limit to match capacity sets position to 0
buffer.clear();

while(true) {
int read = fChannel1.read(buffer);
if(read == -1)
break;
}
```

Reading / Writing Demo

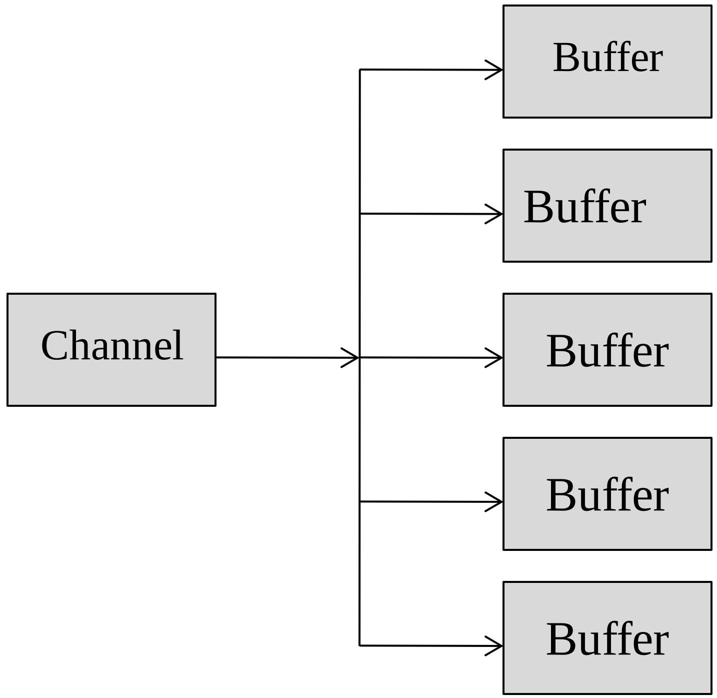
```
buffer.flip();
//sets limit to position and then sets position to 0

while(true)
{
if(buffer.remaining() == 0)
break;
fChannel2.write(buffer);
}
// buffer.remaining() get the difference between position and limit

fChannel1.close();
fChannel2.close();
```

Scatter / Gather

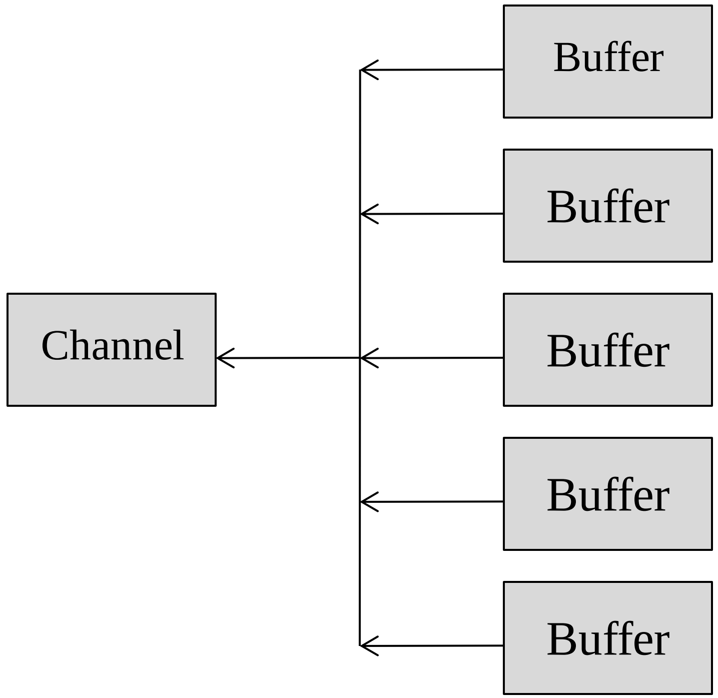
ScatteringByteChannel
    read (ByteBuffer dst);
    read (ByteBuffer dst, int offset, int length);

GatheringByteChannel
    write (ByteBuffer srcs)
    write (ByteBuffer srcs, int offset, int length);

Scattering Read



Gathering write



Scatter / Gather

Scattering read:

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { header, body };
//read  data into buffers
channel.read(buffers);
```

Gathering Write :

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);
//write data into buffers
ByteBuffer[] bufferArray = { header, body };
channel.write(buffers);
```

Demo /UseScatterGather.java

Closing

Channel.

```
boolean isOpen();
void close();
```

```
FileInputStream fis = new FileInputStream( fileName );
FileChannel fChannel = fis.getChannel();
fChannel.read( buffer ) ;

//closing the file channel

if(fChannel.isOpen())
    fChannel.close();
```

File Locking

```
FileLock lock();
FileLock lock( long position, long size, boolean shared );

FileLock tryLock();
FileLock tryLock( long position, long size, boolean shared );
```

File Locking

```
RandomAccessFile raf = new RandomAccessFile( "usefilelocks.txt", "rw" );
FileChannel fc = raf.getChannel();
FileLock lock = fc.lock( start, end, false );
```

Demo
/FileLockDemo.java

Memory-Mapped Files

The NIO framework provides the facility to **map** sections of a file to a ByteBuffer.

Once the mapping is set up, reading from the buffer automatically reads data from the corresponding section of the file, and writing to the buffer similarly results in the contents of the file being updated accordingly.

Memory-Mapped Files

```
RandomAccessFile randomAccessFile =
        new RandomAccessFile("C:\\CountryNames.doc", "rw");
FileChannel fileChannel = randomAccessFile.getChannel();

//acquring MappedByteBuffer from file channel
MappedByteBuffer mappedByteBuffer =
      fileChannel.map(FileChannel.MapMode.READ_WRITE, 0, 1024);

//replacing  the bytes at index
mappedByteBuffer.put(0,(byte)6); mappedByteBuffer.put(1023,(byte)99);

//closing channel and stream
fileChannel.close(); randomAccessFile.close();

/FileMappingDemo.java
```

Channel To Channel Transfer

- In Java NIO you can transfer data directly from one channel to another.
- The FileChannel class has a transferTo() and a transferFrom() method which does this for you.

TransferFrom()

```
RandomAccessFile fromFile = new RandomAccessFile("f romFile.txt ", " rw " );
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile    toFile  =  new RandomAccessFile("t oFile.txt" ,  " rw " );
FileChannel      toChannel  =  toFile.getChannel();

long position  =  0;
long count    = fromChannel.size();

toChannel.transferFrom( position ,  count , fromChannel );
```

TransferTo()

```
RandomAccessFile fromFile   =   new RandomAccessFile(  "fromFile.txt“  ,   "rw“  );
FileChannel  fromChannel   =   fromFile.getChannel();

RandomAccessFile toFile   =   new RandomAccessFile( "toFile.txt“ ,  "rw“ );
FileChannel  toChannel   =   toFile.getChannel();

long position = 0;
long count  =  fromChannel.size();

fromChannel. transferTo(position ,  count ,  toChannel );
```

Just Minute…

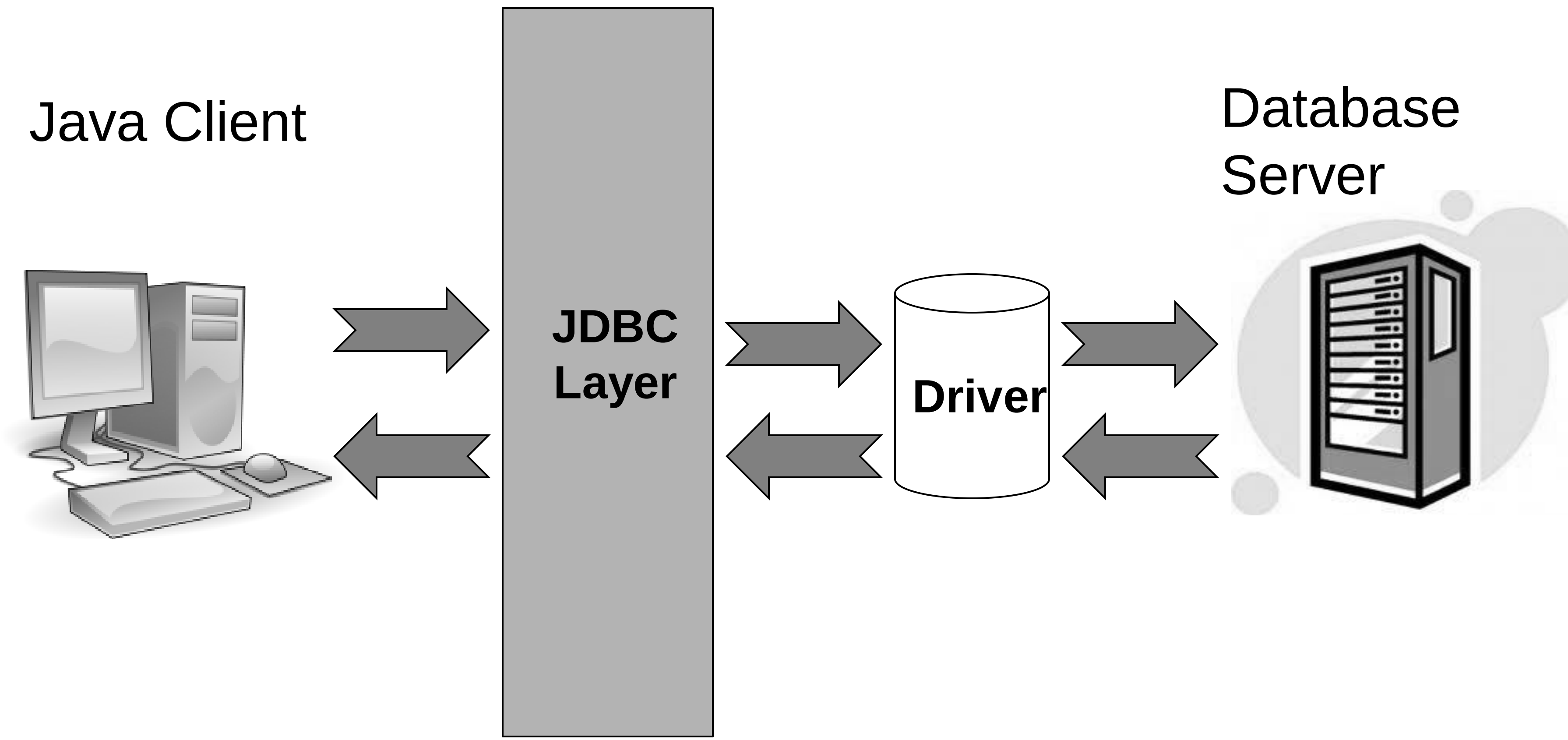
Moduel 16 JDBC :Java Database Connection

- Overview
  - Understanding the JDBC basics
  - Connecting to a database
  - Database operations such as insert, update, delete
  - Prepared statements
  - Batch updates and Transaction management
  - Stored procedures

What does JDBC do?

- Establishes a connection with a data source
- Sends queries and update statements to the data source
- Processes the results
- Invokes stored procedure and functions

What does JDBC do?



Traditional Database Connectivity

- ODBC
  - Open Database Connectivity.
  - Developed Using C Programming Language .
  - An application can use ODBC to query data from a DBMS, regardless of the operating system or DBMS it uses.

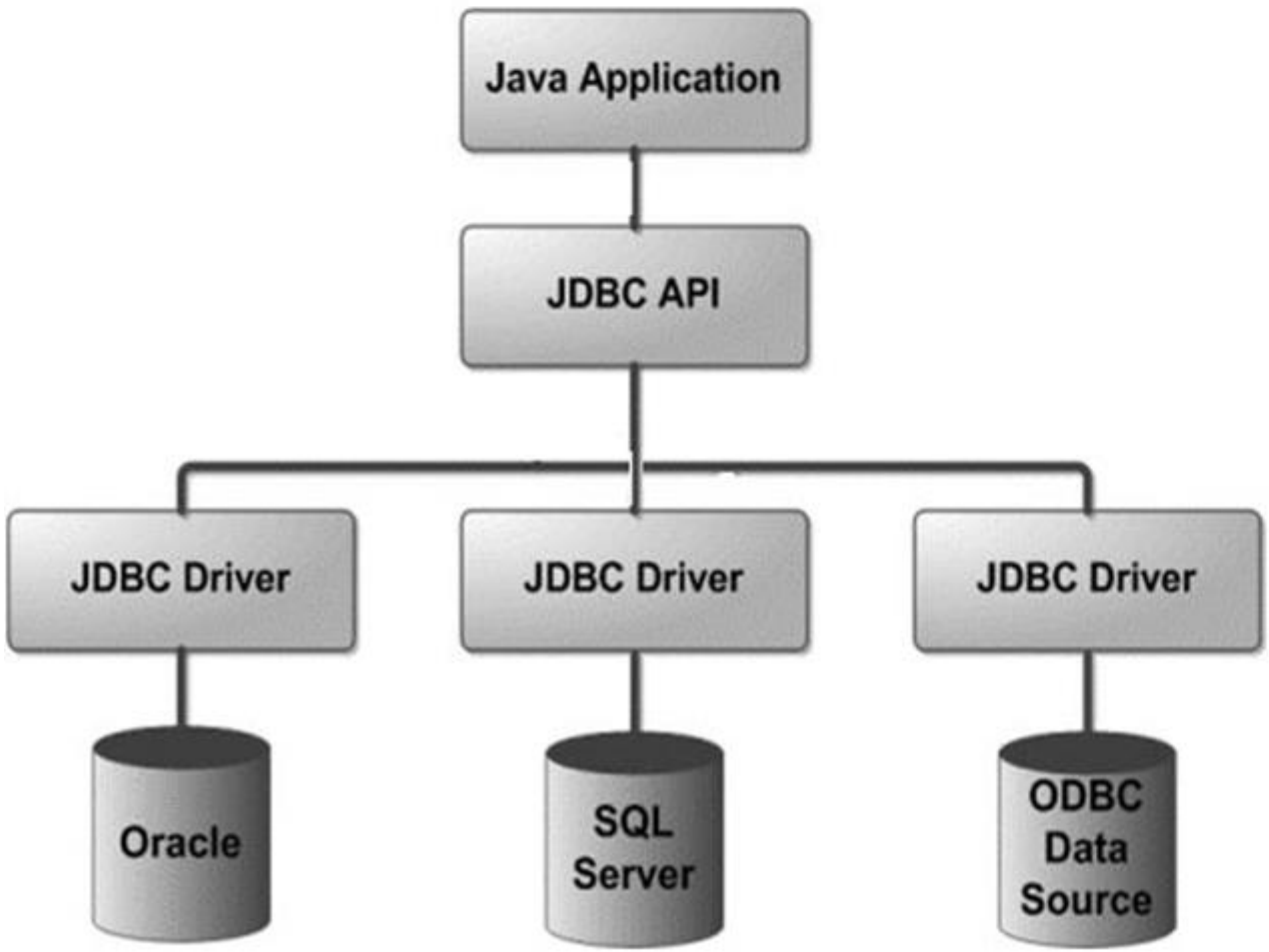
JDBC vs ODBC

- Why do you need the JDBC API over ODBC?

  - **ODBC**
    - ODBC drivers are written in C (Platform Dependant)
    - ODBC is complex thus difficult to learn
    - ODBC needs manual installation

  - **JDBC**
    - JDBC is a specification which encourages drivers written in java
    - Java being secured so is JDBC.
    - Java JDBC combination leads to platform independency so program becomes Write Once Run Anywhere (WORA)
    - Some drivers are automatically installable
    - Simplicity

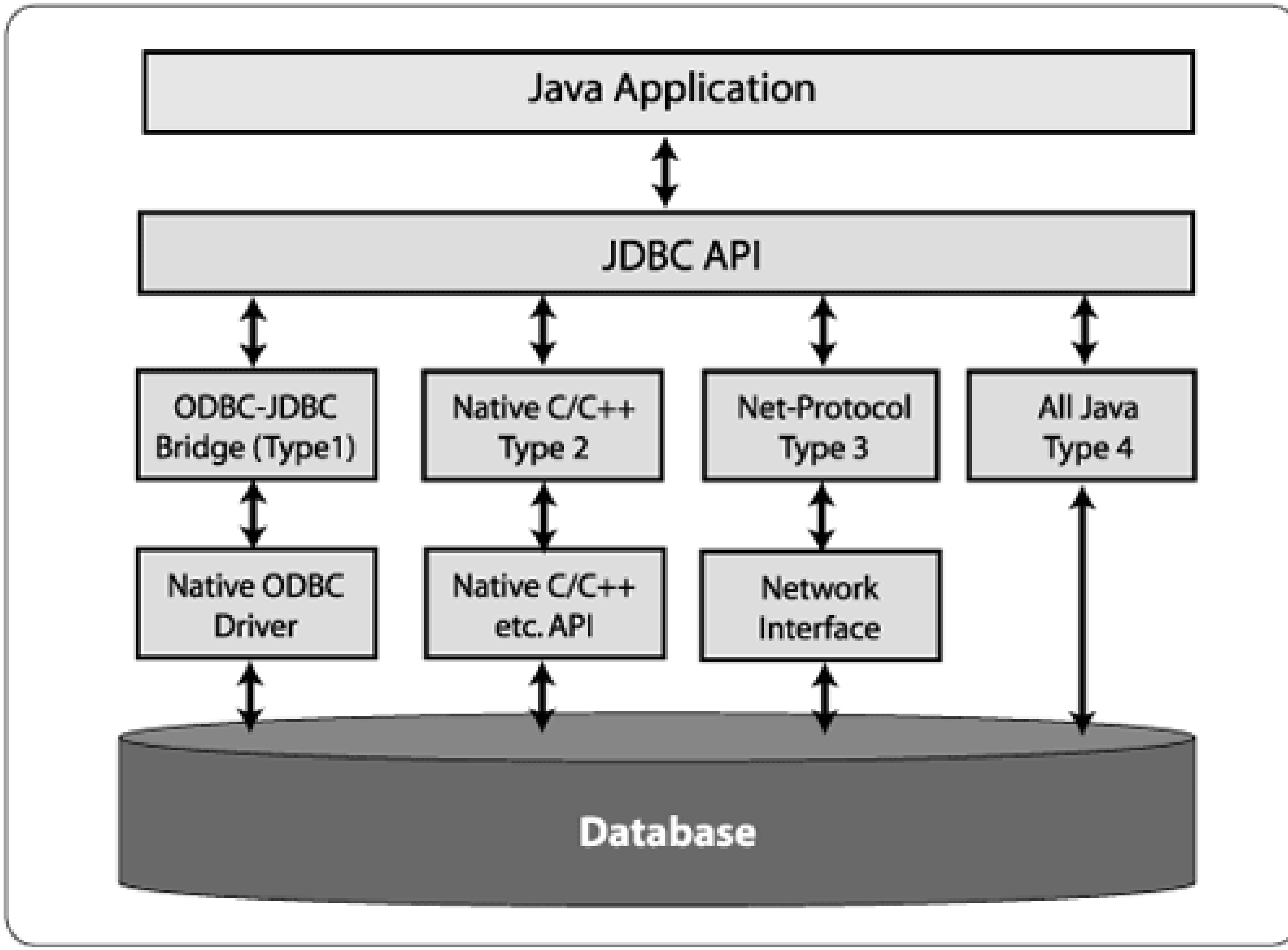
Drivers


Java Application
JDBC API
JDBC Driver
JDBC Driver
JDBC Driver
Oracle
SQL Server
ODBC Data Source

Driver Types

1) Type 1 - JDBC-ODBC Bridge Driver
2) Type 2 - Native-API Partly Java Driver
3) Type 3 - Net-Protocol All-Java Driver
4) Type 4 - Native Protocol All-Java Driver

Driver Types



Steps to Execute a Query in Database

1) Load the appropriate driver
2) Establish the connection
3) Create a statement
4) Execute the query and retrieve the result set
5) Process the result set
6) Close the result set, statement and connection

SimpleDemo.java

```
// 1. Loading Driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// 2. Establishing Connection
Connection connection =  DriverManager.getConnection("jdbc:odbc:JavaDSN", "scott", "tiger");
// 3. Creating Statement
Statement statement= connection.createStatement();
// 4. Creating resultset
ResultSet resultSet=statement.executeQuery("SELECT EMPNO, ENAME FROM EMP");
// 5. Traversing Resultset
while(resultSet.next()){ \\Processing Result
        System.out.print("Employee No :"+resultSet.getString("EMPNO"));
        System.out.println(" Employee Name :"+resultSet.getString("ENAME"));
}
// 6. Closing resources
resultSet.close(); statement.close();connection.close();
```

ConnectionManager.java

```
package packConnection;
import java.sql.* \\Connection,DriverManager,SQLException

public class ConnectionManager {
	private Connection conn = null;
	public  Connection getConnection() throws ClassNotFoundException,SQLException {
	  if (conn==null){
	  Class.forName("oracle.jdbc.OracleDriver"); //Optional from JDBC 4.0 //Type 4 Driver
	  conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", “tiger“);
	  System.out.println("Connection Done...");
	  }
	  return conn;
	}
	public  void closeConnection() throws SQLException {
	  if (conn != null) {
		System.out.println("connection closing is in progress");
		conn.close(); conn=null;
	  }
	}
}
```

TestConnectionManager.java

```
package packTest;

import java.sql.SQLException;
import packConnection.ConnectionManager;

public class TestConnectionManager {

	public static void main(String[] args) {
	  ConnectionManager connManager= new ConnectionManager();
	  try {
		connManager.getConnection();
	  }
	  catch (ClassNotFoundException e) {e.printStackTrace();}
	  catch (SQLException e) {e.printStackTrace();}
	}
}
```

Interface :EmpDAO.java

Employee DAO Business Methods

```
package packDAO;

import java.util.ArrayList;
import packPOJO.Employee;

public interface EmpDAO {

    public ArrayList<String> getEmployeeList();
    public Employee getEmployeeDetails(int employeeID)throws EmployeeNotExistsException;
    public int addNewEmployee(Employee employee);
    public int updateEmployeeSalary(double employeeSalary,int employeeNo);
    public int removeEmployee(int employeeID);
    public void getEmployeeTableStructure();
    public int updateManagerSalary(double percent);
}
```

DAOImpl.java

```
package packDAO;
import java.util.ArrayList;
import packConnection.ConnectionManager;
import packPOJO.Employee;
public class DAOImpl implements EmpDAO {
ConnectionManager connManager;
Connection conn = null;
    public DAOImpl() {
    try {
            connManager = new ConnectionManager();
            conn = connManager.getConnection(); \\Getting connection from ConnectionManager
      }                                         \\We are compromising on scalability
      catch (ClassNotFoundException e) {e.printStackTrace();}
      catch (SQLException e) {e.printStackTrace();}

    public ArrayList<String> getEmployeeList() {\\some code…….}
    public Employee getEmployeeDetails(int employeeID) {\\some code…….}
    public int addNewEmployee(Employee employee) { \\some code…….}
    public int updateEmployeeSalary(double employeeSalary) {\\some code…….}
    public int removeEmployee(int employeeID) {\\some code…….}
    public int updateManagerSalary(double percent) {\\some code…….}
}
```

getEmployeeList() using Statement

```
public ArrayList<String> getEmployeeList() {
  ArrayList<String> empList = null;
  Statement statement = null; ResultSet resultSet = null;
  try {
        statement = conn.createStatement(); //Creating Statement Object to hold Query
        resultSet = statement.executeQuery("SELECT * FROM EMP“);//Storing data into
resultSet
        empList = new ArrayList<String>();
        while (resultSet.next()) {
              empList.add(resultSet.getInt("empno") + ":"+ resultSet.getString("ename"));
        }
  }
  catch (SQLException e) {e.printStackTrace();}
  finally { try {
              resultSet.close();statement.close();conn.close();
              } catch (SQLException e) {e.printStackTrace();}
  } return empList;
}
```

Test code in main

```
EmpDAO empDAO= new DAOImpl();

 for(String emp:empDAO.getEmployeeList()){
                    System.out.println(emp);
 }
```

getEmployeeDetails () using PreparedStatement

```
public Employee getEmployeeDetails(int employeeID) throws EmployeeNotExistsException {
  try {
        String query="SELECT * FROM EMP WHERE EMPNO=?";
        PreparedStatement pStatement=conn.prepareStatement(query); //Parameterized Query
        pStatement.setInt(1, employeeID);
        ResultSet resultSet=pStatement.executeQuery();
        if(resultSet.next()){
                return new Employee(resultSet.getInt("empno"), resultSet.getString("ename"),
  resultSet.getDouble("sal"));
        }       //adding employee Details into Employee POJO
  }
  catch (SQLException e) {e.printStackTrace();}
  throw new EmployeeNotExistsException(); //Custom exception thrown
}
```

Test code in main

```
Employee emp= empDAO.getEmployeeDetails(7788);
System.out.println("Employee Name is :-"+emp.getEmployeeName());
```

PreparedStatement

- The advantages of Prepared Statements :
  - As the execution plan get cached, performance will be better.
  - It is a good way to code against SQL Injection as escapes the input values.
  - When it comes to a Statement with no unbound variables, the database is free to optimize to its full extent. The individual query will be faster, but the down side is that you need to do the database compilation all the time, and this is worse than the benefit of the faster query.
  - Other than training purpose it is better to use PreparedStatement to get full benefits and close all loopholes

addNewEmployee() using POJO

```
public int addNewEmployee(Employee employee) {
    int row=0;
    try {
        String query="INSERT INTO EMP (EMPNO, ENAME, JOB, SAL, DEPTNO) VALUES(?,?,?,?,?)";
        PreparedStatement pStatement= conn.prepareStatement(query);
        pStatement.setInt(1, employee.getEmployeeID());\\ Setting empid
        pStatement.setString(2, employee.getEmployeeName()); \\ employee name from POJO
        pStatement.setString(3, employee.getJobDescription()); \\ employee Job from POJO
        pStatement.setDouble(4, employee.getSalary()); \\ employee Salary from POJO
        pStatement.setDouble(5, employee.getDepartmentNo()); \\employee department name from
  POJO
        row=pStatement.executeUpdate(); \\ will insert employee data in emp table
    } catch (SQLException e) {e.printStackTrace();}
    return row;
  }
```

Test code in main

```
EmpDAO empDAO = new DAOImpl();
    System.out.println(empDAO.addNewEmployee(new Employee(3001, "Sahil", "Trainer", 98980, 10)));
```

Update & Delete

```
public int updateEmployeeSalary(double employeeSalary, int employeeNo) {
     …..//some code
     String query="UPDATE EMP SET SAL=SAL+? WHERE EMPNO=?";
     PreparedStatement pStatement= conn.prepareStatement(query);
     pStatement.setDouble(1, employeeSalary);
     pStatement.setInt(2, employeeNo);
     row=pStatement.executeUpdate();
     …..//some code
}
```

```
public int removeEmployee(int employeeID) {
      …..//some code
     String query="DELETE FROM EMP WHERE EMPNO=?";
     PreparedStatement pStatement= conn.prepareStatement(query);
     pStatement.setInt(1, employeeID);
     row=pStatement.executeUpdate();
      …..//some code
}
```

Complex SQL Statement Execution

- **Scenario :** Update All the Manager's Salary by 10%

- **Query in Database :**

```
UPDATE EMP
SET SAL=SAL+SAL*.10 WHERE EMPNO IN
(SELECT DISTINCT MANAGER.EMPNO
FROM EMP EMPLOYEE JOIN EMP MANAGER
ON EMPLOYEE.MGR=MANAGER.EMPNO)
```

updateManagerSalary() with CallableStatement

```
public int  updateManagerSalary(double percent){
    int row=0;
    try {
      String query=" { call updateManagerSal (?) }"; \\ Store Procedure
      CallableStatement cStatement=conn.prepareCall(query); \\ Preparing call
      cStatement.setDouble(1, percent);
      row=cStatement.executeUpdate();
    } catch (SQLException e) { e.printStackTrace(); }
    return row;
}
```

```
CREATE OR REPLACE PROCEDURE UPDATEMANAGERSAL(PERCENT NUMBER) IS
BEGIN
      UPDATE EMP SET SAL=SAL+ (SAL* PERCENT) WHERE EMPNO IN
      (SELECT DISTINCT MANAGER.EMPNO
      FROM EMP EMPLOYEE JOIN EMP MANAGER
      ON EMPLOYEE.MGR=MANAGER.EMPNO);
END;
```

ResutSetMetaData

```
public void getEmployeeTableStructure() {
	try{
		Statement st=conn.createStatement();
		ResultSet rs=st.executeQuery("select * from emp");
		ResultSetMetaData rsmd=rs.getMetaData();
		System.out.println("number of columns :  "+rsmd.getColumnCount());
		System.out.print(rsmd.getColumnLabel(1));
		System.out.println("-->"+rsmd.getColumnTypeName(1));
		System.out.print(rsmd.getColumnLabel(2));
		System.out.println("-->"+rsmd.getColumnTypeName(2));
	}
	catch(SQLException sqe){
		sqe.printStackTrace();
	}
}
```

DatabaseMetaData

```
public void databaseMetaDataDemo() throws Exception{

   DatabaseMetaData dmd=con.getMetaData();

   System.out.println(dmd.getDriverName());
   System.out.println(dmd.getMaxColumnNameLength());
    System.out.println(dmd.getUserName());
    System.out.println(dmd.getDatabaseProductName());
    System.out.println(dmd.getMaxColumnsInTable());
    System.out.println(dmd.getMaxRowSize());
    System.out.println(dmd.getSchemas);
    System.out.println(dmd.supportsStoredProcedures());
}
```

JDBC: ResultSet

- A ResultSet consists of records. Each records contains a set of columns. Each record contains the same amount of columns, although not all columns may have a value. A column can have a null value. Here is an illustration of a ResultSet:

Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

ResultSet Type, Concurrency & Holdability

- When you create a ResultSet there are three attributes you can set. These are:
  - Type
    - TYPE_SCROLL_SENSITIVE
    - TYPE_SCROLL_INSENSITIVE
    - TYPE_FORWARD_ONLY
  - Concurrency
    - CONCUR_UPDATABLE
    - CONCUR_READ_ONLY

ResultSet methods for traversing

Methods	Description
boolean previous()	Moves cursor to next row relative to its current position
boolean previous()	Moves cursor to previous row relative to its current position
boolean first()	Moves cursor to first row
boolean last()	Moves cursor to last row
boolean absolute(int)	Moves cursor to given row number

ResultSet methods for traversing(contd…)

Methods	Description
void beforeFirst()	Moves cursor before first row
void afterLast()	Moves cursor just after the last row
boolean relative(int)	Moves cursor by given relative value from current position

Inserting Data

```
public void addNewDept (String deptName, String location) {
    int next = getMaxDept() + 10; //will return max Deptno
    try {
      Statement statement= connection.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
      ResultSet resultSet = statement.executeQuery("SELECT DEPTNO, DNAME, LOC FROM
    DEPT");
      resultSet.moveToInsertRow(); // Open Resultset for insert mode
      resultSet.updateInt(1, next);
      resultSet.updateString(2, deptName);
      resultSet.updateString(3, location);
      resultSet.insertRow(); // insert data from resultset to actual data source
      resultSet.close();
    }
    catch (SQLException e) { e.printStackTrace(); }
}
```

Updating Data

```
public void updateDeptName(int deptNo, String deptName) {
      try {
        PreparedStatement preparedStatement=connection.prepareStatement(
                        "select dname from dept where deptno=?",
                        ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);
        preparedStatement.setInt(1, deptNo);
        ResultSet resultSet = preparedStatement.executeQuery();
        resultSet.next();
        resultSet.updateString("DNAME", deptName);
        resultSet.updateRow();
      }
      catch (SQLException e) {e.printStackTrace(); }
}
```

Transactions

Transaction is a set of statements that are executed as a single unit. A transaction is complete only when all the statements have executed sucessfully. If any one statement fails to execute sucessfully , then the whole transaction is rolled back.

Various methods used to carry out an Transaction are :

- void commit( )
- boolean getAutoCommit()
- void rollback()
- void setAutoCommit (boolean enableAutoCommit)

All the above methods throw SQLException

Example on Handling Transaction

After purchase of necessary raw material, now the data should be updated in the Stores table, DeptProduction table and DeptFinance table. If the data has been reflected succesfully to all the tables only then , update the transaction table, else rollback

```
try {
        conn.setAutoCommit(false);

        // code to add data to Stores;
        // code to add data to DeptProduction;
        // code to add data to DeptFinance;
        // code to update the transaction table;

        conn.commit( );
}
catch(Exception e){
        conn.rollback( );
}
```

Mapping SQL and Java Types

JDBC Types	Java Type
CHAR	String
VARCHAR	String
INTEGER	int
REAL	float
FLOAT	float
DOUBLE	double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

RowSet

- A RowSet is a disconnected, serializable version of a JDBC ResultSet.

Types and Uses of Rowsets

- A CachedRowSet class—a disconnected rowset that caches its data in memory; an ideal way to provide thin Java clients.

- A JDBCRowSet class—a connected rowset that serves mainly as a thin wrapper around a ResultSet object to make a JDBC driver look like a JavaBeans component

CachedRowSet and JdbcRowSet

```
RowSet rowSet = new CachedRowSetImpl();
//RowSet rowSet = new JdbcRowSetImpl();
rowSet.setUsername("scott");
rowSet.setPassword("tiger");
rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");

rowSet.setCommand("SELECT * FROM EMP");
rowSet.execute();
while (rowSet.next()) {
    System.out.println(rowSet.getString(""));
}
```

CachedRowSet

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con= DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",
    "scott", "tiger");
Statement st=con.createStatement();
st.setMaxRows(10);
ResultSet rs= st.executeQuery("select * from emp");
CachedRowSet cachedRowSet=new CachedRowSetImpl();
cachedRowSet.populate(rs);
rs.close();
st.close();
con.close();

while (cachedRowSet.next()) {
    System.out.println(cachedRowSet.getString(1));
}
```

Module 17 Threads and Concurrency

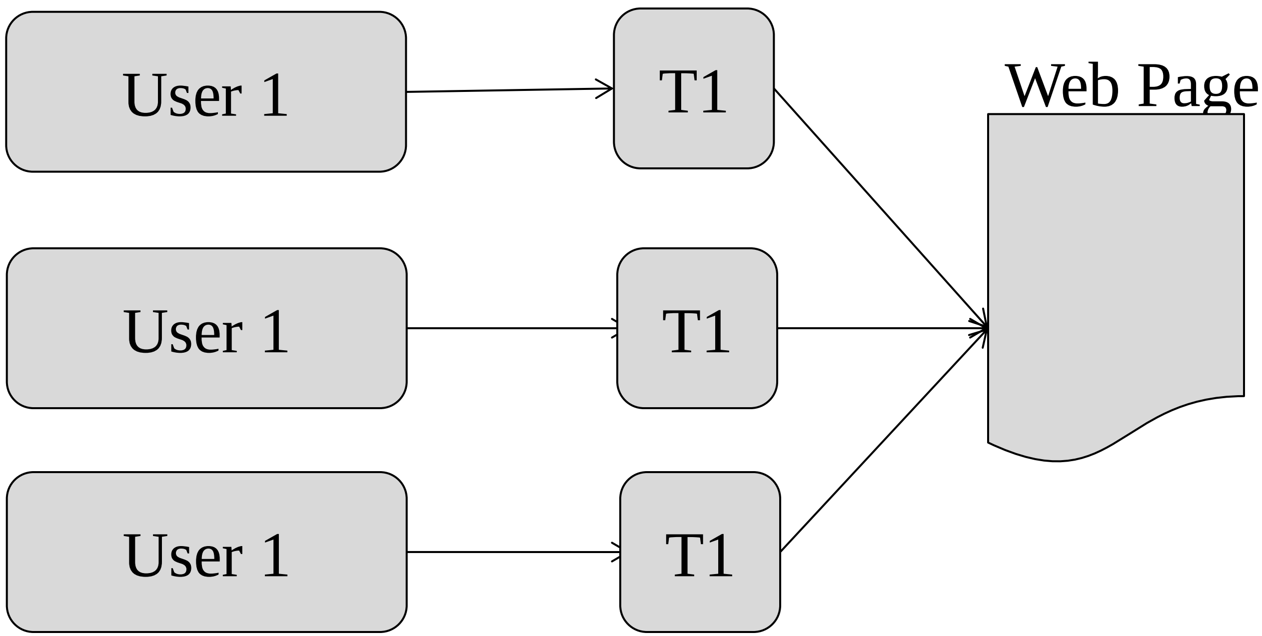
- Overview of Multithreading and Java Memory Model
- Creating Threads
- Thread states and life cycle
- The Executor framework
- Thread Scheduling : Priorities, Sleep, Joins, Latches, Barriers
- Race condition, Synchronization and Semaphores.
- Inter-thread communication and Exchangers
- Deadlock and solution
- Concurrent Collections and Atomic data types
- More on threads- Daemon, Group Fork and Join framework

Thread

- A light weight process which runs under resources of main process.
- Can harness power of parallel processing.
- Actual thread execution highly depends on OS and hardware support.
- The JVM of Thread-non-supportive OS takes care of thread execution.
- On single processor, threads may be executed in time sharing manner.

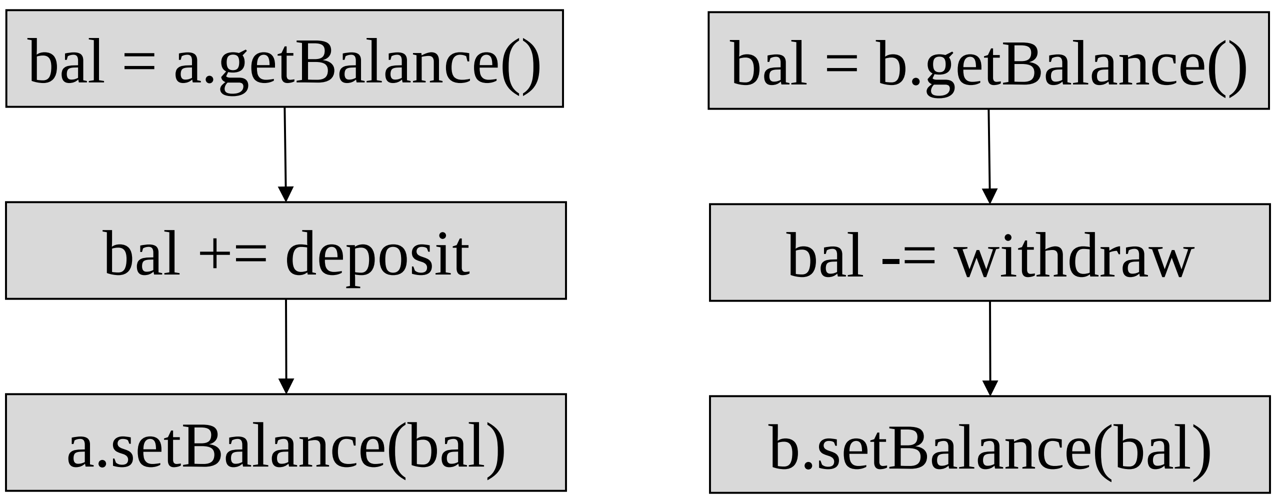
Multithreading

- A website server entertains requests in concurrent manner
- Eclipse doing compilation, execution, scrolling editing.
- Browser downloading different web page components.



Multithreading

- Execution of such steps by different tellers can be considered as multi-threading in a computer jargon.



Creating Threads

- Threads can be created either of these two ways.
  - Creating a **worker** object of the java.lang.Thread class
  - Creating the **task** object which is implementing the java.lang.Runnable interface.
  - Using Executor framework which **decouples Task submission from policy of worker threads**.

Extend Thread Class

```
public class CounterThread extends Thread {

    public void run(){
        for(int i=0; i<5; i++)
        System.out.println(getName()+": "+increAndGet());
    }
}

public static void main(String[] args) {
    CounterThread counter1 = new CounterThread();
    CounterThread counter2 = new CounterThread();
    counter1.start();
    counter2.start();
}
```

Implementing Runnable Interface

```
public class CounterRunnable implements Runnable {

public void run(){
        Thread thisThread = Thread.currentThread();
        String threadName = thisThread.getName();
        // Define task here.
}
public static void main(String[] args) {
        CounterRunnable counter = new CounterRunnable(); // Task
        Thread counter1 = new Thread(counter); // Worker 1
        Thread counter2 = new Thread(counter);          // Worker 2
        counter1.start();
        counter2.start();
}
```

Extending Vs Implementing

S. N.	Extending Thread	Implementing Runnable
1	Basically for creating worker thread.	Basically for defining task.
2	It itself is a Thread. Simple syntax	Thread object wraps Runnable object
3	Can not extend any other class	Can extend any other class
4	A functionality is executed only once on a thread instance.	A functionality can be executed more than once by multiple worker threads.
5	Concurrent framework does have limited support.	Concurrent framework provide extensive support.
6	Thread’s life cycle methods like interrupt() can be overridden.	Only run() method can be overridden.

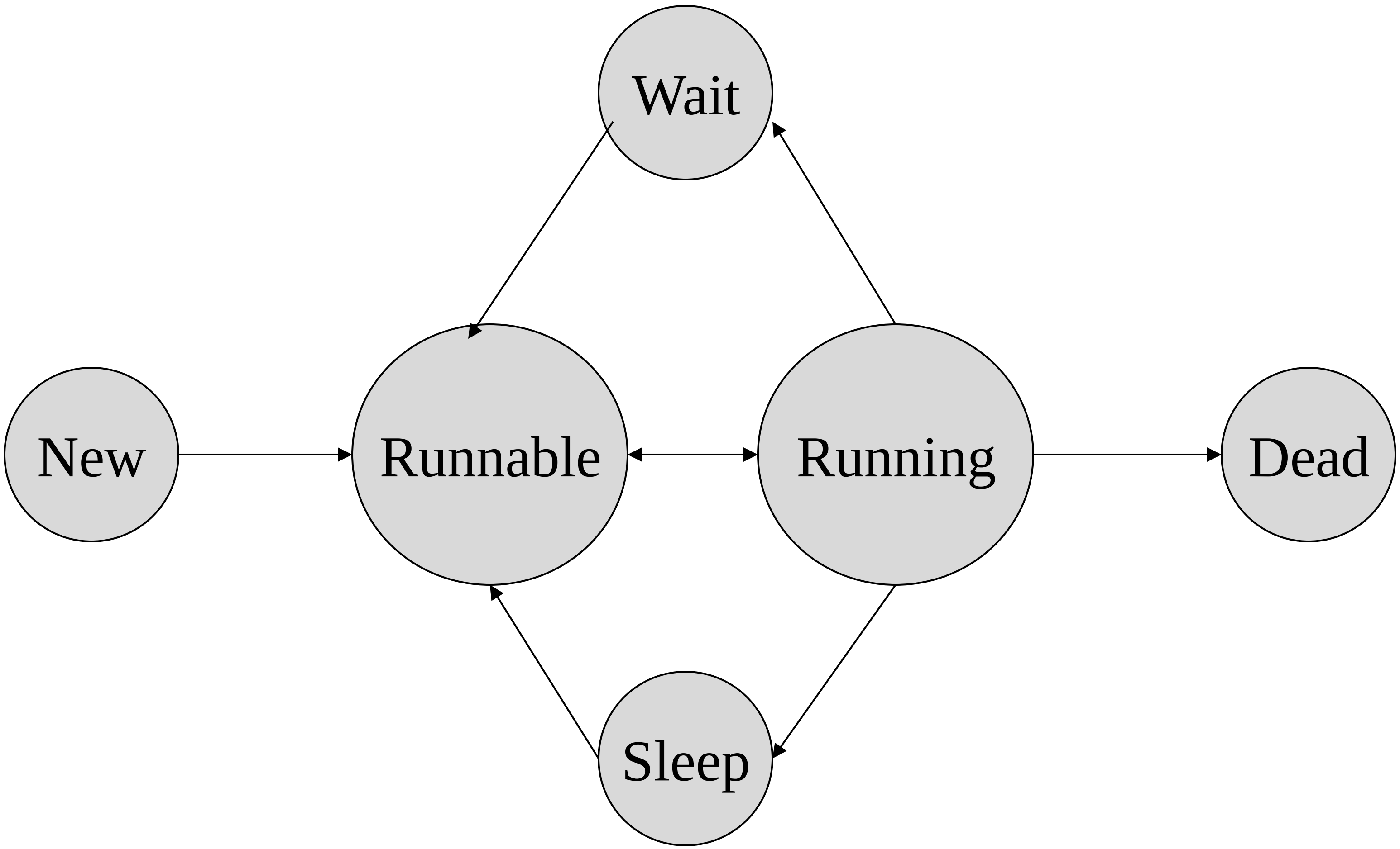
Constructors in Thread Class

- Thread()
- Thread(String name)
- Thread(Runnable target)
- Thread(Runable target , String name)

Few Methods from Java.lang.Thread Class

- public static Thread currentThread()
- public final void setName (String name)
- public final String getName ()
- public final boolean isAlive ()
- public long getId ()
- public Thread.State getState()

Thread States



The Executor framework

- Decouples task submission from mechanics of running each task.
- The framework standardizes…
  - ➢ Invocation
  - ➢ Scheduling
  - ➢ Execution
  - ➢ Controlling asynchronous tasks.

- Creating Executor object to define task running policy

```
public static ExecutorService getExecutorService(){
    return Executors.newSingleThreadExecutor();
}
```

Executors come from java.util.concurrent

Task submission

Task submission

```
// Class implementing Runnable
CounterRunnable counter = new CounterRunnable();

ExecutorService execService = ExecutorFactory.getExecutorService();

execService.execute(counter);

execService.shutdown();// Life cycle method.
```

Executor Service Implementations

Executors come from java.util.concurrent

S.N.	Implementation	Significance
01	SingleThreadExcutor	All tasks are executed by single thread one after another in the order of their submission.
02	FixedThreadPool	Provides a pool of fixed number of threads. Tasks are kept on hold until thread from pool is available.
03	CachedThreadPool	Provides a pool of varying size.

The Callable Interface

```
class StringTask implements Callable<MyPojo> {
	private MyPojo mp;
	public MyPojo call() throws Exception {
		// Do processing
		return mp;
	}
}
```

Runnable	Callable
Introduces void run() method which throws no exception	Introduces <T> call() method which throws Exception.
Thread can not return final result. It is to be extracted through post-mortem	Thread can return final result through Future object.
The run() can not throw any exception	The call() can throw custom exception
Task submission by Executor.execute()	Task submission by Executor.submit()

The submit()

```
ExecutorService pool = Executors.newFixedThreadPool(3);

try {
    for(int i=0; i<10; i++){
        FutureTask<MyPojo> ft = new FutureTask<MyPojo>(new
            StringTask(i));
        pool.submit(ft);
        MyPojo mp = ft.get();
        ------
    }
```

Thread Scheduling - Priorities

1. An integer value if higher for a thread defines higher priority for execution on underlying platform
   1. Window OS supports range 1 to 7
   2. Solaris OS supports range 1` to 231
   3. Java supports MIN_PRIORITY(1) TO MAX_PRIORITY(10).

Note: OS, JVM take liberty to fluctuate priority as per their need.

```
HeavyProcess hp = new HeavyProcess();  // A runnable object
Thread thrdHeavy = new Thread(hp, "HeavyThrd");
thrdHeavy.setPriority(Thread.MAX_PRIORITY);
```

Rules for deciding priorities

Priority Range	Scenario
10	Crisis Management
7-9	Interactive and Event driven
4-6	IO Bound
2-3	Background computation
1	Heavy, time consuming operations.

Thread Scheduling – sleep() method

public static sleep(long milliseconds) throws InterruptedException

1. **Code from Thread processing value**

```
public void run(){

try {
        Thread.sleep(timeOut);  // Interrupted when value is ready
        System.out.println("Time out");
    } catch (InterruptedException e) {
    System.out.println("Processing value:"+value.getValueToProcess());// Process
      value
    }
}
```

interrupt()

1. Code for Thread preparing a value for processing.

```
public void run() {
    // Do processing step 1
    randomSleep();

    // Do processing step 2
    //randomSleep();

    value.setValueToProcess(200); // Value is ready for final processing
    client.interrupt(); // Thread 1 being interrupted.
    }
```

Thread Scheduling – join() method

```
public void run() {
    Thread task1 = new Thread(new Task(), "Task1");
    Thread task2 = new Thread(new Task(), "Task2");

    task1.start();
    task2.start();
    try {
        // Do join
        task1.join();
        task2.join();
    } catch (InterruptedException e) {
    e.printStackTrace();
    }

    // To do task3
}
```

CountDown Latch

The CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

Methods	Purpose	Remark
CountDownLatch(int n)	A constructor initialized latch to the given value.	
await()	Makes current thread to wait until latch has counted down to zero.	Interrupt() may interrupt await().
countDown()	Decrement count of the latch.	Releases all awaiting thread when count reaches zero.
getCount()	Returns the current count.	

- May be used to start threads at same time.

CountDown Latch- Activity

```
// Thread awaiting for other threads to complete task.
public void run() {
    CountDownLatch countDown = new CountDownLatch(2);

    Thread task1 = new Thread(new Task(countDown), "Task1");
    Thread task2 = new Thread(new Task(countDown), "Task2");
    task1.start();
    task2.start();

    try {
        // Do join
        countDown.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    // To do task3
}
```

CountDown latch - Task

```
// The task performing threads…

public void run() {
    //Perform task
    //Perform task
    countDown.countDown(); // It will reduce count. If it becomes 0,
                           // resumes execution of awaiting thread.
     // Close resources this task has opened.
}
```

CountDown latch- Get-set-go

Starting more than one threads at same time…

```
class Process implements Runnable {
        private CountDownLatch startEvent;

        public Process(CountDownLatch startEvent){
            this.startEvent = startEvent;
        }

        public void run() {
                    startEvent.await();
                    // Process to start after green signal.
        }
}
```

CountDown latch- Get-set-go

```
CountDownLatch startEvent = new CountDownLatch(1);

Thread process1 = new Thread(new Process("P1", startEvent));
Thread process2 = new Thread(new Process("P2", startEvent));

process1.start();
process2.start();

// Do steps before all concurrent processes to start
startEvent.countDown();  // Green signal for processes.
```

Cyclic Barrier- Processing Threads

- A barrier for all threads wait at until all threads reach it.
- Example: Scheduled processing at Head office of the data pulled from branch office.

```
public class BranchOffice implements Runnable {
private CyclicBarrier barrier;

public void run() {
        Thread branchOffice = Thread.currentThread();
            // Processing before barrier runs.
            try {barrier.await();}  // Thread waiting for all other threads to reach here.
            // Processing after barrier runs
        catch (InterruptedException e)
            {e.printStackTrace();}
        catch (BrokenBarrierException e)
            {e.printStackTrace();}
      }
}
```

Cyclic Barrier-Barrier point

```
public class MonthlyProcessing implements Runnable {
        public void run() {
                // Barrier Code
        }
}

public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(2, new MonthlyProcessing());

        Thread finance = new Thread(new BranchOffice(barrier), "Kolkata");
        Thread admin = new Thread(new BranchOffice(barrier), "Benluru");

        finance.start();
        admin.start();
}
```

Cyclic Barrier

Methods	Purpose	Remark
CyclicBarrier(int n)	A constructor initialized barrier to number of threads to reach barrier.	If barrier event is not given, barrier is declared but event not fired.
CyclicBarrier(int n, Runnable action)	A constructor to set barrier event.	Event is fired only after all threads reach barrier.
await()	Demarcation of barrier within all threads.	Interrupt() may interrupt await().
await(Time out, Time unit)	After timeout, thread is treated as reached to barrier.	
reset()	Reset the barrier. Waiting threads reaches barrier.	

Countdown Latch Vs Cyclic Barrier

Count Down Latch	Cyclic Barrier
Does not take Runnable tasks.	Takes Runnable task.
Can not be reset	Can be reset
Number of calls to countDown() will result in all waiting threads to release. It can be done by single thread also.	Number of waiting threads which are calling await(). It has to be called not more than once by a thread.

Race Condition

- In single thread model, execution order of processing steps is predictable. Multithreaded model does not guarantee order because…
  - Thread's concurrent execution makes data inconsistent.
  - Compilers optimize code there by may change order of execution.
- Solution
  - Synchronized block, method.
  - Semaphore
  - Volatile variables.

Race Condition

Process 1

x=10, y=20

1. x = y
2. y = x + 10

x = 20, y = 30

Process 2

x=10, y=20

1. y = x
2. x = y + 10

x = 20, y = 10

Process 1&2

x=10, y=20

1. x = y
4. y = x + 10

2. y = x
3. x = y + 10

Process 1&2

x=40, y=30

Race Condition-Synchronization

Process 1

x=10, y=20

1. x = y
2. y = x + 10

x = 20, y = 30

Process 2

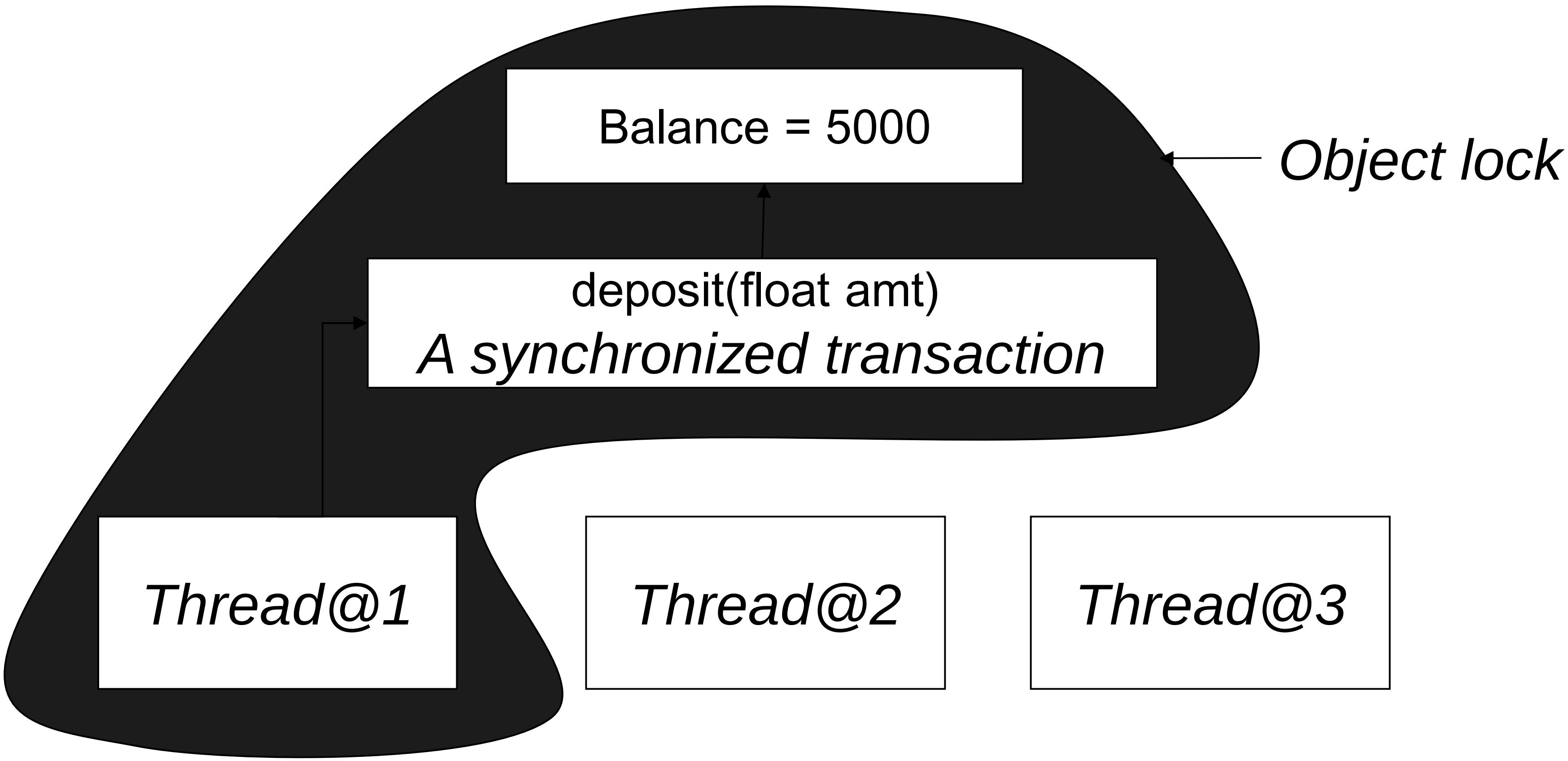
x=20, y=30

1. y = x
2. x = y + 10

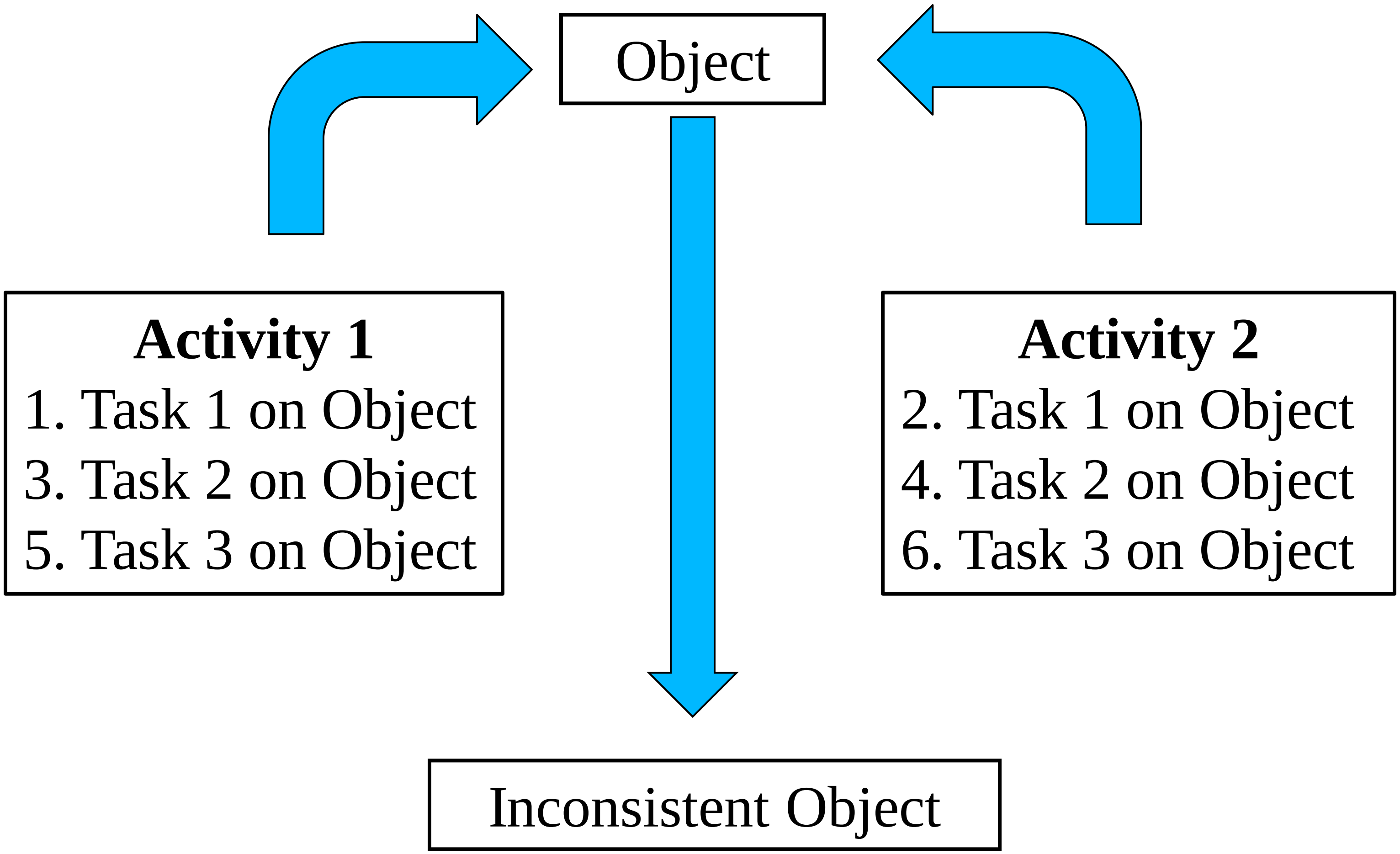
x = 30, y = 20

Atomic steps on object

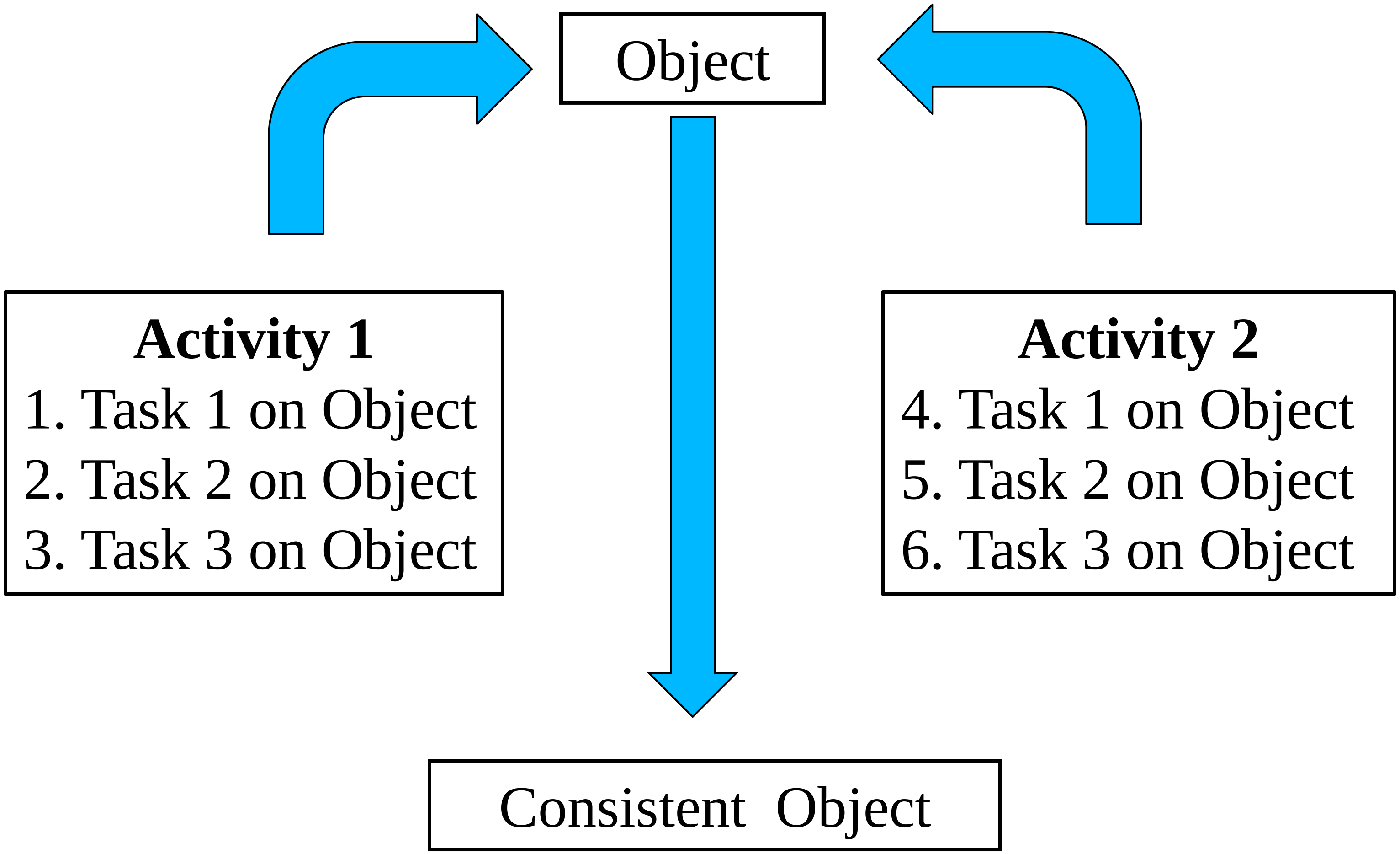
- Acquiring object locks between interleaved executions of multiple threads
- Changes made by one thread will be guaranteed to reflect to another thread.



Atomic steps



Atomic steps- Synchronization



Atomic Steps- Synchronized block

```
// Activity 1
synchronized(modelObject){
    modelObject.process1();

    modelObject.process2();

    modelObject.process3();
} // Block ends here
```

```
// Activity 2
synchronized(modelObject){
    modelObject.process1();

    modelObject.process2();

    modelObject.process3();
} // Block ends here
```

Java Memory Model

- Synchronization
- Locks
- Wait sets

1. Object posses monitor. Thread acquires it to do atomic operation.
2. Other threads go into wait set.
3. After thread completes atomic operation, releases monitor.
4. One of the waiting thread acquires it.
5. Thread starts its own atomic operation.

Atomic Steps- Semaphore

```
// Activity 1
semaphore.acquire();
    modelObject.process1();

    modelObject.process2();

    modelObject.process3();
semaphore.release();
```

```
// Activity 2
semaphore.acquire();
    modelObject.process1();

    modelObject.process2();

    modelObject.process3();
semaphore.release();
```

Atomic steps- Synchronized method

```
public synchronized void withdraw(int accNo, float amt){
        // Get account information
        PBankAcc bankAcc =  accDao.getAccInformation(accNo);

        // Withdrawal business logic
        bankAcc.setAccBal(bankAcc.getAccBal()-amt);

        // Update transaction
        transactionDao.setAccTransaction(bankAcc);
}
```

Thread1 Thread2 Thread3

1. Object posses monitor. Thread acquires it to do atomic operation.
2. Other threads go into wait set.
3. After thread completes atomic operation, releases monitor.
4. One of the waiting thread acquires it.
5. Thread starts its own atomic operation.

Synchronization- Block Vs Method

S.N.	Synchronized block	Synchronized method
1	Thread owes responsibility to block the object.	Object owes responsibility to decide which method to synchronize.
2	Thread can utilized objects in thread-safe/nonthread-safe flavors.	Once declared, thread clients of this method cannot have its un-synchronized version. All clients will start getting this method thread safe.
3	Deadlocks are easy to avoid	Method codes are invisible across layer. Lead to nested monitor problem and thus increases inefficiency and chances of deadlock.
4	Better granularity is possible by deciding which statements to include in synchronized block and which is not.	All statements of the method come under synchronization. Granularity not possible.

Semaphore

Maintains a set of permits. Thread on receipt of permit does atomic operation otherwise wait for a permit. On completion of operation, thread releases permit for any other waiting thread to acquire.

Services	Purpose	Remark
Semaphore(int permits)	Constructor takes number of permits.	Permit size can not be changed for Semaphore object.
Semaphore(int permits, boolean fair)	If fairness is false, Semaphore does not guarantee of order in which threads will acquire permits.	For first come first served policy, fairness should be true.
acquire(), Acquire(int permits)	Gets a permit for a thread. If not available, thread goes into wait queue.	Thread can be interrupted while waiting for acquire.
acquireUninterruptibly( ), acquireUn…(int per)	Acquires a permit from semaphore until made available.	Can not be interrupted.

Semaphore…

Services	Purpose	Remark
availablePermits()	Return current number of available permits.	
getQueuedThread(), getQueueLength()	To get a queue or number of queued items (waiting threads)	
release(), release(int permits)	Releases permit/s	A thread can release a permit despite of who acquired it.
tryAcquire(), tryAcquire(int permits),tryAcquire(int permits, int timeOut, timeUnit)	Acquires a permit only if its available at the time of invocation.	

Semaphore for limited resources

```
// Connection Resource…
public class ConnectionPool {

public Connection getConnection(){
        for(Connection connection:connections){
                if (connection.isConnectionAvailable()){
                        connection.openConnection();
                        return connection;
                }
        }
        return null;
        }
}
```

Semaphore for limited resources

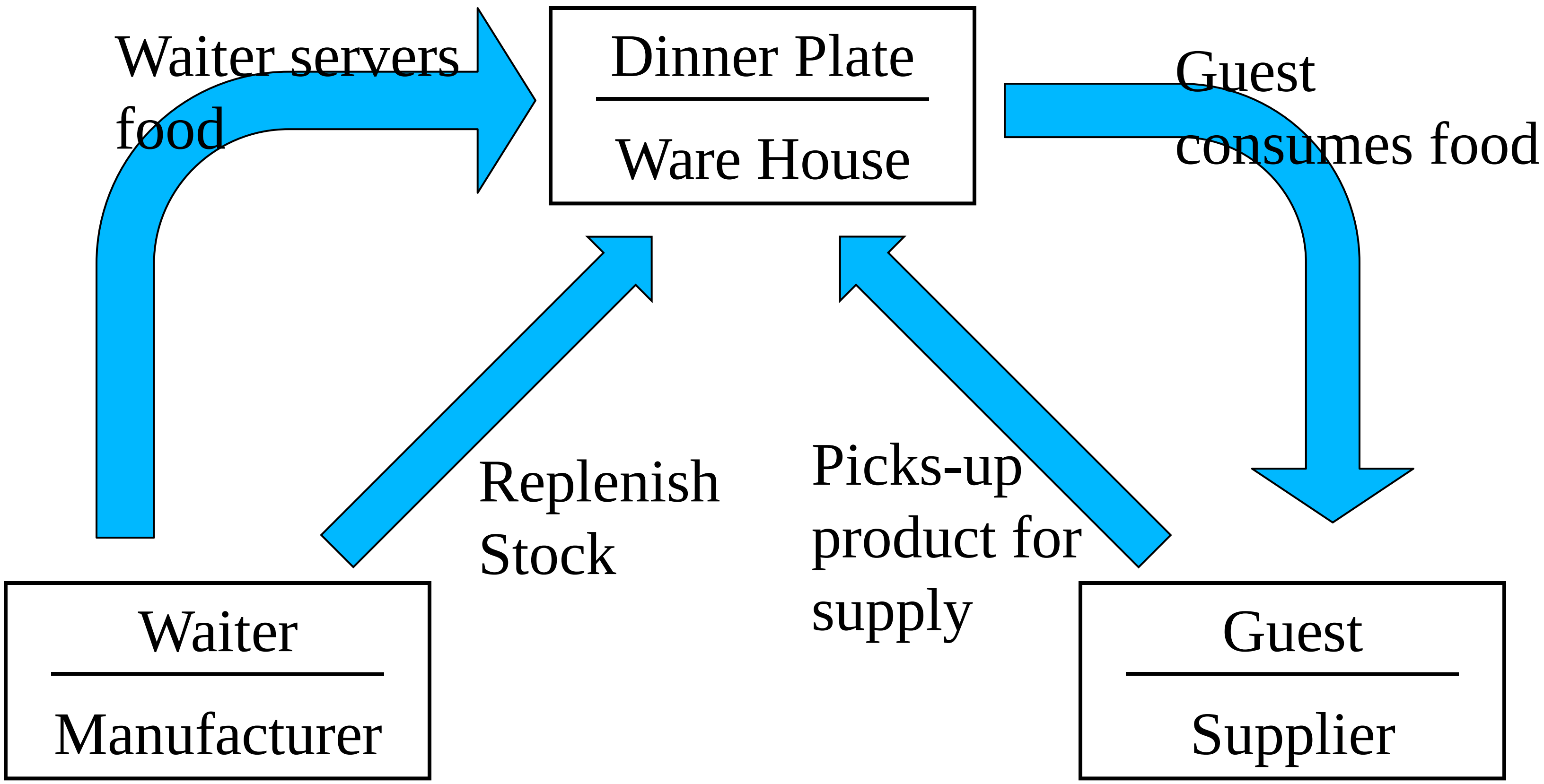
Process Thread

```
private ConnectionPool connections;
private Semaphore control;
public void run() {
    try{
        control.acquire();

        Connection connection = connections.getConnection();
        // Process on connection
        connection.closeConnection();

        control.release();
    } catch (InterruptedException e) { }
}
```

Semaphore for limited resources

```
ConnectionPool connections = new ConnectionPool();
Semaphore control = new Semaphore(2);

Process process = new  Process(connections, control);

Thread process1 = new Thread(process, "Customer1");
Thread process2 = new Thread(process, "Customer2");
Thread process3 = new Thread(process, "Customer3");

process1.start();
process2.start();
process3.start();
```

Supplier-Manufacturer



Interthread Communication

```
class Plate {
      boolean isSpaceInDish = true;

      synchronized String eat () throws InterruptedException {
        if (isSpaceInDish ) wait ();
              isSpaceInDish = true;
        notify ();
        ---
       }

       synchronized void serve (String itemName) throws InterruptedException {
        if (! isSpaceInDish ) wait ();
          isSpaceInDish = false;

        notify ();
       }
}
```

Interthread Communication (Cont…)

```
// Guest thread…
Plate dinnerDish;
String item;
---
public void run() {
        try {

                while (!((item= plate.eat()).equalsIgnoreCase("FingerBowl")));

        } catch (InterruptedException ie) {
            System.out.println("Exception while getting string.");
        }
}
```

Interthread Communication (Cont...)

```
PantryItems pantryItems; // List of items cooked.
Plate dinnerPlate;

public void run() {
        try {
                for (int i = 0; i < pantryItems.getCount(); i++) {
                    dinnerPlate.serve(pantryItems.nextItem());
        }
        dinnerPlate.serve("FingerBowl");
        } catch (InterruptedException ie) {
             ----
        }
}
```

Interthread Communication (Cont…)

```java
public class TestCommunication {

        public static void main(String[] args) {
                Plate dinePlate = new Plate();
                PantryItems pantryItems = new PantryItems();

                Thread waiter = new Thread(new Waiter(dinePlate, pantryItems));
                Thread guest  = new Thread(new Guest(dinePlate));

                waiter.start();
                guest.start();
        }
}
```

Exchanger

A synchronization point where threads can pair and swap elements within pair.

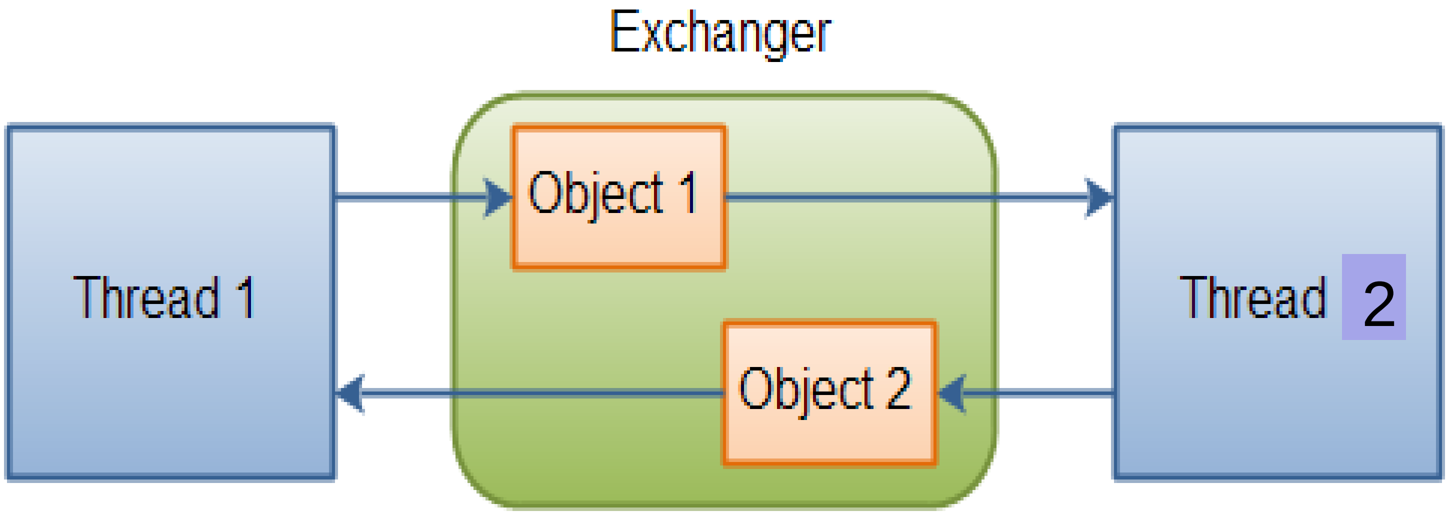


Diagram Courtesy: http://tutorials.jenkov.com/java-util-concurrent/exchanger.html

Exchanger-Manufacturer

Service	Purpose	Remark
Exchanger()	Creates new exchanger.	
exchange(Object)	Waits till another thread arrives to exchange point and then exchanges objects with another thread.	If interrupted, comes out of wait state.
exchange(Object, timeOut, timeUnit)	Waits till another thread arrives to exchange point and then exchanges objects with another thread.	If interrupted or time out, comes out of wait state.

Exchanger- Manufacturer

```
public class Manufacturer implements Runnable {
    Exchanger<DataBuffer> exchanger;
    DataBuffer initialEmptyBuffer = new DataBuffer(); // Defensive copy

      public void run() {
            DataBuffer currentBuffer = initialEmptyBuffer;

            while (currentBuffer != null) {
                // Create result/manufacture product

                currentBuffer.setX(processedValue); // Fill object for exchange

                // Exchanging object
                currentBuffer = exchanger.exchange(currentBuffer);
            }
    }
}
```

Exchanger- Supplier

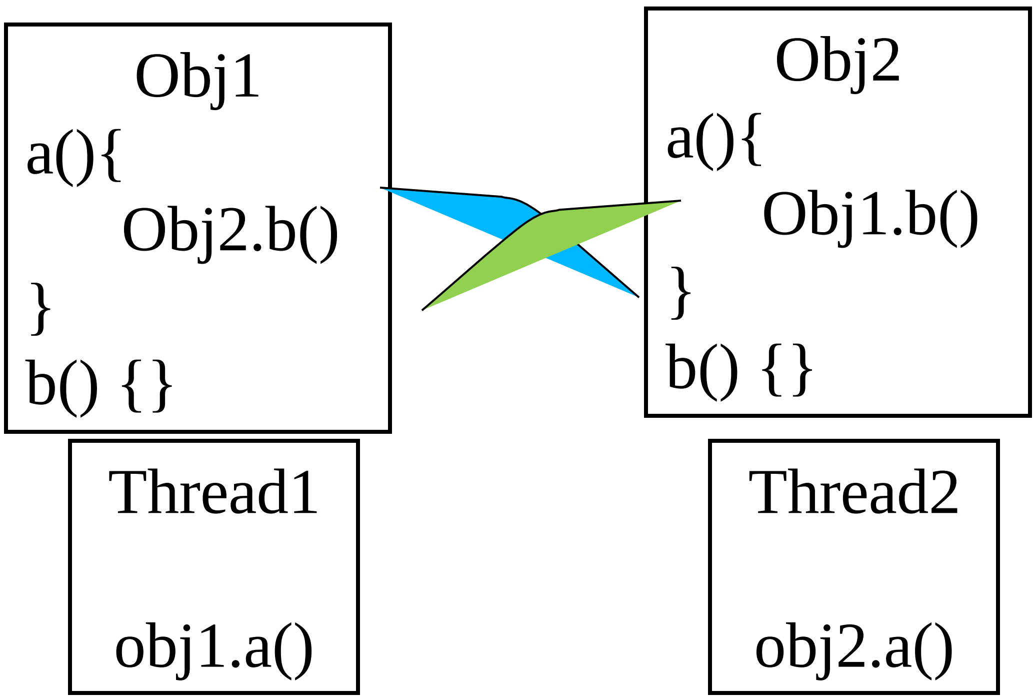
```
public class Supplier implements Runnable {
    Exchanger<DataBuffer> exchanger;
    DataBuffer initialFullBuffer = new DataBuffer(); // Defensive copy

    public void run() {
        DataBuffer currentBuffer = initialFullBuffer;
        while (currentBuffer != null) {

            int x = currentBuffer.getX(); // get Processed value

            // Exchanging empty object
            currentBuffer = exchanger.exchange(currentBuffer);
        }
    }
}
```

Exchanger

```
Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();

// Task submission
Thread supplier = new Thread(new Supplier(exchanger));
Thread manufacturer = new Thread(new Manufacturer(exchanger));

// Task execution
supplier.start();
manufacturer.start();
```

Deadlock

- Two threads try to access each others monitors result into a deadlock or a deadly embrace.
- Neither thread will be able to run in order to acquire the object lock.



Deadlock…

```
class Services1 {
    private Services2 service2;
     public synchronized void service1(){
          lightProcessing();
          service2.service2();
     }
     public synchronized void service2(){
        ------
     }
}
```

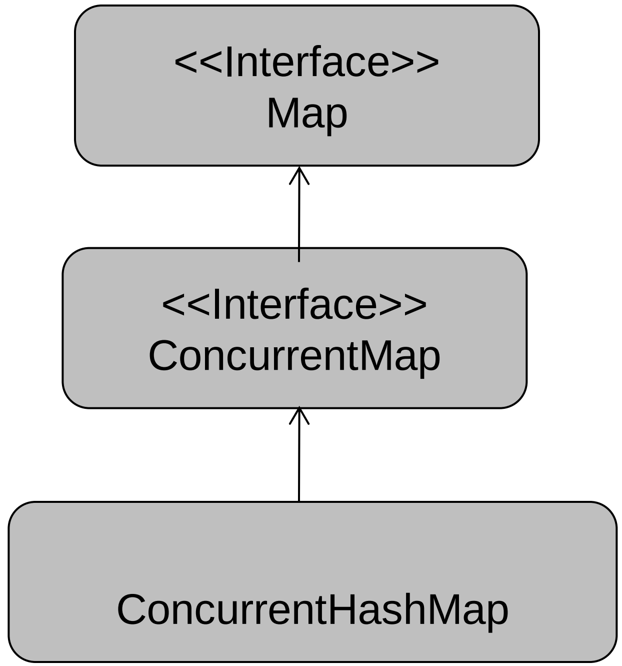
Deadlock…

```
class Services2 {
    private Services1 service1;
     public synchronized void service1(){
         lightProcessing();
         service1.service2();
     }
     public synchronized void service2(){
         ------
     }
}
```

Collection- Concurrent Vs Synchronized

Synchronized Collection	Concurrent Collection
Uses blocking algorithms	Uses non-blocking algorithms
Proves bottleneck in exploiting parallel processing.	Gives better efficiency in parallel processing.
Better data consistency if threads want up-to-date view.	If updates happening frequently and reads happening occasionally.

Concurrent Hash map



It's a thread safe implementation of hash table with non-blocking operations.

- Does not allow 'null' as a key or value.
- The retrieval operations do not entail locking.

ConcurrentHashMap

S.N.	Service	Particulars
1	V putIfAbsent(K key, V value)	If specified key is not associated with value then associate it with given value.
2	boolean remove(K key, V value)	Removes entry for a key only if currently mapped value is ‘value’.
3	V replace(K key, V value)	Replaces entry for a key only if it is currently mapped to some value.
4	boolean replace(Key key, Vo oldValue, Vn newValue)	Replaces entry with newValue only if key is mapped with oldValue.

ConcurrentHashMap

Map<Integer, String> vessel;

vessel = **new ConcurrentHashMap<Integer, String>();**

list.put(key, record);

Atomic Variables

- Supports lock-free thread safe programming on single variables.
- They extend notion of volatile values, fields and arrays.
- Also provide atomic conditional update operation-
  boolean compareAndSet(expectValue, updateValue);

Class	Purpose
AtomicBoolean	boolean value that may be updated atomically
AtomicInteger	‘int’ values that may be updated atomically.
AtomicIntegerArray	‘int array’ in which elements may be updated atomically.
AtomicLong	‘long’ values that may be updated atomically.
AtomicLongArra	‘long array’ in which elements may be updated atomically.
AtomicReference	‘object reference’ that may be updated atomically.
AtomicReferenceArray	‘reference array in which elements may be updated atomically

Daemon Threads

The service provider threads which are expected to provide services to non-daemon threads.

Non-Daemon	Daemon
The service, task or worker threads to execute program code.	The service provider threads to make services available to child threads.
The JVM doesn’t terminate until all these threads are terminated/naturally completed.	The JVM may kill them instantly without prior intimation
They can go into dead state naturally.	They are alive unless terminated by JVM
Its child thread by default is non-daemon.	Its child thread by default is daemon.
	The GC is a kind of Daemon Thread.

Setting thread as Daemon…

thrd.setDaemon(boolean);

Thread Group

- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- Set of threads and sub thread groups that have equal claim to CPU.
- A kind of encapsulation for threads. A thread can access other threads of same or sub threads groups.
- A thread group put a cap on priority on threads of group.

ThreadGroup

Service	Purpose	Remark
ThreadGroup(name)	To construct group object and to name it.	
ThreadGroup(ThreadGroup parent, name)	To create new thread group within parent thread group.	
void destroy()	Destroys empty thread group	Throws IllegalStateException if group is not empty.
void setMaxPriority(int pri)	Set the priority for a group as smaller of 'pri' and priority for parent group.	
int activeCount()	Returns estimate of number of active threads in a group.	
int activeGroupCount()	Returns estimate of number of active children groups in a group.	
void list()	Prints a list of information about a group.	
ThreadGroup getParent()	Returns the parent thread group.	

ThreadGroup

ThreadGroup groupA = **new ThreadGroup("Group A");**
groupA.setMaxPriority(5);
NewThread ob1 = new NewThread("One", groupA);
ob1.setPriority(10);
NewThread ob2 = **new NewThread("Two", groupA);**
ob2.setPriority(3);

// Methods on group
ThreadGroup name is: **groupA.getName()**
Active threads: **groupA.activeCount()**
The maximum priority of a Thread that can be contained: **groupA.getMaxPriority()**
Active Groups: **groupA.activeGroupCount()**
Parent is: **groupA.getParent()**

Just Minute…

Module 18. Reflection and Annotation

- **Overview**
  - Reflection mechanism
  - The Class, Field and Method classes
  - Using Assertion, Enabling and Disabling assertion
  - Annotation

Reflection

- Find out class of an object
- Grab information about a type's modifier, constructor, fields, methods, superclasses, etc.
- Find out the contract of a types interface
- Runtime set and get process on an objects unknown field.
- Invocation of an unknown method of an object in runtime

Reflection (Cont…)

```
public class ReflectionDemo{
      public static void main (String [] args) {
        try {
                Class c = Class.forName ("Module04.inheritance.version01.BankAccount");
                Method mds [] = c.getDeclaredMethods ();
                System.out.println("Method Summary : ");
                for (int i = 0; i < mds.length; i++)
                        System.out.println (mds[i].toString ());
                Constructor ctor []= c.getConstructors();
                System.out.println("\nConstructor Summary : ");
                for (int i = 0; i < ctor.length; i++)
                        System.out.println (ctor[i].toString ());
                Field flds[]= c.getFields();
                System.out.println("\nFields Summary : ");
                 for (int i = 0; i < flds.length; i++)
                        System.out.println (flds[i].toString ());
        }       catch (Throwable e) {   System.err.println (e);       }     }
```

Annotation

```
//An annotation example
    @interface MyAnno {
        String str();
        int val();
    }
```

Annotation Retention Policy

RetentionPolicy.SOURCE

RetentionPolicy.CLASS

RetentionPolicy.RUNTIME

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

Obtaining Annotations at Run Time

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
};

class AnnotationTest {
  //Annotate method
  @MyAnno(str = "Annotation Example", val = 100 )
  public static void myMeth() {
    AnnotationTest ob = new AnnotationTest();
    try {
      Class c = ob.getClass();
      Method m = c.getMethod("myMeth");
      MyAnno anno = m.getAnnotation(MyAnno.class);
```

Obtaining Annotations at Run Time(Cont…)

```
      System.out.println(anno.str() + " " + anno.val());
    } catch(NoSuchMethodException e) {
     System.out.println("Method not found");
    }
  }
  public static void main(String args[ ]) {
    myMeth();
  }
}
```

The Built-In Annotations

- From java.lang.annotation
  - @Retention
  - @Documented
  - @Target
    - Target constant
      - ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE
  - @Inherited

- From java.lang
  - @Override
  - @Deprecated
  - @SuppressWarnings