Objektno orijentirano programiranje

7. Upravljanje pogreškama: Iznimke

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Pojava pogreške: klasični pristup

- Što napraviti kada se u funkciji dogodi greška?
  - prekinuti izvođenje programa?
  - vratiti (ako je moguće) neispravnu vrijednost i postaviti status pogreške?
    - najčešće nema posebnog mjesta za vraćanje statusa
- Što „stariji" jezici (npr. C) rade u slučaju pogreške?
  - 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*
  - Vodi prema kodu s puno if-ova vezanih uz obradu pogreške

Mana klasičnog (C-ovskog) 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*

Moderni pristup - iznimke

- Nema razloga da povratni tip bude pogrešan
  - Ako metoda regularno završi onda vraća ispravan podatak
  - *Iznimka (engl. Exception)* za iznimne situacije koje prekidaju normalno izvršavanje programa
- Iznimka je objekt koji sadrži detaljnije informacije o razlogu i mjestu nastanka iznimne situacije
  - Kažemo da je iznimka *izazvana* ili *bačena* (engl. *thrown*) iz neke metode i može biti *uhvaćena* ili *obrađena* (engl. *caught*)
  - Daljnje izvršavanje programa nastavlja se od mjesta gdje je iznimka uhvaćena (ako je bila uhvaćena – inače se program ruši)

Primjer pojave iznimke

- Interpretiranje sadržaja stringa u obliku broja će uspjeti samo ako se unutra nalazi broj
  - U protivnom, *parseInt* ne može nastaviti dalje
    - Vraćanje broja 0, -1 ili nekog „magičnog broja” nije prikladno, jer ne bi mogli razlikovati pogrešku od situacije u kojoj string sadrži taj broj
    - *parseInt* prekida svoje izvršavanje i izaziva iznimku (baca objekt) tipa *NumberFormatException*

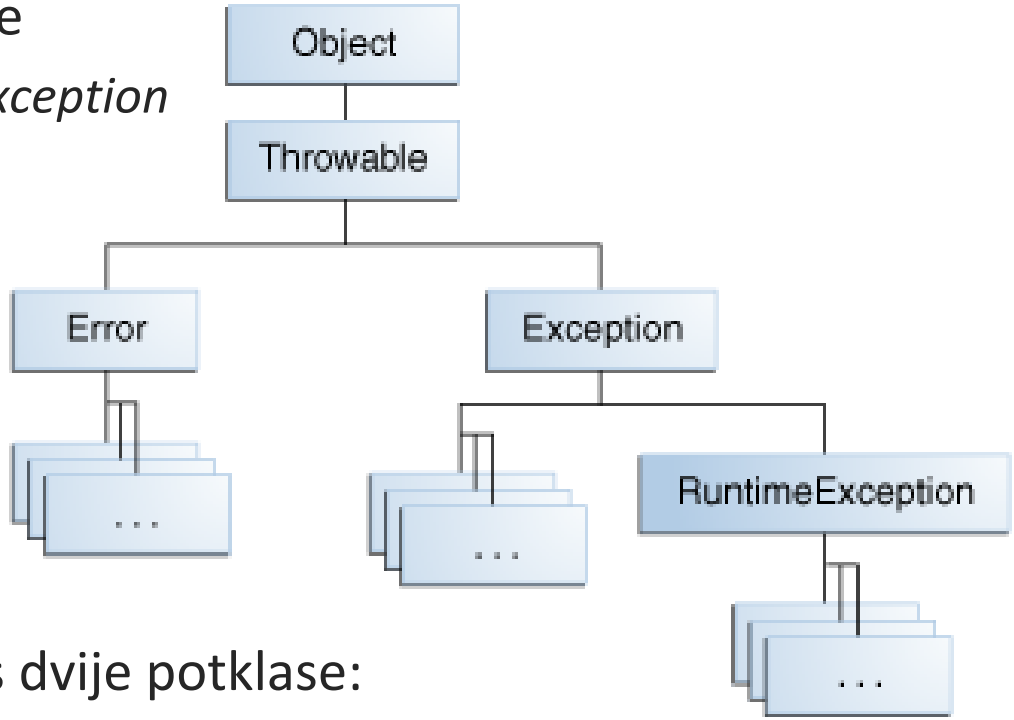
```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<arr.length; i++) {
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
}
System.out.println("Done");
```

07_Exceptions/…/example1/Main.java

```
<terminated> Main (24) [Java Application] C:\Java\jdk-11.0.1\bin\javaw.exe (6. lip 2019. 13:54:59)
12
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
        at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.base/java.lang.Integer.parseInt(Integer.java:652)
        at java.base/java.lang.Integer.parseInt(Integer.java:770)
        at swu.oopj.exceptions.example1.Main.main(Main.java:8)
```

Stablo iznimki

- Za opis iznimne situacije programski jezik Java koristi primjerke klase *Throwable* ili klasa izvedenih iz nje
  - Podjela u tri glavne kategorije
    - *Error*, *Exception* i *RuntimeException* (detaljnije malo kasnije)
- Neke od tipičnih iznimki
  - *NullPointException*
  - *ClassCastException*
  - *ArithmeticException*
  - *IllegalArgumentException*
  - *IndexOutOfBoundException* s dvije potklase:
    - *ArrayIndexOutOfBoundException*
    - *StringIndexOutOfBoundException*



Klasa *Throwable*

- Klasa *Throwable* omogućava pristup podatcima kao što su:
  - poruka pogreške
  - 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
  - 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

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
  - znamo se „oporaviti” od iznimke

07_Exceptions/…/example2/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<arr.length; i++) {
  try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
  }
  catch(NumberFormatException exc){
        System.out.format("Caught exception at step %d: %s%n",
                i, exc.getMessage());
  }
}
System.out.println("Done");
```

```
12
Caught exception at step 1: For input string: "abc"
15
Done
```

Iznimke različitih tipova

- Mala promjena u prethodnom kodu izaziva neku drugu iznimku
  - pokušava se pristupiti elementu van raspona polja
    - ova iznimka nije uhvaćena

07_Exceptions/…/example3/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
  }
  catch(NumberFormatException exc){
        System.out.format("Caught exception at step %d: %s%n",
              i, exc.getMessage());
  }
}
System.out.println("Done");
```

```
12
Caught exception at step 1: For input string: "abc"
15
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
        at hr.fer.oop.exceptions.example3.Main.main(Main.java:10)
```

Try-catch blokovi s više *catch* dijelova

- Moguće imati više *catch* blokova za jedan *try*

07_Exceptions/…/example4/Main.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
  }
  catch(NumberFormatException exc){
        System.out.format("Caught exception at step %d: %s%n",
                i, exc.getMessage());
  }
  catch(ArrayIndexOutOfBoundsException exc){
        System.out.format("Caught exception at step %d: %s%n",
                i, exc.getMessage());
  }
}
System.out.println("Done");
```

```
12
Caught exception at step 1: For input string: "abc"
15
Caught exception at step 3: Index 3 out of bounds for length 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* operatorom |

07_Exceptions/…/example4/MainMultiCatch.java

```
String[] arr = new String[]{ "12", "abc", "15"};
for(int i=0; i<=arr.length; i++) {
  try{
        int num = Integer.parseInt(arr[i]);
        System.out.println(num);
  }
  catch(NumberFormatException | ArrayIndexOutOfBoundsException exc) {
        System.out.format("Caught exception at step %d: %s%n",
                i, exc.getMessage());
  }
}
System.out.println("Done");
```

```
12
Caught exception at step 1: For input string: "abc"
15
Caught exception at step 3: Index 3 out of bounds for length 3
Done
```

Rukovanje iznimkama u *try-catch* blokovima

- Kod za koji očekujemo da može izazvati iznimku pišemo unutar *try* bloka nakon kojeg slijedi jedan ili više *catch* blokova
- Poredak *catch* blokova je bitan
  - U slučaju iznimke, program izvodi programski kod prvog *catch* bloka „kompatibilnog" s nastalom iznimkom
    - onaj namijenjen za točno tu iznimku ili neku od njenih baznih klasa
  - Ostali catch blokovi se ignoriraju
- Ako ne postoji odgovarajući catch block iznimka se propagira dalje
  - Neuhvaćena iznimka uzrokuje prekid programa (prekid rada JVM-a)
  - Primjer za isprobati: **07_Exceptions/…/example5/ExampleStackStrace.java**
- Podsjetnik: izazivanje iznimke nije poziv metode! Nema povratka na naredbu koja bi slijedila nakon naredbe koja je izazvala iznimku
  - U prethodnom primjerima try-catch je bio unutar petlje. Što bi se dogodilo da je try-catch bio izvan petlje?

Izazivanje iznimke

- Programer i sam može izazvati iznimnu situaciju s *throw objekt* gdje je navedeni objekt neke potklase klase *Exception*
  - npr. izračun opsega trokuta na temelju „stranica” koje ne čine trokut može biti zavaravajući i voditi logičnoj pogrešci, pa zato izazivamo iznimku

```
public static void main(String[] args) {
  try{
       if (perimiter(5, 4, 3) > perimiter(3, 2, 1))
              //do something...
  }
  catch(Exception exc){ System.out.println(exc); }
}
public static int perimiter(int a, int b, int c){
  if (!(a + b > c && a + c > b && b + c > a))
     throw new IllegalArgumentException(
       String.format("%d %d and %d cannot make triangle", a, b, c));
  return a + b + c;
}
```

07_Exceptions/…/example6/ThrowingAnException.java

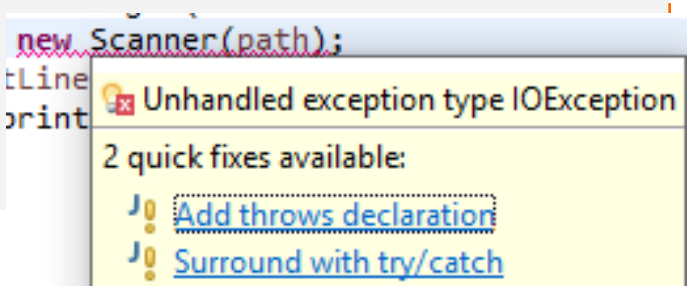
Neprovjeravane iznimke

- Iznimke u prethodnim primjerima spadale su u kategoriju **neprovjeravanih** (engl. *unchecked*) iznimki.
  - Iznimke koje se mogu javiti praktički u svakom koraku izvođenja programa, npr.
    - Neuspješna pretvorba stringa u broja → *NumberFormatException*
    - Pristup nepostojećoj lokaciji polja → IndexOutOfBoundsException
    - Dereferenciranje *null* reference (poziv metode ili pristup članskoj varijabli korištenjem reference koja ima vrijednost *null*) → *NullPointerException*
    - Krivo ukalupljivanje, npr. *downcast* reference tipa *Item* u *Food*, ako referenca pokazuje na objekt tipa *Beverage* → *ClassCastException*
- Za ovaj tip iznimki možemo pisati try-catch blokove, ali ne moramo
- Sve neprovjeravane iznimke su (direktno ili indirektno) izvedene iz *RuntimeException*

Provjeravane (engl. *checked*) iznimke (1/4)

- Pretpostavimo da želimo pročitati neki tekst iz datoteke
  - najjednostavnije obaviti korištenjem klase *Scanner*
- Rješenje bi nalikovalo kodu iz odsječka, ali se ne može prevesti
  - Poziv konstruktora je ispravan, ali…

```
public static void main(String[] args) {
        Path path = Paths.get("src/main/resources/dates.txt");
        Scanner s = new Scanner(path);
        String firstLine = s.nextLine();
        System.out.println(firstLine);
        s.close();
}
```



  - … se javlja pogreška prilikom prevođenja jer se ne obrađuje iznimka koja može nastati u konstruktoru klase *Scanner*

Provjeravane iznimke (2/4)

- Što je to drugačije u konstruktoru klase *Scanner*?

java.util.Scanner.Scanner(Path source) throws IOException

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

source the path to the file to be scanned

Throws:

IOException - if an I/O error occurs opening source

- Eksplicitno se navodi da može izazvati iznimku tipa *IOException* koja nije izvedena iz *RuntimeException* već iz *Exception*
  - Takve iznimke se nazivaju **provjeravane** iznimke (engl. *checked*)
- Programski kod koji poziva metode koje mogu izazvati provjeravanu iznimku mora sadržavati obradu iznimke

Provjeravane iznimke (3/4)

- Moguće rješenje je omotati navedeni kod u try-catch blok

```
public static void main(String[] args) {
  Path path = Paths.get("src/main/resources/dates.txt");
  try {
        Scanner s = new Scanner(path);
        String firstLine = s.nextLine();
        System.out.println(firstLine);
        s.close();
  catch (IOException e) {
        e.printStackTrace();
  }
}
```

07_Exceptions/…/example7/Main.java

Provjeravane iznimke (4/4)

- Alternativa je ne ugraditi obradu iznimke, već deklarirati da se ta iznimka može pojaviti i iz te metode (u ovom slučaju to je *main*)
  - … čime „prenosimo problem” na prethodnog pozivatelja

```
public static void main(String[] args) throws IOException {
        Path path = Paths.get("src/main/resources/dates.txt");
        Scanner s = new Scanner(path);
        String firstLine = s.nextLine();
        System.out.println(firstLine);
        s.close();
}
```

07_Exceptions/…/example7/MainWithThrows.java

- Osim provjeravanih i neprovjeravanih iznimki postoji i treća klasa iznimki – one koje imaju *Error* kao baznu klasu
  - Za njih se obično ne piše kod za obradu iznimki, jer se očekuje da se od njih ne možemo oporaviti

Napomena vezana za *throws* i neprovjeravane iznimke

- U potpisu metode možemo navesti da može izazvati bilo koju vrstu iznimke, pa tako i neprovjeravanu, ali to ne utječe na koncept provjeravanih i neprovjeravanih iznimki.
- Obrada iznimke prilikom poziva neke metode je potrebna samo ako metoda izaziva provjeravanu iznimku

Kod koji treba izvesti neovisno o pojavi iznimke

- Što kad neke programske odsječke treba izvršiti neovisno je li se iznimka pojavila ili ne te neovisno je li bila obrađena (uhvaćena)
  - Npr. treba zatvoriti objekt tipa *Scanner* kojim smo čitali iz datoteke
  - Naivni pristup bi bio pozvati *close* nakon *try-catch* bloka, ali što ako se pojavi neka druga (neuhvaćena) iznimka osim očekivane

```
Path path = Paths.get("src/main/resources/dates.txt");
Scanner s = null;
try {
    s = new Scanner(path);
    String firstLine = s.nextLine();
    LocalDate date = LocalDate.parse(line); //DateTimeParseException?
}
catch (IOException e) {
   e.printStackTrace();
}
s.close();
```

Blok *finally*

- Kodu u bloku *finally* block se uvijek izvodi neovisno o pojavi i obradi iznimke
  - Napomena: Referenca *s* je definirana izvan *try* bloka i može ostati null ako se iznimka pojavi u konstruktoru klase Scanner

```
Path path = Paths.get("src/main/resources/dates.txt");
Scanner s = null;
try {
        s = new Scanner(path);
        String firstLine = s.nextLine();
        LocalDate date = LocalDate.parse(firstLine);
        System.out.format("Day in year: %d%n", date.getDayOfYear());
}
catch (IOException e) { e.printStackTrace(); }
finally {
        System.out.println("This code is always run");
        if (s != null) s.close();
}
```

07_Exceptions/…/example8/ScannerTryCatchFinally.java

Blok *finally*

- Moguće je imati i *try* – *finally* blok, bez *catch* bloka
  - Primjer: **07_Exceptions/…/example9/TryFinallyWithoutCatch.java**
- 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
  - Primjer: **07_Exceptions/…/example9/ExampleWithFinally.java**

Try-with-resources

- Ako klasa implementira sučelje *Closeable* ili *AutoCloseable* (oba definiraju metodu *close*) može se koristiti poseban oblik za *try*
- Za svaki ne-null objekt unutar zagrada kod try, bit će pozvana metoda *close* prilikom izlaska iz try-bloka, bez obzira pojavila se iznimka ili ne
  - Prevodilac generira dodatne try-finally blokove

```
Path path = Paths.get("src/main/resources/dates.txt");
try (Scanner s =  new Scanner(path)) {
        String firstLine = s.nextLine();
        LocalDate date = LocalDate.parse(firstLine);
        System.out.format("Day in year: %d%n", date.getDayOfYear());
}
catch (IOException e) {
        e.printStackTrace();
}
```

07_Exceptions/…/closeable/ScannerTryWithResources.java

Primjer automatskog upravljanja resursima (1/2)

- U prethodnom primjeru nije očito da je metoda *close* pozvana, jer nema ispisa, ali tvrdnju ćemo potkrijepiti na posebnom primjeru
- Neka klasa *Resource* implementira *AutoCloseable* i ispisuje poruke u konstruktoru te prilikom poziva metode *close*

```
package hr.fer.oop.exceptions.closeable;
public class Resource implements AutoCloseable {
	private int i;
	public Resource(int n){
		System.out.println("Creating #" + n);
		i = n;
	}
	@Override
	public void close()  {
		System.out.println("Closing #" + i);
	}
}
```

07_Exceptions/…/closeable/Resource.java

Primjer automatskog upravljanja resursima (2/2)

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

07_Exceptions/…/closeable/Main.java

```
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(System.out);
        }
        finally{
          System.out.println("finally");
        }
}
```

```
Creating #1
Creating #2
Closing #2
Closing #1
Catch...
java.lang.ArithmeticException: / by zero
        at hr.fer.oop.exceptions.closeable.Main.main(Main.java:8)
finally
Main continues...
```

Try-with-resources – iznimke u bloku *close*

- Ako se iznimka dogodi u bloku *close* tada
  - Ako se iznimka nije već ranije pojavila, baca se iznimka iz *close*
  - Ako je iznimka već izazvana ranije, iznimka iz *close* se potiskuje (engl. *suppressed*) – može se dohvatiti s *getSuppressed*

```
public static void main(String[] args) {
  try(ResourceCloseExc r1 = new ResourceCloseExc(1);
      ResourceCloseExc r2 = new ResourceCloseExc(2)){
          int a = 5, b = 0; a = a / b;
          System.out.println("Try block ends.");
   }
   catch (Exception e) {
        System.out.println("Catch..."); e.printStackTrace(System.out);
   }
   finally{ System.out.println("finally"); }
   System.out.println("Mai
```

07_Exceptions/…/closeable/suppressed/*.java

```
Creating #1
Creating #2
Closing and throwing exception #2
Closing and throwing exception #1
Catch...
java.lang.ArithmeticException: / by zero
        at hr.fer.oop.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:8)
        Suppressed: java.lang.RuntimeException: Oh, exception in close...
                at hr.fer.oop.exceptions.closeable.suppressed.ResourceCloseExc.close(ResourceCloseExc.java:12)
                at hr.fer.oop.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:10
        Suppressed: java.lang.RuntimeException: Oh, exception in close...
                at hr.fer.oop.exceptions.closeable.suppressed.ResourceCloseExc.close(ResourceCloseExc.java:12)
                at hr.fer.oop.exceptions.closeable.suppressed.MainExceptionInClose.main(MainExceptionInClose.java:10
finally
Main continues...
```

Omatanje iznimke

- Iznimke se mogu omotati - upakirati u drugu iznimku.
  - Omotane iznimke se mogu dohvatiti metodom *getCause*

07_Exceptions/…/wrap/WrapException.java

```
String s = "a13";
try {
        try  {
                int i = Integer.parseInt(s);
        }
        catch (NumberFormatException exc) {
                System.out.println("Caught NumberFormatException");
                throw new RuntimeException(exc);
        }
}
catch (Exception e) {
        System.out.println("Caught " + e);
        System.out.println("Cause by " + e.getCause());
}
finally {
        System.out.println("Finally 2");
}
```

```
Caught NumberFormatExceptionnull
Caught java.lang.RuntimeException: java.lang.NumberFormatException: For input string: "a13"
Cause by java.lang.NumberFormatException: For input string: "a13"
Finally 2
```

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*

Što odabrati kao baznu klasu za vlastite iznimke? (1/2)

- Sve iznimke su direktno ili indirektno izvedene iz *Throwable*
  - Izvođenje iz klase *Throwable* nepoželjno; to je preopćenita iznimka
- Izvođenje iz grane klase *Error* nema smisla jer se smatra da su to pogreške od kojih se ne možemo oporaviti

→ *Exception* ili *RuntimeException* (provjeravane ili ne)

- Povijesna kontroverza:
  - 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

Što odabrati kao baznu klasu za vlastite iznimke? (2/2)

- Generalni savjet:
  - Ako se očekuje da se program može jednostavno oporaviti od iznimke, neka iznimka bude provjeravana.
  - Ako program(er) ne može napraviti ništa da bi se oporavio od iznimke, preporuča se bacanje neprovjeravane iznimke.
- Razmisliti:
  - Kako bi izgledao kod za obradu takvih iznimki?

- U ovom primjeru odabrane su neprovjeravane iznimke (izvedene iz *RuntimeException*)
  - Izvorni kod primjera nalazi se u

07_Exceptions/hr/fer/oop/exceptions/custom/*