Akademija tehničko-vaspitačkih strukovnih studija
odsek NIŠ

Savremene računarske tehnologije **SRT**

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Prof. dr Zoran Veličković, dipl. inž. el.

2019/2020.

Prof. dr Zoran Veličković, dipl. inž. el.

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Pojam i obrada izuzetaka u OO programiranju

(10)

Sadržaj

- **POJAM IZUZETAKA**
  - Značaj izuzetaka u OOP-u
  - Kategorizacija izuzetaka
- **OBRADA IZUZETAKA**
  - Višestruki **`tray-catch-finally`** blokovi
  - Generička forma obrade izuzetaka
  - Ručno bacanje izuzetaka
- **KLASE ZA OBRADU IZUZETAKA**
  - Klasa **`Throwable`**
  - Klasa **`Error`**
  - **`Try-catch`** arhitektura
  - Naredbe **`throws`** i **`finally`**
- **OBRADA UGRAĐENIH IZUZETAKA**
  - Neproveravani izuzeci
  - Proroveravani izuzeci
- **KREIRANJE I OBRADA SOPSTVENIH IZUZETAKA**
  - Deo javnih metoda klase **`Throwable`**
  - Obrada sopstvenih izuzetaka

Pojam izuzetaka

- **IZUZECI** (engl. *exceptions*) su **NEUOBIČAJENA STANJA** (greške) koja se javljaju **U TRENUTKU IZVRŠENJA** programa.
- Dakle, izuzetak je **GREŠKA** koja se otkriva tek pri **IZVRŠENJU PROGRAMA**!
- **OO** programski jezici, pa i Java, poseduju **POSEBAN MEHANIZAM** za **OBRADU** ovih grešaka - izuzetaka.
- Pored mehanizma za **OBRADU IZUZETAKA**, Java poseduje i mehanizme za **IZAZIVANJE** (kaže se i **ISPALJIVANJE**) izuzetaka!
- **REZERVISANE REČI** koje se u Javi koriste za ove potrebe su:
  - `try` (definiše potencijalno opasan blok naredbi),
  - `catch` (hvata odgovarajući izuzetak),
  - `throw` (ručno baca izuzetak),
  - `throws` (izuzima izuzetak od obrade u tekućoj metodi) i
  - `finally` (obavezno izvršiti bloka naredbi).

Kategorizacija izuzetaka

- **IZVORI GREŠAKA** u vreme izvršavanja programa mogu biti najrazličitiji.
- Generalno, izvori grešaka (izuzetaka) se u Javi mogu **KATEGORIZOVATI** prema sledećim kriterijumima:
  - greške (izuzeci) nastale u KODU ili PODACIMA,
  - izuzeci koje ispaljuju STANDARDNE Javine metode,
  - izuzeci izazvani ispaljivanjem SOPSTVENIH izuzetaka
  - greške u kodu koje je izazvao sam PROGRAMER.
- Na **POJAVU IZUZETKA**, Javino izvršno okruženje – **JVM** formira **PREDEFINISANI OBJEKT** koji se potom **PROSLEĐUJE METODI** koja je izazvala grešku kako bi se nastala greška razrešila, odnosno **OBRADILA**.
- Na pojavu **IZUZETKA**, metoda u kojoj je nastao, **POKUŠAVA** da razreši grešku, a ako u tome ne uspe, **PROSLEĐUJE GA DALJE - DRUGIM** metodama na obradu na **VIŠEM** hijerarhijskom nivou.

Tabela: kategorizacija izuzetaka

R.br.	TIP IZUZETKA	OBRAZLOŽENJE
1	**GREŠKE U KODU ILI PODACIMA**	▪ Pokušaj neispravne konverzije objekata; ▪ Pokušaj korišćenja indeksa izvan granica niza; ▪ Nula kao delilac; ▪ ...
2	**IZUZECI STANDARDNIH METODA**	Primer: Metoda **`substring()`** klase **`String`** može da ispali izuzetak **StringIndexOutOfBoundsException**.
3	**ISPALJIVANJE SOPSTVENIH IZUZETAKA**	Mogu se kreirati i ispaliti **KORISNIČKI** kreirani izuzeci.
4	**JAVINE GREŠKE**	Korisničke greške nastale u samom **PROGRAMU**.

Obrada izuzetaka (1)

- Svaki bačeni izuzetak se **MORA OBRADITI** – najbolje sopstvenim metodama.
- Svaki izuzetak koji ne razreši programer biće obrađen **STANDARDNOM** Javinom procedurom što će izazvati **PREKID IZVRŠENJA** programa.
- **IZUZECI** se mogu i **PROGRAMSKI GENERISATI** (kaže se "ručno" izazavni izuzeci) **ILI** ih automatski generiše sam **JAVIN IZVRŠNI SISTEM**.
- **JAVIN IZVRŠNI SISTEM** generiše izuzetke prilikom narušavanja **PRAVILA JEZIKA** ili ograničenja koja potiču od samog **IZVRŠNOG OKRUŽENJA** (pogledaj prethodnu tabelu).
- Uobičajeno je da se **RUČNO** (programsko) **ISPALJIVANJE IZUZETAKA** koristi za **PROSLEĐIVANJE STANJA GREŠKE** metodi pozivaocu.
- Dobra programerska navika je da se **UNUTAR** bloka **`try`** smeste **SVE PROGRAMSKE NAREDBE** koje mogu izazavati izuzetak.
- Pomoću naredbe **`catch`** "hvata se" eventualno **NASTALI IZUZETAK** i potom se korisničkim **PROGRAMSKIM KODOM** pokušava da **REŠI NASTALI PROBLEM**.

Obrada izuzetaka (2)

- Već je rečeno, Javin izvršni sistem **AUTOMATSKI IZAZIVA** (kaže se "ispaljuje") **IZUZETKE** pri pojavi **SISTEMSKIH** grešaka.
- Sa druge strane, naredbom **`throw`**, izuzetak se može izazvati **PROGRAMSKI - RUČNO**.
- Međutim, ako se **NE ŽELI** obrada izuzetka metodom **U KOJOJ JE NASTAO**, on se može **IZBACITI** iz procesa obrade (te metode) naredbom **`throws`**.
- Često je neophodno **IZVRŠITI** neki blok naredbi **PRE POVRATKA IZ IZUZETKA**, tada taj blok treba označiti naredbom **`finally`**.
- Za **HVATANJE ISPALJENIH IZUZETAKA**, mogu se upotrebiti **VIŠE NAREDBI `catch`**, dakle uhvatiti **VIŠE TIPOVA RAZLIČITIH IZUZETAKA**.
- U Javi blok naredbi **`try`** može da bude **UGNEŽDEN**.
- Već je rečeno da se naredbom **`throws`** mogu **ISKLJUČITI IZUZECI** koji se mogu "uhvatiti" **u `try-catch`** bloku (iako se mogu ispaliti)!

Višestruki tray-catch-finally blokovi

```
try {
    //
    // blok koda koji prati pojavu grešaka
    //
}
   catch (TipIzuzetka_1 exOb) {
    // obrada izuzetaka tipa TipIzuzetaka1
}
   catch (TipIzuzetka_2 exOb) {
    // obrada izuzetaka tipa TipIzuzetaka2
}
            ...
   finally {
      //ovaj blok se mora izvršiti pre kraja bloka tray
}
```

try blok – potencijalno opasne naredbe

Tip izuzetaka 1

Blok obrade izuzetka: **catch**

exOb Objekt koji kapsulira podatke o izuzetku

Blok obrade izuzetka: **catch**

Blok obrade izuzetka: **finally**

Generička forma obrade izuzetaka

Ovi izuzeci se **NE HVATAJU u ovoj metodi**

```
double doSomething(int aParam)
    throws ExeptionType1, ExeptionType2{
    // Kod koji ne baca izuzetke

    // Postavi try/catch/finally blokove

    //Kod koji ne baca izuzetke

    // Postavi try/catch/finally blokove

    //Kod koji ne baca izuzetke

    // Postavi try/catch/finally blokove

    //Kod koji ne baca izuzetke
}
```

Tipična struktura: try-catch-finally

```
try {
    // kod koji baca izuzetak
}
catch (MyExeption1 e){
    // kod koji obrađuje izuzetak
}
```
Ovaj izuzetak se **HVATA**

```
catch (MyExeption1 e){
    // kod koji obrađuje izuzetak
}
```
Ovaj izuzetak se **HVATA**

```
finally {
    // kod koji se izvršava posle try bloka
}
```

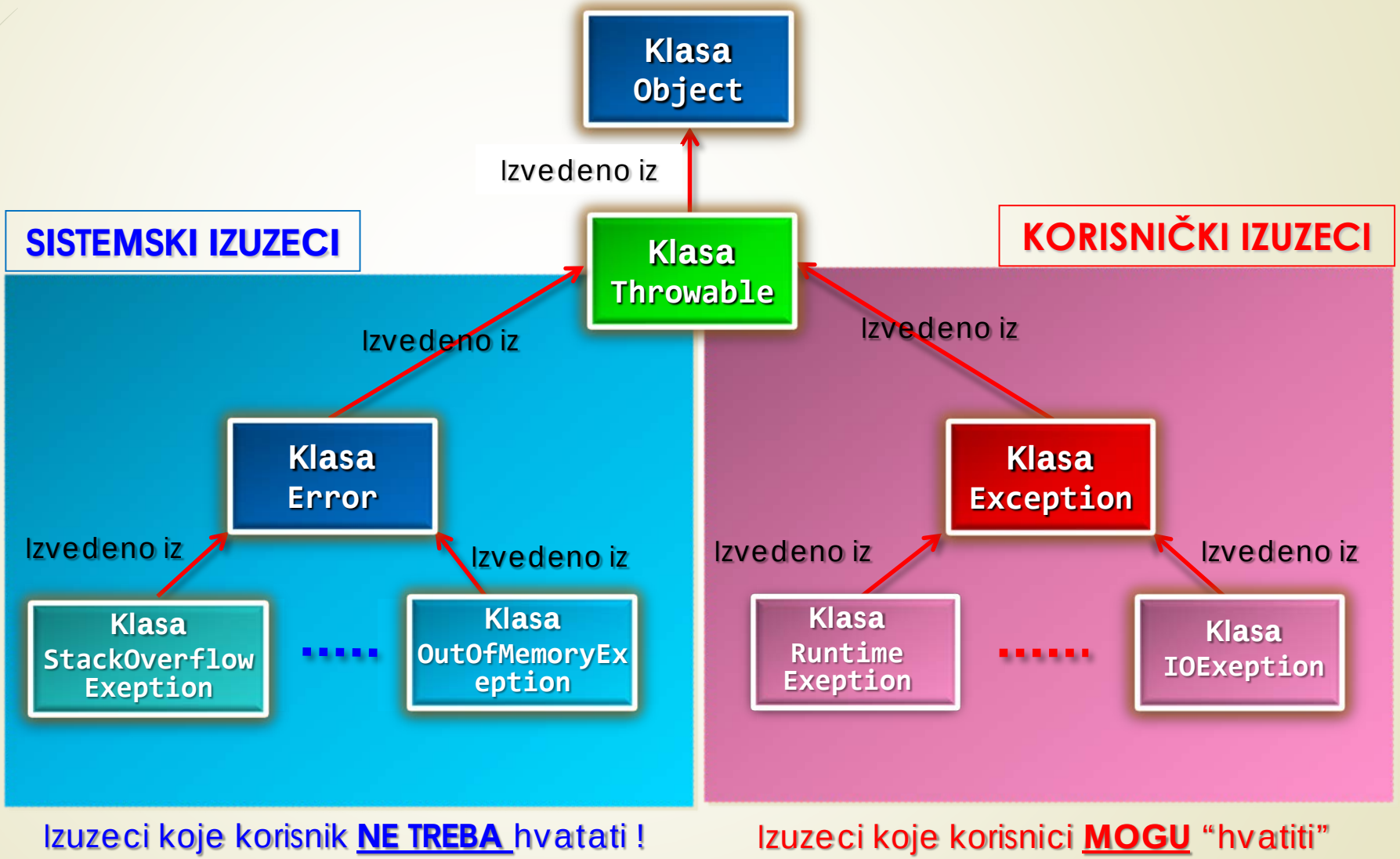
Ručno bacanje izuzetaka

- Izuzetak **UVEK** formira **OBJEKT** neke od potklase klase **`Throwable`**.
- Dakle, **SVI TIPOVI** izuzetaka su potklase ugrađene klase **`Throwable`**.
- Tako, **`Exception`** je potklasa klase **`Throwable`** i odnosi se na izuzetke koje treba da uhvate **KORISNIČKI PROGRAMI**.
- **BACANJE** korisničkog izuzetka se realizuje naredbom **`throw`**:

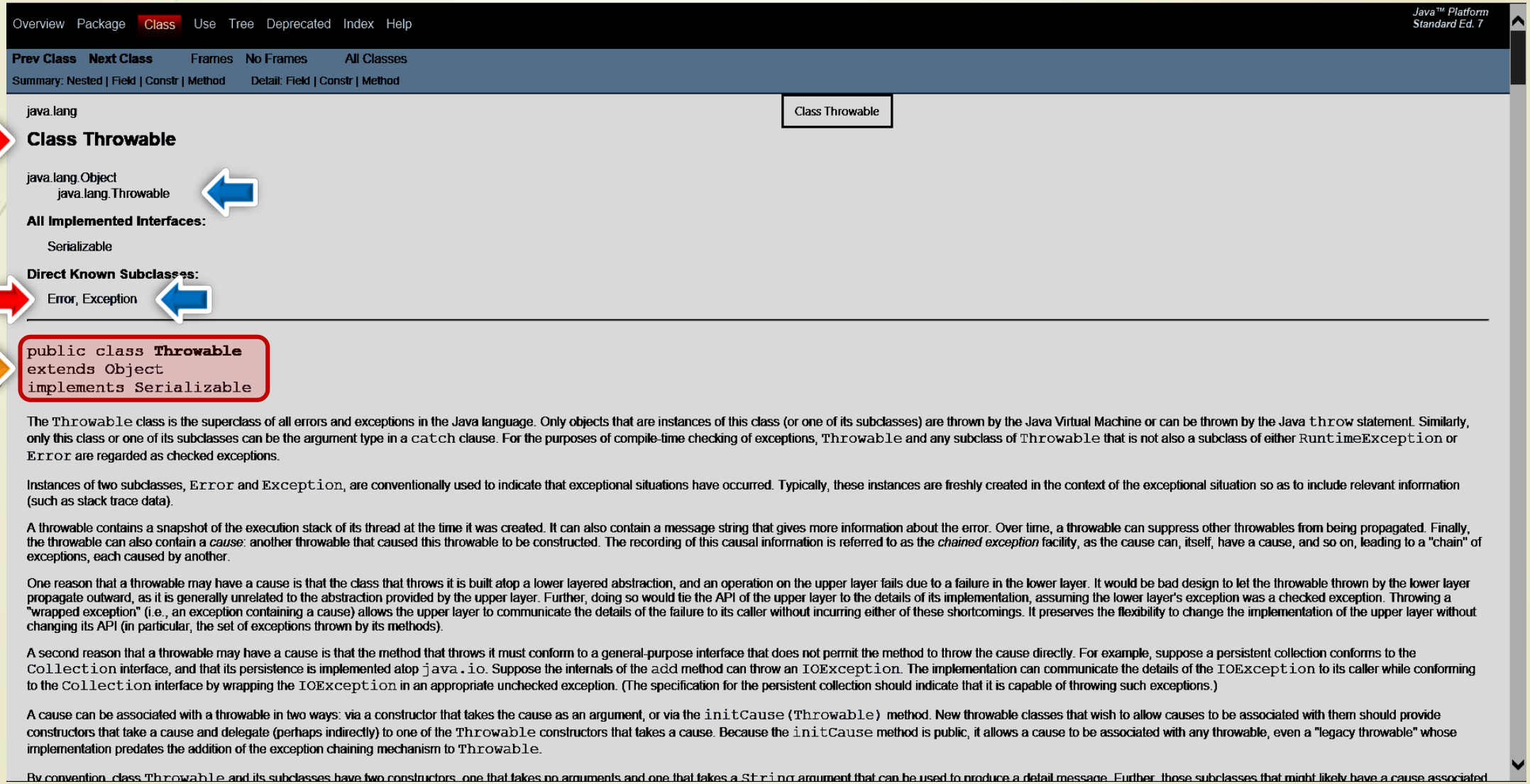
```
throw objekt_tipa_Throwable;
```

- **POTKLASE** klase **`Exception`** se takođe koriste za **KORISNIČKE IZUZETKE**.
- Jedna od često korišćenih potklasa klase **`Exception`** je **`RuntimeException`**.
- Izuzeci ovog tipa se **AUTOMATSKI DEFINIŠU** za programe koje piše **KORISNIK-PROGRAMER**.
- Sa druge strane, **JAVIN IZVRŠNI SISTEM** koristi **IZUZETAK** tipa **`Error`** da se označe izuzeci vezani za Javino okruženje **PRI IZVRŠENJU** programa.

Nasleđivanje klase Throwable



Oracle help: klasa `Throwable` (1)



Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class Throwable

Class Throwable

java.lang.Object
    java.lang.Throwable

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

Error, Exception

```
public class Throwable
extends Object
implements Serializable
```

The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause. For the purposes of compile-time checking of exceptions, `Throwable` and any subclass of `Throwable` that is not also a subclass of either `RuntimeException` or `Error` are regarded as checked exceptions.

Instances of two subclasses, `Error` and `Exception`, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Over time, a throwable can suppress other throwables from being propagated. Finally, the throwable can also contain a *cause*: another throwable that caused this throwable to be constructed. The recording of this causal information is referred to as the *chained exception* facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another.

One reason that a throwable may have a cause is that the class that throws it is built atop a lower layered abstraction, and an operation on the upper layer fails due to a failure in the lower layer. It would be bad design to let the throwable thrown by the lower layer propagate outward, as it is generally unrelated to the abstraction provided by the upper layer. Further, doing so would tie the API of the upper layer to the details of its implementation, assuming the lower layer's exception was a checked exception. Throwing a "wrapped exception" (i.e., an exception containing a cause) allows the upper layer to communicate the details of the failure to its caller without incurring either of these shortcomings. It preserves the flexibility to change the implementation of the upper layer without changing its API (in particular, the set of exceptions thrown by its methods).

A second reason that a throwable may have a cause is that the method that throws it must conform to a general-purpose interface that does not permit the method to throw the cause directly. For example, suppose a persistent collection conforms to the `Collection` interface, and that its persistence is implemented atop `java.io`. Suppose the internals of the `add` method can throw an `IOException`. The implementation can communicate the details of the `IOException` to its caller while conforming to the `Collection` interface by wrapping the `IOException` in an appropriate unchecked exception. (The specification for the persistent collection should indicate that it is capable of throwing such exceptions.)

A cause can be associated with a throwable in two ways: via a constructor that takes the cause as an argument, or via the `initCause(Throwable)` method. New throwable classes that wish to allow causes to be associated with them should provide constructors that take a cause and delegate (perhaps indirectly) to one of the `Throwable` constructors that takes a cause. Because the `initCause` method is public, it allows a cause to be associated with any throwable, even a "legacy throwable" whose implementation predates the addition of the exception chaining mechanism to `Throwable`.

By convention, class `Throwable` and its subclasses have two constructors, one that takes no arguments and one that takes a `String` argument that can be used to produce a detail message. Further, those subclasses that might likely have a cause associated

Oracle help: konstruktori klase `Throwable`

Since:

JDK1.0

See Also:

Serialized Form

See *The Java™ Language Specification*:

11.2 Compile-Time Checking of Exceptions

Constructor Summary

Constructors

Modifier	Constructor and Description
1	**`Throwable()`** Constructs a new throwable with `null` as its detail message.
2	**`Throwable(String message)`** Constructs a new throwable with the specified detail message.
3	**`Throwable(String message, Throwable cause)`** Constructs a new throwable with the specified detail message and cause.
`protected` 4	**`Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)`** Constructs a new throwable with the specified detail message, cause, **suppression** enabled or disabled, and writable stack trace enabled or disabled.
5	**`Throwable(Throwable cause)`** Constructs a new throwable with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of `cause`).

Klasa Error

- Izuzeci tipa **`Error`** nastaju kao posledica **FATALNIH GREŠAKA** i obično ih korisnički (Vaš) program **NE OBRAĐUJE** jer ih svakako **NE MOŽE** rešiti.
- Kada Javin izvršni sistem **OTKRIJE IZUZETAK** (npr. pokušaj deljenje nulom) prvo se napravi **NOVI OBJEKAT ZA TAJ IZUZETAK** a zatim se **BACA ODGOVARAJUĆI IZUZETAK**.
- Zapamtite, **SVAKI IZUZETAK** koji ne obradi Vaš program biće obrađen od strane **UGRAĐENOG** (standardnog) Javinog obrađivača.
- Od **UGRAĐENOG** obrađivača izuzetaka možete očekivati da ispiše tekst sa opisom izuzetka i stanje **PROGRAMSKOG STEKA** za metode koje su ga izazvale, a iza toga, odmah **ZAVRŠAVA PROGRAM**!
- **CILJ** svakog programera je da probleme **NE REŠAVA SISTEM** (jer vam se to neće svideti!) već on **SAM** (programer).
- Već je napomenuto da Java **POSEDUJE MEHANIZME** koje programer može koristiti za **HVATANJE** i **OBRADU** izuzetaka.

Oracle help: klasa Error

java.lang

Class Error

java.lang.Object
  java.lang.Throwable
    java.lang.Error

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationFormatError, AssertionError, AWTError, CoderMalfunctionError, FactoryConfigurationError, FactoryConfigurationError, IOError, LinkageError, ServiceConfigurationError, ThreadDeath, TransformerFactoryConfigurationError, VirtualMachineError

---

```
public class Error
extends Throwable
```

An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The `ThreadDeath` error, though a "normal" condition, is also a subclass of `Error` because most applications should not try to catch it.

A method is not required to declare in its `throws` clause any subclasses of `Error` that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, `Error` and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.

Since:

JDK1.0

See Also:

`ThreadDeath`, Serialized Form

See *The Java™ Language Specification*:

11.2 Compile-Time Checking of Exceptions

Primer: try-catch arhitektura (1)

```
class MultiCatch {
  public static void main(String args[]) {
    try {
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = { 1 };
        c[42] = 99;
    } catch (ArithmeticException e) {
        System.out.println("Deljenje 0: " + e);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Niz indeks oob: " + e);
    }
    System.out.println("Posle try/catch bloka!");
  }
}
```

?

Hvatanje **DVA različita** tipa izuzetaka

Parametri sa komandne linije

Više (2x) naredbi **catch**

Deljenje nulom posle **catch** bloka!

Primer: try-catch arhitektura (2)

```
class ThrowDemo {
  static void demoproc() {
    try {
          throw new NullPointerException ("demo");
        } catch (NullPointerException e) {
           System.out.println("Uhvaćeno unutar procedure");
           throw e;          // ponovno bacanje izuzetka !
         }
}
public static void main(String args[]) {
    try {
           demoproc();
        } catch (NullPointerException e) {
      System.out.println("Ponovo uhvaćen: " + e);
   }
 } // main
} // clasa ThrowDemo
```

Metoda **demoproc()** **baca** i **hvata** izuzetak tipa **NullPointerException**

Poziv metode za **bacanje** i **hvatanje** izuzetka

Hvatanje izuzetka tipa **NullPointerException**

2 puta hvatanje izuzetka tipa **NullPointerException**

Naredbe `throws` i `finally`

- Ako metoda **MOŽE DA IZAZOVE IZUZETAK**, a sama ga **NE OBRAĐUJE** treba ga deklarisati kao **`throws`**.
- Naredbom **`throws`** se **NABRAJAJU** svi tipovi izuzetaka koje metoda može da baca (izuzimajući **`Error`** i **`RuntimeExceptions`**) ali ih **NE HVATA** i **NE OBRAĐUJE**.
- Naredbom **`finally`** treba formirati **BLOK NAREDBI** koje će se izvršavati **NEPOSREDNO PO ZAVRŠETKU** (odnosno izlasku) iz bloka **`tray-catch`**.
- Blok **`finally`** se izvršava **BEZ OBZIRA** na to da li je izuzetak ispaljen ili ne!
- Naredba **`finally`** je korisna prilikom **ZATVARANJA DATOTEKA**, otvorene **MREŽNE KONEKCIJE** i oslobađanja drugih resursa.
- Za **SVAKU NAREDBU `try`** potrebna je **BAR PO JEDNA** odredba:
  - **`catch`** ili
  - **`finally`**!

Primer: finally naredba

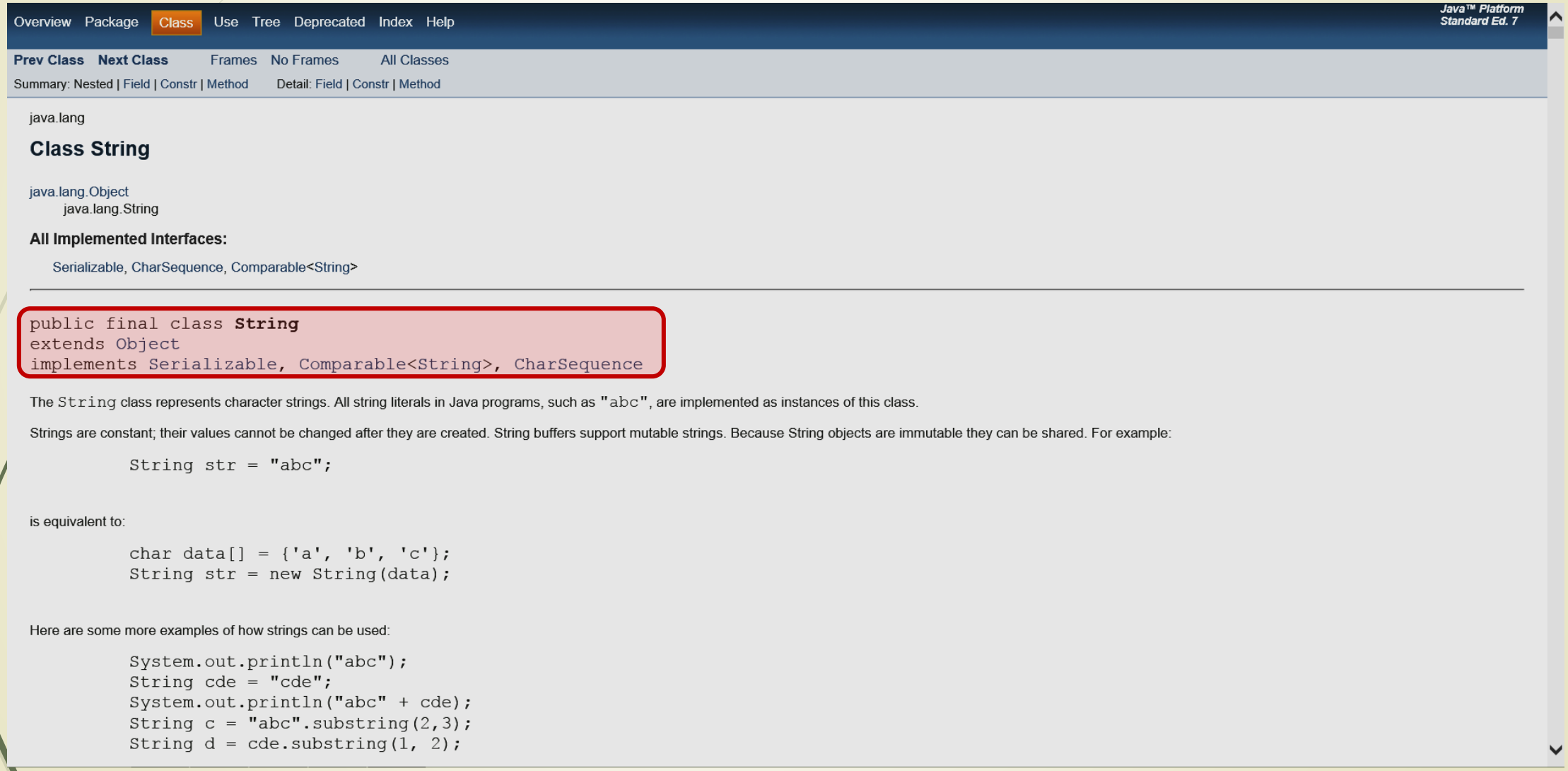
```
// Demonstracija finally naredbe

class FinallyDemo {
  // Bacanje izuzetka izvan metode
   static void procA() {
    try {
            System.out.println("unutar procA");
                      throw new RuntimeException ("demo");
        }
    finally
        {
          System.out.println("procA - finally");
        }
   }
}
```

Obrada ugrađenih izuzetaka (1)

- Već je pomenuto da Java poseduje **UGRAĐENE IZUZETKE**.
- Ovi izuzeci se odnose na **ARITMETIČKE OPERACIJE - DELJENJE NULOM** na **GRANICE NIZOVA**, **NEPOSTOJANJA PROMENLJIVIH**, ...
- U standardnom paketu **`java.lang`** definisano je **VIŠE KLASA IZUZETAKA**.
- Većina izuzetaka su **POTKLASE** standardnog tipa **`RuntimeException`**.
- Spisak **SVIH IZUZETAKA** klase **`RuntimeException`** i njihov opis je dat na sledećem slajdu.
- Sećate se, paket **`java.lang`** se **PODRAZUMEVANO UVOZI** u **SVAKI** Java program.
- Ove izuzetke **NE TREBA** nabrajati u listi iza **`throws`** (moraju se obraditi).
- Ovo su takozvani **NEPROVERAVANI IZUZECI**, **NE PROVERAVAJU SE** u **VREME KOMPAJLIRANJA**.
- Lako je zaključiti da onda postoji lista i **PROVERAVANIH IZUZETAKA**.

Oracle help: izuzeci klase `String` (1)



Overview Package Class Use Tree Deprecated Index Help — Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

Oracle help: izuzeci klase `String` (2)

String

```
public String(int[] codePoints,
      int offset,
      int count)
```

Allocates a new `String` that contains characters from a subarray of the Unicode code point array argument. The `offset` argument is the index of the first code point of the subarray and the `count` argument specifies the length of the subarray. The contents of the subarray are converted to `chars`; subsequent modification of the `int` array does not affect the newly created string.

Parameters:

`codePoints` - Array that is the source of Unicode code points

`offset` - The initial offset

`count` - The length

Throws:

`IllegalArgumentException` - If any invalid Unicode code point is found in `codePoints`

`IndexOutOfBoundsException` - If the `offset` and `count` arguments index characters outside the bounds of the `codePoints` array

Since:

1.5

Oracle help: izuzetaka klase `String/charAt`

charAt

```
public char charAt(int index)
```

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

Neproveravani izuzeci (1)

NEPROVERAVANI IZUZETAK	ZNAČENJE
ArithmeticException	Pojava nedozvoljenog aritmetičkog stanja, kao što je recimo deljenje celog broja nulom.
IndexOutOfBoundsException	Pokušaj korišćenja indeksa čija je vrednost izvan granica prihvatljivih za objekt.
NegativeArraySizeException	Pokušaj definisanja niza sa negativnom dimenzijom.
NullPointerException	Pokušaj korišćenja promenljive čija je vrednost null.
ArrayStoreException	Pokušaj smeštanja u niz objekata koji ne odgovara tipu.
ClassCastException	Pokušaj konverzije objekta u nedozvoljeni tip.
IllegalArgumentException	Prosleđivanje argumenata metodi koji ne odgovaraju po tipu.
SecurityException	Pokušaj izvršavanja nedozvoljene operacije – ugrožena bezbednost.

Neproveravani izuzeci (2)

NEPROPVERAVANI IZUZETAK	ZNAČENJE
`IllegalMonitorStateException`	Pokušaj niti da sačeka na monitor koji joj ne pripada.
`IllegalStateException`	Pokušaj poziva metode u trenutku kada to nije dozvoljeno.
`UnsupportedOperationException`	Pokušaj izvođenja operacije koja nije podržana.

- U tabeli su prikazani tzv. **NEPROVERAVANI IZUZECI** jer **KOMPAJLER NE PROVERAVA** da li ih metoda ispaljuje ili obrađuje.
- Provera ovih izuzetaka SE **OBAVLJA** tek u **VREME IZVRŠAVANJA**!

Proveravani izuzeci

PROVERAVANI IZUZETAK	ZNAČENJE
ClassNotFoundException	Klasa nije nađena.
CloneNotSupportedException	Pokušaj kloniranja objekta bez intefejsa Cloneable.
IllefalAccesException	Odbijen zahtev za pristup klasi.
InstantionalException	Pokušaj da se napravi objekt abstraktne klase ili interfejsa.
InterruptedException	Jedna programska nit je prekinula drugu.
NotSuchFieldException	Zahtevano polje ne postoji.
NotSuchMethodException	Zahtevani metod ne postoji.

- Ovi izuzeci se **PROVERAVAJU** u vreme kompajliranja.
- Ovo su takozvani **PROVERAVANI IZUZECI** koji se **MORAJU UKLJUČITI** u listu **`throws`**.

Obrada ugrađenih izuzetaka (2)

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;
    try {
        d = 0;
        a = 42 / d;
      System.out.println("Ovo se nikada neće printati.");
    } catch (ArithmeticException e) {        // hvata pokušaj deljenja nulom
      System.out.println("Deljenje nulom.");
    }
    System.out.println("Posle catch naredba.");
  }
}
```

Monitoriše blok koda

Zašto?

Kada se izvršava?

Hvata eventualnu neregularnost deljenja nulom (**divide-by-zero** error)

Kreiranje i obrada sopstvenih izuzetaka (1)

- Pravljenje **SOPSTVENIH IZUZETAKA** je neophodno za razrešavanje **SOPSTVENIH GREŠAKA**.
- Za ove potrebe koristi se potklasa **`Exception`** (ona je takođe potklasa klase **`Throwable`**).
- Klasa **`Exception`** **NE DEFINIŠE SOPSTVENE METODE**, već nasleđuje metode svoje natklase **`Throwable`**:
  - **`toString()`**,
  - **`printStackTrace()`**,
  - **`getMessage()`**, ...
- Na ovaj način, **KORISNIČKIM IZUZECIMA**, stoje na raspolaganju **SVE METODE** natklase **`Throwable`**.
- Omogućeno je i **ULANČAVANJE IZUZETAKA**.
- Na sledećem slajdu je prikazana tabela sa nekim od raspoloživih metoda klase **`Throwable`**.

Oracle help: sve metode klase Throwable,

Method Summary

Methods

Modifier and Type	Method and Description
void	**addSuppressed**(**Throwable** exception) Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	**fillInStackTrace**() Fills in the execution stack trace.
Throwable	**getCause**() Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	**getLocalizedMessage**() Creates a localized description of this throwable.
String	**getMessage**() Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace**() Provides programmatic access to the stack trace information printed by **printStackTrace()**.
Throwable[]	**getSuppressed**() Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception.
Throwable	**initCause**(**Throwable** cause) Initializes the *cause* of this throwable to the specified value.
void	**printStackTrace**() Prints this throwable and its backtrace to the standard error stream.
void	**printStackTrace**(**PrintStream** s) Prints this throwable and its backtrace to the specified print stream.
void	**printStackTrace**(**PrintWriter** s) Prints this throwable and its backtrace to the specified print writer.
void	**setStackTrace**(**StackTraceElement**[] stackTrace) Sets the stack trace elements that will be returned by **getStackTrace()** and printed by **printStackTrace()** and related methods.
String	**toString**() Returns a short description of this throwable.

Deo javnih metoda klase Throwable

METOD	OPIS
`String getMessage()`	Vraća sadržaj poruke, opisujući time aktuelni izuzetak. Najčeće je to puno kvalifikovano ime klase izuzetaka i kratak opis izuzetka
`void printStackTrace()`	Prikazuje poruku i evidenciju praćenja steka izvršavanja u standardnom izlaznom toku greške. Kod konzolnih programa to je ekran.
`void printStackTrace(PrintStream s)`	Identičan prethodnom, stim što se kao argument dostavlja izlazni tok. Primer: e.printStackTrace(System.err)
`String toString()`	Vraća objekt tipa String sa opisom izuzetaka
`String getLocalizedMessage()`	Vraća lokalizovan opis izuzetka

Primer: Obrada sopstvenih izuzetaka (1)

```
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return "MyException[" + detail + "]";
  }
} // kraj class
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
     if(a > 10)
       throw new MyException(a);
    System.out.println("Normalan izlaz");
} // kraj compute
```

Konstruktor **nove klase** `MyException`

Redefinisanje metode **`toString()`**: prikazuje se kao vrednost izuzetka

Metoda **`compute()`** baca izuzetak **`MyException`** kada je **`a > 10`**

Ispaljuje izuzetak **`MyException`** ali ga sama metoda **NE OBRAĐUJE!**

Primer: Obrada sopstvenih izuzetaka (2)

```
// Test, nastavak prethodnog primera
public static void main (String args[])
{
    try {
          compute(1);
          compute(20);
        }
    catch (MyException e)
       {
         System.out.println("Uhvaćen" + e);
       }
} // kraj ExceptionDemo
```

2x se baca izuzetak

Obrađivač izuzetaka

```
Called compute (1)
Normalan izlaz
Called compute (20)
Uhvaćen MyException
```