Java Full Stack

Java is a platform independent & object oriented programming language.

Softwares required

* JDK 8
* Eclipse IDE for Enterprise Java & Web developers

Datatypes in Java

Datatypes are set of keywords to represent the type of data a variable can store.

There are two types in datatypes.

1. Primitive types – byte, short, int, long, double, float, boolean, char
2. Derived types – classes, arrays, interface

Type Conversion: Converting from one type to another, since there are two types of conversion for each type of datatypes.

1. Primitive types
   1. Auto-Widening
   2. Explicit-Narrowing
2. Derived types
   1. Auto-Upcasting
   2. Explicit-Downcasting

Type Conversion

**package** com;

**public** **class** PrimitiveTypeConversion {

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

**byte** a = 10;

**int** b = 20;

**byte** c = (**byte**)b; // Explicit Narrowing

**int** d = a; // Auto-Widening

System.***out***.println("c = "+c);

System.***out***.println("d = "+d);

**int** e = 130;

**byte** f = (**byte**)e;

System.***out***.println("f = "+f);

}

}

Inheritance: Process of acquiring the properties & behaviours of an object from another object

Types of inheritance

1. Single level
2. Multi level
3. Hierarchical
4. Multiple (not supported through class, but possible through interface)

Person.java

**package** com;

**public** **class** Person {

String name;

**void** updateName() {

System.***out***.println("updateName() inside Person");

}

}

Student.java

**package** com;

**public** **class** Student **extends** Person {

String usn;

String grade;

**void** updateGrade() {

System.***out***.println("updateGrade() in Student");

}

}

Employee.java

**package** com;

**public** **class** Employee **extends** Person {

**int** id;

**double** salary;

**void** updateSalary() {

System.***out***.println("updateSalary() inside Employee");

}

}

TestInheritance.java

**package** com;

**public** **class** TestInheritance {

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

Person p1 = **new** Person();

p1.name = "Alex";

p1.updateName();

System.***out***.println("---------------------");

Employee e1 = **new** Employee();

e1.id = 100;

e1.name = "Bruce";

e1.salary = 35000;

e1.updateName();

e1.updateSalary();

System.***out***.println("-------------------");

Student s1 = **new** Student();

s1.name = "Charles";

s1.usn = "1ABC001";

s1.grade = "A+";

s1.updateName();

s1.updateGrade();

}

}

Note: Above program is to show that a subclass object can access super class members.

Access Modifiers

There are 4 access modifiers in java

1. Private: within the class
2. Package scope: within the package
3. Protected: within package & outside package but only to the subclass
4. Public: everywhere

OOPS features

There are four important features

1. Encapsulation: Hiding the data and accessing them through public setters & getters
2. Inheritance: Acquiring properties & behaviours of an object from another object
3. Polymorphism: Ability of a method to perform more than one task
4. Abstraction: Hiding the complexity and showing the necessary details to the user.

Employee.java

**package** com;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **double** salary;

**public** Employee() {

}

**public** Employee(**int** id, String name, **double** salary) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** **int** getId() {

**return** id;

}

}

TestEncapsulation.java

**package** com;

**public** **class** TestEncapsulation {

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

Employee e1 = **new** Employee(100, "Alex", 35200);

Employee e2 = **new** Employee(200, "Bruce", 20000);

System.***out***.println("Id = "+e1.getId()+", Name = "+e1.getName()+", Salary = "+e1.getSalary());

System.***out***.println("Id = "+e2.getId()+", Name = "+e2.getName()+", Salary = "+e2.getSalary());

System.***out***.println("----------------------------------------");

e1.setName("Alexandar");

e2.setName("Brook");

System.***out***.println("Id = "+e1.getId()+", Name = "+e1.getName()+", Salary = "+e1.getSalary());

System.***out***.println("Id = "+e2.getId()+", Name = "+e2.getName()+", Salary = "+e2.getSalary());

}

}

Inheritance: In inheritance every subclass constructor calls default constructor of its parent class by default, but you can use super(args) to call parameterized constructor of the parent class.

Note: If a class is not extending any class then it automatically extends Object class

**package** com;

**class** A {

A() {

System.***out***.println("A() constructor");

}

}

**class** B **extends** A {

B() {

System.***out***.println("B() constructor");

}

}

**class** C **extends** B {

C() {

System.***out***.println("C() constructor");

}

}

**public** **class** TestInheritance {

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

A a1 = **new** A();

System.***out***.println("-----------------");

B b1 = **new** B();

System.***out***.println("-----------------");

}

}

Polymorphism

A method with many forms (or) an action has many forms

* Compile time polymorphism = method overloading = happens in the same class
* Runtime polymorphism = method overriding = happens in the subclass

Method overloading: method name will be same but signature will be different like number of parameters, type of parameters

Method overriding: method name & its signature will be same but will have different logics in the sub-class.

Abstraction: Hiding the complexity and showing the necessary details to the user.

Exception Handling

Exceptions are run time errors which should be handled else the program will be abnormally terminated.

Hence you can handle the exceptions using some of the mechanism in exception handling i.e,

try, catch, finally, throw & throws.

try block: Write those statements that would cause an exception

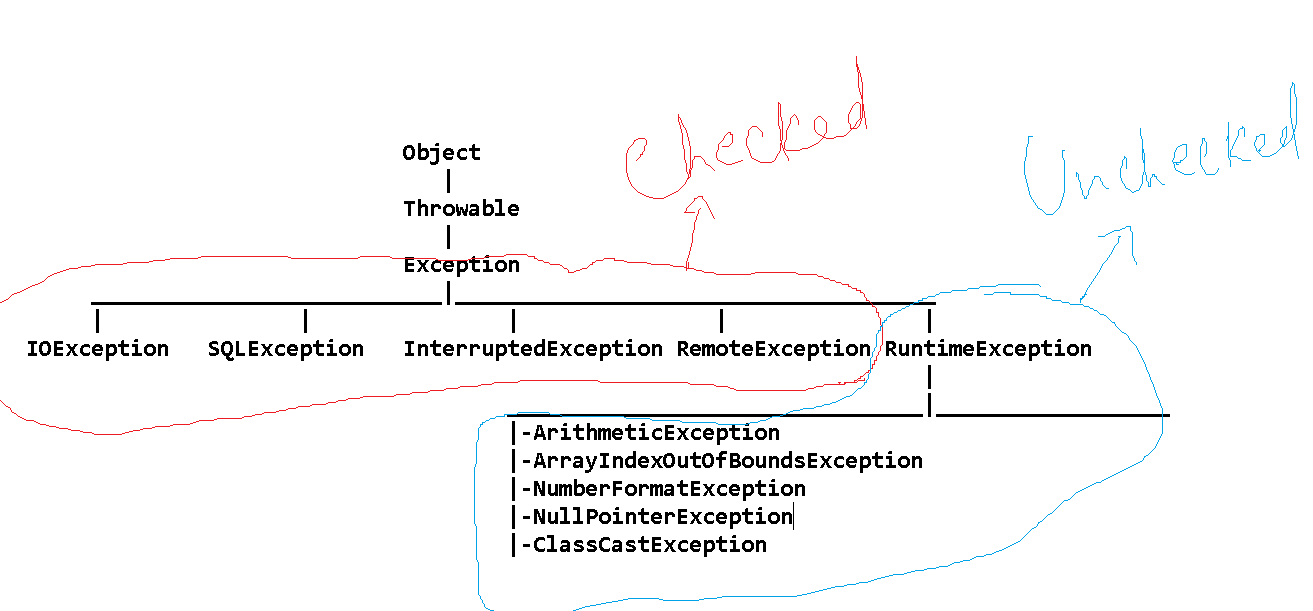
catch block: Write exception handling logic

finally block: It is an optional block, but it will be mandatorily executed for both normal & abnormal termination

throw: It is used manually generate the exception

ex: 10/0 -> exception created by JRE

ex: throw new exception\_name(); it is manually created exception



Checked Exception: These exceptions are checked at compilation time, it has to be handled, it wouldn’t cause any problem at runtime

Unchecked Exception: These exceptions are not identified at compilation time, it need not to be handled but it will cause problem at runtime.

Array:

It is a container that can store multiple values of same type in a single variable.

You can create arrays for primitives as well for derived types.

Ex: int[] arr = {1, 2, 3, 6, 4, 5};

Ex: int[] arr = new int[5];

arr[0] = 20; arr[1] = 30; … arr[4] = 45;

Ex: Employee[] e = new Employee[5]; ‘e’ is array of employee that can store 5 employees

Drawbacks in array

* Array size is fixed
* Array can only store same type of data
* Manipulating the array is difficult when you have different types requirements, because you need to write the program manually

Ex: Sorting, Searching, Remove, Update, maintaining unique items and so on.

Collection Framework

Collection is a framework which is used to maintain multiple items of same/different types & it is dynamic, it will provide its own algorithm using which you can manipulate the data easily without writing any extra algorithm

i.e, Sorting, Searching, Marinating Uniqueness, Removing the items, increasing the size and so on.

Collection provides an interface that has common methods to maintain the data like add, remove, clear, size and so on.

Collection has 3 sub-interfaces

1. List
2. Set
3. Queue

List: It is an interface that maintains the data in sequential order & it allows duplicates, it has 3 implementations

1. ArrayList – stores elements in contiguous memory address, retrieval is faster but addition and deletion is a bit slower
2. LinkedList – stores elements in non-contiguous memory address, retrieval is a bit slower but adding and removing the items is faster
3. Vector – it is similar to array list however it is thread safe and legacy class

Employee.java

**package** com;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **double** salary;

**public** Employee() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Employee(**int** id, String name, **double** salary) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

}

TestList.java

**package** com;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** TestList {

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

// a list which will have strings

List<String> listStrings = **new** ArrayList<String>();

// a list which will have employees

List<Employee> listEmployees = **new** ArrayList<Employee>();

listStrings.add("hello");

listStrings.add("demo");

listStrings.add("hello");

listStrings.add("welcome");

**for**(String s : listStrings) {

System.***out***.println(s);

}

Employee e1 = **new** Employee(1, "Alex", 35000);

Employee e2 = **new** Employee(2, "Bruce", 42000);

listEmployees.add(e1);

listEmployees.add(e2);

listEmployees.add(e2);

**for**(Employee e : listEmployees) {

System.***out***.println("id = "+e.getId()+", name = "+e.getName()+", salary = "+e.getSalary());

}

}

}

Set: It allows only unique items

It has 3 implementations

1. HashSet: It maintains elements in random order but retrieval is faster
2. LinkedHashSet: It maintains elements in insertion order
3. TreeSet: It maintains elements in sorted order

TestSet.java

**package** com;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** TestSet {

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

Set<Integer> set = **new** HashSet<Integer>();

set.add(20);

set.add(20);

set.add(10);

set.add(50);

set.add(60);

System.***out***.println(set);

Employee e1 = **new** Employee(1, "Alex", 35000);

Employee e2 = **new** Employee(2, "Bruce", 42000);

Employee e3 = **new** Employee(3, "Charles", 70000);

Set<Employee> setEmployee = **new** HashSet<Employee>();

setEmployee.add(e1);

setEmployee.add(e1);

setEmployee.add(e1);

setEmployee.add(e1);

System.***out***.println("Size: "+setEmployee.size());

System.***out***.println(setEmployee);

}

}

Queue: It is used to perform FIFO operations, it mainly operates on the item after removing, there are 2 implementations in Queue

1. LinkedList: Removes elements in FIFO order
2. PriorityQueue: Removes elements based on the priority i.e., sorted order

In both the case you should use poll() method to remove

**package** com;

**import** java.util.LinkedList;

**import** java.util.PriorityQueue;

**import** java.util.Queue;

**public** **class** TestQueue {

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

Queue<Integer> queue = **new** LinkedList<Integer>();

// instead of LinkedList we are using PriorityQueue

queue = **new** PriorityQueue<Integer>();

queue.add(50);

queue.add(10);

queue.add(30);

queue.add(20);

queue.add(40);

System.***out***.println(queue);

// FIFO or Sorted Order based on the implementation

queue.poll();

System.***out***.println(queue);

}

}

Map: It is stores the items in key & value pairts, it has 3 implementations

1. HashMap: stores in random order
2. TreeMap: stores in sorted order
3. LinkedHashMap: stores in insertion order

Map is an interface with following methods

* put(key, value): used to store
* get(key): used to retrieve
* remove(key): used to remove

**package** com;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**public** **class** TestMap {

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

Map<String, String> simpleMap = **new** HashMap<String, String>();

simpleMap.put("username", "user1234");

simpleMap.put("password", "pa$$w0rd");

System.***out***.println("Username: "+simpleMap.get("username"));

System.***out***.println("Password: "+simpleMap.get("password"));

Map<String, List<Employee>> complexMap = **new** HashMap<String, List<Employee>>();

List<Employee> accountsList = **new** ArrayList<Employee>();

// add some employees to accounts

accountsList.add(**new** Employee(1, "Alex", 35200));

List<Employee> salesList = **new** ArrayList<Employee>();

salesList.add(**new** Employee(2, "Bruce", 35200));

salesList.add(**new** Employee(3, "Charles", 45200));

// add some employees to sales

// add these lists to the map

complexMap.put("account", accountsList);

complexMap.put("sales", salesList);

List<Employee> list = complexMap.get("account");;

System.***out***.println(list.size());

}

}

String:

It creates immutable objects, it can’t be modified once created, it is class with many methods

toUpperCase(), toLowerCase(), equals(), concat() and so on.

**package** com;

**public** **class** TestStrings {

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

String s1 = "hello";

String s2 = s1.toUpperCase();

String s3 = s2.concat("1234");

System.***out***.println(s2); // HELLO

System.***out***.println(s1); // hello

System.***out***.println(s3); // HELLO1234

}

}

For Mutable string object you can use either StringBuffer/StringBuilder

StringBuilder & StringBuffer operation wise both are same, but StringBuffer is older & synchronized whereas StringBuilder is newer & non-synchronized.

Both has same methods like

append(), insert(), delete() and so on

Regular Expression:

It is for pattern matching, it is mainly used in find and replace operations, email validation operations

It has 3 important API’s

Pattern: gives the set of characters that has to be searched

Matcher: will have the text that needs to be searched for a pattern

PatternException: Generated when pattern is invalid.

Some of the regular expression pattern

[abc]: to match a, b or c

[a-z]: to match all the letters from a to z

[A-Z]: to match all the letters from A to Z

[0-9]: to match all the digits

[\\d]: same as [0-9]

[\\w]: a word that has digits & alphabets

Some of the quantifiers

\*: 0 or more

+: 1 or more

?: 0 or once

Suppose we have a text like

Hey! veryvery good veryveryverybad very sad

TO match only vowels

[aeiou]: gets e & a

(very): gets a word matching very

(very){1,2}: gets a word repeating 1 to 2 times

[^ \\w]: except space, digits & alphabets

**package** com;

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** TestRegularExpression {

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

// to find only alphabets in lowercase

Pattern pattern = Pattern.*compile*("[a-z]");

// to find uppercase & lowercase letters

pattern = Pattern.*compile*("[A-Za-z]");

Matcher matcher = pattern.matcher("Hello Pa$$w0rd");

**while**(matcher.find()) {

System.***out***.println(matcher.group());

}

}

}

**package** com;

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**public** **class** TestRegularExpression {

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

// to find only alphabets in lowercase

Pattern pattern = Pattern.*compile*("[a-z]");

// to find uppercase & lowercase letters

//pattern = Pattern.compile("[A-Za-z]");

// pattern that matches hello

//pattern = Pattern.compile("(hello)");

// pattern that matches hello which is repeating minimum 2 times max 3 times

pattern = Pattern.*compile*("(hello){2,3}");

Matcher matcher = pattern.matcher("Hello Pa$$w0rd hello everyone hellohello everybody hellohellohello welcome");

**while**(matcher.find()) {

System.***out***.println(matcher.group());

}

}

}

Inner class

They can be written within a class or a method

An inner class inside the method is also called as local inner class

Problem Statement in Lesson 2

Enter input from keyboard and valid if it’s a valid email id

Hint: Use Scanner class to take input from keyboard

Submit the solution in the LMS