Iznimke u Javi

Jednostavni primjeri programskog koda s pogreškama (1/2)

- Ukoliko se izvrši sljedeći programski kod:

```
Scanner unos = null;
int prviBroj = unos.nextInt();
int drugiBroj = unos.nextInt();

System.out.println("Rezultat " + prviBroj + "/"
                + drugiBroj + " = " + prviBroj/drugiBroj);
```

- tijekom izvođenja programa dolazi do pogreške zbog toga što se pozivaju metode nad objektom “unos”, a njegova referenca pokazuje na “null”:

```
Exception in thread "main" java.lang.NullPointerException
at test.ExceptionTest.main(ExceptionTest.java:13)
```

Jednostavni primjeri programskog koda s pogreškama (2/2)

- Kako ne postoji programski kod koji obrađuje tu pogrešku, zbog pogreške se prekida trenutno izvođenje programa
- Detaljnije informacije o tome koja pogreška se dogodila i u kojoj liniji programskog koda ispisuje se u *konzoli*
- Pomoću tih informacija moguće je lakše rekonstruirati problem i korištenjem *debug* načina izvođenja programa otkloniti ga
- Način signaliziranja pogrešaka u Java programima realiziran je **iznimkama**

Iznimke (engl. *Exceptions*)

- Događaji koji nastaju tijekom izvođenja programa i koji narušavaju normalan tijek izvođenja naredbi
- Mogu ih izazvati različiti događaji, od ozbiljnih sklopovskih problema (npr. kvar čvrstog diska), do pogrešaka u samom programu
- Iznimke se predstavljaju u obliku objekata koji sadrže podatke o stanju programa u trenutku nastanka iznimnog događaja
- Ti podaci koriste se za dohvaćanje informacija potrebnih za obradu i ispravljanje pogrešaka u programu
- Proces stvaranja objekta iznimke i izvođenje koda za signalizaciju pogreške naziva se "bacanje iznimke" (engl. *throwing exception*)

Podaci o iznimkama (1/2)

- Ako se izvede sljedeći programski kod:

```
Scanner unos = new Scanner(System.in);
System.out.print("Unesite brojeve : ");
int prviBroj = unos.nextInt();
int drugiBroj = unos.nextInt();
System.out.println("Rezultat " + prviBroj + "/" + drugiBroj
                               + " = " + prviBroj/drugiBroj);
```

- I unesu brojevi npr. "5" i "0", generira se sljedeća iznimka:

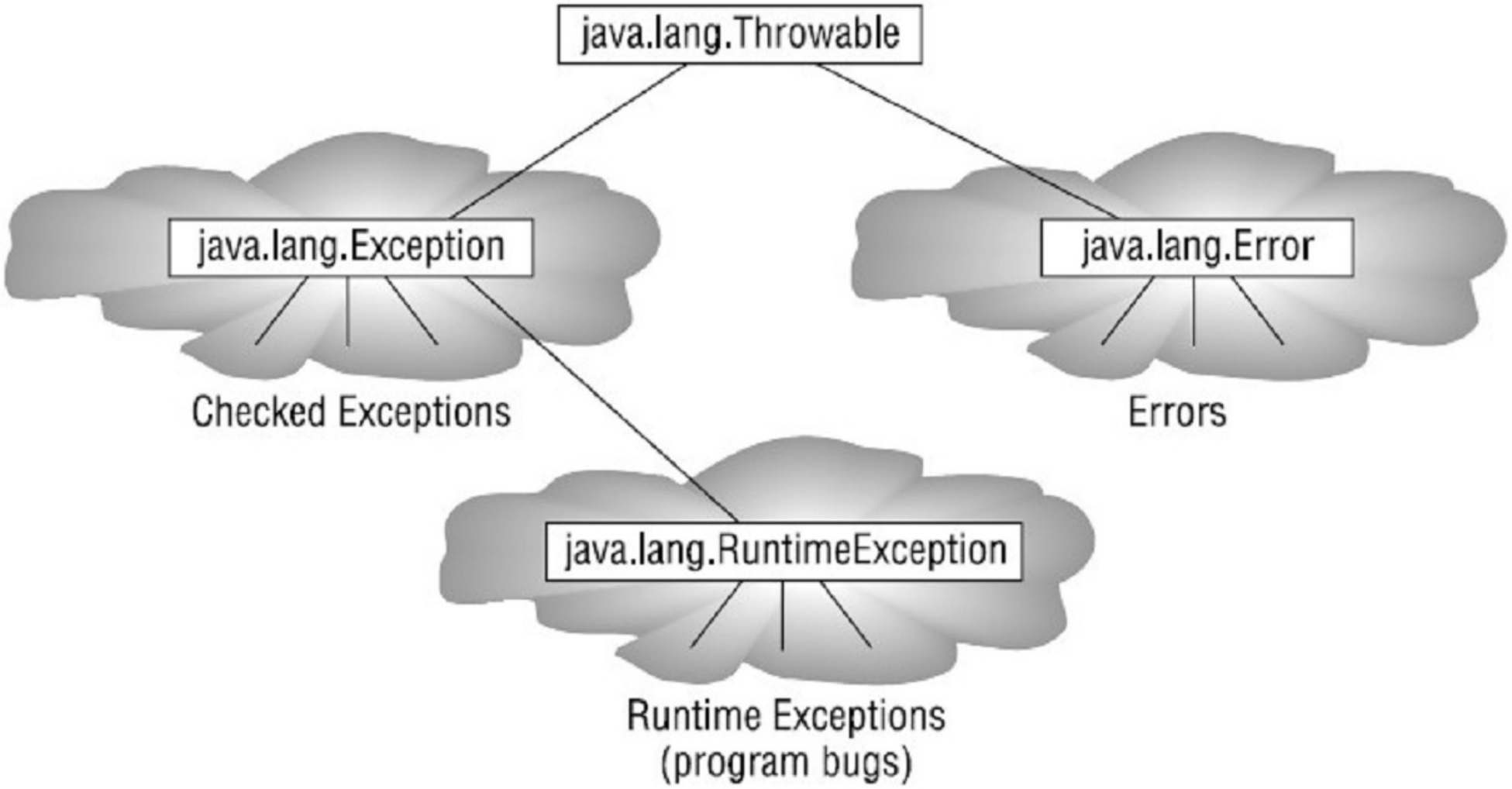
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at test.ExceptionTest.main(ExceptionTest.java:18)
```

Podaci o iznimkama (2/2)

- Podaci koji su se ispisali u konzoli definiraju sljedeće detalje:
  - "`Exception in thread "main"`" – iznimka se dogodila tijekom izvođenja glavne niti programa
  - "java.lang.ArithmeticException" – klasa iznimke koja se dogodila (sam naziv iznimke često puno govori o vrsti iznimke)
  - "`/ by zero`" – poruka koja se nalazi unutar iznimke i pomoću koje je moguće saznati dodatne detalje o uzroku iznimke
  - "`at test.ExceptionTest.main(ExceptionTest.java:18)`" – naziv metode i broja linije gdje se iznimka dogodila

Klase iznimaka (1/2)

- Postoji veliki broj klasa u Javi koje označavaju iznimke, a one se dijele u četiri skupine:



Klase iznimaka (2/2)

- **Throwable**: klasa koju nasljeđuju sve iznimke u Javi
- **Error**: označavaju ozbiljne pogreške unutar JVM-a i nije predviđeno da se obrađuju unutar programa, već se prosljeđuju JVM-u
- **Exception**: **označene iznimke** (engl. *checked exceptions*), događaju se u slučaju pogrešaka koje nisu prouzročene pogreškama u programiranju (npr. otvaranje neispravnog URL-a, otvaranje datoteke koja ne postoji itd.)
- **RuntimeException**: iznimke kod izvođenja (engl. *runtime exception*), događaju se u slučaju pogrešaka u programiranju (*ClassCastException*, *NullPointerException*, *ArithmeticException*...)

Obrađivanje iznimaka (1/3)

- Za svaku iznimku moguće je napisati programsku rutinu koja će se obavljati u slučaju generiranja pogreške
- U Javi postoje dvije ključne riječi koje označavaju programski kod koji može rezultirati bacanjem iznimke i programska rutina koja se obavlja u slučaju bacanja iznimke: **try** i **catch**:

```
try {
   System.out.print("Unesite brojeve : ");
   int prviBroj = unos.nextInt(); int drugiBroj = unos.nextInt();
   System.out.println("Rezultat " + prviBroj + "/" + drugiBroj
                                  + " = " + prviBroj / drugiBroj);
} catch(ArithmeticException ex) {
   System.out.println("Dogodila se aritmetička pogreška prilikom
                              izračunavanja: " + ex.getMessage());
   System.out.println("Radi se o sljedećoj iznimci: ");
   ex.printStackTrace();
}
```

Obrađivanje iznimaka (2/3)

- Navedeni programski kod unutar *catch* bloka za obradu pogreške ispisati će sljedeće informacije:

```
Dogodila se aritmetička pogreška prilikom izračunavanja: / by zero
Radi se o sljedećoj iznimci:
java.lang.ArithmeticException: / by zero
at test.ExceptionTest.main(ExceptionTest.java:23)
```

- Svaki objekt iznimke omogućava pozivanje metoda za dohvaćanje dodatnih informacija o iznimkama:
  - **getMessage()** - dohvaća poruku koja je vezana uz iznimku
  - **printStackTrace()** - ispisuje "stazu stoga" iznimke koja sadrži informacije o lokaciji generiranja iznimke

Obrađivanje iznimaka (3/3)

- Osim toga moguće je provjeravati i je li dostupan neki resurs kao što je datoteka:

```
String filePath = "C:/fajla.txt";
File datoteka = new File(filePath);
FileReader citac = null;
try {
   citac = new FileReader(datoteka);
   System.out.println("Ovo se ne izvrši ako se baci iznimka!");
} catch (FileNotFoundException ex) {
   System.out.println("Datoteka '" + filePath + "' ne postoji!");
   System.exit(1);
}
```

- Ako datoteka ne postoji, ispisuje se poruka:

"Datoteka 'C:/fajla.txt' ne postoji!"

Try-catch-finally blokovi (1/2)

- Označavaju dijelove programskih kodova koji su vezani uz obrađivanje pogreške:
  - ***Try*** blok: u njega je potrebno napisati programski kod koji može baciti iznimku
  - ***Catch*** blok: u njemu se moraju nalaziti naredbe koje se izvode u slučaju generiranja iznimke u *try* bloku
  - ***Finally*** blok: programski kod koji se nalazi unutar njega se uvijek izvodi, bez obzira dogodila se iznimka ili ne (često je potrebno zatvoriti datoteku ili konekciju prema bazi podataka, bez obzira na to je li programski kod završio uspješno ili se dogodila pogreška)

Try-catch-finally blokovi (2/2)

- Opći oblik *try-catch-finally* bloka:

```
try {
   //pozivanje metoda koje bacaju iznimke
} catch (tip_iznimke1 identifikator1) {
   //programski kod za analizu iznimke1
} catch (tip_iznimke2 identifikator2) {
   //programski kod za analizu iznimke2 ...
} finally {
   //programski kod koji se obavlja nakon analize
   // iznimaka
}
```

- U Javi 7 dodan je "*multicatch*" blok koji može "hvatati" više od jedne vrste iznimaka

Catch blok koji hvata više iznimki (1/3)

- U slučaju kad se unutar jednog ***try*** bloka nalazi programski kod koji može baciti veći broj iznimki, npr:

```
public static void metoda(int i) throws InvalidPasswordException,
                 IllegalDataException, DatabaseUnavailableException
{
   if (i < 0) {
      throw new InvalidPasswordException();
   }
   else if (i == 0) {
      throw new IllegalDataException();
   }
   else {
      throw new DatabaseUnavailableException();
   }
}
```

Catch blok koji hvata više iznimki (2/3)

- Programski kod za obradu takvih pogrešaka izgleda nepotrebno opsežno:

```
try {
   metoda(i);
}
catch(InvalidPasswordException ex) {
   System.out.println("Neispravna lozinka!");
   ex.printStackTrace();
}
catch(IllegalDataException ex) {
   System.out.println("Neipravni podaci");
   ex.printStackTrace();
}
catch(DatabaseUnavailableException ex) {
   System.out.println("Nedostupna baza podataka!");
   ex.printStackTrace();
}
```

Catch blok koji hvata više iznimki (3/3)

- Od Jave 7 uveden je tzv. ***catch*** blok koji omogućava hvatanje više iznimki odjednom (engl. *multicatch block*), kojim se programski kod može sažeti na sljedeći način:

```
try {
   metoda(i);
}
catch(InvalidPasswordException | IllegalDataException |
      DatabaseUnavailableException ex)
{
   System.out.println("Pogreška u radu sa sustavom!");
   System.out.println("Detalji:");
   ex.printStackTrace();
}
```

- Programski kod je poopćen (ista poruka se ispisuje za svaku iznimku), a izgled programskog koda je sažetiji i pregledniji

Označena iznimka i iznimka kod izvođenja (1/2)

- Razlikuju se po tome da označena iznimka mora biti obrađena unutar *try-catch* bloka, dok iznimka kod izvođenja ne mora biti obrađena
- Na primjer, ukoliko se programski kod koji može baciti iznimku ne napiše unutar try-catch bloka, kompajler javlja pogrešku kod prevođenja, npr:

```
String filePath = "C:/fajla.txt";
File datoteka = new File(filePath);
FileReader citac = null;
citac = new FileReader(datoteka);
System.out.println("Ovo se ne izvrši ako se baci iznimka!");
```

- Javlja se pogreška "**Unhandled exception type FileNotFoundException**" kojom se naznačava obveza da se iznimka obradi i tek nakon toga je moguće prevesti i izvršiti programski kod

Označena iznimka i iznimka kod izvođenja (2/2)

- Programski kod koji može generirati iznimku kod izvođenja ne mora biti unutar *try-catch* bloka, međutim, ukoliko se dogodi pogreška, prekida se izvođenje programa
- Na primjer, programski kod:

```
int[] polje = new int[] {1, 2, 3};
System.out.println(polje[5]);
```

- rezultira pogreškom:

Exception in thread "main"

java.lang.ArrayIndexOutOfBoundsException: 5

at iznimke.IznimkaTest2.main(IznimkaTest2.java:10)

- U praksi nije izvedivo obrađivanje svih mogućih iznimaka kod izvođenja, jer bi programski kod bio krajnje nepregledan

Ispisivanje staze stoga (1/2)

- Staza stoga (engl. *stack trace*) sadrži ključne informacije o nastanku iznimke i olakšava ispravljanje pogrešaka
- Te informacije se mogu dohvatiti pozivom metode "printStackTrace()"
- Na primjer, ukoliko sljedeći programski kod pozove metodu "printStackTrace()":

```
try {
   citac = new FileReader(datoteka);
   System.out.println("Ovo se ne izvrši ako se baci iznimka!");
} catch (FileNotFoundException ex) {
   System.out.println("Datoteka '" + filePath + "' ne postoji!");
   ex.printStackTrace();
   System.exit(1);
}
```

Ispisivanje staze stoga (2/2)

- Ispisati će se sljedeće:

```
Datoteka 'C:/fajla.txt' ne postoji!
java.io.FileNotFoundException: C:\fajla.txt (The system cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(Unknown Source)
at java.io.FileReader.<init>(Unknown Source)
at test.ExceptionTest.fileCheck(ExceptionTest.java:41)
at test.ExceptionTest.main(ExceptionTest.java:31)
```

- Staza stoga sadrži sve potrebne informacije za dijagnosticiranje pogrešaka u programu, od naziva metoda i klasa u kojima se pogreška dogodila pa sve do broja linije programskog koda gdje je došlo do pogreške i opisa same pogreške

Bacanje iznimaka

- Osim hvatanja iznimaka koje se događaju tijekom izvođenja programa, za obilježavanje iznimnih događaja moguće je i bacati iznimke (engl. *throw exceptions*)
- Ključna riječ za bacanje iznimki je **throw**
- Prije bacanja same iznimke potrebno je kreirati objekt iznimke i baciti ga korištenjem ključne riječi **throw**
- Na primjer, ako je iznimku kod izvođenja potrebno baciti u ovisnosti o vrijednosti neke varijable, to je moguće napraviti na sljedeći način:

```
if (temperatura > 30) {
   throw new RuntimeException("Previsoka temperatura!");
}
```

- Ulazni parametar konstruktora klase "RuntimeException" može definirati poruku koja je vezana uz samu iznimku

Metode koje bacaju iznimke

- Ako je potrebno u definiciji same metode naznačiti da može baciti iznimku, to je moguće napraviti na sljedeći način, korištenjem ključne riječi **throws**:

  public int metoda() throws FileNotFoundException
- Time je svakom programskom kodu koji poziva definiranu metodu naznačeno da metoda može baciti iznimku, te ako je potrebno, taj programski kod može tu iznimku i obraditi
- Ukoliko metoda baca označenu iznimku, programski kod koji poziva tu metodu može baciti dalje tu iznimku (engl. *rethrow exception*) ili je obraditi
- Ako metoda baca iznimku kod izvođenja, programski kod koji poziva tu metodu ne mora obraditi tu iznimku, niti je baciti dalje, međutim, ako se ona dogodi, prekida izvođenje programa

Kreiranje vlastitih iznimki

- Ponekad nije dovoljno korištenje iznimki koje su uključene u standardni skup Java klasa, već je potrebno kreirati vlastite
- Kod kreiranja vlastitih iznimki najprije je potrebno definirati svojstva koja ta iznimka treba posjedovati
- Svojstva se temelje na dvije osnovne skupine iznimki: označene iznimke ili iznimke kod izvođenja
- Iznimka poprima svojstva označene iznimke ako nasljeđuje klasu **java.lang.Exception**
- Ako se iznimka treba ponašati kao iznimka kod izvođenja, mora nasljeđivati klasu **java.lang.RuntimeException**
- Svaka od klasa mora imati vlastiti skup konstruktora za kreiranje objekata klase iznimaka, a najčešće se taj skup svodi na tri osnovna konstruktora

Primjer kreiranja vlastitih iznimaka (1/2)

- Naziv klase mora detaljnije opisivati iznimni događaj za koji je iznimka vezana, npr.

```
public class InvalidPasswordException extends Exception {
   public InvalidPasswordException(String message) {
      super(message);
   }

   public InvalidPasswordException(Throwable cause) {
      super(cause);
   }

   public InvalidPasswordException(String message, Throwable cause) {
      super(message, cause);
   }
}
```

- Konstruktori pozivaju konstruktore nadklase

Primjer kreiranja vlastitih iznimaka (2/2)

- Svaki konstruktor može definirati poruku vezanu uz samu iznimku, uzročnu iznimku koja je prethodila aktualnoj iznimci ili oboje
- Ukoliko je potrebno baciti navedenu iznimku, prvo je potrebno kreirati objekt te klase (kao i svake druge) i koristiti ključnu riječ ***throw***:

  throw new InvalidPasswordException("Neispravna lozinka!");
- Osim toga moguće je prvo kreirati objekt klase pa tek onda baciti objekt:

  InvalidPasswordException ipe = new InvalidPasswordException("Neispravna lozinka!");

  throw ipe;

Try blok s resursima (1/3)

- Osim na način da se unutar ***try*** bloka definiraju sve naredbe za pristupanje resursima:

```
String filePath = "C:/fajla.txt";
File datoteka = new File(filePath);
FileReader citac = null;
try {
   citac = new FileReader(datoteka);
   char znak = (char) citac.read();
   System.out.println("Pročitani znak je " + znak);
   citac.close();
} catch (FileNotFoundException ex) {
   System.out.println("Datoteka '" + filePath + "' ne postoji!");
   ex.printStackTrace();
} catch (IOException ex) {
   System.out.println("Pogreška kod čitanja!");
   ex.printStackTrace();
}
```

Try blok s resursima (2/3)

- moguće je definirati tzv. ***try*** blok s resursima (engl. *try-with-resources*) u kojem se inicijaliziraju resursi koji će se koristiti:

```
String filePath = "C:/fajla.txt";
File datoteka = new File(filePath);

try(FileReader citac = new FileReader(datoteka)) {
   char znak = (char) citac.read();
   System.out.println("Pročitani znak je " + znak);
}
catch (FileNotFoundException ex) {
   ex.printStackTrace();
}
catch (IOException ex) {
   ex.printStackTrace();
}
```

Try blok s resursima (3/3)

- Programski kod se korištenjem ***try*** bloka s resursima može drastično skratiti
- Osim toga automatski se poziva metoda "close" koja zatvara tokove podataka i datoteke koji se inicijaliziraju unutar ***try*** bloka s resursima
- Ukoliko se prilikom inicijalizacije resursa dogodi pogreška (npr. tražena datoteka ne postoji), izvođenje programa se preusmjerava u *catch* blokove koji obrađuju pogrešku koja se dogodila, bez ulaska u programski kod koji se nalazi unutar samog try bloka
- ***Try*** blok s resursima uveden je u Javu s inačicom 7

Pogreške u aplikacijama – logiranje (1/2)

- Često je potrebno u samom programskom kodu aplikaciji ugraditi naredbe koje služe za zapisivanje događaja u zasebne datoteke koje je nazivaju ***log*** datoteke
- *Log* datoteke su tekstualne datoteke koje se spremaju na lokalno računalo ili poslužitelj i mogu se analizirati kasnije neovisno o radu aplikacije
- Time se omogućava praćenje aktivnosti korisnika u aplikaciji, te pregledavanje staza stoga iznimki koje su se eventualno mogle dogoditi tijekom rada aplikacije
- Na taj način je moguće lakše dijagnosticirati probleme (engl. ***bugs***) u programskom kodu
- Postoji nekoliko gotovih rješenja (Java *library*-a) koje je moguće dodati u vlastiti Java projekt i time koristiti izravno funkcionalnosti *logiranja*

Pogreške u aplikacijama – logiranje (2/2)

- Najaktualnija je biblioteka koja se naziva "Logback"
- "Logback" se konfigurira pomoću XML datoteke i na taj način definiraju pravila za popunjavanje *log* datoteka
- Zapisi u *log* datotekama se dijele na različite razine važnosti i ozbiljnosti:
  - ERROR
  - WARN
  - INFO
  - DEBUG
  - TRACE
- Java naredbe kojima se događaji zapisuju u *log* datoteke izgledaju ovako:

  logger. info(„Korisnik je pokrenuo aplikaciju“);

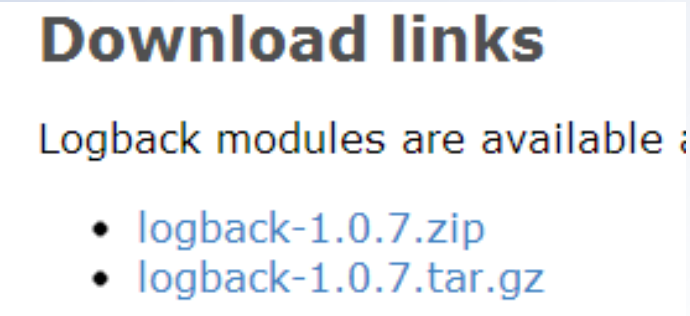
  logger. debug(„Pogreška kod unosa broja!“, iznimka);

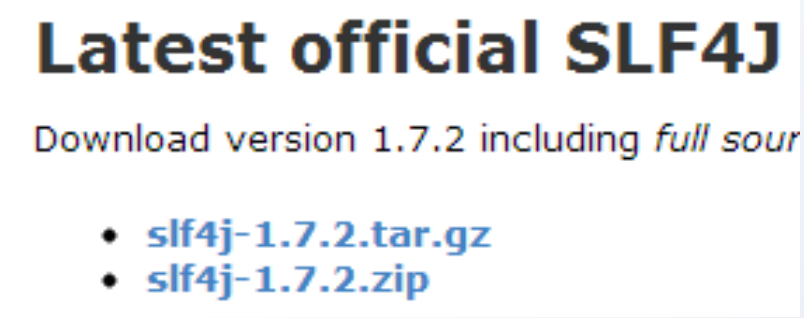
Uključivanje vanjskih biblioteka u Java aplikaciju (1/6)

- Kako bi se proširila funkcionalnost Java aplikacije, često je potrebno s mrežnih stranica preuzeti vanjsku biblioteku i dodati je u *classpath* Java projekta
- Prvi korak u tome predstavlja preuzimanje biblioteka s mrežnih stranica, npr. Logback biblioteke sa stranica **http://logback.qos.ch/**:

Uključivanje vanjskih biblioteka u Java aplikaciju (2/6)

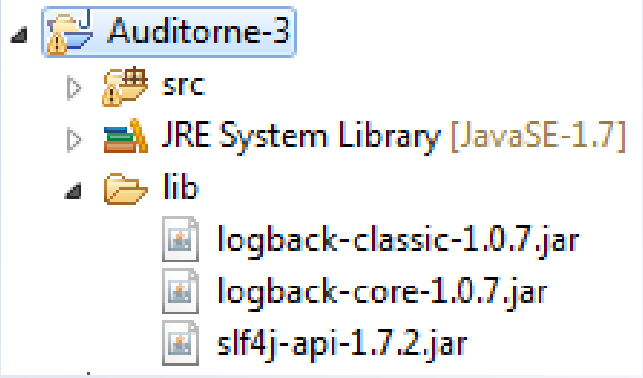
- Biblioteke se najčešće preuzimaju u obliku "ZIP", "TAR.GZ" ili "JAR" (*Java ARchive*) arhive (7.1 MB i 4.1 MB):



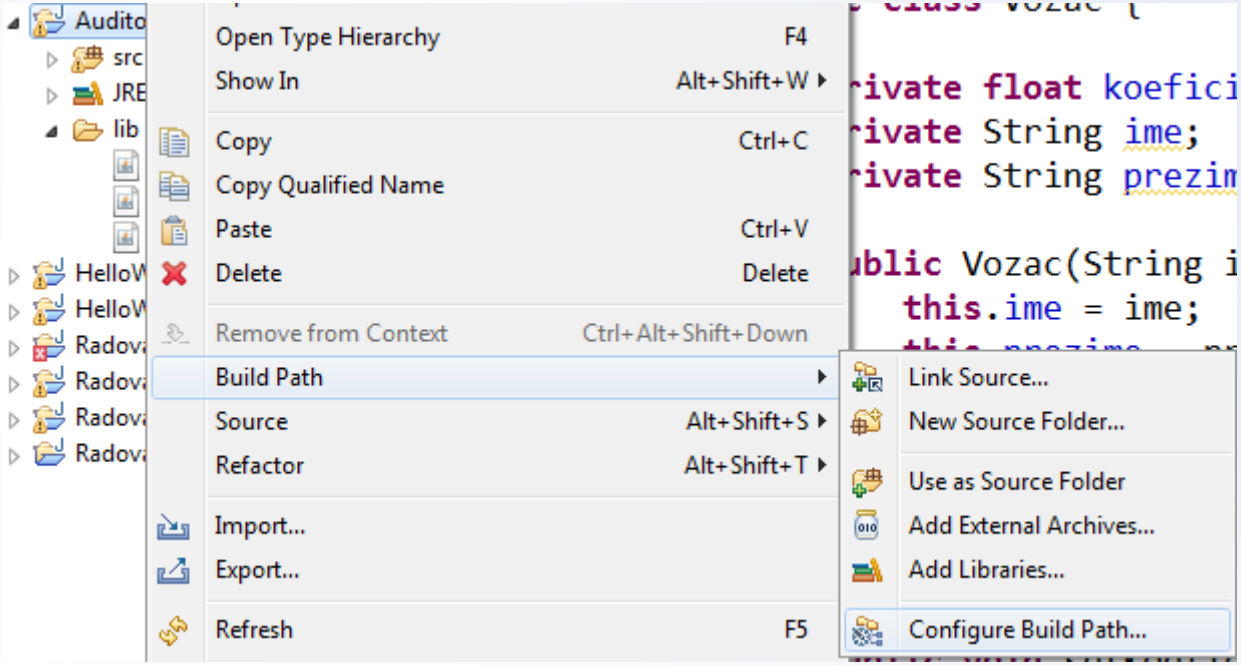


- U *classpath* je moguće dodati cijelu "ZIP" datoteku, ali zbog veličine je dovoljno dodati samo dio te arhive, npr. samo datoteke "logback-classic*.jar", "logback-core*.jar" i "slf4j-api-*.jar"
- Za to je najprije potrebno kreirati "lib" mapu u Java projektu unutar Eclipsea, koja će služiti za pohranjivanje vanjskih biblioteka (kako projekt ne bi ovisio o trenutnoj lokaciji na disku računala, već da se nalazi unutar projekta)

Uključivanje vanjskih biblioteka u Java aplikaciju (3/6)

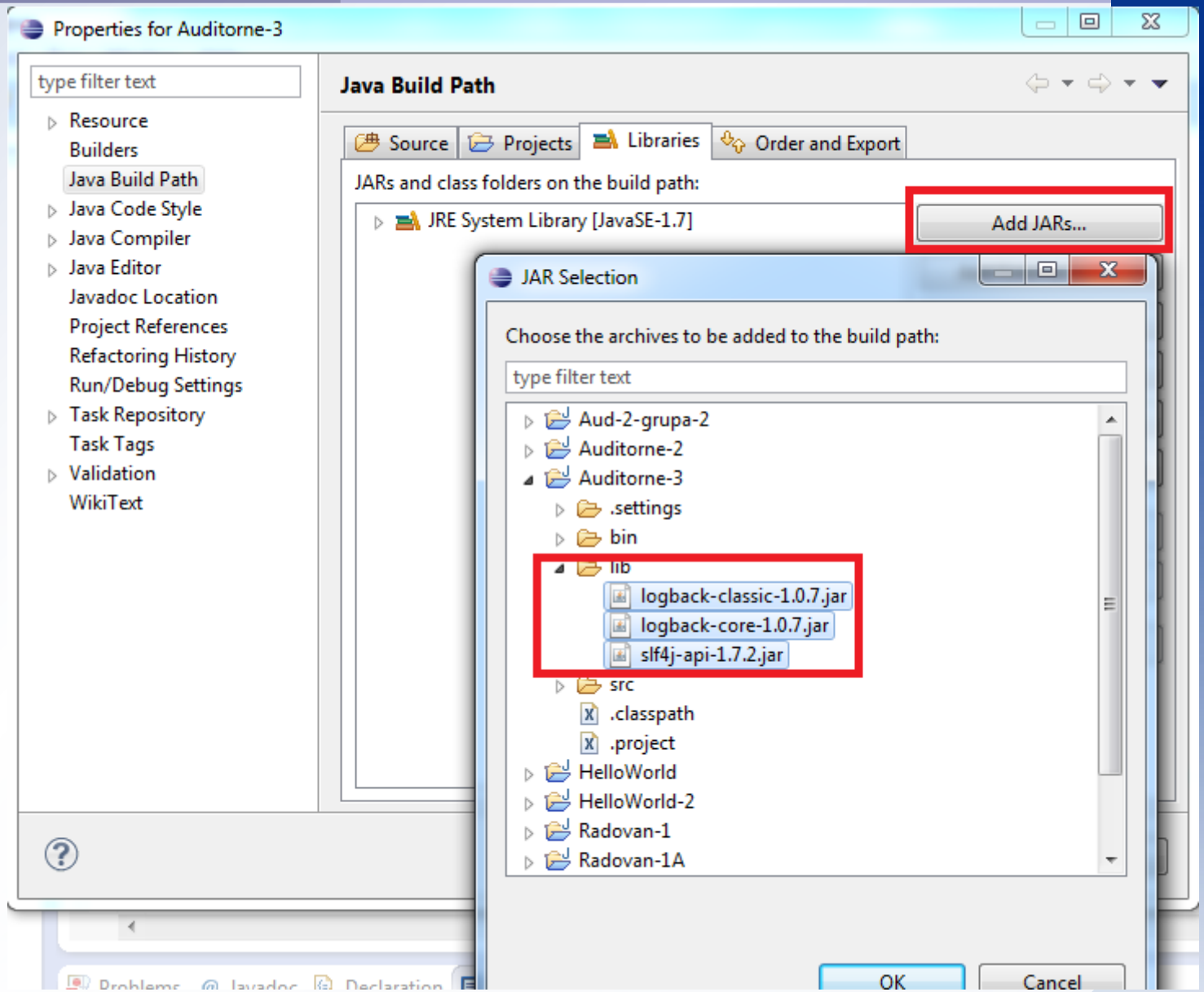


- Dodavanje datoteka u *classpath* projekta obavlja se korištenjem opcije "Configure Build Path...":



Uključivanje vanjskih biblioteka u Java aplikaciju (4/6)

- Odabirom opcije "Add JARs..." na kartici "Libraries" otvara se dijalog za odabir sve tri "JAR" datoteke



Uključivanje vanjskih biblioteka u Java aplikaciju (5/6)

- Slijedi definiranje "Logback" konfiguracije u datoteci "logback.xml" koja se mora nalaziti u "src" mapi:

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <appender name="SIFT" class="ch.qos.logback.classic.sift.SiftingAppender">
    <discriminator>
      <Key>screen</Key>
      <DefaultValue>unknown</DefaultValue>
    </discriminator>
    <sift>
      <appender name="FILE-${screen}" class="ch.qos.logback.core.FileAppender">
        <File>./pogadjanje.log</File>
        <Append>false</Append>
        <layout class="ch.qos.logback.classic.PatternLayout">
          <Pattern>%d [%thread] %level %mdc %logger{35} - %msg%n</Pattern>
        </layout>
      </appender>
    </sift>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="SIFT" />
  </root>
</configuration>
```

Uključivanje vanjskih biblioteka u Java aplikaciju (6/6)

- Na kraju je još potrebno dodati određene dijelove programskog koda u samu aplikaciju i generiranje log datoteka će se obavljati automatski:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
private static final Logger logger = LoggerFactory.getLogger(Glavna.class);
...

do {
        ...
        try {
            provjera(uneseniBroj);
            pogodio = true;
        } catch (PremaliBrojException e) {
            logger.info(e.getMessage());
            System.out.println(e.getMessage());
        } ...
}while(pogodio == false);
```