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.

There are four pilars for OOP Oriented Object programming