**Gestionarea Exceptiilor**

* Exceptiile permit sa anticipam erorile care ar putea aparea in timpul utilizarii codului**.**
* Exceptiile permit sa gestionam o eroare in alt loc decat cel in care ea a aparut.
* Gestionarea exceptiilor permit programului sa ruleze mai departe chiar si cand apare o exceptie.
* Gestioanarea exceptiilor cu if nu e o idee buna, deoarece, daca de ex apare o eroare intr-o functie ce trebuie sa faca o instructiune simpla si rapida, si daca se produce de ex, impartirea la 0, functia ceea trebuie sa se ocupe de gestionarea acestei probleme, si asta nu e rolul ei.
* Gestionarea exceptiilor permite ca exceptiile sa fie gestionate intr-un anumit loc, indifferent ca exceptiile apar in functii apelate sau altundeva. Oricum obiectul de tip Exception e mereu aruncat catre blocul ce a chemat instructiunea ce genereaza o exceptie.
* Clasa Exception mosteneste de la Throwable,ce are:
* doi constructori

Throwable()

Throwable(String message)

* Doua metode:

String getMessage() – returneaza doar mesajul setat

Void prinStackTrace() – afiseaza mesajul setat in terminal

* Cand apare o exceptie, se creaza un obiect care contine toate datele despre exceptie, si se trimite la codul care l-a cauzat.

Exceptiile sunt de 2 tipuri:

1.User-defined exceptions

2.Built-in exceptions(prezente in librariile Java):

- checked(verificate cand scriem codul, la compile-time )

- unchecked(verificate cand se executa programul,runtime)

**Checked Exceptions**

* Aceste exceptii sunt verificate chiar cand scriem codul. Fara ca sa executam codul, vom vedea ca compilatorul ne arata ca ceva nu e bine in cod, si nici nu ne va lasa sa il rulam. Acest tip de exceptii sunt cerute atunci cand se creaza obiecte de anumite clase, ca FileReader, FileInputStream etc.
* Trebuie neaparat sa cream un bloc try catch pentru ele,sau sa aruncam exceptia cu throws si sa o lasam pe mana compilatorului(daca e metoda main), deorece in momentul crearii unui obiect de acest tip(FileInputStream etc.), compilatorul ne va obliga sa tratam exceptia de tip FileNotFoundException , sau o dam pe mana la alte metode, desi Java incurajeaza sa le gestionam in metoda unde apar.

**Unchecked Exceptions**

* Aceste exceptii apar doar atunci cand rulam programul.
* Cand apare exceptia intr-o metoda, are loc urmatorul plan:

1.metoda creaza un obiect de tip Exception.

2.Compilatorul creaza o lista de metode care au apelat-o pe cea care a dat o exceptie.

3. Obiectul de tip Exception incepe a cauta de la metoda in care a aparut pana la cea care a dus la apelarea ei vreun bloc try-catch care sa o gestioneze.

4.Daca un asemenea bloc nu se gaseste, exceptia e tratata de compilator, si programul se opreste.

* Daca Java gaseste vreo exceptie, e ca si cum ea ar scrie:

Throw new TypeException(“Message”);

in locul nostru acolo unde exceptia apare. Ea anume asa si creaza obiectele de tip exceptie

**Try**

* Blocul try, la fel ca while,for si do while, are propriul bloc, si deci nu putem accesa din programul principal variabilele declarate acolo.
* Este circular, adica daca el apeleaza o metoda ce apeleaza si ea o metoda si tot asa, daca niciuna din metode nu gestioneaza o exceptia aparatua, ea se va intoarce la fiecare metoda apelanta si va ajunge deci in cele din urma la blocul try
* Nu poate exista fara catch, desi daca nu exista niciun catch, trebuie sa existe un finally macar.
* Odata ce in try se gaseste o exceptie, instructiunile de mai departe din el nu se vor mai executa niciodata!

**Catch**

* Ca si try, are un bloc propriu
* Gestioneaza diferite exceptii prinse in try
* Nu poate exista fara try
* Blocurile catch trebuie ordonate dupa gradul de generalitate. Intai se pun blocurile catch mai specifice, apoi mai generale, si ultima se pune un bloc catch ce trateaza o exceptie generala de tip Exception. Acest lucru are loc

din cauza ca odata ce se arunca o exceptie la rulare, ea se va gestiona in primul bloc catch gasit ce o poate gestiona, restul fiind ignorate, chiar daca mai jos si altele ar putea sa o gestioneze.E principiu de la downcasting, cand obiectul de tip parinte poate fi egalat cu un obiect de tip copil, caci oricum gaseste in el blocul sau de care are nevoie, dar invers acest lucru nu se face automat! Java nici nu ne va permite sa plasam catch mai generale in fata celor mai particulare!

try{  
 throw new Exception("error");  
}  
catch (Exception e){  
 System.*out*.println(e.getMessage());  
}  
~~catch(ArithmeticException a){~~  
   
}

* Daca se arunca un obiect ce nu este propriu zis gestionat de un catch, dar este gestionata de un catch cu un obiect de tip parinte al sau, acel bloc catch il va prinde oricum.

try{  
 throw new ArithmeticException("error");  
}  
catch(Exception a){  
  
}

* Putem avea si bloc multiCatch:

**catch(Type1 | Type2 | Type3 | … name )**

**Parametrii din catch trebuie aranjati in ordinea generlitatii, de la cele mai concrete la cele generale. In caz contrar, vom avea o eroare!**

**Throw**

* El creaza manual o exceptie si o arunca catre codul sursa.
* Putem arunca doar obiecte de tip Throwabale
* Daca folosim throw, compilatorul deja va verifica daca exista vreun bloc care sa il si prinda. Daca utilizam throw intr-o metoda fara un bloc try si catch, compilatorul va cauta oricum in celelalte metode ce au apelat-o daca nu se gestioneaza exceptia.Daca e intr-un bloc try, si nu gaseste un catch potrivit, iar se va cauta in alte metode. Insa acest lucru nu e valabil si pentru checked exceptions!Nu se va cauta vreo metoda anterioara care sa le gestioneze pana metoda nu va avea scris dupa declarare **throws**!Compilatorul poate cauta in alte metode daca metoda aruncata cu throw e gestionata fara a avea throws,doar daca e vorba de unchecked exceptions . **Throws Exceptie** va asigura ca exceptia data e gestionata de o metoda ce o apeleaza, si deci va permite compilatorului sa o trimite deja ei. Asa Java incurajeaza gestionarea la checked exception in metoda unde apar.
* public static void main(String[] args) {  
   try {  
   *func*();  
   }  
   catch (FileNotFoundException e){  
   }  
   }  
   public static void func() throws FileNotFoundException{  
   throw new FileNotFoundException();  
   }  
  }
* **Throawble se comporta ca o checked exception, desi nu e.**

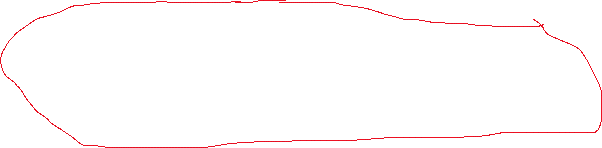
public static void main(String[] args){  
 *f*(-1);  
 *f*(10);  
 *f*(20000);  
}  
static void f(int a) {  
 try {  
 if (a < 0) {  
 throw new Exception("Negative number");  
 } else if (a > 10000) {  
 throw new Throwable("Too big number");  
 } else {  
 System.*out*.println("The provided number is OK!");  
 }  
 } catch (Exception e) {  
 System.*out*.println(e.getMessage());  
 }  
}

* **Daca dam throw la o exceptie intr-un bloc catch, exceptia aruncata nu se va cauta in urmatoarele blocuri catch din acest bloc try, dar acel catch va trebui sa aiba si el sa aiba un bloc try/catch sau primul try sa aiba mai multe blocuri try si deci catch in el. Catchurile din acelasi bloc try nu arunca exceptiile una la alta.**
* public static void main(String[] args) {  
   try {  
   throw new ArithmeticException("ERROR1");  
   } catch (ArithmeticException e) {  
   throw new IndexOutOfBoundsException("ERROR2");  
   } catch (IndexOutOfBoundsException e) {  
   System.*out*.println("Nu se va ajunge in acest bloc catch!");  
   e.printStackTrace();  
   }  
  }

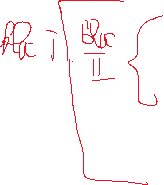
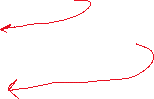


Solutii:

public static void main(String[] args){  
 try{  
 throw new ArithmeticException("ERROR1");  
 }  
 catch (ArithmeticException e){  
 try{  
 throw new IndexOutOfBoundsException("ERROR2");  
 }  
 catch (IndexOutOfBoundsException e2){  
 System.*out*.println("Aici se va ajunge!");  
 e2.printStackTrace();  
 }  
 }  
 catch (IndexOutOfBoundsException e){  
 System.*out*.println("Nu se va ajunge in acest bloc catch!");  
 e.printStackTrace();  
 }  
}



public static void main(String[] args){  
 try{  
 try{  
 throw new ArithmeticException("ERROR1");  
 }  
 catch (ArithmeticException e){  
 throw new IndexOutOfBoundsException("ERROR2");  
 }  
 }  
 catch (IndexOutOfBoundsException e){  
 System.*out*.println("Se va ajunge in acest bloc catch!");  
 e.printStackTrace();  
 }  
}



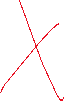
**Throws**

* **throws** are rolul de a obliga metoda1 ce a apelat metoda2, dupa care se scrie throws, sa gestioneze exceptia indicata de throws.Poate obliga sa gestionee doar checked Exception! Daca scriem dupa throws o unchecked exception, nu se va intampla nimic.Putem asa doar informa cei ce vor vedea codul ca exceptia e trata altudneva.

public static void main(String[] args)  
{  
 *func*();  
}  
public static void func() throws ArithmeticException{  
 // nici-o eroare  
}



public static void main(String[] args)  
{  
 *func*();  
}  
public static void func() throws FileNotFoundException{  
 // EROARE  
}



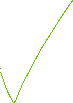
public static void main(String[] args)  
{  
 try {  
 *func*();  
 }  
 catch (FileNotFoundException e){  
 }  
}  
public static void func() throws FileNotFoundException{  
   
}



* **throw** poate arunca atat checked cat si unchecked exceptions. Totusi, modul in care compilatorul gestioneaza cele 2 tipuri de exceptii e diferit.
* **Unchecked Exceptions** – dupa ce throw arunca o unchecked exception, fie ca o facem manual, fie ca o face programul in momentul executiei, exceptia va fi aruncata catre urmatorul catch ce va putea sa o gestioneze. Daca in metoda nu e nici-un catch, si in genere s-a dat throw nu intr-un bloc try, programul va cauta un bloc catch corespunzator in metoda ce a apelat-o pe aceasta, si daca nu gaseste nici in ea, se duce in alta metoda ce a apelat-o si tot asa. Daca se ajunge la metoda main, si tot nu se gestioneaza exceptia, atunci compilatorul se ocupa de ea, arata mesajul, insa opreste programul.Acelasi principiu este si cand ea se arunca intr-un bloc try si nu se gaseste un catch potrivit. Tot se vor cauta in metodele din urma.
* **Checked Exceptions –** acestea se trateaza putin diferit. Checked exceptions sunt o grupa separata de exceptii.Daca throw arunca o checked exception, si de nu e gasita in metoda, ea nu va fi cautata automat in metoda anterioara pana nu se va pune **throws Exceptie** in metoda, ca sa se arate ca metoda anterioara e fortata sa gestioneze exceptia. Asa se si incurajeaza gestionarea lor in metoda unde apar.
* Anumite clase,la instantiere, ne obliga a gestionam anumite exceptii,si deci cand am crea o instantiere a acelei clase, ca **FileInputStream();** ,compilatorul ne va obliga sa gestionam o exceptie de tip **FileNotFoundException**. Daca nu le-am gestiona, ele ar ajunge automat la compilator, si ar opri tot programul, dar Java considera ca acestea nu sunt exceptii ce trebuie sa opreasca tot programul, de aceea mai bine sa le gestionam noi. Daca metoda nu gestioneaza exceptia data, compilatorul nu o va cauta in metodele anterioare pana nu vom utiliza throws, pentru a arata ca am fortat metoda anterioara sa o gestioneze si nu am uitat de ea.
* public static void main(String[] args)  
  {  
   try{  
   *func*();  
   }  
   catch (FileNotFoundException f){  
     
   }  
  }  
  public static void func() {  
   FileInputStream file = new FileInputStream("aa");  
  }



public static void main(String[] args)  
{  
 try{  
 *func*();  
 }  
 catch (FileNotFoundException f){  
  
 }  
}  
public static void func() throws FileNotFoundException{  
 FileInputStream file = new FileInputStream("aa");  
}



public static void main(String[] args)  
{  
 *func*();  
}  
public static void func(){  
 throw new ArrayIndexOutOfBoundsException();  
}



public static void main(String[] args)  
{  
 try {  
 *func*();  
 }  
 catch (FileNotFoundException f){  
   
}  
}  
public static void func(){  
 throw new FileNotFoundException();  
}



* Metoda main tot poate folosi throws, si atunci compilatorul se va ocupa de gestionarea exceptiei. Insa, daca se arunca o asemenea exceptie, compilatorul o va gestiona si tot programul se va opri.

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

System.*out*.println("Aici nu se ajunge");  
}  
public static void func() throws FileNotFoundException{  
 FileInputStream file = new FileInputStream("aaaa");  
 System.*out*.println("Aici nu se ajunge");  
}

**Finally**

* E optional si se pune mereu dupa ultimul catch.
* Un bloc try poate exista fara catch doar daca exista finally
* El se executa mereu dupa ce try si catch termina treaba, ca se gaseste vreo exceptie sau nu
* **Chiar daca intr-un bloc catch se gaseste return,sau throw, finally tot mereu va fi executat. Daca metoda vede un return sau throw intr-un bloc catch, sau try, mereu se va verifica daca nu exista un finally, si daca exista, intai se va executa el, si abea apoi se va face return/throw. Finally e important, deoarece el inchide fisiere, procese etc. si nu trebuie in niciun caz de trecut pest el!**
* public static void main(String[] args){  
   System.*out*.println(*func*());  
  }  
    
  static boolean func(){  
   try{  
   throw new Exception("ERROR");  
   }  
   catch (Exception e){  
   System.*out*.println(e.getMessage());  
   return false;  
   }  
   finally {  
   System.*out*.println("Se executa inaintea lui return!");  
   }  
  }

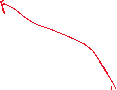
**Crearea Exceptiilor personalizate**

* Putem crea propriile exceptii prin mostenirea unei clase a claselor Exception, Throwable, RuntimeException si orice copii de ale lor.
* Mostenirea lui Throwable si Exception ar crea checked exceptions, deci vor trebui mereu tratate ca checked exceptions cand se vor arunca cu throw sau se vor crea obiecte. Aruncarea lor nu ar fi posibila fara a le gestiona, asa cum si checked exceptions le mostenesc, si deci nu ar fi clar cum am vrea sa le folosim, ca checked ori unchecked, de aceea Java alege sa le trateze ca checked exceptions.
* Mostenirea lui RuntimeException creaza unchecked exceptions, ce nu trebuie tratate neaparat.
* Unele compilatoare ar putea sa nu dea eroare daca nu gestionam exceptiile checked create de noi, doar avertisment, dar altele vor da eroare.

**public static void main(String[] args) {  
 throw new MyException();  
 }  
}  
 class MyException extends Exception{  
 }**



public static void main(String[] args) {  
 MyException my = new MyException();  
 }  
}  
 class MyException extends Exception{  
 }



public static void main(String[] args) {  
 MyException my = new MyException();  
 }  
}  
 class MyException extends RuntimeException{  
 }

