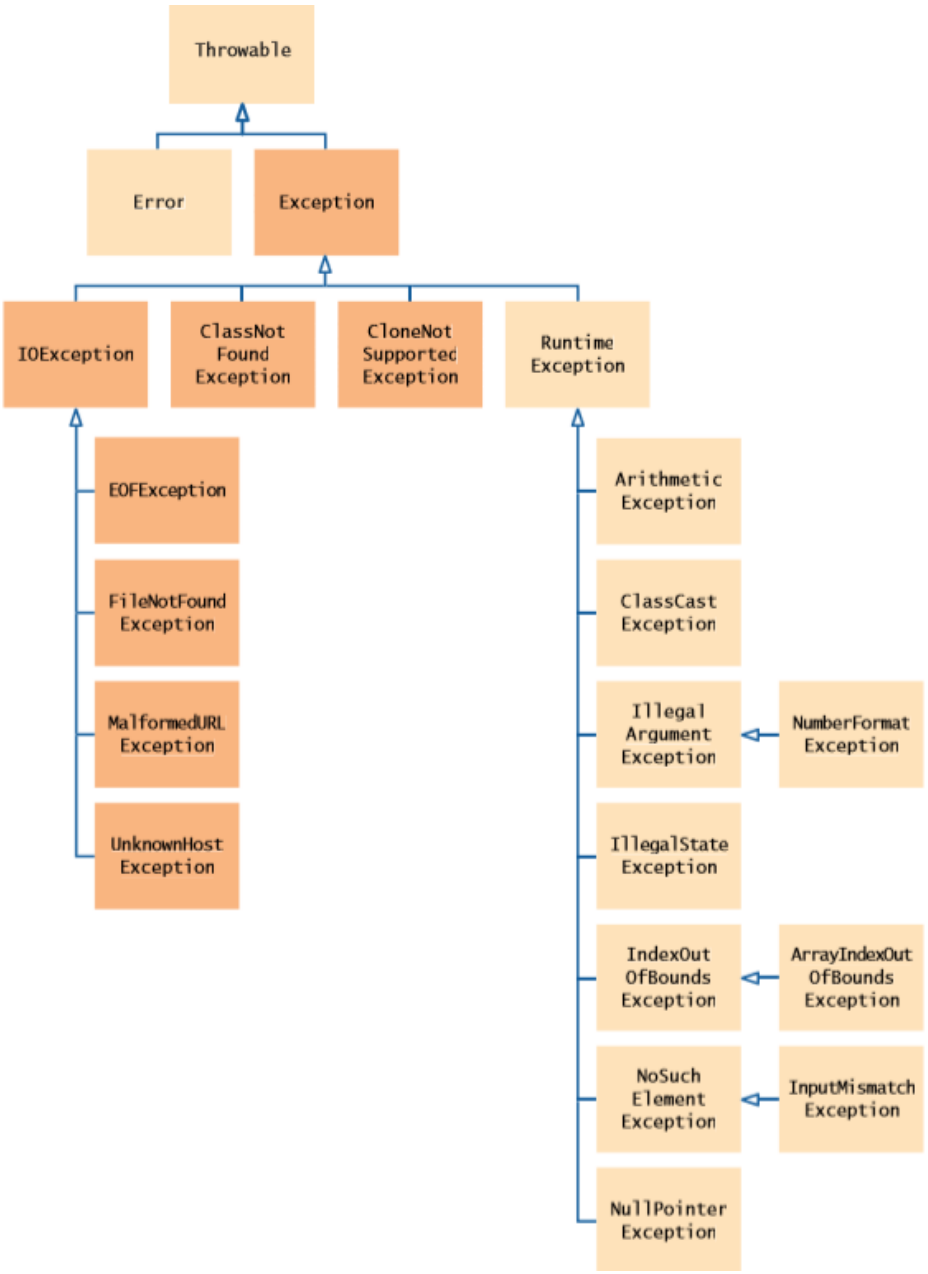
Programlama Dili Prensipleri

Lab Notları – 12

Hata Yakalama

Programlama dillerinde sıra dışı durumlar ile ilgilenme yöntemi 2 türlüdür. Raporlama (reporting), Kurtarma (recovery). Hata yakalamadaki en büyük problem, raporlamanın, kurtarmadan çok farklı bir yerde olmasıdır. Örneğin bir liste yapısında Find (bul) metodu aranan elemanı bulma noktasında raporlama olarak elemanın olmadığını bildirebilir. Ama bu noktada ne yapacağını bilemez yani kurtarma kısmı muallaktır. Java programlama dili, hatanın raporlanması ile kurtarma arasında esnek bir mekanizma sağlar. Yapılan işlem, bir hata durumu tespit edildiğinde sıra dışı durumu temsil eden bir nesne fırlatılır. Bu nesnenin hangi sınıftan türetilmesi gerektiğini bilmek için Java'nın hata sınıfları diyagramını bilmek gerekir. Bu diyagram aşağıda görülmektedir.



Örneğin aşağıda bir Hesap sınıfı ve ParaÇek metodu görülmektedir. Bu metot parametre olarak aldığı miktarı hesaptan çeker. Peki ya hesaptaki paradan daha büyük bir miktar çekilmek istenirse ne olacaktır. Burada if kontrolü koyup şartları sağlıyorsa parayı çek demek yeterli olamamaktadır. Çünkü bu sınıfı kullanan kişi bir tepki beklemektedir. Bu metodun birşey döndürmediğini de düşünürsek olumsuz tepki vermek pek mümkün görünmemektedir. Fakat aşağıdaki gibi hatanın fırlatılması en doğru programlama tasarımıdır.

```
public class Hesap {
   private double para;
   public Hesap(){
      para=0;
   }
   public void ParaYatir(double miktar){
      para += miktar;
   }
   public void ParaCek(double miktar){
      if(miktar > para){
         throw new IllegalArgumentException("Yeterli bakiye yok");
      }
      para = para + miktar;
   }
}
```

Yukarıdaki kod bloğunda Hesap sınıfını yazan kişi görevini yerine getirmiş ve istenmeyen bir durumun oluşmasında hatayı fırlatmıştır. Hatanın fırlatılması durumunda miktarın para eklendiği satır çalıştırılmayacak ve hatanın fırlatıldığı noktadan çıkış yapılacaktır. Fakat bu sınıfı kullanacak kişi eğer sadece aşağıdaki gibi kodu kullanırsa hata fırlatılacak fakat yakalanmadığı için program çökecektir.

```
public static void main(String[] args) {
      // TODO code application logic here
      Hesap h = new Hesap();
      h.ParaCek(100);
   }
```

```
Exception in thread "main" java.lang.IllegalArgumentException: Yeterli bakiye yok
        at aaab.Hesap.ParaCek(Hesap.java:23)
        at aaab.AAAB.main(AAAB.java:21)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Aslında yukarıdaki durum kontrol edilmeyen hata fırlatmasından kaynaklanmaktadır. Java'da istenmeyen durumlar iki kategoride incelenir. **Kontrol edilen (checked)** ve **Kontrol edilmeyen (unchecked) durumlar.** Eğer bir hata fırlatma kontrol edilen kategoride ise derleyici programcıyı try, catch blokları içinde kullanmaya zorlar Bu hata fırlatmasını görmezden gelemezsiniz. Yukarıdaki hata çeşidi kontrol edilmeyen olduğu için derleyici programcıyı try, catch blokları için zorlamaz ve eğer try cathc'e konulmaz ve hata durumu oluşursa program çökecektir. Kontrol edilen durumlar genelde dosya işlemleri gibi hayati önem taşıyabilecek durumlarda vardır.

Yazmış olduğunuz hata fırlatan kendi metotlarınızda aşağıdaki gibi yazmanız. Kullanacak programcıyı bilgilendirmek açısından önemlidir.

```
public void ParaCek(double miktar) throws IllegalArgumentException {
   ...
}
```

Önemli: Java'da Eğer metodunuz kontrol edilen bir hata fırlatıyorsa yukarıdaki gibi yazmak zorundasınız.

C tarafına bakıldığında, C programlama dilinde hata fırlatma ve yakalama deyimleri desteklenmez fakat buna benzer yapıyı gerçekleştirmek için jumper yapıları sunar bunlar setjmp.h kütüphanesi içerisinde bulunur. Temel mantık istenmeyen bir durumu önceden sezip programcının oraya jumper değerini değiştirecek bir kod

koymasıdır. Dolayısıyla fonksiyon çalışacağı sırada değer kontrol edilirse istisnai durumun oluşması engellenip programın sonlanmasına izin verilmez.

```
#include "stdio.h"
#include "setjmp.h"
jmp_buf jumper;
int Bol(int x,int y){
        if(y == 0)
                longjmp(jumper, -3);
        return x/y;
}
int main(){
        int a=10, b=0;
        if(setjmp(jumper) == 0){
                printf("%d",Bol(a,b));
        }
        else{
                printf("Sifira Bolunme Hatasi");
        }
        return 0;
}
```

try-catch Blokları JAVA

Programın çökmesini engellemek için sadece hatanın fırlatılması yeterli değildir. Aynı zamanda hatanın yakalanması gerekir. Bu durumda devreye try-catch blokları girer. try bloğu normal yapılmak istenen işlemlerin yazıldığı blok iken catch bloğu hata oluşması durumunda içine girilecek olan bloktur. Bir try bloğu içinde farklı hata sınıfları içeriyorsa farklı sayıda catch bloğu kullanılabilir. Aşağıdaki örneği inceleyelim.

```
...
public void ParaYatir(double miktar) throws ArithmeticException{
    if(miktar <= 0){
       throw new ArithmeticException("Çekilecek miktar sıfırdan büyük olmaldır.");
    }
    para += miktar;
  }
....
```

```
public static void main(String[] args) {
    // TODO code application logic here
    Hesap h = new Hesap();
    try{
       h.ParaCek(100);
       h.ParaYatir(-1);
    }
    catch(IllegalArgumentException ex){
       System.out.println(ex.getMessage());
    }
    catch(ArithmeticException ex){
       System.out.println(ex.getMessage());
    }
  }
```

```
// Yada tek catch bloğunda birleştirilebilir.
public static void main(String[] args) {
    // TODO code application logic here
    Hesap h = new Hesap();
    try{
       h.ParaCek(100);
       h.ParaYatir(-1);
```

```
    }
    catch(IllegalArgumentException | ArithmeticException ex){
       System.out.println(ex.getMessage());
    }
  }
```

Java'daki aynı kodun C dilinde gerçeklenmesi

```
#ifndef DOSYA_H
#define DOSYA_H

#include "stdio.h"
#include "stdlib.h"
#include "setjmp.h"

struct DOSYA{
        char *yol;
        char *icerik;
        jmp_buf jumper;
        char* (*Oku)(struct DOSYA*);
        void (*Yoket)(struct DOSYA*);
};
typedef struct DOSYA* Dosya;

Dosya DosyaOlustur(char*);
char* TumIcerikOku(const Dosya);
void DosyaYoket(Dosya);
#endif
```

```
#include "Dosya.h"

Dosya DosyaOlustur(char *yol){
        Dosya this;
        this = (Dosya)malloc(sizeof(struct DOSYA));
        this->yol=yol;
        this->icerik=NULL;
        this->Oku = &TumIcerikOku;
        this->Yoket=&DosyaYoket;
        return this;
}

char* TumIcerikOku(const Dosya d){
  char *icerik = NULL;
   int boyut = 0;
   FILE *fp;

   fp = fopen(d->yol, "r");
   if(!fp)longjmp(d->jumper,-3);
   fseek(fp, 0, SEEK_END);
   boyut = ftell(fp);
   rewind(fp);

   icerik = (char*) malloc(sizeof(char) * boyut);
   fread(icerik, 1, boyut, fp);
   fclose(fp);

   d->icerik = icerik;
   return icerik;
}

void DosyaYoket(Dosya d){
```

```
        if(d == NULL)return;
        if(d->icerik != NULL)free(d->icerik);
        free(d);
        d=NULL;
}
```

```
#include "Dosya.h"

int main(int argc, char *argv[]){
        Dosya d;
        d=DosyaOlustur("D:\\dene.txt");
        if(setjmp(d->jumper)==0){
                printf("%s",d->Oku(d));
        }
        else printf("Dosya okuma hatasi\n");
        d->Yoket(d);
        return 0;
}
```

Önemli: Bir hata oluşma durumunda ona varsayılan bir değer atamak yerine hatanın fırlatılması en doğru yoldur. Örneğin dosyadan bir sayı okuma örneğinde eğer dosyada sayı yoksa değere sıfır vermek hatalı işlemlere neden olabileceği için doğru değildir. Orada hatanın fırlatılması gerekir.

finally Bloğu

Bazı durumlarda (hatanın oluşsun ya da oluşmasın) özel bir işlem yapmak istenebilir. Böyle bir durumda finally bloğu kullanılmalıdır. Hata oluşsun ya da oluşmasın bu blok çalışacaktır. Bu bloğa şöyle bir senaryoda ihtiyaç duyulabilir. try içerisinde fırlatılan hata catch içerisinde yakalanıp işleyiş devam ederken catch içerisinde tekrar istenmeyen durum oluşmuş ve hata fırlatılmıştır.

Java'da Bazı durumlarda (hatanın oluşması ya da oluşmaması durumlarında) özel bir işlem yapmak isteyebilirsiniz. Böyle bir durumda finally bloğu kullanılmalıdır. Hata oluşsun ya da oluşmasın bu blok çalışacaktır. Aşağıdaki kodu inceleyelim. Burada bakiye ekrana yazdırılmak isteniyor fakat catch bloğu içerisinde tekrar hata fırlatıldığı için yazdırılamıyor.

```
public class Hesap {
   private double para;
   public Hesap(){
      para=0;
   }
   public void ParaYatir(double miktar){
      if(miktar <= 0){
         throw new ArithmeticException("Çekilecek miktar sıfırdan büyük olmaldır.");
      }
      para += miktar;
   }
   public void ParaCek(double miktar){
      if(miktar > para){
         throw new IllegalArgumentException("Yeterli bakiye yok");
      }
      para = para + miktar;
   }
   @Override
   public String toString() {
      return String.valueOf(para);
   }
}
```

```
public static void main(String[] args) {
      Hesap h = new Hesap();
```

```
    try{
      try{
        h.ParaCek(100);
      }
      catch(IllegalArgumentException ex){
        System.out.println(ex.getMessage());
        h.ParaYatir(-1);
      }
      System.out.println(h);
    }
    catch(ArithmeticException ex){
      System.out.println(ex.getMessage());
    }
  }
```

Eğer bakiyenin yazdırıldığı (koyu renkli satır) finally bloğu içerisine alınsaydı hata fırlatılsın ya da fırlatılmasın ekrana yazdırılacaktı.

```
…
finally{
   System.out.println(h);
}
…
```

Önemli: finally bloğunun kullanıldığı kod örneklerine bakıldığında daha çok dosya kapama veri tabanı bağlantısı kesme gibi işlemlerde kullanıldığı görülür. C++'ta finally blokları yoktur. Buradaki amaç sorumluluğun kullanıcıdan tasarımcıya verilmiş olmasıdır. Java programlama dilinde otomatik çöp toplayıcı sistemi vardır dolayısıyla açılmış olan bir kaynak boşa düştüğünde ne zaman çöp toplayıcı tarafından kapatılacağı bilinemez. Burada kod olarak bunu kapatabilecek bir yapıya ihtiyaç vardır bu da finally bloğudur. C++'ta böyle bir sistem olmadığı için finally bloğuna da ihtiyaç yoktur.

Kendi Exception (Hata) Sınıfını Tasarlamak JAVA

Bazen Java'nın sağladığı hata sınıfları sizin hatanızı ifade etmede yetersiz kalabilir. Bu durumda kendi hata sınıfınızı yazıp bu hata sınıfından bir nesne fırlatmanız gerekecektir. Yine yukarıdaki örnek ile devam edersek, YetersizBakiye isminde bir hata sınıfı yazılabilir. Bu hata sınıfını kontrol edilen ya da edilmeyen olarak tasarlayabilirsiniz. Burada karar verilmesi gereken yazacağınız sınıfın hangi Java hata sınıfından kalıtım alacağıdır. Aşağıdaki örneği inceleyelim. Örneğin aşağıda hata sınıfımız kontrol edilen bir hata sınıfı olan IOException sınıfından kalıtım almaktadır. Bu durumda bu hatayı fırlatan her metot try bloğu içerisinde kullanılmalıdır.

```
public class YetersizBakiye extends IOException {
  public YetersizBakiye() { }
  public YetersizBakiye(String hataMesaji){
    super(hataMesaji);
  }
}
```

```
...
public void ParaCek(double miktar) throws YetersizBakiye{
    if(miktar > para){
      throw new YetersizBakiye("Yeterli bakiye yok");
    }
    para = para + miktar;
  }
...
```

```
public static void main(String[] args) {
    // TODO code application logic here
    Hesap h = new Hesap();
```

```
    try{
      h.ParaCek(100);
    }
    catch(YetersizBakiye ex){
      System.out.println(ex.getMessage());
    }
  }
```

Burada kontrol edilen ya da edilmeyen sınıflardan kalıtım alırken dikkat edilmesi gerekmektedir. Çünkü Hesap sınıfını kullanacak programcı her ParaCek metodunu bir try bloğu içerisine yazmak zorunda kalacağı için sıkıntı yaşayabilir. Bu durumda bu tip dosya ile ilgili olmayan hataları kontrol edilmeyen hata sınıflarından kalıtım almak daha doğrudur. Zaten böyle bir durumu önlem alınmaz ise program çökecektir.

Kendi Exception (Hata) Sınıf Tasarımının C dilinde Benzetimi

```
// Hata.h dosyası
#ifndef HATA_H
#define HATA_H

#include "stdio.h"
#include "stdlib.h"
#include "setjmp.h"

struct HATA{
        char* mesaj;
        void (*Yoket)(struct HATA*);
};
typedef struct HATA* Hata;
Hata HataOlustur(char*);
void HataYoket(Hata);

#endif
```

```
//Hata.c dosyası
#include "Hata.h"

Hata HataOlustur(char* msg){
        Hata this;
        this=(Hata)malloc(sizeof(struct HATA));
        this->mesaj=msg;
        this->Yoket=&HataYoket;
}
void HataYoket(Hata h){
        if(h == NULL)return;
        free(h);
        h=NULL;
}
```

```
// SifiraBolunme.h dosyası
#ifndef SIFIRABOLUNME_H
#define SIFIRABOLUNME_H

#include "Hata.h"
typedef enum BOOL{false, true}boolean;
struct SIFIRABOLUNME{
        Hata super;
```

```
        char* (*Mesaj)(struct SIFIRABOLUNME*);
        void (*Yoket)(struct SIFIRABOLUNME*);
};
typedef struct SIFIRABOLUNME* SifiraBolunme;
SifiraBolunme SifiraBolunmeOlustur();
char* HataMesaj(const SifiraBolunme);
void SifiraBolunmeYoket(SifiraBolunme);

#endif
```

```
// SifiraBolunme.c dosyası
#include "SifiraBolunme.h"

SifiraBolunme SifiraBolunmeOlustur(){
        SifiraBolunme this;
        this=(SifiraBolunme)malloc(sizeof(struct SIFIRABOLUNME));
        this->super = HataOlustur("Sifira Bolunme Hatasi");
        this->Mesaj = &HataMesaj;
        this->Yoket = &SifiraBolunmeYoket;
}
char* HataMesaj(const SifiraBolunme s){
        return s->super->mesaj;
}
void SifiraBolunmeYoket(SifiraBolunme s){
        if(s == NULL)return;
        s->super->Yoket(s->super);
        free(s);
        s=NULL;
}
```

```
// Test.c dosyası
#include "SifiraBolunme.h"
jmp_buf jumper;
SifiraBolunme sb=NULL;
int Bol(int x,int y){
        if(y == 0){
                sb = SifiraBolunmeOlustur();
                longjmp(jumper, -3);
        }
        return x/y;
}
int main(){
        int a=10, b=0;
        if(setjmp(jumper) == 0){
                printf("%d",Bol(a,b));
        }
        else{
                printf(sb->Mesaj(sb));
        }
        return 0;
}
```

```
hepsi: derle calistir

derle:
        gcc -I ./include/ -o ./lib/Hata.o -c ./src/Hata.c
```

```
	gcc -I ./include/ -o ./lib/SifiraBolunme.o -c ./src/SifiraBolunme.c
	gcc -I ./include/ -o ./bin/Test ./lib/Hata.o ./lib/SifiraBolunme.o ./src/Test.c

calistir:
	./bin/Test
```

Hazırlayan

Arş. Gör. Dr. M. Fatih ADAK