Java Questions

Java Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the programs instructions.

The exception object contains info about the errir, including its type and the state of the program when the error occurred.

Types of exceptions:

The Checked Exception: exceptiosn that the java applicationc should be able to cope with.

Sample FileNotFoundException→ The compiler knows the FileReader constructor can throw a FileNotFoundException and it´s up to the calling code to handle this exception.

public static void main(String[] args) throws FileNotFoundException{

FileReader fileInput = null;

//Open the input file

fileInput = new FileReader("Untitled.txt");

}

Or

public static void main(String[] args){

FileReader fileInput = null;

try {

//Open the input

file fileInput = new FileReader("Untitled.txt");

} catch(FileNotFoundException ex) {

//tell the user to go and find the file

}

}

Errors

The exception obkject derives from throwable class, this class has two main subclasses -Error and Excpetion. The Error class denotes an exception this exception is considered rare, for example JVM might run out of resources due to the hardware not being able to cope witrh all processes it is having to deal with. The application has a posibility to catch the error to notify the user but tipically the app is going to close until the problem is dealt with.

Runtimne Exceptions

A runtime exception occurs simply because the programmer has made a mistake. Accesing an element of an array that doesn´t exist or a logic error caused a mehtod to be called with a null value. Or any mistakes a programmer can make.

Erros and Runtime Exceptions fall into the category of unchecked exceptions.

Loose Coupling

In computing and systems design a loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. Sub-areas include the coupling of classes, interfaces, datam and services.

Ipods are a good example of tight coupling: once the battery dies you might as well buy a new iPod because the battery is soldered fixed and won´t come loose, thus making replacing very expensive. A loosely coupled player would allow effortlessly changing the battery. The same 1:1 goes for software development.

Consider a simple shopping cart application that uses a CartContents class to keep track of the items in the shopping cart and an Order class for processing a purchase. The Order needs to determine the total value of the contents in the cart, it might do that like so:

Tightly Coupled Example:

public class CartEntry

{

public float Price;

public int Quantity;

}

public class CartContents

{

public CartEntry[] items;

}

public class Order

{

private CartContents cart;

private float salesTax;

public Order(CartContents cart, float salesTax)

{

this.cart = cart;

this.salesTax = salesTax;

}

public float OrderTotal()

{

float cartTotal = 0;

for (int i = 0; i < cart.items.Length; i++)

{

cartTotal += cart.items[i].Price \* cart.items[i].Quantity;

}

cartTotal += cartTotal\*salesTax;

return cartTotal;

}

}

Notice how the Order Total method (and thus the Order class) depends on the implementation details ot the CartContents and the CarEntry classes. If we were to try to change this logic to allow for discounts, we´d likely have to change all 3 classes. Also, if we change to using a List collection to keep track of the items we´d have to change the Order class as well.

Now here's a slightly better way to do the same thing.

Less Coupled Example:

public class CartEntry

{

public float Price;

public int Quantity;

public float GetLineItemTotal()

{

return Price \* Quantity;

}

}

public class CartContents

{

public CartEntry[] items;

public float GetCartItemsTotal()

{

float cartTotal = 0;

foreach (CartEntry item in items)

{

cartTotal += item.GetLineItemTotal();

}

return cartTotal;

}

}

public class Order

{

private CartContents cart;

private float salesTax;

public Order(CartContents cart, float salesTax)

{

this.cart = cart;

this.salesTax = salesTax;

}

public float OrderTotal()

{

return cart.GetCartItemsTotal() \* (1.0f + salesTax);

}

}

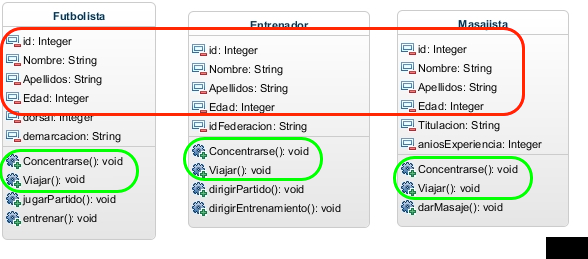
The logic that is specific to the implementation of the cart line item or the cart collection or the order is restricted to just that class. So we could change the implementation of any of these classes without having to change the other classes. We could take this decoupling yet further by improving the design, introducing interfaces, etc, but I think you see the point.

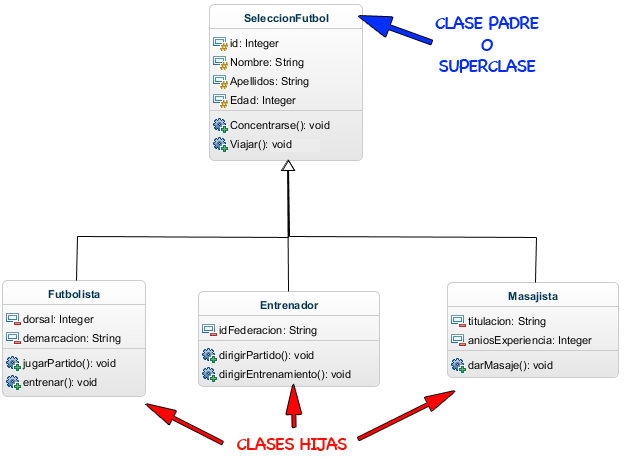
**Four OOP Pilars**

There are four pilars for OOP Oriented Object programming Inheritance, Polymorphism, Abstraction, Encapsulation.

**Inheritance**: is a mechanism that allows the definition of a class using the definition of another one that exist. Inheritance allows us to share automatically methods and data between classes, subclasses and objects. So let´s say the inheritance is something like a “dynamic Copy-Paste” or a way to obtain the “common factor” from the code tha we wrote.

Note: in this digrams the constructors and the getter and setter methods are not shown, alhotught in a good class diagram thye must appear to be consistent with the encapsulation principle of the OOP.



In java we use three reserved words:

- extends : shows a child class which is going to be its father class.

public class Futbolista extends SeleccionFutbol

Here we are saying to the Futbolista class that his father is SeleccionFutbol.

- protected: when we use this its makes an attribute visible only from a child class and not from any other.

- super: we use this to call the father class constructor

The private attributes from a father class are not visible from its children classes.

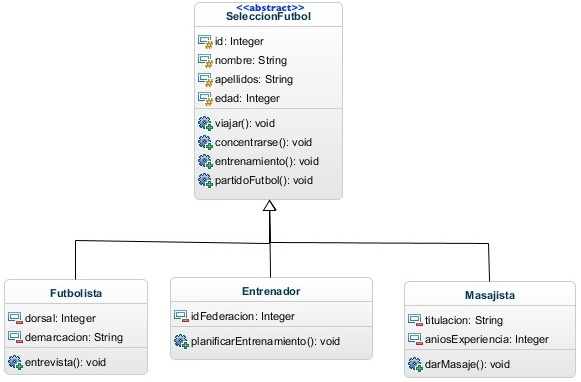
More references here:

<http://jarroba.com/herencia-en-la-programacion-orientada-a-objetos-ejemplo-en-java/>

**Polimorphism**

It is a word from greek roots tha means “many forms”. So we use this term to “refer to a property in which it's possible to sent equal sintactic messages to distinct object types ”

Note: in this digrams the constructors and the getter and setter methods are not shown, alhotught in a good class diagram they must appear to be consistent with the encapsulation principle of the OOP.



public abstract class SeleccionFutbol {

protected int id;

protected String nombre;

protected String apellidos;

protected int edad;

*// constructores, getter y setter*

public void viajar() {

System.out.println("Viajar (Clase Padre)");

}

public void concentrarse() {

System.out.println("Concentrarse (Clase Padre)");

}

*// IMPORTANTE -> METODO ABSTRACTO => no se implementa en la clase abstracta pero si en la clases hijas*

public abstract void entrenamiento();

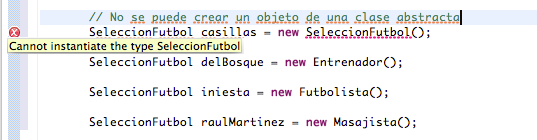
public void partidoFutbol() {

System.out.println("Asiste al Partido de Fútbol (Clase Padre)");

}

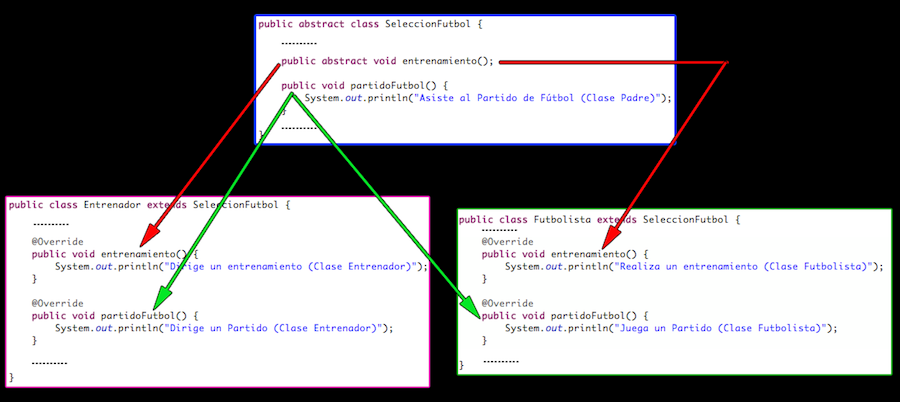
}

Here we add a new reserved word abstract, so the SeleccionFutbol it's an abstract class and it cannot be instantiated so we can never make a new SeleccionFutbol(). And when we used this word in a method it means that all the children of the this class must implement this method in a mandatory way.



In inheritance and polimorphism the children classes can inherit methods (or their immplementation) and so they can specialize that methods, this means they can redefine the methods of its father class, so we can rewrite this method and so specialize it.

Here the children classes implements in a mandatory way the abstract method from the father class, we can see this in the method entrenamiento(). We can see another label “@Override” this means that the childen class redefines this method that exist too in tha father class, this is on the abstract method entrenamiento() and others like partidoFutbol();



public class Futbolista extends SeleccionFutbol {

private int dorsal;

private String demarcacion;

*// constructor, getter y setter*

@Override

public void entrenamiento() {

System.out.println("Realiza un entrenamiento (Clase Futbolista)");

}

@Override

public void partidoFutbol() {

System.out.println("Juega un Partido (Clase Futbolista)");

}

public void entrevista() {

System.out.println("Da una Entrevista");

}

}

public class Entrenador extends SeleccionFutbol {

private int idFederacion;

*// constructor, getter y setter*

@Override

public void entrenamiento() {

System.out.println("Dirige un entrenamiento (Clase Entrenador)");

}

@Override

public void partidoFutbol() {

System.out.println("Dirige un Partido (Clase Entrenador)");

}

public void planificarEntrenamiento() {

System.out.println("Planificar un Entrenamiento");

}

}

Let's see the next code :

public static ArrayList<SeleccionFutbol> integrantes = new ArrayList<SeleccionFutbol>();

SeleccionFutbol delBosque = new Entrenador(1, "Vicente", "Del Bosque", 60, 28489);

SeleccionFutbol iniesta = new Futbolista(2, "Andres", "Iniesta", 29, 6, "Interior Derecho");

integrantes.add(delBosque);

integrantes.add(iniesta);

//Concentracion

System.out.println(“Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método”);

for (SeleccionFutbol integrante : integrantes){

System.out.print(integrante.getNombre() + “” + integrante.getApellidos());

integrante.concentrarse();

}

System.out.println(“Todos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método”);

for (SeleccionFutbol integrante: integrantes){

System.out.print(integrante.getNombre()+ “ ”integrante.getApellidos());

integrante.viajar();

}

Here we have the execution of this code, the children classes calls the father's methods.

Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)

Vicente Del Bosque -> Concentrarse (Clase Padre)

Andres Iniesta -> Concentrarse (Clase Padre)

Todos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)

Vicente Del Bosque -> Viajar (Clase Padre)

Andres Iniesta -> Viajar (Clase Padre)

Let's see how the objects act with the calling of “entrenamiento” and “partidoFutbol”

*// ENTRENAMIENTO*

System.out.println("nEntrenamiento: Todos los integrantes tienen su función en un entrenamiento (Especialización)");

for (SeleccionFutbol integrante : integrantes) {

System.out.print(integrante.getNombre() + " " + integrante.getApellidos() + " -> ");

integrante.entrenamiento();

}

*// PARTIDO DE FUTBOL*

System.out.println("nPartido de Fútbol: Todos los integrantes tienen su función en un partido (Especialización)");

for (SeleccionFutbol integrante : integrantes) {

System.out.print(integrante.getNombre() + " " + integrante.getApellidos() + " -> ");

integrante.partidoFutbol(););

}

This is the result of this code

Entrenamiento: Todos los integrantes tienen su función en un entrenamiento (Especialización)

Vicente Del Bosque -> Dirige un entrenamiento (Clase Entrenador)

Andres Iniesta -> Realiza un entrenamiento (Clase Futbolista)

Partido de Fútbol: Todos los integrantes tienen su función en un partido (Especialización)

Vicente Del Bosque -> Dirige un Partido (Clase Entrenador)

Andres Iniesta -> Juega un Partido (Clase Futbolista)

In this results we see that the classes call the specialized methods instead of the fathers methods.

Finally we can execute the unique methods that every object has, the method planificarEntrenamiento() can only be called from the class Entrenador, and the method entrevista() can only be called from the Futbolista class.

*// PLANIFICAR ENTRENAMIENTO*

System.out.println("nPlanificar Entrenamiento: Solo el entrenador tiene el método para planificar un entrenamiento:");

System.out.print(delBosque.getNombre() + " " + delBosque.getApellidos() + " -> ");

((Entrenador) delBosque).planificarEntrenamiento();

*// ENTREVISTA*

System.out.println("nEntrevista: Solo el futbolista tiene el método para dar una entrevista:");

System.out.print(iniesta.getNombre() + " " + iniesta.getApellidos() + " -> ");

((Futbolista) iniesta).entrevista();

So this are the results of the execution

Planificar Entrenamiento: Solo el entrenador tiene el método para planificar un entrenamiento:

Vicente Del Bosque -> Planificar un Entrenamiento

Entrevista: Solo el futbolista tiene el método para dar una entrevista:

Andres Iniesta -> Da una Entrevista

Polimorphism is an advanced concept and this could be useful to put hierachy and to give a common behaviour pattern to many objects that inherit from the same class.

Reference here :

<http://jarroba.com/polimorfismo-en-java-parte-i-con-ejemplos/>

Polimorphidsm with interfaces

The interface concept goes beyond the concept of an abstract class, and bastract class is a class that it's not able to be instantiated (create an object of that class) but it's able to define attributes and implement methods inside of this class so its children classes can use them. So an interface is a pure asbtract class in which its methods are abstracts thereby they can not be implemented in the interface class. On a first thought this might sound unusefull because why do I need a class that cannot implement methods and that its children classes must implement that methods. So we use the interfaces to define the form that a class must have.

On the other hand we cannot define attributes unless they will be statics o constants so they use the static o final reserved word.

Lets see the following example:

public interface Parlanchin {

public abstract void habla();

}

So we make that a class called Animal implements Parlanchin and so we inherit Animal in Perro and Gato. Let´s see the class hierachy.

public abstract class Animal implements Parlanchin{

public abstract void habla();

}

class Perro extends Animal{

public void habla(){

System.out.println("¡Guau!");

}

}

class Gato extends Animal{

public void habla(){

System.out.println("¡Miau!");

}

Let`s see another class hierarchy the one that inherits from la class Reloj. The class Cucu implements the interface Parlanchin thereby it must define mandatorily the function habla() declared in that interface.

public abstract class Reloj {

}

class Cucu extends Reloj implements Parlanchin{

public void habla(){

System.out.println("¡Cucu, cucu, ..!");

}

}

Let´s Define the function hazleHablar() so this will aknowledge the object that is passed not using a class but an interface, the interface Parlanchin. To this function we can pass any object that implements the interface Parlanchin, even if it is not in the hierarchy clases

public class PoliApp {

public static void main(String[] args) {

Gato gato=new Gato();

hazleHablar(gato);

Cucu cucu=new Cucu();

hazleHablar(cucu);

}

static void hazleHablar(Parlanchin sujeto){

sujeto.habla();

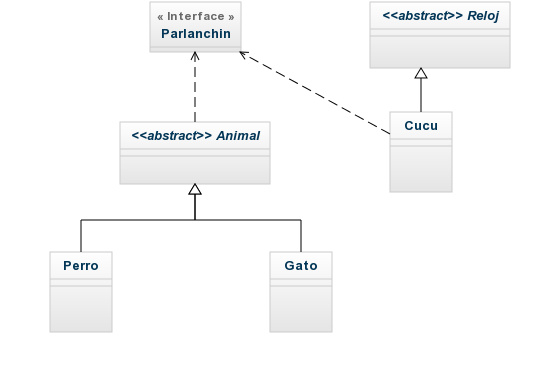
}

}

This is the output of the above code:

¡Miau!

¡Cucu, cucu, ..!



Here with the interface we can obligate that the Cucu class implements the habla() method, if we wouldn't have the interfaces we must make the Cucu class a child from Animal which is not logical. Or we couldnt pass the cucu object to hazleHablar() method.

With interfaces any object from any family can implement the interface Parlanchin and so it can be passed to the hazleHablar() function, for this reason the interfaces provides more polimorphism than the one that we obtain with simple inheritance and its classes hierarchy.

Two links for this:

http://jarroba.com/polimorfismo-en-java-interface-parte-ii-con-ejemplos/

http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/interfaces.htm

OPPS concepts

Asociation in java

Association establish a relationship between two classes through their objects. The relationship can be one to one, one to many, many to one and many to many.

Let's see the following example:

class Student {

Set<Transcript> transcripts;

}

class Transcript {

Student student;

Course course;

Date subscriptionDate;

}

class Course {

Set<Transcript> transcripts;

}

Agregation in java

Is an special form of association that is directional it means is a one way asociation. It represents a Has-a relationship. It means that a class is part of another class (weak composition). These are two independent objects.

Let´s say we have the class watch has-a set of clock-hands

final class Watch {

private ClockHands clockHands;

void setclockHands(ClockHands clockHands) {

this.clockHands = clockHands;

}

void move() {

if (clockHands != null)

clockHands.work();

}

}

From the point of view of the manufacturer if there is an error or a wrong perfomance of the clock hands he can change them.

links:

http://beginnersbook.com/2013/05/aggregation/

Composition in java

It´s a restricted form of aggregation in which two entities (or classes) are higly dependent on each other. For e.g. Human and Heart. A human needs a heart to live and a heart needs a humabn body to survive. In other words when the classes (entities) are dependent on each other and their life span are the same (if one dies then the other one too) then it´s composition. Heart class has no sense if Human class is not present.

Let`s see the forward example:

final class Watch {

private final ClockHands clockHands;

Car(ClockHandsSpecs specs) {

clockHands = new ClockHands(specs);

}

void move() {

engine.work();

}

}

From the point of view of a customer if there is an error with the clockhands the watch doesn't work anymore, so the watch can not work properly without the clock hands and the clock hands will be useless without the watch.

Links:

<http://stackoverflow.com/questions/11881552/implementation-difference-between-aggregation-and-composition-in-java>

<http://beginnersbook.com/2013/05/association/>

JVM

Java Virtual Machine

1. **There is no manual way of doing garbage collection in Java. But you can use System.gc() that calls Runtime.gc, but this merely suggest the JVM to run GC.**
2. Java Heap is divided into three generation for sake of garbage collection. These are young generation (nursery ), tenured or old generation and Perm (permanent generation) area.
3. New objects are created into young generation and subsequently promoted (moved) to old generation.
4. String pool is created in Perm area of Heap, garbage collection can occur in perm space but depends upon JVM to JVM, in the perm area are stored the class definitions and related data.
5. Young Generation is divided into multiple spaces – Eden and Survivor-. Minor garbage collection is used to move object from Eden space to Survivor 1 and Survivor 2 space and Major collection is used to move object from young to tenured generation.
6. Whenever Major garbage collection occurs application threads stops during that period which will reduce application’s performance and throughput.
7. There are few performance improvement has been applied in garbage collection in java 6 and we usually use JRE 1.6.20 for running our application.
8. JVM command line options –Xmx and -Xms is used to setup starting and max size for Java Heap. Ideal ratio of this parameter is either 1:1 or 1:1.5 based upon my experience for example you can have either both –Xmx and –Xms as 1GB or –Xms 1.2 GB and 1.8 GB.

Addition

In oracle's JVM the permanent generation is not part of the heap. In java 6 and earlier, interned strings were also stored in the permanent generation. In Java 7 and 8, interned strings are stored in the main object heap.

For JAVA 8 there is the complete removal of the Permanent Generation (PermGen) and now it uses somnething called Metaspace, now the allocations for class metadata are now allocated out of native memory.

Links:

<http://java-performance.info/string-intern-in-java-6-7-8/>

<http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html>

<http://stackoverflow.com/questions/1481178/how-to-force-garbage-collection-in-java>

<http://stackoverflow.com/questions/13855013/understanding-java-memory-management>

http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

<https://dzone.com/articles/java-8-permgen-metaspace>

<https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html>