**Java In-Depth: Become a Complete Java Engineer!**

# Section 1: Course Introduction

## What will this course cover?

This course consists of Core Java which is roughly 50-60% (Section 1-17) of the course

* We will study simple syntax, inheritance and polymorphism
* Generics, Exceptions
* Input-Output: reading and writing data

Advance Java is the second part of this course:

* Advanced Generics
* Multi threading
* Databases and JDBC
* Functional programming and Lamba expressions
* Using Java EE to produce full stack apps

# Section 2: A High-Level Overview

## What is Java

Java is multi-threaded, meaning we can run multiple processes concurrently. Platform independent – allows java to run on different types of machines and different OS’s

## What is Compilation

**Machine Code**

A computer executes instructions of a program in machine code (binary)

**Assembly Language**

This was created by computer scientists to create programs. Assembly language is converted to machine code using assembler

This includes low level details like memory locations, this is still a very involved

**High level Language**

These are far more like plain english and hides low level details. In general assembly code has far more peformance which is why operating systems are written in assembly

**Source Code**

Source code is the code written in high level language

**Compilation**

Compilation is the process of converting source code to a target language. This is can be mnachine code, bytecode or another programming language

**Core Compilation Operations**

Verification of syntax and semantics of code, code optimisations, Generate machine code

## Platform Dependency

**What is Platform Dependency?**

If we compiled a C file on a windows machine, it would produce and executeable

If we compiled a C file on Linux then a out file would be produced.

These two outputs are not cross compatible and if you wanted to run the program you would need to compile it on the target machine

**Why have platform dependency**

Operating System, system calls (file opening, writing, printing to console), hardware differences

## Interpreter

### What is an Interpreter

Converts source code to results

**Virtual Machine**

An interpreter is essentailly a virtual machine, it simulates a CPU using the fetch and execute cycle

**CPU Fetch and Execute Cycle**

This is the most fundamental process of a CPU

When the program is executed and machine code is place in memory and the CPU fetches an instruction from memory as well as fetch data, and data is stored within memory.

**Interpreter Fetch and Cycle Cycle**

The Interpreter fetches the next statement of of the source code and understands the statement

Then executes the statement using precompiled machine code. E.g. pop a stack

**Platform Independence**

The intreper is specific to the platform which then produces machine code specific to the platform

**Limitations:**

Costly memory access, assemby is efficient inherintely

Source code is reinterpreted every time

## Platform Independence

### Why is Java code Platform Independent?

The Java compiler converts the source code to Java bytecode

The Java interpreter (JVM) converts compiled byte code to machine code

The JVM is specific to each platform but the Java bytecode is platform is independent

Java source code within a Java file is compiled into Java bytecode, this produces a class which is then interpreted into machine code by the JVM on the target machine

### Execution Speed

Bytecode intepretation is much faster

JVM also does Just-In-Time compilation which allows for faster programs

## An overview of Java SE

### What is Java SE

The Java Standard Edition is used to develop stand alone applications which typically run desktops

This can be used for hospital management systems

### What is Java EE

This is used to make large scale applications for servers

For example a website which can be used in real time by large amounts of users

This can be used for eCommerce websites

Its built on Java SE

### What is Java ME

This is used for applications for resource constrained devices

This is a subset of Java SE

### What is a JDK?

This includes as an implementation of the JVM specification, code for Java API and Developer Tools

### What is JRE?

This is an environment of just running Java applications do not develop it

The JRE consists of the JVM and Java API

# Section 3: Class, Object and their Members

## Class and Objects

### What is a class?

Class is a blueprint of a real entity with behaviours and state

An object is an instance of a class with a state (data) and behaviour(methods)

**For Example:**

Class

State

methods

Student

id

name

gender

updateProfile()

Class members refer to methods and variables of a class

|  |
| --- |
| **public** **class** Student {  //variable declarations  **int** id;  String name;  String gender;    //method definitions  **boolean** updateProfile(String newName) {  **this**.name = newName;  **return** **true**;  }  }  ///////////////////////////////////////////////////////////  **public** **class** StudentTest {  **public** **static** **void** main(String[] args) {  Student student = **new** Student();  student.id =1;  student.gender = "Male";  student.name="Shiv Kumar";  student.updateProfile("Jonny");  }  } |

## Demo: Absolute Java Basics

### Syntax of variables and methods

Java is case sensitive, so foo() and Foo() would be read as different methods

**Operations**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int** i = 6;  **int** j = 2;  System.***out***.println(i+j); // 8  System.***out***.println(i-j); // 4  System.***out***.println(i\*j); // 12  System.***out***.println(i/j); // 3  System.***out***.println(i%j); // 0  } |

## Variables: Introduction

Variables in java are static, meaning once a variable is declared int it can never contain a string

Variables containing basic data like numbers/characters is a primitive variable

Variables containing non primitives are called object references this includes Strings

Variables containing references to objects are called objective reference. E.g., Student s = new Student() is an object reference

### Variable Declaration

<type> <variableName> = [literal or expression]

These are known as a declaration statement and can appear anywhere in the class

### Reinitialising Variables

Once a variable is initialised, it can be reinitialised but not within the class

### Example

|  |
| --- |
| **public** **class** StudentClass {  **int** id;  **int** nextId = id+1;    **void** compute() {  id=1000; // assignment statements CAN appear in methods but not at class level  System.***out***.println("id: "+id);  System.***out***.println("nextid: "+nextId);  }  **public** **static** **void** main(String[] args) {  StudentClass student= **new** StudentClass();  student.compute(); // 1000 //1  }  } |

## Assignment 1:

|  |
| --- |
|  |

## Vairables: Primitive Types

### Primitive types

There are 8 primitive types

1. Boolean
2. Number -> Integer ->byte, short, int, long,

-> Floating Point -> float, double

-> Character -> char

### Demo: Primitive Integers

When declaring a short, byte or int in an expressio, Java intrepets the literal as an int regardless of the datatype. We can also have long literals too

|  |
| --- |
| **int** id =1000;  **byte** age =18;  //byte rank = 165;  // the above will not compile as 165 is not in range of byte  //int phone = 222222222222222;  // the above is an invalid literal as the number is not in range of int  **long** phone = 2222222222222L;    //boxed primitives:  **int** minVal = Integer.***MIN\_VALUE***;  **int** maxVal = Integer.***MAX\_VALUE***; |

## Floating-Point Numbers

We have floats and doubles

Float has 6-7 decimal places at the most

Double has twice the precision with 15-16 decimal places

### Internal Representation

Java internal represents floating point numbers using the IEEE 754 floating point scheme

## Demo - Primitive Variables: floating-point Numbers

|  |
| --- |
| // float gpa = 3.8; - this is not valid as the literal is interpreted as a double bly default  **float** gpa = 3.8f;  **double** height = 11.1; // we can also write the literal as 11.1d    //scientific notation  **double** largeNumber = 3.8E10;    //large precision  **double** largePrecision = 3.88888833354354449469162;  **float** lowPrecision = 3.88888833354354449469162f;    **public** **static** **void** main(String[] args) {  DemoFloatingPointNumbers d = **new** DemoFloatingPointNumbers();  System.***out***.println(d.largeNumber); //3.8E10  System.***out***.println(d.largePrecision); //3.8888883335435445 - truncation has occured  System.***out***.println(d.lowPrecision); //3.8888884 - floats have less precision  } |

## Demo – Floating point pitfalls

|  |
| --- |
| System.***out***.println(1-0.9); // 0.09999999999999998  System.***out***.println(1-0.7); // 0.30000000000000004  // these issues are a result of how real numbers are represented in binary  // we can not exactly represent 0.1 ibn binary  // when working with money, you should not use float/decimal  // we can use BigDecimal to help represent numbers exactly    BigDecimal first = **new** BigDecimal("0.1");  BigDecimal second = **new** BigDecimal("0.2");    System.***out***.println(first.add(second)); // 0.3 |

## Character Data Type

These are single letter characters

They are internal represented as unsigned integers from 0 – 2^16 -1

## Variable Kinds

A variable plays a certain role depending on where its declared

### Instance Variables

These are variables declared in class level and can only be accessed from instances of the class

### Local Variables

These are variables declared within methods. Note: these do not get default values when declared

### Static variables

These are variables owned by the class and not by the instance. This is also known as a class variable

|  |
| --- |
| **public** **class** DemoVariableKinds {    **static** **int** *computeCount*;    String name; // these un initalised variables would get default values  String Gender;  **int** id;  **int** age;  **boolean** international;  **double** tutitionFees= 12000.0;  **double** internationalFees = 500.0;    **void** compute() {  *computeCount*++;  System.***out***.println("id: "+id);  System.***out***.println("name: "+name);  System.***out***.println("Gender: "+Gender);  System.***out***.println("id: "+id);  System.***out***.println("age:: "+age);  System.***out***.println("international?: "+international);  System.***out***.println("computeCount: "+*computeCount*);  }    **public** **static** **void** main(String[] args) {  DemoVariableKinds student = **new** DemoVariableKinds();  student.id=11;  student.name="Shiv";  student.name = "john";  student.age=17;  student.international=**false**;  student.compute(); //1  student.compute(); //2  DemoVariableKinds student2 = **new** DemoVariableKinds();  System.***out***.println(student2.*computeCount*); //2  // we would typically access this static variables by calling the class:  System.***out***.println(DemoVariableKinds.*computeCount*); //2  }  } |

## Multi Declaration Variables

We can declare multiple variables of the same type on the same line:

|  |
| --- |
| String var1, var2= "hello", var3; |

## Type Casting

### What is Type Casting

This is when we assign a variable or literal to a variable of another type

Only numeric to numeric casting can occur – this includes characters

### Implicit Casting

This is when we cast from a type of a small range to a type of a larger range

E.g. int x = 65;

Long y = x;

### Explicit Casting

This is when we assign a variable of a larger range to a type of smaller range

E.g long y = 42L;

Int x = (int)y;

### Example

|  |
| --- |
| System.***out***.println((2+3)/2); // 2  //explicit casting  System.***out***.println((**double**) (3+2)/2); //2.5 |

## Object Reference

We have reference types when we have a variable which has type of a class or interface

E.g. Student s = new Student();

This statement achieves 3 things in JVM

**Student s** allocaties space for reference variable

**New Student()** allocates space for new student onject

**=** assigns the memory address of student object to variable s

### Where Objects Stored?

Objects are stored in the heap in memory

## Satements

All statements end with a semi colon and are commands which are to be executed. It changes the state of the program

Statements involve one or more expressions

### Statement Types

1. Declaration statements e.g. count = 25;
2. Expression statements, e.g. assigning values, calling methods
3. Control flow statement, e.g. if else statements

IMPORTANT: We cannot have expression or control flow statements within the body of our classes

|  |
| --- |
| **public** **class** DemoStatements {  **static** **int** *x* = 4;  //x = 5; this is not allowed    //the below will not compile:  /\*  if (x>1) {  System.out.println("hello");  }  \*/  **public** **static** **void** main(String[] args) {    }  } |

## Arrays

### What is an array?

This is a container object which holds a fixed amount of values of same type

## Assignment 2: Currency Converter

### My code:

|  |
| --- |
| **public** **class** CurrencyConverter {  **double**[] exchangeRates = {63.0, 3.0, 3.0, 595.0, 18.0, 107.0, 2.0};    **void** printCurrencies() {  System.***out***.println("rupee: " + exchangeRates[0]);  System.***out***.println("dirham: " + exchangeRates[1]);  System.***out***.println("real: " + exchangeRates[2]);  System.***out***.println("chilean\_peso: " + exchangeRates[3]);  System.***out***.println("mexican\_peso: " + exchangeRates[4]);  System.***out***.println("\_yen: " + exchangeRates[5]);  System.***out***.println("$australian: " + exchangeRates[exchangeRates.length-1]);  }    **public** **static** **void** main(String[] args) {  CurrencyConverter cc = **new** CurrencyConverter();  cc.printCurrencies();  }  } |

## 2D Arrays

### Creating a 2D array

Int[][] myArray = new int[4][2];

JVM internall represents a 2D array of length 4 and each element is an array of length 2

You can see this by looking at the declaration:

Int[][] 2dArray = new int[r][c] => this is an array with a type of int[] – another array

### Array with Irregular Rows

We can have a 2d array where the rows have different number of columns

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[][] array = **new** **int**[2][]; // this array has 2 rows    array[0] = **new** **int**[] {1,2,3};  array[1] = **new** **int**[] {4};    System.***out***.println(array[0][0]); //1  System.***out***.println(array[0][1]); //2  System.***out***.println(array[0][2]); //3  // System.out.println(array[0][3]); - out of bounds  System.***out***.println(array[1][0]);  // System.out.println(array[1][1]); - out of bounds      } |

## 3D Arrays

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[][][] unitsSold = **new** **int**[][][]{  { // New York  {0,0,0,0}, // Jan  {0,0,0,0}, // Feb  {0,0,0,0}, // Mar  {0,850,0,0}// Apr  },  { // San Francisco  {0,0,0,0}, // Jan  {0,0,0,0}, // Feb  {0,0,0,0}, // Mar  {0,0,0,0} // Apr  },  {  {0,0,0,0},  {0,0,0,0},  {0,0,0,0},  {0,0,0,0}  },  {  {0,0,0,0},  {0,0,0,0},  {0,0,0,0},  {0,0,0,0}  }  };    System.***out***.println("unitsSold[0][3][1]: " + unitsSold[0][3][1]);  } |

## Methods

### What is a method?

A method is self contained logic which can be called many times and read and generate data

### Method Declaration

|  |
| --- |
| returnType methodName(type parameter1, type parameter 2...) {  ...  <**return** someValue>  } |

The method signature is the methodName and its parameters

|  |
| --- |
| **public** **class** Methods {  **static** **double** sum(**double** x, **double** y) {  **return** x+y;  }  **static** **float** floatSum(**float** x, **float** y) {  **return** x+y;  }  **public** **static** **void** main(String[] args) {  System.***out***.println(*sum*(2, 2.3)); //4.3  // we can do implicit casting  // so we can use floats instead of double  System.***out***.println(*sum*(12.2f, 12.3f)); //24.5    //System.out.println(floatSum(1.23f, 1.23d)); however we cant do this due to inability to implicit cast  //however we can do explicit cast:  System.***out***.println(*floatSum*(1.2f,(**float**)1.2d)); //2.4  }  } |

## Method Types

A Class can have instance methods and static methods

### Instance Methods

These are methods which can only be called by objects

Invocation: **ObjectReference.methodName()**

Typically affect the object state but can also affect other instances

### Static Methods

These have no axxess to the variables/methods of instances however they can access other static variables and methods

invocation: **className.methodName()**

## Assignment 3: Currenct Cinverter – invoking methods of an object

My Solution

|  |
| --- |
| **public** **class** MoneyTransferService {  CurrencyConverter cc = **new** CurrencyConverter();    **double** computeTransferAmount(**int** countryIndex, **double** amount) {  **return** cc.computeTransferAmount(countryIndex, amount);  }  **double** computeTransferFee(**int** countryIndex, **double** amount) {  **return** cc.computeTransferAmount(countryIndex, amount)\*0.02;  }      **public** **static** **void** main(String[] args) {  MoneyTransferService mts = **new** MoneyTransferService();  System.***out***.println( mts.computeTransferAmount(0, 1000)); //63000.0  System.***out***.println( mts.computeTransferFee(0, 1000)); //1260.0  }    } |

## Methods: Passing Data

We can either pass primitives or object references into methods

### Pass by Value

Value of argument is passed to paramter

## Method Overloading

This is when we have methods of the same name but a different parameter list

We cannot have the same parameters but a different return type

### Valid Examples

Orginal: Void updateProfile(in newId){}

Void updateProfule(int newId, char gender){}

Void updateProfile(char gender, in newId){}

Void updateProfile(short newId){}

### Inalid Examples

Boolean updateProfile(int newId){}

Void updateProfile(int id){}

Static void updateProfile(){}

## Methods: Varargs

This a parameter with a variable number of arguments. It can supplement the arguments of a method but also be the only one

### Synatx and Invocation

The varargs paramter is declare by a type following three dots

### Example

|  |
| --- |
| **static** **void** foo(**boolean** truth, **int**... nums){  **if** (truth) {  System.***out***.println(nums.length);  }  }  **public** **static** **void** main(String[] args) {  *foo*(**true**); //0  *foo*(**true**,1,2,3,4); //4  *foo*(**false**); //returns nothing  } |

### Restrictions

Vararg must be the final parameter and we cannot have multiple vararg parameters

The printf is a built in java method which has a vararg parameter. E.g. we can have printf(“this statement has var1: %d and var2: %d”, 1.1,1.2)

## Constructor

The purpose of the constructor is to initialise the properties of an instance

When we are instantiating a class without the constructor, we are actually invoking the default constructor

### Syntax

ClassName{type paramter1, type parameter2…){ … }

### Example

|  |
| --- |
| **public** **class** Student {    **int** id;  Student(**int** newId) {  id = newId;  }    **public** **static** **void** main(String[] args) {  Student student = **new** Student(21);  System.***out***.println(student.id); //21  }  } |

Even if there is a condition which returns in the constructor, the rest of the constructor would still be executed:

|  |
| --- |
| Student(**int** age, **int** id){  **if** (**false**) {  **return**;  }  **this**.age = 1;  **this**.id = 2;    }    **public** **static** **void** main(String[] args) {  Student s = **new** Student(18,1);  System.***out***.println("age: "+s.age+" id: "+s.id); //age: 1 id: 2    } |

## Constructor Demo

|  |
| --- |
| **public** **class** Student {  **static** **int** *studentCount*;    **int** id;  String name;  String gender;  **int** age;  **long** phone;  **double** gpa;  **char** degree;    **boolean** international;  **double** tuitionFees = 12000.0;  **double** internationalFees = 5000.0;    Student(){ // no args constructor  }    Student( **int** id, String name, String gender, **int** age, **long** phone, **double** gpa, **char** degree, **boolean** international){  **this**.id = id;  **this**.name = name;  **this**.age = age;  **this**.phone = phone;  **this**.gender = gender;  **this**.gpa = gpa;  **this**.degree = degree;  **this**.international = international;  }    **void** compute() {  *studentCount* = *studentCount* + 1;  **if** (international) {  tuitionFees+=internationalFees;  }    System.***out***.println("id: " +id);  System.***out***.println("name: " +name);  System.***out***.println("gender: " +gender);  System.***out***.println("age: " +age);  System.***out***.println("phone: " +phone);  System.***out***.println("gpa: " +gpa);  System.***out***.println("degree: " +degree);  System.***out***.println("international: " +international);  System.***out***.println("tuitionFees: " +tuitionFees);  System.***out***.println("studentCount: " +*studentCount*);  }    **public** **static** **void** main(String[] args) {    Student s1 = **new** Student(1000, "john","male",18, 123456,3.8, 'B',**false** );  s1.compute();  Student s2 = **new** Student(1001, "shiv", "male", 23, 111111, 3.5, 'C',**true**);  s2.compute();  Student s3 = **new** Student(1002, "norman", "female", 93, 13333333, 2.7, 'D', **false**);  s3.compute();  System.***out***.println(*studentCount*); //3  }  } |

## Constructor Overloading

Method overloading is where we have methods of same name but of different parameters. Similiarily with constructor overloading, we have same method names but different parameter list.

### Why use Constructor overloading

Simply for convience for instantiating classes

### Example

The Java library has a FileOutputStream constructor:

1. FileOutputStream(String name, boolean append)
2. FileOutputStream(String name)
3. FileOutputStream(File file)
4. FileOutputStream(File file, boolean append)
5. FileOutputStream(FileDescriptor fdObj)

**Constructor Delegation**

**Using Constructor overloading, we are able to call a different constructor overload from one constructor overload:**

|  |
| --- |
| Student(){ // no args constructor  }  Student( **int** id, String name, String gender, **int** age, **long** phone, **double** gpa, **char** degree){  **this**(id, name, gender, age, phone,gpa,degree, **false**);  }  Student( **int** id, String name, String gender, **int** age, **long** phone, **double** gpa, **char** degree, **boolean** international){  **this**.id = id;  **this**.name = name;  **this**.age = age;  **this**.phone = phone;  **this**.gender = gender;  **this**.gpa = gpa;  **this**.degree = degree;  **this**.international = international;  } |

### ****Restrictions****

We can not have:

1. The this() statement NOT being the first statement
2. Multiple this() statements
3. Recursive statements, i.e. the this() statement having the same parameter list as the method its contained within
4. Instance variable as parameter

## Constructor Overloading

In the above example we used delegation to go from the **primary constructor** (the constructor overload with the most parameters) to a constructor overload with less arguments.

We can go in the opposite direction also:

|  |
| --- |
| Student( **int** id, String name, String gender, **int** age, **long** phone, **double** gpa, **char** degree){  **this**.id = id;  **this**.name = name;  **this**.age = age;  **this**.phone = phone;  **this**.gender = gender;  **this**.gpa = gpa;  **this**.degree = degree;  }    //primary constructor:  Student( **int** id, String name, String gender, **int** age, **long** phone, **double** gpa, **char** degree, **boolean** international){  **this**(id, name, gender, age, phone,gpa,degree);  **this**.international = international;  } |

## Demo: Reinitialising Object References

|  |
| --- |
| Student s1 = **new** Student(1000, "john","male",18, 123456,3.8, 'B',**false** );  s1.compute();  Student s2 = **new** Student(1001, "shiv", "male", 23, 111111, 3.5, 'C',**true**);  s2.compute();  Student s3 = **new** Student(1002, "norman", "female", 93, 13333333, 2.7, 'D', **false**);  s3.compute();    s1.updateName("jon");  System.***out***.println("student 1's update name: "+s1.name);    //this object reference is pointing to same object in memory  Student s4 = s1;  s4.updateName("Mike");  System.***out***.println(s1.name); //returns Mike |

# Section 4: Method Building Blocks and Control Flow

## Introduction

In this section we will cover how to build methods, use control flow structures, variable scope, recursion

## Operators

This is a symbol which does an operation which produces a result. E.g. +,-\*,/,%

### Operator Types

Assignment, Arithmetic, Comparison, Logical, Bit wise

### Unary, Binary and Ternary Operators

A unary operator acts on one variable. **E.g. x++, -x**

A binary operator acts on two operants. **E.g. x + y**

A ternary operator acts on thee operants. It will always use **?: E.g. (x>0) ? x : 0**

## Arithmetic Operators

The five arithmetic operations in java are: +, -, \*, /, %

### Compound Arithmetic Operators

+=, -=, \*=, /=. %=

### Post and Pre operators

We can have the operator before the variable (pre operator) or after the variavble (post operator)

E.g x++ is a post and ++x is a pre

This affects the way variables are assigned:

int x = 5;

int y = x++; // x,y=6

int y = x; // y = 5

x = x+ 1; //x = 7

**Post Operator**

Int x =5;

int y = ++x; // x,y=6



x = x+ 1; //x = 6

int y = x; // y = 6

**Pre Operator**

## Arithmetic Operations Rules

How is this evaluated: in **x =** **5 + 9 – 3 + 2 \* 5**?

### Operations Precedence Rules

1. Multiplicative operators ( \*, /, %) have higher precedence over additive operators (+, -)

* **x = 5 + 9 - 3 + (2\*5)**

1. Operators of the same group are evaluated left from right:

* **x = ((5 + 9 – 3) + (2\*5))**
* **x = 21;**

### Operand Promotion

Operands smaller than int are ( byte and short) are promite to int

E.g.:

127 (byte) + 1(byte) -> 127 (int) + 1 (int) -> 128 (int)

This includes chars too

‘a’ + ‘b’ -> 195 (int)

### Same Typer operations

If both operands are int, long, float or double ⇒ the operations are carried out in that type and evaluated to that type

5 + 6 -> 11

1 / 2 -> 0

0.5 gets truncated to an integer

### Mixed-Type Operations

If operands belong to different types, then the smaller type is promoted to larger type

Order of promotion: **int** -> **float** -> **double**

1 / 2.0 -> 0.5

1.0 / 2 -> 0.5

**char** + **float** -> **float** + **float** -> **float**

9 / 5 \* 20.1 -> (9/5) \* 20.1

## Demo: Comparison Operators

|  |
| --- |
| //Comparison operators are used to comapre two operands  //relational operators  **int** age = 25;  System.***out***.println("(age > 21): "+(age > 21)); //true  System.***out***.println("(age >= 21): "+(age >= 21)); //true  System.***out***.println("(age < 21): "+(age < 21)); //false  System.***out***.println("(age == 21): "+(age == 21)); //false  System.***out***.println("(age != 21): "+(age != 21)); //true    // >, >= etc are only valid for numeric operands    // System.out.println("a">1); -- invalid  //System.out.println(true >= 1); -- invalid    //object comparison  Student s1 = **new** Student("Shiv");  Student s2 = **new** Student("Shiv");    System.***out***.println("s1==s2: "+(s1 == s2)); //false  s1 = s2;  System.***out***.println("s1==s2: "+(s1==s2)); //true |

## Logical Operators

There are 3 types of logical operators: AND, OR and NOT

Logical operators can allow us to test comparisons on multiple conditions

## Bitwise Operators

Operates on individual bits og operands

The operands include integers (it intrperets the binary representation) and boolean

### Applications

Used mostly in embedded systems where resources are limited

Also used in hashtables, compression and and encryption

### Bitwise Operators

1. **Bitwise AND (&)** – returns 1 if both input bits are 1. E.g. x=1;y=2; => (x & y) -> 1
2. **Bitwise OR (|)** – returns 1 if either input bits are 1. E.g x=1,y=3 => ( x | y ) -> 3
3. **Bitwise XOR (^)** – returns 1 if only one of the input bits are 1. E.g. x=1,y=3 => (x ^ y) = 3
4. **Bitwise Not (~)** – inverts the input bit. E.g. ~(1) -> -2

**Compund Bitwise Assignment**

E.g. the following:

**Operand1 = Operand1 & Operand 2**

**Operand1 &= Operand 2**

Can be simplified

## Bit Shift Operators

There are 3 bit-shift operators:

1. Left Shift Operator: **<<**
2. Unisgined Right Operator: **>>>**
3. Signed Right Operator: **>>**

### Left-shift Operator (<<)

This is equivalent to multiplication of powers of 2

### Unsigned Right Operator

This is equivalent to division by powers of 2

### Signed Right Operator

This is equivalent to division by powers of 2 while preserving sign

## Switch Statement

Switch statements let us look into particular cases that a variable might take

### Example

|  |
| --- |
| **int** month = 3;  **switch** (month) {  **case** 1:  System.***out***.println("Jan");    **case** 2:  System.***out***.println("FEB");  **case** 3:  System.***out***.println("MAR");  **case** 4:  System.***out***.println("APR");  **default**:  System.***out***.println("IDK");  } |

**RESULT**: MAR APR IDK

As you can see, we obtained 3 outputs. The reason for this is because we did not include a break statement at the end of each case. The following code would work as intended

### Using Break

|  |
| --- |
| **int** month = 3;  **switch** (month) {  **case** 1:  System.***out***.println("Jan");  **break**;  **case** 2:  System.***out***.println("FEB");  **break**;  **case** 3:  System.***out***.println("MAR");  **break**;  **case** 4:  System.***out***.println("APR");  **break**;  **default**:  System.***out***.println("IDK"); } |

### Switch Expression Type

These expressions can be of type Integer, String or Enum. DOES NOT work on doubles

### Case Label

Must be in range of data type of switch statement and must be known at compile time

## Control Flow: Ternary

This is a shorthand for if-else with single statements

result = (boolean-expression) ? true-expr : false-expr;

## Control-flow: for Statement

### Example

|  |
| --- |
| **int**[] myArray = { 0,1,2,3,4,5,6,7,8};  **for** (**int** i =0; i<myArray.length;i++) {  System.***out***.println(myArray[i]); } |

### Multiple Condtions

We are able to place multiple expressions/conditions in the for loop

|  |
| --- |
| **for** (**int** i =0, j =myArray.length-1; i<myArray.length && j>=0;i++,j--) {  System.***out***.println("i: "+i+" j: "+j+" i+j: "+(i+j));  } |

## Coding Exercise 3: Computing GPA

### Shiv’s Solution:

|  |
| --- |
| **package** Section4.CodingExercises.Exercise3;  **import** java.util.ArrayList;  **public** **class** StudentUtil {    **public** **static** **double**[] calculateGPA(**int**[] studentIdList, **char**[][] studentsGrades) {  // Grade A =4, B=3, C-2  **double**[] returnedGPA = **new** **double**[studentsGrades.length];  **for** (**int** i=0; i<studentsGrades.length; i++ ) {  returnedGPA[i] = 0;  **for** (**int** j = 0; j <studentsGrades[i].length; j++ ) {  **char** grade = studentsGrades[i][j];  **double** gradePoint = 0;  **switch** (grade) {  **case** 'A':  gradePoint = 4.0;  **break**;  **case** 'B':  gradePoint = 3.0;  **break**;  **case** 'C':  gradePoint = 2.0;  **break**;  }  returnedGPA[i]+=gradePoint;  }  returnedGPA[i] /= studentsGrades[i].length;  }    **return** returnedGPA;  }    **public** **static** **int**[] getStudentsByGPA(**double** lower, **double** higher, **int**[] studentIdList, **char**[][] studentsGrades) {  // perform parameter validation. Return null if passed parameters are not valid  **if** (lower > higher || lower<0 || higher<0) {  **return** **null**;  } **else** {  **double**[] result = *calculateGPA*(studentIdList, studentsGrades);  **int** validStudents = 0;  ArrayList<Integer> validStudentList = **new** ArrayList<>();  **for** (**int** i =0; i<studentIdList.length; i++) {  **if** (lower < result[i] && result[i]<higher) {  validStudentList.add(studentIdList[i]);  }  }    **return** validStudentList.stream().mapToInt(i -> i).toArray();    }      // invoke calculateGPA    // construct the result array and return it.  //You would need an extra for loop to compute the size of the resulting array  }    **public** **static** **void** main(String[] args) {  **int**[] studentList= **new** **int**[] {1001,1002};  **char**[][] studentGrades = **new** **char**[][] {  { 'A', 'A', 'A', 'B' },  { 'A', 'B', 'B' }  };    System.***out***.println(*calculateGPA*(studentList, studentGrades)[0]);  System.***out***.println(*calculateGPA*(studentList, studentGrades)[1]);  System.***out***.println(*getStudentsByGPA*(5, 4, studentList, studentGrades));  }    } |

## While Loop

Whie loops are an iteration statement which has only a condition statement.

Incrementing/decrementing occurs in the body of the while loop

|  |
| --- |
| **while**(condition) {  //do something  // increment something  } |

### Do-While Statements

Unlike While statements, the body of a do-while statement is always executed at least once

|  |
| --- |
| **int** x = 1;  **do** {  System.***out***.println("hello");  } **while** (x>9999); |

### Infinte Loops

Infinite loops are not limited to while loops, we also can have infinite loops in for loops. E.g.:

|  |
| --- |
| **for**(;;) {  System.***out***.println("hi");  } |

## Control Flow: Break Statement

A break statement immediately encloseling switch or loop

|  |
| --- |
| **for**(;;) {  System.***out***.println("hi");  **break**;  } |

### Invalid Syntax

You can not have a break statement without an enclosing if or while statement

### Labeled Break Statement

We can break a specific loop if we have labeled it

|  |
| --- |
| num=0;  outermost: **for** (**int** i =0; i<10; i++) {  **for** (**int** j =0; j<10; j++) {  **if** (i==5 & j ==5) {  **break** outermost;  }  num++;  }  }  System.***out***.println(num); //55 |

## Control Flow: Continue Statement

The continue statement can only occur within a loop unlike breaks which can be used in switch statements

When this statement is executed, the next iteration of the loop is started

### Labeled Continue Statements

We can have labeled continue statements

Label: loop statement

Continue label

### Example

|  |
| --- |
| **int** num = 0;  outermost: **for** (**int** i =0; i<10; i++) {  **for** (**int** j =0; j<10; j++) {  **if** (i == 5 && j == 5) {  **continue** outermost;  }  num++;  }  }  System.***out***.println(num); //95 |

## Recursion

We use recursion to solve a sub-problem of a problem that we are trying to solve with a method

Recursion stops when we reach a base case.

### Factorial

n! = n \* (n-1)! = n \* (n-1) \* (n-2)! = … = n \* (n-1) \* … \* 1!

### Demo

|  |
| --- |
| **static** **int** factorial (**int** n) {  **if** (n == 1 || n==0) {  **return** 1;  } **else** {  **return** n \* *factorial*(n-1);  }    }  **public** **static** **void** main(String[] args) {  System.***out***.println(*factorial*(3)); //6  System.***out***.println(*factorial*(0)); //1  } |

### Example: Binary Search

|  |
| --- |
| **static** **int** binarySearch(**int**[] a, **int** l, **int** h, **int** key) {  **if** (l==h) {  **if** (key == a[l]) {  **return** l;  } **else** {  **return** -l;  }  }    **int** mid = (l+h)/2;    **if** (key == a[mid]) {  **return** mid;  } **else** **if** (key > a[mid]) {  **return** *binarySearch*(a, mid+1, h, key);  } **else** {  **return** *binarySearch*(a, l, mid-1, key);  }  }    **public** **static** **void** main(String[] args) {  **int**[] a = {11, 19, 24, 34, 55, 65, 71, 83, 91 };  **int** index = *binarySearch*(a, 0, 8, 91);  System.***out***.println(index); //8  index = *binarySearch*(a, 0, 8, 65);  System.***out***.println(index); //5  } |

# Section 5: Packages, Strings and Information Hiding Principle

## Chapter introduction

We shall discuss

* the Java API via the Oracle’s website
* the use of packages to organise code, naming conventions
* Strings, string manipulation, how strings are stored internally, optimisations
* Access levels
* The design principal of information hiding

## The Java API

### What is the Java API?

This is a library of well-tested classes developed by Java experts and used by programmer base. It is a component of the JRE and JDK

### Packages of the Java API

Some of the packages of the Java API include:

* **Java.lang** which includes classes like Class, String, Object
* **Java.util** which includes classes like ArrayList, HashMap
* **Java.io** which includes classes like File and BufferedReader

**Why use Packages?**

The packages of the Java API adds meaningful structure to all these classes. This also benefits us with name scoping, i.e. we can have two classes with the same name. This also adds secruity as we can limit interactivity of classes

### Java API: Important Packages

### Java.lang – fundamental classes

1. **Java.util** – Data structures
2. **Java.io** – Reading and Writing
3. **Java.net** – networking
4. **Java.sql** – databases

### Java API Documentation

We can examine the documentation of the Java API on oracle’s website: [String (Java Platform SE 8 ) (oracle.com)](https://docs.oracle.com/javase/8/docs/api/)

**3rd Party APIs**

We are not limited to the Java API, we can also use third party APIs like Hibernate, Spring, Common

### API Benefits

We can focus on writing new logic rather than reinventing the wheel, over time APIs improve in performance and gain new functionality

## Accessing Packages

### Accessing Classes

Same package – direct access, we can just call the class name

Different package – we can either do an import OR use the fully qualified class name

### Import Statement

We can import a package using the import keyword at the top of our Java class

E.g.:

|  |
| --- |
| **import** java.util.ArrayList;  **class** FooClass {  **void** foo() {  ArrayList list = **new** ArrayList();  }  } |

### Import Single vs Multiple Classes

We can import a single class from a package, this would be an explicit import

We can use \* import to import multiple classes on demand

### \* import

This imports *all* classes in a package:

|  |
| --- |
| **import** java.util.\*; |

### Why should we avoid \* Imports?

\* import can break the code

Suppose we want to use the Date class from the java.util class and imported java.util.\*

If we had another \* import, and that package introduced a new class with name Date. This would result in compilation error

### Invalid Imports

We cannot use two explicit imports for classes with identical names

E.g., if java.sql did have a Date class then the following would be invalid:

|  |
| --- |
| **import** java.util.Date;  **import** java.sql.Date; //invalid |

### Default Input

Java.lang is imported by default!!!

## Creating Packages and Classpath Management

We use periods in the packages to represent a subdirectory in the windows file explorer

### Package Statement

As a result, we need a package statement within our class

E.g.

|  |
| --- |
| **package** Section4.Demos; |

### Effect of Creating Package

The package name is now part of the class name

|  |
| --- |
| **java** BasicsDemo //will NOT work  **java** com/semanticssquare.basics.BasicsDemo //will work |

## Naming Packages

### Package Naming

Let’s look at a scenario where package naming conventions have not been used.

**Scenario:** let’s say Anita from stanford.edu has created a public package called math.geometry and also john from oracle.com creates a public package called math.geometry

If someone were to use these pacakges, issues will arrise from the JVM

This is why we have the reversed internet domain name

### Reverse Internet Domain Name

Anite can call her package: **edu.stanford.math.geometry**

John can call his package **com.oracle.math.geometry**

### Component Naming Conventions

* Lower cased digits
* Short
* Meaningful abbreviations and acronyms
* Never start with Java or Javax

## Access Levels

What if we wanted to limit a class within a package to only be accessible within the same package. Or what if we want to not give any access to a class to the user

Access levels provide restrictions on accessing classes and their members

### Accessibility for Classes/Interfaces

We can provide access to classes inside and outside of packages using the **public** keyword

### Accessibiloty for Class Members

We may want to restrict the access of a member only within the class, this is achieved using **private** keyword

We can make a member private to the class -this would be implicity achieved with the **default** keyword

We may want to only limit access to within the class and its subclasses (these could be outside the package) – we would use the **protected** keyword

## More on Private Modifier

If we have properties with the private access modifier, then that property is private to the class.

However this does not imply properties are private to the objects of its class

**Going back to the Student class, if we have two student instances with a private id then those two instances CAN access the id of the other instance**

One reason for this paradigm is that in order to compare instances of a class, we need to compare whether private fields on instances are equal or not

## Strings: Introduction

### Strings

Strings are instances of the class **java.lang.String**

String literals are also instances of the String class

We can therefore pass String literals as a parameter to the String constructor

**E.g. String s = new String(“my String”)**

We can also pass a char array to construct strings.

E.g. **char[] charArray = { ‘h’, ‘e’, ‘l’, ‘l’, ‘o’ };**

**String s = new String(charArray);**

However it is recommended to just instantiate String from the literals: **String s = “string”;**

### String Representation

Internall, Java Strings are stored as a character array to store text. These characters are uicode chracters

Strings are immutable – once defined it cannot be changed

String literals are stored in the String Pool ~ this saves memory. This is why we recommend the use of literals vs constructor instantiation

## String Class

### Common Operators:

* Comparing
* Searching
* Examine individual characters
* Extract substring
* Case translation
* Replace
* Split

### 3rd Party String Utilities

Some third party utilities include:

* StringUtils – Apache Commons Lang
* Guava’s String Utility Class

## Assignment 4: Calculating Term Frequency in a Document

You will implement a measure called “term frequency”. This is a vital measure used by search engines like Google to measure the relevance of a document based off a search

### Term Frequency Calculation

We calculate Term Frequency as:

**tf**(term, document) = number of times term appears in document **/** total number of words in document

### Pseudocode

We wish to implement the following pseudocod:

|  |
| --- |
| public class **IRUtil** {      public static int getFrequencyCount(String term, String doc) {          Step 1: Convert both term & doc into lower-case          Step 2: Initialize variable frequencyCount to 0          Step 3: Initialize variable index with the index position of term in doc. Hint: Use indexOf() method          Step 4: If index >= 0 (i.e.., term appears in doc), then go to next step. Else go to last step.          Step 5: Increment frequencyCount          Step 6: Re-compute doc with the string appearing **after** term till end of doc. Hint: You can use substring() & length() methods          Step 7: Re-compute index with index position of term in the re-computed doc. Go to step 4 to continue processing.          Step 8: Return frequencyCount      }      public static double termFrequency(String term, String doc) {          Step 1: int frequencyCount = getFrequencyCount(term, doc);          Step 2: Create variable totalTermCount to hold the total number of terms appearing in doc. Hint: You can use split() with white-space " " as delimiter          Step 3: return frequencyCount / totalTermCount. This is the tf formula.      }  } |

### My Solution

|  |
| --- |
| **public** **static** **int** getFrequencyCount(String term, String doc) {  term = term.toLowerCase();  doc = doc.toLowerCase();  **int** frequencyCount = 0;  **int** index = doc.indexOf(term);    **while** (index>= 0) {  frequencyCount++;  doc = doc.substring(index+term.length(), doc.length());  index = doc.indexOf(term);  **if** (index>=0) {  **continue**;  } **else**  **break**;    }  **return** frequencyCount;    }  **public** **static** **double** termFrequency(String term, String doc) {  **int** frequencyCount = *getFrequencyCount*(term, doc);    **int** totalTermCount = doc.trim().split(" ").length;    **return** (**double**) frequencyCount / totalTermCount;  } |

NOTE: I got it right!!!

## String Pool and Interning

### String Literal + Using new

When strings are created through literals, the string is stored in the ***string pool*** on heap

Literals of the same contentg share the same storage location

Strings created through the new keyword is treated as an object and each object has its own location. I.e. there is no sharing of storage. This would be stored within the heap and point to a literal in the string pool

|  |
| --- |
| String s1 = "hello world";  String s2 = "hello world";  System.***out***.println(s1 == s1); // true  String s3 = s2;  System.***out***.println(s3 == s1); //true  String s4 = **new** String("hello world");  System.***out***.println(s4 == s1); // false  String s5 = **new** String(s4);  System.***out***.println(s4 == s5); // false |

### String Pool

String pool stores a single copy of each string literal as string object

There is only one string ppoll

### String interning

This is the process of building a String pool

Each string element is an **intern**

### String Interning by the JVM

Whem encountering a string literal for the first time:

* Creates new String object with given literal
* Invokes inten():

|  |
| --- |
| If (string in string pool) {  Return existing reference  } else {  add to string pool and return reference  } |

### Few Examples

String s = “hel” + “lo”; ~ interned

String s1 = “lo”;

String s2 = “hel” + s1; ~ not interned

s2 = s2.intern; ~ explicit interning

**Is explicit interning useful?** Most cases its not not!

## String Immutability

Once a string is initialised, then it cannot be assigned its value cannot be altered. This isn’t to suggest that the value of an object can’t be changed because it certainly can:

|  |
| --- |
| String s1 = "1234";  s1 = "2345"; |

But the above object is abandoned

### Why Immutability?

1. String interning – if strings were mutable then storage sharing would not be possible
2. Concurrency – if strings were mutable, multithreading would pertain to issues
3. Secruity

## String Concatenation

When using the plus operator, non string values are coerced into string objects

As we know, Java evaluates operations from left to right:

|  |
| --- |
| System.***out***.println("Hello "+ "world " + 12 + 2); // Hello world 122  System.***out***.println(12+2+" Hello "+ "world"); //14 Hello world |

### StringBuilder

This class was created in Java 5

E.g.:

|  |
| --- |
| StringBuilder sb = **new** StringBuilder();  sb.append("hello ");  sb.append("world");  String s = sb.append(" Good").append(" Morning").toString();  System.***out***.println(s); // hello world Good Morning |

As StringBuilder does allow for mutability, it is not thread-safe as its not synchronised

### Stringbuffer

This is an OBSOLETE class, however it is synchronised but as a result is **slow** in performance

### Demo

|  |
| --- |
| String str = "Hello "+"World!";  StringBuffer strB = **new** StringBuffer(str);  strB.append(" morning :)");  System.***out***.println(strB.toString()); // Hello World! morning :)  System.***out***.println(strB.length()); // 23  strB.delete(1, 5);  System.***out***.println(strB.toString()); // H World! morning :)  strB.insert(1, "ey");  System.***out***.println(strB.toString()); //Hey World! morning :) |

## String Concatenation Performance

Using the plus operator is not the most performance efficient method.

Each time you concatenate a string, a copy of *both* strings is made

A new StringBuilder is instantiated and appends the content of the strings

And finally a toString() method is called on the StringBuillder object.

This is not very efficient if you are concatenating a list of strings to one string

## Escape Sequence

In order to include certain characters within a string literal, we need to use escape sequences

We can escape most characters using backslash: \

## Coding Exercise 4

Implement the methods of the following class:

|  |
| --- |
| public class SentimentAnalyzer {  // Tip: labeled continue can be used when iterating featureSet + convert review to lower-case  public static int[] detectProsAndCons(String review, String[][] featureSet, String[] posOpinionWords,  String[] negOpinionWords) {  int[] featureOpinions = new int[featureSet.length]; // output    // your code ~ you will be invoking getOpinionOnFeature    return featureOpinions;  }  // First invoke checkForWasPhrasePattern and  // if it cannot find an opinion only then invoke checkForOpinionFirstPattern  private static int getOpinionOnFeature(String review, String feature, String[] posOpinionWords, String[] negOpinionWords) {    // your code    }  // Tip: Look at String API doc. Methods like indexOf, length, substring(beginIndex), startsWith can come into play  // Return 1 if positive opinion found, -1 for negative opinion, 0 for no opinion  // You can first look for positive opinion. If not found, only then you can look for negative opinion  private static int checkForWasPhrasePattern(String review, String feature, String[] posOpinionWords, String[] negOpinionWords) {  int opinion = 0;  String pattern = feature + " was ";  // your code  return opinion;  }    // You can first look for positive opinion. If not found, only then you can look for negative opinion  private static int checkForOpinionFirstPattern(String review, String feature, String[] posOpinionWords,  String[] negOpinionWords) {  // Extract sentences as feature might appear multiple times.  // split() takes a regular expression and "." is a special character  // for regular expression. So, escape it to make it work!!  String[] sentences = review.split("\\.");  int opinion = 0;    // your code for processing each sentence. You can return if opinion is found in a sentence (no need to process subsequent ones)  return opinion;  }  public static void main(String[] args) {  String review = "Haven't been here in years! Fantastic service and the food was delicious! Definetly will be a frequent flyer! Francisco was very attentive";    //String review = "Sorry OG, but you just lost some loyal customers. Horrible service, no smile or greeting just attitude. The breadsticks were stale and burnt, appetizer was cold and the food came out before the salad.";    String[][] featureSet = {  { "ambiance", "ambience", "atmosphere", "decor" },  { "dessert", "ice cream", "desert" },  { "food" },  { "soup" },  { "service", "management", "waiter", "waitress", "bartender", "staff", "server" } };  String[] posOpinionWords = { "good", "fantastic", "friendly", "great", "excellent", "amazing", "awesome",  "delicious" };  String[] negOpinionWords = { "slow", "bad", "horrible", "awful", "unprofessional", "poor" };  int[] featureOpinions = detectProsAndCons(review, featureSet, posOpinionWords, negOpinionWords);  System.out.println("Opinions on Features: " + Arrays.toString(featureOpinions));  }  } |

### My Solution:

|  |
| --- |
| **public** **class** SentimentAnalyzer {  // Tip: labeled continue can be used when iterating featureSet + convert review to lower-case  **public** **static** **int**[] detectProsAndCons(String review, String[][] featureSet, String[] posOpinionWords,  String[] negOpinionWords) {  **int**[] featureOpinions = **new** **int**[featureSet.length]; // output  System.***out***.println("Review: "+review);  System.***out***.println("-------------------");  // your code ~ you will be invoking getOpinionOnFeature  **for** (**int** i = 0; i < featureSet.length; i++) {  **for** (**int** j = 0; j < featureSet[i].length; j++ ) {  **if** (review.contains(featureSet[i][j])) {  System.***out***.println("The feature of \""+featureSet[i][j]+ "\" is contained in this review ");    featureOpinions[i]= *getOpinionOnFeature*(review,featureSet[i][j], posOpinionWords, negOpinionWords);  System.***out***.println("the feature: "+featureSet[i][j]+ " has an opinion of: " +featureOpinions[i]);  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }    }  }    **return** featureOpinions;  }  // First invoke checkForWasPhrasePattern and  // if it cannot find an opinion only then invoke checkForOpinionFirstPattern  **private** **static** **int** getOpinionOnFeature(String review, String feature, String[] posOpinionWords, String[] negOpinionWords) {  **int** wasPattern = *checkForWasPhrasePattern*(review, feature, posOpinionWords, negOpinionWords);  **if** (wasPattern==0) {  **return** *checkForOpinionFirstPattern*(review, feature, posOpinionWords, negOpinionWords);  } **else** {  **return** wasPattern;  }        }  // Tip: Look at String API doc. Methods like indexOf, length, substring(beginIndex), startsWith can come into play  // Return 1 if positive opinion found, -1 for negative opinion, 0 for no opinion  // You can first look for positive opinion. If not found, only then you can look for negative opinion  **private** **static** **int** checkForWasPhrasePattern(String review, String feature, String[] posOpinionWords, String[] negOpinionWords) {  review = review.replace('!', ' ');    String pattern = feature + " was ";    **int** ind = review.indexOf(feature);  **if** (ind == -1) {  **return** 0; // no opinion on feature  } **else** {    //looking for positive match  **for** (String positive : posOpinionWords) {  **if** (review.contains(pattern + positive ) ) {    System.***out***.println("found a positive pattern!");  **return** 1;  }  }    //looking for negative match  **for** (String negative : negOpinionWords) {  **if** (review.contains(pattern + negative ) ) {    System.***out***.println("found a negative pattern!: "+pattern + negative );  **return** -1;  }  }    System.***out***.println("found neither a positive or negative \"was\" pattern for the feature: "+feature);  **return** 0;    }    }    // You can first look for positive opinion. If not found, only then you can look for negative opinion  **private** **static** **int** checkForOpinionFirstPattern(String review, String feature, String[] posOpinionWords,  String[] negOpinionWords) {  // Extract sentences as feature might appear multiple times.  // split() takes a regular expression and "." is a special character  // for regular expression. So, escape it to make it work!!  String[] sentences = review.split("[.!]");    **for** (String sentence:sentences) {  **for** (String positive : posOpinionWords) {  **if** (sentence.toLowerCase().contains(positive+ " "+feature)) {  System.***out***.println("Positive opinion found!: "+positive + " "+feature );  **return** 1;  }  }  **for** (String negative: negOpinionWords) {  **if** (sentence.toLowerCase().contains(negative + " "+feature)) {  System.***out***.println("Negative opinion found!: "+negative + " "+feature );  **return** -1;  }  }    }  System.***out***.println("no opinions found :( on feature: "+feature);  **return** 0;    // your code for processing each sentence. You can return if opinion is found in a sentence (no need to process subsequent ones)    }  **public** **static** **void** main(String[] args) {  String review = "Haven't been here in years! Fantastic service and the food was delicious! Definetly will be a frequent flyer! Francisco was very attentive";    //String review = "Sorry OG, but you just lost some loyal customers. Horrible service, no smile or greeting just attitude. The breadsticks were stale and burnt, appetizer was cold and the food came out before the salad.";    String[][] featureSet = {  { "ambiance", "ambience", "atmosphere", "decor" },  { "dessert", "ice cream", "desert" },  { "food" },  { "soup" },  { "service", "management", "waiter", "waitress", "bartender", "staff", "server" } };  String[] posOpinionWords = { "good", "fantastic", "friendly", "great", "excellent", "amazing", "awesome",  "delicious" };  String[] negOpinionWords = { "slow", "bad", "horrible", "awful", "unprofessional", "poor" };    **int**[] featureOpinions = *detectProsAndCons*(review, featureSet, posOpinionWords, negOpinionWords);  System.***out***.println("Opinions on Features: " + Arrays.*toString*(featureOpinions));  }  } |

## Information Hiding

### Encapsulation

This is a language facilitation which lets us keep data and methods together.

### Example

|  |
| --- |
| **public** **class** Student {  // variable declarations  **public** **int** id;  **public** String name;  **public** String gender;    //method declarations  **public** Boolean updateProfile(String newName) {  name = newName;  **return** **true**;  }    } |

In the above class has its properties exposed , this can result in tight coupling

### Tight Coupling

Tight coupling means:

1. we can not enforce a range of our classes. E.g. the gender property could be changed to something non sensical
2. We also cant change data representation, i.e. we can not change gender from String to int

### Information Hiding

In public classes use accesor methods, NOT the fields directly:

|  |
| --- |
| **public** **class** Student {  // variable declarations  **private** String gender;    **public** **void** setGender(String gender) {  **if** (gender.equals("male")||gender.equals("female")){  **this**.gender = gender;  } **else** **throw** **new** IllegalArgumentException("only can set to male or female");    }    **public** String getGender() {  **return** gender;  }    } |

This resolves the issue of type invariance

|  |
| --- |
| **public** **class** Student {  // variable declarations  **private** String gender;  **private** **int** iGender;    **public** **void** setGender(String gender) {  **if** (gender=="male") { iGender = 1;}  **else** **if** (gender=="female") { iGender = 2; }    **if** (iGender == 0) {  **throw** **new** IllegalArgumentException("invalid gender passed");  } **else** {  **this**.gender = gender;  }    }    **public** **static** **void** main(String[] args) {  Student s = **new** Student();    s.setGender("bleh"); // throws exception  }  } |

this achieves **loose coupling**

### Loosely Coupled Systems

This helps us develop, test and optimise code in isolation.

## Minimise Accessibilty of Classes and Methods

We should design a minimal public API of your class

Make all other members private

If essential, make the members default

### Accessibility for Classes/Interfaces

If possible let it be default

If only one classes uses it, make it private and nested

# Section 7: More java.lang and Coding Conventions

## Chapter Introduction

We shall look at the Math package, boxed datatypes, autoboxing, final modifier, and coding conventions

## Initialisers: Static and Instances

### Static Initialiser

Initialision needs multiple lines

Which is useful for populating data

### Instant Initialiser

We can initialise an instance variable

## Final Variable

A variable whose value cannot be changed once declared can use the final keyword to achieve this

If the variable is assigned to the primitive, then the value remains constant

However, if its assigned to an object reference, that object CAN be changed

### Final Instance Variable

This would be constant for the life of the object

Therefore it must be initialised in declaration, intiliaser or constructor

### Final Local Variable

Constant for life of the block

### Final Static Variable

## Constant Variables

These are compile time constants, we do not have to wait at runtime to know its values

## Boxed Primnitives and Static Factory Methods

### Boxed Primitives

Boxed primitive is wrapped/boxed. Eqach of the primitive types have a wrapper class:

1. int ~ Integer
2. long ~ Long
3. short ~ Short
4. byte ~ Byte
5. boolean ~ Boolean
6. float ~ Float
7. double ~ Double

### Static Factor Methods

When we create instances of the wrapper classes, we can use static Factory methods – these yield some performanc efficiencies when used

In particular, the string over loaded methods for the number classes should NOT be used

### Declaring Boxed Primitives:

|  |
| --- |
| Integer boxedInt = Integer.*valueOf*(9); // static Factory  System.***out***.println(boxedInt); // 9  Boolean boxedBoolean = Boolean.***TRUE***;  System.***out***.println(boxedBoolean); // true    Character c = Character.*valueOf*('C');  System.***out***.println(c); // C    Double d = Double.*valueOf*(23.9); |

### Unboxing Wrapper Instances:

|  |
| --- |
| **int** primitiveInt = boxedInt.intValue();  **boolean** primitiveBool = boxedBoolean.booleanValue();  **char** primitiveC = c.charValue(); |

## Why Use Boxed Primitives?

The Wrapper Classes offer method overloads which can be really useful for processing and manipulating data

For example, we can parse strings to obtain numeric value:

|  |
| --- |
| **int** x = Integer.*parseInt*("11");  System.***out***.println(x); // 11 |

## Autoboxing

### Autoboxing

This would introduced in Java 5

This allows us to directly assign primitives to a wrapper class instance:

|  |
| --- |
| // autoboxing  Integer boxedInt = 23;  Integer boxedInt1 = **new** ~~Integer~~(21);    // auto unboxing  **int** unboxedInt = boxedInt;  **int** unbocedInt1 = boxedInt.intValue(); |

### Method Invocation

|  |
| --- |
| // invocation  ArrayList<Integer> list = **new** ArrayList<Integer>();  list.add(**new** ~~Integer~~(32)); // this is unneccesary  list.add(24); // perfectly valid |

## Prefer Primitivies to Boxed Primitives

If boxed primitives are used carelessly then issues can arrise

### Why?

Primitives are time and space efficient, autoboxing and auto unboxing will lead to performance overheads

## Code Exercise 5: Parsing API Response

Add an overloaded parse method for the below class, so that it can parse the book information from an XML response:

|  |
| --- |
| **public** **class** APIResponseParser {    /\*\*  \* Parses the input text and returns a Book instance containing  \* the parsed data.  \* **@param** response text to be parsed  \* **@return** Book instance containing parsed data  \*/  **public** **static** Book parse(String response) {  Book book = **new** Book();  String endRule = "<";    String startRule = "<title>";  String title = *parse*(response, startRule, endRule);  book.setTitle(title);    // Your code  **return** book;  }    // write overloaded parse method with the 3 parameters response, startRule, endRule    **public** **static** **void** main(String[] args) {  String response = "<work>" +  "<id type=\"integer\">2361393</id>" +  "<books\_count type=\"integer\">813</books\_count>" +  "<ratings\_count type=\"integer\">1,16,315</ratings\_count>" +  "<text\_reviews\_count type=\"integer\">3439</text\_reviews\_count>" +  "<original\_publication\_year type=\"integer\">1854</original\_publication\_year>" +  "<original\_publication\_month type=\"integer\" nil=\"true\"/>" +  "<original\_publication\_day type=\"integer\" nil=\"true\"/>" +  "<average\_rating>3.79</average\_rating>" +  "<best\_book type=\"Book\">" +  "<id type=\"integer\">16902</id>" +  "<title>Walden</title>" +  "<author>" +  "<id type=\"integer\">10264</id>" +  "<name>Henry David Thoreau</name>" +  "</author>" +  "<image\_url>" +  "http://images.gr-assets.com/books/1465675526m/16902.jpg" +  "</image\_url>" +  "<small\_image\_url>" +  "http://images.gr-assets.com/books/1465675526s/16902.jpg" +  "</small\_image\_url>" +  "</best\_book>" +  "</work>";    APIResponseParser.*parse*(response);  }  } |

### My Solution:

|  |
| --- |
| **public** **class** APIResponseParser {    /\*\*  \* Parses the input text and returns a Book instance containing  \* the parsed data.  \* **@param** response text to be parsed  \* **@return** Book instance containing parsed data  \*/  **public** **static** Book parse(String response) {  Book book = **new** Book();  String endRule = "<";    String startRule = "<title>";  String title = *parse*(response, startRule, endRule);  System.***out***.println("title: "+title);  book.setTitle(title);    String author = *parse*(response, "<id type=\"integer\">10264</id>" +  "<name>", endRule);  book.setAuthor(author);  **int** pubYear = Integer.*parseInt*(*parse*(response,"<original\_publication\_year type=\"integer\">", endRule));  book.setPublicationYear(pubYear);    **double** averageRating = Double.*parseDouble*(*parse*(response,"<average\_rating>", endRule ));  System.***out***.println("avgrating: "+averageRating);  book.setAverageRating(averageRating);    **int** ratingsCount = Integer.*parseInt*(*parse*(response, "<ratings\_count type=\"integer\">", endRule).replaceAll(",", "")) ;  book.setRatingsCount(ratingsCount);    String imageURL = *parse*(response, "<image\_url>",endRule);  book.setImageUrl(imageURL);    // Your code  **return** book;  }    // write overloaded parse method with the 3 parameters response, startRule, endRule    **private** **static** String parse(String response, String startRule, String endRule) {  **int** startIndex = response.indexOf(startRule)+startRule.length();    response = response.substring(startIndex);  **int** endIndex = response.indexOf(endRule);  ;  **return** response.substring(0, endIndex);    }  **public** **static** **void** main(String[] args) {    APIResponseParser.*parse*(response);  System.***out***.println(APIResponseParser.*parse*(response).toString());  }  } |

Result: Book [title=Walden, author=Henry David Thoreau, publicationYear=1854, averageRating=3.79, ratingsCount=116315, imageUrl=http://images.gr-assets.com/books/1465675526m/16902.jpg]

NOTE: My solution worked!

## Coding Conventions: Naming

### Naming Conventions

These suggestions come from the effective Java book

### Typographical – Packages

* Packages should only contain alphabetic characters
* Meaningful abbreviations
* Never start with java or javax
* Use organisations’s reverse internet domain

### **Typogrpahical** – Case

* Classes:
  + Use upper camel case
  + Methods and variables:
* Lower camel case
  + Static final variables should be all caps with underscores

### Typographical – Abbreviations

* Class, Methods and Fields
  + Abbreviations should be avoided
  + Acryonyms are fine
* Local variables: meaningful, single character names are fine

### Grammatical – Classes

* Singular nouns should be used. E.g. BufferedWriter

### Grammatical – Methods

* Performing action:
  + Use verbs, e.g. calculateDistance
  + Longer clear names are preferred
* Boolean return type:
  + “is” followed by noun or adjective
  + E.g. isDigit, isActive, setActive
* Non boolean attribute of object
  + Noun or noun phrase, e.g. gender, hashCode
  + getAttribute if there is a setter

### Special Methods

* Object type conversions:
  + **toType,** e.g. toString
* Static Factory Methods:
  + valueOf, getInstance, getType, newType

**Grammatical – fields**

* Boolean: usually adjectives, e.g. active
* Non-boolean: noun
* Data structures: singular or pluralised depending on structure,. E.g item vs list
* Name objects of same class by purpose. E.g. void sendMessage(User sender, user reciever){}

## Coding Conventions: Structuring Programs

### Class Organisation

Members should be ordered by the following

1. Variables – static first, instance later
2. Static initialiser
3. Static nested classes
4. Static methods
5. Instance Initialiser
6. Constructors
7. Instance nested classes
8. Methods

### Class Size:

Class size should abide the Single Responsibility Principle

Less than ~2000 lines

### Methods:

Small and focused, should be less than ~20 lines

Instead of having long logic within one method, the method should invoke other methods

Methods should be gruped by functionality

Cope of local variables should be minimised, i.e. declare a local variable just above where its first used. I.e. you wont lose track of variables declared at the top of a method

For Loops should be used over while loops, this limits the scope of the initialised variable

Coding Conventio