УВОД У ПРОГРАМИРАЊЕ

Едиција:

„**Red Monkey**“

САДРЖАЈ:

Увод

Нескривено себични и претенциозни циљ ове књиге је пре свега развој **логичког**, **аналитичког** и **креативног** начина размишљања код ученика, што по мом скромном мишљењу чини део главних **компетенција** које треба да има свако, ко мисли да успе не само у програмирању, него и у било ком другом послу у реалном времену и свету око нас. Да би то остварио, сваки ученик мора да оствари **фокус** на материји која му се представља (између осталог и путем ове књиге), као и висок степен **концентрације** током трајања курса подељеног између теоријске и практичне наставе. По искуству, ове две ствари обично недостају већини ученика.

У циљу што бољег савладавања **основа програмирања**, сваки ученик би пожељно требао да има следеће **навике**:

1. Вежбања – што више практичног програмирања, увек из угла решавања конкретних проблема.

2. Проучавања – погледајте како су други програми написани. Читањем кода написаног од стране других програмера, може се доста научити. Никад при том не заборавите, да нема најбољег решења, већ само оног које је боље у одређеном контексту (*пример: најбрже, најмање, најлакше за коришћење и сл.*).

3. Навика да будете упорни и постојани – програмирање је тежак али интересантан посао. Већина не успе у програмирању, не зато што су глупи, већ што нису довољно упорни. Као и у свему не претерујте ни у упорности. Ако за 30 минута не решите проблем, или потражите помоћ, или пустите да проблем пренoћи. Решавањем (*безуспешним*) проблема целу ноћ, све што ћете имати ујутру су умор и изиритираност.

Anders Hejlsberg[1], главни мозак и дизајнер **C#** језика:

> „Немој бити уплашен. Само зато што ти људи говоре да нешто не може бити урађено, не значи да је нужно и тако. То само значи да они то не могу даураде. Ја мислим да је увек забавно размишљати „ван калупа" и налазити нова решења за постојеће проблеме.
> Ја мислим да је једноставност увек добитна комбинација. Могућност проналаска једноставнијег решење за нешто је мој водећи принцип. Увек пробај да направиш једноставније. Мислим да би био добар у нечему, мораш да будеш страствен у томе.
> То је нешто што не можеш да научиш.
>
> Ја сам упао у програмирање, не да би направио гомилу пара или зато што ме је неко на то наговорио. Ушао сам у то тако што сам био потпуно усисан програмирањем. Једноставно ме нисте могли зауставити. Морао сам да пишем програме. То је била једина ствар коју сам желео да радим. Био сам, веома, веома страствен у томе. И морате да имате ту страст да би сте постали заиста добри у нечему".

[1] Исечак из интервјуа са Федерико Бјанкузијем [Federico Biancuzzi], 2008.

1. Компјутери и програми

Људи уз помоћ компјутера обављају тренутно готово све озбиљније послове, а свет око нас је све више „компјутеризован“. Тако нас уопште више не изненађују најновије вести, типа да је **LG**, реномирана Јужно-Корејска компанија позната по врхунским електронским уређајима широког асортимана (*као и слогану „**Life is Good**“*) избацила најновији фрижидер са **Windows 10 системом**, уз помоћ кога на пример можемо између осталих ствари и да учинимо врата потпуно прозирним. Већ видим разговор два моја типична ученика:

> ***Ђоле*** *(усплахирено): „**Миле**, дођи брзо, купио сам LG фрижидер!!!“*
>
> ***Миле*** *(хладнокрвно): „**Ђоле**, кул је све то, искулирај нам једну кокишку за Нолетов меч сутра, не могу сада, нешто сам поспан.“*
>
> ***Ђоле*** *(скоро вриштећи): „Ма **Миле**, одмах, да видиш како сада добро ради Counter Strike!!!“*

Углавном, овде морамо да поставимо ствари **одмах** на своје право место.

1.1. Компјутери

Сам компјутер у ужем смислу речи, представља хардверски уређај, метално-електричну кутију, која често има и лош обичај да хучи. Добра ствар је кад нас изнервира и ако одлучимо да га шутнемо, можемо само да поломимо ногу.

Чекај мало, шта је ту добро??? Па то је мојаааа нога!!! Ах, да, добро је то што је наш компјутер исто тако добрица по нарави, па нам неће узвратити ударац.

Успут, сваки програмер ће вам ноншалантно рећи да је при том, тај исти компјутер и прилично глупа машина, и да сам не уме ништа да ради а камоли уради. Али бивајући добрица, он (*мој компјутер*) ми неће замерити речи о глупости, вероватно мислећи то исто за мене, а поготово у тренуцима када му издајем лоше наредбе.

Најинтересантнији део те хардверске приче је инжењерски аспект. Ако знамо да је и сам Никола Тесла био пре свега иноватор, изумитељ, али и инжењер, а и ја сам дупли (*скромност ми је мана, што сте већ и сами увидели*) инжењер, цела прича почиње да поприма одређену димензију. Тесла је, и преко струје и преко јединствене визије комуникације, чијег смо значаја тек сада свесни (*интернет*), убрзао развој самих компјутера и информационо-комуникационих система, у овом облику у ком су сада нама познати.

? :
Да је Тесла тада могао да користи садашње компјутере у свом процесу креирања иновација и изума, да ли би цела садашња Африка била снабдевена бесплатном енергијом, а мој лични рачун за потрошену електричну енергију приближан цени мог омиљеног сладоледа?

Компјутер у том смислу, представља један својеврстан инжењерски подухват и подвиг, где је основни циљ био направити што **бржу** (*време*) и у исто време **енергетски** што ефикаснију машину (*шта сам вам рекао о двојцу без кормилара: време-енергија*). Обилазак једне од Гугл (*Google*[2]) серверских соба (*читај фабрика*) би вам то лепо илустровао, а ако волите уз то и да се играте, могли бисте да одиграте једну партију рата звезда у мору зелених (*добри момци*), жуто-наранџастих (*узбуњивачи*) и црвених (*лоши момци, можда зарадите отказ ако сте несрећни администратор у тој соби*) лампица. За рат звезда симулацију, потребне су вам још и саме звезде, али само не тражите од већ намученог администратора да вам изнесе све сервере на простор под отвореним ведрим небом, јер ћете се одмах ви сами наћи испод истог.

1.2. Програми

Када говоримо о компјутерима у свакодневном животу, обично мислимо на уређаје напаковане са најновијим оперативним системом и нашим омиљеним програмима. Будимо свесни да само **програми** у том пакету имају неку стварну употребну вредност, упркос томе што те исте програме не можемо ни да опипамо, ни да шутнемо, као што смо могли да радимо са нашим компјутером (*пре стављања гипса наравно*).

Ω Дефиниција 1: Програм
Програм представља скуп инструкција којима говоримо компјутеру шта тачно да ради.

Апликација, **код** или **софтвер** су алтернативни изрази за **програм**, коришћени зависно од контекста, ситуације или личних преференција. **Наредбе**, **команде** или **изјаве** су алтернативни изрази за **инструкције**, коришћени такође зависно од контекста, ситуације или личних преференција. Аналогно, сам компјутер можемо дефинисати у складу са претходно наведеном дефиницијом:

Ω Дефиниција 2: Компјутер
Компјутер је уређај који процесуира информације у складу са задатим, прецизним инструкцијама.

[2] *Светски технолошки див бр.1*

Процесор процесуира, а обрађивач обрађује, мала дигресија на избор речи у претходно наведеној дефиницији компјутера.

Коначно, **програми** су увек резултат **процеса програмирања**.

Ω Дефиниција 3: Програмирање

Програмирање представља процес креирања и испоруке решења одређеног проблема у форми коју компјутер може да разуме и изврши.

Напомена:

Програмирање је иначе обично дефинисано од стране већине људи као зарађивање велике суме новца радећи нешто што нико не може да разуме.

Ω Дефиниција 4: Процес

Процес представља серију акција које производе нешто или воде ка одређеном резултату.

Ω Дефиниција 4б. Процес (системски приступ):

Процес представља структуирани скуп активности дизајнираних да постигну неки циљ.

Ако не видите разлику између 4а и 4б, нећу да вас пуно кривим.

Сваки **процес** прихвата једну или више дизајнираних **улазних величина** и претвара их у дефинисани **излазни резултат**. Магија је у временском моменту претварања. На примеру пице, то је један термички процес у рерни (загрејаној на 220 степени Целзијуса), када се услед топлоте тестенина трансформише у једну врелу мирисно укусну и у том тренутку неодољиву пицу.

Претходна дефиниција самог појма процеса је изузетно важна да би смо схватили и сам компјутер и наше програме. Наиме и они се управо тако понашају. Прихвате неке податке и трансформишу их у нама жељене и (*обично преко*) потребне информације.

Дијаграм 1. Програм као процес

1.3. Како до програма?

Ако хоћемо да компјутер уради тачно оно што желимо, морамо да набавимо програм. Иако постоји маса бесплатних програма, велика је вероватноћа да ће наша жеља за програмом морати да има и одређену финансијску подршку (кеш).

Проблем: Док седите у вашем омиљеном кафићу и размишљате опуштено о оној истој Деветој планети коју смо већ поменули, прилази вам од вашег познаника познаник по препоруци, пошто му је речено да сте ви програмер и да може да вам се обрати са својим проблемом (куд баш сада, рекли би сте ви, али позната је ствар да живот пише романе). Наиме он је развио успешан посао продавајући прозоре са дупло ојачаним стаклом. Сада је вољан да део од прихода уложи у програм за израчунавање укупног трошка материјала потребног за израду сваког од тих прозора. Наиме, рачунајући тај трошак на досадашњи начин, увидео је да троши превише свог драгоценог времена, које би му знатно више користило у прибављању нових клијената.

Иако вас овде мами могућност брзе и лаке зараде, поготово због пристиглих рачуна и предстојећег Божићног празника, задржаћете у сваком моменту професионални став и објаснити вашем клијенту да је најлакше, најбрже, а обично и најјефтиније да се купи готов програм. Постоји десетине хиљада програма на тржишту, за готово све области људског деловања и интересовања. Најчешћи проблем овог приступа је што обично, готов купљени програм не задовољава све наше захтеве и потребе. У том случају мора да се напише нови програм. Знао је овај познаник познаника да дође на право место.

Да резимирамо овде: постоје три начина (*mpehu?*):

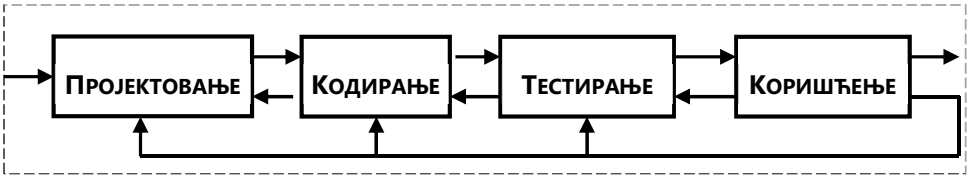
1. Купити постојећи готов програм (и надати се да програм ради у потпуности у складу са нашим потребама)
2. Написати нови програм (ето прилике за нас)
3. Купити програм са изворним (source) кодом, и допунити га недостајућим функцијама.

Иако могућ, овај трећи начин је доста редак, па га на овом месту помињемо само из академских разлога.

1.4. Процес програмирања

Резултат овог процеса је наравно сам **програм**, а што се тиче скупа главних **активности** имамо:

1. Пројектовање програма
2. Писање
3. Тестирање
4. Коришћење програма



Дијаграм 2. Процес програмирања

1.4.1. Пројектовање (дизајн) програма

У свим фазама програмер и наручилац програма морају да делују као један тим.

Ми ћемо наравно добро да саслушамо нашег клијента, и све то лепо да забележимо.

Најважније је да схватимо шта је тачно проблем који решавамо.

Не ретко, напишу се програми који решавају оно што не би требали, а то је погрешно дефинисан или добро дефинисан али споредан проблем.

Ви као програмер, морате да будете свесни да већина клијента зна да им треба нека врста програма, да имају проблем (*или чак више њих*), и то би било углавном све што један просечан клијент зна! Ваш задатак је да уз помоћ интервјуа, састанака и разговора са свим учесницима у том пројекту (*а креирање програма јесте један специфичан пројекат*) сазнате шта је тачно проблем који ће ваш програм да реши.

Ту вам је потребна сва логика, аналитика и креативност које смо помињали.

Кад сте у овом делу начисто о природи проблема, припремићете нешто што се зове **спецификација функционалног дизајна**, и натерати вашег клијента да то и потпише, јер је тај документ основа онога што ћете наплатити вашем клијенту, једном када ваш програм стварно и испоручи све пописане и захтеване функције.

У следећем кораку, а у зависности од претходно дефинисане функционалности, можемо да дефинишемо и све улазно-излазне податке и одредимо начин чувања самих података.

На крају фазе пројектовања бирамо **архитектуру** програма, и алат, пре свега избором програмског (може, а често и мора више) језика који ће се користити за писање самог програма.

Од методологија, најефектнија и најпопуларнија је свакако метода **објектно-оријентисаног програмирања (ООП)** [види **1.6.**]. За решавање конкретних проблема најчешће се користи нешто што зовемо алгоритми [види **1.7.**].

1.4.2. Писање програма

Имамо дакле изабрани **програмски језик**, никад доста времена, енергије не мањка, и проблем по проблем, линија по линија кода, ми напишемо све **наредбе** које чине наш **програм**. Једном написан програмски код (аналогно књигама као књижевним делима), за сваког субјективног и сујетног програмера представља и мало уметничко дело, па се упркос финансијској надокнади од стране нашег клијента, тешко растаје од њега.

Али мора и да се живи.

Све што је потребно сада да довршимо у овој фази је да такав изворни код **преведемо** уз помоћ другог **програма** који се назива **преводилац** (*Compiler [компајлер]*), и кога ћемо у наставку да зовемо само компајлер. Тај компајлер у процесу превођења трансформише **изворни код** у **бинарни код** (*нула, један, нула...*).

У фази изградње програма (***build***) генерише се један (*и само један*) стартни извршни фајл. У већини случајева генерише се још хрпа повезаних фајлова, што извршних, што помоћних. Током читаве ове фазе разликујемо још један **потпроцес** (*ах, процеси у налету са децом потпроцесима*) **откривања грешака** или **дебаговања** (*debugging*).

Баг (*bug*) је програмска **грешка**.

Разликујемо три врсте грешака: 1. **Синтаксне** 2. **Логичке** грешке 3. Грешке **намере**.

Следећа фраза демонстрира прве две: "*Постоје две гррешке у овој реченици.*"

Прва грешка је синтаксна, јер је реч грешка погрешно написана. **Синтакса** се односи на **правопис** и **граматику** језика. Друга грешка је логичка, а ви ћете је сами открити.

Наш компајлер, наш бади (*body, друг који помаже на енглеском*) сам открива **синтаксне грешке** а већином их сам и отклања.

Логичке грешке су теже, јер захтевају више времена за проналажење и отклањање.

Грешке намере (*немојте их мешати са намерним грешкама*) су грешке које се дешавају приликом извођења одређене намере током извршавања програма.

Пример намере је покушај приказивања слике која је у међувремену избрисана или премештена. Ове грешке се избегавају тако што се пише посебан код за сваку намеру у програму, а где постоји шанса да та намера крене по злу у реалном времену извршавања програма.

На примеру фотографија, наш програм би требао уместо да се сруши због недостајуће слике, да понуди кориснику дијалог уз помоћу кога ће он (*корисник*) да одреди нову централну локацију албума фотографија, где је слика премештена (*сада већ можда и случајно, можда и намерно, али то наш програм и не треба да интересује превише*).

1.4.3. Тестирање програма.

У овој фази треба испитати програм у условима његовог рада, тако што се симулирају све могуће комбинације улазних података, и контролише вредност излазних резултата који треба да одговарају дефинисаним функцијама које врше трансформацију улазних у излазне величине.

1.4.4. Коришћење програма.

Из ове фазе имамо два излаза, као што је приказано на дијаграму 2: један је крај програма, а други је улаз у нови циклус развоја програма, кажемо обично или отклањање проблема у претходном дизајну, или избацивање нове побољшане верзије програма у складу са захтевима тржишта, а пре свега услед притиска конкурентских програма и њиховог развоја.

Најбољи део ове фазе је да смо сада већ потпуно наплатили наш програм, као и да смо спремни за предстојеће Божићне празнике.

1.5. Програмски језици

Ω Дефиниција 5: Језик

Језик је средство за представљање и преношење информација, као и за комуникацију између два или више корисника.

1.5.1. Језик као средство комуникације

Језике можемо поделити на природне, програмске и остале у које спада између осталих и језик димних сигнала северно-америчких домородаца званих Индијанци.

Природне језике користе људи и постоји више стотина језика ове врсте, као што су: српски, енглески, француски, шпански, немачки, кинески, јапански и сл.

Програмски језици су специјални језици створени за потребе комуницирања са компјутером и креирање компјутерских програма.

Као и природни језици, програмски језици има своју стриктну **граматику**, **синтаксу** и **семантику**. Поштовањем тих правила у стању смо да комуницирамо са компјутером.

Граматика чини скуп правила која одређују како се над скупом симбола граде елементарне и сложене конструкције језика.

Синтакса чини скуп правила за формирање правилних конструкција језика.

Семантика изучава значење конструкција језика.

Напомена:

Пошто познавање језика представља право мало богатство, писац ових редова поносно изјављује да говори по важећој спецификацији 9 природних, 9 програмских, а као 19-ти у својој биографији наводим и претходно поменути језик димних сигнала.

Већина програмских језика је императивна, наредбодавног карактера, али имамо и декларативне језике, када се специфицира жељени резултат, али не и како се долази до њега.

Прва генерација језика су били машински језици.
Друга генерација су асемблерски језици.
Трећа, четврта и пета генерација су програмски језици високог нивоа.

Главни проблем комуникације између човека и компјутера је тај што компјутер разуме само свој тешко разумљиви машински језик у виду бесконачне секвенце нула и јединица – 11000111 11100110 10000001. Ако бих вам рекао да ових 24 нула и јединица представља мој тренутни баланс на банковном рачуну, али и име вашег омиљеног глумца или глумице, схватићете зашто је компјутер глуп и тежак за сарадњу.

Како год било, ову ситуацију ћемо премостити уз помоћ наших програмских језика, који савршено обављају суштину сваке комуникације сажету у следећем: ***како са што мање речи што више, а да нас у исто време друга страна са којом комуницирамо савршено тачно и разуме***.

Програмски језици су састављени од (*углавном*) симбола енглеског алфабета, стотинак кључних речи (*такође углавном преузетих из енглеског језика*) и још гомиле специјалних симбола, где су тачка и зарез само неки од карактеристичних симбола. Овако формиран језик изгледа знатно прихватљивији за човека, али је потпуно неприхватљив за компјутер.

У зависности од имплементације програмског језика имамо две главне групе:

- Компајлирани (преведени) и
- Интерпретирани

Зато уз програмске језике иду и **преводиоци**, посебни програми који претварају текст разумљив људима у серију нула и јединица, облику који је потпуно разумљив компјутеру.
Постоји више десетина, ако не и стотина програмских језика од којих су неки од најпознатијих у породици (фамилији) C, па тако: лично његово височанство C[3], али и C++[4], C#[5], Java и сл. Језици веба такође никад популарнији и траженији.
Да набројимо неке од њих као што су: HTML/CSS, Java/JavaScript, PHP, Python, Ruby и сл.

[3] коришћен пре пола века за прављење нпр. чувеног Unix оперативног система
[4] овде смо нешто додали, сигнализирају два плуса
[5] чиста музика, е-фис-гис-цис E-F#-G#-C#

1.5.2. Програмски језик Pascal

Напоменули смо претходно да је **Паскал** програмски језик који се традиционално користи на нивоу средње-школског образовања у Републици Србији, а пошто сам ја покрио 17 година, одговорно тврдим овде да је то и истина. Да ли бих га ја променио, то вам не могу открити овде, јер сам емотивно везан за сваки од мојих 19 језика, укључивши и сам Паскал.

Паскал припада компајлираним програмским језицима високог нивоа, тзв. језицима треће генерације. Такође подржава већину главних парадигми[6] програмирања као што су императивно, структурно, процедурално, функционално програмирање или програмирање вођено догађајима и објектно-орјентисано програмирање.

Паскал је развијен од стране швајцарског професора Niklaus Wirth-a. Wirth је дао име свом језику по француском физичару, математичару и филозофу Blaise Pascal-у, творцу прве механичке рачунске машине за сабирање.

Паскал је представљен 1965 године.

Прва верзија језика је објављена 1968. године (*и то је била година* ***Браон Мајмуна***), са преводиоцем написаним у FORTAN-у[7].

Друга верзија преводиоца написана је у самом Паскалу, и објављена је 1971. године. Напокон, 1974. године објављена је сређена верзија која се узима као стандард, и прави почетак **Паскал** програмског језика.

Паскал је, уз **Basic**[8] језик, и језика поменуте C фамилије, један од најпопуларнијих и најраширенијих језика намењен студентима за стицање ***основа програмирања***.

Услед сталног развијања самог језика, могућности **Паскала** су знатно повећане, тако да је могуће правити и знатно комплексније програме, који се по брзини извршавања готово могу мерити са програмима написаним у **C** језику.

На основама Паскала развијен је рецимо и програмски језик **ADA**, а као директан наследник у ери Windows-а **Delphi**. Постоји и сада приличан број програмера којима је Паскал основни језик у коме раде. И за оне који се определе за други језик, учење основа програмирања у **Паскалу** није губитак, већ једно значајно искуство. Прелазак на било који други језик биће знатно лакши.

---

[6] Најопштији модел по којем се граде поједини принципи и закони

[7] ***For****mula* ***Tran****slator*

[8] **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode [1965]

1.6. Објектно-оријентисано програмирање (ООП)

Код иоле већих, комплекснијих програма усвајање метода креирања програма, од дизајна до имплементације је изузетно важно. Један (али не и једини) метод доказан у пракси је свакако метод обијектно-орјентисаног програмирања.

Код овог метода решавање проблема и креирање програма се изводи преко дизајна заснованог на **објектима**.

Објекти су генерално свуда око нас, па пошто наши програми решавају проблеме реалних система, онда је и најлогичније да се и они састоје од тих истих објеката, само у нама прикладној верзији потребној да се реши специфични проблем.

1.6.1. Шта је то објектно-оријентисано програмирање?

Ω Дефиниција 6: Објектно-оријентисано програмирање

Објектно-оријентисано програмирање је метод програмирања који подразумева креирање интелектуалних објеката који представљају модел пословног проблема који ми покушавамо да решимо.

Објекат представља инстанцу (*примерак*) класе. Са сваким објектом ми моделујемо асоциране податке и понашање.

Карактеристике објекта називамо **особинама**. Понашање објекта називамо **методама**[9]**.** Особине и методе заједно чине чланове класе (*members*).

Класе и објекте можемо најбоље да разумемо преко аналогије калупа за колаче и самих колача. Калуп представља класу, а сваки појединачни колач који произведемо у калупу представља једну инстанцу или примерак (објекат).

[9] Препознајемо их најлакше по називу, који увек треба да има глаголски облик, тј. да означава одређену радњу.

1.6.2. Мој први објектно-оријентисани програм

Дађемо пример на замишљеној компјутерској игри, атрактивној за програмирање, названој **МајмунскаПосла**. Демонстрација је написана у псеудо-коду[10]:

Код 1. Програм MonkeyBusiness

```
Програм MonkeyBusiness
     … код изостављен
     Мајмун МајмунЏо (Име = „Џо“);
     Штампај (МајмунЏо.Име);
     МајмунЏо.Једе(три банане);
     МајмунЏо.Скаче(висина=1м, даљ=2м);
     МајмунЏо.Спава;
     … код изостављен
Крај Програма.
```

Из овог илустрованог примера видимо да имамо две класе објеката: Мајмун и Банана. Банане ћемо тренутно ставити ван разматрања. Што се тиче класе Мајмуна којој припада и наш Мајмун Џо, она има 4 члана и то, једну особину: **Име**, и три методе: **Једе, Скаче, Спава**.

Прва наредба креира нову инстанцу (**Мајмун Џо**) класе **Мајмун** и при рађању га крсти именом Џо. На исти начин, би смо морали у једном тренутку да убацимо и Мајмуницу Џејн, али тај део је изостављен, или остављен ученицима да сами то допуне. У том тренутку би било упутно и увести 2 нове методе: **Воли** и **Мрзи**, у зависности од тога како би ишло у одређеним тренуцима њиховог међусобног дружења.

[10] Значи ни у једном специфичном програмском језику

1.6.3. Принципи објектно-оријентисаног програмирања

У реалном животу мајмуни, као и људи имају на стотине особина и комплексно понашање које може да укључи и на стотине метода. Колико особина и метода ћемо узети у разматрање, зависиће од дизајна програма, а то издвајање само одређених поља објекта називамо **апстракцијом**.

Апстракција нам омогућава да обухватимо комплексне идеје док у исто време игноришемо ирелевантне детаље који би нас само збуњивали.

У фундаменталне принципе (стубове) ООП-а спадају **енкапсулација, наслеђивање и полиморфизам**.

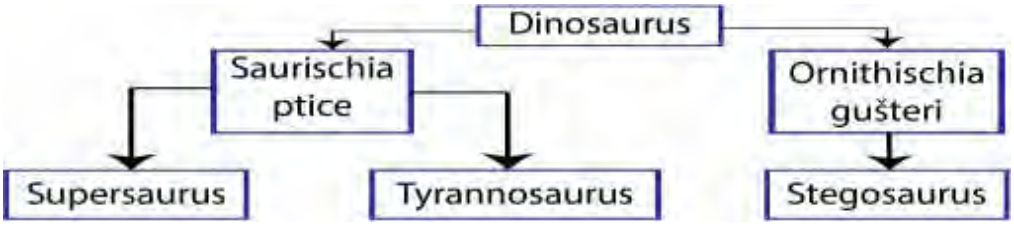
Енкапсулација значи фокусирање на то како нешто ради без разматрања комплексности која је потребна да би се то нешто урадило. У правом животу само седнемо у аутомобил и возимо, без да се замарамо начином рада свих 12 цилиндара мотора.

Класе подржавају **наслеђивање**. Могу да наслеђују и да буду наслеђиване, па тако имамо децу, родитеље, баке и деке по оба родитеља посебно (*вишеструко наслеђивање*) итд.

Наслеђивање је засновано на концепту **генерализације/специјализације.**

Генерализација нам дозвољава да разматрамо генералне категорије објеката који имају заједничке особине и тако формирамо основне (генералне) класе. Онда користимо **специјалистичке** изведене класе које наслеђују особине основних, а у исто време имају додатно и своје посебне, специфичне особине и методе.

Пример диносауруса:



Дијаграм 3: Класе и објекти

Напомена: Овде је Дарвинова теорија од изразите користи, јер после Мајмуна иду Људи, где обе специјалистичке класе имају заједничку генералну особину, а то је да се с времена на време прилично глупирају, а што погрешно зовемо само „мајмунским послом" уместо „људско-мајмунским послом".

Полиморфизам је настао од грчке речи: „више облика", и представља способност интеракције са објектима као генералним категоријама, без обзира на њихову специјалност.

1.7. Алгоритми

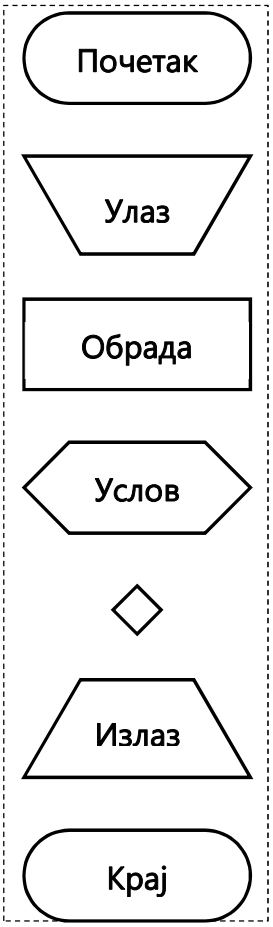
Ω Дефиниција 7: Алгоритам
Алгоритам представља скуп правила које прецизно дефинишу секвенцу операција или активности.

Ω Дефиниција 7б: Алгоритам
Алгоритам за неку класу задатака представља коначан скуп правила помоћу којих решење сваког проблема из те класе може бити нађено аутоматским путем.

Решење проблема, рецептура. Можемо да кажемо и за програм да је алгоритам који компјутер може да обради. Појединачно упутство назива се корак алгоритма, и мора да буде еквивалент једној познатој операцији.

За решавање једне функције или проблема можемо имати више алгоритама. У том случају, неки од критеријума за избор алгоритма могу бити: време (извршавања), једноставност и елеганција, величина алгоритма и сл.

Обично се за представљање алгоритма користе блок дијаграми, алгоритамске шеме, дијаграми тока и сл., а где се сваки корак представља одређеним графичким симболом.



Дијаграм 4. Графички елементи за коришћење блок дијаграма

1.8. Мој први написани програм

Вратили смо се на наш проблем трошкова дупло ојачаних прозора [види **1.3.**].

Дефинисање проблема

Рекли смо три ствари:

1. Које информације улазе у систем
2. Које излазе
3. Шта систем ради са информацијама

За почетак, уместо блок дијаграма користићемо писани текст за сваку фазу:

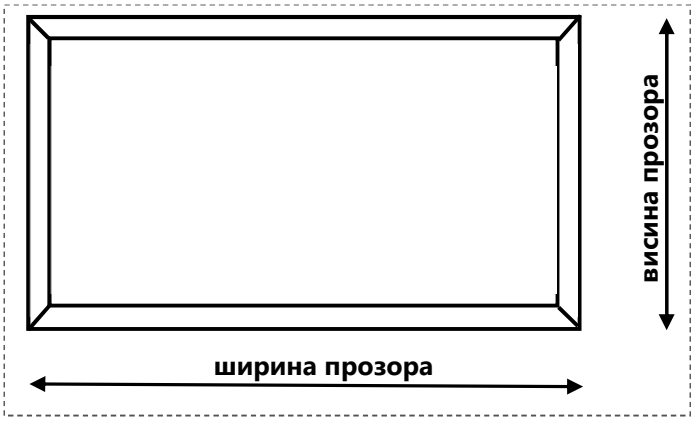
Улазне информације

- Ширина прозора
- Висина прозора

Излазне информације:

- Површина стакла потребна за дупло ојачани прозор
- Дужина дрвета потребног за израду оквира.

Ово што нам је потребно можете видети на дијаграму.



Дијаграм: Графички приказ прозора

Површина стакла је једнака ширини помноженој са висином, па дуплирано, а да направимо дрвени оквир биће нам потребна два комада дрвета дужине једнаке ширини прозора и два комада дрвета дужине једнаке висини прозора.

Ω Дефиниција 8: Метаподатак
Метаподатак (*metadata*) представља информацију о самој информацији.

У случају нашег прозора, метаподатак ће нам рећи које вредности су коришћене и произведене, прецизније јединица мере у којима се информације изражавају, као и валидан опсег дозвољених вредности које информација може да има.

На примеру прозора, иђи ћемо да су и ширина и висина прозора изражени у метрима, са прецизношћу до 1-не стотинке метра, или на две децимале. Најважније је да се ова информација исправно назначи корисницима програма преко корисничког интерфејса. Што се тиче валидног опсега, то би нам рекао клијент у стилу: *„Правим прозоре чија је ширина у опсегу од 0.5 метара до 3.5 метара, док је висина у опсегу од 0.5 метара до 2 метара“*.

Зарад упрошћавања проблема, валидацију, или проверу исправности унетих вредности, важни аспект сваког програма засада ћемо да искључимо из кода.

Шта програм у ствари ради

Програм може да испоручи две вредности у складу са следећим једначинама:

површина стакла = (ширина прозора * висина прозора) * 2
дужина дрвета = (ширина прозора + висина прозора) * 2

Водили смо рачуна да је стакло дуплирано за сваки појединачни прозор.

Сада када имамо спецификацију програма коју је прихватио са нама и наш клијент, и потписао исту, рад може да почне.

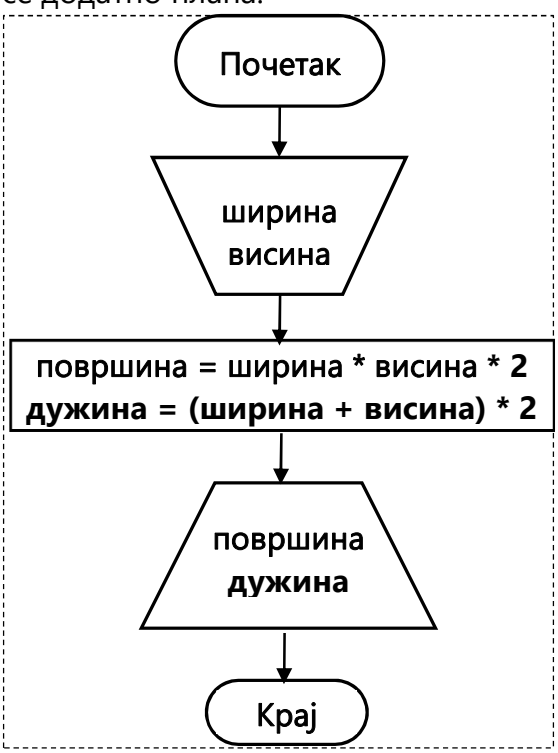
Доказ да програм ради

У стварном свету ми бисмо сада извели тест који доказује да програм ради рекавши нешто слично следећем: *„Ако дам овом програму ширину од 2 метара и висину од 1 метар, програм ће ми рећи да је потребна површина стакла од 4 квадратна метра, док је дрвени оквир дужине 6 метара."*

Наплата програма

Када смо коначно уверили нашег клијента да програм стварно ради онако како је дизајнирано, он неће имати другу опцију него да нас исплати за наш труд и непроспаване ноћи. Често може да се деси да клијент хоће додатне функционалности од вашег програма, и ту је ствар проста: додатно се додатно плаћа.

Блок дијаграм нашег програма изгледа овако:



Дијаграм 5. Блок дијаграм програма Prozor

Сличан, писани облик алгоритма (*са валидацијом*):

- ✓ Учитај ширину
- ✓ Провери вредност ширине
- ✓ Учитај висину
- ✓ Провери вредност висине
- ✓ Израчунај ширина пута висина пута 2 и штампај
- ✓ Израчунај (ширина + висина) пута 2 и штампај

Програмерски део је сада да само претворимо горњи опис у језик који може бити коришћен на компјутеру.

2. Основни елементи програмског језика

У зависности од изабраног **програмског језика**, изабраћемо и окружење за развој наших програма. Добра вест овде је да постоје сјајна бесплатна развојна окружења. **Интегрисано развојно окружење** је оптимизовано за брз развој програма. Кључна реч овде је брзина, јер је у реалном животу развој програма увек са тесним роковима саме имплементације. Главни елементи оваквог окружења су ***компајлер, едитор кода и дибагер***. Постоји још пуно опционих алата које користимо у зависности од потреба самог програма који развијамо.

Напомена:
Примери су урађени у Lazarus IDE верзији 1.8, са Паскал имплементацијом Free Pascal верзија 3.0.0.

2.1. Структура програма

Традиционално сваки програмски језик се представља са HelloWorld демонстрацијом.

Ова чињеница нам говори да је програм HelloWorld убедљиво најпопуларнија апликација икад написана за компјутер, упркос исто тако чињеници, да сам програм не ради ништа више до штампања поздрава „Hello World!“ на екрану.

Напомена:
Програме у зависности од корисничког интерфејса можемо да поделимо на програме са текстуалним корисничким интерфејсом (конзолне) и програме са графичким корисничким интерфејсом.

Када у Лазарусу изаберемо **File > New... > Project – Simple Program**, добићемо најједноставнији конзолни програм који се састоји само од заглавља програма и (празне) блок наредбе. Другим речима генерисали смо прави програм са минималистичком структуром. Сложеније структуре ћемо видети касније.

Код 2. Програм HelloWorld

```
program HelloWorld;                          // Заглавље програма
begin                                        // Блок наредби
   Write('Hello World!');
end.
```

Заглавље програма се састоји од кључне речи program праћене исправним идентификатором HelloWorld. Као и свака друга наредба завршава се са тачка зарезом. Наш посао се састојао од дописивања наредбе Write('Hello World!') која штампа задати текст. **Write** је прва предефинисана метода коју смо користили. Могли смо да користимо WriteLn која би урадила исти посао плус пребацила курсор на нову линију екрана. Да је нашој забави крај сигнализира последњи коришћени симбол: тачка, која означава крај извршавања самог програма.

2.1.1. Унити (units)

Претходна структура програма је задржана због компатибилности. Основни модул са програмским кодом у Паскалу назива се унит. Један програм може имати више унита. Сваки унит има свој фајл и преводи се посебно. Преведени унити се повезују да би креирали јединствени програм/апликацију.

Унит који има улазну тачку (почиње ток извршавања програма) назива се апликација. Само један унит може бити стартни. Кад се апликација покрене, креира се нови домен апликације. Више различитих инстанци апликације може бити покренуто на истој машини. и свака ће имати свој домен. Најпростија, минимална структура унита изгледа овако:

```
unit a;
interface
implementation
end.
```

2.1.2. Библиотеке готовог корисног кода

Кључна реч uses наводи листу свих **библиотека** које ће бити инкорпориране у сам програм. Ови спољни унити могу такође да имау своју uses клаузулу. Библиотеке можемо да схватимо као ризницу корисног и већ написаног кода, који би иначе морали сами да пишемо.

Тако на пример, библиотека **SysUtils** пружа низ корисних функција као што су функције конвертовања броја у текст и обрнуто: **FloatToStr** (*претвара децимални број у текст*), **StrToFloat** (*претвара текст у децимални број*), **IntToStr** (*претвара цео број у текст*), **StrToInt** (*претвара текст у цео број*) и још много других.

При коришћењу шаблона програма са графичким корисничким интерфејсом, аутоматски се убацују следеће библиотеке:

```
uses Classes, SysUtils, FileUtil, Forms, Controls,
Graphics,Dialogs;
```

2.2. Основни симболи језика

Основни симболи су недељиве јединице језика из којих се образују остале сложеније јединице. Скуп симбола Паскал језика састављен је од:

1. Великих и малих слова енглеског алфабета: A..Z a..z
2. Цифара декадног бројног система: 0..9
3. Специјалних симбола: ' + - * / = < > [ ] : ( ) ^ { } @ $ # & %

   или парови специјалних симбола: << >> ** <> >< <= >= := += -= *= /= (* *) (. .) //
4. **Кључних речи**[11].

Кључне речи су иако сложенији и даље **елементарни** симболи програмског језика.

Колико ће их тачно бити увек зависи од специфичне имплементације самог програмског језика.

Тако TurboPascal има 53, Free Pascal 58, Object Pascal (Delphi) 72.

Кључне речи ћемо истицати са плавом бојом и подебљаним фонтом.

Наш HelloWorld програм је користио три: program, begin и end.

Минимална унит верзија користи четири: unit, interface, implementation, end.

2.3. Стандарди кодирања

2.3.1. Коментари

Коментари су игнорисани од стране преводиоца и немају никаквог утицаја на ток извршавања програма. Првенствена улога им је да објасне другим људима шта је писац кода хтео да каже. Посебно су важни за тимски рад. Пишу се када комплексност кода захтева додатно појашњење решења које је употребљено. Коментаре ћемо истицати зеленом бојом.

Коментаре можемо писати:

```
{Ovo je komentar, TurboPascal stil}
// Sve je komentar do kraja linije, vidi HelloWorld, linije 1 i 2
```

[11] Службене, резервисане речи

2.3.2. Идентификатори

У програму сваки објекат или елемент, почевши од самог програма мора да има име или **идентификатор**. Најважнији елементи које идентификујемо су програми, методе, променљиве, константе, типови итд. Правила за формирање идентификатора су да мора почети словом (или _ симболом), а после тога може било која секвенца слова, симбола _ или цифара. Размак није дозвољен, и не може бити кључна реч.

Дозвољено: HelloWorld, _Hello_World, WorldNo123

Недозвољено: ~~123Kreni, Hello World, Begin~~

Напомена:
У Паскалу употреба малих и великих слова не прави разлику, па тако HelloWorld, HELLOWORLD, helloworld и hELLOwORLD представља идентичне идентификаторе.

Сама имена морају бити концизна и самообјашњавајућа, па тако имамо на пример: sirina * visina, а не a * b. Када је име састављено од више речи у зависности од објекта који именујемо (и његовог места у програму) користићемо комбинацију **PascalCasing** за имена програма, метода и објеката (MojProgram, MojaMetoda, итд.), и **camelCasing** за имена променљивих (prosecnaOcena, punoIme).

2.3.3. Табулација кода

Значај табулације најбоље ћемо демонстрирати са једно-линијском функционално идентичном верзијом HelloWorld програма:

```
program HelloWorld; begin Write('Hello World!') end.
```

2.4. Наредбе

Програмски код који чини једну апликацију се састоји од **наредби** формираних од **кључних речи**, **израза** и **оператора**.

Значи уз помоћ основних синтаксних елемената формирамо сложеније форме: **наредбе**. Наредбе су еквивалент реченицама у природним језицима уз једну разлику да се уместо са тачком завршавају са тачка зарезом (;).

Ω Дефиниција 9: Наредба
Наредба представља синтаксно комплетну јединицу програма која изражава једну врсту акције или декларације.

2.4.1. Типови наредби

Приказаћемо неке од карактеристичних типова наредби, које ћемо касније објаснити детаљније.

Табела 1. Класификација наредби

Категорија	Паскал кључне речи / напомене

Наредбе декларација	Наредбе декларације креирају разне елементе програма као што су променљиве, константе или методе. У складу са тим се врши алокација радне меморије потребне за извршавање програма. Декларација променљивих може опционо да додели вредност променљивој, док је код декларације константи додела обавезна.
	const PI = 3.14; // Naredba deklaracije konstanti.
	var povrsina, poluPrecnik: double; // Naredbe deklaracije promenljivih.
	// Opciono sa inicijalnom dodelom vrednosti: poluPrecnik: double=2;
	procedure MojaProcedura; // Naredba deklaracije procedure.
	function MojaFunkcija :string; // Naredba deklaracije funkcije.

Наредбе израза	Наредба израза која рачуна вредност мора ту вредност да ускладишти у променљивој.
	Write('Poluprecnik: '); ReadLn(poluPrecnik); // Pozivi metoda
	povrsinaKruga:= PI * (poluPrecnik * poluPrecnik); // Dodela
	WriteLn(povrsinaKruga); // Naredba izraza (poziv metode).

Условне наредбе	Условне наредбе омогућавају гранање на различите секције кода у зависности од једног или више услова. Кључне речи: if, then, else, case, of, end.
Наредбе циклуса	Наредбе циклуса омогућавају итерацију чланова низа или понављање извршавања сета наредби све док се одређени критеријум не испуни. Кључне речи: for, to, downto, do, while, repeat, until.

Наредбе извршавају акције програма као што су декларација променљивих, додељивање вредности или позив метода.

Редослед извршавања наредби одређује ток контроле или извршавања програма**.**
Ток извршавања може сваки пут при покретању програма да варира у зависности од реакције програма на улаз примљен за време извршавања.

Наредба може да се састоји од једне линије кода која завршава са тачка зарезом (;), или од серије наредби у блоку.

Блок наредба спада у групу **сложених наредби** и може да се састоји од [ниједне,] **једне** или **више** наредби које се извршавају у **секвенци**. Имплементација блок наредбе је изведена преко кључних речи begin и end.

Напомена:
Последња наредба у блоку може а и не мора да се заврши са ; (тачка-зарезом). Код већине програмских мора, тако да је због стварања навике боље је користити иако не морамо.

2.4.2. Наредба доделе

Наредба (оператор) доделе спада у групу простих наредби и изводи се преко пара симбола :=[двотачка једнако]). Ради тако што израчуна израз на десној страни и тако израчунату вредност додели променљивој на левој страни.

```
  povrsinaKruga:= PI * (poluPrecnik * poluPrecnik);
```

Напомена:
Једноставан концепт све док се не поистовети са математичком једнакошћу. Онда упадамо у проблем како је нешто налик следећем: a := a * 1.1 не само могућа, већ и потпуно исправна наредба у програмирању, док математички израз: a = a * 1.1 одмах пада на тесту логике.

Као додатак овом традиционалном Паскал оператору Free Pascal подржава и конструкције у C-стилу као што су: +=, -=, *= и /=.

Ово је приказано у табели бр. 2 и коду бр.3.

Табела 2. Паскал оператори доделе у C-стилу

Додела	**Резултат**
a += b	b дода a, и онда ускладишти резултат у a. Исто као: a:= a + b;
a -= b	Одузме b од a, и онда ускладишти резултат у a. Исто као: a:= a - b;
a *= b	Помножи a са b, и онда ускладишти резултат у a. Исто као: a:= a * b;
a /= b	Подели a са b, и онда ускладишти резултат у a. Исто као: a:= a / b;

Код 3. Програм Kalkulator

```
program Kalkulator;
var a, b: double;
begin
  Write('a? '); ReadLn(a);
  Write('b? '); ReadLn(b);
  a+=b;                                  //Saberi
  WriteLn('Zbir: ', a:8:2);
  a-= b; a-= b;                          //Reset pa oduzmi
  WriteLn('Razlika: ', a:8:2);
  a+= b; a*= b;                          //Reset pa pomnozi
  WriteLn('Proizvod: ', a:8:2);
  a/= b; a/= b;                          //Reset pa podeli
  WriteLn('Kolicnik: ', a:8:2);
end.
```

2.5. Интерфејс програма

Раније (види 1.8), успели смо да алгоритамски решимо проблем везан за израчунавање трошкова израде дупло ојачаних прозора. Сада смо спремни да напишемо и сам програм.

2.5.1. Програми са текстуалним корисничким интерфејсом (Конзолни)

Код 4. Програм Prozor

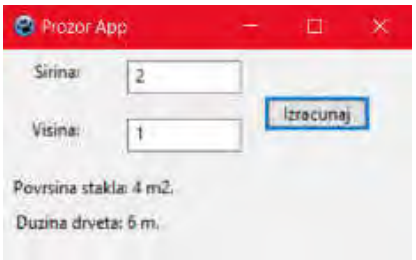
```
program Prozor;
var sirina, visina, duzinaDrveta, povrsinaStakla: double;
begin
   Write('Sirina: ');  ReadLn(sirina);
   Write('Visina: '); ReadLn(visina);
   duzinaDrveta := 2 * (sirina + visina);
   povrsinaStakla := 2 * (sirina * visina);
   WriteLn('Povrsina stakla: ', povrsinaStakla, ' m2.');
   WriteLn('Duzina drveta je: ', duzinaDrveta, ' m.');
end.
```

2.5.2. Програми са графичким корисничким интерфејсом (ГКИ)

На главној форми поставићемо 7 контрола: 4 лабеле, 2 едит контроле и 1 дугме. Кориснички интерфејс програма прозор приликом извршавања за ширину прозора 2 метара и висину 1 метар је приказан на слици десно.

Особине контрола су подешене према табели:

Контрола	Задате особине контрола
Форма	**Name**='fProzor', **Caption**='Prozor'
Лабела1	**Caption**='Sirina:'
Лабела2	**Caption**='Visina:'
Лабела3	**Name**='lbPovrsina', **Caption**='Povrsina stakla: '
Лабела4	**Name**='lbDuzina', **Caption**='Duzina drveta: '
Едит1	**Name**='edSirina', **Text**=''
Едит2	**Name**='edVisina', **Text**=''
Дугме1	**Name**='btIzracunaj', **Caption**='Izracunaj'



Код 4a. Програм Prozor (ГКИ)

```
procedure TfProzor.btIzracunajClick(Sender: TObject);
var sirina, visina, duzinaDrveta, povrsinaStakla: double;
begin
  sirina:= StrToFloat(edSirina.Text);
  visina:= StrToFloat(edVisina.Text);
  duzinaDrveta:= 2 * (sirina + visina);
  povrsinaStakla:= 2 * (sirina * visina);
  lbPovrsina.Caption:=lbPovrsina.Caption +
                        FloatToStr(povrsinaStakla) + ' m2';
  lbDuzina.Caption:=lbDuzina.Caption+FloatToStr(duzinaDrveta)+' m.';
end;
```

2.6. Променљиве

За све манипулације подацима, програм мора све податке које користи приликом извршавања да ускладишти у радној меморији. Те елементе програма називамо променљивима. Променљиве нису једини начин чувања података програма, али су најважнији са становишта функционалности програма. У аналогији са рецептом неког кулинарског јела, променљиве представљају основне састојке коришћене за припрему јела. Да би их користили морамо прво да их креирамо или декларишемо.

Ω Дефиниција 10: Променљива
Променљива је именована меморијска локација одређеног типа, чија се вредност може мењати током извршавања програма.

Декларацијом променљиве одваја се потребан простор у меморији, чија величина зависи од типа променљиве. На примеру програма прозор, видимо да су нам потребне 4 променљиве (по две за улаз и излаз), и да смо се одлучили за тип double, 64-битни „дупло-прецизни" тип намењен раду са децималним бројевима. Прва ствар коју програм уради по покретању је да резервише меморију за своје потребе у оквиру свог домена (види 2.2). Недостатак ресурса или слободне меморије на компјутеру где се програм покреће би довео до непријатне поруке (случај управљаног кода) оперативног система да нема довољно меморије за наш програм, док би у случају не управљаног и у исто време непажљивог кода највероватнији исход био крах целог система. Прво додељивање вредности променљивој у програму назива се **иницијализација** променљиве. Почетна вредност променљивих приликом алокације меморије зависи од конкретне имплементације одређеног компајлера. Стандард је да се нумерички типови поставе на нулу. Најбоља пракса по овом питању, али и неким другим је да се никад не ослањамо на сам компајлер ако нисмо на то принуђени.

Код 5. Програм KontrolaBrzine

Штампа **брзину** [v] (met/sek] за пређени **пут** [s] (километри) за **време** [t] (исказано у сатима, минутима и секундама), ако знамо да је $v = \frac{s}{t}$.

```
program KontrolaBrzine;
const KM_M=1000; VREME60=60;
var brzina, put, sat, min, sek: double;
begin
  Write('Put(km): '); ReadLn(put);
  Write('Vreme(sat): '); ReadLn(sat);
  Write('Vreme(min): '); ReadLn(min);
  Write('Vreme(sek): '); ReadLn(sek);
  brzina:= (put * KM_M) / (((sat*VREME60) + min)*VREME60+sek);
  WriteLn('Brzina(m/s): ', brzina);
end.
```

2.7. Типови података

Тип података представља скуп вредности које се могу доделити и операција које се могу извршити над тим вредностима. Свака променљива и константа има тип, као и сваки израз који евалуира у одређену вредност.

Сваки тип носи следеће информације:

- величина складишног простора коју захтева променљива тог типа
- опсег од минималне до максималне вредности
- локација меморије која ће бити алоцирана у времену извршавања
- врсте дозвољених операција

Предефинисане просте типове можемо поделити на (а) **нумеричке** и (б) **не-нумеричке**. **Нумерички** („аритметички“) даље могу бити **целобројни** и **децимални** („реални“). **Не-нумерички** обухватају **логички** и **знаковни** тип.

Од сложенијих типова поменућемо сада и текстуални тип string (кључна реч).

Напомена:
Текстуални типови су такви са становишта корисника програма, са становишта компјутера и они су нумерички подаци.

Табела 3. Предефинисани типови података:

Тип	Значење	Опсег	Иниц. вред.	Прец.
single	32-битни децимални, једнострука прецизност	1.5e-45..3.4e38	0.0f	7-8
real	32 или 64-битни	single или double, зависно од платформе	???	???
double	64-битни децимални двострука прецизност	5.0e-324..1.7e308	0.0d	15-16
currency	64-битни децимални валутни	-922337203685477.5808.. 922337203685477.5807	0.0m	19-20
extended	80-битни децимални	3.6e-4951..1.1e4932	0.0m	19-20
byte	8-битни целобројни	0..255	0	
shortint		-128..127		
smallint	16-битни целобројни	-32768..32767		
word		0..65535		
integer	16 или 32-битни	smallint или longint зависно од платформе		
longint	32-битни целобројни	-2147483648..2147483647		
longword		0..4294967295		
int64	64-битни целобројни	-9223372036854775808 .. 9223372036854775807		
qword		0 .. 18446744073709551615		
boolean	8-битни логички	True, False	False	
char	8 или 16-битни	ASCII или UNICODE карактер	\x0000	
string	Секвенца од нула или више карактера	ASCII или UNICODE текст		

2.7.1. Целобројни тип

Целобројни тип обухвата коначан подскуп целих бројева. Погодни су за коришћење у петљама као бројачи или за индексирања. Треба их користити кад год је то могуће зато што резултују најбољим перформансама за дати процесор и оперативни систем.
У нашим примерима користићемо углавном **integer** тип.
Важе оператори: унарни промена знака: + -, као и бинарни: + - * / div mod. Пример:

```
var brojUcenika: integer;
```

2.7.2. Реални (децимални) тип

Реални типови представљају децималне бројеве. Код реалних типова осим величине која диктира опсег дозвољених вредности, битан нам је и број значајних цифара, тј. прецизност одређеног типа. У нашим примерима користићемо **double** тип, осим за валутне податке, где ћемо користити **currency** тип који је оптимизован за такву употребу. На реалном типу дефинисани су стандардни оператори: унарни: + - као и бинарни: + - * /.

Пример:

```
var brzina: double; cena: currency;
```

2.7.3. Логички тип (boolean)

Ово је такође стандардни, предефинисани тип и може да има само две вредности: true и false. Ово су у исто време и кључне речи. Пример:

```
var istina: boolean;
```

2.7.4. Знаковни тип (char)

Char је стандардни, унапред дефинисани редни тип података и представља линеарно уређени коначан скуп знакова (*слова, бројеви, посебни симболи*). Сваком знаку придружен је редни број (његов интерни код). Дефинисан је сет од 256 карактера – ANSI сет где првих 128 знакова припада тзв. ASCII (American Standard Code for Information Interchange) сету. Заузима 1 бајт. Пример:

```
var pocetnoSlovo: char;
```

2.7.5. Текстуални тип (string)

Променљива стринг типа представља секвенцу карактера динамичке дужине, максималне величине између 1 и 255. Пример:

```
var tekst: string[50];                    //Maksimum 50 karaktera
  maxTekst: string;                       //Do 255 karaktera
```

Када се у декларацији не наведе дужина између средњих заграда, онда се одваја локација за максималну дужину стринга od 255 karaktera.

Стринг тип се може приказати и разумети и као низ елемената знаковног типа Char: array[1..255] Of **char**;

Текст се при додели уоквирује са једноструким наводницима tekst:= 'Neki tekst!'. При штампању текст наравно нема наводнике (биће Neki tekst!). Уколико треба да се виде наводници онда се користи tekst:= '''NekiTekst!''' што даје при штампању 'NekiTekst!'.

Стрингови дозвољавају само операције спајања преко оператора + и поређења, па важе релацијски оператори: =, <>, >, <, >=, <=. Када се врши поређење, операција се врши карактер по карактер у зависности од позиције карактера у ASCII табели. Тако је стринг 'MISAO' не само различит од стринга 'misao', већ је и мањи ('M' < 'м').

Коришћење већине наведених предефинисаних типова података демонстрираћемо једноставним, али корисним програмом, чија се варијација користи у свим пословним системима.

Код 6. Програм Ekonomija (ГКИ)

Рачуна **добит, профитабилност** и **рентабилност**: $D = (OP * PC) - (FT + OP * VT)$,

$Pb = \frac{D}{K}$, $R = \frac{FT}{PC - VT}$ где је: OP – Обим производње (kom), PC – Продајна цена (nj/kom), FT – Фиксни трошкови (nj), VT – Варијабилни трошкови (nj/kom), K – Капитал (nj).

```
procedure TfEkonomija.btStampajClick(Sender: TObject);
var  obimProizvodnje: integer;
dobit,prodajnaCena,fiksniTrosak,varijabilniTrosak,kapital:
currency; rentabilnost, profitabilnost: double;
begin
   obimProizvodnje:= StrToInt(edObimProizvodnje.Text);
   prodajnaCena:= StrToFloat(edProdajnaCena.Text);
   fiksniTrosak:= StrToFloat(edFiksniTrosak.Text);
   varijabilniTrosak:= StrToFloat(edVarijabilniTrosak.Text);
   kapital:= StrToFloat(edKapital.Text);
   dobit:= obimProizvodnje*prodajnaCena - fiksniTrosak +
                  obimProizvodnje * varijabilniTrosak);
   profitabilnost:= dobit / kapital;
   rentabilnost:= fiksniTrosak / (prodajnaCena - varijabilniTrosak);
   lbDobit.Caption:= FloatToStr(dobit);
   lbKapital.Caption:= FloatToStr(profitabilnost);
   lbRentabilnost.Caption:= FloatToStr(rentabilnost);
end;
```

2.8. Изрази

Видели смо да се програми есенцијално састоје од наредби којима се врше операције над подацима. То имплицира да нам је суштински потребан начин да чувамо податке и методе за манипулисање њима. Ове две функционалности су доступне преко **променљивих** [види 1.6] и **израза**.

Израз је конструкција која се појављује унутар наредби и производи неку вредност.

Састоји се од **оператора** и **операнда**.

Оператори дефинишу операције које треба извршити над операндима да би се добио резултат. Обично су бинарни, тј. захтевају два операнда. Имамо такође и унарне операторе који захтевају само један операнд, као што су на пример оператори промене знака.

Неки оператори се понашају другачије у зависности од типа података са којим раде.

Операција пресликава коначан скуп података (операнда) у коначан скуп података (резултата).

Операторе у зависности од типа израза над којим се врше операције делимо:

2.8.1. Аритметички оператори

Аритметички оператори се појављују у аритметичким операцијама над реалним и целобројним типовима.

Табела 4. Аритметички оператори

+	–	*	/	div	mod
сабирање	одузимање	множење	дељење	целобројно дељење	остатак целобројног дељења

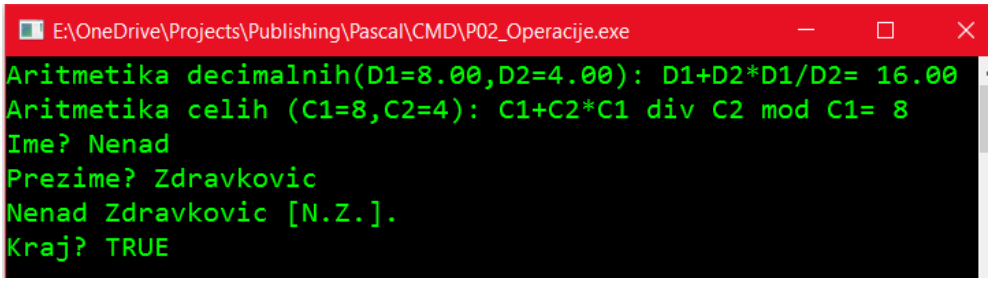
Имамо још и унарне промене знака: + (*плус*) и – (*минус*).

Оператор + има дуално понашање, са бројевима ради сабирање, док са текстом врши спајање текста.

Код 7. Програм Operacije

```
program Operacije;
var  C,C1,C2: integer;                          // celobrojni
  D,D1,D2: double;                              // decimalni
  ime, prezime, punoIme, poruka: string;        // tekst
  slovoIme, slovoPrezime: char;                 // jedan karakter
  kraj: boolean;                                // logika
```

```
begin
  D1:= 8.00; D2:= 4.00; C1:= 8; C2:=4;
  ime:=''; prezime:='';                          // prazan string
  kraj:= false;
  poruka:='Aritmetika decimalnih(D1=8.00,D2=4.00): D1+D2*D1/D2= ';
  D:= D1 + D2 * D1 / D2;
  WriteLn(poruka, D:4:2);
  poruka:= 'Aritmetika celih (C1=8,C2=4): C1+C2*C1 div C2 mod C1= ';
  C:= C1 + C2 * C1 div C2 mod C1;
  WriteLn(poruka, C);
  Write('Ime? '); ReadLn(ime);
  Write('Prezime? '); Readln(prezime);
  punoIme:= ime + ' ' + prezime;
  slovoIme:= Ime[1];  slovoPrezime:= Prezime[1];
  poruka:= punoIme + ' [' + slovoIme +'.'+slovoPrezime + '.].';
  WriteLn(poruka);
  kraj:= not kraj;                 // sad je stvarno zabavi kraj?
  WriteLn('Kraj? ', kraj);
end.
```



Обратите пажњу да симболички оператори + - * / пошто не могу бити део идентификатора могу да се лепе уз њих, док оператори div и mod из дијаметрално супротног разлога не трпе то исто.

2.8.2. Логички boolean оператори

Ово су логички оператори над типовима величине једног бита.
Разликујемо: унарни not – логичка негација и бинарни and – логичко, и or – логичко или. Сва три оператора су у исто време и кључне речи.

Табела 5. Логичке операције

q	p	not p	p and q	p or q
false	false	true	false	false
true	false	true	false	true
false	true	false	false	true
true	true	false	true	true

Код 8. Програм Logika

```
program Logika;
var istina, laz, rezultat: boolean;
begin
  istina:= true;
  laz:= not istina;                    //false
  rezultat:= istina and istina;        //true
  rezultat:= istina and laz;           //false
  rezultat:= laz and laz;              //false
  rezultat:= istina or istina;         //true
  rezultat:= istina or laz;            //true
  rezultat:= laz or laz;               //false
end.
```

2.8.3. Релациони оператори

Разликујемо:

=	<>	>	<	>=	<=
једнако	различито	веће	мање	веће једнако	мање једнако

Табела 6. Релациони оператори

Уз помоћ релационих и логичких оператора формирају се сложени логички, условни изрази.

2.8.4. Редослед важности оператора

Израз се израчунава у складу са рангом оператора, а у случају да су оба оператора истог ранга, израз се рачуна са леве према десној страни.

Табела 7. Редослед важности оператора

Ранг 1	not						
Ранг 2	*	/	div	mod	and		
Ранг 3	+	-	or				
Ранг 4	=	<>	<	>	<=	>=	in

На пример израз: 5 * 4 – 8 / 2 враћа вредност 16 а не 6.

2.9. Константе

Константе су специјалан облик променљивих које би могли да зовемо и непроменљивим.

Ω Дефиниција 11: Константа
Константа је именована меморијска локација одређеног типа, чија се вредност не може мењати током извршавања програма.

Могу се користити било где у програму уместо стварне вредности. Вредност израза који се додељује константи мора да буде познат у време компајлирања.

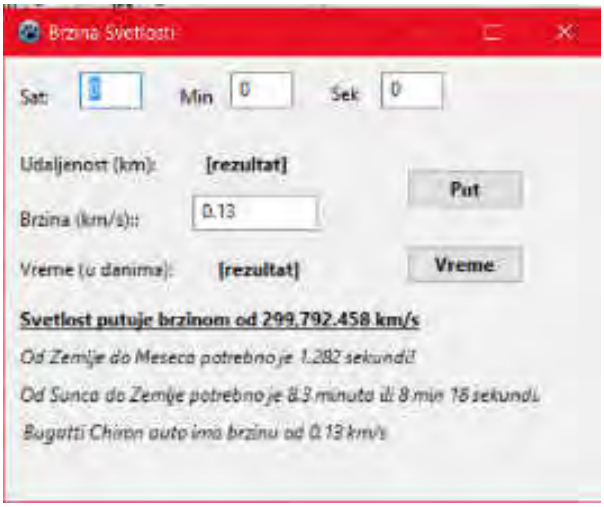
Предности коришћења константи могу се сажети следећим:

1. **Код је бржи.** Пошто се вредност константи не мења и уписује само једном, онда оне заузимају део радне меморије који је заштићен од писања. Самим тим коришћење константи убрзава рачун израза у којима се константе користе.

2. **Код је разумљивији и лакши за читање.** Тако: *put := vreme * MAX_BRZINA* делује боље од *put := vreme * 250*. Бројеви као што су 250 у овом примеру се иначе називају „срећним" бројевима, што значи да постоји добра шанса да осим аутора нико други не зна о чему се тачно ту ради. Самим тим, сваки срећни број је добар кандидат за секцију константи.

3. **Лакша измена кода.** Уколико се нека константна вредност користи на сто места у коду, а дошло је до измене њене вредности, измена кода преко дефинисане константе се обавља на једном месту (месту где смо дефинисали константу). У супротном морали би смо да измену извршимо на свим местима где се појављује та вредност, што је напорно и мучно. За демонстрацију константи и потребе за њима винућемо се мало у дубине универзума ☺:

Код 9. Програм BrzinaSvetlosti (ГКИ)



Процедура **Пут** претвара време изражено у сатима, минутима и секундама у **пут** изражен у километрима коју пређе светлост. Тачна брзина светлости износи 299,792.458 километара у секунди (km/s). Процедура **Vreme** рачуна време у данима потребно да се при одређеној брзини пређе претходно израчунати пут.

```
procedure btPutClick;
const BRZINA_SVETLOSTI=
299792.458;
  X60 = 60;
var  sati, minuti, sekunde: double;
  put: extended;
begin
  sati:= StrToFloat(edSati.Text);
  minuti:= StrToFloat(edMinuti.Text);
  sekunde:= StrToFloat(edSekunde.Text);
  put:=(sati * X60 * X60 + minuti * X60 + sekunde) *
BRZINA_SVETLOSTI;
  lbPut.Caption:= FloatToStr(put);
end;


procedure TfBrzinaSvetlosti.btVremeClick;
const DAN_SEK = 24 * 60 * 60;
var  put, brzina, vreme: extended;

begin
  put:= StrToFloat(lbPut.Caption);
  brzina:= StrToFloat(edBrzina.Text);
  vreme:= put / brzina / DAN_SEK;
  lbVreme.Caption:= FloatToStr(vreme);
end;
```