Napredni koncepti objektnog programiranja

Autori:
Goran Savić
Milan Segedinac

1. Statički atributi i metode

Statičke metode

Ranije smo videli kako se u okviru klase mogu definisati metode. Metode opisuju ponašanje objekta. Specifičan slučaj je metoda čiji rezultat ne zavisi od stanja objekta. Drugim rečima, ovakva metoda ne pristupa atributima objekta u svom izvršavanju. Metoda obavlja svoj zadatak i izračunava povratnu vrednost na osnovu parametara prosleđenih metodi. Naravno, metoda može da definiše i svoje lokalne promenljive. Ovakav tip metoda se nazivaju **statičke metode**. U Javi, statičke metode se definišu ključnom rečju static u zaglavlju metode. Dat je primer jedne statičke metode.

```
static int kvadriraj(int x) {
      return x * x;
}
```

Obzirom da statička metoda ne zavisi od stanja objekta, kaže se da statička metoda pripada klasi, a ne objektu. Iz tog razloga, nije potrebno statičku metodu pozivati nad nekom određenom instancom klase, jer ova metoda ne koristi podatke konkretne instance. Zato se statička metoda poziva navođenjem imena klase i imena metode. Dat je primer poziva statičke metode iz prethodnog primera, ako se klasa u kojoj je metoda definisana zove MatematickeOperacije.

```
int rez = MatematickeOperacije.kvadriraj(5);
```

Sada možemo i da razjasnimo *main* metodu od koje kreće izvršavanje programa. Obzirom da na početku programa ne postoji kreirana nijedna instanca nekog objekta, ova metoda je statička i JVM je poziva nad klasom u kojoj je definisana.

Statički atributi

Osim statičkih metoda, moguće je definisati i **statičke atribute**. Slično statičkoj metodi, statički atribut pripada klasi i sve instance klase dele isti statički atribut kao jednu promenljivu. To znači da ako jedna instanca izmeni vrednost statičkog atributa, i druge instance će pri pristupu ovom atributu dobiti izmenjenu vrednost. Statički atribut se definiše ključnom rečju static. Primer definisanja statičkog atributa je dat u nastavku.

```
static int brojac;
```

Ovako definisanom atributu se pristupa navođenjem imena klase i atributa. Dat je primer postavljanja vrednosti statičkog atributa brojac, ako je ovaj atribut definisan u klasi Evidencija.

```
Evidencija.brojac = 1;
```

Treba naglasiti da statička metoda, osim svojim parametrima, može da pristupa i statičkim atributima klase. Pristup je moguć jer se statički atributi ne odnose ni na jednu instancu klase, pa ovi atributi ne definišu stanje objekta. Takođe, iz statičke metode je moguće pozivati samo druge statičke metode klase (jer ni ove metode ne zavise od stanja nekog konkretnog objekta).

Statički blok

Kada je reč o ključnoj reči static, treba pomenuti i još jedan scenario u kojem se ona upotrebljava. U klasi je moguće definisati statički blok koda. Ovaj blok koda se poziva kada JVM učitava klasu. Učitavanje klase se dešava kada se klasa instancira ili kada se pristupa statičkoj metodi ili statičkom atributu klase. Klasa može da ima definisano više static blokova. U nastavku je dat primer definisanja static koda u klasi.

```
class Example  {
      static {
            System.out.println("Initialization");
      }
}
```

Konstante

Ovde ćemo objasniti i još jedan pojam koji u Javi stoji u vezi sa statičkim atributima. Reč je o konstantama. Konstanta je vrednost koja ne može biti izmenjena u toku rada programa. Pri inicijalizaciji promenljive, postavlja se njena vrednost koja se dalje ne menja. Konstante se koriste kako bi se određena vrednost koja se često koristi u programu evidentirala centralizovano. Tako da se pri kasnijoj promeni te vrednosti, izmena vrši samo na jednom mestu u kodu. Takođe, obzirom da je konstanta neka imenovana promenljiva, korišćenje konstanti povećava čitljivost koda jer se vrednost koristi prema imenu koje ukazuje na značenje vrednosti, a ne navođenjem konkretne vrednosti.

Konstanta se u Javi definiše kao statički atribut označen ključnom rečju final. Dat je primer definisanja konstante u Javi.

```
static final int PDV = 20;
```

Konvencija imenovanja konstanti je da se koriste sva velika slova.

2. Ključna reč this

U metodama klase moguće je referencirati objekat nad kojim se metoda poziva. Za ovu svrhu koristi se ključna reč **this**. Ova reč predstavlja referencu na objekat nad kojim se metoda izvršava. Dat je primer korišćenja ključne reči this.

```
class Example {
      String vrednost; //atribut klase
      void ispis() {
            String vrednost = "Sad"; // lokalna promenljiva vrednost

            // this.vrednost je atribut objekta
            System.out.println(this.vrednost + " " + vrednost);
      }
}
Example e = new Example();
e.vrednost = "Novi";
e.ispis();
```

Kod iz prethodnog primera će ispisati Novi Sad, jer this.vrednost referencira atribut vrednost objekta nad kojim je metoda ispis() pozvana. Kao što vidimo, to je objekat e čiji atribut ima vrednost Novi.

3. Konstruktori

Kao što smo objasnili, objekat klase se kreira pozivom specijalne metode pod nazivom konstruktor. Ova metoda ima isti naziv kao i klasa i nema povratni tip. Konstruktor može da ima i parametre. Ovi parametri se koriste za inicijalizaciju objekta pri kreiranju. Tako možemo dobiti kreiran objekat već popunjen inicijalnim podacima. Dat je primer konstruktora sa parametrima.

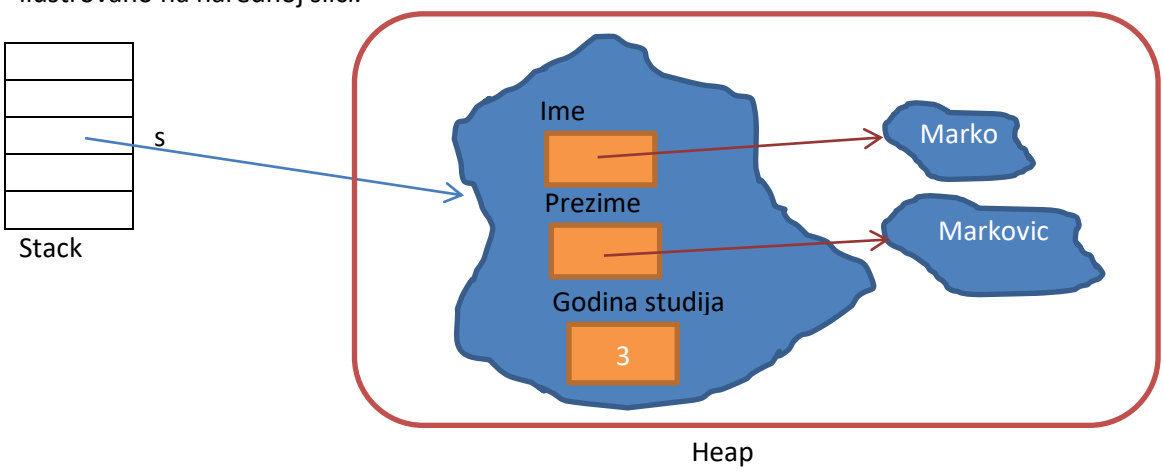
```
class Student {
      String ime;
      String prezime;
      int godinaStudija;
      Student(String i, String p, int g) {
            ime = i;
            prezime = p;
            godinaStudija = g;
      }
}
```

Kao što vidimo, konstruktor preuzima podatke prosleđene kao parametre i upisuje ih u odgovarajuće atribute objekta.

Važno je naglasiti da ako klasa ima definisan konstruktor, kompajler neće automatski generisati prazan konstruktor bez parametara. Za klasu koja je data u prethodnom primeru, moguće je napraviti objekat na sledeći način.

Student s = new Student(“Marko“, “Markovic“, 3);

Prethodni kod će pozvati gore definisani konstruktor i nakon izvršavanja ove naredbe, promenljiva s će biti referenca na objekat klase Student koji ima podatke prosleđene u konstruktoru. Ovo je ilustrovano na narednoj slici.



Praksa je da se parametri konstruktora nazivaju istim imenima kao i odgovarajući atributi koje treba da inicijalizuju. U tom slučaju se u telu konstruktora mora staviti prefiks this kada se pristupa

atributima objekta. Ako se ne stavi ovaj prefiks, program pristupa istoimenom parametru. Dat je modifikovan konstruktor iz prethodnog primera koji koristi ista imena parametara i atributa.

```
Student(String ime, String prezime, int godinaStudija) {
      this.ime = ime;
      this.prezime = prezime;
      this.godinaStudija = godinaStudija;
}
```

Ukoliko želimo da atribut objekta inicijalizujemo na neku predefinisanu vrednost, moguće je u konstruktoru upisati tu vrednost u objekt. Pored toga, moguće je i pri deklaraciji atributa odmah postaviti inicijalnu vrednost. Ova inicijalizacija će biti izvršena pre naredbi iz konstruktora klase. Dat je primer inicijalizacije atributa klase.

```
class Student {
      boolean budzetski = true;
}
```

4. Modifikatori pristupa

Jedan od važnih koncepata objektnog programiranja je **enkapsulacija**. Enkapsulacija predstavlja princip da objekat sakriva svoju internu strukturu i logiku ponašanja od drugih objekata. Pri dizajnu klase se za svaki atribut i metodu određuje ko ima pravo pristupa atributu, odnosno pravo poziva metode. Pravo pristupa je definisano **modifikatorom pristupa** koji se navodi pri definisanju atributa ili metode.

Razlog za uvođenje enkapsulacije je taj što drugi objekat direktnim pristupom može da ugrozi ispravan rad objekta. Pogledajmo to na primeru jedne klase.

```
class  KruznoDodeljivanje {
      int niz[] = {2, 5, 8};
      int indeksSledeceg = 0;

      int preuzmiSledeci() {
            int retVal = niz[indeksSledeceg];
            indeksSledeceg = (indeksSledeceg + 1) % 3;
            return retVal;
      }
}
```

Kao što vidimo, klasa skladišti niz brojeva i omogućuje prezimanje narednog elementa, pri čemu se pri svakom pozivu metode pomera naredni element. Nakon poslednjeg elementa niza sledi opet prvi element niza i tako u krug. Pogledajmo kako možemo koristiti objekat ove klase.

```
KruznoDodeljivanje kd = new KruznoDodeljivanje();
int a = kd.preuzmiSledeci();
int b = kd.preuzmiSledeci();
```

Promenljive a i b imaće vrednosti 2, odnosno 5 nakon izvršavanja prethodnog koda, što je ispravno ponašanje. Pogledajmo sada neznatno modifikovan prethodni kod.

```
KruznoDodeljivanje kd = new KruznoDodeljivanje();
```

```
int a = kd.preuzmiSledeci();
kd.indeksSledeceg = 0;
int b = kd.preuzmiSledeci();
```

U prikazanom primeru, drugi poziv metode preuzmiSledeci() neće vratiti sledeći element nego ponovo prvi element niza (element koji ima vrednost 2). Vidimo da je omogućavanje direktnog pristupa atributu *indeksSledeceg* imalo za posledicu da klasa više ne radi ispravno. Uzastopni pozivi metode *preuzmiSledeci* sada više ne vraćaju redom elemente niza.

Da bi se izbeglo da korisnik klase može da ugrozi ispravnost rada klase, koriste se modifikatori pristupa. Modifikatorima pristupa je moguće kontrolisati koji delovi koda imaju pravo pristupa elementima klase (atributima i metodama klase). U nastavku je dat modifikovan prethodni primer u kojem je pristup atributima klase moguć samo iz metoda unutar te klase.

```
class  KruznoDodeljivanje {
      private int niz[] = {2, 5, 8};
      private int indeksSledeceg = 0;

      int preuzmiSledeci() {
            int retVal = niz[indeksSledeceg];
            indeksSledeceg = (indeksSledeceg + 1) % 3;
            return retVal;
      }
}
```

Kao što vidimo, ključnom rečju *private* je definisano da se atributima može pristupati samo unutar klase. Kompajler bi prijavio grešku ukoliko bismo iz spoljnih klasa probali da pristupimo nekom od ova dva atributa. Dakle, kod prikazan u sledećem primeru bi pri kompajliranju izazvao grešku.

```
KruznoDodeljivanje kd = new KruznoDodeljivanje();
int a = kd.preuzmiSledeci();
kd.indeksSledeceg = 0; // greska
```

Postavlja se pitanje da li je iz drugog objekta moguće pozvati i metodu preuzmiSledeci(). Kao što vidimo u primeru, za ovu metodu nije naznačen modifikator pristupa. Ako ne naznačimo modifikator pristupa, metodi ili atributu klase je moguće pristupiti samo iz klasa koje su istom paketu kao ova klasa. U slučaju da želimo da omogućimo pristup atributu ili metodi iz bilo koje klase, tada se stavlja modifikator pristupa *public*. U narednoj tabeli dat je pregled pomenutih modifikatora pristupa.

Modifikator pristupa	Opis
private	Elementu klase je moguće pristupiti samo iz koda koji se nalazi unutar te klase
public	Elementu klase je moguće pristupiti iz bilo koje klase
nije naveden	Elementu klase je moguće pristupiti samo iz klasa koje su u tom istom paketu

Treba naglasiti da se i za klasu može postaviti modifikator pristupa. Ako se modifikator pristupa ne navede (kao u dosadašnjim primerima), klasa je vidljiva samo klasama iz istog paketa. Ako se postavi

modifikator pristupa public, klasa je vidljiva i za klase iz drugih paketa. Obzirom da je ovo najčešće željeno ponašanje, obično se ispred imena klase stavlja modifikator pristupa *public*.

Dobra je praksa da klasa nikad ne omogućuje direktan pristup svojim atributima od strane drugih objekata. Iz tog razloga, atributi se postavljaju da imaju modifikator pristupa *private*. Putem posebnih javnih (*public*) metoda se onda omogućuje preuzimanje ili postavljanje vrednosti atributa ako je potrebno. Konvencija je da metode koje preuzimaju vrednost atributa imaju naziv get<Naziv atributa>, a metode koje menjaju vrednost atributa da imaju naziv set<Naziv atributa>.

Dakle, enkapsulacija omogućuje apstrahovanje internog funkcionisanja objekta. Korisnik objekta ne mora da se bavi internom logikom rada objekta, već samo koristi njegove funkcionalnosti kroz skup javno dostupnih metoda. Skup funkcionalnosti koje neka programska komponenta pruža spoljnjem korisniku se naziva *Application Programming Interface* (API). U tom smislu, javne metode koje klasa pruža nazivamo API klase.

U skladu sa svim navedenim u ovom poglavlju, dat je primer ranije korišćene klase Student sa modifikatorima pristupa i get, odnosno set metodama.

```
public class Student {
	private String ime;
	private String prezime;
	private int godinaStudija;
	private Grad prebivaliste;

	public String getIme() {
		return ime;
	}

	public void setIme(String ime) {
		this.ime = ime;
	}

	public String getPrezime() {
		return prezime;
	}

	public void setPrezime(String prezime) {
		this.prezime = prezime;
	}

	public int getGodinaStudija() {
		return godinaStudija;
	}

	public void setGodinaStudija(int godinaStudija) {
		this.godinaStudija = godinaStudija;
	}

	public Grad getPrebivaliste() {
		return prebivaliste;
	}

	public void setPrebivaliste(Grad prebivaliste) {
```

```
            this.prebivaliste = prebivaliste;
      }

      void ispisiStudenta() {
            System.out.println(ime + " " + prezime
                  + "; živi u: " + grad.naziv
                  + "; godina studija: " + godinaStudija);
      }
}
```

5. Izuzeci

U toku rada programa može se desiti i odstupanje od predviđenog toka instrukcija u programu. Na primer, program treba u promenljivu da uskladišti peti element niza, a niz sadrži samo 4 elementa. Pojavu ovakvih problema pri izvršavanju programa nazivamo **izuzetak** (eng. *exception*). Moguće je napisati programski kod koji će se izvršiti kao reakcija na pojavu izuzetka.

U Javi, reakcija na izuzetak se vrši navođenjem try ... catch bloka. U okviru try bloka, izvršava se kod koji može da dovede do pojave izuzetka. Reakcija na pojavu izuzetka se definiše u okviru catch bloka. Dat je primer korišćenja try...catch bloka.

```
private void dbConnect() {
      try {
            Connection conn = DriverManager.getConnection(
                  "jdbc:mysql://192.168.1.1:3306/test);
            Statement s = conn.createStatement();
      } catch (SQLException e) {
            System.out.println("Neuspesno povezivanje.");
      }
}
```

Prikazana metoda vrši povezivanje na bazu podataka. Nećemo ulaziti u način povezivanja, obzirom da je svrha prikazanog koda samo da ilustruje pojavu izuzetka u programu. Postoji mogućnost da u trenutku izvršavanja programa povezivanje neće biti moguće izvršiti (npr. jer računar na kojem je baza podataka nije uključen). Dakle, poziv prikazane metode getConnection() može da dovede do pojave izuzetka. Program je dužan da uzme u obzir ovu činjenicu. Iz tog razloga, poziv metode se mora da se nalazi unutar try bloka. Ukoliko se desi izuzetak, tok programa će preći u deo koda koji se nalazi unutar catch bloka. Dakle, program će preskočiti naredbe koje se nalaze ispod poziva metode getConnection() i odmah preći na naredbe definisane unutar catch bloka. Izvršavanje catch bloka se često naziva hvatanje izuzetka. Vidimo da se u zaglavlju catch bloka navodi izuzetak na koji kod reaguje. Svaki tip izuzetka je predstavljen odgovarajućom klasom. Konkretno, ako povezivanje na bazu podataka ne uspe, biće kreiran izuzetak u formi objekta klase SQLException. Unutar catch bloka je moguće izvršiti bilo koji kod kao reakciju na pojavu izuzetka. Ovo zovemo obrada izuzetka (eng. *exception handling*). U datom primeru, samo je ispisana informacija o tome da povezivanje nije moguće.

Izuzetak koji se desio, predstavljen je objektom koji je u prikazanom primeru nazvan e. Može se koristiti proizvoljno ime pri hvatanju izuzetka. Ovaj objekat sadrži različite metode koje nam mogu dati više informacija o izuzetku koji se desio. Metoda getMessage() daje poruku o detaljima izuzetka. Najčešće se koristi metoda printStackTrace(). Ova metoda ispisuje kompletan trag izvršavanja koji je

doveo do pojave izuzetka. Trag izvršavanja podrazumeva spisak svih metoda koje su redom pozivane u programu sve do metode koja je izazvala izuzetak. Najčešće se pri hvatanju izuzetka putem poziva ispiše u konzolu ili uskladišti u fajl trag izvršavanja koji je doveo do izuzetka. Dakle, naredba System.out.println iz prethodnog primera bi se mogla zameniti naredbom e.printStackTrace().

U okviru jednog try bloka mogu se nalaziti pozivi raznih metoda koje izazivaju izuzetke različitog tipa. Moguće je uhvatiti različite izuzetke i na različit način ih obrađivati navođenjem više catch blokova. Pri tome, program ulazi u prvi izuzetak koji odgovara tipu izuzetka koji se desio. Moguće je uhvatiti izuzetak bilo kojeg tipa tako što se napiše catch blok koji kao tip izuzetka prima objekat klase Exception.

Treba pomenuti i da je nakon svih catch blokova moguće navesti i finally blok koji će se izvršiti bez obzira da li se izuzetak desio. U ovom bloku se najčešće vrše završne operacije nad korišćenim objektima (npr. zatvaranje veze ka fajlu).

U nastavku je dat prethodni primer proširen finally blokom, kao i podrškom za hvatanje izuzetka bilo kojeg tipa putem objekta klase Exception.

```
private void dbConnect() {
      try {
            Connection conn = DriverManager.getConnection(
                  "jdbc:mysql://192.168.1.1:3306/test);
            Statement s = conn.createStatement();
      } catch (SQLException e) {

            System.out.println("Neuspesno povezivanje.");
      } catch (Exception e) {
            e.printStackTrace();
      } finally {
            System.out.println("Kraj metode");
      }
}
```

Izvršava se ako se desi izuzetak tipa SQLException

Izvršava se ako se desi izuzetak bilo kojeg drugog tipa

Izvršava se uvek (bez obzira na to da li se desio izuzetak)

U prethodnom primeru smo videli kako kod može da reaguje na pojavu izuzetka. Postavlja se pitanje gde i kako se izuzetak izaziva. Izuzetak se izaziva u Java programskom kodu. Metode se dizajniraju tako da izazovu izuzetak ako ne mogu da realizuju predviđeno ponašanje. Ovo se često naziva izbacivanje izuzetka jer se vrši navođenjem ključne reči throw. Ako metoda izaziva izuzetak, potrebno je da se to i navede u zaglavlju metode putem ključne reči throws.

Dat je primer koda koji vrši izbacivanje izuzetka.

```
public int preuzmiElement(int indeksElementa) throws Exception {
      if (indeksElementa < 0) {
            throw new Exception("Negativan indeks");
      }
      return niz[indeksElementa];
}
```

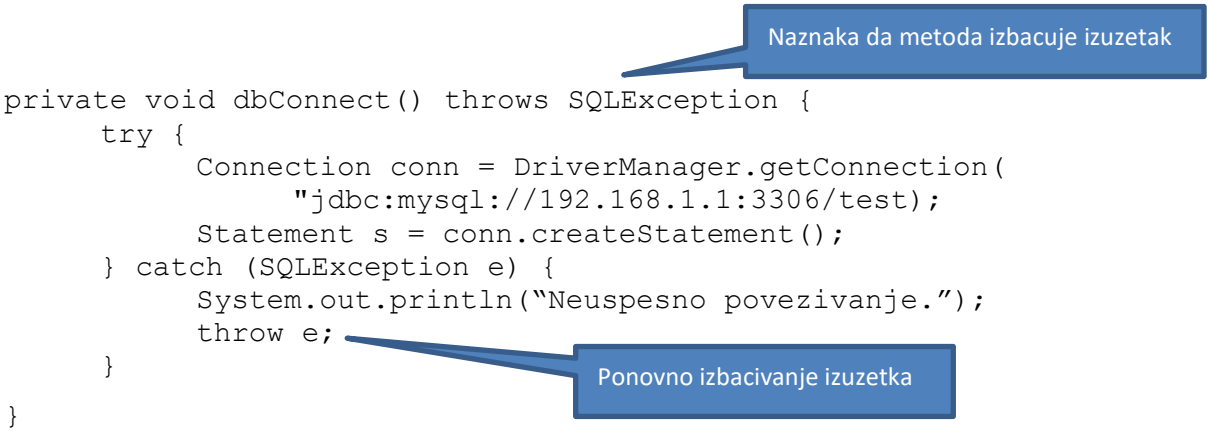
Prikazana metoda treba da iz niza preuzme element koji se nalazi na traženoj poziciji. Ukoliko je pozicija nevalidna (vrednost manja od nule), tada metoda izaziva izuzetak navođenjem ključne reči throw i kreiranjem objekta odgovarajućeg tipa izuzetka. U ovom primeru kreiran je objekat koji predstavlja generički tip izuzetka predstavljen klasom Exception. Umesto toga, moguće je kreirati objekat koji ukazuje na neki konkretniji tip izuzetka (kao što prikazana metoda getConnection() kreira objekat klase SQLException). Kao što vidimo, pri konstruisanju objekta, moguće je kao parametar poslati string koji dodatno opisuje razlog za izbacivanje izuzetka.

Naredba throw prekida dalje izvršavanje metode. Dakle, ako se desi izuzetak, neće se izvršiti preostali kod u metodi. Treba primetiti i da je u zaglavlju metode naznačeno da metoda vraća izuzetak predstavljen objektom klase Exception. Ovde se navodi ona klasa čija je instanca kreirana pri izbacivanju izuzetka u throw naredbi.

Na ovako izbačen izuzetak, pozivalac je dužan da reaguje. Ranije smo videli hvatanje izuzetka kao jedan vid reakcije na pojavu izuzetka. Druga varijanta je da pozivalac odluči da ne vrši hvatanje i obradu izuzetka, već da izuzetak dalje prosledi svom pozivaocu. U tom slučaju, ne treba pisati try... catch blok za hvatanje izuzetka nego se u zaglavlju metode ključnom rečju throws naglasi da ova metoda dalje izbacuje izuzetak svom pozivaocu. Dat je modifikovan primer metode sa početka poglavlja gde metoda ne vrši hvatanje i obradu izuzetka nego dalje propagira izuzetak svom pozivaocu.

```
private void dbConnect() throws SQLException {
      Connection conn = DriverManager.getConnection(
            "jdbc:mysql://192.168.1.1:3306/test);
      Statement s = conn.createStatement();
}
```

Pomenute dve varijante reakcije na izuzetak se međusobno ne isključuju. Metoda može uhvatiti izuzetak i obraditi ga u try...catch bloku i onda dalje propagirati izuzetak svom pozivaocu. Ovo se vrši ponovnim izbacivanjem uhvaćenog izuzetka. Kao i prvi inicijalnom izbacivanju izuzetka, koristi se ključna reč throw. Ovo je ilustrovano narednim primerom.



Izuzetak je moguće propagirati sve do pozivaoca na najvišem nivou hijerarhije, što je main metoda. Ukoliko i ova metoda ne vrši hvatanje izuzetka, može ga propagirati dalje. U tom slučaju će, ako se izuzetak desi, JVM reagovati na izuzetak tako što će zaustaviti program i ispisati trag izvršavanja koji je doveo do izuzetka (eng. *stack trace*).

Ranije smo pomenuli da je kod dužan da reaguje na izuzetak koji izaziva metoda koju poziva (hvatanjem i obradom kroz try...catch blok ili daljom propagacijom putem reči throws u zaglavlju). Ovo važi samo za takozvane ***checked*** izuzetke. Ovi izuzeci se nazivaju još i compile-time izuzeci i, kao što im i ime kaže, u toku kompajliranja se vrši provera da li su obrađeni. Takođe, u programskom kodu se mora eksplicitno navesti ako se želi izbaciti neki od izuzetaka ovog tipa.

Postoji i drugi tip izuzetaka koje će JVM automatski da izazove po potrebi. Na primer, ako objekat ima vrednost null i probamo da pozovemo njegovu metodu, JVM će automatski izazvati izuzetak tipa NullPointerException. Ovaj tip izuzetaka se nazivaju ***unchecked*** izuzeci**.** Ukoliko ih programski kod ne izaziva, izazvaće ih JVM automatski u toku izvršavanja programa, pa se zato ovaj tip izuzetka naziva i run-time izuzeci. Nije obavezno postavljati try...catch blok za hvatanje run-time izuzetaka, niti se mora rečju throws u zaglavlju metode naglasiti da metoda može da izbaci ovakav izuzetak.

Po potrebi, programski kod može da postavi try...catch blok koji hvata neki run-time izuzetak, ako je potrebno reagovati na ovakav izuzetak. Takođe, osim JVM, i programski kod može direktno da izbaci izuzetak ovog tipa putem ključne reči throw i kreiranja objekta koji opisuje neki run-time izuzetak. Dat je izmenjen raniji primer koji izbacuje izuzetak pri pristupu nepostojećem elementu niza. Umesto generičkog izuzetka, sada se kreira run-time izuzetak tipa IndexOutOfBoundsException koji preciznije ukazuje na tip izuzetka koji se desio.

```
public int preuzmiElement(int indeksElementa) {
      if (indeksElementa < 0) {
            throw new IndexOutOfBoundsException("Negativan indeks");
      }
      return niz[indeksElementa];
}
```

Pozivalac ove metode sada nije obavezan da postavi try...catch blok za hvatanje izuzetka ovog tipa. Ukoliko pozivalac ne postavi try...catch blok, izuzetak će se dalje propagirati njegovom pozivaocu i tako redom. Takođe, treba primetiti da pošto metoda izbacuje run-time izuzetak nije u zaglavlju navedeno da izbacuje ovakav izuzetak.