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

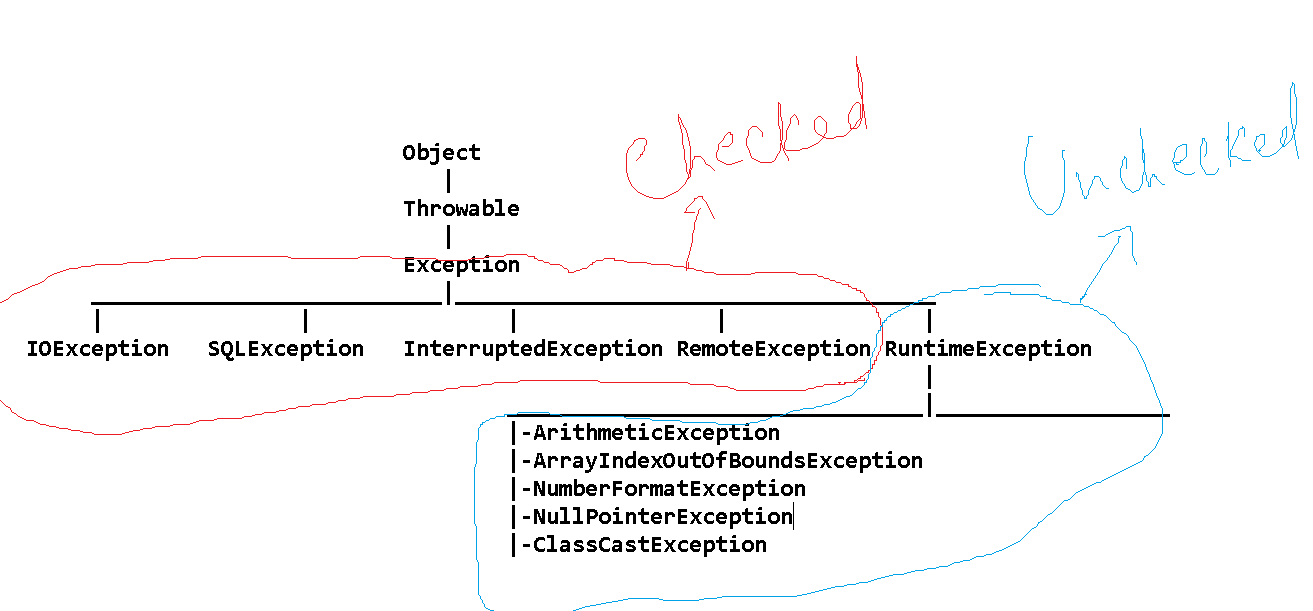
catch block: Write exception handling logic

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