10. Изузеци и тврдње

Један од кључних циљева програмера је писање програма који немају грешке у фази извршавања или тзв. багове (енг. bugs). Такође, пожељно је да се програми не гасе без озбиљније потребе посебно у ситуацијама када људски животи зависе од тога или гашење може да изазове велики финансијски губитак, нпр. аутопилоти, контролни системи на железницама, берзански софтвер итд. Уколико би за написани програм постојао формални доказ (верификација) да никад не може да погреши онда би једини ризици за успешно извршење циљног посла били спољни фактори попут нестанка струје, временских непогода и слично. Реалност у већини програмских језика, а посебно у императивним, где припада и објектно-оријентисана парадигма, је другачија, и ту је врло вероватно да иоле комплекснији програми не могу техникама тестирања бити потпуно проверени. Из овог разлога, Јава прибегава употреби две корисне технике реаговања на непредвиђене ситуације:

1. Изузеци - механизам сигнализирања и пропратног реаговања на непредвиђену ситуацију - користан у фази употребе програма;
2. Тврдња - механизам обуставе рада програма у случају непредвиђене ситуације је искључиви приступ који има за циљ чување ингегритета програма - користан у фази развоја програма.

10.1. Изузеци

Изузеци представљају догађаје који се дешавају за време рада програма и чине одступање од нормалног (очекиваног) тока извршавања. Пошто они настају када у Јава програмима ствари крену наопако, изузетке треба узети у озбиљно разматрање када се дизајнира апликација и када се пишу програми. Две кључне користи од употребе изузетака су:

1. Раздвајање кода који обрађује неочекиване ситуације од кода који се извршава када ствари теку глатко.
2. Присиљавање програмера да размотри и реагује на одређене врсте грешака.

Не треба све грешке у програмима сигнализирати изузецима – само неуобичајене или катастрофалне. На пример, ако корисник не унесе исправан улазни податак, за то не треба користити изузетке. Разлог је што руковање изузецима укључује много додатног процесирања што успорава целокупан програм.

Унутар Јава програма, изузетак се моделује као објекат који у себи носи информацију о непредвиђеној ситуацији која се десила. За изузетак се каже да је “подигнут” или “избачен” (енг. thrown) унутар неке методе у којој се детектовала непредвиђену ситуацију. Надаље, изузетак (објекат) може бити:

1. пропагиран у позиве метода које су претходиле позиву методе у којој се десио;
2. обрађен унутар неке од метода које претходе позиву методе у којој се десио.

Пример 1. Пример демонстрира ситуацију у којој се дешава изузетак приликом приступа елементу низа ван граница низа и притом се изузетак обрађује унутар `main` функције.□

```
package rs.math.g10.p01.izuzeciIndeksVanGranica;

public class IndeksVanGranica {

    public static void main(String[] args) {
```

```
        try {
           int a[] = new int[2];
           System.out.println("Приступам елементу:" + a[3]);
        } catch (ArrayIndexOutOfBoundsException e) {
           System.err.println("Изузетак избачен:" + e);
        }
     }

}
```

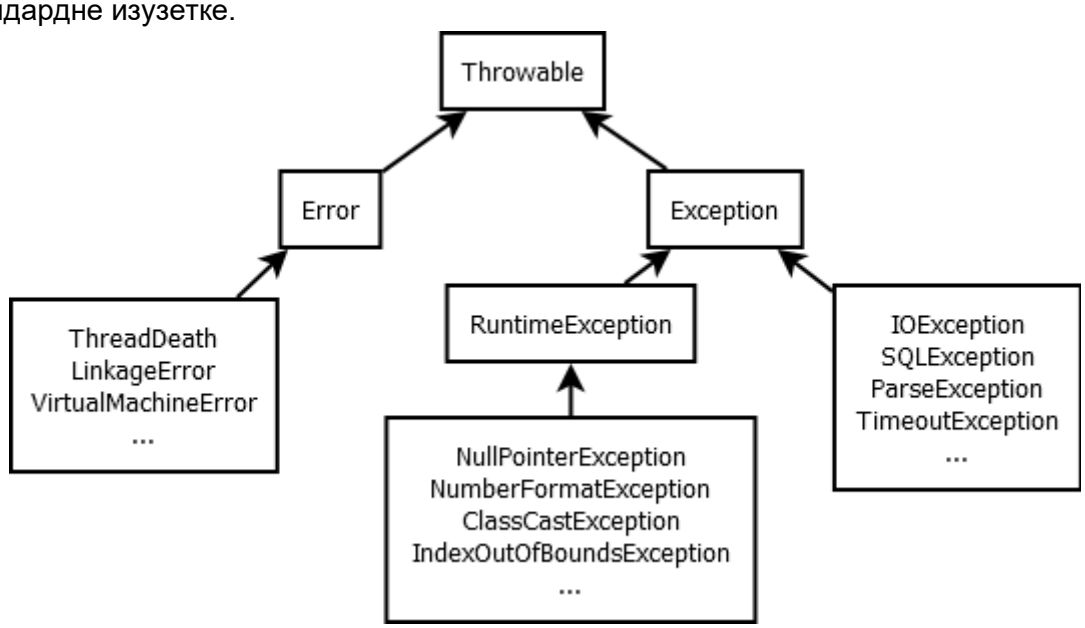
Изузетак објекат је у овом случају подигнут у оквиру оператора индексирања, односно током приступа елементу низа на задатом индексу.

```
Изузетак избачен:java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 2
```

Овај оператор, као и сви остали оператори, реализовани су у позадини Јаве као позиви метода тако да је реч најпре о пропагацији изузетка из оператора индексирања ка `main` методи, а потом и његовој обради унутар тела `main` методе. ■

10.1.1 Типови изузетака

Изузетак је увек објекат неке потклассе стандардне класе `Throwable`. То важи и за изузетке које сами дефинишемо, као и за стандардне, већ уграђене изузетке. Две директне потклассе класе `Throwable` – класа `Error` и класа `Exception` – покривају све стандардне изузетке.



`Throwable` класа и њене подкласе

Изузеци типа Error

Изузеци дефинисани класом `Error` и њеним потклассама карактеришу се чињеницом да се од програмера не очекује да предузима ништа, не очекује се да их хвата. Класа `Error` има неколико директних потклаcа, а неке значајније су:

- `ThreadDeath` – избацује се када се нит која се извршава намерно стопира.
- `LinkageError` – озбиљни проблеми са класама у програму, нпр. некомпатибилност међу класама или покушај креирања објекта непостојећег класног типа и слично.

- `VirtualMachineError` – садржи четири поткласе изузетака који се избацују када се деси катастрофални пад JVM.

Изузеци који одговарају објектима класа изведених из `LinkageError` и `VirtualMachineError` су резултат катастрофалних догађаја и услова. У таквим ситуацијама обично све што програмер може да уради јесте да прочита поруку о грешци која се генерише када се избаци изузетак, посебно у случају LinkageError изузетка. На основу поруке треба да покуша да схвати шта је у написаном коду могло да изазове такав проблем.

Пример 2. Пример демонстрира ситуацију прекорачења стек меморије `StackOverflowError` који је једна од подкласа класе `VirtualMachineError`.□

```
package rs.math.g10.p02.izuzeciPrekoracenjeStekMemorije;

public class FaktorijelBezUslovaIzlaska {

	static int faktorijelBezUslovaIzlaska(int n) {
		return n * faktorijelBezUslovaIzlaska(n - 1);
	}

	public static void main(String[] args) {
		try {
			System.out.println(faktorijelBezUslovaIzlaska(10));
		} catch (StackOverflowError e) {
			System.err.println("Ухваћена грешка:" + e);
		}
	}
}
```

Прекорачење настаје, јер рекурзивна метода за факторијел нема услов изласка из рекурзије па се креирање нових стек оквира за сваки наредни позив методе наставља све док се не потроши целокупна доступна стек меморија.

```
Ухваћена грешка:java.lang.StackOverflowError ■
```

Изузеци типа RuntimeException

За скоро све изузетке представљене поткласама класе `Exception`, мора се у програм укључити код који ће руковати њима уколико наш код може изазвати њихово избацивање. Изузетак су објекти класе `RuntimeException`. Преводилац допушта да их програмер игнорише јер они генерално настају због грешака у написаном програмском коду. Изузеци класе `RuntimeException`, дакле, указују на то да је нешто лоше у самој логици написаног програма.

Пример 3. Претходно наведени пример приступа елементу низа ван његових граница представља пример `RuntimeException`. У нешто измењеном сценарију у односу на поменути пример, нека корисник уноси индекс елемента низа којем жели да приступи. Уместо реаговања на такав изузетак, програмер би требао да се постара да до њега ни не дође тако што би код садржао проверу да ли тражени индекс уопште припада дозвољеном опсегу вредности - између нула и дужине низа умањене за један.□

```
package rs.math.g10.p03.izuzeciSprecavanjeIndeksaVanGranica;

import java.util.Scanner;

public class SprecavanjeIndeksaVanGranica {
```

```
	public static void main(String[] args) {
		int a[] = new int[] { 1, 2, 3, 4, 5 };
		System.out.println("Унесите индекс елемента којем приступате:");
		Scanner skener = null;
		try {
			skener = new Scanner(System.in);
			int i = skener.nextInt();
			if(i<0 || i>=a.length)
				System.err.println("Индекс ван граница.");
			else
				System.out.println("Број на траженом индексу је "+a[i]);
		} finally {
			if (skener != null)
				skener.close();
		}
	}
}
```

Тако се добија на пример следећи резултат у случају индекса ван граница.

```
Унесите индекс елемента којем приступате:
10
Индекс ван граница.■
```

Постоји јако велики број изузетака који су директне подкласе класе `RuntimeException`, а неки од познатијих су:

- `ArithmeticException` - недозвољена ситуација у примени аритметичких операција, нпр. дељење са нулом.
- `IndexOutOfBoundsException` - индекс низа којем се приступа није валидан.
- `NegativeArraySizeException` - алокација низа са негативном димензијом.
- `NullPointerException` - покушај приступа методи или пољу објектне променљиве која има специјалну референтну вредност `null`.
- `ClassCastException` - покушај конверзије објектне променљиве у недозвољени тип.

Пример 4. Код демонстрира ситуацију када се дешава `ClassCastException`.□

```
package rs.math.g10.p04.izuzeciKonverzijaUPogresanObjektniTip;

public class KonverzijaUPogresanObjektniTip {

	public static void main(String[] args) {
		Integer broj = 5;
		Object objekat = broj; //имплицитна конверзија ка општијем типу
		Boolean logickoSlovo = (Boolean) objekat;
	}
}
```

Резултат извршавања овог кода је информисање корисника о изузетку.

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
cannot be cast to class java.lang.Boolean (java.lang.Integer and java.lang.Boolean
are in module java.base of loader 'bootstrap')
	at
rs.math.g10.p04.izuzeciKonverzijaUPogresanObjektniTip.KonverzijaUPogresanObjektniTi
p.main(KonverzijaUPogresanObjektniTip.java:8)■
```

Јава компајлер не преиспитује, тј. не проверава овакве некоректне конверзије тако да се оне могу само детектовати, а не и спречити. Стога је на програмеру да овакве логичке проблеме отклони и спречи појаву оваквог и сличних изузетака - њиховим хватањем се не постиже пуно, тј. само се одлаже суочавање са логичким проблемом у развоју алгоритма.

Изузеци типа Exception

Карактеристика `Error` класе и њених потклаcа је да програмер не може нити да спречи нити да реагује и поправи ситуацију када се изузетак деси. Са друге стране, `RuntimeException` када се деси, могућа је поправка, али је она изнуђено решење и треба стремити ка спречавању оваквих изузетака. Стога у оба ова случаја Јава компајлер не захтева постојање кода за обраду оваквих типова изузетака. Када су у питању сви остали типови изузетака из класе `Exception`, ту је реч о ситуацијама које нису ни последица лошег програмирања попут `RuntimeException` нити су непоправљиве попут `Error`. Стога ће Јава компајлер проверити да ли је испуњена једна од ове две ствари у оквиру методе која може да избаци изузетак:

1. Хватање изузетка (`try-catch` блок);
2. Прослеђивање изузетка (наредба `throws`) надметоди, односно методи која је позвала актуелну методу.

Уколико није урађено ни једно ни друго, тада се програм неће превести. Постоји велики број директних подкласа класе `Exception` (изузев `RuntimeException`), а неке од њих су:

- `IOException` - општи изузетак у вези са улазно/излазним операцијама. Нпр. класа `FileNotFoundException` као потклаcа ове класе је везана за ситуацију када тражена датотека не постоји.
- `ParseException` - указује на проблем у парсирању текста.
- `SQLException` - општи проблем при повезивању на базу података или прављењу SQL упита, обради резултата упита и слично.
- `TimeoutException` - изузетак који се избацује када истекне време за чекање неке блокирајуће операције, најчешће се појављује у вишенитном програмирању.

10.1.2. Руковање изузецима

Претпоставимо да наш метод позива неки метод који може избацити изузетак који није типа потклаcе `RuntimeException` нити `Error` класе. Нека је изузетак нпр. типа `IOException`. Најмање што морамо да урадимо јесте да декларишемо да може бити избачен изузетак. Како се то ради? Једноставно се дода `throws` клауза у дефиницију метода, на пример:

```
double myMetod() throws IOException{...}
double myMetod() throws IOException,FileNotFoundException {...}
```

Дакле, само се дода кључна реч `throws` и листа изузетака који могу бити избачени, раздвојених запетама. Ако неки други метод позива овај метод, он мора да узме у обзир изузетке које овај може избацити, па ће их надаље или обрађивати или ће и он декларисати да избацује изузетке истог типа. Уколико се не уради ни једно ни друго, преводилац ће то утврдити и доћи ће до грешке превођења, па се Јава код неће превести у бајт-код.

Ако се одлучи да се рукује изузецима тамо где се они десе, потребно је укључити три врсте блокова кода у метод који рукује изузецима, и то су:

1. `try` блок – обухвата код где се може јавити један или више изузетака. Код који може да избаци изузетак који желимо да ухватимо мора бити у `try` блоку.
2. `catch` блок – обухвата код који је намењен да рукује изузецима одређеног типа који могу бити избачени у придруженом `try` блоку.
3. `finally` блок – увек се извршава пре него се метод заврши, без обзира да ли је било који изузетак избачен у `try` блоку или није.

try блок

Када треба да се ухвати изузетак, код метода који може избацити изузетак мора бити обухваћен `try` блоком. Код који може изазвати изузетке не мора бити у `try` блоку, па у декларацији метода треба бити истакнуто да тај метод може избацити типове изузетака који нису ухваћени. `try` блок чини кључна реч `try` за којом следи пар витичастих заграда које окружују код који може избацити изузетак.

```
try {
	// код који може избацити један или више изузетака
}
```

`try` блокови су неопходни и када желимо да се ухвате изузетци типа `Error` или `RuntimeException`.

catch блок

Код за руковање изузетком датог типа се ограђује `catch` блоком. `catch` блок се мора налазити непосредно иза `try` блока који садржи код који може избацити тај одређени изузетак. `catch` блок се састоји од кључне речи `catch` праћене једним параметром унутар облих заграда којим се идентификује тип изузетка којим блок рукује. Ово прати код за руковање изузетком који се налази унутар пара витичастих заграда:

```
try {
	// код који може избацити један или више изузетака
}catch(FileNotFoundException e) {
	// код за руковање изузетком типа FileNotFoundException
}
```

Вишеструки catch блок

У претходном примеру, `catch` блок је руковао само изузецима типа FileNotFoundException. То повлачи да је то једина врста изузетака која може бити избачена у `try` блоку. Ако могу бити избачени и други (изузев `Error` и `RuntimeException`), претходни код се неће успешно превести.

Генерално, параметар за `catch` блок мора бити типа `Throwable` или неке њене поткласе. Ако класа која се зада као параметар `catch` блока има поткласе, од `catch` блока се очекује да процесира изузетке тог типа, али и свих поткласа тог типа. Ако `try` може да избаци неколико различитих врста изузетака, тада је потребно поставити више `catch` блокова за руковање њима након `try` блока.

```
try {
	// код који може избацити један или више изузетака
}catch(FileNotFoundException e) {
	// код за руковање изузетком типа FileNotFoundException
}catch(IOException e) {
```

```
        // код за руковање изузетком типа IOException
}
// извршавање се наставља овде...
```

Приметити да је у претходном примеру други `catch` блок намењен хватању `IOException` изузетака који су надкласа изузетка `FileNotFoundException`. Ово значи да ако је изузетак типа `FileNotFoundException` или неке његове потклисе, онда ће први `catch` блок обрадити изузетак. У супротном, ће се проверити да ли изузетак припада хијерархији класа `IOException`, и након тога ће се обрадити у другом `catch` блоку. Дакле, уђи ће се у највише један `catch` блок. Приметити да је обрнути редослед, у којем је најпре наведено хватање `IOException`, а потом `FileNotFoundException` погрешан будући да се никад не би улазило у обраду `FileNotFoundException` изузетка, јер је сваки такав изузетак такође и потклиса класе `IOException`. Стога је у навођењу `catch` блокова потребно водити рачуна о хијерархији изузетака и увек прво наводити оне специфичније па потом оне општије.

finally блок

Природа изузетака је таква да се извршавање `try` блока прекида по избацивању изузетка без обзира на значај кода који следи тачку у којој је изузетак избачен. То уводи могућност да изузетак изазове неконзистентно стање делова програма. На пример, може се догодити да се отвори датотека и да се, пошто је избачен изузетак, не извршава код за затварање те датотеке.

`finally` блок обезбеђује средство да се ”почисти” на крају извршавања `try` блока. `finally` блок се извршава увек, без обзира да ли су или нису избачени изузеци за време извршавања придруженог `try` блока. Као и `catch` блок, тако је и `finally` блок придружен одређеном `try` блоку и мора бити смештен непосредно након `catch` блокова за тај `try` блок. Ако нема `catch` блокова, `finally` блок се смешта непосредно након `try` блока. Није могуће да постоји само `try` блок, већ њега увек мора да прати бар један од `catch` и `finally` блокова. Иначе се програм неће успешно превести. Уколико је коришћењем `return` наредбе враћена нека вредност унутар `finally` блока, то поништава `return` наредбу која је евентуално извршена у `try` блоку. Структура комплетне `try-catch-finally` наредбе:

```
try{
        // код који може избацити изузетке...
}catch(ExceptionType1 e) {
        // ...
}catch(ExceptionType2 e) {
        // …
        // ако је потребно, још catch блокова...
}finally{
        // код који се увек извршава након try-блока
}
```

Пример 5. Овај, нешто сложенији пример, демонстрира употребу `try-catch-finally` блокова приликом читања и парсирања датума са конзоле.□

```
package rs.math.g10.p05.izuzeciParsiranjeDatuma;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
import java.util.Scanner;

public class ParsiranjeDatuma {

	public static void main(String[] args) {
		Date datum = null;
		Scanner skener = null;
		DateFormat datumFormat = new SimpleDateFormat("dd.MM.yyyy");
		Boolean unetValidanFormat = false;
		try {
			skener = new Scanner(System.in);
			while (!unetValidanFormat) {
				try {
					System.out.println("Датум dd.MM.yyyy:");
					String datumString = skener.next();
					datum = datumFormat.parse(datumString);
					System.out.println("Валидан датум: " + datum);
					unetValidanFormat = true;
				} catch (ParseException e) {
					System.out.println("Погрешан формат датума!");
				}
			}
		} finally {
			skener.close();
		}
	}
}
```

У случају уноса формата датума који није у складу са очекиваним, програм ће детектовати изузетак и потом исписати грешку кориснику. За разлику од претходних примера, овде је приказан сценарио где се не зауставља програм већ се кориснику даје прилика да поново унесе података у траженом формату. Приметити да програм користи угњеждени `try-catch` блок унутар петље која се извршава зависно од тога да ли је индикатор за валидан унос добио вредност тачно. Спољни `try-finally` блок се брине како би се отворени Scanner објекат увек затворио без обзира на потенцијалне неухваћене изузетке. Пример једног извршавања претходног кода је:

```
Датум dd.MM.yyyy:
10-12-2022
Погрешан формат датума!
Датум dd.MM.yyyy:
10.12.2022
Валидан датум: Sat Dec 10 00:00:00 CET 2022■
```

10.1.3. Пропагирање изузетака

У многим ситуацијама, када се у неком методу појави изузетак, програмер може да одлучи да тај изузетак не обрађује на у том методу већ да га пропагира у позивајући метод. Мотивација за такву одлуку је што позивајући метод може бити свеснији контекста у ком је изузетак настао, па се на том нивоу лакше може одлучити које акције треба предузети. Пропагирање изузетка у метод-позивалац се реализује помоћу кључне речи `throws` иза које следи листа изузетака којима је допуштено пропагирање. **Пример 6.** Овај пример представља модификацију претходног у којем је за парсирање датумског стринга издвојена посебна статичка метода. С обзиром да ова метода нема контролу над главном петљом у којој се покушавају поновни уноси од стране корисника,

пропагација изузетка у `main` методу има више смисла него реаговање на изузетак унутар ње.□

```
package rs.math.g10.p06.izuzeciPropagiranje;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

public class ParsiranjeDatumaPropagiranjeIzuzetka {

	final static DateFormat datumFormat = new SimpleDateFormat("dd.MM.yyyy");

	static Date parsirajDatum(String datumString) throws ParseException{
		Date datum = datumFormat.parse(datumString);
		return datum;
	}

	public static void main(String[] args) {
		Date datum = null;
		Scanner skener = null;

		Boolean unetValidanFormat = false;
		try {
			skener = new Scanner(System.in);
			while (!unetValidanFormat) {
				try {
					System.out.println("Datum dd.MM.yyyy:");
					String datumString = skener.next();
					datum = parsirajDatum(datumString);
					System.out.println("Валидан датум: " + datum);
					unetValidanFormat = true;
				} catch (ParseException e) {
					System.out.println("Погрешан формат датума!");
				}
			}
		} finally {
			skener.close();
		}
	}
}
```

■

10.1.4. Избацивање изузетака

До сада је било речи о реаговању на изузетке и њиховој пропагацији. Да би се било шта од ова два десило, изузетак се најпре мора избацити. Методе унутар стандардних Јава класа које избацују изузетак увек у својим имплементацијама имају наредбу која изазива избацивање изузетка и та наредба се зове `throw` (без "s" на крају). На пример, реализација функције дељења у Јави, скраћено записана као оператор "/", поседује у себи проверу да ли је делилац нула, и ако јесте, применом наредбе `throw` се креира објекат класе `ArithmeticException` са одговарајућом поруком да је у питању дељење

нулом. Слично, програмери могу креирати сопствене изузетке за ситуације које су прилагођене логици програма који пишу.

Пример 7. Пример демонстрира употребу новокреиране класе за изузетке који се дешавају при раду са дводимензионим низовима. Конкретно, сигнализира се неусаглашеност спољашњих или унутарашњих димензија прослеђених дводимензионих низова приликом њиховог сабирања.□

```
package rs.math.g10.p07.izuzeciKorisnickiDefinisan;

public class Niz2DIzuzetak extends Exception{

	public Niz2DIzuzetak(String poruka) {
		super(poruka);
	}

	static double[][] saberi2DNizove(double[][] a, double[][] b) throws
Niz2DIzuzetak{
		if(a.length!=b.length)
			throw new Niz2DIzuzetak("Лоше спољне димензије.");
		double[][] c = new double[a.length][];
		for(int i=0; i<a.length; i++) {
			if(a[i].length!=b[i].length)
				throw new Niz2DIzuzetak("Лоше унутрашње димензије.");
			c[i]=new double[a[i].length];
			for(int j=0; j<c[i].length; j++)
				c[i][j]=a[i][j]+b[i][j];
		}
		return c;
	}

	public static void main(String[] args) {
		double[][] m1 = new double[][] {{1,2,3}, {4,5,6}};
		double[][] m2 = new double[][] {{1,2}, {3,4},{5,6}};
		double[][] m3 = new double[][] {{1,1,1}, {1,1,1}};
		try {
			double[][] m4 = saberi2DNizove(m1, m3);
			System.out.println("Прво сабирање успело.");
			double[][] m5 = saberi2DNizove(m1,m2);
			System.out.println("Друго сабирање успело.");
		} catch (Niz2DIzuzetak e) {
			System.err.println(e);
		}
	}
}
```

Испис након извршавања је:

```
Прво сабирање успело.
rs.math.g10.p07.izuzeciKorisnickiDefinisan.Niz2DIzuzetak: Лоше спољне димензије.■
```

Слично овоме, програмер је могао и да не креира нови тип изузетка већ да искористи неки постојећи па чак и онај најопштији класе `Exception`. Међутим, због будућег суптилнијег реаговања на изузетке и употребе вишеструких catch блокова најчешће је смисленије имати некакву хијерархију кориснички дефинисаних изузетака који одговарају логици програма.

10.1.5. Препоруке за коришћење изузетака

У претходном тексту су већ наведене неке препоруке за употребу изузетака. Неке од најбитнијих су:

- употреба `finally` блока са или без постојећих catch блокова у циљу затварања отворених ресурса, нпр. `Scanner` објекта;
- изузетке који су подкласа класе `RuntimeException` као и класе `Error` не треба хватати - прве из разлога што указују на грешку у програмирању што значи да је потребно спречити њихову појаву, а друге, с обзиром на то што њиховим хватањем најчешће не можемо ситуацију да поправимо попут грешке `StackOverflowError`;
- преферирати што специфичније изузетке који боље описују проблематичну ситуацију - ово се спроводи не само кроз креирање нових класа за изузетке већ и кроз адекватно описивање проблема путем поруке која се прослеђује кроз конструктор;
- унутар вишеструких `catch` блокова најпре хватати специфичне изузетке па потом општије с обзиром да се улази у највише један catch блок па би обрнути редослед довео до потпуног игнорисања специфичнијих изузетака;
- не треба хватати најопштији могући изузетак односно класу `Throwable` - ово ће ефективно маскирати појаву било каквих проблема у раду апликације што значи да ће програм настављати да ради, али са потенцијално озбиљним проблемима који су занемарени попут преоптерећења меморије;
- у фази програмирања не игнорисати изузетке, посебно изведене из класе `RuntimeException`, јер они често указују на проблеме који ће у фази употребе програма постати још учесталији и већи;
- не претеривати са употребом, односно избацивањем изузетака будући да њихова пропагација и хватање има своју цену односно изазива одређено успорење у раду програма.

10.2. Тврдње

Тврдња је наредба која омогућава проверу испуњености неког услова у програму. На пример, може се проверавати ненегативност индекса елемента у низу, да ли је површина круга позитивна, да ли је брзина кретања честице у симулацији мања од брзине светлости и слично. Уколико услов није испуњен програм аутоматски избацује грешку и прекида се. Овај ригорозни приступ проверавању испуњености услова помаже програмеру током програмирања у разрешавању грешака и повећавању степена поверења у то да програм заиста ради оно што се очекује.

10.2.1. Наредба assert

Тврдње се реализују помоћу наредбе `assert` и `java.lang.AssertionError` класе. Тврдња започиње кључном речи `assert` након чега следи логички израз:

```
assert [логички израз];
```

Ако је логички израз тачан, ништа се не дешава и извршавање се наставља од наредне наредбе. Са друге стране, ако је логички израз нетачан, креира се објекат класе `AssertionError`. Програм провера у току свог рада тврдње само ако су оне активиране - што се постиже задавањем аргумента `-ea` (енг. enable assertions) виртуелне машине током извршавања. На овај начин се једноставно, без измене програмског кода може прелазити из решима употребе у режим програмирања и отклањања грешака.

Пример 8. Пример демонстрира неиспуњеност услова. □

```
package rs.math.g10.p08.tvrdnjeNenegativanBroj;

public class NenegativanBrojTvrdnja {
          // покренути програм са аргументом -ea виртуелне машине (VM)
          public static void main(String[] args)
          {
             int x = -1;
             assert x >= 0;
          }
}
```

Испис након извршавања доста подсећа на онај који се добија приликом изузетка који није обрађен, тј. ухваћен у програму:

```
Exception in thread "main" java.lang.AssertionError
       at
rs.math.g10.p08.tvrdnjeNenegativanBroj.NenegativanBrojTvrdnja.main(NenegativanBrojTv
rdnja.java:8)
```

Пример 9. Пример демонстрира неиспуњеност услова, али сада са додатном поруком коју задаје програмер. □

```
package rs.math.g10.p09.tvrdnjeNenegativanBrojSaPorukom;

public class NenegativanBrojTvrdnjaSaPorukom {
         // покренути програм са аргументом -ea виртуелне машине (VM)
          public static void main(String[] args)
          {
             int x = -1;
             assert x >= 0 : "x < 0";
          }
}
```

Испис:

```
Exception in thread "main" java.lang.AssertionError: x < 0
       at
rs.math.g10.p09.tvrdnjeNenegativanBrojSaPorukom.NenegativanBrojTvrdnjaSaPorukom.mai
n(NenegativanBrojTvrdnjaSaPorukom.java:8)
```

Честа употреба тврдњи је у проверавању такозваних предуслова или постуслова односно стања пре или после извршавања неке методе.

Пример 10. На пример, ако програмер реализује методу за сортирање низа очекивано је да ће стање низа након извршавања те методе (постуслов) бити сортирано. □

```
package rs.math.g10.p10.tvrdnjeSortiranjePostuslov;

public class SortiranjeSaPostuslovom {
```

```
        // покренути програм са аргументом -ea виртуелне машине (VM)
          public static void main(String[] args)
          {
             int[] niz = { 20, 91, -6, 16, 0, 7, 51, 42, 3, 1 };
             sortiraj(niz);
             for (int e: niz)
                System.out.printf("%d ", e);
             System.out.println();
          }

          private static boolean jeSortiran(int[] x)
          {
             for (int i = 0; i < x.length - 1; i++)
                if (x[i] > x[i + 1])
                   return false;
             return true;
          }

          // сортирање уметањем
          private static void sortiraj(int[] x)
          {
             int j, a;
             for (int i = 1; i < x.length; i++)
             {
                a = x[i];
                j = i;
                while (j > 0 && x[j - 1] > a)
                {
                   x[j] = x[j - 1];
                   j--;
                }
                x[j] = a;
             }
             assert jeSortiran(x): "низ није сортиран";
          }
}
```

Будући да је сортирање уметањем у овом примеру исправно реализовано, на излазу неће бити пријављена грешка, већ ће бити исписан низ сортираних бројева.

```
-6 0 1 3 7 16 20 42 51 91
```

10.2.2. Препоруке за коришћење тврдњи

Битно је уочити разлику између намене тврдњи и намене изузетака. Тврдње се користе како би програм у току извршавања указао на немогуће односно на потпуно неприхватљиве околности. С тога је потребно да програмер поправи програм и избегне такве околности - то опет не гарантује да проблем ради оно што је потребно, али се добром “покривеношћу” тврдњама смањује шанса да ће нека неприхватљива околност да се деси у фази употребе. Са друге стране, изузеци се користе када је реч о грешкама који су зависни и од других фактора, а не само од програмерске логике попут стања система датотека, неадекватне обраде корисничких уноса и слично.

Тврдње нису замена за изузетке и за разлику од њих не подржавају суптилно реаговање на проблем, односно његову обраду - већ једноставно долази до заустављања програма са евентуалним исписом прилагођене поруке. Тврдње се обично у фази употребе

програма (продукционом окружењу) онеспособљавају, односно програми се обично извршавају без параметра `-ea` виртуелне машине.

10.3. Резиме

У овом поглављу објашњени су концепти изузетка и тврдње. Систем изузетака је настао из потребе за транспарентнијим и суптилнијим начином реаговања на проблематичне сценарије у току извршавања програма. Кроз конструкције try-catch-finally програмеру је омогућено да на читљив начин реагује на грешку или алтернативно да одложи реаговање и пребаци га у контекст неке друге методе на линији позивања употребом кључне речи throws. Изузеци као објекти могу у себи садржати разноврсне корисне информације које указују на узрок проблема, а додатно применом наслеђивања је могуће правити још јаснију диференцијацију њихове намене. Са друге стране, тврдње представљају најригорознији приступ провери коректности програма унутар императивних језика и њихов циљ је да помогну програмеру у фази развоја програма, а не у фази његове употребе (продукционом окружењу).

10.4. Питања и задаци