PREDMET

Osnove Java Programiranja

Cas 7-8

KONTROLNE STRUKTURE

Oktobar 2011.

SADRŽAJ

Čas 7 - 8

Kontrolne strukture

Naredbe kontrole toka

Prilikom izrade aplikacije često je potrebno da se dinamički menjamo tok aplikacije, u zavisnosti od nekakvih parametara. To obezbeđuju naredbe kontrole toka. Osnovna uloga naredbe kontrole toka je grananje aplikacije, odnosno odabir mogućih puteva. Dakle, imamo nekoliko situacija koje mogu da se dese, a pomoću naredbe kontrole toka mi određujemo koja od tih situacija će biti za konkretni slučaj. Ovo sve doprinosi dinamičnosti aplikacije. Bez naredbi kontrole toka aplikacija bi bila pravolinijska, ne bi imala dinamičnost, ne bi mogli da se određuju razne situacije, nego bi uvek moralo da ide po istom scenariju. Zahvaljujući naredbama kontrole toka, postoji dinamičnost u aplikacijama.

Naredbe grananja – spisak

Prva, osnovna i najčešće korišćena naredba grananja je naredba **if**, **if - else** konstrukcija je takođe veoma često korišćena. Ako imamo više situacija, **if** naredba nam rešava odlučivanje, ako jeste - uradi to, ako nije - nikom ništa, **if** - **else** je ako jeste - uradi to, ako nije – uradi nešto drugo. Kod **if** – **else** - **if** obezbeđuje sistem višestrukog odlučivanja (ako je to – uradi to, ako nije, onda: ako jeste – uradi to, pa ako nije ni to, e onda ako jeste, itd), dakle, **if** obezbeđuje višestruko grananje. Kombinacije **if** – **else** može da se koristi i u varijanti ugnježdenih **if** – **else**-a. Ako je ispunjen uslov, onda mogu da se dese sledeće varijante. Onda u okviru tog **if**-a imamo nove naredbe grananja. To su tzv ugnježdene **if** – **else** konstrukcije. Zatim imamo metodu **switch**. To je kao neka vrsta preklopnika koji za određeni parametar daje šta za tu vrednost parametara treba uraditi.

Naredbe grananja:

- if
- if – else
- if – else – if
- ugnježdeni if – else
- switch

Mehanizmi koji se koriste za grananje

Pored naredbi grananja postoje i mehanizme koji se koriste u istu svrhu. Jedan od najčešćih i najupotrebljivanijih mehanizama u Javi je **try** – **catch** mehanizam. **Try** – **catch** mehanizam obezbeđuje da se nešto izvrši (tj da se proba da se nešto izvršimo) i, u slučaju da je uspešno izvršeno, sve je uredu. U slučaju da nije uspešno izvršeno, da je došlo do nekih izuzetaka, onda se poziva **catch** deo mehanizma, koji određuje šta će se desiti u slučaju kada nije uspešno izvršen deo pod try. Iako na prvi pogled ovo ne izgleda kao mehanizam grananja, on u suštini jeste to jer, u slučaju da je uspelo, onda će se izvršiti onaj deo naredbi pod **try.** U slučaju da nije uspešno izvršena naredba, onda će se izvršiti deo pod **catch**, tako da bukvalno imamo grananje aplikacije u toj situaciji. **Try** – **catch** mehanizam je specifičan mehanizam koji se koristi uvek kada postoji nekakav kod koji može da dovede do pojave greške. **Overriding** smo učili u delu hijerarhije i videli da pomoću overriding-a možemo da nateramo decu klase da pokreću svoje metode umesto metoda roditelja. Ovim imamo, takođe, mehanizam grananja, iako to nije na prvi pogled vidljivo. Mehanizam grananja se sastoji u tome što, u slučaju da je instanca jedne klase biće izvršena jedna metoda, u slučaju da je instanca druge klase biće izvršena druga metoda. Nemamo potrebe da se eksplicitno postavlja pitanje da li je

to ili ovo, već samim tim što koristimo različite klase u različitim situacijama, dobićemo različito ponašanje. To je dinamika, odnosno vrsta grananja, i zbog toga **overriding** spada u mehanizme grananja. Još jedan mehanizam poznat od ranije, **overloading**, takođe spada u mehanizme grananja. Ako postoji potreba da neka metoda vrši različito ponašanje u zavisnosti od parametra, onda to ne mora da se radi sa pitalicama (if), već može da se kreira više metoda a svaka metoda da radi svoj deo posla (svoju varijantu). Zahvaljujući različitom tipu ili broju parametra biće pozvana jedna od tih odgovarajućih metoda. Ovo je mehanizam grananja jer program radi različite stvari u zavisnosti od parametra koji se u tom trenutku pozove.

Mehanizmi grananja:

- try – catch
- overriding
- overloading

Naredbe koje utiču na tok programa

Pored naredbi grananja i mehanizama grananja, postoje i naredbe koje utiču na tok programa. One takođe mogu da se smatraju vrstama naredbi grananja, jer određuju dinamiku programa, odnosno drugačiji tok programa. Naredba **break** spada u naredbe grananja zbog toga što prekida tok izvršavanja programa u određenom trenutku i prosleđuje na neko drugo mesto nastavak izvršavanja programa (obično izlazak iz aktuelnog bloka naredbi), i samim tim utiče na grananje (tok) programa. Naredba **continue** takođe utiče na tok programa time što prekida započeti tok i vraća petlju na početak, pre njenog logičkog kraja. Naredba **return** se takođe koristi kao naredba kontrole toka. Pomoću naredbe **return** mi možemo da prekinemo tok neke metode, odnosno možemo da vratimo rezultat pre nego što se dođe do kraja metode, i time praktično utičemo na tok programa. **Return** je jedna od naredbi koje se često koriste za kontrolu toka. Javina naredba kojom se prekida izvršavanje aplikacije takođe može da se smatra naredbom koja utiče na programski tok jer na taj način se prekida logički tok programa i završava se aplikacija

Naredbe koje utiču na kontrolu toka

- break
- continue
- return
- System.exit(0)

Primer obične if naredbe

Primer jedne metode (**običanIf**), koja prima dva parametra: x i y (oba su celi broj). Ova metoda vraća manji broj:

```
public int obicanIf(int x, int y)
{
	if (x < y)
	{
		return x;
	}
	return y;
}
```

U primeru je data metoda koja vraća manji broj. U primeru je samo **if** naredba bez else dela iako se vraćaju dve različite varijante. Ovo dodatno ponašanje se postiže pomoću naredbe **return**. Ovde se

vidi kako naredba **return** može da utiče na tok programa. U slučaju da nije korišćena **return** naredba, onda bi bilo pozvano i ono što se nalazi nakon završene **if** naredbe. Dakle, u trenutku kad bi se ispunio uslov da je x<y i izvršilo ono što je u bloku nakon if, onda bi, da nema **return** x, da je tu neki drugi kod, bio izvršavan kod koji se nalazi nakon **if** naredbe. Zahvaljujući tome što ovde postoji **return** naredba, prekinut je taj podrazumevani tok naredbe i vraćen je tok na onaj deo aplikacije koji je pozvao ovu metodu. Ovde se vidi kako izgleda jedna najprostija **if** naredba (jedino se pita da li je x<y) čije rešenje mora da bude **true** ili **false** logička promenljiva. U slučaju da je **true** ta logička promenljiva, biće izvršeno ono što se nalazi u vitičastim zagradama ispod **if**-a. U slučaju da nije, biće izvršeno ono nakon **if** naredbe, odnosno nakon zatvorene vitičaste zagrade.

Primer if – else naredbe

U primeru ispod imamo metodu koja radi istu stvar kao i metoda iz prethodnog primera. Dakle, prima dva cela broja, ustanovljava koji je manji od ta dva i vraća manji broj. Razlika u odnosu na prethodni primer je pojava **else** dela **if** naredbe, koja određuje šta se dešava kada uslov nije ispunjen. Dakle, imamo **if** naredbu, i u zagradama uslov. Uslov, koji se nalazi u zagradama, može da vrati **true** ili **false**. U slučaju da je vrednost uslova **true**, biće izvršeno ono što se nalazi u vitičastim zagradama iza **if**. U slučaju da se vrati **false**, biće izvršeno ono što se nalazi u vitičastim zagradama iza **else** dela naredbe. Treba obratiti pažnju da iza **if** naredbe i iza **else** naredbe ne postoje ; . Ovo je jedna od čestih programerskih grešaka da nakon naredbe stave ; po navikama, jer se obično naredbe završavaju sa ; , međutim, **if** i **else** ne smeju da imaju ; jer tu nije kraj naredbe, to je tek početak **if** naredbe i ovim vitičastim zagradama praktično se određuje domen važenja te naredbe.

```
public int ifElse(int x, int y)
{
	if (x < y)
	{
		return x;
	} else
	{
		return y;
	}
}
```

Primer višestrukog if – else

U ovom primeru grananja imamo metodu koja vraća ispis na ekran u zavisnosti od toga koji je dan u nedelji. Na početku metode se određuje slučajnim odabirom jedan dan u nedelji. Ovde je namerno napravljena jedna greška, dakle, napravljeno je da vraća brojeve od 0 do 7. To je urađeno tako što je za odabir slučajnog broja korišćena matematička naredba **random** iz Javinog paketa Math, koja vraća broj u pokretnom zarezu od 0 do 1. Množenjem tog broja sa brojem 7 povećava se opseg na opseg od 0 do 7. Koristeći drugu matematičku naredbu **round**, zaokružujemo broj na određenu celu vrednost. Na primer, ako je rezultat 6,8 onda će metoda **round** zaokružiti to na 7. Ovim dobijamo cele brojeve u intervalu od 0 do 7. Namerna greška je napravljena u smislu da u intervalu od 0 do 7 ima 8 cifara, a imamo samo 7 dana (za cifru 0 dobićemo grešku).

Krećemo sa višestrukom **if** – **else**. Prvo se pitamo da li je dan veći ili jednak sa 6; to znači da li je 6 ili 7 u ovom slučaju. Ako jeste, izbaci na ekran vikend. Ako nije else, onda se postavlja nov uslov if. Treba obratiti pažnju na konstrukciju **else** – **if**; dakle, u slučaju da prvi uslov nije ispunjen, postavljamo nov uslov. Nov uslov nam kaže da li je dan drugi ili četvrti (utorak ili četvrtak). Ovde treba obratiti pažnju kako se slažu logički izrazi: prvo se porede vrednost promenljive dan sa dvojkom, pa se ona nezavisno poredi sa četvorkom i, ako je bilo koji od uslova ispunjen, dobija se (kao rezultat ovog složenog logičkog izraza) vrednost **true**. U slučaju da nijedan od uslova nije ispunjen, dobija se **false**. To znači da, ako je utorak ili četvrtak, dobiće se tekst koji je prikazan na sledećem primeru.

Ovde se vidi tekst koji će biti upisan u slučaju da je utorak ili četvrtak; dakle, biće ispisano idi na predavanje. Postoji još jedan **else** u slučaju da ni ovaj drugi slučaj nije ispunjen. Onda će out.print

ispisati uči. Vratiće nam dan koji je odabran onom Math naredbom. Ovde se vidi višestruka **if** – **else** konstrukcija, gde se posle svakog **else** postavlja nov uslov šta dalje ispitujemo.

```
public int visestrukiIfElse()
{
	int dan = (int) Math.round(Math.random() * 7);
	if (dan > 6)
	{
		System.out.println("Vikened");
	} else if (dan == 2 || dan == 4)
	{
		System.out.println("Idi na predavanja");
	} else
	{
		System.out.println("Uci");
	}
	return dan;
}
```

Primer ugnježdenog if – else

Na ovom primeru se vidi primer ugnježdenih **if** – **else** naredbi. Prikazan je sličan primer kao i prošli. Dakle, ako je dan veći ili jednak od 6, pa sad to nije dovoljno, već treba da odredimo da li je baš 6 ili 7. Dakle, ako je dan veći ili jednak 6 niće ispisan vikend (ne radi se), ali ako je dan jednak 6 napisaće da ipak mora malo da se radi. U slučaju da nije ni subota ni nedelja, onda će biti ispunjen ovaj **else**. Dakle, ovde postoje dva **if**-a. Jedan **if** se nalazi u drugom (jedan u okviru drugog **if**-a), a **else** se odnosi na prvi **if**. Pravljenje programa sa korišćenjem ugnježdenih **if** – **else** i sa višestrukim **if** – **else** nije pregledno, i treba ih izbegavati. Dakle, ugnježdene **if** – **else** konstrukcije i konstrukcije višestrukog **if** – **else**-a treba koristiti samo u slučaju kada je to krajnje neophodno. Bolje rešenje je koristiti **overload**-ing ili **overriding**, hijerarhiju to je uvek preglednije i elegantnije rešenje nego korišćenje ugnježdenih i višestrukih if – else naredbi.

```
public int ugnjezdeniIfElse()
{
	int dan = (int) Math.round(Math.random() * 7);
	if (dan > 6)
	{
		System.out.println("Vikened - ne radi se");
		if (dan == 6)
		{
			System.out.println("Ipak mora malo da se i radi");
		}
	} else
	{
		System.out.println("Radni dan");
	}
```

```
        return dan;
}
```

Primeri uslova za if naredbu

Šta sve može da bude uslov za **if** naredbu? Dat je primer metode koja vraća logičku promenljivu. Ako je slučajni broj manji od 0.5 metoda će vrati **true**, u suprotnom vraća **false**. Dakle, ovo je jedna mala logička metoda koja vraća **true** ili **false**. U primeru je dat jedna prost uslov a < 120. To je jedan prost logički uslov, ako je a < 120 onda će rezultat ovog kratkog uslova biti **true**, u slučaju da nije biće **false**.

Pogledajte naredbu **if(bul)**. U prethodnom primeru je bio deklarisan **bul** kao **boolean** i on po **default**-u ima vrednost **false**. Međutim, toj promenljivoj je moguće iz programa da se promeni vrednost na **true**, i onda će ovaj **if** biti izvršen. Kao parametar (uslov) **if** naredbe može da stoji samo jedna jedina promenljiva, i vrednost te promenljive će praktično određivati da li će **if** naredba biti urađena ili neće biti urađena. Ovo se često koristi, i to je u programiranju poznat mehanizam flag-ova (ili zastavica) gde nam promenljiva govori stanje (da li jeste ili nije- **true** ili **false**). Recimo, može da se postavi da je uslov da li je korisnik žensko i, ako jeste, postaviće se promenljiva korisnikŽensko na **true**, a ako nije ostaviće se na **false**. Posle je moguće svuda u programu gde nam je potrebno odlučivanje razlike između muško i žensko pozvati ovu zastavicu i, u zavisnosti od njenog stanja **true** ili **false**, odrediće se jedno ili drugo ponašanje. Često se promenljive koriste kao uslovi **if** naredbi.

Logička metoda koju je ranije pripremljena ovde se koristi kao uslov **if** naredbe. Iako može da se čini čudno što se poziv metoda koristi u okviru **if** naredbe, ovo je sasvim logično što se programskog koda tiče, jer rezultat ove metode će biti **true** ili **false**. **If** naredba samo proverava da li je **true** ili **false**. U slučaju da je **true** biće izvršeno, u slučaju **false** neće biti izvršeno. **True** ili **false** se određuju u metodi koju pozivamo. Tok izvršavanja ove **if** naredbe je prikazan u primeru:

Nakon ovog u primeru je prikazan složeniji logički uslov. To je logički uslov koji je dobijen slaganjem dva jednostavna logička izraza (a<120 i b>40), ovaj kod se izvršava tako što se prvo određuje da li je a<120 (tu se dobija vrednost **true** ili **false**), a onda da li je b>40 (i tu se dobija **true** ili **false**), na kraju slažemo ta dva logička izraza logičkom metodom **end**. Ako su oba logička izraza true onda je rezultat ovog složenog logičkog izraza **true**, u svim ostalim slučajevima je **false**.

Konačno tu je i mogućnost slaganja izuzetno složenih logičkih izraza kombinacijom logičkih operatora i svih gore pokazanih varijanti za uslove. Dakle, moguće je da se koriste logičke promenljive, jednostavne logički izrazi, pozivi logičkih metoda, i da se sve to kombinuje pomoću logičkih operatora **end** ili **or.** Zahvaljujući tome na kraju se dobije konačan logički izraz čija je vrednost **true** ili **false**. U datom primeru postoje promenljiva **bul** (koja ima vrednost **true** ili **false**) koja se pomoću **end** operatora slaže sa jednostavnim logičkim izrazom a<120. Rezultat njihovog slaganja će biti opet promenljiva **true** ili **false**. Sa druge strane ovog izraza je poziv logičke metode, koja opet vraća **true** ili **false**. Ova dva logička izraza se međusobno kombinuju logičkom naredbom **or**. Kada se izvrši sve ovo, rezultat svih ovih logičkih operacija će biti jedan **true** ili **false**. Dakle, promenljiva koja ima vrednost **true** ili **false**, i **if** će, u zavisnosti od njene vrednosti, biti izvršen ili neće biti izvršen.

```
private boolean logickaMetoda()
{
        if (Math.random() < 0.5)
                return true;
        return false;
}
public void primeriUslovi()
{
        boolean bul = false;
        if (a < 120)
        {
                System.out.println("PrimerIf.primeriUslovi.jednostavanUslov");
```

```
        }
        if (bul)
        {
              System.out.println("PrimerIf.primeriUslovi.promenljivaKaoUslov"
        );
        }
        if (logickaMetoda())
        {
              System.out.println("PrimerIf.primeriUslovi.LogickaMetoda");
        }
        if (a < 120 && b > 40)
        {
              System.out.println("PrimerIf.primeriUslovi.slozeniUslov");
        }
        if ((bul && a < 120) || logickaMetoda())
        {
              System.out.println("PrimerIf.primeriUslovi.kombinacijaUslova");
        }
}
```

Primer Switch naredbe

U primeru je prikazana upotreba **switch** naredbe. U prikazanom slučaju **switch** naredba će da prima jedan ceo broj **int**, i za to će se koristiti isti onaj metod od ranije za određivanje dana koji, kao što je već rečeno, ima jedan mali bug (postoji 0 kao vrednost dana koja neće biti predviđena programom. Predviđeno je da je 1 – ponedeljak, 2 – utorak,...). Switch naredba prima **int** (ceo broj). **Switch** naredba ne može da primi ništa drugo osim celog broja ili char-a (slovnog podatka).

Case 1 znači u slučaju da je vrednost promenljive dan 1 biće ispisano početak nedelje. Case 3 u slučaju da je vrednost 3, case 5 u slučaju da je vrednost 5 ispisaće se “Uci”, i imamo naredbu **break**. Treba obratiti pažnju šta se dešava u slučaju da je vrednost 5. U tom slučaju biće izvršen samo deo koji ispisuje “Uči”, odnosno na ekranu će biti ispisano samo “uči” i **break** će izbaci programski tok van **switch** naredbe. U slučaju da je vrednost naredbe 3, biće ponovo izvršeno isto, jer nema nikakve **break** naredbe između case 3 i system.out.println(“Uci”). To znači da na ovaj način može da se za dve vrednosti promenljive stavi ista naredba, odnosno dodeli joj se ista naredba, kao što je to prikazano za case 3 i case 5. Nakon case 1 i ispisivanja na ekran “početak nedelje” neće biti prekinut tok aplikacije, već će biti izvršen i deo system.out.println(“Uci”). To znači da u slučaju da se pojavio broj 1, biće prikazano “početak u nedelji uči”, i tek onda će programski tok uz pomoć naredbe break da izađe van switch naredbe. Još jedna situaciju koja može da se primeni u slučaju **switch** naredbe je slaganje nekoliko naredbi, time što se izostavlja naredba **break**. Uprimeru je namerno izostavljena naredba **break**. U drugom delu primera urađena je slična varijanta kao sa vrednostima 3 i 5. Dakle, složene su dve vrednosti tako da se izvrše iste naredbe (za vrednost 2 i 4 biće izvršena naredba idi na predavanje).

Default se izvršava samo u slučaju da ni jedan od prethodnih slučajeva nije izvršen. Dakle, u slučaju da se pojavi vrednost 0, onda će biti izvršen **default**.

```
public int primerSwichInt()
{
      int dan = (int) Math.round(Math.random() * 7);
      System.out.println("Danas je " + dan + " dan u nedelji");
```

```
        switch (dan)
        {
                case 1:
                        System.out.println("Pocetak nedelje");
                case 3:
                case 5:
                        System.out.println("Ucii");
                        break;
                case 2:
                case 4:
                        System.out.println("Idi na predavanja");
                        break;
                case 6:
                        System.out.println("Idi na pijacu");
                case 7:
                        System.out.println("Odmaraj se");
                default:
                        System.out.println("Bog je stvorio samo sedam dana");
        }
        return dan;
}
```

Parametri switch naredbe

Parametri **switch** naredbe mogu da budu ceo broj **int** ili slovni podatak **char**. Oba tipa podataka u memoriji zauzimaju 32 bita, što znači da **switch** naredba radi samo sa ovim tipovima promenljivih.

Šta sve može da bude parametar switch naredbe? Kao što je pokazano u predhodnom primeru, promenljiva tipa **int** može da bude parametar switch naredbe, a to može da bude i promenljiva tipa **char**. Kao i u slučaju logičkih metoda, i ovde može da postoji metoda koja vraća vrednost **int** ili char, i ona može da se koristi kao parametar **switch** naredbe. Izraz čiji je rezultat **int** ili **char**, takođe može da bude parametar switch naredbe.

- int (32 bita)
- char (32bita)

parametar može biti:

- promenljiva tipa **int** ili **char**
- metoda koja vraća **int** ili **char**
- izraz čiji je rezultat **int** ili **char**

Primer switch naredbe sa char parametrom

Prikazan je primer izračunavanja aritmetičke operacije nad dva broja u zavisnosti od toga koji simbol (koja računska radnja) je prosleđen. Metoda prima tri parametra: dva broja nad kojima se primenjuje

operacija i sam simbol operacije kao char podatak. **Switch** naredbi se prosleđuje, simbol koji je metoda primila pa, u slučaju da je simbol +, vrati a+b, u slučaju da je simbol -, vrati a-b. U slučaju da je simbol *, pomnoži ova dva broja, u slučaju da je simbol /, vrati deljenje ova dva broja, i **default** vrati 0. U slučaju da je poslat neki drugi simbol, koji nije ni jedan od ova četiri osnovna aritmetička simbola, biće vraćena 0 kao rezultat ove metode. Nigde u ovoj **switch** konstrukciji nema naredbe **break**. Naredba **break** služi da se prekine jedan slučaj i da se završi sa izvršavanjem tog bloka. Međutim, u primeru je za prekid daljnjeg izvršavanja korišćena naredba **return**. Naredba **return** se česti koristi za kontrolu toka. Ovo je jedan dobar primer korišćenja naredbe **return** za kontrolu toka.

```
public double racunskeRadnje(int a, int b, char simbol)
{
	switch(simbol)
	{
		case '+':
			return a+b;
		case '-':
			return a-b;
		case '*':
			return a*b;
		case '/':
			return a/b;
		default:
			return 0d;
	}
}
```

Primer poziva prethodne metode

Objektu račun pozivamo metodu računske radnje, i u pozivu šaljemo dva broja (4 i 5) kao i simbol koji treba da predstavlja računsku operaciju. U ovom slučaju, rezultat rada ove metode će biti broj 20.

```
racun.racunskeRadnje(4, 5, '*');
```

- promenljiva **racun** je instanca klase u kojoj se nalazi metoda
- obratite pažnju na upotrebu naredbe return u predhodnom primeru

Kontrola toka pomoću try – catch mehanizma

U primeru je prikazan tipičan primer **try** – **catch** mehanizma. U okviru try dela try-catch mehanizma se izvršava deo koda koji je iz nekog razloga rizičan; **catch** deo ima parametar exception (neispravno izvršavanje rizičnog koda će da izbaci **exception**, vratiće promenljivu tipa **exception**), i u **catch** delu može da se pomoću te promenljive ustanovi šta se desilo (koja situacija se dogodila, zbog čega je izbačena greška, zbog čega izvršavanje rizičnog koda nije uspelo).

```
try
{
	//rizičan kod
}catch (Exception e)
{
	//u slučaju da dođe do izuzetka
```

```
}
```

Rizičan kod

Tipičan primer rizičnog koda je snimanje i čitanje sa diska. U trenutku pisanja aplikacije ne može da se zna da li će korisniku biti pun disk, pa će da se vrati greška, ili će fajl biti oštećen, pa neće moći da se učita. Iz tog razloga, uvek kada se poziva naredba čitanja ili pisanja po disku, to smeštamo u **try** – **catch** mehanizam, i hvatamo odgovarajući **exception** koji se javlja u slučaju greške. Dakle, u slučaju da ne uspe pisanje ili čitanje sa diska, ta naredba će da baci **IOexception** (konkretni **exception**), koji ćemo da uhvatimo u catch delu i, u zavisnosti od toga šta se desilo (recimo disk je bio pun), ispisaće se na ekranu korisnika npr grešku ili će da se preduzmu druge aktivnosti. Različiti tipovi grešaka imaju različite exception. Recimo, snimanje i čitanje sa diska baca **IOexception**, dakle, konkretan izuzetak **input** – **output**. Rad sa mrežom takođe izbacuje **IOexception**. **IOexception** je ulazno – izlazni izuzetak, i sve što se tiče ulaza i izlaza iz virtual mašine ima mogućnost da napravi **IOexception**. To može da bude čitanje i pisanje po disku ili slanje i primanje podataka kroz mreže. Naravno, osim **IOexception**-a postoje i drugi tipovi **exception**-a. Na primer, pretvaranje jedne klase u drugu klasu (tzv kastovanje) može da izazove **ClassCastException**. Dakle, ako probamo da kastujemo jednu klasu u drugu, a nisu međusobno odgovarajuće, jednostavno će se desiti **ClassCastException**. U slučaju parsiranja podataka, u slučaju pretvaranja stringa u kome se nalazi broj u broj, može da dođe do **ParseException**-a, tj može da se desi da je u tom stringu ono što smo mislili da je broj 1 u stvari smo slovo l, pa onda dođe do greške jer to ne može da se parsira, prepozna kao broj. Tada će biti izbačen **ParseException.** U zavisnosti od toga šta se dešava u try delu odnosno koji je rizični kod, zavisi i koji će **exception** biti izbačen. Može da se desi da u okviru jednog dela rizičnog koda se pojavi nekoliko **exception**-a. Recimo čitamo objekat sa diska (može da se desi **IOexception**) i pretvaramo ga u tip klasu koja je unutra. Dakle, s obzirom na to da na disku može da bude nekih oštećenja ili može da bude neki drugi objekat, a ne onaj koji mi hoćemo, može da nam se desi **IOexception** ili može da se desi **ClassCastException**.

Nekoliko tipičnih exceptiona

- snimanje i čitanje sa diska (IOException)
- rad sa mrežom (IOException)
- pretvaranje jedne klase u drugu – kastovanje (ClassCastException)
- parsiranje podataka (ParseException)

Hvatanje više izuzetaka

Kada postoji rizičan kod koji može da izazove nekoliko različitih **exception**-a. pomoću **catch**-a dela se hvataju ti exceptioni i može da se uhvati više excepotina, na primer **IOexception** nakon toga ide deo koda koji će se desiti u slučaju da se desio **IOexception**. Nakon toga ide sledeci **catch** sa parametrom **ParseException**, koji će da hvata one situacije kada je došlo do **ParseException**-a. Ako postoji još neka greška za koju nismo sigurni kog je tipa, moguće je i nju uhvatiti sa **catch** i parametar **exception**, na ovaj način biće uhvaćeni svi izuzeci osim ona prethodna dva. Na ovaj način može da se uhvatite nekoliko različitih grešaka koje mogu da se pojave na istom delu koda. U zavisnosti od toga koja greška se desila, biće pozvana odgovarajuća **catch** metoda. To znači da u slučaju da se desio **IOexception**, neće biti izvršen deo **catch**-a sa običnim **exception**-om, niti će biti izvršen deo **catch**-a sa **ParseException**-om. Biće izvršena samo **catch** deo sa **IOexception**-om, i na taj način može u zavisnosti od toga koji **catch** je stavljen, da se izvrši određen deo koda.

```
try
{
      //rizičan kod
}catch (IOException e)
{
      //u slučaju da dođe do
      //ulazno-izlaznih izuzetka
```

```
}catch (ParseException e)
{
      //u slučaju da dođe do greske u parsiranju
}catch (Exception e)
{
      //Hvata sve izuzetke koji su preostali
}
```

Korišćenje try-catch-finally

Postoji i reč **finally** u **try** – **catch** mehanizmu, i ona će biti izvršena u svakom slučaju. Dakle, bez obzira da li je uspešno izvršen deo rizičnog koda ili je došlo do nekog izuzetka, deo pod **finally** će biti izvršen. Primena **finally** dela nije preporučljiva, i zaista su izuzetno retke situacije kada postoji realna potreba za ovom naredbom. U slučaju da se, prilikom dizajna aplikacije, pojavi potreba za **finally** kodom, treba izvršiti rekonstrukciji i izmene aplikacije, pre nego da se koristi **finally** deo.

```
try
{
      //rizičan kod
}catch (Exception e)
{
      //u slučaju da dođe do izuzetka
}finally
{
      //izvršava se u svakom slučaju
}
```

Implementacija equals()

Java ima root klasu to je kalsa koja se nasleđuje kad ne nasledimo ni jednu drugu

Sve klase u javi imaju ovu korenu klasu i to je klasa Object

Klasa Object ima u sebi par predefinisanih metoda između kojih su i

- toString()
- equals(Object o)

equals metoda se implementira isto kao i toString upotrebom anotacije @Override

Primer:

```
@Override public boolean equals(Object obj) {
        Osoba osoba = (Osoba) obj;
        if(prezime.equals(osoba.getPrezime())                    &&
ime.equals(osoba.getIme()))
                return true;
        return false;
}
```

Implementacija compareTo

Metoda compareTo nije deo Object klase

Ova metoda je definisana interface-om Comparable<T>

T označava klasu na koju se odnosi poređenje
Interface Comparable ima potpis samo jedne metode

```
public int compareTo(T o);
```

Klasa u kojoj želimo da imamo implemenaciju compareTo metode mora prvo da implementira interface Comparable
Primer

```
public class Osoba implements Comparable<Osoba>{
```

Implementirani interface će zahtevati postojanje metode compareTo
Primer implementacije compareTo metode za klasu osoba koja ima dva podatka ime i prezime i gde želimo da prvo ustanovljavamo redosled (poređenje) po prezimenima a ako su prezimena identična onda po imenima

```
public int compareTo(Osoba o) {
        if(prezime.equals(o.getPrezime()))
            return ime.compareTo(o.getIme());
        return prezime.compareTo(o.getPrezime());
    }
```

Ova metoda nema @Override jer slična metoda ne postoji u nekoj nad klasi. Ovu metodu smo dobili implementacijom Interface.

Dva osnovna pravila OOP

Postoji jedan relativno trivijalan ali izuzetno efikasan način merenja kvaliteta OO koda.
Ovaj postupak ocene kvaliteta se zasniva na dva osnovna pravila, koja mogu brzim vizuelnim pregledom koda da se ustanove
Poštovanje ova dva pravila forsira programere da prave pravilne i kvalitetne OO aplikacije
Svaka klasa mora da ima samo jedno zaduženje
Metoda ne sme da bude duža od ekrana editora razvojnog okruženja

Zašto se ne koristi switch

Parametar ne može da bude objekat samo int ili char prosti tipovi čak ne može da bude ni njihove wraper klase
Naredba break koja predstavlja nasilni prekid rada i koju takođe ne bi trebalo koristiti je sastavni deo mehanizma.
Za svaki case mora da se napišu bar 3 reda koda što za iole ozbiljniju upotrebu znači da će samo ovaj mehanizam biti nekoliko desetina redova. Ovo značajno umanjuje čitljivost i upotrebljivost koda

problem sa upotrebom if

if mehanizam je uobičajeni način grananja u aplikacijama, pogotovu kad imamo višestruke izbore.
Metoda u kojoj se ovakvo grananje nalazi obično ne može da stane u ekran editora.
Čitljivost ovakve metode je često problematična, pa samim tim i mogućnost daljeg proširenja aplikacije

Primer rada sa konstantama

```
public class Jelovnik {
    public final static int A = 0;
    public final static int B = 1;
```

```
    public final static int AB = 2;
    public final static int O = 3;

    public Jelovnik() {
        int krvnaGrupa = 1;
        dorucak(krvnaGrupa);
    }
    private void dorucak(int krvnaGrupa) {
        if(krvnaGrupa == A){
        System.out.println("Za dorukac treba da jedete Grejp i sok od
naradze");
        }
        else if(krvnaGrupa == B){
        System.out.println("Za dorucak treba da jedete sir i jogurt");
        }
Primer rada sa konstantama
        else if(krvnaGrupa == AB){
            System.out.println("Za dorucak treba da jedete kornfleks sa
jogurtom");
        }
        else if(krvnaGrupa == O){
            System.out.println("Za dorucak treba da jedete sunku");
        }
    }
}
```

U predhodno prikazanom primeru koristili smo konstante umesto objekata i pravili tipičan modularni program.

Metoda doručak nije mogla da stane na jedan slajd iako je imala samo po jedan red za svaku konstantu.

Eventualno proširenje bi značajno iskomplikovalo aplikaciju

Dodavanje nove metode ručak bi kreiralo još jednu slično ogromnu i komplikovanu metodu

Objektni primer

```
public class JelovnikObjektno {
    public JelovnikObjektno() {
        System.out.println("Za dorucak treba da jedete " + dorucak(new
O()));
    }
    private String dorucak(O o) {
        return "sunku";
    }
    private String dorucak(A o) {
        return "Grejp i sok od naradze";
    }
Objektni primer
    private String dorucak(B o) {
```

```
        return "sir i jogurt";
    }

    private String dorucak(AB o) {
        return "korn fleks i jogurt";
    }
}
```

Svaka metoda je izuzetno kratka, čak nekoliko metoda staje na isti slajd
Prošerenje ovakve aplikacije je jednostavno i efikasno
Dodavanje nove metode ručak znači dodavanje overload metode ručak što je relativno jednostavno
Kod je čist, jednostavan, čitljiv, proširiv, višestruko upotrebljiv