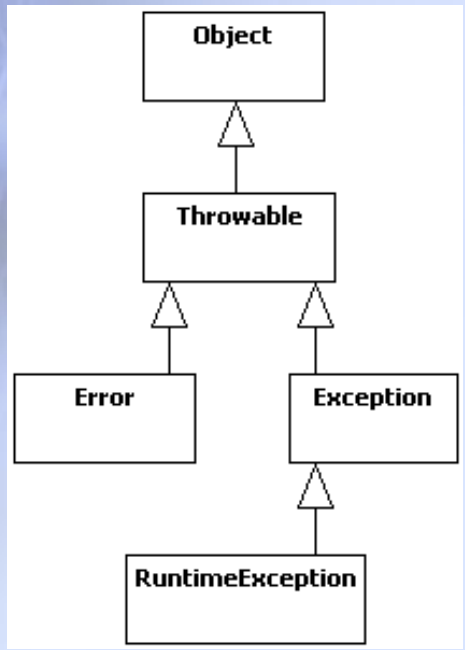
Izuzeci

Programski jezici II

Izuzeci

- Izuzetak je događaj koji se dešava za vrijeme izvršavanja programa koji prekida normalan tok programa (programskih instrukcija)
- Izuzeci u Javi su objekti



- Osnovna klasa svih izuzetaka i grešaka je Throwable klasa

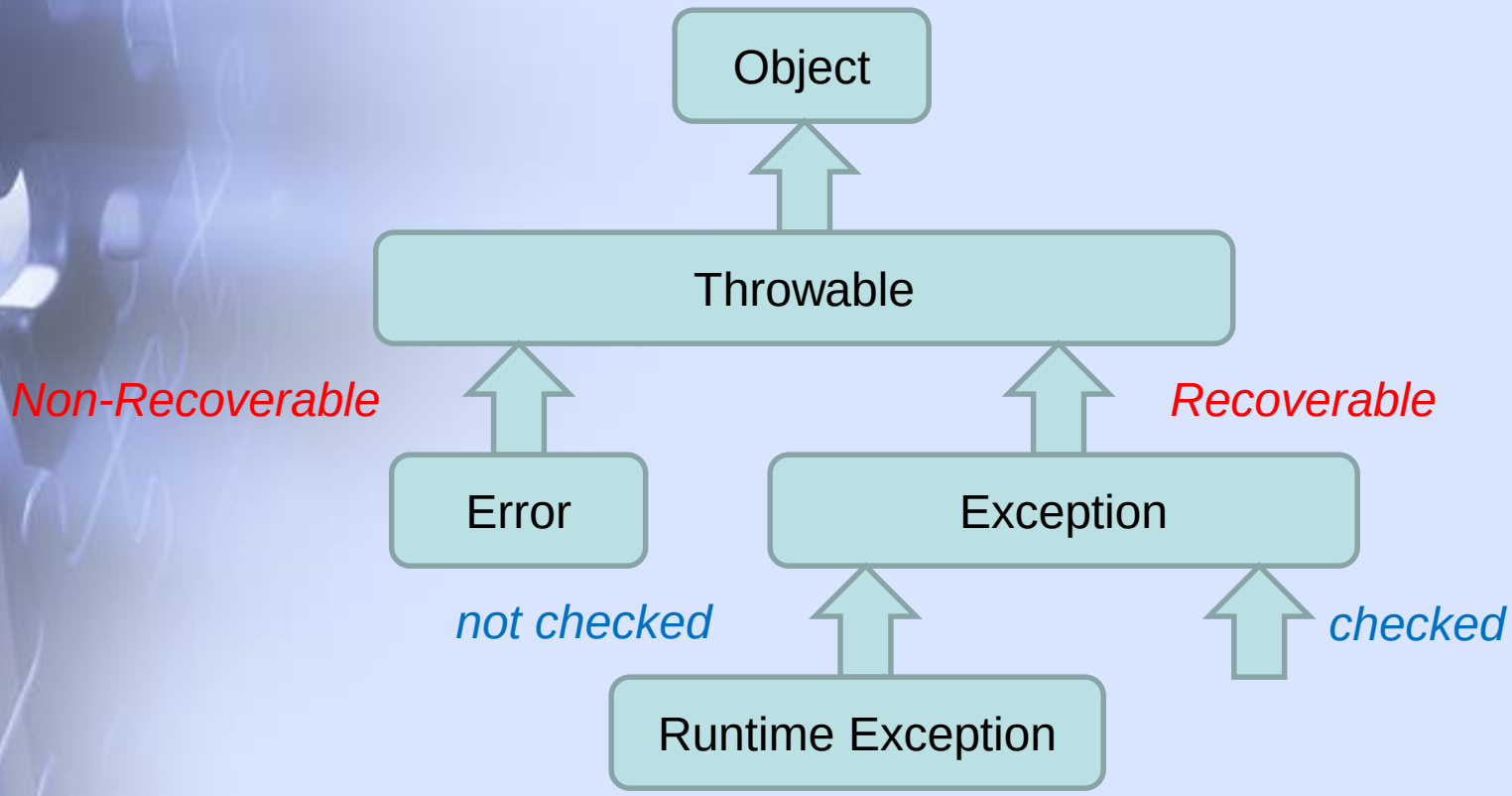
Klasa Throwable

- Klasa Throwable je roditeljska klasa svih grešaka (klasa Error i njene klase nasljednice) i svih izuzetaka (klasa Exception i njene klase nasljednice)

```
String getMessage()
void printStackTrace()
String toString()
```

- Samo objekti koji su instance ove klase ili neke od njenih klasa nasljednica mogu biti bačeni od strane JVM ili putem throw naredbe. Slično, samo ova klasa ili neka od njenih klasa nasljednica može biti tip argumenta catch klauzule

Izuzeci

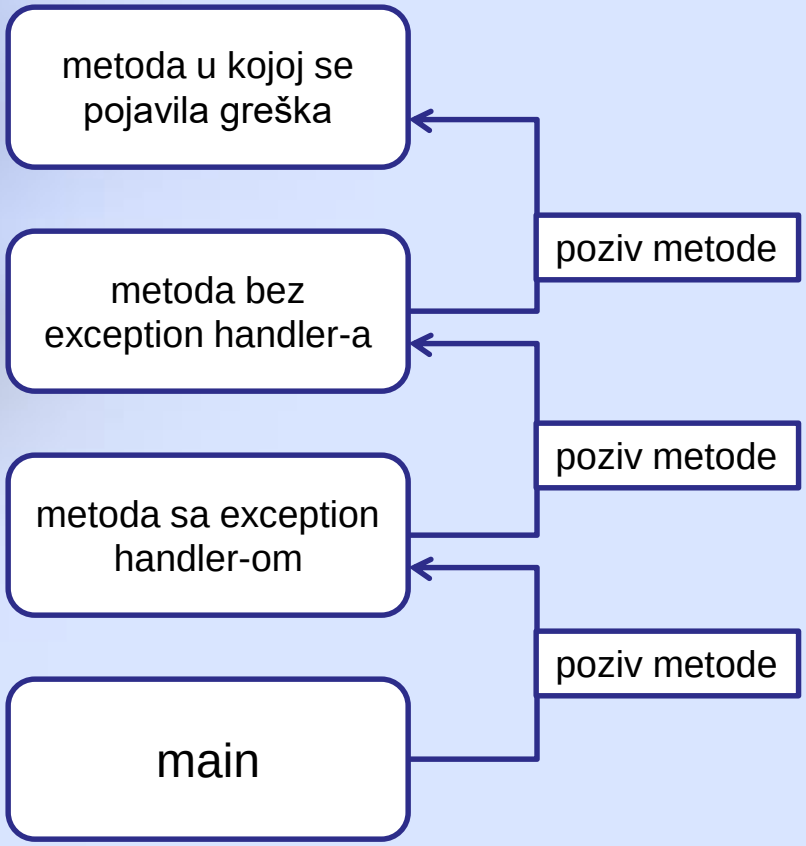

Object
Throwable
Non-Recoverable
Recoverable
Error
Exception
not checked
checked
Runtime Exception

Izuzeci

- izuzetak je događaj koji se dešava za vrijeme izvršavanja programa koji prekida normalan tok programskih instrukcija
- kad se desi greška unutar metode, metoda kreira objekt i predaje ga *runtime* sistemu – bacanje izuzetka (*throwing an exception*)
- *exception* objekt sadrži informacije o grešci, uključujući njegov tip i stanje programa u trenutku pojavljivanja greške
- nakon što je izuzetak bačen, runtime sistem prolazi kroz listu metoda (*call stack*) koje su bile pozvane sve do metode u kojoj se desila greška

Izuzeci

- *call stack*

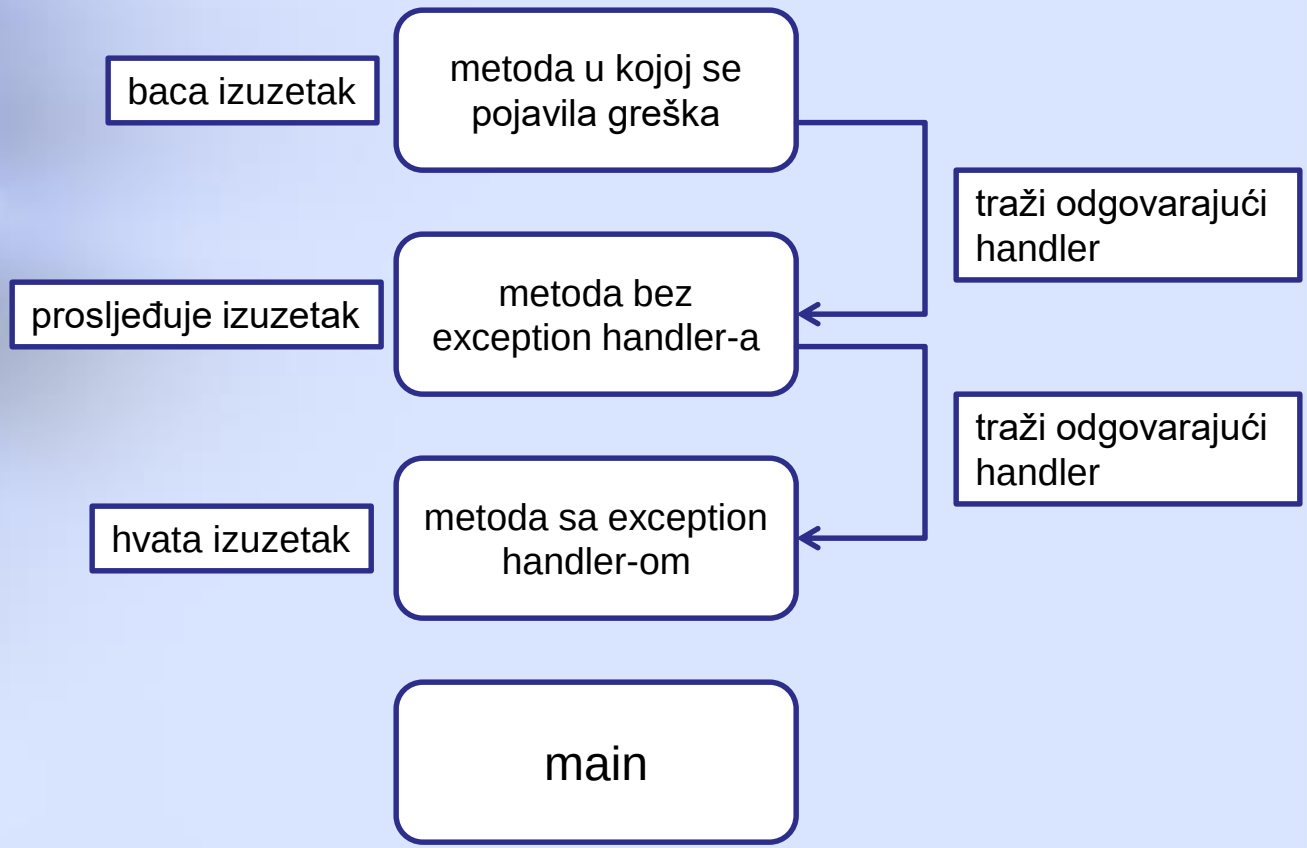


Izuzeci

- *runtime* sistem pretražuje *call stack* kako bi pronašao odgovarajući blok koda (*exception handler*) koji može obraditi izuzetak
- pretraživanje počinje u metodi u kojoj se desila greška i nastavlja kroz *call stack* u obrnutom redoslijedu od onog kojim su metode pozivane
- kad se pronađe odgovarajući *handler*, *runtime* sistem prosljeđuje izuzetak *handler*-u
- *handler* je odgovarajući ako tip objekta bačenog izuzetka odgovara tipu koji *handler* može obraditi – izabrani handler je uhvatio izuzetak
- ako *runtime* sistem ne pronađe odgovarajući *handler*, *runtime* sistem (i program) će prekinuti s izvršavanjem

Izuzeci

- pretraživanje *call stack*-a



Izuzeci

- *catch* ili *specify* zahtjev
- kod koji može izazvati izuzetak mora:
  - imati try – catch blok, ili
  - metodu koja specificira da može baciti izuzetak – throws
- kod koji ne ispunjava niti jedno od navedenog neće biti uspješno kompajliran
- ne moraju svi izuzeci da ispunjavaju navedene zahtjeve – postoje 3 osnovne kategorije izuzetaka, samo 1 je obavezna da ispunjava navedene zahtjeve

Izuzeci

- *checked exception* – dobro napisan program će uhvatiti ovaj izuzetak i obraditi ga
- *checked exception* – je obavezan da ispuni *catch* ili *specify* zahtjev
- svi izuzeci su *checked exception*, osim onih koji su označeni klasama *Error*, *RuntimeException*, i njihovih klasa nasljednica

Izuzeci

- druga vrsta izuzetaka – *error* – klasa *Error* i njene podklase
- izuzeci koji su eksterni za aplikaciju, i aplikacija ih obično ne može predvidjeti, niti načiniti oporavak
- primjer:
  - aplikacija otvara datoteku, ali ne može je čitati zbog hardverske ili sistemske greške – java.io.IOError – aplikacija može uhvatiti ovaj izuzetak i obavijestiti korisnika o problemu, ali je prihvatljivo i ispisati *stack trace* i prekinuti izvršavanje
  - StackOverflowError
  - OutOfMemoryError
- *error*-i nisu obavezni da ispune *catch* ili *specify* zahtjev

Izuzeci

- treća vrsta izuzetaka – *runtime exceptions* – klasa *RuntimeException* i njene podklase
- izuzeci koji su interni za aplikaciju, i aplikacija ih obično ne može predvidjeti, niti načiniti oporavak
- obično su u pitanju programerske greške (*bug*), npr. logičke greške ili nepravilno korištenje API-ja
- primjer:
  - aplikacija treba da otvara datoteku, ali zbog programerske greške FileReader-u je proslijeđen null, umjesto imena datoteke – NullPointerException – aplikacija može uhvatiti ovaj izuzetak, ali je prihvatljivije eliminisati programersku grešku
  - dijeljenje s nulom
  - pokušaj pristupa van granica niza
- *runtime* izuzeci nisu obavezni da ispune *catch* ili *specify* zahtjev
- *error*-i i *runtime* izuzeci se jednim imenom nazivaju *unchecked exceptions*

Obrada izuzetaka

- mehanizam obrade izuzetaka baziran je na try-catch-finally konstrukciji

```
try {
        <naredbe>
} catch (<tip izuzetka> <parametar>) {
        <naredbe>
} finally {
        <naredbe>
}
```

Izuzeci

- *try* blok

```
try {
   <naredbe>
}
catch and finally blocks...
```

- segment označen sa <*naredbe*> sadrži jednu ili više linija koda koje mogu baciti izuzetak
- nakon try bloka, mora doći barem jedan catch blok ili finally blok
- moguće je da catch blok ne postoji, ali u toj situaciji finally blok mora biti specificiran
- moguće je da postoji više catch blokova
- blokovi catch i finally moraju da se pojave nakon try bloka i ne mogu da se pojave bez try bloka

Izuzeci

- *catch* blok

```
try {
<naredbe>
} catch (<tip izuzetka> <parametar>) {
<naredbe>
} catch (<tip izuzetka> <parametar>) {
<naredbe>
}
```

- tip izuzetka označen sa <tip izuzetka> mora biti ime klase koja nasljeđuje *Throwable* klasu – tip izuzetka koji catch blok obrađuje specificira se u zaglavlju catch bloka
- u svakom catch bloku nalazi se programski kod koji se koristi za obradu uhvaćenog tipa izuzetka
- *runtime* sistem poziva *exception handler* i to prvi *handler* u *call stack*-u čiji tip izuzetka označen argumentom <tip izuzetka> odgovara tipu bačenog izuzetka – ovaj tip mora biti Throwable ili neka od klasa naljednica klase Throwable

Izuzeci

- *catch* blok

```
try {
<naredbe>
} catch (<tip izuzetka> <parametar>) {
<naredbe>
} catch (<tip izuzetka> <parametar>) {
<naredbe>
}
```

- *exception handler*-i mogu vršiti oporavak od greške, pitati korisnika da odluči, propagirati grešku handler-u višeg nivoa koristeći ulančane izuzetke
- samo izlazak iz try bloka koji je prouzrokovan bacanjem izuzetka dovešće do prenosa kontrole u catch blok
- nakon završetka catch bloka, izvršavanje se nastavlja u finally bloku, ako je ovaj blok specificiran, i to bez obzira na to da li je novi izuzetak bačen u catch bloku.
- situacija u kojoj catch blok ima argument čiji je tip u hijerarhiji iznad tipa argumenta nekog od narednih catch blokova nije dozvoljena – ovakvu situaciju prijaviće kompajler, jer naredni catch blok ne bi nikad mogao da se izvrši

Izuzeci

- *catch* blok – hvatanje više od jednog tipa izuzetka pomoću jednog *exception handler*-a (više tipova izuzetaka u zaglavlju catch bloka)

```
try {
<naredbe>
} catch (<tip izuzetka> | <tip izuzetka> <parametar>) {
<naredbe>
}
```

- situacija u kojoj se u catch bloku nalazi argument čiji je tip u hijerarhiji iznad tipa argumenta nekog od preostalih argumenata nije dozvoljena – ovakvu situaciju prijaviće kompajler
- Napomena: postoji od Java 7

Izuzeci

- *finally* blok – uvijek se izvršava, bez obzira na to da li se desio izuzetak u try bloku i da li je izvršen catch blok
- kako se ovaj blok uvijek izvršava, može se koristiti za zadatke oslobađanja resursa, kao što su zatvaranje otvorenih datoteka ili mrežnih konekcija
- ako se u finally bloku ne desi izuzetak, niti se izvrši bezuslovna naredba grananja, onda će izvršavanje try i catch bloka odrediti na koji način će se izvršavanje nastaviti nakon finally bloka
  - ako se u try bloku nije desio izuzetak ili je on obrađen u catch bloku, nastaviće se normalno izvršavanje, nakon finally bloka
  - ako se desio novi izuzetak koji nije obrađen (zbog toga što odgovarajući catch blok nije pronađen ili se desio izuzetak u catch bloku), izvršavanje metode se prekida i izuzetak se propagira, nakon izvršavanja finally bloka
  - ako se u finally bloku desi izuzetak, on će biti propagiran, bez obzira na to kako su try i catch blok izvršeni
  - ako se u finally bloku izvrši bezuslovna naredba grananja, poput return naredbe, onda će ova naredba odrediti kako će se izvršavanje nastaviti, bez obzira na to kako su try i catch blok izvršeni
  - posebno, vrijednost vraćena return naredbom u finally bloku je uvijek primarna u odnosu na vrijednost vraćenu return naredbom u try bloku

Izuzeci

- *try with resources* – od Java 7
- *try-with-resources* – try statement koji specificira korišćenje jednog ili više resursa
  - resurs je objekt koji mora biti zatvoren kada više nije potreban
  - *try-with-resources* obezbjeđuje da svaki resurs bude zatvoren na kraju bloka
    - resurs će biti zatvoren bez obzira na to da li je try blok okončan normalno ili „iznenadno“
  - svaki objekat koji implementira java.lang.AutoCloseable interfejs, što uključuje i sve objekte koji implementiraju java.io.Closeable interfejs mogu se koristiti kao resursi
    - public interface Closeable extends AutoCloseable

Izuzeci

- *try with resources* – od Java 7

```
try (<resource>) {
    <naredbe>
}
```

- resursi – objekti klasa koje implementiraju `java.lang.AutoCloseable`
- resurs otvoren u try with resources biće zatvoren, bez obzira da li je u try bačen izuzetak

```
try (BufferedReader br =  new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
```

- rješenje u Java 6

```
BufferedReader br = null;
try{
    br =  new BufferedReader(new FileReader(path));
    return br.readLine();
}finally{
    if(br!=null)
          br.close();
}
```

Do potisnutih izuzetak može se doći metodom **Throwable.getSuppressed()**

Ako izuzetak može biti bačen iz try (readLine()) i iz try-with-resources naredbe (close()), onda prednost ima izuzetak bačen iz try, dok je izuzetak iz try-with-resources potisnut

Ako readLine i close mogu baciti izuzetke, onda prednost ima izuzetak bačen iz finally, dok je izuzetak iz try potisnut

Izuzeci

- ako postoji više resursa u try, onda se svi zatvaraju pri izlasku iz try bloka, ali u redoslijedu obrnutom od redoslijeda njihovog kreiranja

```
try (BufferedReader br =  new BufferedReader(new
 FileReader(path));
   BufferedReader br2 =  new BufferedReader(new
 FileReader(path))) {
      br.readLine();
      br2.readLine();
   }
```

Ugnježdeni try

- izuzetak bačen u unutrašnjem try bloku, a koji nije uhvaćen i obrađen, prosljeđuje se spoljašnjem try bloku gdje može (a ne mora) biti obrađen

```
try {
    try {
        throw new Test1Exception("test 1");
    }finally {
        System.out.println("finally in nested try");
    }
}catch(Test1Exception e) {
    System.out.println("catch in outer");
    e.printStackTrace(System.out);
}finally {
    System.out.println("finally in outer try");
    try {
        throw new Test2Exception("test 2");
    } catch (Test2Exception e) {
        System.out.println("try catch in finally");
        e.printStackTrace(System.out);
    }
}
```

Ključna riječ throws

- specificiranje izuzetaka koje metoda može baciti – ključna riječ *throws*

```
public void writeList() throws IOException,
ArrayIndexOutOfBoundsException {
```

- u ovom primjeru `ArrayIndexOutOfBoundsException` je *unchecked exception* tako da može i

```
public void writeList() throws IOException
{
```

Ključna riječ throws

- provjereni izuzeci koje metoda može baciti moraju biti navedeni u throws klauzuli – kompajler provjerava
- pored navedenih izuzetaka, metoda može baciti i izuzetke koji su podklase izuzetaka navedenih u throws klauzuli (polimorfno ponašanje)
- u metodi, provjereni izuzetak može biti bačen direktno, programski, korištenjem naredbe throw ili indirektno, pozivom druge metode koja može baciti provjereni izuzetak – ako je provjereni izuzetak bačen u metodi, mora biti obrađen na jedan od tri načina:
  - korištenjem try-catch bloka, i to hvatanjem izuzetka i njegovom obradom u catch bloku
  - korištenjem try-catch bloka, i to hvatanjem izuzetka i bacanjem novog izuzetka koji je neprovjereni izuzetak ili bacanjem izuzetka koji je deklarisan u throws klauzuli
  - eksplicitnim dozvoljavanjem propagacije izuzetka u metodu iz koje je tekuća metoda pozvana – deklaracijom u throws klauzuli

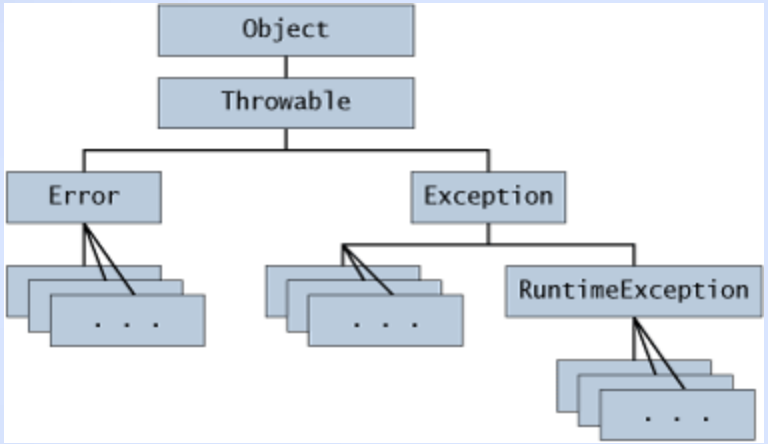
Izuzeci

- bacanje izuzetaka – uvijek korišćenjem throw naredbe

```
throw someThrowableObject;
```

- throw naredba

```
try{
      if(errorCheck())
                  throw new MyException(“problem”);
}catch(MyException e){
      System.out.println(“Excepton: ” + e)
}
```



Izuzeci

- *Error* klasa – kad se desi greška prilikom dinamičkog linkovanja ili druga greška u JVM, virtuelna mašina baci *Error* objekat. Tipični programi obično ne bacaju i ne hvataju ovu vrstu izuzetka.
- *Exception* klasa – većina programa bacaju i hvataju objekte ove ili klasa nasljednica. Ovi objekti indiciraju da se desio problem, koji nije sistemski.
  - *RuntimeException* jedna od klasa nasljednica *Exception* klase

Izuzeci

- ulančani izuzeci
- metode i konstruktori Throwable klase koji podržavaju ulančane izuzetke:
  - `Throwable getCause()`
  - vraća uzrok ili null ako uzrok ne postoji ili nije poznat
  - `Throwable initCause(Throwable)`
  - inicijalizuje uzrok na zadatu vrijednost
  - `Throwable(String, Throwable)`
  - argumenti: detaljna poruka i uzrok
  - `Throwable(Throwable)`
  - argument: uzrok

```
try {
} catch (IOException e) {
    throw new MyException("message", e);
}
```

Izuzeci

- stack trace – daje informacije o istoriji izvršavanja trenutne niti i daje imena klasa i metoda koje su bile pozvane u trenutku kad se izuzetak dogodio
- koristan alat za *debugging*

```
catch (Exception cause) {
   StackTraceElement elements[] =
        cause.getStackTrace();
        for (int i = 0, n = elements.length; i < n; i++) {
                System.err.println(elements[i].getFileName() +
   ":" + elements[i].getLineNumber() + ">> " +
   elements[i].getMethodName() + "()");
        }
   }
-------------------------
cause.printStackTrace()
```

Korisnički definisani izuzeci

- korisnički izuzeci predstavljaju nove izuzetke koji se obično kreiraju kada je potrebno opisati neku specifičnu situacija, a da se pri tome ne koristi neki od postojećih izuzetaka
- novi izuzeci se kreiraju nasljeđivanjem klase Exception ili neke od njenih klasa nasljednica
- ako novi izuzetak nasljeđuje RuntimeException klasu ili neku od njenih klasa nasljednica nastaće novi neprovjereni izuzetak
- preporuka je da se kod kreiranja korisničke klase izuzetaka u nazivu same klase dodaje sufiks Exception
- kao i svaka druga klasa, tako i klasa korisničkog izuzetka može deklarisati polja i metode – ova polja i metode mogu sadržavati dodatne informacije o razlozima dešavanja izuzetka i načinima oporavka
- poziv metode super može se koristiti za postavljanje detaljne poruke o izuzetku

Korisnički definisani izuzeci

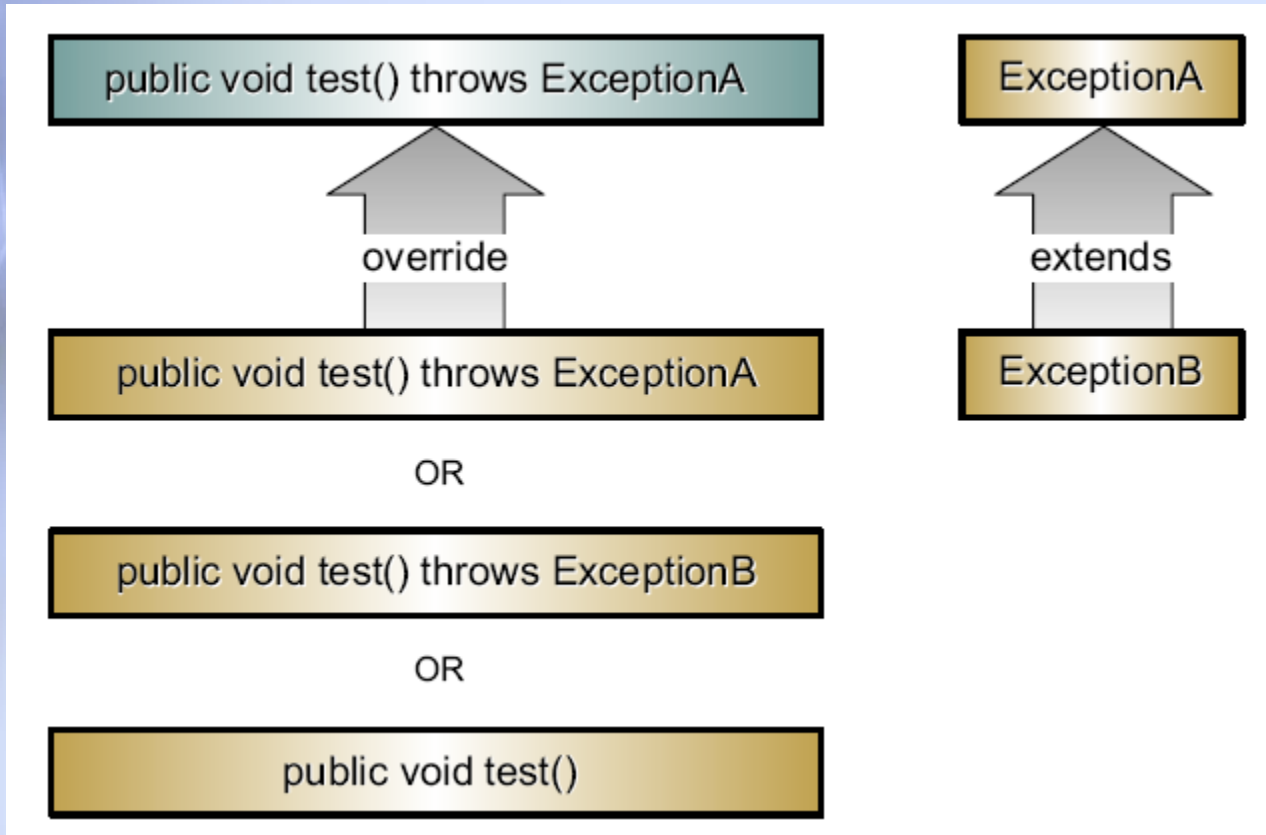
```
public class MyException extends Exception {
	public MyException(){
		super();
	}
	public MyException(String message){
		super(message);
	}
}
```

Izuzeci

- izuzeci i redefinisanje metoda
- redefinisana metoda može:
  - baciti manje izuzetaka ili izostaviti throws klauzulu,
  - baciti iste izuzetke ili
  - baciti izuzetke koji su podklase onih koji su bačeni u metodi koja je redefinisana

Izuzeci

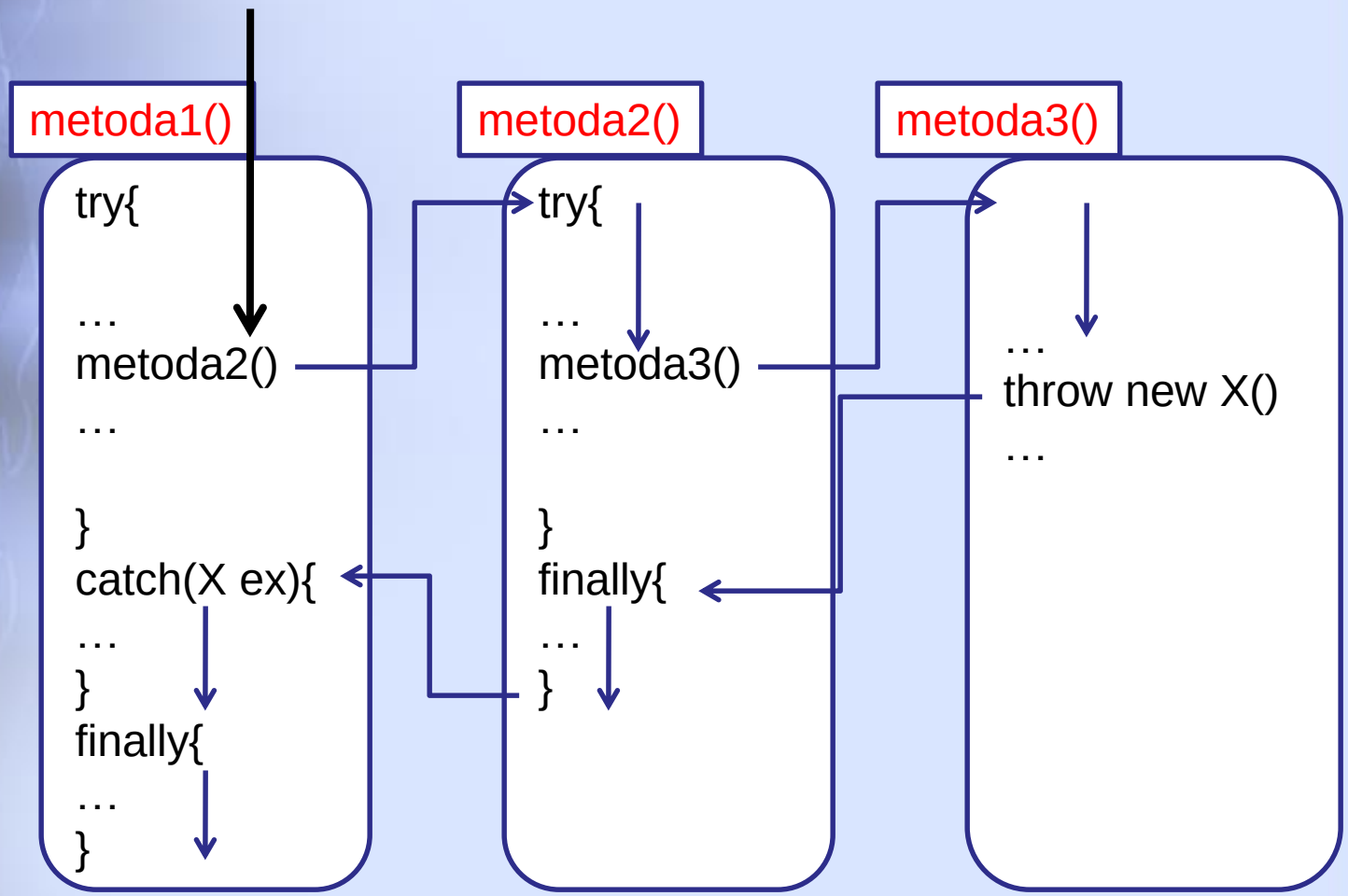
- izuzeci i redefinisanje metoda



Izuzeci

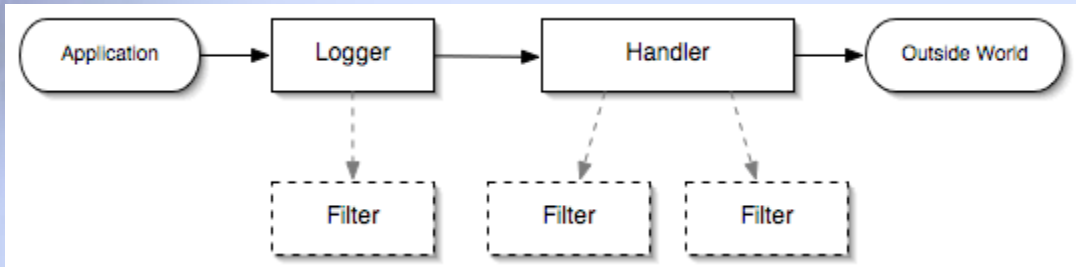
- prednosti korištenja izuzetaka:
  - razdvajanje *error-handling* koda od “regularnog" koda
  - propagacija grešaka kroz *call stack*
  - grupisanje i razlikovanje grešaka

Izuzeci


metoda1()
try{
…
metoda2()
…
}
catch(X ex){
…
}
finally{
…
}
metoda2()
try{
…
metoda3()
…
}
finally{
…
}
metoda3()
…
throw new X()
…

Logging API

- omogućava logovanje informacija iz aplikacije korišćenjem podrške za logovanje dostupne kroz klase u paketu java.util.logging



- Od verzije Jave 1.4

Logging API

- Logovanje se vrši putem instance klase Logger
- Nivoi:
  - SEVERE (najveća vrijednost)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (najmanja vrijednost)
- Postoje i 2 dodatna nivoa:
  - OFF
  - ALL
- Logovi se generišu za sve nivoe jednake ili veće od postavljenog nivoa. Npr. ako je nivo postavljen na INFO, logovi će se generisati za INFO, WARNING i SEVERE

Logging API

- Handlers:
  - java.util.logging.Handler – apstraktna klasa
  - MemoryHandler i StreamHandler (ConsoleHandler, FileHandler, SocketHandler)
  - korisnički definisan handler

public class MyHandler extends StreamHandler

```
try {
        Handler handler = new FileHandler("OutFile.log");
        Logger.getLogger("").addHandler(handler);
        throw new Exception();
} catch (Exception e) {
        Logger logger = Logger.getLogger("package.name");
        logger.log(Level.WARNING, "err", e);
}
```

Logging API

- Formatters:
  - java.util.logging.Formatter – apstraktna klasa
  - SimpleFormatter i XMLFormatter
  - korisnički definisana Formatter klasa

```
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

public class MyFormatter extends Formatter {

   @Override
   public String format(LogRecord record) {
      return record.getThreadID()+"::"+record.getSourceClassName()+"::"
            +record.getSourceMethodName()+"::"
            +new Date(record.getMillis())+"::"
            +record.getMessage()+"\n";
   }

}
```

Logging API

- Log Manager:
  - java.util.logging.LogManager
  - klasa zadužena za čitanje logging konfiguracije, kreiranje i održavanje logger instanci.

```
LogManager.getLogManager().readConfiguration(new
FileInputStream("my_logging.properties"));
```

Logging API

- Postoje i drugi API-ji:
  - Log4J,
  - SLF4J,
  - Apache Commons Logging,
  - LogBack,
  - …

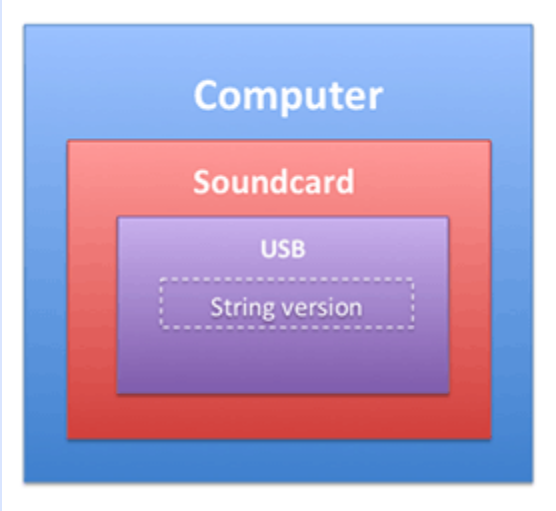
Optional

- Primjer:

```
class Computer{
      private SoundCard soundCard;
      public SoundCard getSoundCard() {
         return soundCard;
      }
}

class SoundCard{
      private USB usb;
      public USB getUsb() {
         return usb;
      }
}

class USB{
      private String version;
      public String getVersion() {
         return version;
      }
}
```



Optional

- Šta je potencijalni problem sa sljedećim kodom?

```
String usbVersion =
    computer.getSoundCard().getUsb().getVersion();
```

NullPointerException

- Potencijalno rješenje:

java.util.Optional

  - od Java 8