Objektno orijentirano programiranje

8. Upravljanje pogreškama

Creative Commons

- **slobodno smijete:**
  - **dijeliti** — umnožavati, distribuirati i javnosti priopćavati djelo
  - **remiksirati** — prerađivati djelo
- **pod sljedećim uvjetima:**
  - **imenovanje**. Morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
  - **nekomercijalno**. Ovo djelo ne smijete koristiti u komercijalne svrhe.
  - **dijeli pod istim uvjetima**. Ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.

U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela. Najbolji način da to učinite je linkom na ovu internetsku stranicu.

Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.

Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.

Tekst licencije preuzet je s http://creativecommons.org/.

Pojava pogreške: klasični pristup

- Što napraviti kada se u funkciji dogodi greška?
  - prekinuti izvođenje programa
  - vratiti status pogreške
    - najčešće, nema posebnog mjesta za vraćanje statusa
  - funkcija vraća i podatak i status na istom mjestu → loše
- Primjer: funkcija iz C-a za čitanje znaka sa standardnog ulaza

```
int getchar(void);
```

  - ako je nastupila pogreška, rezultat je EOF (vrijednost manja od nule)
  - inače, rezultat treba pretvoriti u *(unsigned) char* i tumačiti kao pročitani podatak
  - „zlouporaba” povratne vrijednosti pa povratni tip nije *char* već *int*

Nedostatak klasičnog pristupa

- Koristimo li ovaj pristup, kod se pretvara u nešto poput

```
naredba1;
if(rezultat je greška) { akcija1; }
naredba2;
if(rezultat je greška) { akcija2; }
naredba3;
if(rezultat je greška) { akcija3; }
naredba1;
```

ili

```
naredba1;
if(!(rezultat je greška)) {
  naredba2;
  if(!(rezultat je greška)) {
    naredba3;
    if(rezultat je greška) {
      čišćenje3;
    }
  } else {
    čišćenje2;
  }
} else {
  čišćenje1;
}
```

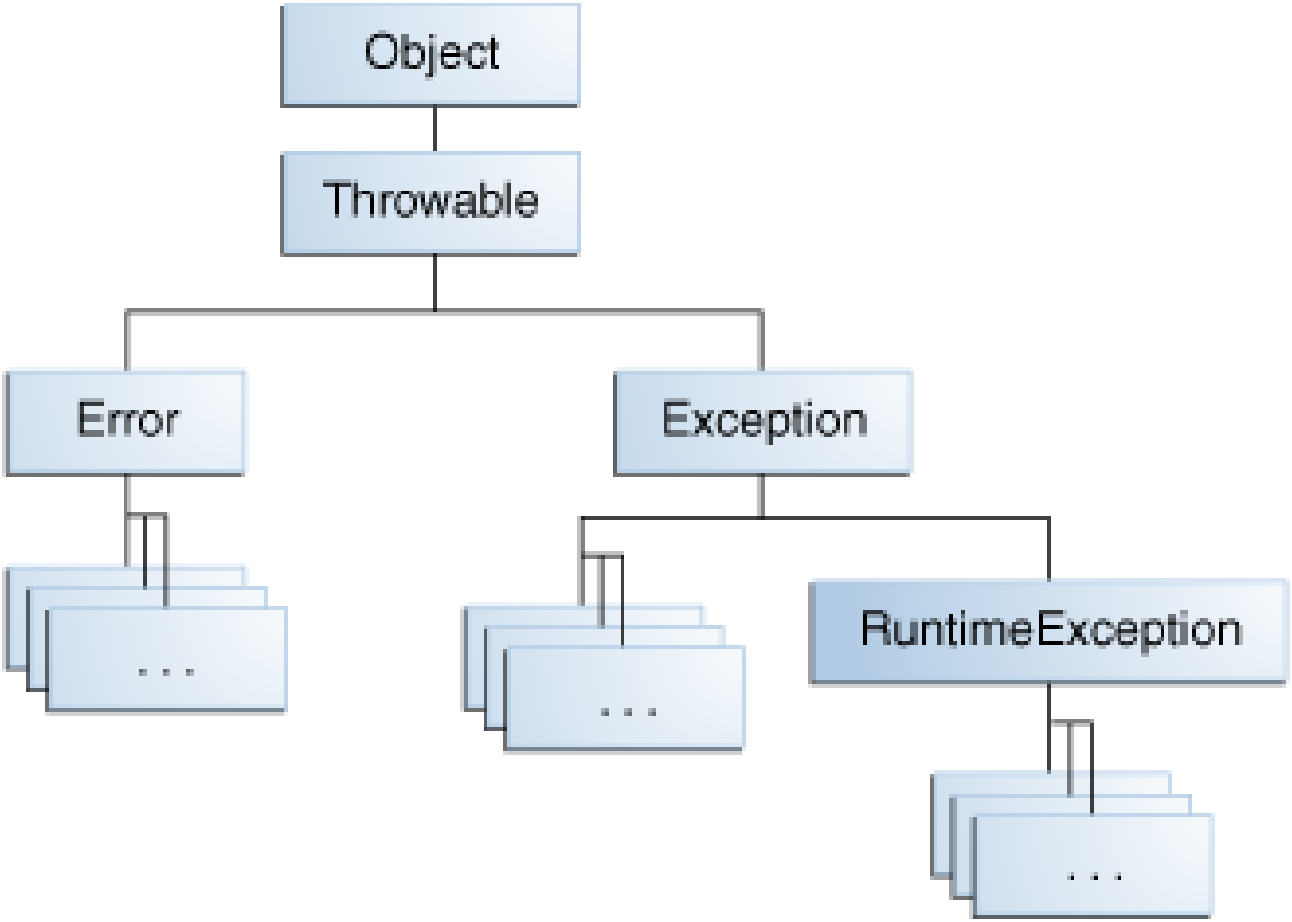
uz mogućnost „pojednostavljenja” korištenjem naredbe *goto*

Pojava pogreške: moderniji pristup

- Noviji programski jezici (primjer je već C++) uvode pojam **iznimke** koja opisuje **iznimnu situaciju**
- Engleski termin je *Exception*
- Ako metoda regularno završi onda sigurno vraća podatak
  - stoga nema razloga da povratni tip bude pogrešan
- Ako se tijekom izvođenja metode dogodi pogreška, izazvat će se iznimna situacija koja će biti opisana prikladnim objektom
  - Taj objekt nazivamo **iznimkom i** sadrži detaljnije informacije o razlogu nastanka iznimne situacije

Iznimke

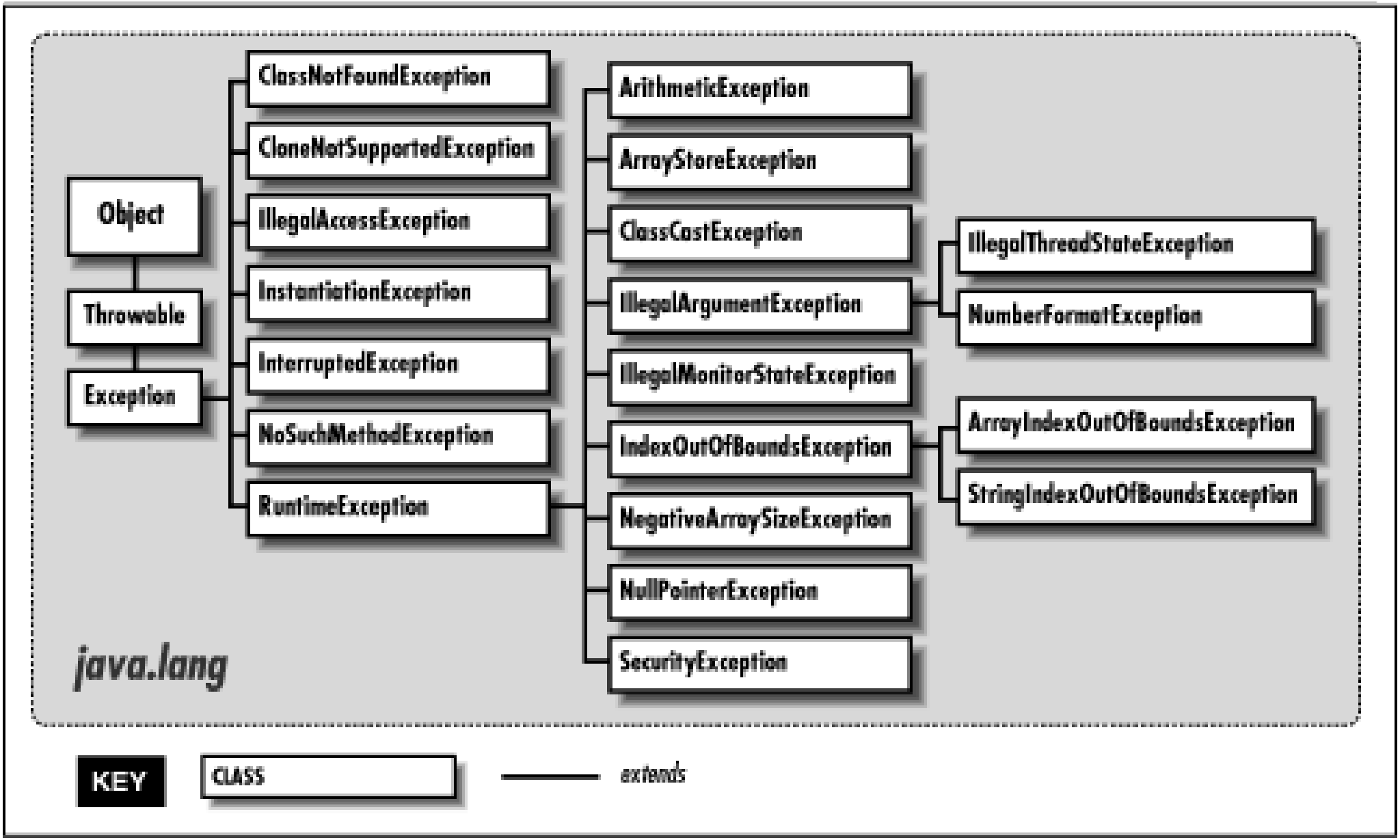
- Za opis iznimne situacije programski jezik Java koristi primjerke klase *Throwable* ili klasa koji su iz nje (direktno ili indirektno) izvedeni



Klasa *Throwable*

- Klasa *Throwable* omogućava pristup podatcima kao što su:
  - cjelokupno stanje na stogu u trenutku kada je nastala iznimna situacija, npr. main() → m1() → m2() → m3() → iznimka
  - točna lokacije iznimke u kodu (koja datoteka, koji redak) za svaku metodu
  - poruka pogreške
  - pristup do "omotane" iznimke, ako takva postoji
- Omogućava i pristup metodama poput metode *printStackTrace* za ispis svih informacija na standardni izlaz za pogreške
- Klase izvedene iz klase *Throwable* dodavat će druge prikladne informacije ovisno o vrstama iznimnih situacija koje opisuju

Stablo iznimki



Primjer pojave iznimke

- Izvršavanje sljedećeg programskog odsječka izaziva iznimku tipa *NumberFormatException* (i rušenje programa) u zadnjem prolazu kroz petlju, jer `"abc"` nije moguće pretvoriti u broj

```
String[] arr = new String[]{ "12", "15", "abc"};
for(int i=0; i<3; i++){
   int num = Integer.parseInt(arr[i]);
   System.out.println(num);
}
System.out.println("Done");
```

hr.fer.oop.topic8.exceptions.example1.Main

```
C:\eclipse\workspace\08_Exceptions>java -cp bin hr.fer.oop.topic8.exceptions.exa
mple1.Main
12
15
Exception in thread "main" java.lang.NumberFormatException: For input string: "a
bc"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at hr.fer.oop.topic8.exceptions.example1.Main.main(Main.java:8)
```

Debug

- Main (14) [Java Application]
  - hr.fer.oop.topic8.exceptions
    - Thread [main] (Suspended (exception NumberFormatException))
      - Integer.parseInt(String, int) line: not available
      - Integer.parseInt(String) line: not available
      - Main.main(String[]) line: 8

Primjer obrade iznimke

- Kôd u kojem se može očekivati iznimka stavimo u *try* blok iza kojeg slijedi *catch* blok s kôdom koji služi za obradu iznimke

```
String[] arr = new String[]{ "12", "15", "abc"};
for(int i=0; i<3; i++){
   try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
   }
   catch(NumberFormatException exc){
        System.err.println("Error: " + exc);
   }
}
System.out.println("Done");
```

hr.fer.oop.topic8.exceptions.example2.Main

```
C:\eclipse\workspace\08_Exceptions>java -cp bin hr.fer.oop.topic8.exceptions.exa
mple2.Main
12
15
Error: java.lang.NumberFormatException: For input string: "abc"
Done
```

Obrada iznimke

- kôd koji može izazvati iznimku smješta se u *try* blok
- ovisno o vrstama iznimaka koje se očekuju piše se jedan ili više *catch* blokova
- redoslijed *catch* blokova je bitan, jer se po nastupanju iznimne situacije ispitivanje provodi od prvog bloka *catch*
- prvi blok koji deklarira obradu iznimke koja je „kompatibilna” s nastalom iznimkom bit će odabran za izvođenje
  - ako takvog nema, iznimka se propagira dalje
- nakon izvođenja *catch* bloka, program se nastavlja s prvom naredbom koja slijedi iza čitave try-catch strukture
  - nema povratka u try blok (iznimka nije nalik pozivu metode)!

Primjer s više mogućih iznimki

- U sljedećem primjeru iznimku može izazvati parsiranje broja, ali i pristup članu polja koji ne postoji

```
String[] arr = new String[]{ "12", "15", "abc"};
for(int i=0; i<4; i++){
   try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
   }
   catch(NumberFormatException exc){
        System.err.println(exc);
   }
   catch(ArrayIndexOutOfBoundsException exc){
        System.err.println(exc);
   }
}
System.out.println("Done");
```

hr.fer.oop.topic8.exceptions.example3.Main

```
C:\eclipse\workspace\08_Exceptions>java -cp bin hr.fer.oop.topic8.exceptions.example3.Main
12
15
java.lang.NumberFormatException: For input string: "abc"
java.lang.ArrayIndexOutOfBoundsException: 3
Done
```

Multi-catch

- U slučaju da se obrada više vrsta iznimaka obavlja na identičan način, umjesto kopiranja obrade u više blokova možemo koristiti *multi-catch*
- Više tipova iznimaka „polijepi” se znakom |

hr.fer.oop.topic8.exceptions.example3.MainMultiCatch

```
try{
   int num = Integer.parseInt(arr[i]);
   System.out.println(num);
}
catch(NumberFormatException | ArrayIndexOutOfBoundsException exc){
   System.err.println(exc);
}
```

Primjer neispravnog poretka *catch* blokova

- Pri poretku catch blokova treba voditi računa o nasljeđivanju iznimki.
  - Npr, zašto je sljedeći programski odsječak logički pogrešan?

```
try {
    ...
} catch(Exception ex) {
    ...
} catch(NumberFormatException ex) {
    ...
}
```

Iznimke i čišćenje resursa

- Neovisno o tome da li je došlo do iznimke ili ne, ponekad treba obaviti dio kôda koji će predstavlja završno čišćenje resursa
  - npr., zatvoriti resurs koji je zatvoriv, tj. implementira sučelje *Closeable,* npr. *Scanner*

```
Scanner s = new Scanner(...);
try {
   String line = s.readLine(); //read line that should contain date
   LocalDate date = LocalDate.parse(line);
   ...
   s.close();
}
catch(DateTimeParseException ex) {
   … do something …
    s.close();
}
```

- Ako ima više vrsta pogrešaka, ovakav kod postaje nepraktičan
  - Staviti zatvaranje scannera iza try-catch bloka? Što ako se pojavi iznimka koja nije predviđena niti jednim *catch* blokom?

finally blok

- Rješenje za zajednički kod *try* i *catch* bloka može biti *finally* blok
- Kod smješten u blok *finally* izvodi se uvijek, neovisno o tome kako je završeno izvođenje bloka *try-catch*.

```
Scanner s = new Scanner(...);
try {
    ...
}
catch(SomeResourceException ex) {
   … do something …
}
finally {
    s.close();
}
```

Konstrukcija *try – finally* (bez catch)

- Moguće je imati i *try – finally* blok, bez *catch* bloka
- Kôd u *finally* bloku će se uvijek izvršiti, neovisno da li je nastala iznimka u *try* dijelu, ali nema koda za obradu iznimke
  - propagira se dalje
  - eventualna nova iznimka u finally bi maskirala originalnu

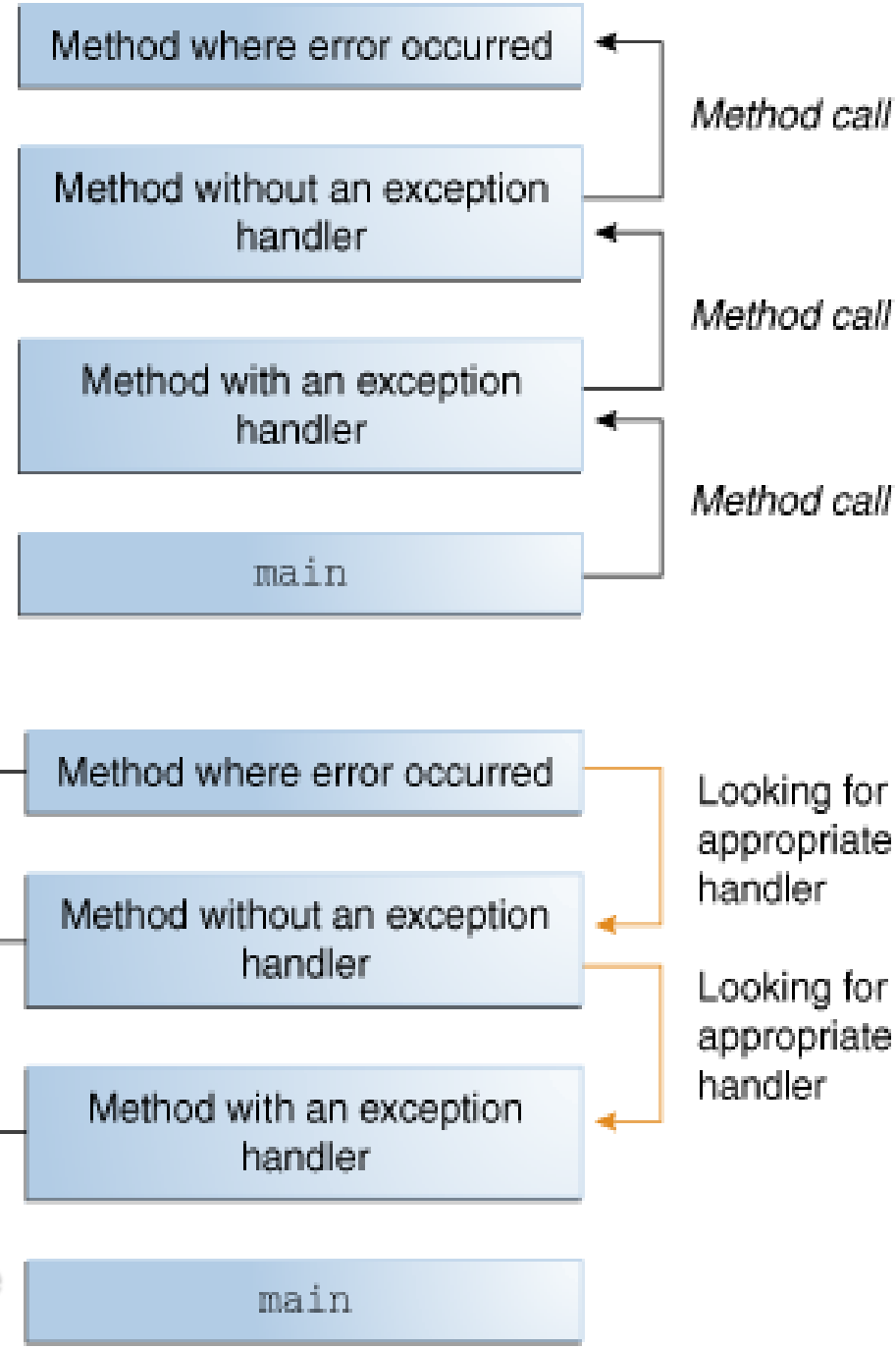
```
String s = "a13";
try{
      try{  int i = Integer.parseInt(s);  }
      finally{ System.out.println("Finally 1"); }
}
catch(Exception e){
      System.out.println("Catch");
}
finally{ System.out.println("Finally 2"); }
```

hr.fer.oop.topic8.exceptions.example4.Main

```
Ispis:
Finally 1
Catch
Finally 2
```

Općeniti primjer obrade iznimke (1)

- Npr. neka main poziva m1 koji poziva m2 koji poziva m3 u kojem se potom dogodi iznimna situacija
- Potraga za prikladnim *catch* blokom kreće od metode u kojoj je iznimka izazvana, i vraća se metodu po metodu unatrag po stogu pozivatelja
  - Prilikom povratka po stogu, izvode se svi *finally* blokovi (ako postoje)
- Prvi try-catch blok koji uhvati izazvanu iznimku odredit će naredbu s kojom će se nastaviti izvođenje programa
  - to će biti prva naredba iza tog try-catch bloka
- ako nitko ne uhvati iznimku, uhvatit će je pozivatelj prve metode
  - u ovom slučaju virtualni stroj koji će potom terminirati dretvu koja je izvodila ovaj kôd
  - u jednodretvenim aplikacijama (sve što smo do sada radili), to za posljedicu ima terminiranje aplikacije



Općeniti primjer obrade iznimke (2)

- U jezicima koji nemaju automatsko upravljanje memorijom, ovakav mehanizam oporavka od pogrešaka može dovesti do problema curenja memorije
  - To je jedan od razloga zašto u ranim verzijama jezika C++ mehanizam iznimaka nije bio korišten u većoj mjeri
- Kako programski jezik Java ima automatsko skupljanje smeća, ovo više nije problem

Izazivanje iznimne situacije

- Programer u Javi i sam može izazvati iznimnu situaciju
- Najprije je potrebno stvoriti prikladan primjerak razreda koji opisuje nastalu iznimnu situaciju a potom pokrenuti postupak obrade iznimne situacije uporabom ključne riječi *throw*.

hr.fer.oop.topic8.exceptions.example5.Main

```
public static boolean isTriangle(int a, int b, int c){
   if (a <= 0 || b <=0 || c<=0)
        throw new IllegalArgumentException(
              "Arguments must be greater than 0");
        if (a + b > c && a + c > b && b + c > a)
              return true;
        else
              return false;
   }
```

Omatanje iznimki

- Ako metoda ne može smisleno obraditi tu iznimku, a također je ne može baciti iz metode (jer primjerice ne smijemo mijenjati signaturu metode), originalnu iznimku možemo zamotati:

```
try {
   ...
} catch(WrongException ex)
{
      throw new OKException(ex);
}
```

- Pretpostavka je da *OKException* ima konstruktor koji prima referencu na drugu iznimku
- Do omotane iznimke može se doći pozivom metode *getCause()* koja je definirana u razredu *Throwable*

Prosljeđivanje iznimki

- Također, ako je potrebno obaviti neku akciju prije no što se obrada iznimke proslijedi dalje, iznimku je moguće uhvatiti, napraviti potrebnu akciju pa ponovno aktivirati

```
try {  ...}
catch(SomeException ex) {
      … do here something …
      throw ex;
}
```

- Pretpostavka: u bloku *catch* napisani kod ne obrađuje iznimku (možda samo nešto bilježi ili slično)
  - Ako taj kod slučajno izazove novu iznimku, ta iznimka maskira originalnu

Podjela pogrešaka u Javi

- U prethodnim primjerima radili smo s iznimkama koje su izvedene iz klase *RuntimeException* i spadaju u tzv. *neprovjeravane iznimke*.

- Programski jezik Java iznimne situacije dijeli u tri velike porodice (dani su neformalni nazivi):
  - grube pogreške (*errors*)
  - neprovjeravane iznimke (*runtime exceptions*)
    - česte, očekivane, …
  - provjeravane iznimke (*checked exceptions*)

Grube pogreške (engl. errors)

- Stablo iznimki počinje od klase *Throwable*
- Klasa *Error* i njeno podstablo modelira opise iznimnih situacija od kojih se ne očekuje da je moguć oporavak
  - npr. kad virtualni stoj prilikom čitanja datoteke s byte-kodom ustanovi da je datoteka pogrešnog formata → *ClassFormatError*
  - Ili kad nema dovoljno memorije kako bi se zadovoljio zahtjev za alokacijom objekta → *OutOfMemoryError*
- Većina programa u Javi nikada ne obrađuje grube pogreške

Iznimke (engl. *exceptions*)

- Stablo iznimki počinje od klase *Throwable*
- Klasa *Exception* i njeno podstablo modelira opise iznimnih situacija od kojih se očekuje da je moguć oporavak
- Dijele se u dvije porodice:
  - neprovjeravane iznimke (engl. u*nchecked exceptions*) - one koje su izvedene iz klase *RuntimeException* (koja je direktni potomak klase *Exception*)
  - provjeravane iznimke (engl. c*hecked exceptions*) - izvedene iz klase *Exception*

Neprovjeravane iznimke (1)

- **Neprovjeravane iznimke** modeliraju situacije koje se mogu javiti praktički na svakom koraku izvođenja programa, npr.:
  - Pokušaj pretvorbe stringa "abc" u broj → *NumberFormatException*
  - pristup lokaciji polja koja ne postoji → IndexOutOfBoundsException
  - dereferenciranje *null* objekta: nad njim se poziva metodu ili pristupa članskoj varijabli → *NullPointerException*
  - pokušava se napraviti nelegalno ukalupljivanje: referencu na *Item* ukalupljujemo na *Food*, a referenca je u stvarnosti pokazivala na objekt tipa Beverage → *ClassCastException*

Neprovjeravane iznimke (2)

- Prilikom pisanja koda ne treba eksplicitno naglašavati da pojedine metode mogu izazvati takve iznimne situacije
- Nije nužno ugraditi upravljanje takvim iznimkama
  - ako se pojave, a programer nije napisao kod za obradu iznimke, virtualni stroj će prekinuti dretvu koja je izvodila kod koji je rezultirao pojavom iznimne situacije (potencijalno izazivajući i prekid samog procesa: više o tome nakon što obradimo višedretvenost)

Provjeravane iznimke

- **Provjeravane iznimke** modeliraju situacije koje nisu fatalne po aplikaciju (oporavak je moguć), ali ih se mora eksplicitno obrađivati ili naglasiti da se prosljeđuju dalje
- Primjeri provjeravanih iznimki:
  - tražimo od virtualni stroja da učita klasu za koju virtualni stroj nema byte-kod → *ClassNotFoundException*
  - prilikom korištenja API-a za pristup datotekama i tijekom pristupa došlo je do pogreške → *IOException* i podstablo
  - pri korištenju API-a za pristup bazi podataka jezikom SQL i tijekom pristupanja došlo je do pogreške → *SQLException* i podstablo

Primjer rada s provjeravanim iznimkama (1)

- Za dohvat svih redaka tekstne datoteke može se iskoristiti statička metoda *readAllLines* u klasi *Files*.
  - Metoda vraća listu stringova
  - Iznimka koja se pritom može dogoditi je izvedena iz *IOException*.
    - Konkretno, ako datoteka ne postoji vraća se *NoSuchFIleException*

hr.fer.oop.topic8.exceptions.example6.Main

```
public static void main(String[] args) {
  try{
      Path filename = Paths.get("C:/temp", "nofile.txt");
      List<String> list = Files.readAllLines(filename);
      for(String line : list)
            System.out.println(line);
  }
  catch(IOException exc){ System.err.println(exc); }
}
```

Primjer rada s provjeravanim iznimkama (2)

- Što da u prethodnom primjeru nismo hvatali iznimku?
- *IOException* spada u kategoriju provjeravanih iznimki koje se moraju obrađivati ili se metoda u kojoj se ta iznimka može dogoditi mora označiti kao metoda koja baca iznimku
  - i to vrijedi dalje za sve metode koje pozivaju te metode, sve do metode koja obrađuje tu pogrešku ili do metode *Main*

hr.fer.oop.topic8.exceptions.example6.MainThrows

```
public static void main(String[] args) throws IOException {
   Path filename = Paths.get("C:/temp", "nofile.txt");
   List<String> list = Files.readAllLines(filename);
   for(String line : list)
       System.out.println(line);
}
```

Obrada iznimke: automatsko upravljanje resursima (*try with resources*)

- Za jednostavnije upravljanje određenom vrstom resursa, od Jave 7 dostupan je još jedan oblik bloka *try* koji nas oslobađa pisanja repetitivnog kôda
  - takav oblik try bloka naziva se *try with resources*
  - može se koristiti samo s klasama koje implementiraju *AutoCloseable* i/ili *Closeable*.
  - resursi ovog tipa otvaraju se unutar zagrade koja slijedi odmah nakon ključne riječi try
  - nema potrebe za pisanjem eksplicitnoga kôda koji će se brinuti za zatvaranje ovih resursa: kompajler će sam generirati potreban kôd za finally blok
  - ovakav način rada s resursima koristit ćemo kod rada s datotekama (odnosno tokovima podataka općenito)

```
package java.lang;
public interface AutoCloseable {
        void close() throws Exception;
}
package java.io;
public interface Closeable extends AutoCloseable {
        public void close() throws IOException;
}
```

Primjer automatskog upravljanja resursima (1)

- Neka klasa *Resource* implementira *Autocloseable* i ispisuje poruke u konstruktoru te prilikom poziva metode *close*

```
public class Resource implements AutoCloseable {
   private int i;
   public Resource(int n){
       System.out.println("Creating #" + n);
       i = n;
   }

   @Override
   public void close() throws Exception {
       System.out.println("Closing #" + i);
   }
}
```

hr.fer.oop.topic8.exceptions.example7.Resource

Primjer automatskog upravljanja resursima (2)

- Nakon izlaska iz try bloka (bez obzira da li se iznimka dogodila ili ne), automatski se poziva metoda *close*

hr.fer.oop.topic8.exceptions.example7.Main

```
public static void main(String[] args) {
   try(Resource r1 = new Resource(1);
       Resource r2 = new Resource(2)){
              int a = 5, b = 0;
              a = a / b;
   }
   catch (Exception e) { e.printStackTrace(); }
}
```

```
C:\eclipse\workspace\08_Exceptions>java -cp bin hr.fer.oop.topic8.exceptions.exa
mple5.Main
Creating #1
Creating #2
Closing #2
Closing #1
java.lang.ArithmeticException: / by zero
        at hr.fer.oop.topic8.exceptions.example5.Main.main(Main.java:8)
```

Iznimke prilikom automatskog upravljanje resursima

- Prilikom automatskog zatvaranja resursa u bloku *finally* koji generira sam prevoditelj moguće je da pokušaj zatvaranja resursa izazove novu iznimku
  - u tom slučaju generirani će kôd dalje proslijediti originalnu iznimku (izazvanu u bloku *try*) pri čemu će njoj u listu potisnutih iznimaka dodati iznimku generiranu i uhvaćenu u bloku *finally*.
  - do popisa potisnutih iznimki može se doći pozivom metode *getSuppressed*() koja je definirana u razredu *Throwable*.
    - Pogledajte njezinu dokumentaciju!

Stvaranje vlastitih iznimki

- U okviru standardnih Javinih biblioteka dostupno je mnoštvo raznovrsnih iznimaka koje se mogu slobodno koristiti
- Ponekad je, međutim, praktično definirati novu vrstu iznimke (ili čak novu porodicu iznimki)
  - primjerice, za rad s matricama ima smisla definirati novu vršnu iznimku *MatrixException*
  - ako korisnik pokuša zbrajati nekompatibilne matrice, zgodno je imati *IncompatibleMatrixException*
  - ako pokuša invertirati neinvertibilnu matricu, zgodno je imati *SingularMatrixException*

Stvaranje vlastitih iznimki

- Da bismo klasu *MatrixException* mogli koristiti pri izazivanju iznimne situacije, nužno je da je izvedemo iz klase koji je barem *Throwable*
  - Izvođenje iz klase Throwable je nepoželjno; to je preopćenita iznimka
  - Izvođenje iz grane klase *Error* nema smisla jer ovo nisu neoporavive pogreške
  - Ostaje odluka hoćemo li klase izvesti iz nekog od klasa koje su izvedeni iz klase *RuntimeException* ili ne (drugim riječima, hoće li iznimke biti provjeravane ili ne)

Provjeravane iznimke ili ne?

- Od uvođenja iznimaka u Javu traju kontroverze oko njihove uporabe
  - Jedna strana tvrdi da uporaba (i stvaranje novih) provjeravanih iznimki tjera korisnike da nepotrebno opterećuju kod nizanjem throws deklaracija kod metoda koje ne obrađuju iznimke
  - Druga strana tvrdi da je upravo poanta natjerati programera ili da iznimku obradi ili da jasno kaže da metoda izaziva takvu iznimku („iznimka ne smije biti neprovjeravana”)
  - Neke poveznice na tu temu:
    - Unchecked Exceptions — The Controversy http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
    - Java theory and practice: The exceptions debate http://www.ibm.com/developerworks/library/j-jtp05254/
    - The Trouble with Checked Exceptions http://www.artima.com/intv/handcuffs2.html
    - Java's checked exceptions were a mistake http://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html

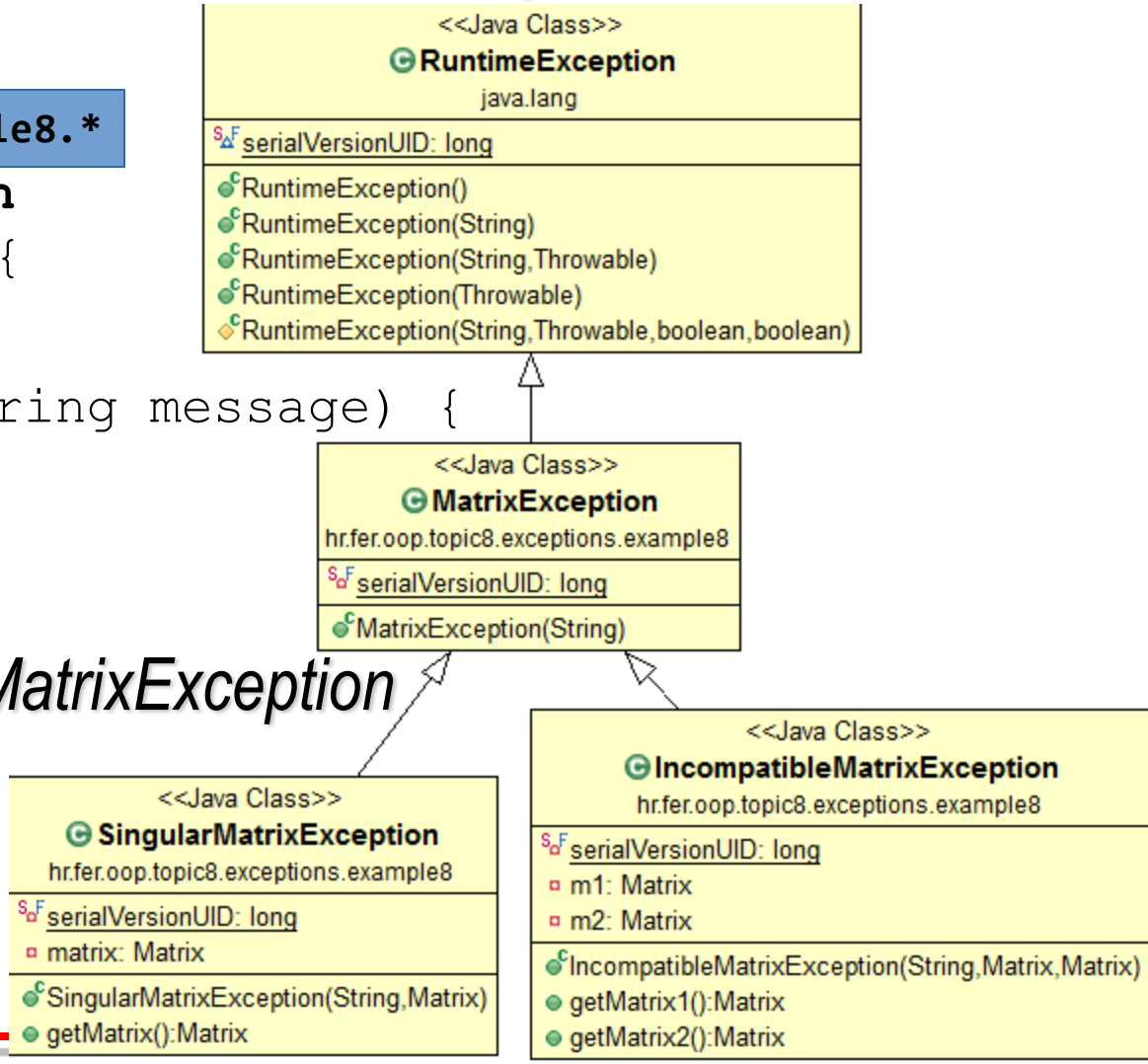
Primjer vlastite iznimke

- Odlučit ćemo se za uporabu klase *RuntimeException* kao bazne klase

```
hr.fer.oop.topic8.exceptions.example8.*
public class MatrixException
   extends RuntimeException {
   …
   public MatrixException(String message) {
       super(message);
   }
}
```

- Ostale iznimke izvodimo iz *MatrixException*

```
public class
   SingularMatrixException
   extends MatrixException
{   ...  }
```



Rad s vlastitim iznimkama

- Vlastite iznimke koristimo kao i ugrađene

```
public Matrix add(Matrix other) {
  if (rows != other.rows || cols != other.cols) {
      throw new IncompatibleMatrixException(
                  "Addition is not possible.",
                  this, other);
  }
```

hr.fer.oop.topic8.example8.Matrix

```
try {
  Matrix m5 = m4.add(m3);
  ...
  } catch(IncompatibleMatrixException ex) {
      ...
  }
```

hr.fer.oop.topic8.example8.Main