# Obrada i generisanje izuzetaka

U idealnom okruženju ništa loše se ne dešava u toku izvršavanja jedne aplikacije. Fajl koji je potreban aplikaciji je uvek dostupan na fajl sistemu, ne dolazi do problema sa mrežnom konekcijom i Java virtualna mašina nikada ne ostaje bez memorije kada je potrebno da instancira objekat.

Za razliku od ovoga, stvarnost je malo drugačija. Aplikacija često pokušava da otvori nepostojeći fajl, dobavi određeni podatak preko mreže koja ne funkcioniše ili potraži još memorije koju JVM ne može da isporuči. Naš cilj je da napišemo kod koji će na pravi način odgovoriti ovim vanrednim situacijama koje odstupaju od normalnog izvršavanja programa. Baš zbog toga se ovakve situacije i nazivaju *izuzeci*.

## Razumevanje pojma *izuzetak*

Izuzetak reprezentuje grešku koja se može dogoditi u toku izvršavanja programa (*runtime*). Greška je odstupanje od normalnog ponašanja aplikacije. Izuzetak je u stvari događaj koji nastaje u toku izvršavanja programa kada dođe do greške i koji remeti normalni tok odvijanja programa.

Uzrok izuzetka mogu biti različiti problemi. Kao što smo pomenuli, moguće je da aplikacija zahteva određeni fajl sa fajl sistema koji ne postoji. U ovom slučaju nije napravljena nikakva greška u kodu, osim što je programer uzeo u obzir da će potrebni fajl uvek postojati, što je loša praksa. Pored ovoga, izuzetak se može javiti i zbog loše napisanog koda. Pogledajmo sledeći primer koji ilustruje kod koji je nedovoljno dobro napisan, pa zbog toga sadrži grešku koja dovodi do pojave izuzetka.

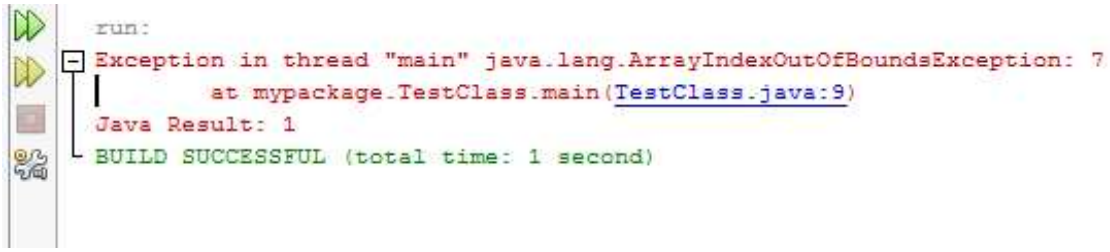
```
public class TestClass
{
      public static void main(String[] args)
      {
            int[] nums = new int[4];
            nums[7] = 10;     // ovde se javlja izuzetak
      }
}
```

U gornjem kodu pokušavamo da inicijalizujemo vrednost člana niza na poziciji 7, ali niz `nums` ima samo četiri člana. Zbog toga će gornji kod definitivno izazvati izuzetak.

## Posledice pojave izuzetka

Izuzetak se može javiti u okviru metoda. Ako malo razmislite, to je logično. Ukoliko nešto nije dobro definisano, prilikom kreiranja komponenti program se neće ni prevesti. Kada se greška pojavi u toku izvršavanja jedne metode, kreira se objekat koji se predaje izvršnom sistemu. Ovaj objekat naziva se objekat izuzetka (*exception object*) i on sadrži informacije o samoj grešci, koje obuhvataju tip greške i stanje programa u trenutku kada se greška dogodila. Proces kreiranja objekta izuzetka i predaja ovog objekta izvršnom sistemu naziva se izbacivanje izuzetka (*throwing an exception*).

Nakon što metoda u kojoj je došlo do greške izbaci izuzetak, izvršni sistem pokušava da pronađe deo koda koji će izvršiti obradu nastalog izuzetka. U hijerarhijskom ustrojstvu metoda, koji se inače naziva *stack poziva* (call stack), pretražuje se ona metoda koja sadrži kod za rukovanje nastalim izuzetkom, odnosno njegovo *hvatanje*. Ukoliko se ni u jednoj metodi ne pronađe kod za obradu nastalog izuzetka, izuzetak završava u virtualnoj mašini. Problem je što JVM-a u ovoj situaciji prekida izvršavanje aplikacije i prikazuje poruku izuzetka, odnosno greške. Ukoliko pokrenemo kod iz gornjeg primera, možemo videti proizvedenu grešku:




*Slika 24.1 Posledica neobrađenog exceptiona*

Ovo nikako nije nešto što biste voleli da se dogodi korisniku koji koristi vašu aplikaciju. Iako su poruke grešaka koje postoje u okviru izuzetaka korisne u toku razvoja aplikacije, ovakvi scenariji su nedopustivi u finalnoj aplikaciji. Zato se pribegava tehnici koja se naziva „hvatanje izuzetaka".

## Obrada izuzetaka

Hvatanje izuzetaka odnosno njihova obrada postiže se korišćenjem blokova **try-catch**. Pogledajmo kako bismo izvršili *hvatanje* izuzetka u prethodnom primeru.

```
public static void main(String[] args)
{
	int nums[] = new int[4];
	try	// blok koda u kome očekujemo grešku
	{
		nums[7] = 10;
	}
	catch(Exception exc)	/* blok koda za obradu greške (parametar
	Exception je obavezan) */
	{
		System.out.println("Index out-of-bounds!");
	}
}
```

Na ovaj način, neće doći do prekida izvršavanja aplikacije, već ćemo na izlazu dobiti poruku:

```
Index out-of-bounds!
```

U primeru prepoznajemo dve ključne reči **try** i **catch**. Ove ključne reči identifikuju blokove koji se tiču rukovanja izuzetkom. Prvi blok koji je inače definisan i kao **try** blok sadrži kod u kome očekujemo grešku, dok drugi blok predstavlja blok u kome se nalazi kod koji će tu grešku obraditi.

Oba bloka su obavezna i nije ih moguće navesti posebno. **Catch** blok obavezno prihvata i parametar **Exception** tipa. **Exception** klasa je osnovna klasa izuzetka i izuzetak ove klase u parametru će biti uvek uhvaćen (kao u primeru).

Pored **try** i **catch** blokova, postoji i treći blok, koji se naziva **finally**. Ovaj blok se izvršava svaki put, bez obzira na ishod prethodnih blokova. U njemu se obično nalaze sistemi za zatvaranje resursa (zatvaranje toka, konekcije ka bazi i slično).

Vratimo se na **catch** blok. Ovaj blok prihvata parametar sa tipom očekivanog izuzetka. U prethodnom primeru će biti uhvaćen izuzetak tipa **Exception**. Ipak, ovo nije dobra praksa u rukovanju izuzecima. Umesto korišćenja generalnog tipa (**Exception**), uvek je bolje hvatati prvo specijalizovane izuzetke, pa tek onda generalne. Praktično, to znači da bi modifikovani prethodni primer izgledao ovako:

```
public static void main(String[] args)
{
	int nums[] = new int[4];
	try
	{
		nums[7] = 10;	// "probijen" indeks niza
	}
	catch(ArrayIndexOutOfBoundsException exc)/* specijalizovani
	izuzetak */
	{
		System.out.println("Index out-of-bounds!");
	}
}
```

Ovaj primer uhvatiće specijalizovani izuzetak „probijanja" indeksa, ali ne i druge izuzetke, tako da, ako bi se u **try** bloku našla i linija koja izbacuje drugi izuzetak, došlo bi do situacije neuhvaćenog izuzetka, i samim tim i greške u izvršenju programa:

```
public static void main(String[] args)
{
	int nums[] = new int[4];
	try
	{
		double variable = 5/0;  /* deljenje nulom (neće biti uhvaćen
		izuzetak) */

		nums[7] = 10;
	}
	catch(ArrayIndexOutOfBoundsException exc)
	{
		System.out.println("Index out-of-bounds!");
	}
}
```

U gornjem primeru dodali smo još jednu liniju u kojoj pokušavamo da izvršimo deljenje nulom, što naravno nije moguće i izaziva grešku. Ova greška će izazvati izuzetak koji neće biti uhvaćen, pošto je u **catch** bloku naveden kao parametar specijalizovan tip izuzetka. Zbog toga na izlazu dobijamo:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
mypackage.TestClass.main
```

Da bi ovaj primer bio pravilan, potrebno je naknadno uhvatiti i drugi očekivani izuzetak. Takođe, poželjno je da se na kraju liste očekivanih izuzetaka pojavi i generalni izuzetak klase **`Exception`**. Od verzije Java 7, ovo je znatno olakšano, tako što je omogućeno hvatanje višestrukih izuzetaka u okviru jednog **`catch`** bloka.

```
public static void main(String[] args)
{
	int nums[] = new int[4];
	try
	{
		double variable = 5 / 0;
		nums[7] = 10;
	}

	/* hvatanje višestrukih izuzetaka u 1 catch bloku (pomoću
	operatora | (ili)) */
	catch(ArrayIndexOutOfBoundsException | ArithmeticException e)
	{
		System.out.println(e);
	}

	catch(Exception e)	// generalni izuzetak (na kraju)
	{
		System.out.println(e);
	}
}
```

U gornjem primeru vidimo na koji način možemo da izvršimo hvatanje višestrukih izuzetaka u okviru jednog **`catch`** bloka upotrebom karaktera „**|**". Pravilo je da se izuzeci uvek hvataju od specifičnog ka globalnom, zbog čega se na kraju liste izuzetaka nalazi i blok koji hvata generalni izuzetak. On će se aktivirati ukoliko izuzetak ne spada ni u jednu grupu specijalizovanih izuzetaka.

## Korišćenje bloka `finally`

Ponekad želite da definišete blok koda koji će se izvršiti kada **`try/catch`** blokovi završe sa svojom logikom. Na primer, kada dođe do pojave izuzetka u metodi koja rukuje mrežnom konekcijom, doći će do toga da konekcija ostane otvorena, što može izazvati razne probleme. Ovakav problem lako se može rešiti upotrebom **`finally`** bloka, koji se može iskoristiti da se u okviru njega reše slične situacije.

Prethodni primer sa dodatkom **`finally`** bloka bi izgledao ovako:

```
public static void main(String[] args)
{
	int nums[] = new int[4];
	try
	{
		double variable = 5 / 0;
		nums[7] = 10;
	}
	catch (ArrayIndexOutOfBoundsException | ArithmeticException e)
	{
		System.out.println(e);
```

```
        }
        catch (Exception e)
        {
              System.out.println(e);
        }
        finally     // uvek će se izvršiti!
        {
              System.out.println("Leaving try.");
        }
  }
```

**Finally** blok izvršiće se u bukvalno svakoj situaciji kada izvršavanje programa napusti **try/catch** blok.

## ARM (Automatic Resource Management)

Od verzije Java 7 dostupna je jedna veoma korisna olakšica kada je u pitanju oslobađanje resursa i korišćenja **finally** bloka. Ova funkcionalnost je naročito korisna prilikom rada sa tokovima ili konekcijama, kada je nakon završetka određenih operacija potrebno osloboditi zauzete resurse. To je naravno moguće uraditi i u okviru **finally** bloka, koji se do pojave ove funkcionalnosti i koristio u te svrhe. Ova funkcionalnost predstavlja pandan *using* direktivi .NET razvojnog okruženja. Pogledajmo kako se ova funkcionalnost koristi u praksi.

Na ovaj način je moralo da se vrši zatvaranje toka u verzijama pre Jave 7.

```
public static void main(String[] args)
{
    String myText = "Hello World";
    FileOutputStream fs = null;
    try // pre Java 7
    {
        fs = new FileOutputStream("myFile.txt");
        fs.write(myText.getBytes());
    }
    catch(IOException exc)
    {
        System.out.println(exc);
    }
    finally
    {
        try // pre Java 7
        {
            if(fs != null)
                fs.close();
        }
        catch(IOException ex)
        {
            System.out.println(ex);
        }
    }
}
```

Upotrebom *Automatic Resource Management* dovoljno je uraditi ovo:

```
        public static void main(String[] args)
        {
            String myText = "Hello World";

            // moguće je navesti i više inicijalizacija resursa
            try(FileOutputStream fs = new FileOutputStream("myFile.txt"))
            {
                fs.write(myText.getBytes());
            }
            catch(IOException exc)
            {
                System.out.println(exc);
            }
            /* na kraju se vrši automatsko oslobađanje resursa koji su
    definisani u try zagradama */
        }
```

Primećujemo da smo inicijalizaciju toka izvršili u okviru zagrada **try** bloka. Na ovaj način će doći do automatskog oslobađanja resursa koji su definisani u zagradama. Takođe je moguće navesti i više inicijalizacija resursa. Ova novina predstavlja veoma korisnu mogućnost i elegantno rešenje za oslobađanje resursa. Prilikom rada sa tokovima u narednom modulu ova funkcionalnost biće više nego korisna.

## Generisanje i izbacivanje izuzetaka

Do sada smo videli na koji način možemo *hvatati* izuzetke u programskom jeziku Java. Međutim, da bi neki izuzetak bio uhvaćen, prethodno mora biti generisan od strane nekog dela koda. Bilo koji kod može generisati izuzetak: korisnički kod, kod iz Java paketa koji je napisan od strane nekih drugih programera ili kod ugrađenih klasa. Bez obzira ko vrši generisanje i izbacivanje izuzetka, to radi upotrebom ključne reči **throw**.

Sve metode moraju koristiti **throw** naredbu kako bi generisale i izbacile izuzetak. Naredba **throw** zahteva jedan argument, i on mora biti instanca klase koja nasleđuje **Throwable** klasu. Ovo znači da makar jedna od roditeljskih klasa u lancu mora biti dete **Throwable** klase. Evo jednog primera izbacivanja izuzetka:

```
throw someThrowableObject;
```

Pogledajmo sada kako možemo proizvesti neki izuzetak u okviru metode:

```
public class Main
{
    // metoda generiše izuzetak tipa Exception
    static void errorMethod() throws Exception
    {
        Exception exc = new Exception(); /* neparametrizovani
konstruktor */

        throw exc;
    }
```

```
        public static void main(String[] args) throws Exception
        {
            errorMethod();  // ovde se izuzetak „ispaljuje“
        }
    }
```

Ovaj primer rezultira greškom uzrokovanom **Exception**-om. U okviru metode errorMethod izvršeno je instanciranje objekta exc tipa **Exception** koji će biti izbačen prilikom pozivanja ove metode.

**Exception** konstruktor može biti neparametrizovan (kao u prethodnom primeru) i parametrizovan, kada se kao parametar prosleđuje tekst poruke o izuzetku. Takođe, praksa je da se izuzetak instancira i izbacuje anonimno:

```
static void errorMethod() throws Exception
{
    throw new Exception("Error message.");
}
```

U ozbiljnijoj arhitekturi projekta kreiraju se i namenske klase izuzeci čije će karakteristike odgovarati aktuelnom sistemu. U tom slučaju nasleđuje se odgovarajuće dete **Throwable** klase. Superklasa svih izuzetaka je klasa **Throwable**, i klasa **Exception** nasleđuje klasu **Throwable**. Klasu **Throwable** nasleđuju i klase koje predstavljaju proveravane i neproveravane izuzetke (**Exception** i **Error**), pa je iz tog razloga za izuzetak koji ćemo kreirati u kodu ispod prikladnije koristiti klasu **Exception.**

Recimo da želimo da napravimo korisnički definisan izuzetak za preveliki broj:

```
    /* korisnički definisana klasa (u posebnom fajlu TooBigNumber.java)
*/
    public class TooBigNumber extends Exception
    { }

    public class Main
    {
        // metoda za proveru prosleđenog broja
        static void checkNumber(int num) throws TooBigNumber
        {
            if(num > 100)   // preveliki broj
                throw new TooBigNumber();

        public static void main(String[] args)
        {
            try
            {
                    // poziva checkNumber() metodu za broj veći od 100
                checkNumber(110);
            }
            catch(TooBigNumber ex)
            {
                System.out.println(ex);
            }
        }
    }
```

U gornjem kodu kreirali smo korisnički definisanu klasu **ToBigNumber**, koja će predstavljati izuzetak. Ova klasa, da bi predstavljala izuzetak, mora da nasledi klasu **Throwable** ili neku klasu koja već nasleđuje ovu klasu. Već je rečeno da gornjem primeru najviše odgovara upotreba klase **Exception**.

Zatim smo napravili metodu za proveru prosleđenog broja – **checkNumber**. Ova metoda kao parametar prima jednu celobrojnu vrednost i, ukoliko je ona veća od 100, doći će do generisanja i izbacivanja izuzetka klase **ToBigNumber**. Zatim smo iz **main** metode izvršili pozivanje **checkNumber** metode prosledivši joj broj veći od 100. Ovo će izazvati generisanje izuzetka. Na izlazu ćemo dobiti naziv paketa i klase izuzetka:

```
myjavaprogram.ToBigNumber
```

Da bismo definisali neku razumnu poruku, prilikom hvatanja izuzetka u klasi izuzetka je moguće izvršiti reimplementaciju nasleđenih metoda. Na primer, metode **toString**:

```
public class ToBigNumber extends Exception
{
    @Override
    public String toString()
    {
        return "The number is to big!";
    }
}
```

Možemo reimplementirati i metodu **getMessage**:

```
@Override
public String getMessage()
{
    return "The number is to big!";
}
```

U prethodnim primerima ste verovatno primetili da je neophodno u metodi koja je moguće da proizvede izuzetak navesti dodatak u potpisu: **throws**, nakon čega sledi naziv klase izuzetka. Ovim se naglašava koje izuzetke metoda može da izbaci. Bez ovoga dodatka, okruženje (i prevodilac) bi prijavilo grešku. Razlog tome je što nasleđivanje **Exception** klase podrazumeva implementaciju proverenog izuzetka. Provereni izuzetak zahteva ekstenziju na metodi o pojavi izuzetka unutar nje, kao i obavezno hvatanje izuzetka korišćenjem **try/catch** blokova. Osim proveravanih, postoje i neproveravani (*checked* i *unchecked*) izuzeci. Neproveravani izuzeci ne podrazumevaju obavezne **try/catch** blokove i **throws** deklaracije.
Razlog postojanja ove razlike u izuzecima je taj što neki izuzeci mogu biti očekivani (na primer, nepostojanje fajla), dok neke ne možemo predvideti (nedostatak memorije, prezauzetost procesora i sl).

Da bismo izuzetak deklarisali kao neproveravani, potrebno je da nasledimo klasu **Error**:

```
// neproveravani izuzetak (ne podrazumeva try/catch i throws)
public class TooBigNumber extends Error { }

public class Main
{
    static void checkNumber(int num)
```

```
        {
            if(num > 100)
                throw new TooBigNumber();
        }

        public static void main(String[] args)
        {
            checkNumber(110);
        }
    }
```

## Rezime

- Izuzetak je događaj koji nastaje u toku izvršavanja programa kada dođe do greške, i koji remeti normalni tok odvijanja programa.
- Kada se greška pojavi u toku izvršavanja jedne metode, kreira se objekat koji se predaje izvršnom sistemu i koji se naziva objekat izuzetka (*exception object*); on sadrži informacije o samoj grešci, koje obuhvataju tip greške i stanje programa u trenutku kada se greška dogodila.
- Proces kreiranja objekta izuzetka i predaja ovog objekta izvršnom sistemu naziva se izbacivanje izuzetka (*throwing an exception*).
- Proces koji definiše kod koji će reagovati na određeni izuzetak naziva se obrada ili hvatanje izuzetka.
- *Hvatanje* izuzetka odnosno njegova obrada postiže se korišćenjem blokova **`try-catch`**.
- **`Blok finally`** definiše kod koji će se izvršiti kada **`try/catch`** blokovi završe sa svojom logikom.
- Proces kreiranja i emitovanja izuzetka naziva se *izbacivanje* (*throwing*) izuzetka.
- Sve metode moraju koristiti **`throw`** naredbu kako bi generisale i izbacile izuzetak.

## Primeri:

### Primer 1:

```
public class Main
{
    public static void main(String[] args)
    {
        int i = 12;

        /*telo for-petlje sadrži try-catch blok (izbacivanje izuzetka ne uzrokuje prekid petlje)*/
        for(int j = 3; j >= -2; j--)
            try
            {
               System.out.println("Uslo se u try: i=" + i + ", j=" + j);
                System.out.println("i/j = " + i / j);
                System.out.println("kraj try\n");
            }
```

```
            catch(ArithmeticException e)
            {
              System.out.println("Uhvacen izuzetak: deljenje nulom!\n");
            }

        System.out.println("posle catch");
    }
}
```

**Primer 2:**

```
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
import java.util.Scanner;

/*U primeru je korišćen try-with-resources koji deklarise 2 resursa: sc i
sc1. Deklaracije resursa razdvajaju se tacka-zarezom (;). 1. konstruktor
klase Scanner može izbaciti izuzetak tipa FileNotFoundException, a 2.
IOException. FileNotFoundException je podklasa od IOException. U slučaju
postojanja hijearhije klasa izuzetaka koji mogu biti izbačeni, unutar try-
bloka izvršava se samo 1. catch-blok, koji može uhvatiti izbačeni izuzetak.
Kada nema fajla ulaz.txt, biće izvrsen samo catch koji hvata
FileNotFoundException. Da nema tog catch-bloka, izuzetak bi bio uhvaćen
catch-blokom koji hvata IOException izuzetke (izuzetke tipa IOException i
tipa podklase od IOException). Takođe, u slučaju postojanja hijerarhije klasa
izuzetaka, neophodno je navesti najpre catch koji rukuje "najuzim" tipom
izuzetka, dok se catch za "najsiri" tip navodi na kraju. (Navođenje prvo
catch za IOException, pa tek onda za FileNotFoundException dovelo bi do
greške pri kompajliranju.)*/
public class Main
{
    public static void main(String[] args)
    {
        // otvaranje fajla za citanje:
        //  stari način
        //  novi način (Java 7) - objašnjeno u nastavnoj jedinici 27
        try(Scanner sc = new Scanner(new File("ulaz.txt"));
            Scanner sc1 = new Scanner(Paths.get("ulaz1.txt"),
StandardCharsets.UTF_8.name()))
        {
            /*... ovde možemo da čitamo i obrađujemo podatke iz fajlova
ulaz.txt i ulaz1.txt ...*/
        }
        /* ako se ovaj catch-blok zakomentarise, izvršiće se donji (koji
hvata IOException) */
        catch(FileNotFoundException e)
        {
            // System.err predstavlja standardni izlaz za grešku (ekran)
            System.err.println("Uslo se u catch za FileNotFoundException");

            e.printStackTrace();   /* podrazumevano vrši ispis u tok
System.err */
        }
        catch(IOException e)
```

```
        {
            System.out.println("Uslo se u catch za IOException.");

            /* metodu printStackTrace moguće je zadati izlazni tok u koji će
vršiti ispis */
            e.printStackTrace(System.out);
        }
    }
}
```

## Vežbe:

### *Vežba 1*

Postojeću aplikaciju potrebno je obezbediti tako da ne prijavljuje grešku.

```
public class Main
{
    static int calculate(int a, int b, String op)
    {
        if(op.equals("+"))
            return a + b;
        if(op.equals("-"))
            return a - b;
        if(op.equals("/"))
            return a / b;
        if(op.equals("*"))
            return a * b;

        return 0;
    }

    public static void main(String[] args)
    {
        int x = calculate(5, 0, "/");
        System.out.println(x);
    }
}
```

### *Rešenje:*

```
public class Main
{
    static int calculate(int a, int b, String op)
    {
        if(op.equals("+"))
            return a + b;
        if(op.equals("-"))
            return a - b;
        if(op.equals("/"))
            return a / b;
        if(op.equals("*"))
            return a * b;
```

```
        return 0;
    }

    public static void main(String[] args)
    {
        int x = 0;
        try
        {
            x = calculate(5, 0, "/");
            System.out.println(x);
        }
        catch(ArithmeticException ex)
        {
            System.out.println("Rezultat nije moguce izracunati.");
        }
    }
}
```

***Vežba 2***

Postoji sledeća klasa `User`:

```
public class User
{
    public int id;
    public String firstName;
    public String lastName;
    public String email;

    public User(int id, String firstName, String lastName, String email)
    {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.email = email;
    }
}
```

Potrebno je kreirati klasne izuzetke za nepravilan unos ID-ja, imena, prezimena i E-maila.

Potrebno je implementirati sistem provere u konstruktor klase `User` tako da ukoliko je ID veći od 100, bude izbačen **`InvalidIdException`**, ako su firstName, lastName i E-mail polja prazna, bude izbačen **`InvalidFirstNameException`**, **`InvalidLastNameException`** ili **`InvalidEmailException`**.

Potrebno je instancirati ovu klasu u `Main` projektu.

***Rešenje:***

```
class InvalidEmailException extends Exception { }
class InvalidFirstNameException extends Exception { }
class InvalidIdException extends Exception { }
class InvalidLastNameException extends Exception { }

public class User
```

```
{
    public int id;
    public String firstName;
    public String lastName;
    public String email;

    public User(int id, String firstName, String lastName, String email)
throws InvalidIdException, InvalidFirstNameException,
InvalidLastNameException, InvalidEmailException
    {
        if(id > 100)
            throw new InvalidIdException();
        else
            this.id = id;

        if(firstName.equals(""))
            throw new InvalidFirstNameException();
        else
            this.firstName = firstName;

        if(lastName.equals(""))
            throw new InvalidLastNameException();
        else
            this.lastName = lastName;

        if(email.equals(""))
            throw new InvalidEmailException();
        else
            this.email = email;
    }
}

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            User u = new User(10, "Petar", "Petrovic", "petrov@mail.mm");
        }
        catch(InvalidIdException ex)
        {
            System.out.println("Nepravilan ID");
        }
        catch(InvalidFirstNameException ex)
        {
            System.out.println("Nepravilno Ime");
        }
        catch(InvalidLastNameException ex)
        {
            System.out.println("Nepravilno prezime");
        }
        catch(InvalidEmailException ex)
        {
            System.out.println("Nepravilan e-mail");
        }
    }
}
```