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);

}

}

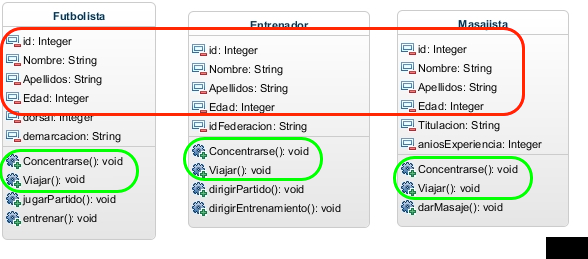
The logic that is specific to the implementation of the cart line item or the cart collection or the order is restricted to just that class. So we could change the implementation of any of these classes without having to change the other classes. We could take this decoupling yet further by improving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

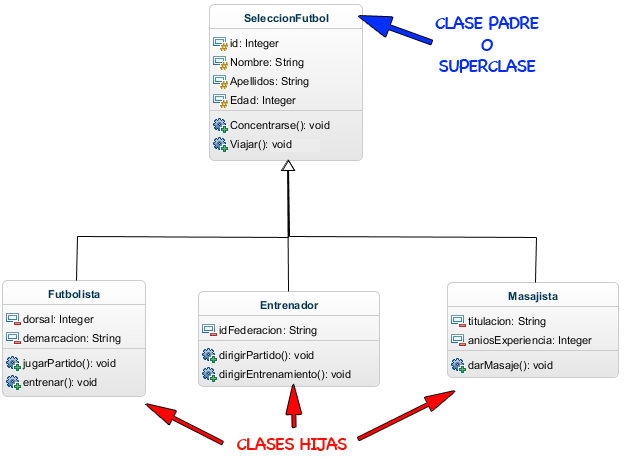
**Four OOP Pilars**

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.

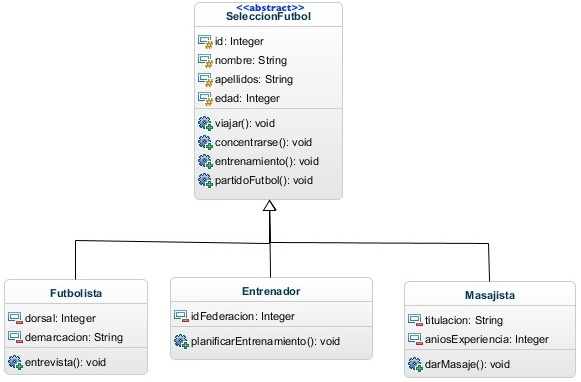
More references here:

<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();

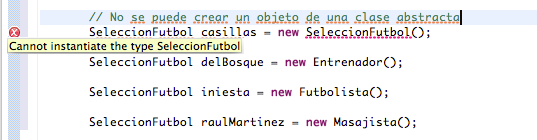
public void partidoFutbol() {

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

}

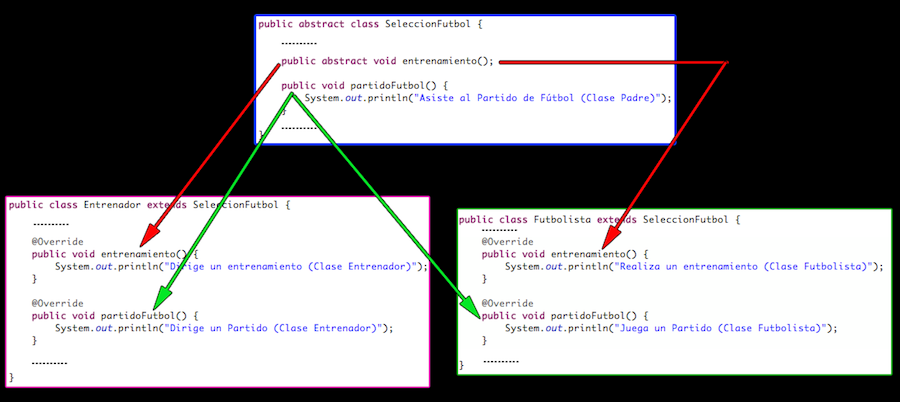
}

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