Objects First With Java
A Practical Introduction Using BlueJ

Felhantering

Dagens meny

- Defensiv programmering.
  - Att förutse vad som kan gå fel.
- Exceptionella händelse.
- Felrapportering.
- Enkel filhantering.

Några orsaker till att det blir fel

- Felaktig implementation.
  - Programmet följer inte specifikationen.
- Felaktig objektreferens.
  - T.ex. fel index.
- Inkonsistens eller felaktigheter i ett objekts tillstånd.
  - T.ex. när man använder en klasserhierarki som man har dålig kunskap om på ett felaktigt sätt.

Inte alltid programmerarens fel ...

- Fel uppkommer ofta vid interaktion med omgivningen:
  - Felaktig URL skrivs in av en användare.
  - Problem med nätverkskommunikationen.
- Filhantering ger ofta fel:
  - Filer saknas.
  - Användaren har inte rätt behörighet.
  - Felaktigt filnamn.
  - Finns inte mer utrymme på sekundärminne.

Felundersökning

- Vi skall titta på felaktiga situationer med hjälp av bokens projekt *address-book*.
- Två aspekter:
  - Felrapportering.
  - Felhantering.

Defensiv programmering

- Klient-server:
  - Skall en server anta att klienterna är välprogrammerade?
  - Eller skall en server anta att klienterna kan ställa till problem?
- Innebär stora skillnader när det gäller implementationen av servern.

Vad man bör tänka på

- I vilken omfattning måste servern kontrollera metodanrop?
- Hur skall fel rapporteras?
- Hur kan en klient förutse vad som kan gå fel?
- Hur kan en klient hantera fel som uppkommer?

Ett exempel

- Försök att ta bort en icke-existerande post från servern.
- Vi får ett exekveringsfel ...
  - Vems ‘fel’ är det?
- Det är bättre att försöka förutse och förhindra fel generellt.

Parametervärden

- Ett server-objekt är sårbart när det gäller felaktiga parametervärden.
  - Parametervärden till konstruktorn initialiserar tillståndet.
  - Parametervärden bidrar ofta till hur objektet kommer att fungera.
- Att kontrollera parametervärden är en bra defensiv åtgärd.

Kontrollera nycklar

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

Felrapportering från servern

- Hur skall felaktiga parametervärden rapporteras?
  - Till användaren?
    - Finns det någon mänsklig användare?
    - I så fall kan hon/han lösa problemet?
  - Till klient-objektet?
    - Skall ett *felvärde* returneras?
    - Eller - skall en *exceptionell händelse* genereras?

Returnera ett *felvärde*

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Klientens gensvar

- Testa returvärdet.
  - Försök att göra en återhämtning om det blev fel.
  - Försök att undvika att programmet kraschar.
- Ignorera returvärdet.
  - Kan ändå inte göra något vettigt.
  - Innebär ofta att programmet kraschar.
- Hantera felet som en *exceptionell händelse*.

Principer för exceptionella händelser

- Bör kunna uttryckas i programmerings-språket.
- Inget speciellt *returvärde* skall behövas.
- Fel skall inte kunna ignoreras av klienten.
  - Den normala exekveringsgången skall avbrytas.
- Felåterhämtning skall underlättas.

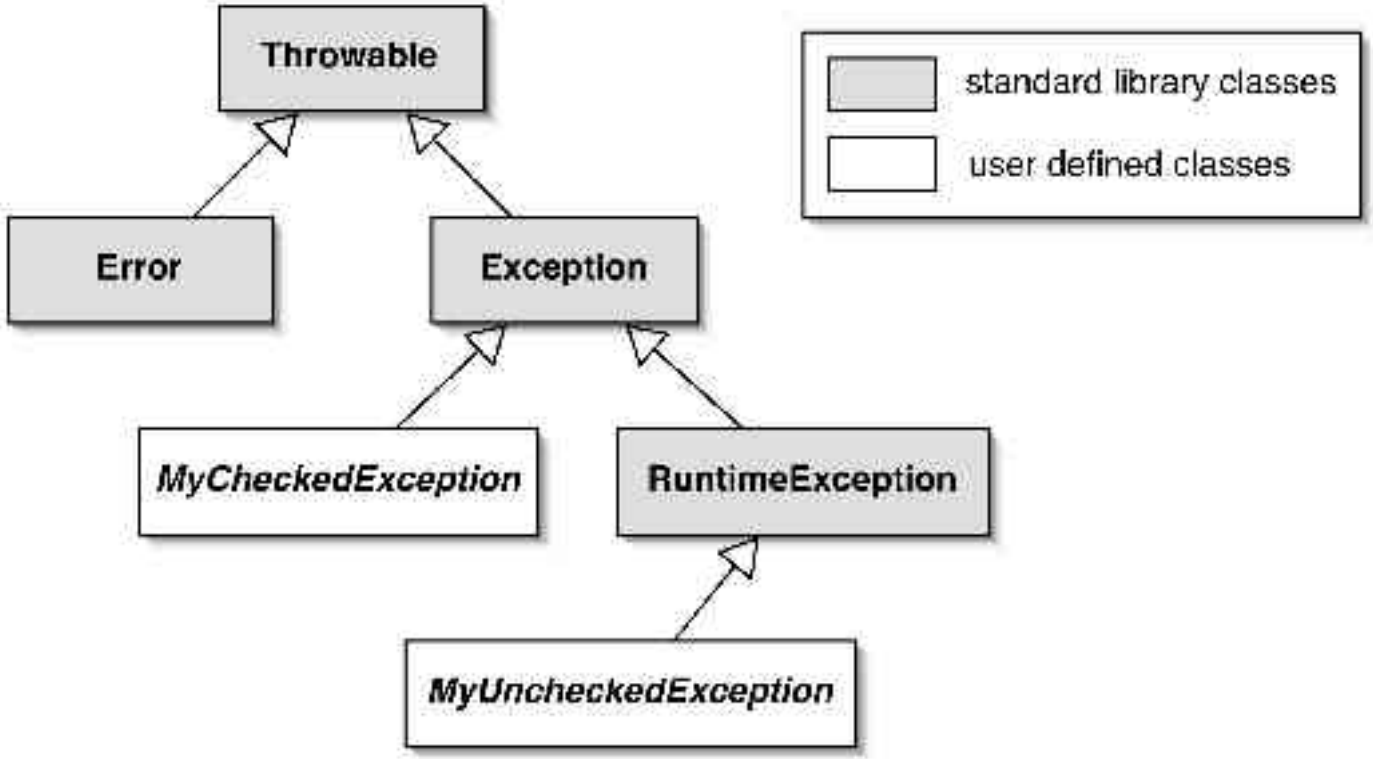
Att generera en exceptionell händelse

```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key,
 *         or null if there are none matching.
 * @throws NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
    if(key == null){
        throw new NullPointerException(
                              "null key in getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Konstruera - signalera

- Ett objekt av en klass som är subklass till klassen `java.lang.Throwable` skapas:
  - `new ExceptionType("...");`
- objektet signaleras (*is thrown*):
  - `throw ...`
- Javadoc dokumentation:
  - `@throws ExceptionType reason`

Klasshierarki för exceptionella händelser



Två slags exceptionella händelser

- Kontrollerade (*checked*) exceptionella händelser
  - Subklass till `Exception`
  - Används för exceptionella händelser som är möjliga att förutse.
  - Oftast kan man göra felåterhämtning.
- Okontrollerade (*unchecked*) exceptionella händelser
  - Subklass till `Error` eller `RuntimeException`
  - Används för exceptionella händelser som kan vara svåra eller omöjliga att förutse.
  - Oftast kan man inte göra felåterhämtning.

Effekter av en kontrollerad exceptionell händelse

- Metoden som genererar den exceptionella händelsen avslutas i förtid.
- Den ger inte något returvärde.
- Exekveringen återvänder inte till satsen efter metodanropet.
  - Innebär att klienten inte kan fortsätta sin exekvering.
- En klient kan däremot fånga (*catch*) den exceptionella händelsen.

Okontrollerade exceptionella händelser

- Kompilatorn kontrollerar inte att det finns kod för att fånga exceptionella händelser.
- Innebär att programmet stoppas om de inte finns någon sådan kod.
  - Det är vad som oftast händer.
- `IllegalArgumentException` är ett typexempel.

Kontroll av parametervärden

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                            "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                            "Empty key passed to getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Kontroll av (en kombination av) parametervärden

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```

Hantering av exceptionella händelser

- Kontrollerade exceptionella händelser förutsätts bli fångade.
- Kompilatorn kontrollerar koden för att fånga och ta hand om händelsen.
  - Både i server och klient.
- Om de används på avsett sätt så ger de ofta en möjlighet till felåterhämtning.

Specifikation av *throws*

- Metoder som genererar en kontrollerad exceptionell händelse måste specificera detta i metodhuvudet:

```
public void saveToFile(String destinationFile)
    throws IOException
```

try-blocket

- Klienter som fångar en exceptionell händelse måste skydda anropet med ett *try*-block:

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
```

try-blocket

1. Den exceptionella händelsen uppstår när något går fel här.

```
try{
    addressbook.saveToFile(filename);
    tryAgain = false;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    tryAgain = true;
}
```

2. Händelsen fångas upp och exekveringen fortsätter här.

Fångst av flera exceptionella händelser

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Take action on an end-of-file exception.
    ...
}
catch(FileNotFoundException e) {
    // Take action on a file-not-found exception.
    ...
}
```

`finally`-blocket

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
finally {
    Perform any actions here common to whether or not
    an exception is thrown.
}
```

`finally`-blocket

- Satserna i ett *finally*-block exekveras även om en *return*-sats exkveras i *try*- eller *catch*-blocket!
- Om en exceptionell händelse inte fångats eller propagerats vidare så gäller fortfarande att satserna i *finally*-blocket kommer att exekveras.

Egendefinierade exceptionella händelser

- Bilda en subklass till `Exception` eller `Runtime-Exception`.
- Innebär att vi får en egendefinierad exceptionell händelsetyp som kan ge bättre felrapportering.
  - Inkludera mer precis information om händelsetypen, varför den uppkommit, hur felåterhämtning skett etc.

```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```

Felåterhämtning

- Klienter skall programmeras så att de observerar och kan hantera fel.
  - Kontrollera returvärden!
  - Ignorera inte exceptionella händelser!
- Skriv kod som kan hantera felåterhämtning.
  - Kräver ofta en slinga.

Försök till felåterhämtning

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Report the problem and give up;
}
```

Undvik att fel uppstår

- Klienter kan ofta använda metoder hos servern för att kontrollera om något låter sig göras (*query methods*).
  - Robusta klienter innebär att en server kan skrivas effektivare utan onödiga kontroller.
  - Okontrollerade exceptionella händelser kan t.ex. användas.
  - Förenklar programmeringen av klienterna.
- Men - ökar graden av koppling mellan klient och server.

In- och utmatning av text

- Fel uppstår ofta i samband med in- och utmatning.
  - Innebär interaktion med en okontrollerbar omgivning.
- Paketet `java.io` innehåller stöd för in- och utmatning.
- `java.io.IOException` är en kontrollerad exceptionell händelse.

Readers, writers, streams

- *Readers* och *writers* hanterar in- och utmatning av text.
  - Baseras på `char`-typen.
- *Streams* hanterar binärdata.
  - Baseras på `byte`-typen.
- Projektet *address-book-io* ger exempel på in- och utmatning av text.

Textutmatning

- Använd klassen `FileWriter`.
  - Öppna en fil.
  - Skriv till filen.
  - Stäng filen.
- Om det blir fel någonstans:

```
IOException
```

Textutmatning

```
try {
    FileWriter writer = new FileWriter("name of file");
    while(there is more text to write) {
        ...
        writer.write(next piece of text);
        ...
    }
    writer.close();
}
catch(IOException e) {
    something went wrong with accessing the file
}
```

Textinmatning

- Använd klassen `FileReader`.
- Koppla till `BufferedReader` för att få radbaserad inmatning.
  - Öppna en (text)fil.
  - Läs från filen.
  - Stäng filen.
- Om det blir fel någonstans:

```
IOException
```

Textinmatning

```
try {
    BufferedReader reader = new BufferedReader(
                            new FileReader("name of file "));
    String line = reader.readLine();
    while(line != null) {
        do something with line
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    the specified file could not be found
}
catch(IOException e) {
    something went wrong with reading or closing
}
```

Sammanfattning

- Fel under exekveringen kan ha många orsaker.
  - En klient anropar ett server-objekt på ett felaktigt sätt.
  - En server kan inte utföra en begäran.
  - Programmeringsfel i klienten och/eller servern.
  - ...

Sammanfattning

- Fel under exekveringen innebär ofta att ett program kraschar.
- Defensiv programmering försöker att föregripa fel - både i klient och i server.
- Exceptionella händelser ger en bra utgångspunkt både för felrapportering och felåterhämtning.