Počítačové praktikum 1

V. Černý

Preliminary, 25.8.2005

Chudák Martin

Motto:
Záhoráci sa v povznesenej nálade vracali domov z oslavy. Jeden bol naozaj unavený a celú cestu podchvíľou opakoval: "Chuďák Fera!". Jedna z tiet to už nevydržala a spýtala sa: "Ná Jaro, tak nám už rekni, proč je ten Fera chuďák?". "Ále, chcel dat Jožovi po papuli, a já sem tam nastrčil hubu!"

Ideu týchto praktík vyhútal Martin Mojžiš. Vymyslel tak chomút, do ktorého som ja strčil hlavu. Chudák Martin ...

Volá sa to počítačové praktikum, ale malo by to byť najmä o fyzike. Pri riešení problémov a počítačovom experimentovaní sa treba snažiť porozumieť "ako tá fyzika funguje". Zázraky čakať nemožno. Bude to len 10 až 12 úloh alebo tematických celkov, tam celú fyziku nevopcháš. Ale veľmi by som chcel, aby sa to nezvrhlo na to otravné "treba odovzdať 12 protokolov" (v novom newspeaku "reportov"). Alebo tiež "Behali tam po obrazovke zelené čiary, ale inak sme nerozumeli, o čom to bolo". (To nie je žart, to je citát.) Zázraky sa nedejú, aspoň nie každý deň. Takže priveľkú ilúziu si nerobím. Ale poprosím, skúste spolupracovať.

Je to nový predmet, nevyskúšaný, bude sa aj improvizovať. Ale nebude sa veľa programovať, hoci to je *počítačové* praktikum. Nemyslím, že každý fyzik má byť aj programátorom. Teda aktívnym programátorom, ktorý vie napísať príslušné programy od nuly. Ale asi by bolo dobre, keby väčšina fyzikov mala dosť chochmesu, aby vedela zhruba porozumieť kľúčové úseky kódu, v ktorých "sa deje nejaká fyzika". Potom aj spoznať, ktoré úseky kódu to vyrábajú "tie čiary, ktoré práve behajú po obrazovke" a potom, možno, zažiť aspoň malý "aha-pocit", že takto to funguje. A potom aj zmeniť kúsky kódu a pochopiť ako to funguje v iných podmienkach alebo zmenené.

Základná téza je, že aj veľmi slabý programátor môže úspešne preeditovť kus kódu, ak je ten kód pôvodne rozumne napísaný, a poučiť sa pri tom aj o programovaní aj o realite, ktorú onen kód zobrazuje. Snažil som sa ale, aby aj tí, ktorí majú väčšie programátorské ambície, si našli niečo pre seba.

Jedno je dôležité: nepodceňujte sa! Vaša hlava na to má, len pre niekoho môže byť nezvyklé, že sa dá "vysomáriť" i v situácii, ktorá na prvý pohľad nie je "nič pre mňa".

Neviem odhadnúť, či tento predmet prežije viac ako prvý rok. Ale snažil som sa začať to stavať tak, ako keby mal. Ak to nevyjde, nasledujúce riadky treba chápať ako iróniu.

```
Я памятник себе воздвиг нерукотворный,
К нему не зарастёт народная тропа,
Вознёсся выше он главою непокорной
Александрийского столпа.
```

Skôr, ako začnete

Tento text je sprievodným textom k predmetu Počítačové praktikum 1. Je to nový predmet, preto sa aj tento text bude v priebehu semestra možno upravovať. To čo je zhruba nachystané pred začiatkom semestra je "infraštruktúra", teda pomocné materiály a programové zabezpečenie. Námety pre jednotlivé úlohy a "pracovné listy" budú zrejme vznikať v priebehu semestra, podľa toho ako budem stíhať a podľa toho, aké budú prvé skúsenosti. Praktiká povedieme podľa počtu prihlásených pravdepodobne viacerí. Študenti zo skupín, ktoré nepovediem, samozrejme, môžu konzultovať problémy, pripomienky a návrhy aj so mnou.

Takže kontakt na mňa. Neváhajte zaklopať na moje dvere hocikedy alebo pošlite mejl.

Vladimír Černý
Katedra teoretickej fyziky a didaktiky fyziky
pavilón F2, miestnosť 137
e-mail: cerny@fmph.uniba.sk
www osobný: sophia.dtp.fmph.uniba.sk/~cerny/
www pre toto praktikum: www.t3.fmph.uniba.sk/~cerny/

Nezľaknite sa tohto skripta. Ak to budete čítať, narazíte na viaceré slová a celé časti, ktorým nebudete celkom rozumieť. Počítače sú predsa len komplikovaná vec a vývoj ide rýchlym tempom. Ani malé deti nespoznávajú svet systémovo logicky, ale skladajú mozaiku z úlomkov poznania a svet sa im zrazu vynorí ako obraz. Aj tento text obsahuje také útržky. Nie všetky zapadnú hneď do mozaiky. A ten obraz sa celý nevytvorí koncom semestra. Marián Klein ma napríklad upozornil, že na ktoromsi mieste zrazu z ničoho nič hovorím o run leveloch v linuxe. Nikde predtým a vlastne ani nikde potom ten pojem netreba. Marián povedal, že on vie, o čo ide a teda rozumie, čo je tam napísané, ale ten, kto sa s tým ešte nestretol, nič nepochopí. No, je to taký kúsok mozaiky, ktorý hádam niekam zapadne až neskôr a asi len tým, čo sa budú s linuxom viac hrať, inštalovať si ho doma a správcovať. Medzi nami, ja o run leveloch dosť veľa tuším, ale že by som to presne poznal, to teda nie. Ostane vám teda možno iba pocit že run level dajako súvisi so štartovaním linuxu. Ale to je presne ono. Takže som sa rozhodol ten pojem nechať tam, kde je.

Napriek tomu, aby text bol férovejší voči tým ktorí pochopiť chcú a nie vlastnou vinou zatiaľ nemôžu, budú niektoré pasáže v texte označené marginálnou ikonou ako vidíte tu na okraji, ktorá má označovať čosi ako geeka. Geek je na webe populárny termín s ťažko definovaným významom, označujúci kockatú hlavu, mierne vyšinutého excentrika, experta, guru, fanatika, zmes toho všetkého.

Ak teda uvidíte na okraji geeka, znamená to, že je to odstavec, ktorý je pre vyšinuté hlavy a kto sa zaujíma len o "mierny pokrok poznania v medziach zákona", môže takú pasáž kľudne preskočiť. Rádoby-geekovia si také pasáže môžu prečítať a nemusí ich trápiť, ako im všetko hneď nezapne. Dlhšie geekovské pasáže textu budú onačované ikonami geeka zo šípkou začiatku alebo konca (ako dopravné značky zákaz státia začiatok a zákaz státia koniec).

Takže ešte raz: neváhajte zaklopať na dvere a spýtať sa. Nie na všetko budem vedieť odpovedať, ale pokúsiť sa môžme.

Časť I

Infraštruktúra, podporné prostriedky

Kapitola 1

Predpoklady a prostriedky

Praktikum bude v Jave (1.4.2) pod linuxom. Predpokladá sa, že študent má elementárne znalosti z programovania v Jave. Je dobré, keď si niekedy niečo vyskúšal so swingom (javax.swing je package na tvorbu grafického užívateľského prostredia pre aplikácie), ale nie je to nutné. Prostriedky swingu budeme využívať len pasívne, nebude potrebné niečo aktívne konštruovať ani modifikovať. Praktikum voľne nadväzuje na skúsenosti z prednášok a cvičení v Jave a bude používať podobné pracovné prostredie (Linux KDE + SciTE). V princípe ale stačia základné skúsenosti s Javou získané v iných kurzoch alebo samostatne.

Ďalej predpokladáme základnú znalosť práce s word procesorom ako Microsoft Word alebo OpenOffice Writer a s tabuľkovým kalkulátorm (spreadsheet) typu Microsoft Excel alebo OpenOffice Calc.

Pod linuxom budeme v učebni pracovať v grafickom prostredí KDE v distribúcii Debian. Okrem prostriedkov Java jdk budeme ešte potrebovať programátorský editor v princípe podľa vlastnej chuti, ale začiatočníkom odporúčam používať SciTE aby sme boli navzájom kompatibilní aj na úrovni pokynov typu "Teraz stlačte F9!". Ďalej budeme potrebovať OpenOffice a pre prácu s obrázkami GIMP.

Kto sa chce hrať aj doma, môže aj pod Windowsami, ak si nainštaluje potrebný softvér.

Učebnicovým podkladom je tento materiál, ktorý práve čítate. Okrem toho som nazhromaždil kopec ďalších informačných materiálov a (podľa mňa) užitočného softvéru na tzv. "sprievodnom CD". Fyzicky je to buď naozaj CD alebo súbory prístupné na sieti, linka na ne bude na praktikovom webe www.t3.fmph.uniba.sk/~cerny/.

Úlohy na jednotlivé cvičenia sa budú postupne objavovať na webe. Odporúčam zoznámiť sa s príslušným materiálom dopredu pred príchodom na cvičenie. Detaily odovzdávania protokolov (reportov) a princípy hodnotenia si dohodneme na prvom cvičení.

1.1 Windows

V tomto odseku pár slov pre tých, čo si chcú inštalovať vhodné prostredie doma pod operačným systémov Windows. Odporúčam Windows XP alebo 2000, nižšie už dnes nemá zmysel chodiť.

Nainštalujte si Javu. Dnes už dlho žije stabilná verzia 1.5.x ale pod Debian Linuxom je stále rozumne dostupná len verzia 1.4.2, preto ak chcete doma kompilovať classy, ktoré budete spúšťať na T3, tak musíte mať verziu 1.4.2. Stiahnite si ju z `java.sun.com` alebo zo sprievodného CD (`Java\windows\j2sdk-1_4_2_05-windows-i586-p.exe`). Spustíte to a nainštaluje sa to bez problémov samo.

Ďalej potrebujete buď Microsoft Office alebo OpenOffice. OpenOffice je zadarmo na webe (`www.openoffice.org/`).

Na prácu s obrázkami sa zíde GIMP (`gimp-win.sourceforge.net/stable.html`), potrebujete nainštalovať najprv grafickú podporu gtk+ a potom GIMP. Potrebné inštalačné programy sú aj na sprievodnom CD.

Na sprievodnom disku nájdete niekoľko testovacích skriptov, aby ste si mohli vyskúšať, že Java funguje.

Budete písať alebo upravovať programy, potrebujete preto rozumný programátorský textový editor (nie word procesor). Môžete používať ľubovoľný podľa vášho vkusu, v prvom ročníku sa na praktikách používal editor SciTE, je voľne dostupný na sieti, nájdete ho aj na sprievodnom CD.

1.2 Linux

Treba popravde povedať, že hlavným dôvodom pre výber linuxu ako pracovného prostredia v učebni T3 bol fakt, že linux je zadarmo. Na druhej strane v zahraničnom prostredí "fyzikálnej vedy" je dominantný linux. Neberte teda prácu v prostredí linux ako núdzový stav, bez linuxu sa dnes profesionálne fyzika robí len veľmi ťažko.

Linux, to dnes prakticky znamená niektorú z veľa distribúcií linuxu, ktoré sa navyše vyskytujú každá vo viacerých verziách. Prakticky to znamená, že nainštalovanie aplikačného softvéru z binárnych distribúcií nie je vôbec triviálne. Treba zohnať balík (package) pre vašu konkrétnu distribúciu linuxu alebo potom kompilovať zo zdrojákov, čo je často ešte menej triviálne.

V tomto obmedzenom priestore preto nemôžme diskutovať jemnosti, našťastie linuxová komunita žije s webom a po istej námahe sa dá nájsť, ako na to. Na učebni budeme pracovať s distribúciou Debian. Keďže tam nebudete nič inštalovať, môže vám to byť jedno.

Tým, ktorí sú doma zvyknutí na Windows, odporúčam vyskúšať si linux a úskalia jeho inštalácie. Najprv sa treba zoznámiť s operačným systémom ako užívateľ. Na sprievodnom disku som zhromaždil niekoľko informačných zdrojov od krátkej tabuľky základných príkazov až po rozsiahlu publikáciu Linux Dokumentačný projekt. Web je plný ďalších tutoriálov.

Na prvý pokus odporúčam nerozhádzať si počítať ostrou inštaláciou na hard disk, ale začať s Knoppixom. Dobre je mať počítač bootovateľný z CD, i keď sa to dá prekonať, ak si vyrobíte bootovacie diskety, ale to už začínate rovno s komplikáciou. Knoppix znamená bootovacie CD (najnovšie bootovacie DVD, verzia 4.0. Odporúčam poslednú verziu CD 3.9, stiahnite si CD image z webu (www.knoppix.org), napáľte si CD, vložte ho do mechaniky a reštartujte počítač. Ak nemáte nejaký naozaj exotický stroj, mali by ste sa prebudiť vo svete linuxu. Kto zažil inštalovanie s množstvom otázok, o ktorých dobre nevedel, čo znamenajú, to veľmi ocení. Knoppix je pozoruhodne dobrý v detekcii hardvéru a spravidla sa vysomári sám. Ak nie, máte smolu.

Jediný problém je, že tam nie je Java. Teda presnejšie. Je tam Java run time support jre 1.4.2 a môžete spúšťať to, čo ste si niekde inde vytvorili a skompilovali do classov. Ale nebeží javac. Teda vlastne beží, ale nie je to Sun jdk. Príkazom javac sa spustí veľmi obmedzený kompilátor GNU gcj, ktorý nepozná awt ani swing, takže môžete vyrábať riadkovo orientované aplikácie. Dôvody sú čisto legálne, Knoppix je založený na Debiane a Debian je puritánske open source prostredie. Java od Sunu sa môže voľne distribuovať ako binárky, ale nie ako zdrojový kód. A je problém. Rieši sa viacerými spôsobmi. Kto má ozajstný Debian, môže použiť techniku "apt-get" (na sprievodnom CD nájdete nejaké poučenie) a nainštalovať si Blackdown distribúciu JDK. Knoppixu sa dá trocha pomôcť. Našiel som to na adrese `http://neubia.com/archives/2004_10.html` a ešte trocha upravil aby nebolo treba robiť persistent home directory inštaláciu knoppixu. Na sprievodnom disku je návod spolu s direktóriom, ktoré treba skopírovať (pod Windowsami) niekam na windows disk.

Upozornenie!

V poslednej vete slová "skopírovať pod Windowsami" sú dôležité. Knoppix (všeobecne linux) vidí totiž windows disky, ale disky alebo partície typu NTFS vie dobre iba čítať!!! Zapisovať na NTFS disk pod linuxom je samovražedná aktivita, môžete si zničiť všetky dáta. Knoppix mountuje všetky windows disky ako read-only, takže ich vidíte, ale písať sa nedá. Ale húževnatý užívateľ to dokáže prekonať. Partície FAT a FAT32 sú ok, môžete do nich pod linuxom písať, ale NTFS nie. Veľký Bill pracuje húževnato na tom, aby bol nepostrádateľný. Ak neviete, čo je partícia a FAT32, nezapisujte pod linuxom na Windows disky nič.

Pre uľahčenie života som navaril remasterovaný Knoppix CD priamo pre potreby tohto praktika. Je tam aj Java aj SCiTe aj knižnice, potrebné pre toto praktikum, takže stačí vložiť do mechaniky, bootovať počítač a ste v správnom svete. Mimochodom, pridal som tam aj základný TEX. Ak chcete také CD, navštívte ma.

Na sprievodnom disku nájdete niekoľko testovacích skriptov, aby ste si mohli vyskúšať, že Java funguje.

1.3 Linux: Prvá pomoc

Pod linuxom budeme pracovať v grafickom užívateľskom prostredí KDE. Stroje v učebni sa inicializujú pod linuxom do run-level 5, teda rovno do grafického módu, v ktorom prebieha aj prihlasovanie (login). Ak prídete niekam na návštevu a dajú vám k dispozícii linuxový počítač, môže vás zaskočiť, že sa inicializuje do run-level 3, teda do riadkového módu. Treba

sa s tým zoznámiť, k linuxu to proste patrí. A už vôbec sa tomu nevyhnete, keď budete chcieť pracovať na diaľku, prihlásiť sa na vzdialený počítač.

Je dobré vedieť, ako sa prechádza z riadkového do grafického módu a naopak. Treba si to zapamätať. Neviem prečo, ale rôzne helpy vám práve toto nepovedia, keď to najviac potrebujete.

Teda grafický mód sa v riadkovom móde odštartuje (najčastejšie) príkazom

```
startx
```

ak toto nefunguje, musíte si nechať poradiť od lokálneho experta, inak strávite život študovaním XWindows skriptov.

Naopak návrat z grafického módu (do ktorého ste sa dostali cez startx) do riadkového sa deje stlačením kláves

```
Ctrl-Alt-Backspace
```

Ak ste v grafickom móde, pretože sa mašina inicializovala do runlevel 5, tak vás to ale zase len privedie do grafického módu a ponúkne login. Alternatívne môžete skúsiť, čo to urobí, stlačiť

```
Ctrl-Alt-F1
alebo
Ctrl-Alt-F2
```

V grafickom login prompte si spravidla môžete vybrať riadkový mód (terminal mode) správnym kliknutím na nejaké tlačidlo označené ako "Mode" alebo "Option" alebo dačo také. Občas sa vám stane, že takýmito prieskumnými akciami zlikvidujete nejaký počítač a nenapadne vás nič lepšie len ho reštartovať. Potom vás príde prizabiť niekoľko zúrivých userov, ktorí boli na ňom práve po sieti nalogovaní.

Všetko toto vôbec neplatí, ak nesedíte za počítačom ale za X-terminálom, ale to už je exotika

Dobre, toto sa nedá systematicky vysvetľovať, sú to mystériá tajného okultného spolku. Zasväcovanie trvá dlho a dôležité je, aby plné pochopenie prišlo skôr ako senilita.

Takže už len dva obrázky mystických tabuliek s magickými slovami tohto bratstva. Je to reklama stiahnutá z `http://www.oreilly.com/catalog/debian/chapter/book/index.html`

Linux Quick Reference

SOME USEFUL COMMANDS

Command	Task
File/Directory Basics	
ls	List files
cp	Copy files
mv	Rename files
rm	Delete files
ln	Link files
cd	Change directory
pwd	Print current directory name
mkdir	Create directory
rmdir	Delete directory
File Viewing	
cat	View files
less	Page through files
head	View file beginning
tail	View file ending
nl	Number lines
od	View binary data
xxd	View binary data
gv	View Postscript/PDF files
File Creation and Editing	
emacs	Text editor
vim	Text editor
umask	Set default file protections
soffice	Edit Word/Excel/ PowerPoint docs
abiword	Edit Word documents
gnumeric	Edit Excel documents
File Properties	
stat	Display file attributes
wc	Count bytes/words/lines
du	Measure disk usage
file	Identify file types
touch	Change file timestamps
chown	Change file owner
chgrp	Change file group
chmod	Change file protections
chattr	Change advanced file attributes
lsattr	List advanced file attributes

Command	Task
File Location	
find	Locate files
slocate	Locate files via index
which	Locate commands
whereis	Locate standard files
File Text Manipulation	
grep	Search text for matching lines
cut	Extract columns
paste	Append columns
tr	Translate characters
sort	Sort lines
uniq	Locate identical lines
tee	Copy stdin to a file and to stdout simultaneously
File Compression	
gzip	Compress files (GNU Zip)
compress	Compress files (Unix)
bzip2	Compress files (BZip2)
File Comparison	
diff	Compare files line by line
comm	Compare sorted files
cmp	Compare files byte by byte
md5sum	Compute checksums
Disks and Filesystems	
df	Show free disk space
mount	Make a disk accessible
fsck	Check a disk for errors
sync	Flush disk caches
Backups and Remote Storage	
mt	Control a tape drive
dump	Back up a disk
restore	Restore a dump
tar	Read/write tape archives
cdrecord	Burn a CD
rsync	Mirror a set of files

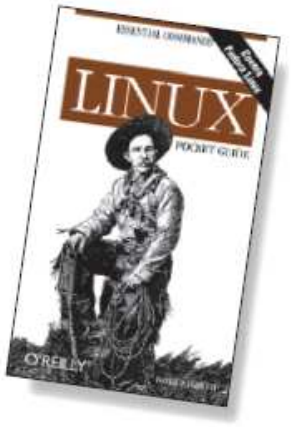
O'REILLY®
www.oreilly.com

Linux Quick Reference

SOME USEFUL COMMANDS

Command	Task
Printing	
lpr	Print files
lpq	View print queue
lprm	Remove print jobs
Spelling Operations	
look	Look up spelling
aspell	Check spelling interactively
spell	Check spelling in batch
Processes	
ps	List all processes
w	List users' processes
uptime	View the system load
top	Monitor processes
xload	Monitor system load
free	Display free memory
kill	Terminate processes
nice	Set process priorities
renice	Change process priorities
Scheduling Jobs	
sleep	Wait for some time
watch	Run programs at set intervals
at	Schedule a job
crontab	Schedule repeated jobs
Hosts	
uname	Print system information
hostname	Print the system's hostname
ifconfig	Set/display network information
host	Look up DNS
whois	Look up domain registrants
ping	Check if host is reachable
traceroute	View network path to a host

Command	Task
Networking	
ssh	Securely log into remote hosts
telnet	Log into remote hosts
scp	Securely copy files between hosts
sftp	Securely copy files between hosts
ftp	Copy files between hosts
evolution	GUI email client
mutt	Text-based email client
mail	Minimal email client
mozilla	Web browser
lynx	Text-only web browser
wget	Retrieve web pages to disk
slrn	Read Usenet news
gaim	Instant messaging/IRC
talk	Linux/Unix chat
write	Send messages to a terminal
mesg	Prohibit talk/write
Audio and Video	
grip	Play CDs and rip MP3s
xmms	Play audio files
cdparanoia	Rip audio
audacity	Edit audio
xcdroast	Burn CDs



O'REILLY®
www.oreilly.com

Excerpted from Linux Pocket Reference

Kapitola 2

Java

V tejto kapitole si stručne zopakujeme niečo o jazyku Java. Nebude to samonosný učebný text. Predpokladáme, že čitateľ má s jazykom Java elementárne skúsenosti. Tu si len pripomenieme základné konštrukcie Javy. Podrobnejšie spomenieme len techniky a programové balíky, ktoré budeme v počítačovom praktiku potrebovať, ale s ktorými sa možno čitateľ v základnom kurze Javy nestretol.

2.1 Čo je Java

Počítač je zložité zariadenie, je tam procesor, pamäť, diskové jednotky, klávesnica, displej, modem, karta pre pripojenie na Internet, webkamera, mikrofón, CD napaľovačka, USB, všeličo. Súhru toho všetkého navzájom, komunikáciu jednotlivých uzlov, vôbec riadenie, zaobstaráva operačný systém (Windows, Linux, Mac OS, UNIX, VMS, mnoho iných). Operačný systém je program nízkej úrovne, priamo komunikuje s hardvérovými komponentami. Tie komponenty a operačný systém musia byť navzájom kompatibilné. Nemôžete len tak kúpiť nový modem, pripojiť ho na počítač, i keď sa to napríklad hardvérovo dá a modem alebo počítač nezhorí. Ešte sa môže stať, že operačný systém a modem si nerozumejú. Operačný systém ovláda nejaký hardvérový interfejs, ten tomu modemu posiela elektrické signály podľa pokynov operačného systému, ale modem tým signálom nerozumie. Lebo je zvyknutý komunikovať s iným operačným systémom. Mám doma ADSL modem pripojený cez univerzálny USB port, pod Windowsami nemá problém, ale s linuxom si nerozumie. Niekedy sa to dá riešiť tak, že výrobca (alebo šikovný hacker) napíše softvérový medzikus, driver, ktorý sa zaradí medzi operačný systém a hardvérový interfejs a ten driver tlmočí príkazy operačného systému pôvodne koncipované pre nejaký iný modem do príkazov, ktorým sa vyrobia elektrické signály, ktorým rozumie ten cudzí modem. Niekedy sa to proste nedá, aj v bežnej reči poznáme nepreložiteľné prekladateľské oriešky typu "Kolik třešní, tolik višní". Linuxová podpora pre ten môj modem neexistuje. Píšem tento text a prepínam sa medzi Windowsami a linuxom. Potrebujem inštalovať zo siete nové balíky do linuxu. Musel som si kúpiť iný modem, ktorý vie žiť s oboma systémami.

Prečo to tak zdĺhavo vykladám. Aby bolo jasné, že keď programátor píše nejaký program, ktorý má niečo kresliť na obrazovke, niečo čítať z disku a niečo posielať po sieti, nie je to také jednoduché. Programátor píše v jazyku, ktorý je ľudsky zrozumiteľný, a kompilátor to prepíše do jazyka, kódu, ktorý je zrozumiteľný počítaču. Tak hovorí ľudová múdrosť. Ale nie je to celá pravda. Ono je to zrozumiteľné ani nie tak počítaču ako operačnému systému. Programátor neprikazuje obrazovke, aby zobrazila "A". Programátor v skutočnosti požiada operačný systém, aby on prikázal obrazovke, aby napísala "A". Takže program, ktorý napíše programátor a kompilátor ho skompiluje, je programom nie pre počítač ale pre počítač spolu s operačným systémom. Napíšte program pre PC s Windowsami, spustite na tom istom PC linux a ten program nepobeží. Niektoré programy sú tzv. prenositeľné: skompilujete to kompilátorom, ktorý vyrobí program pre Windowsy a pobeží to pod Windowsami. Potom ten istý program skompilujete kompilátorom pre linux a pobeží to pod linuxom. Jednoduché programy napísane v c-éčku budú takto fungovať. Ale nenamaľujete s nimi na obrazovke ani obyčajnú čiaru.

A potom prišla Java. Trik je podobný ako s hardvérovými drajvermi. Pre každý operačný systém je treba urobiť softvérový medzikus, ktorý sa zaradí *pred* operačný systém, teda medzi aplikačný program a operačný systém. V Jave sa tento medzikus volá JVM (Java Virtual Machine). Ako názov hovorí, je to vlastne virtuálny počítač, s virtuálnym procesorom, virtuálnym displejom , virtuálnym operačným systémom. Program napísaný a skompilovaný pre Javu komunikuje s týmto virtuálnym počítačom JVM. Java program "si myslí" že komunikuje so skutočným počítačom. Podobne, ako keď hráte s počítačom šachy, môžete si myslieť, že hráte so skutočným šachistom. Ale je to len simulovaný virtuálny šachista.

Takže Java program a Windows spolupracujú tak, že Windowsy spustia simulátor JVM, ten predstiera, že je počítač, ktorý si z Java programom dobre rozumie, a potom tlmočí spôsobom, ktorému rozumejú zase Windowsy, požiadavky na hardvér, ktorý naozaj niečo urobí. Java program a linux spolupracujú tak, že linux spustí *svoj* simulátor JVM, ktorý si z Java programom dobre rozumie, atď.

Takto pracujú programy, napísané v Jave, ktorým sa hovorí **aplikácie**. Mierne inak fungujú programy, ktorým sa hovorí **applety**[1].

Hoci v tomto praktiku applety nebudeme používať, niekoľko poznámok aj o nich. Applet nie je samostatne funkčný, potrebuje web browser, ktorý okolo neho vytvorí potrebné užívateľské prostredie. Funguje to tak, že na počítači beží browser, ktorý keď na zobrazovanej stránke narazí na applet, stiahne appletový kód zo servera, spustí Java Virtual Machine, ktorú požiada, aby mu appletový kód interpretovala a potom výsledok tej interpretácie zobrazí akoby súčasť stránky, ktorú prezentuje.

V pozadí tejto komplikovanej konštrukcie je všeobecná myšlienka, ktorú si prosím prečítajte a zamyslite sa nad ňou.

[1]Slovo applet je tak trocha slovná hračka, môže to znamenať jabĺčko (apple → applet). Niektoré knihy to aj tvrdia. Ale pravdepodobnejší sa mi zdá výklad že je to "malá aplikácia (application → applet). Toto tvrdia napríklad Looney Tunes V komiksovom tutoriáli od Warner Bross Looney Tunes teach the Internet (http://www.warnerbros.com/ltti/)

Toto si prečítajte!

Na svete bol web a pomocou neho sa dali rýchlo hľadať a čítať informácie. A bola tam aj možnosť po webe posielať a v browseri ukazovať obrazové informácie. Ale bol s tým problém. Obrázok je pre počítač spravidla veľká kopa farebných bodiek zobrazených tesne jedna vedľa druhej, aby vznikol dojem obrazu. Tých bodiek je strašne veľa. Dnes sú siete rýchle, ale vo webovom stredoveku ten prenos trval veľmi dlho, pamätníci vedia, že sa v browseri vypínalo zobrazovanie obrázkov, aby sa nezaťažovala sieť. Potom niekomu napadlo, že obrazové súbory treba komprimovať, zip niečo pomohol, ale málo. Vymyslel sa JPEG a dramaticky to znížilo prenosové nároky, možno 10-krát. Potom si niekto všimol, že mnoho tých obrázkov vzniklo nie fotografovaním, ale na počítačoch, pomocou príkazov počítačovej grafiky. To je ono! Pár jednoduchých príkazov môže vyrobiť obrázok, ktorý na prvý pohľad vyzerá zložito. A JPEG je prihlúpy na to, aby došiel na to, čo sa deje. Takže nápad: namiesto hotového obrázku poslať grafický program, ktorý ten obrázok na druhom konci sveta nanovo nakreslí. Pár príkazov typu

g.drawLine(20,30,300,540)

Lenže bol problém: na web serveri nie je jasné, aký počítač s akým procesorom a akým operačným systémom bude na druhej strane webu ten obrázok kresliť. Bolo treba niečo absolútne prenositeľné. Narodil sa Java applet.
A teraz pozor! Znamená to toto: ak porozumiem tomu, čo je na obrázku nakreslené, viem to spolubratovi komunikovať oveľa efektívnejšie, ako hocijakým iným trikom. Treba proste prísť na to, ako je **to** spravené. A teraz to otočme. Ak si máme zapísať alebo komunikovať **čosi**, čo sme pozorovali, a prídeme na nejaký nečakane efektívny a krátky spôsob zápisu, potom môžme pojať podozrenie, že sme prišli na to, ako to **čosi** bolo spravené, ako funguje. Tomu sa hovorí teoretická fyzika. Mať dobrú teóriu znamená mať veľmi efektívne zozipovanú informáciu. Ľudia dlho zhromažďovali množstvo kilobytov starých pergamenov a hlinených tabuliek o pozorovaní oblohy. A potom prišiel Ptolemaios a celé to zozipoval do cyklov a epicyklov a epicyklov nad epicyklami. Geniálny zip! Hoci dnes sa to v niektorých textoch vykladá tak, ako keby to bol dajaký trkvas. Potom Koperník, Tycho a Kepler to postupne zozipovali do elíps so spoločným ohniskom. A potom, v azda najväčšom rozmachu ľudského ducha všetkých čias, to Newton zozipoval do dvoch rovníc

$$\begin{aligned} \overrightarrow{F} &= m\overrightarrow{a} \\ \overrightarrow{F} &= -\kappa\frac{m_1 m_2}{r^3}\overrightarrow{r} \end{aligned}$$

Tam zozipoval nielen planéty a Slnko, ale aj každé jablko padajúce z každého stromu, aj loptu, ktorá v zápase NBA trafila do koša a priniesla strelcovi milióny.

J.Satinský by tento fejtón asi skončil slovami: Oznamujem láskam Vašim, že toto je prvá myšlienka, ktorú vám na týchto praktikách chcem natĺcť do hlavy, že ako funguje tá fyzika.

2.2 Základné konštrukcie Javy

Základné konštrukcie jazyka Java tu nebudeme systematicky opakovať. Namiesto toho uvedieme krátky zdrojový program BasicJava.java, ktorý nemá žiadny rozumný funkčný zmysel, je to len ukážka. Na malý priestor sme povkladali všetky základné jazykové konštrukcie,

takže čitateľ, ktorý to všetko dakedy vedel, ale už niečo zabudol, si rýchle môže potrebnú konštrukciu vyhľadať. Vysvetľovať nič nebudeme, čitateľ sa už rozpamätať musí sám, prípadne sa musí vrátiť k nejakému učebnému textu o jazyku Java. Dobré zdroje nájde aj na sprievodnom CD.

Poznamenávam ešte, že komentáre v listingoch sú písané bez diakritiky, lebo do listingu mäkčene jednoducho nepatria. (Do zdrojákov nepatria v skutočnosti slovenské komentáre. Aj hudobníci do nôt píšu "allegro" a nie "fčuľ friško".)

Listing 2.1: BasicJava.java

```
1  //Nasledujuci program nema ziaden uzitocny vyznam, je len ukazkou zakladnych
2  //konstrukcii jazyka Java, pre tych, ktori vedeli, ale zabudli, a potrebuju si
3  //rychlo pripomenut. Kto nevedel, nebude vediet, ani ked si toto precita.
4  //Teda okrem profikov, ktori programuju v inom jazyku, a Javu este nevideli
5  //
6  //Program sa da skompilovat a spustit, ale zamerne skonci run-time-errorom!!!!
7  //
8  public class BasicJava {
9    int n = 1000;
10   long m = 10000000000000L;
11   float r = 1.0f;
12   double d=1.0;
13   double []f = new double[4];
14   String s = "jojo";
15   boolean t = true;
16   //////////////constructors///////////////
17   public BasicJava() {
18   }
19
20
21   public BasicJava(int n){
22     this.n=n;
23   }
24
25   ////////////////simple statements//////////////////////
26   public void basicStatements(){
27   ///// cykly /////
28     for(int i=0;i<n;i++){
29       m++;
30     }
31     int i=0;
32     while(i<n){
33       i++;
34       m++;
35     }
36     i=0;
37     while(true){
38       i++;
39       if(i<100) continue;
40       m++;
41       if(i>=n) break;
42     }
43     i=0;
44     do{
45       i++;
46       m++;
47       if(i>=n)break;
48     }while(true);
49 ////////////////////////// if, else /////////////
50     if(n>100){
51       m++;
52       s = "100";
53     }
54     else if(n>200) s="200";
55     else if(n>300) s="300";
56     else s="0";
57 ////////////////////// switch /////////////
58     switch(n){
59       case 1: s="jedna"; break;
60       case 2: s="dva";break;
```

```
61        case 3: s="tri"; break;
62        default: s="undefined";
63      }
64    }
65  ////////////////// Arrays //////////////////////
66    public void Arrays(){
67      int [] aoi;  // array of ints (deklarovane ale pamat na tie int nepriradena)
68      double [] aod = new double[4]; //array od doubles (pamat alokovana)
69      double [] aod5 = new double[5];
70      //java nema viacindexove polia ale ma polia poli
71      double [][] aoaod = new double[3][4]; //array of arrays of double
72
73      aoi = new int[5]; // az tu sa sa alokuje pamat
74      for(int i=0;i<5;i++) aoi[i] = 2*i;
75
76      //tu je priklad na vyskusanie si ako funguje pole poli, ktory index robi co
77      //nebudeme to vysvetlovat, staci precitat si nasledujuci kusok kodu, zbehnut
78      //program a pozriet si vystup a potom podumat, co to tu vlastne ukazujeme
79      for(int i=0;i<4;i++) aod[i] = 0.;
80      for(int i=0;i<3;i++)
81        for(int j=0;j<4;j++) aoaod[i][j]= 1.2*i+4.8*j;
82      aod = aoaod[2];
83      for(int i=0;i<4;i++) System.out.println(aod[i]+" "+aoaod[2][i]);
84
85      //nasledujuci kusok je len pre fajnsmekrov, je zamerne chybne
86      //naprogramovany aby sa ukazala ista vlastnost javy
87      //pole aod5 je pridlhe, presledujte, co to spravi
88      for(int i=0;i<5;i++) aod5[i] = 0.;
89      System.out.println("dlzka pola aod5: "+aod5.length);
90      aod5 = aoaod[2]; //pole aod5 bolo deklarovane ako 5-prvkove
91                       //tu ale bolo presmerovane na stvorprvkovu strukturu!!!
92      //Ak uz na to prideme, presvedcime sa o tom nasledujucim vypisom, ale
93      //tak lahko nas to nenapadne.
94      System.out.println("dlzka pola aod5: "+aod5.length);
95      System.out.flush();
96      //na nasledujucom riadku sa to zruti, ale az pri behu programu, kompilacia
97      //samozrejme nenajde nic zle, ani programator si nemusi uvedomit
98      //ze pise zle: ved pole aod5 bolo deklarovane ako 5-prvkove
99
100     for(int i=0;i<5;i++) System.out.println(aod5[i]);
101
102     //Na jave je sympaticke ze ma run-time kontrolu pretecenia indexu pola
103     //takze program tu skonci cez ArrayIndexOutOfBoundsException
104     //Uplne pochopit, co sa vlastne stalo sa neda bez znalosti a istej
105     //skusenosti s tym, co su to pointre. Pointre v Jave nie su explicitne
106     //ale vnutorna realizacia je cez pointre a tu je jedno z miest, kde pointre
107     //vystrtcia rozky. Nemozno byt dobrym programatarom v Jave a neprecitat
108     //si c-cko a c++ a pochopit, co su pointre a ako su realizovane polia
109     //a vobec organizacia pamate, co je memory leak a co vlastne robi
110     //garbage collector v Jave
111
112   }
113
114 ////////////////////Math. functions ///////////////
115   public double Functions(double x){
116     switch(n){
117       case 0: return Math.sin(x); //tu nesmie byt break, lebo je tam return,
118                                   //k prikazu break by sa to nikdy nedostalo
119       case 1: return Math.sin(Math.PI/4.);
120       case 2: return Math.exp(x);
121       case 3: return Math.sqrt(x);
122       case 4: return Math.pow(Math.E,x);
123       case 5: return Math.log(x);
124       default: return 0.;
125     }
126
127   }
128
129 ////////////////////String operations, len zopar ukazok /////////////////
130   public String StringOps(String name){
131     switch(n){
132       case 0: return name;
133       case 1: name="Dr."+name; return name;//po return nesmie byt break
134       case 2: return name.toUpperCase();
```

```
135         case 3: return name.replace('i','y');
136         case 4: return name.substring(0,name.lastIndexOf("ova"));
137         case 5: return ("Vysledok je:"+ (d+Double.parseDouble(name)));
138         default: return "";
139       }
140
141     }
142 ////////////////Throwing and handling exceptions//////////
143     public double MySqrt(double x) throws IllegalArgumentException{
144       if(x<0) throw new IllegalArgumentException("x must be >=0");
145       else return Math.sqrt(x);
146     }
147     public double CorrecectedSqrt(double x){
148       double y;
149       try{
150         y = MySqrt(x);
151       }
152       catch (IllegalArgumentException e){
153         y = 0;
154       }
155       return y;
156     }
157 ////////////////////////////////////
158
159
160
161     public static void main(String[] args) {
162       System.out.println("Tento program zamerne konci run-time-errorom!");
163       BasicJava bJ = new BasicJava();
164       // vypis command line parametrov
165       for(int i=0;i<args.length;i++)
166         System.out.println(args[i]);
167       bJ.n=4;
168       System.out.println(bJ.StringOps("Kirschenrova"));
169       bJ.n=5;
170       System.out.println(bJ.StringOps("1.4"));
171       bJ.Arrays();
172     }
173
174 }
```

2.3 Edit – Compile – Run

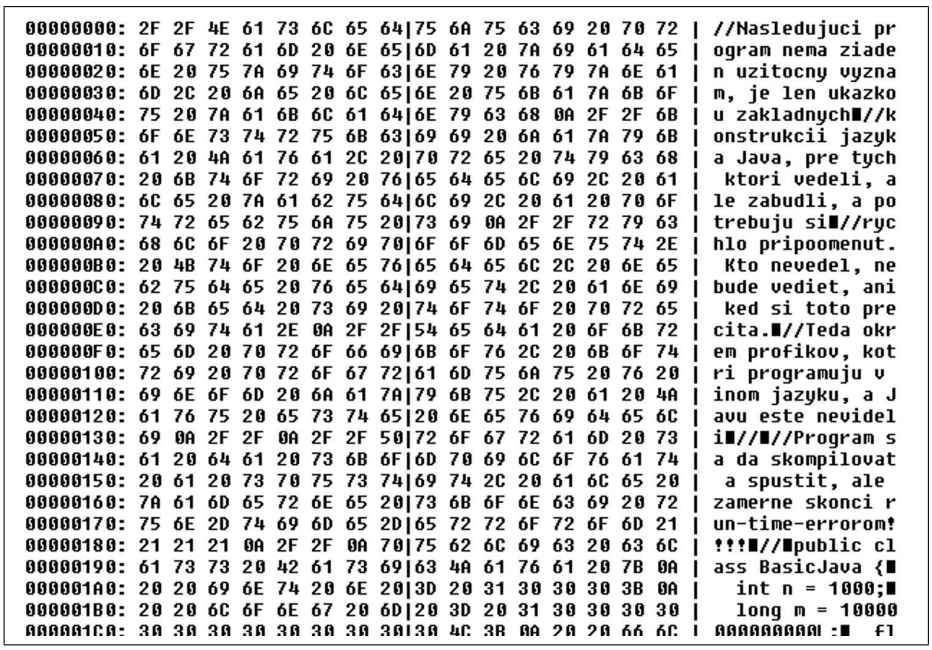
Programátori sú spravidla zvyknutí pracovať v neustále sa opakujúcom pracovnom cykle edituj, kompiluj, spusti. Pri práci s Javou je to trocha inak: kompilácia a "spustenie" majú mierne iný význam.

V predchádzajúcom odseku sme videli dlhý demo-program v Java. Uložený je v súbore `BasicJava.java`. Je to textový súbor, ktorý môžeme, ak treba, upravovať nejakým textovým editorom. V praktiku budeme používať editor `SciTE`. Zdôraznime pre istotu, že to musí byť textový editor, nie word procesor, word processor pridáva k textu rôzne pomocné informácie o formátovaní textu. Zdrojový súbor pre kompilátor Javy musí byť uložený ako prostý textový súbor znak po znaku a nič iné. Hexadecimálny výpis súboru (spolu s pomocným znakovým výpisom v pravom stĺpci) vidíme na obr.2.1. Súbor začína dvoma lomítkami, hexadecimálny výpis dvoma rovnakými hexadecimálnymi číslami 2F, z čoho môžme usúdiť, že hexadecimálny kód lomítka je 2F, čo v desiatkovej sústave znamená

$$2 * 16 + 15 = 47$$

Vo štvrtom riadku výpisu na hexadecimálnej pozícii 4C je kód 0A, teda v desiatkovej sústave 10. V textovom výpise v pravom stĺpci vidíme čudný znak štvorček. Znamená to, že kódu

0A nezodpovedá žiadny znak abecedy. Je to v skutočnosti formátovací znak a jeho význam je "prejdi na nový riadok" (ASCII line feed). V textovom súbore teda v skutočnosti môže byť aj niekoľko málo typov formátovacích znakov. Textový editor tieto formátovacie znaky nezobrazuje, len prípadne vykoná činnosť, ktorú kódujú, teda v danom prípade prejde na nový riadok[2].



Obrázok 2.1: Hexadecimálný výpis súboru BasicJava.java

Zdrojový súbor BasicJava.java skompilujeme príkazom

```
javac BasicJava.java
```

[2]Kódovanie anglickej abecedy dvojcifernými hexadecimálnmymi číslami pomocou kódu, ktorý sa volá ASCII (American Standard Code for Information Interchange) vymysleli ešte v dobách pred počítačmi diaľnopisoví ľudia. Preto okrem znakov abecedy tam zakódovali aj niekoľko znakov na ovládanie diaľnopisu na diaľku. Diaľnopis, ktorý prijal ASCII kód 0A pootočil valec s papierom na ďalší riadok, diaľnopis, ktorý prijal kód O7 (ASCII bell) cengol zvončekom, po prijatí kódu 0D (ASCII carriage return) presunul valec s papierom na ľavý doraz, teda na začiatok riadku. Diaľnopis, ktorý má začať písať na ďalší riadok musel prirodzene vykonať operácie 0D aj 0A. Kto ešte nevidel mechanický písací stroj s valcom, ktorý chodí zľava doprava a naspäť pomocou veľkej chrómovanej páky ktorá súčasne s cvrkaním pretáča valec na nový riadok, príďte ku mne do pracovne, mám tam krásneho veterána.

Preto v textových súboroch, určených pre operačné systémy DOS/Windows sa koniec riadku označuje dvojicou znakov 0D 0A. V súboroch určených pre Unix/linux je koniec riadku označovaný len znakom 0A (lebo "veď každý vie že na novom riadku treba začať písať od ľavého okraja"). Táto nejednotnosť vedie k zmätkom pri prenosoch súborov medzi rozličnými systémami a treba si na to dať pozor. Niekedy chcú byť počítače zvlášť priateľské a bez upozornenia pri prenosoch pridajú alebo uberú znak 0D na konci riadku. Ako Timur a jeho družina, ktorí ochotne previedli babku cez cestu, hoci sa bránila. Takže pri prenosoch ftp pozor na správnu voľbu módu binary.

Poznamenajme, že Java vznikla v c-čkovej komunite a unix/linux písali c-čkári, takže java má linuxovú konvenciu na označovanie konca riadku len znakom 0A a to aj vtedy, keď beží pod Windowsami. Lebo veď, ako vieme, ona nebeží pod Windowsami ale na virtuálnom stroji JVM.

Preto ak máme v Jave dlhý reťazec (String), ktorý má popisovať viacero riadkov textu, vkladáme ako oddeľovač riadkov "escape sekvecniu" \n, čo prosto znamená znak s hex-kódom 0A.

výsledkom bude súbor preložený do jazyka, ktorému rozumie JVM s menom `BasicJava.class`

Súbor typu `*.class` nie je samostatne spustiteľný, lebo nie je v jazyku fyzického počítača, na ktorom pracujeme. Je v jazyku virtuálneho počítača JVM. Preto by sme márne klikali na ten súbor tak, ako sme v grafickom prostredí zvyknutí spúšťať programy.

Ak chceme vidieť výsledok nášho programátorského snaženia, musíme najprv odštartovať virtuálny počítač JVM a tomuto počítaču podstrčiť program, ktorý sme skompilovali. Oboje naraz sa spraví príkazom

```
java BasicJava
```

Príkazom java spúšťame JVM, ďalšie informácie v príkazovom riadku sú už dáta pre počítač JVM. String `BasicJava` má význam `BasicJava.class`, ale príponou `class` **musíme** vynechať.

Ak pracujeme s programátorsky priateľským editorom ako SciTE, potom si ho môžme nakonfigurovať tak, že stlačenie klávesu F7 vyvolá kompiláciu a stlačenie F5 zbehnutie programu príkazom java. Je to šikovné ale podporuje to reflexy a pomáha vypínať mozog. Nič proti reflexom, aj najinteligentnejší hokejista musí používať reflexy, ak chce byť dosť rýchly. Ale naozaj dobrý hokejista tvorí hru mozgom[3]. Nič proti malým zošitkom, tzv. blbníčkom, popísaných symbolmi ako F5. Ale prídete k inak nakonfigurovanému počítaču a blbníček nepomôže. Skúste si spustiť kompiláciu aj "manuálne" a skúste prísť na to, kde je to v tom SciTE zariadené (nakonfigurované) že to F7 naozaj kompiluje.

Treba tomu rozumieť viac, lebo dakedy treba použiť zložitejšie formy oných príkazov na kompiláciu a run.

Treba napríklad vedieť, že kompilátor javac pri kompilácii nejakej triedy (teda súboru *.java), potrebuje mať prístup k už skompilovaným triedam, na ktoré sa tá práve kompilovaná trieda odvoláva.

Ak ide o triedy, ktoré sú priamo v jazyku Java, potom to nie je problém, JVM vie, kde ich má hľadať. Ale ak sú to triedy, ktoré ste si predtým skompilovali sami, alebo používate knižnice (a teda triedy v nich obsiahnuté) z iných zdrojov, potom musíte JVM napísať, kde ich hľadať, teda v ktorých direktóriách sa tie používané triedy nachádzajú. Zoznamu tých direktórií sa hovorí "classpath". Ten zoznam je reťazec vytvorený z plne kvalifikovaných mien všetkých relevantných direktórií navzájom oddelených dvojbodkami (pod linuxom), resp bodkočiarkami (pod Windowsami). Zložitý príkaz na kompiláciu môže teda vyzerať napríklad takto

```
Pod Windowsami
javac -classpath .;c:\moje1;d:\tvoje\kniznica1 Trieda.java

Pod linuxom
javac -classpath .:/moje1:/tvoje/kniznica1 Trieda.java
```

[3]To tvrdia Hríb s Kušnierikom a oni majú permanentky na Slovan, takže sa vyznajú.

Všimnite si, že zoznam začína bodkou, ktorá má význam "aktuálne direktórium". Keby sme tam tú bodku nedali, kompilátor by nenašiel tie skompilované triedy, ktoré sa nachádzajú v aktuálnom direktóriu.

Niekedy poskytovateľ knižnice zhromaždí všetky skompilované triedy do archívneho súboru typu ".jar". Na takýto súbor sa potom hľadí akoby na direktórium, teda môžme ho uviesť v zozname classpath a kompilátor uvidí všetky skompilované triedy uschované v citovanom .jar súbore. Príkazy na kompiláciu teda môžu znieť napríklad takto

```
Pod Windowsami
javac -classpath .;c:\moje1;d:\tvoje\kniznica1;d:\jeho\lib.jar Trieda.java

Pod linuxom
javac -classpath .:/moje1:/tvoje/kniznica1:/jeho/lib.jar Trieda.java
```

Zoznamy potrebných direktórií sú niekedy dlhé, preto je možné uložiť celý zoznam classpath do systémovej (environment) premennej CLASSPATH. Potrebný zoznam tak vypíšem iba raz a potom už používam príkaz javac bez udania zoznamu classpath. Program javac totiž automaticky prehľadáva systémovú premennú CLASSPATH, ak je definovaná.

Príkazy na kompiláciu potom znejú takto

```
Pod Windowsami
set CLASSPATH = .;c:\moje1;d:\tvoje\kniznica1;d:\jeho\library.jar
javac Trieda.java

Pod linuxom so shellom typu sh, bash
CLASSPATH=.:/moje1:/tvoje/kniznica1:/jeho/library.jar Trieda.java
export CLASSPATH
javac Trieda.java

Pod linuxom so shellom typu csh, tcsh
setenv CLASSPATH .:/moje1:/tvoje/kniznica1:/jeho/library.jar Trieda.java
javac Trieda.java
```

Všetko, čo sme povedali o zoznamoch classpath pre kompilátor javac, platí rovnako aj pre program java.

2.4 Test inštalácie

Ak si inštalujete prostredie pre toto praktikum na svojom počítači, otestujte si, či funguje, ako má. Na sprievodnom disku sú pripravené základné testy.

2.5 IO v Jave

Tento odsek venujeme jednoduchým vstupno-výstupným operáciám (IO, input-output) v Jave.

V celej komplexnosti je to zložitá tematika, ktorá presahuje rámec tohto textu. Princípy nebudeme vôbec vysvetľovať, ukážeme si len použitie jednoduchých prostriedkov.

Pod IO tu budeme rozumieť zápis a čítanie textových diskových súborov a štandardného vstupu a výstupu.

Čo je súbor na disku, každý vie. Nám pôjde o textové súbory, teda súbory ktoré sú zapísané v bezprostredne ľudsky čitateľnom tvare v zásade pomocou znakov abecedy a znakov pre cifry. (Znaky abecedy i cifry sú samozrejme kódované pomocou čísel, jednotlivé znaky ale navzájom nezávisle. Takže ak viem, že kód znaku A je 65, znak B je 66, C je 67 a podobne, môžem textový súbor priamo "čítať". Ďalšou charakteristikou textových súborov je, že sú organizované do textových riadkov, podobne ako text na papieri. Jednotlivé riadky nemusia mať a ani nemávajú fixný počet znakov. Sú navzájom oddelené špeciálnymi kódmi (napr. 10, 13) ktoré nezodpovedajú žiadnemu znaku abecedy, iba signalizujú pre počítač, že je koniec riadku.

Na rozdiel od textových súborov existujú binárne súbory, kde sa používa ľudsky netransparentné kódovanie. Napríklad čísla nie sú zapísané pomocou cifier v desiatkovej sústave ale sú kódované do "počítačového" binárneho kódu, ktorý je často komplikovaný a často pre rôzne počítače alebo rôzne operačné systémy iný. Binárne súbory okrem čísel môžu niesť kódovanú informáciu takmer o čomkoľvek, môžu kódovať obrazy, zvukový signál, zašifrovaný text, komprimovaný videosignál. Binárne súbory sú neinterpretovateľné bez znalosti kódovacích protokolov. Nebudeme sa nimi zaoberať.

Okrem diskových súborov budeme používať ešte štandardný vstup (STDIN) a štandardný výstup (STDOUT). Nie je to síce úplne presné, ale pod štandardným vstupom si môžeme predstaviť klávesnicu a pod štandardným výstupom obrazovku, presnejšie (ak používame oknovo orientovaný grafický užívateľský interfejs ako Windows alebo Linux-KDE) myslíme tým textové terminálové okno otvorené na pracovnej ploche pre našu konkrétnu aplikáciu.

Najjednoduchšou vecou je výstup na STDOUT. Java ma pripravený príjemný polotovar, volanie metódy

```
System.out.println(String s)
```

Nemusíme rozumieť, prečo toľko bodiek a čo vlastne znamená "System" a "out", stačí ak to budeme vedieť používať. Metóda zobrazí v terminálovom okne riadok znakov zodpovedajúci stringu s a prejde na nový riadok. V Jave je príjemné, že fakticky každý objekt vie o sebe podať nejakú textovú informáciu v podobe String. A stringy sa dajú jednoducho spájať pomocou operátora +. Takže napríklad kódový snippet (bežný žargón označujúci útržok kódu)

```
int i= 12;
String name="Jozef";
System.out.println(name + " este nema " + i + " rokov.");
```

zobrazí zmysluplnú vetu, pričom automaticky nahradí celočíselnú premennú i jej textovou reprezentáciou.

Opak je menej príjemný, teda používanie STDIN. Existuje síce čosi, čo sa volá (nebudeme tu vnikať do toho, čo to vlastne je)

```
System.in
```

problém je ale v tom, že nenájdeme niečo šikovné, ako by mohlo byť

```
//ukazka zleho kodu, kompilacia skonci chybou
String s;
System.in.readln(s); //takato metoda neexistuje
```

Pre potreby tohto praktika je preto pripravený malý balíček (package) IO utilít

```
sk.uniba.fmph.pocprak.ioutils
```

obsahujúci súbory InTextFile.java a OutTextFile.java Balíček je súčasťou knižnice pocprak.jar. Ak ho chcete používať, musíte mať na ceste classpath viditeľný archívny súbor pocprak.jar (viď odsek 2.3)

Trieda OutTextFile umožňuje jednotným spôsobom zapisovať tak do STDOUT ako aj do ľubovoľného textového súboru na disku. V podstate rozširuje jednoduchý spôsob zápisu do STDOUT typu System.out.println pre textové súbory na disku.

Trieda InTextFile umožňuje jednotným spôsobom čítať textové riadky tak zo STDIN ako aj z ľubovoľného textového súboru na disku.

Triedy realizujú základnú technológiu práce programátora z diskovými súbormi podľa schémy

1. Najprv súbor otvorím
2. Potom do súboru píšem alebo ho čítam
3. Nakoniec súbor zatvorím

Namiesto dlhého vysvetľovania malá ukážka. Najprv príklad použitia triedy OutTextFile.

Listing 2.2: TestIOout.java

```
1  import sk.uniba.fmph.pocprak.ioutils.*;
2  public class TestIOout {
3
4    public static void main(String[] args) throws Exception{
5      OutTextFile ostd = OutTextFile.open();  //otvara STDOUT
6      OutTextFile of = OutTextFile.open("OutTest.txt"); //otvara diskovy subor
7      ostd.println("Toto je ukazka vystupu na STDOUT");
8      ostd.println(28.4 + "  " + 0.123);
9      of.println("Toto je ukazka vystupu do textoveho suboru");
10     of.println(28.4 + "  " + 0.123);
11     ostd.close(); //pre STDOUT je to nic nerobiaca operacia
12     of.close(); //zatvara subor
13   }
14 }
```

Pre profesionálov poznamenajme, že trieda OutTextFile rozširuje triedu PrintStream, ale metóda close() je prepísaná tak, že nehrozí, že by sme si zatvorili STDOUT: ak objekt triedy OutTextFile je totožný so STDOUT, metóda clsoe() nerobí nič.

Ukážme si teraz použitie triedy InTextFile

Listing 2.3: TestIOin.java

```
1  import sk.uniba.fmph.pocprak.ioutils.*;
2
3  public class TestIOin {
4
5    public static void main(String[] args) throws Exception {
6      InTextFile instd = InTextFile.open(); //otvara STDIN ako textovy subor
7      InTextFile inf = InTextFile.open("Test.txt"); //textovy subor na disku
8      System.out.println("Zadaj nejaky string a stlac ENTER"); //vypise na
9                                  //terminalovom okne prompt
10     String s=instd.readLine();//caka na input z klavesnice a nacita textovy riadok
11     System.out.println(s);//vypise nacitany riadok
12     instd.close();//pre STDIN je to nic nerobiaca operacia
13     double [] data;
14     while (true) {
15       data = inf.readlnDoubles();
16       if (data==null) break;
17       if (data.length==2) //vieme ze v riadkoch maju byt vzdy dve doule cisla
18         System.out.println(data[0]+"   "+data[1]);
19     }
20   }
21 }
```

Poznamenajme, že tento kód predpokladá, že na disku existuje súbor Test.txt, ktorý obsahuje riadky s číslami typu double navzájom oddelenými jednou alebo niekoľkými medzerami.

Pre profesionálov poznamenajme, že trieda InTextFile rozširuje triedu LineNumberReader a otvára diskové textové súbory ako objekty tejto triedy. Hlavný fígeľ je potom v tom, že STDIN (teda vlastne objekt System.in, ktorý je objektom triedy InputStream) rozšíri na objekt triedy LineNumberReader. Preto tak diskové textové súbory ako aj STDIN budú čitateľné riadok po riadku cez volanie metódy LineNumberReaader.readLine(), ktorá vráti aktuálny riadok súboru ako reťazec.

V riadku 15 je ukážka použitia metódy `readlnDoubles()` pre "priame" čítanie riadkov, na ktorých sú v textovom tvare zapísané čísla interpretovateľné ako double oddelené medzerami (jednou alebo niekoľkými). Inak by sme museli prečítať každý riadok ako String a potom jeho obsah dekódovať. Príklad možnej techniky dekódovania (uvádzame pre záujemcov) je v

nasledujúcom krátkom listingu. Je to kódový snippet metódy readlnDoubles vystrihnutý zo zdrojáka triedy InTextFile.

```
1  public double[] readlnDoubles()throws IOException{
2      String line = this.readLine();
3      if (line == null) return null;
4      java.util.StringTokenizer st =  new java.util.StringTokenizer(line, " ");
5      int n = st.countTokens();
6      double [] values = new double[n];
7      for(int i=0;i<n;i++){
8        values[i]=Double.valueOf(st.nextToken()).doubleValue();
9      }
10     return values;
11   }
```

Ak metóda readlnDoubles() narazí na koniec súboru a nemôže už prečítať ďalší riadok, vráti namiesto poľa double[] hodnotu null. Ak narazí na prázdny riadok, vráti pole nulovej dĺžky. Po týchto dvoch poznámkach by ste mali porozumieť predchádzajúcu ukážku použitia metódy readlnDoubles().

Viac ako prácu s triedami InTextFile a OutTextFile nebudeme v tomto praktiku z IO operácií potrebovať.

2.6 Java bez classov

Java je objektovo orientovaný jazyk. Až tak veľmi, že sa nedá napísať nič, kde by sa nevyskytovali classy. To je niekedy príliš puritánsky prístup: jednoduché veci si chcem spraviť jednoducho. Napísať krátky program z voleja bez rozmýšľania o jeho štruktúrovaní a návrhu tried, ktoré urobia, čo treba. Dá sa to ale ľahko urobiť aj v Jave tak, že všetko skryjem ako statické metódy a premenné do jednej triedy, ktorá nemá žiadnu programovú úlohu okrem faktu, že "zabalí" napísaný neobjektový program do objektovej podoby. Namiesto dlhého rečnenia jednoduchá ukážka. Mám napísať krátky program, ktorý prečíta súbor `obezita.txt` Ten súbor obsahuje dopredu neznámy počet riadkov, každý z nich obsahuje dve čísla (výška, hmotnosť) oddelené medzerou. Úlohou je určiť strednú hodnotu a varianciu výšky. Tu je možné riešenie ako listing.

Listing 2.4: SimpleProgram.java

```
1  import sk.uniba.fmph.pocprak.ioutils.*;
2  public class SimpleProgram {
3    static double[] height= new double[5000];
4    static int nevent = 0;
5    static void ReadData() throws Exception{
6      double[] data;
7      InTextFile in = InTextFile.open("obezita.txt");
8      while (true){
9        data = in.readlnDoubles();
10       if(data==null) break;
11       if(data.length==2){
12         height[nevent]=data[0];
13         nevent++;
14       }
15     }
16     in.close();
17   }
18   public static void main(String[] args) throws Exception{
19     ReadData();
20     double sum = 0.;
```

```
21      double sum2 = 0.;
22      for(int i=0;i<nevent;i++){
23        double h = height[i];
24        sum = sum + h;
25        sum2 = sum2 + h*h;
26      }
27      double mean = sum/nevent;
28      double sigma = Math.sqrt(sum2/nevent-mean*mean);
29      OutTextFile STDOUT = OutTextFile.open();
30      STDOUT.println("Stredna hodnota = "+mean+"     sigma = "+sigma);
31    }
32  }
```

2.7 Náhodné čísla

Dôležitým nástrojom počítajúceho fyzika sú simulácie. Je to príspevok informatiky k technike myšlienkového experimentu: počítačom modelujeme priebeh nejakého fyzikálneho deja aby sme sa poučili, overili si funkčnosť návrhu nejakého experimentu alebo pomocou simulácie vypočítali niečo, čo analytickými technikami vypočítať nedokážeme. Pretože skutočné deje často ovplyvňuje náhoda (externé vplyvy, ktoré nemáme pod kontrolou a javia sa ako náhodné), potrebujeme náhodu simulovať aj na počítači. Využívame pritom generátory náhodných čísel, ktoré sú súčasťou knižnice každého slušného programovacieho jazyka.

Základný náhodný generátor generuje náhodné čísla typu double rovnomerne rozdelené vnútri intervalu $(0, 1)$. V začiatkoch vývoja počítačov sa ľudia pokúšali pracovať s "naozajstnými" náhodnými číslami, generovanými pomocou reálnych náhodných fyzikálnych procesov: napríklad registrovali šum elektrónkového zosilňovača a z tohto náhodného elektrického signálu získavali náhodné hodnoty. Ukázalo sa, že takéto generátory boli značne nestabilné a navyše často nie "dosť náhodné": neprežili obvyklé štatistické testy náhodnosti. Až v posledných rokoch sa výskum obracia znova k náhodným fyzikálnym procesom: už sú komerčne dostupné hardvérové náhodné generátory využívajúce principiálne náhodný charakter kvantovej mechaniky. V bežnej praxi sa však používajú generátory tzv. pseudonáhodných čísel. Tieto generátory produkujú postupnosti čísel, ktoré sú absolútne deterministické, sú generované jednoduchým algoritmom, v ktorom náhoda nehrá žiadnu rolu. Pre pozorovateľa, ktorý vidí len generovanú postupnosť čísel sa tie čísla zdajú úplne náhodné. Ak na generovanú postupnosť aplikuje známe štatistické testy náhodnosti, neobjaví žiadnu zákonitosť.

2.7.1 Generický náhodný generátor

Nebudeme teoretizovať, ale je dobré, keď programátor má hrubú predstavu o tom, ako taký generátor môže pracovať. Uvedieme si tu generátor populárny začiatkom sedemdesiatych rokov, dnes používame zložitejšie a oveľa kvalitnejšie generátory. Tu je jednoduchý zdrojový program.

Listing 2.5: Rndm.java

```
1  /**
2   * Jednoduchy random number generator. Sluzi len ako ukazka principu
3   * funkcie generatora, pre prakticke pouzitie je to uz zastarala
4   * koncepcia a neodporuca sa pouzivat
5   * generuje pseudonahodne cisla rovnomerne rozdelene v intervale (0,1)
```

```
6   */
7   public class Rndm {
8     private static int seed = 1234567;
9     private static final long mult = 69069;
10    private static final long m =2147483647; //2^31 -1
11    private static final double rescale = 1./(double)m;
12    /**
13     * Pri kazdom zavolani vrati dalsie pseudonahodne cislo zo sekvencie
14     * nastartovanej hodnotou seed platnou pri prvom zavolani
15     * @return double: hodnota generovaneho nahodneho cisla v intervale (0,1)
16     */
17    public static double rndm(){
18      //calculates mult*seed  mod  m
19      long tmp = mult*(long)seed;
20      seed = (int)(tmp - (tmp/m)*m); //hodnota seed je vacia ako 0 a mensia ako m
21      return seed*rescale;//preskalovanie; generovane cislo bude z inetrvalu (0,1)
22    }
23    /**
24     * Nastvi novu hodnotu pre seed, teda startovacie cislo generatora, ktore
25     * determinuje generovanu postupnost. Znovunastartovanie s rovnakou hodnotou
26     * seed vygeneruje identicku postupnost pseudonahodnych cisel
27     * @param s int: cislo, ktore sa pouzije ako seed pri generovany dalsich
28     * clenov postupnosti
29     */
30    public static void setSeed(int s){
31      seed = s;
32    }
33    /**
34     * Vrati momentalne aktualnu hodnotu seed. Pri zastaveni generatora a jeho
35     * znovunastartovani s touto hodnotou seed sa pokracuje v generovani
36     * pseudonahodnej postupnosti ako keby k zastaveniu nebolo prislo
37     * @return int: aktualna hodnota seed
38     */
39    public static int getSeed(){
40      return (int)seed;
41    }
42    /**
43     * Constructor je privatny, aby sa zabranilo vytvarat objekty tejto triedy.
44     */
45    private Rndm(){}
46    }
```

Prv ako okomentujeme algoritmus a jeho použitie, malá programátorská poznámka. Všimnime si privátny constructor. Atribút private znamená, že ho užívateľ nemôže volať, a teda nemôže vytvoriť žiaden objekt tejto triedy. Tak je to naozaj myslené. Táto trieda má len statické dáta a metódy. Je to vlastne trik, ako definovať v jazyku Java globálne premenné a globálne funkcie. Z ľubovoľného miesta užívateľského programu môžem volať statickú metódu, napríklad takto

```
double r=Rndm.rndm();
```

Programátori jazyka C++ v tom zbadajú, že sa efektívne vyrobil namespace Rndm.

Všimnite si tiež ako sú formátované komentárové riadky. Je to konvencia Javy, ktorá potom môže automaticky vyrobiť dokumentáciu k vytvorenému produktu príkazom

```
javadoc Rndm.java
```

Skúste si, akú komplexnú dokumentáciu to vyrobí.

Teraz k samotnému algoritmu. Použitý algoritmus je zastaralý, ale veľmi jednoduchý a demonštruje základné črty i moderných náhodných generátorov.

Kľúčovou metódou generátora je funkcia Rndm.rndm(). Pri zavolaní vráti náhodné číslo z intervalu $(0, 1)$. Algoritmus generovania je založený na celočíselnej aritmetike, návratová hodnota sa vyrába reškálovaním z celého čísla v riadku 21. Vnútorne vlastne algoritmus generuje sekvenciu pseudonáhodných celých čísel z množiny $\{1, 2, 3, ..., 2^{31}-1\}$. Algoritmus je plne deterministický, nasledujúce číslo v sekvencii sa vyrobí z predchádzajúceho vynásobením fixným činiteľom a vypočítaním zvyšku pri delení číslom $2^{31} - 1$.(Riadky 19 a 20.)

Pri prvom volaní sa vychádza z pevne zvoleného (viď riadok 8) štartovacieho čísla zvaného seed (anglicky seed znamená semeno). Znamená to, že ak generátor využívame v nejakej aplikácii pri novom spustení sa bude generovať presne rovnaká postupnosť náhodných čísel ako pri predchádzajúcom spustení. To je na prvý pohľad vadné. Veď ak by sme na základe takého generátora naprogramovali nejakú počítačovú hru, potom by prebiehala pri každom spustení vždy rovnako (napríklad pri hádzaní počítačovou "kockou" by padali tie isté čísla v tom istom poradí ako minule).

Pri hrách to naozaj nie je dobrý nápad, preto pri takýchto aplikáciách je potrebné nastaviť napred nejakú "prvú náhodnú hodnotu" seedu zavolaním metódy `Rndm.setSeed(int s)`. Ako užitočný náhodný seed sa napríklad používa počet sekúnd, ktoré práve uplynulo od začiatku roku 1980, alebo čosi podobné, čo sa už nikdy nebude opakovať. Takýto trik spravidla zabezpečí dostatočne "náhodný" štart generátora. Niektoré knižnice takéto randomizované štartovanie ponúkajú automaticky bez zásahu programátora. Tam zase naopak, ak chcem mať možnosť opakovať už použitú sekvenciu, musím najprv nastaviť vhodný počiatočný seed (a zapamätať si ho pre budúce použitie).

Možnosť opakovania predošlej sekvencie náhodných čísel je pre programátora vitálna. Ak pri testovaní programu zbadá počas behu nejakú chybu, musí mať možnosť beh programu presne opakovať, aby mal šancu príčinu chyby identifikovať a potom odstrániť. ak by generátor generoval naozaj náhodné čísla, na pozorovanú chybu by sa mi už nemuselo nikdy podariť "natrafiť".

Ešte jedna dôležitá úloha seedu. Často musím beh programu prerušiť, hoci simulácia ešte netrvala dostatočne dlho, a teda doterajšie výsledky ešte nie sú štatisticky signifikantné. Ak pre nič iné, môže napríklad vypadnúť elektrina. Preto je potrebné pri plánovaní dlhých programových behov, zapisovať z času na čas všetky relevantné medzivýsledky, ktoré mi umožnia obnoviť beh programu od určitého štádia tak, ako keby k prerušeniu nebolo došlo. Pokiaľ ide o náhodný generátor, práve na to slúži procedúra `Rndm.getSeed()` ktorá vráti aktuálnu hodnotu premennej `seed`. Keď sa táto hodnota potom použije pri volaní metódy `Rndm.setSeed(int s)` a potom sa pokračuje v generovaní náhodných čísel, pokračuje sa v generovaní predchádzajúcej postupnosti tak, ako keby k prerušeniu nebolo došlo.

Uvedený algoritmus demonštruje ešte jednu vlastnosť obvyklú pre náhodné generátory: periodičnosť. Generátor spravidla je schopný generovať len konečný počet rôznych náhodných čísel, potom sa generované čísla začnú opakovať. Tu uvedený generátor má periódu rádovo 10^8. To pre náročné simulácie zďaleka nemusí stačiť.

V praxi používané generátory majú oveľa väčšie periódy a líšia sa kvalitou, s ktorou vyhovujú všakovým dômyselným testom náhodnosti. Spravidla určitá komunita fyzikov pracujúca v

jednej oblasti má "svoj" náhodný generátor, ktorému verí. Novic, ktorý chce pracovať v tej oblasti, si musí zistiť lokálny folklór, ku ktorému patrí aj "cechový" náhodný generátor.

2.7.2 Náhodné generátory v Jave

Java vo svojej štandardnej knižnici poskytuje akoby dva náhodné generátory, a síce v triede `java.lang.Math` existuje metóda

```
static double java.lang.Math.random()
```

ktorá vráti náhodné číslo z intervalu $(0, 1)$ Je to statická metóda (metóda triedy), teda dá sa rovno použiť kdekoľvek v kóde bez nutnosti konštrukcie nejakého objektu. Poznamenajme, že v súčasnej implementácii sa pri prvom použití automaticky inicializuje objekt triedy `java.util.Random` a potom sa vnútorne volá metóda `nextDouble()` toho objektu.

V triede `java.util.Random` je metóda

```
double java.lang.Random.nextDouble()
```

Keďže sa nejedná o statickú metódu, musíme pred jej použitím inicializovať objekt tejto triedy. Trieda má dva constructory

```
public java.lang.Random()
public java.lang.Random(int seed)
```

Pri použití prvého (default) constructora sa implicitne použije seed odvodený od času počítačových hodín a programátor nemá kontrolu aká sekvencia náhodných čísel bude generovaná. Odporúčam preto používať inicializáciu s explicitným použitím seedu. Z rovnakých dôvodov neodporúčam používať generátor triedy `java.lang.Math`.

Generátor `java.util.Random` má i metódu `java.util.Random.setSeed(long seed)`, ktorou sa dá nastaviť seed, ale nemá metódu typu `getSeed()` potrebnú pre použitie v pokračovacích runoch.To je vážny nedostatok.

Pre potreby nášho praktika tento generátor plne postačuje, má i metódu generujúcu gaussovské náhodné čísla so stredom 0 a varianciou 1:

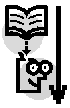
```
double java.lang.Random.nextGaussian()
```

Ak potrebujem generovať gaussovské čísla so strednou hodnotou μ a varianciou σ urobí to jednoduché reškálovanie a posunutie

```
double gauss = sigma*java.lang.Random.nextGaussian()+mu;
```

Ďalšie podrobnosti o triede `java.util.Random` možno nájsť v dokumentácii k Jave.

2.8 Interface, class, factory



V tomto odseku krátko uvedieme zopár pokročilejších trikov z Java programátorskej kuchyne, ktoré budeme potrebovať pre porozumenie použitia niektorých knižníc.

Zopakujme si najprv, čo je to interface. Interface je v podstate formálna špecifikácia určitého protokolu správania sa tried, ktoré tento protokol implementujú. Znamená to toto. Často sa stáva, že je užitočné, aby viaceré, aj navzájom hierarchicky nesúvisiace triedy, obsahovali metódy s rovnakým názvom, rovnakými formálnymi parametrami, ktoré majú podobnú, i keď nie nevyhnutne totožnú funkčnosť. Napríklad v štandardnej knižnici Javy je definovaných asi 60 tried, ktoré slúžia na vstupno výstupné operácie všemožného druhu a všetky potrebujú implementovať metódu, ktorá sa volá `close()`. Jej účelom je ukončiť prepojenie danej aplikácie a vstupno-výstupného zariadenia, odomknúť uzamknuté súbory, aby na ne mohli pristupovať iné aplikácie, niekedy uvoľniť pamäť obsadenú buffermi, proste všakové operácie spojené s "ukončením služby, zatvorením biznisu". Je veľmi rozumné, aby sa operácia tohto typu naozaj jednotne volala vo všetkých triedach. Nebolo by dobre, keby metódu tohto typu nazval implementátor jednej triedy napríklad `close()` a iný implementátor inej triedy `ShutDown()`. Je lepšie, keď sa všetci dohodnú na spoločnej konvencii, na spoločnom protokole. Formálnym vyjadrením takého protokolu je interface, v našom prípade tvorcovia Javy definovali interface `java.io.Closeable`. Tu je definícia (objavila sa až vo verzii Java 1.5, v dokumentácii a zdrojákoch nižších verzií by ste to márne hľadali.)

Listing 2.6: Closeable.java

```
1  /*
2   * @(#)Closeable.java    1.4 03/12/19
3   *
4   * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
5   * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
6   */
7
8  package java.io;
9
10 import java.io.IOException;
11
12 /**
13  * A <tt>Closeable</tt> is a source or destination of data that can be closed.
14  * The close method is invoked to release resources that the object is
15  * holding (such as open files).
16  *
17  * @version 1.4 03/12/19
18  * @since 1.5
19  */
20
21 public interface Closeable {
22
23     /**
24      * Closes this stream and releases any system resources associated
25      * with it. If the stream is already closed then invoking this
26      * method has no effect.
27      *
```

```
28        * @throws IOException if an I/O error occurs
29        */
30       public void close() throws IOException;
31
32   }
```

Všimnime si, že metóda `close()` je tu len deklarovaná, ale nie definovaná. Je to vlastne abstraktná metóda, nemá žiadnu funkčnosť. Celý protokol vlastne obsahuje jedinú vec. Hovorí, že každá trieda, ktorá o sebe bude tvrdiť, že implementuje interface Closeable musí implementovať metódu deklarovanú ako

```
public void close() throws IOException
```

Poznamenajme že v skutočnosti kompilátor netrvá na tom, aby v deklarácii bola aj špecifikácia `throws IOException`, ale z viacerých dôvodov nie je dobrý nápad nedodržať to.

Urobme si malý príklad takej triedy. Trieda nebude mať žiaden normálny zmysel, nemá nič spoločné s IO operáciami, bude to len formálny príklad.(Ospravedlňujem sa: tu treba verziu Java 1.5 aby sa to dalo skompilovať, v ostatných príkladoch stačí verzia Java 1.4)

Listing 2.7: DemoCloseable.java

```
1   import java.io.IOException;
2   import java.io.Closeable;//len java 1.5 a vyssie verzie!!!
3   public class DemoCloseable implements Closeable {
4     public DemoCloseable() {
5     }
6     public void close() throws IOException{
7       System.out.println("closing");
8     }
9     public static void main(String[] args) throws IOException {
10      // main sluzi len na ukazky pouzitia
11      DemoCloseable demo = new DemoCloseable();
12      demo.close();
13      Closeable demo1 = new DemoCloseable();
14      demo1.close();
15    }
16  }
```

Všimnime si v listingu, že trieda `DemoCloseable` sa dá používať dvoma spôsobmi. Prvý spôsob je naznačený v riadku 11, keď vytvoríme objekt (inštanciu) triedy `DemoCloseable` a potom použijeme jej metódu `close()`. Druhý spôsob je rafinovanejší: v riadku 13 deklarujeme objekt demo1 ako inštanciu nie triedy (class) ale interfejsu (interface) `Closeable` hoci na pravej strane priraďovacieho príkazu potom vytvoríme inštanciu triedy `DemoCloseable`. Takáto konštrukcia pripomína podobný trik, ktorý sa používa pri dedičnosti tried, keď vytvoríme objekt nejakej triedy, ale používame ho iba ako by to bol objekt nadradenej triedy (Teda z objektu "this" využívame iba tie metódy, ktoré sú deklarované i pre jeho "super". Uvedomme si ale, že rozširujúca trieda môže niektoré metódy predefinovať, môže teda ísť o polymorfizmus!).

Java teda pripúšťa, že sa na objekt, z ktorého chceme využívať iba metódy, ktoré sú deklarované v nejakom interface pozerať akoby na objekt toho interface. Pritom to ale musí byť objekt nejakej triedy, ktorá daný interface implementuje a jeho deklarované metódy definuje.

Často sa ale pre vytváranie objektov triedy, ktorej hlavnou úlohou je implementovať nejaký interfejs používa iná programovacia technika, takzvaný **factory pattern**.

Uveďme najprv krátku ukážku takej techniky, preprogramujme vyššie uvedený listing využijúc factory pattern. (Ospravedlňujem sa: tu treba verziu Java 1.5 aby sa to dalo skompilovať, v ostatných príkladoch stačí verzia Java 1.4)

Listing 2.8: DemoFactory.java

```
1  import java.io.IOException;
2  import java.io.Closeable;//len java 1.5 a vyssie verzie!!!
3  public class DemoFactory implements Closeable{
4    private DemoFactory() {
5    }
6    /**
7     * Toto je factory pre vytvorenie objektu typu Closeable
8     * @return Closeable
9     */
10   public static Closeable createCloseable(){
11     //privatne volanie constructora, pre pouzivatela ostane skryte
12     //pouzivatel vlastne nevie objekt akej triedy sa mu vrati ako
13     //Closeable
14     DemoFactory df = new DemoFactory();
15     return df;
16   }
17   public void close() throws IOException{
18     System.out.println("closing");
19   }
20
21   public static void main(String[] args) throws IOException{
22     Closeable c = DemoFactory.createCloseable();
23     c.close();
24   }
25 }
```

Všimnime si dve podstatné zmeny oproti predchádzajúcemu listingu.

Po prvé pribudla metóda `Closeable createClosable()`, ktorá vytvorí a vráti objekt typu Closeable. Takejto metóde sa hovorí v programátorskom žargóne factory metóda. Preto factory, lebo vyrába objekt. (Obvykle objekt "vyrábame" pomocou constructora pomocou operátora `new`. Factory je špeciálny "výrobný prostriedok".)

Druhou zmenou je to, že constructor sme deklarovali ako `private`. To nie je síce nevyhnutné, ale je to účelné. Zabránime tak užívateľovi, aby použil pre výrobu objektu constructor a prinútime ho použiť ponúkanú factory metódu. Donútime tak užívateľa aby s objektom našej triedy pracoval iba ako s objektom interfejsu, ktorý naša trieda implementuje. Pre užívateľa ostane úplne skryté, aká trieda vlastne implementuje objekt interface, ktorý používa. Je to dôležité najmä vtedy, ak pri písaní kódu pre užívateľa ešte ani nevieme, aká trieda bude užívateľom používané objekty implementovať. Zdá sa to prehnané, ale toto je napríklad cesta, ako zariadiť prenositeľnosť programovej štruktúry: môžeme potom dosiahnuť, aby pod Linuxom užívateľovi prístupné chovanie zabezpečovala nejaká trieda a pod Windowsami to robila úplne inak koncipovaná trieda.

Technika factory pattern prekračuje úroveň zodpovedajúcu programátorským návykom potrebným pre toto počítačové praktikum. Museli sme ju spomenúť iba preto, že balík JAIDA, ktorý budeme používať, hojne factory techniku využíva a nerozumeli by sme dobre, o čo tam ide.

Technika factory pattern sa používa i pri iných príležitostiach, ktoré tu nebudeme rozoberať. Všeobecne povedané ide tu o vyššiu školu programovania, kde sa už neučia len elementárne konštrukcie toho ktorého jazyka, ale vyučujú sa celé idiómy (patterny). Záujemcu o takú vyššiu programovaciu techniku odkazujeme na knižku J.W.Cooper: The design Patterns, Java companion. Jej pdf verzia je voľne prístupná na internete, vygooglite si ju. A potom najmä na vynikajúcu knihu J.Bloch: Effective Java – Programming language guide. Vyšla aj po česky vo vydavateľstve Grada a je považované za najlepšiu knihu o Jave pre pokročilých.

2.9 Model–View–Controler

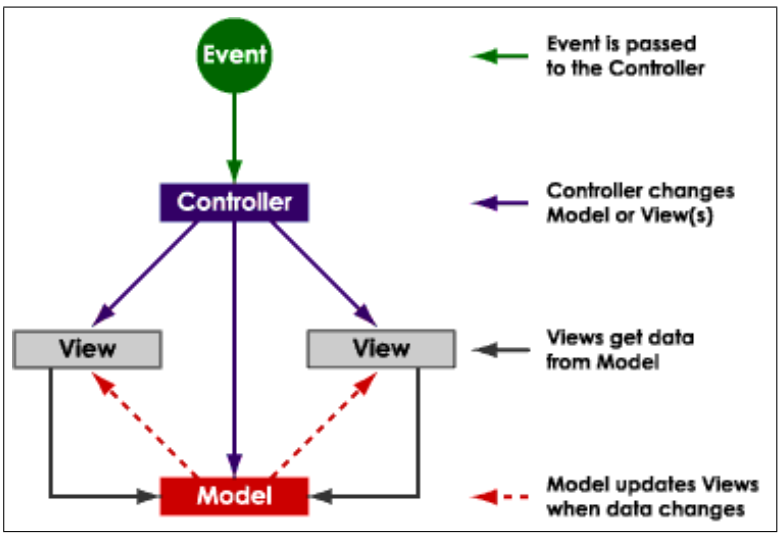
Základnou paradigmou moderného programovania je modulárna štruktúra programu. To, čo patrí logicky funkčne k sebe, vložiť do spoločného modulu a taký modul držať čo možno najoddelenejšie od iných modulov. Moduly majú spolupracovať len dohodnutými dobre dokumentovanými spôsobmi, ktoré tvoria protokol vzájomnej interakcie modulov.

Modulárna výstavba sa má forsírovať na všetkých úrovniach. Na najnižšej úrovni sú to štandardne zgrupované skupiny príkazov (makrá), ktoré často ani nie sú formálne deklarované a predstavujú skôr štandardný rukopis toho ktorého programátora. Na ďalšej úrovni sú to procedúry a dátové štruktúry. Na ďalšej úrovni sú to objekty (triedy). Na ďalšej úrovni sú to štandardizované typy dizajnov objektov a ich vzájomných komunikácií. A na ešte vyššej úrovni makroskopický dizajn celej aplikácie rozdelený do modulov podľa funkčných skupín.

V oblasti spracovania dát sa často stretáme s makroskopickým programovým dizajnom typu MVC (Model–View–Controler). Toto logo označuje istú modulárnu výstavbu aplikačného programu a zvýrazňuje tri typické moduly (obr.2.2).

- **Model** je symbolické označenie pre špecifickú reprezentáciu dát, spracúvaných danou aplikáciou. V objektovo orientovanom programe je to spravidla trieda alebo balík tried, ktoré slúžia na efektívne uloženie, vyhľadávanie, modifikovanie štruktúrovaných dát.
- **View** je označenie pre programový modul, ktorého cieľom je prezentácia dát a vzťahov medzi nimi spravidla vo vizuálnej forme vhodnej pre percepciu používateľom a umožňujúcu potom efektívnu interakciu užívateľa s dátami a aplikáciou, ktorá dáta spracúva. Jednému dátovému modelu môže zodpovedať niekoľko rôznych view-reprezentácií zdôrazňujúcich rôzne výbery z dátovej štruktúry alebo rôzne vzťahy vnútri dátových štruktúr.
- **Controler** je označenie pre modul sprostredkúvajúci interakciu užívateľa s modulmi Model a View. Aktivita vychádza od užívateľa, ktorý sa rozhodne pre akciu (napríklad kliknutím na tlačidlo (button)). Takáto akcia sa nazýva event (udalosť) a hovoríme potom o programe, že je event-driven (riadený udalosťami).

Event-driven programy sa vyznačujú tým, že kľúčové rozhodnutia nie sú dopredu naprogramované, ale realizujú sa až počas behu programu (počas run-time) rozhodnutím riadiacej osoby. Tá sa rozhoduje až na základe nejakých medzivýsledkov, ktoré sú aplikáciou spravidla vizuálne prezentované. Event-driven programy sa koncepčne odlišujú od batch-programov (dávkových programov), ktoré bežia bez zásahu nejakej riadiacej osoby. Všetky alternatívy,



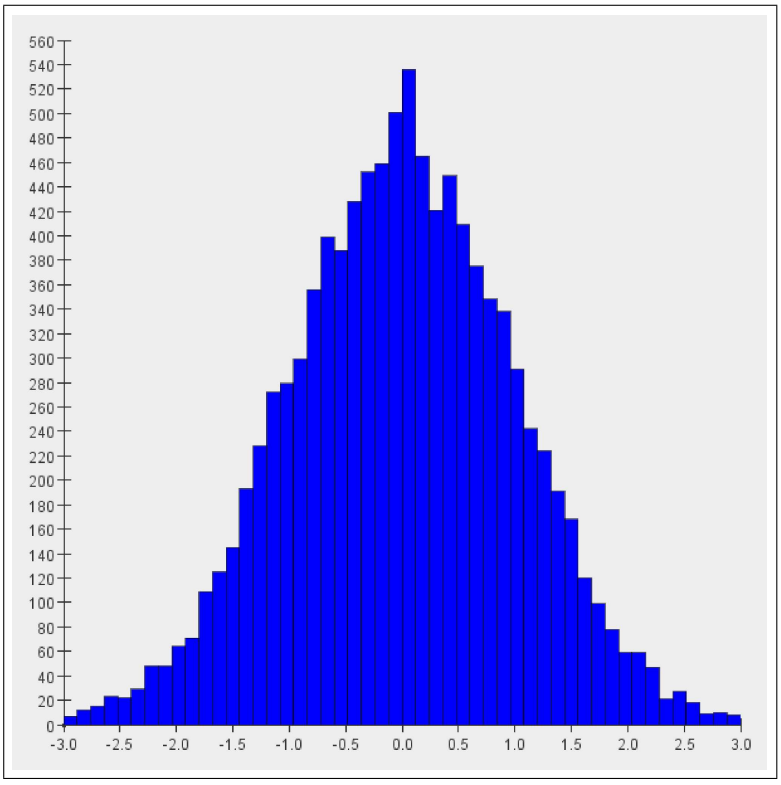
Obrázok 2.2: Model–View–Controler

ktoré môžu počas behu takého programu nastať musia byť teda programátorom dopredu analyzované a "oprogramované".

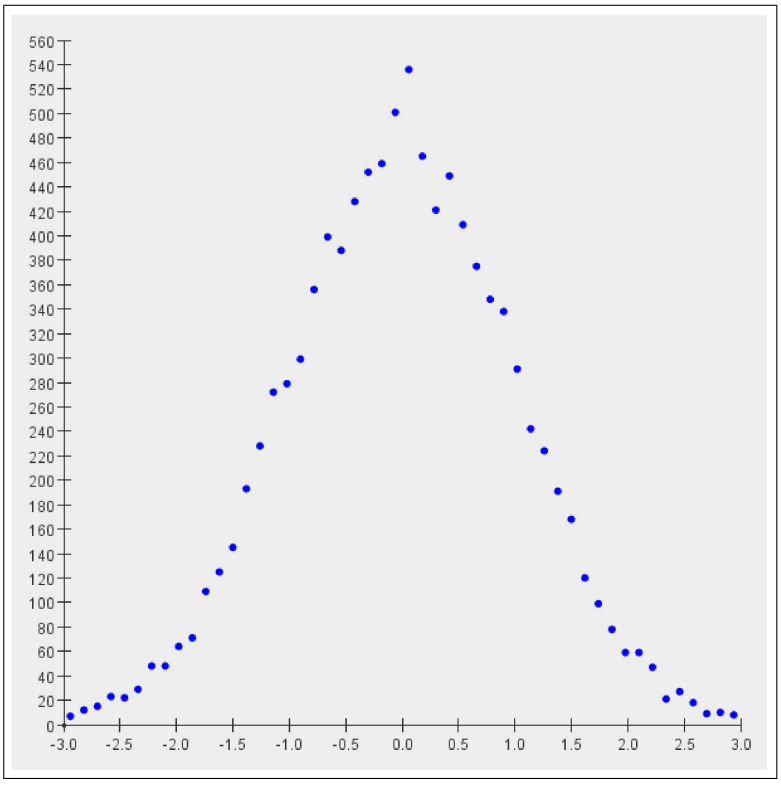
Modulárna výstavba programu pomáha udržiavať poriadok, umožňuje prácu v tíme, uľahčuje identifikáciu chýb v programe, umožňuje opakované použitie menších či väčších úsekov kódu vo viacerých aplikáciách, keď celý modul navrhnutý a odladený pre jednu aplikáciu použijeme pri stavbe inej aplikácie. Al možno ešte dôležitejšie je, že modulárna výstavba pomáha lepšie si zorganizovať vlastnú hlavu.

Malý príklad. Keď fyzik povie "histogram" má tým spravidla na mysli obrázok typu "schodový graf" ako na obr.2.3. Ten histogram sa ale dá zobraziť aj pomocou "dátových bodov", ako je to na obr.2.4.

Všimnime si vetu, ktorú sme práve použili: "Ten histogram sa dá..."To, čo sme chceli vyjadriť, je, že **tie isté dáta** sa dajú nakresliť aj ako graf z "dátových bodov". Slovo "histogram" sme tu použili nie na označenie typu obrázku ale na označenie istým spôsobom štruktúrovaných dát. V myšlienkach sa nám teda prelína dátový model a spôsob vizualizácie. Modulárny prístup MVC pomáha upratať myšlienky, aby sme si ujasnili, kedy hovoríme o type dátovej štruktúry a kedy o spôsobe vizualizácie. Počítačoví fyzici si dnes pod pojmom histogram predstavia príslušnú dátovú štruktúru. K problematike sa vrátime, keď budeme hovoriť o dátových modeloch používaných v balíku JAIDA.



Obrázok 2.3: Ukážka histogramu



Obrázok 2.4: Histogram zobrazený pomocou "dátových bodov"

Kapitola 3

Vizualizácia a spracovanie dát. Úvod do problematiky.

Fyzika od čias Galilea je kvantitatívnou vedou: meraním získava hodnoty (fyzikálnych veličín) charakterizujúce skúmané systémy, získané dáta analyzuje, snaží sa porozumieť vzťahom medzi nimi a nájsť matematické modely, ktoré umožňujú pozorované dáta vysvetliť ale najmä predpovedať výsledky budúcich meraní.

Zber a analýza dát patria teda k základným technikám fyziky. Je samozrejmé, že sa vyvinuli určité štandardné postupy, ako namerané dáta uchovávať, triediť, zobrazovať a analyzovať. Niektoré také postupy si priblížime. Nebudeme sa snažiť o systematický výklad, uvedieme niekoľko typických situácií a základné metódy si ukážeme na príkladoch.

Začneme jednoduchou mierne štylizovanou úlohou. Predstavme si, že chceme preskúmať, či obyvatelia istého regiónu sú obézni. Na vybranej vzorke obyvateľov urobíme merania: každému "respondentovi" zmeriame výšku a váhu. Nebudeme sa zaoberať otázkou ako vyberať dostatočne reprezentatívnu vzorku: okolo toho existuje celá veda. Sústredíme sa len na spracovanie dát.

V súbore obezita.txt sú nasimulované dáta. Je to textový (teda nie binárny) súbor, každý riadok obsahuje dve reálne čísla, výšku "respondenta" v cm a jeho váhu v kg. Každý riadok sa teda týka jedného "prípadu" merania dvojice relevantných veličín na jednom objekte. Akt merania na jednom "záujmovom objekte" sa niekedy nazýva **prípad** alebo **udalosť**, v žargóne sa najčastejšie používa anglické **event**. Jednému eventu je teda v dátovom súbore venovaný jeden riadok, obsahujúci v dvojicu čísel.

Takáto situácia sa pri spracovaní dát vyskytuje veľmi často. Súbor dát predstavujú n-tice čísel, udávajúcich hodnoty nejakých veličín získaných meraním v jednotlivých navzájom nezávislých eventoch. V žargóne namiesto n-tica používame anglické **n-tuple**.

Uveďme tu prvých niekoľko riadkov (teda dáta o niekoľkých eventoch) zo súboru obezita.txt.

Listing 3.1: Úryvok zo súboru obezita.txt

```
174  71
154  47
165  65
182  82
157  71
176  86
166  75
182  78
174  80
173  70
162  76
175  102
153  58
146  49
...
```

Súbor obezita.txt je výsledkom simulácie, nie naozajstného merania. Zdrojový program simulácie je na sprievodnom CD v súbore ObezitaData.java, listing uvádzame aj tu.

Listing 3.2: ObezitaData.java

```
1  import sk.uniba.fmph.pocprak.ioutils.*;
2  import java.util.*;
3  import java.io.*;
4
5  public class ObezitaData {
6    public static double meanHeightMan = 178.;
7    public static double meanHeightWoman = 165.;
8    public static double heightSigma = 10.;
9    public static double shiftFromIdeal = 10.;
10   public static double weightSigma = 10.;
11   public static Random rnd = new Random(1);
12   int h;
13   int w;
14
15   public boolean isMan(){
16     return (0.5<rnd.nextDouble());
17   }
18   public double height(){
19     if(isMan())
20       return (meanHeightMan+heightSigma*rnd.nextGaussian());
21     else
22       return (meanHeightWoman+heightSigma*rnd.nextGaussian());
23   }
24
25   public double idealWeight(double height){
26     return height-100.;
27   }
28   public double weight(double height){
29     return (idealWeight(height)+shiftFromIdeal+weightSigma*rnd.nextGaussian());
30   }
31   public static void main(String[] args) throws Exception{
32     OutTextFile out = OutTextFile.open("obezita.txt");
33     ObezitaData o = new ObezitaData();
34     for(int i = 0;i<1000;i++){
35       o.h = (int)(o.height()+0.5);
36       o.w = (int)(o.weight(o.h)+0.5);
37       out.println(o.h+"␣␣"+o.w);
38     }
39   }
40 }
```

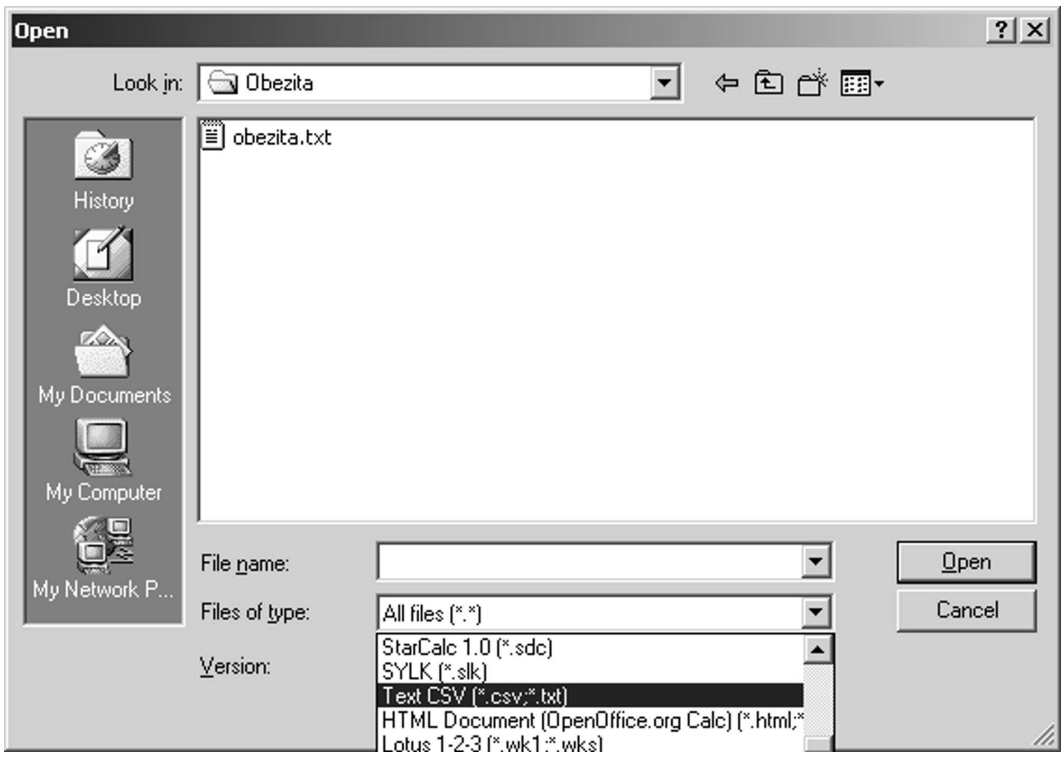
3.1 Tabuľkové kalkulátory

Prv, než sa začneme zaoberať "serióznymi" systémami na spracovanie dát, chceme tu pripomenúť, že narýchlo alebo "v núdzi" je užitočné a často i dostačujúce použiť univerzálne tabuľkové kalkulátory (spreadsheet) ako Excell alebo OpenOffice Calc. Spreadsheety majú v repertoári veľa rôznych štatistických funkcii i dobrých "sprievodcov" (wizard) na vytvorenie jednoduchých grafov.

Excel dnes ovláda každá šikovná sekretárka alebo obchodný agent, nepodceňujte to a naučte sa spracovať a prezentovať dáta pomocou tabuľkového kalkulátora. Zíde sa vám to. Možno v živote nebudete spracovávať dáta o produkcii Higgsových častíc ale bilancie predaja citrónov v posledných troch štvrťrokoch.

Tabuľkové kalkulátory majú pomerne dobrý wizard pre import prostých textových súborov do spreadsheetu. Náš súbor obezita.txt sa preto dá rovno otvoriť v tabuľkovom kalkulátore cez menu File→Open. V ďalšom popíšeme postup v OpenOffice Calc, postup v Microsoft Excell je dosť podobný, ba zdá sa mi že ešte intuitívnejší, takže excellovské detaily tu špeciálne diskutovať nebudeme.

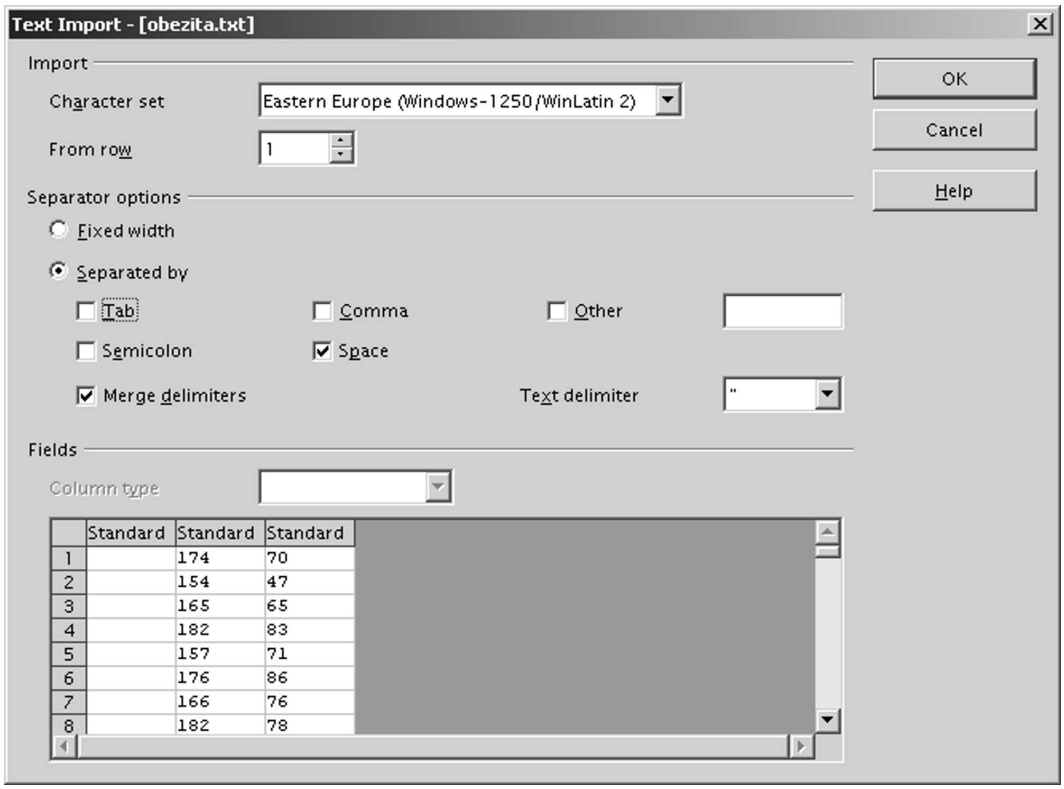
V prvom kroku treba správne určiť typ otváraného súboru, viď obr.3.1



Obrázok 3.1: Otvorenie textového súboru, prvý krok, voľba typu csv,txt

Poznamenajme, že ak by sme v OpenOffice Calc zvolili ako typ súboru prostý txt (teda nie *.csv,*.txt) otvoril by sa textový súbor do word procesora a nie do tabuľkového kalkulátora.

Požiadavka na otvorenie textového súboru vyvolá wizard pre importovanie textového súboru. Wizardu treba oznámiť ako sú jednotlivé stĺpce navzájom oddelené, potom aký typ dát v tomktorom stĺpci je. Treba sa s tým pohrať, po niekoľkých pokusoch je spravidla textový súbor správne importovaný. K štandardným záludnostiam takých importov patrí, že kalkulátor chce mať pri používaní slovenského locale desatinnú čiarku a nie bodku, čísla interpretuje ako text alebo text ako čísla a podobne. Správne začiarknutie informácií pre wizard v prípade súboru obezita.txt je na obr.3.2

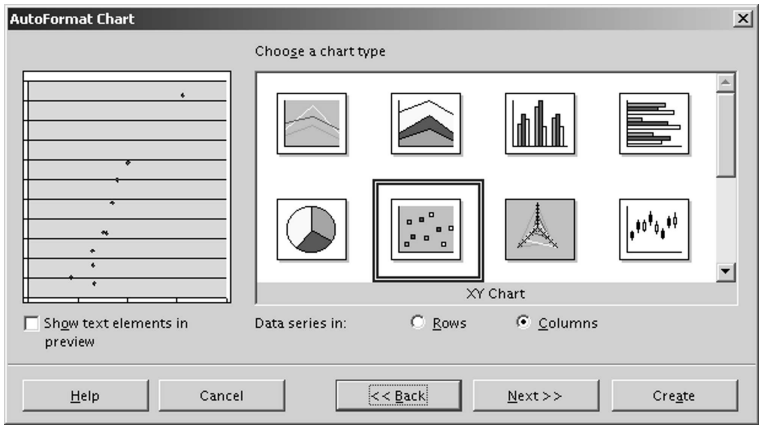


Obrázok 3.2: Otvorenie textového súboru, druhý krok, voľba delimiterov

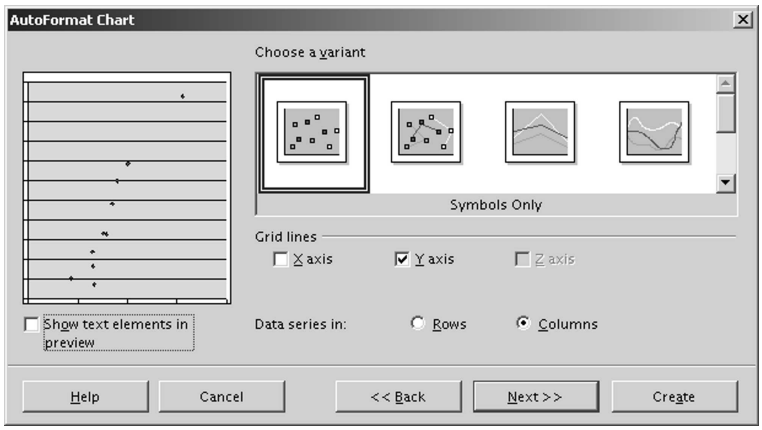
Keď je už súbor načítaný a dáta máme pekne uložené do stĺpcov, je treba sa rozhodnúť, ako tie dáta spracujeme alebo vizualizujeme. Tu si ukážeme vizualizácie pomocou grafu, ktorému vo fyzikálnom žargóne hovoríme scatter-plot.

Scatter-plot je veľmi názorný spôsob vizualizácie pre prípad, že každý event je charakterizovaný dubletom reálnych čísel. Event znázorníme bodkou (alebo iným vhodným symbolom) v rovine v súradnicovom systéme, kde osi zodpovedajú dvom veličinám z dubletu.

V menu zvolíme Insert → Chart. Otvorí sa wizard, ktorému musíme povedať aký typ grafu chceme. Tabuľkové kalkulátory používajú z hľadiska fyzika trocha divnú terminológiu a očakávajú, že budeme chcieť kresliť typické prezentačné (novinárske) koláčové a iné podobné diagramy. Normálny "graf funkcie" neponúkajú príliš okato. Ak chceme bežný scatter-plot, musíme to oznámiť selekciou v dvoch nasledujúcich krokoch Wizardu, zobrazených na obr.3.3 a obr.3.4.

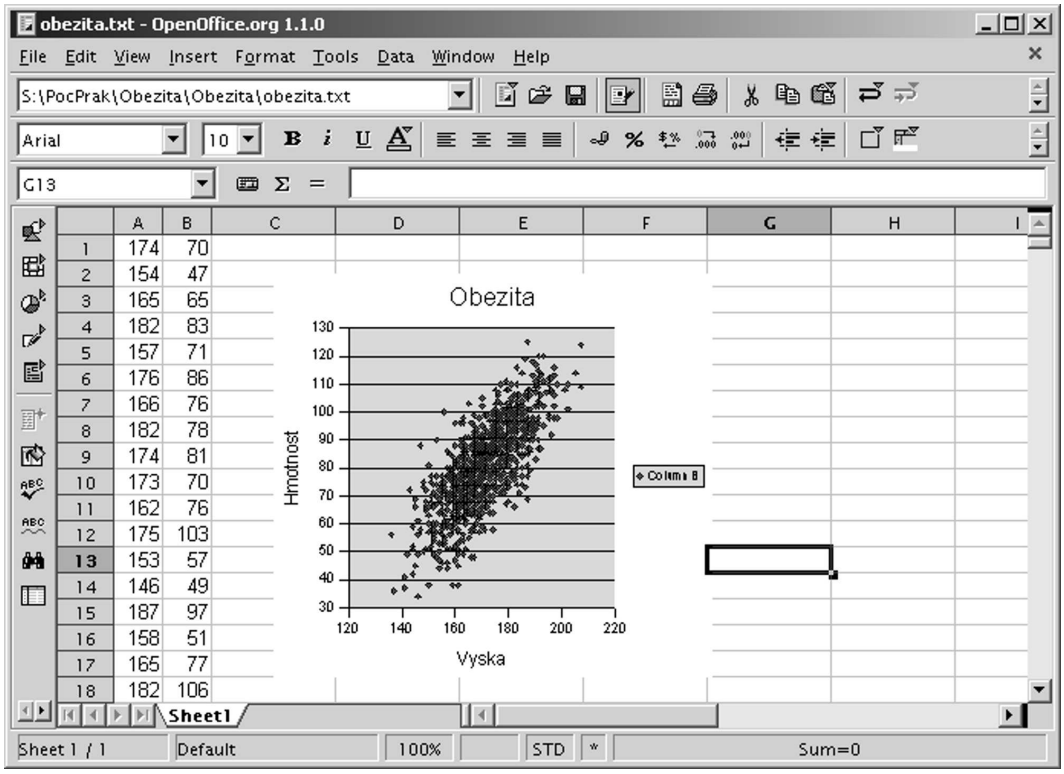


Obrázok 3.3: Voľba typu diagramu, prvý krok



Obrázok 3.4: Voľba typu diagramu, druhý krok

Ďalšie kroky sú pomerne intuitívne, výsledkom je diagram, ako ukazuje obr.3.5.



Obrázok 3.5: Scatter/plot: výška versus hmotnosť

Veľmi odporúčam pohrať sa v tabuľkovom kalkulátore s vytvoreným grafom, skúsiť zmeniť popis osí, pozadie, nakresliť scatter-plot len pre nejakú časť importovaných dát. Objavujte, čo všetko sa s tým dá urobiť: zistíte, že možno napríklad preložiť mračnom bodov priamku. Experimentuje, klikajte pravou a ľavou ťupkou na myši, použite menu, skúste help. Zistíte, že autori tam všeličo zaujímavé nachystali, ale niekedy tvrdohlavo robia, čo oni chcú, nad niečím máte len malú kontrolu, niečo nerozumiete presne, čo sa to tam vlastne urobilo (napríklad čo presne za priamku to tam nakreslilo). Je to často príjemné, keď sa to naučíte trocha ovládať. Pre prípravu prezentácie je to veľmi rýchle, ale pracuje to podľa hesla: "Bill Gates najlepšie vie, čo vy potrebujete". Pre serióznejšiu "vedu" je tam často málo kontroly a porozumenia.

3.2 Všetko si robím sám

Opačným extrémom je nepodľahnúť veľkému Billovi a urobiť si všetko sám. Má to výhodu, že človek rozumie tomu, čo vytvorí (teda dakedy aj nie, keď si len myslí, že naprogramoval to, čo chcel). Má to nevýhodu, že to trvá dlho . A na nejakú estetiku a dokonalosť človek spravidla už nemá trpezlivosť, podľa hesla "Popis osí síce chýba, ale kto by sa s tým babral". Tu je príklad takého amatérskeho programu pre nakreslenie scatter-plotu z dát `obezita.txt`.(Inak toto zase tak pomalé nebolo, mal som to skôr ako v tom tabuľkovom kalkulátore.)

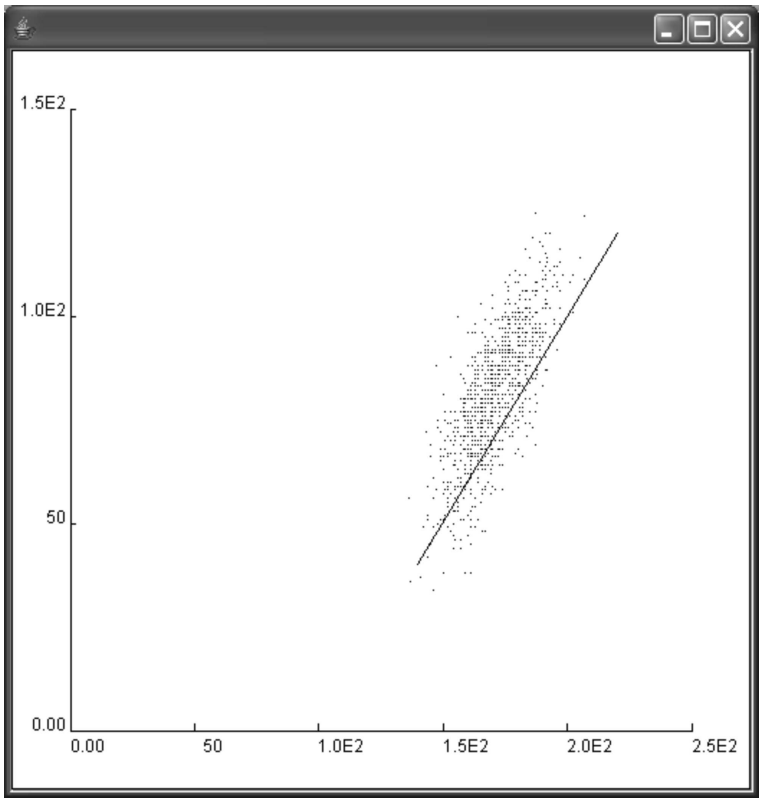
Listing 3.3: Obezitascatterplot.java

```
1  import sk.uniba.fmph.pocprak.ioutils.*;
2  import sk.uniba.fmph.pocprak.simplegraphics.*;
3
4  public class Obezitascatterplot {
5    public double[] height= new double[5000];
6    public double[] weight= new double[5000];
7    public int nevent = 0;
8    public Obezitascatterplot() throws Exception{
9      super();
10     double[] data;
11     InTextFile in = InTextFile.open("obezita.txt");
12     while (true){
13       data = in.readlnDoubles();
14       if(data==null) break;
15       if(data.length==2){
16         height[nevent]=data[0];
17         weight[nevent]=data[1];
18         nevent++;
19       }
20     }
21     in.close();
22   }
23
24   public void MakePlot(){
25     GrGraphics gr=SimpleGraphics.CreateGrEnvironment();
26     gr.setBasePoint(gr.LL);
27     gr.setUserFrameSize(0.,0.,250.,150.);
28     GrAxisX xaxis = new GrAxisX(0.D,250.D,0.D);
29     xaxis.setLTicks(0.0D,50.D);
30     xaxis.draw(gr);
31     GrAxisY yaxis = new GrAxisY(0.D,150.D,0.D);
32     yaxis.setLTicks(0.0D,50.D);
33     yaxis.draw(gr);
34     for(int i = 0;i<nevent;i++){
35       gr.drawPoint(height[i],weight[i]);
36     }
37     gr.repaint();
38     gr.drawLine2D(140.,40.,220.,120.);
39     gr.repaint();
40   }
41
42   public static void main(String[] args) throws Exception{
43     Obezitascatterplot obezitascatterplot = new Obezitascatterplot();
44     obezitascatterplot.MakePlot();
45   }
46 }
```

Tento program vyrobí scatter-plot znázornený na obr.3.6

Výhoda postupu "urob si sám" je v tom, že si (v princípe, ak máme dosť času a húževnatosti) môžeme nakresliť čokoľvek. Ako príklad sme do scatter-plotu prikreslili "priamku ideálnej hmotnosti". Zdravotnícky to korektné nie je, ale ľudová pranostika hovorí: ideálna hmotnosť = výška - 100. No tak sme to tam nakreslili, najmä aby sa ukázala výpovedná schopnosť scatter-plotu. Zjavne vidno, že dáta sú rozložené okolo "ideálnej priamky" nesymetricky, na stranu vyšších hmotností. Keby to celé nebola len vymyslená simulácia, tak by sme práve boli dokázali, že národ je obézny.

Ešte niekoľko slov k programu. Program využíva pre účely týchto praktík pripravený polotovar: balík (package) simplegraphics. Využijeme ho častejšie a bude o ňom podrobnejšia reč. Ale vrelo odporúčame prečítajte si detailne program Obezitascatterplot.java najprv bez dokumentácie k balíčku simplegraphics. Po chvíli lúštenia by vám malo byť približne jasné, čo sa deje v každom z riadkov programu. Takže si môžte sami napísať podobný program bez toho, aby ste v úplnosti rozumeli použitým triedam a metódam. Nezdá sa to ako návod



Obrázok 3.6: Scatter/plot: výška versus hmotnosť

"dôstojný pravého vedca" ale v skutočnosti je programátor často vyslovene odkázaný na lúštenie príkladov použitia softvérových balíkov. Dokumentácia býva slabá alebo nijaká a často nezodpovedá skutočnosti. Balíky robievajú v detailoch niečo celkom iné, ako autori tvrdia v dokumentácii. Ak sú aj v dokumentácii popísané jednotlivé triedy a metódy, menej tam býva povedané ako sa z toho "lega" má postaviť niečo zmysluplné. Modulárny a objektovo orientovaný softvérový balík je naozaj lego. Každá kocka je dobre definovaná. Ale skúste napísať pre (skutočnú hračku) lego abstraktný návod na použitie. Aj hračkári to robia tak, že ukážu niekoľko príkladov, ako sa postaví dom, bager a záhrada so stromami. Zbytok je tvorivá fantázia decka, pokusy a omyly. Nemusí sa vám páčiť, že bankový softvér, ktorý má chrániť vaše peniaze, sa robí metódou pokusov a omylov. Ani sa k niečomu takému nik neprizná. Až hackeri vybielia dajaké konto, potom sa ukáže, že to robilo aj niečo iné, než sa pôvodne predpokladalo.

Takže drobná úloha na experimentálny výskum. Keď sa v programe konštruujú osi, má konštruktor tri parametre, prvé dva zjavne určujú rozsah osi, ale načo je tam ten tretí. Brilantný mozog typu Mycroft Holmes na to príde aj vo foteli, ale Sherlock Holmes zmení ten parameter a pozrie, čo to s vytvoreným obrázkom urobí. Pohrajte sa.

3.3 Profesionálne balíky na spracovanie dát

Vo vedeckých komunitách sa spravidla na spracovanie dát používajú profesionálne pripravené programové balíky alebo celé programové systémy. Balíky sú pripravené tak, že rešpektujú folklór obvyklý v danej komunite. Lebo, hoci ide o **vedecké** spracovanie dát, z časti naozaj ide o folklór. Napríklad niekto obľubuje charakterizovať vzorku nameraných dát ich priemernou hodnotou, niekto mediánom. Niekde obľubujú dvojdimenzionálne box-histogramy, niekde lego-ploty. Niekde dáta najprv filtrujú, potom fitujú, niekde by filtrovanie považovali za znesvetenie dát. Niekde robia Studentov test aj na dátach, ktoré sú očividne negaussovské, ale to už je viac patológia ako folklór[1].

Na začiatok malá ukážka ako nakresliť scatter-plot pre dáta zo súboru `Obezita.txt` pomocou balíka JAIDA. Tu je zdrojový text

Listing 3.4: ObezitaJaida.java

```
1  import hep.aida.*;
2  import java.util.Random;
3  import sk.uniba.fmph.pocprak.ioutils.InTextFile;
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6
7  public class ObezitaJaida {
8    public ICloud2D cl2D;
9    public IPlotter plotter;
10   public ObezitaJaida(){
11     IAnalysisFactory af = IAnalysisFactory.create();
12     IHistogramFactory hf = af.createHistogramFactory(null);
13     cl2D = hf.createCloud2D("Obezita");
14     plotter = af.createPlotterFactory().create("Plot");
15
16   }
17   public void FillCloud() throws Exception {
18     double[] data;
```

[1] Netrápte sa ak nerozumiete žargónu v tomto odstavci. Kým zostarnete, so všetkým sa zoznámite. Teraz ide len o to, vyvolať istý "umelecký dojem" čo mám na mysli pod slovom folklór.

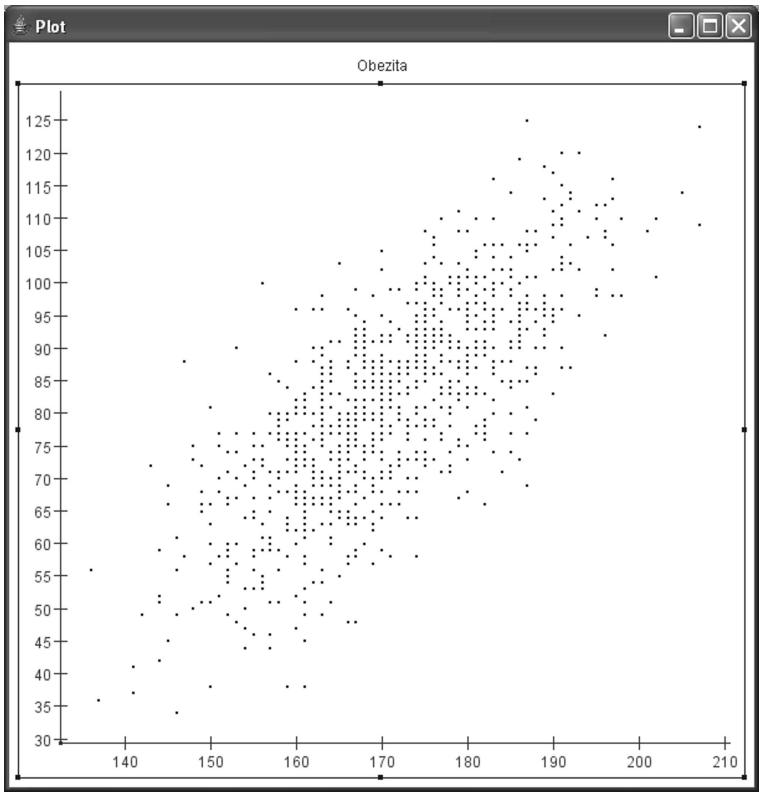
```
19
20      InTextFile in = InTextFile.open("obezita.txt");
21      while (true){
22        data = in.readlnDoubles();
23        if(data==null) break;
24        if(data.length==2){
25          cl2D.fill(data[0],data[1]);
26        }
27      }
28      in.close();
29
30    }
31    public void Plot(){
32      plotter.region(0).plot(cl2D);
33      plotter.show();
34    }
35    public static void main(String[] argv) throws Exception {
36      ObezitaJaida oj = new ObezitaJaida();
37      oj.FillCloud();
38      oj.Plot();
39    }
40  }
```

Nakreslí to scatter-plot uvedený na obr.3.7. Prečítajte si ten zdroják a skúste o každom riadku uhádnuť, čo je jeho úlohou. Niekde sa vám to podarí dobre uhádnuť, niekde budete mať len nejasné tušenie, niekde nebudete ani tušiť. Ale opakovať ako opica môžte aj bez tušenia. Niečo sa neskôr v tomto texte dozviete, niečo si prečítate v programovej dokumentácii, niečo budete stále len tušiť, niečo budete stále len opakovať ako opica. Treba sa s tým zmieriť. Toto je problém každého, kto používa "žijúci" softvér: vývoj ide rýchlejšie, ako dokumentácia stačí sledovať. Výhodou "open source" softvéru je dostupnosť zdrojových kódov. Keby ste naozaj potrebovali s nejakým softom niečo vážne robiť, dokumentácia spravidla nestačí. Musíte zohnať zdrojáky a relevantné časti prelúskať. S prekvapením často zistíte, ako veľmi sa to líši od toho, čo je v dokumentácii.

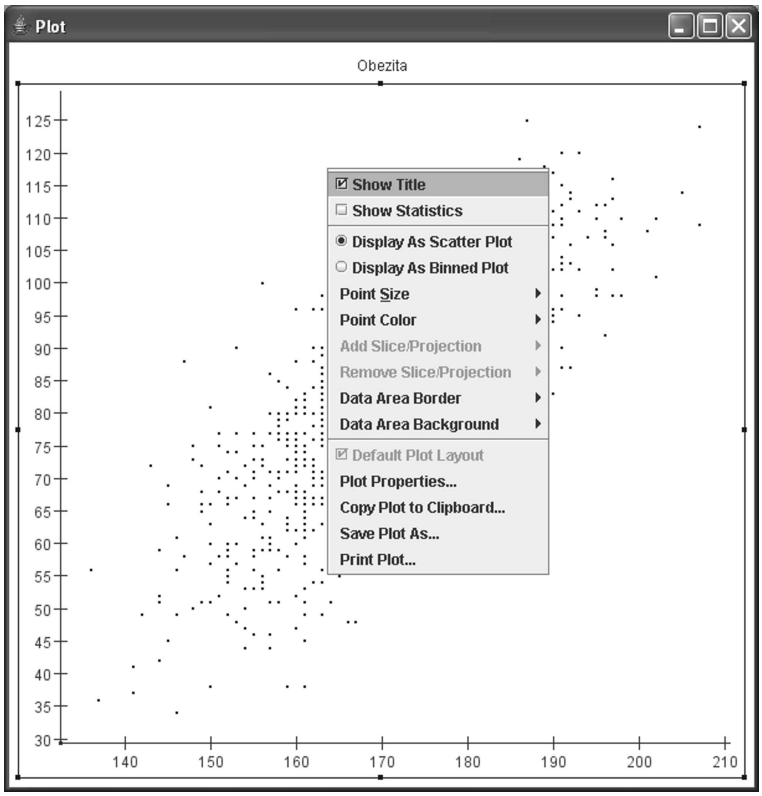
Na druhej strane začiatočník má tendenciu všetko prehlásiť za bug v používanom softvéri. Až tak zlé to zase nebýva. Väčšinou je chyba na strane zákazníka a on sa po krátkom mučení aj prizná. Takže nevzdávajte hľadanie **vašej vlastnej chyby** priskoro. Ono to väčšinou je vaša vlastná chyba.

Programový balík JAIDA je predsa len profesionálny balík a prejavuje sa to aj v tom, že po vykreslení uvedeného grafu môžem ešte všeličo upravovať. Myšací kurzor na ploche grafu a kliknutie pravou myšacou ťupkou znamená vyvolanie pop-up menu, viď obr.3.8. Poexperimentujte s tým menu. Skúste pridať textový popis osí, zmeniť farbu, tvar a veľkosť dátových bodov. Zistite, čo ďalšie sa s tým dá robiť.

V tomto praktiku máme pozitívnu preferenciu pre jazyk Java, preto balík JAIDA. Ten balík ma nejakú históriu a súvisí s inými systémami pre spracovanie dát, ale povieme si to bližšie až v kapitole 5.



Obrázok 3.7: Scatter-plot nakreslený programom ObezitaJaida.java

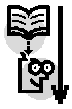


Obrázok 3.8: Pop-up menu vyvolané pravou ťupkou

Kapitola 4

Package simplegraphics

Pre potreby praktika som nachystal malú knižnicu pocprak.jar, obsahujúcu najmä package simplegraphics. Aby sa dala jednoduchá grafika robiť naozaj jednoducho. Môže vzniknúť otázka, prečo. Veď o Jave sa možno dočítať, že grafika v nej sa robí jednoducho. Jedným z motívov pre vznik Javy bola práve potreba grafiky pre web (pozri diskusiu o appletoch v odseku 2.1.



Dôvodov pre napísanie vlastného balíka bolo niekoľko:

- Grafika je jednoduchá v appletoch v aplikáciach je to už zložitejšie
- Tá jednoduchá grafika nevyrába obrázok, iba "kreslí po obrazovke". Vzniká tak problém obrázok uložiť ako grafický súbor prípadne ho vytlačiť
- Obrazovkový súradnicový systém pre grafiku je neprirodzený, jednotkou sú obrazové body, nie fyzikálne koordináty, treba si zautomatizovať prerátavanie jedného na druhé
- Nie sú k dispozícii vyššie grafické objekty typu "súradnicová os"
- Nie je k dispozícii ani najjednoduchšia 3D grafika

Balík simplegraphics je celkom dobre dokumentovaný tak, ako býva pre knižnice Javy zvykom, dokumentácie ja na sprievodnom CD v direktóriu `pocprak/Api`. Ak neviete čítať API dokumentáciu v Jave, pár poznámok o tom je v odseku 5.6. Ak chcete triedy balíka simplegraphics používať musíte mať na ceste classpath nastavenú viditeľnosť archívneho súboru pocprak.jar (pozri odsek 2.3).

Balík simplegraphics môže zdatnejší klient používať ako súbor metód, z ktorých si poskladá svoju aplikáciu. Na najelementárnejšej úrovni si užívateľ nechá pripraviť jednoduché štandardn grafické prostredie a potom pracuje s grafikou veľmi podobne ako v applete, ale má k dispozícii "fyzikálne koordináty" objekty na nakreslenie súradnicových osí v grafe, možnosť zápisu obrázku na disk a jednoduchú (veľmi primitívnu, ale predsa len) 3D grafiku.

Popísaný elementárny prístup sprostredkúva trieda SimpleGraphics. Je to trieda, slúžiaca ako obálka (wrapper) triedy GrGraphics. Vytvorí jednoduché grafické prostredie pričom užívateľ nemusí poznať prácu s prostriedkami swing. Práca v tomto grafickom prostredí využíva objekt triedy GrGraphics, s ktorým sa dajú vykonávať základné grafické operácie veľmi podobne, ako sa to robí pomocou objektu triedy `java.awt.Graphics`. Nasledujúci listing ilustruje použitie tejto triedy.

Listing 4.1: SimpleGraphicsDemo.java

```
1  import sk.uniba.fmph.pocprak.simplegraphics.*;
2  import java.awt.geom.*;
3  import java.awt.*;
4  public class SimpleGraphicsDemo {
5    public static void main(String[] args) throws Exception {
6      GrGraphics gr=SimpleGraphics.CreateGrEnvironment();
7      Graphics2D g = gr.displayG;
8      gr.margin=40;
9      gr.setBasePoint(gr.CENTER);
10     gr.setUserFrameSize(0.,0.,1.,1.);
11     GrAxisX xaxis = new GrAxisX(-1.,1.,0.);
12     xaxis.setLTicks(-1.,0.2);
13     xaxis.draw(gr);
14     GrAxisY yaxis = new GrAxisY(-1.,1.,0.);
15     yaxis.setLTicks(-1.0,0.2);
16     yaxis.draw(gr);
17     gr.drawLine2D(-0.5,-0.3,0.6,0.7);
18     gr.drawPoint(0.4,0.2,3);
19     gr.drawString("Test",0.1,0.2);
20     gr.drawEllipse2D(0.3,0.7,0.6,0.5);
21     g.draw(new Ellipse2D.Double(gr.displayX(0.3),gr.displayY(0.7),
22             gr.userWidthToDisplayWidth(0.5),gr.userHeighToDisplayHeight(0.4)));
23     gr.drawCircle2D(-0.4,0.4,0.3);
24     gr.repaint();
25   }
26 }
```

Komentár k listingu 4.1

- V riadku 6 používame factory `SimpleGraphics.CreateGrEnvironment()` na vytvorenie objektu triedy `GrGraphics` s ktorým sa dajú vykonávať základné grafické operácie veľmi podobne, ako sa to robí pomocou objektu triedy `java.awt.Graphics` Treba však zdôrazniť, že trieda `GrGraphics` nie je dedičom triedy `java.awt.Graphics2D` a teda priamo nepodporuje všetky grafické metódy tejto triedy akoby predefinované do užívateľských súradníc, hoci by to niekto z tohto demo-listingu mohol dedukovať. Ak ale potrebujeme nejakú metódu triedy `java.awt.Graphics2D`, ktorej analóg trieda `GrGraphics` priamo nepodporuje, môžeme tak vždy urobiť, ako je to demonštrované v riadkoch 7 a 21 tohto listingu

- V riadku 7 získavame pomocou verejne prístupnej premennej `GrGraphics.displayG` prístup k objektu `Graphics2D g`, pomocou ktorého môžme podľa potreby robiť triedou `GrGraphics` priamo nepodporované grafické operácie, ako je to ukázané v riadku 21.

- V riadku 8 nastavujeme šírku "bezpečnostného okraja"okolo celého obrázku v jednotkách bodov na displeji (pixel). O tento bezpečnostný okraj je skutočná obrazovková kresliaca plocha väčšia než by zodpovedalo maximálnemu rozsahu užívateľských koordinát. Zobrazia sa tak aj grafické objekty, ktoré mierne pretečú cez rozsah koordinát, čo sa ľahko stáva napríklad pri textových objektoch. Objekty sú teda naozaj obrezané (clip), až keď pretečú aj cez bezpečnostný okraj. V skutočnosti hodnota `gr.margin=40`

je defaultovou hodnotou, ktorú trieda `GrGraphics` poskytuje automaticky. Nastavujeme ju tu zbytočne, ale chceme ukázať, kde je miesto, kde tak treba urobiť, ak chceme nejakú inú hodnotu.

- V riadku 9 nastavujeme referenčný bod do stredu zobrazovacej plochy. Referenčný bod je bod, voči ktorému potom definujeme rozsah užívateľských súradníc na zobrazovacej ploche v riadku 10 Inou alternatívou by bolo použiť príkaz `gr.setBasePoint(gr.LL)`, ktorým by sme položili referenčný bod do ľavého dolného rohu zobrazovacej plochy (do je defaultové nastavenie).

- V riadku 10 definujeme rozsah užívateľských súradníc na zobrazovacej ploche. Prvé dva parametre sú užívateľské súradnice referenčného bodu, druhé dva parametre sú užívateľské súradnice pravého horného rohu zobrazovacej plochy.

- V riadku 11 vytvárame grafický objekt, x-ovú os. Prvé dva parametre určujú rozsah osi (v užívateľských súradniciach), ktorý bude vykreslený, tretí parameter je y-ová súradnica x-ovej osi (x-ová os grafu nemusí prechádzať bodom y=0).

- V 12 riadku nastavujeme pomenované značky[1] na osi (mierku). Prvý parameter udáva x-ovú súradnicu prvej pomenovanej značky zľava, druhý parameter potom hodnotu kroku (vzdialenosti medzi pomenovanými značkami).

- Riadkom 13 sa tá os (virtuálne, pozri komentár k riadku 24) nakreslí.

- V riadkoch 14 až 16 sa vyrobí y-ová os

- Nasleduje demonštrácia niekoľkých grafických príkazov. V riadku 17 sa nakreslí úsečka (parametrami metódy sú užívateľské súradnice počiatočného a koncového bodu. V riadku 18 sa nakreslí "bod" ako malý vyplnený krúžok. Jeho veľkosť (v obrazových bodoch, pixeloch) je daná hodnotou tretieho parametra, prvé dva parametre sú súradnice toho bodu. V riadku 19 sa vypíše reťazec. Ľavý spodný vrchol virtuálneho obdĺžnika, ktorý obopína ten reťazec sa umiestni do bodu, ktorého súradnice udávajú druhý a tretí parameter metódy. V riadku 20 sa nakresli elipsa. Súradnice ľavého horného rohu jej ohraničujúceho obdĺžnika udávajú prvé dva parametre, šírku a výšku toho obdĺžnika potom druhé dva parametre. V riadku 23 sa nakreslí kružnica so zadaným stredom a polomerom

- Riadok 21 ukazuje priame použitie objektu Graphics2D patriaceho kreslenému obrázku. Na virtuálne kreslenie môžme teda použiť štandardné metódy Javy, ktoré, pravda, vyžadujú zadávať geometrické objekty v displejových súradniciach, nie užívateľských. Z riadku 21 je súčasne zrejmé, ako taký prepočet užívateľských súradníc na displejové urobiť.

- Riadok 24 vykreslí vytvorený obraz na displeji. Zdôraznime, že všetky grafické operácie, ktoré sme tu videli, sa robia na "dátovom zdroji" typu BufferedImage, ktorý je potom vizualizovaný na obrazovke na swift komponente JLabel. K tej vizualizácii príde až vyvolaním procedúry repaint(), a to buď programovo, ako sme to urobili v riadku 24 alebo implicitne, ak napríklad myšou prekryjeme zobrazované okno úplne inou aplikáciou a potom to tieniace okno zasa zrušíme, naše vlastné okno sa prekreslí a pre dotknutú komponentu to spravidla vyvolá procedúru repaint(). Výhody tohto prístupu sú dve. Jednak máme v pamäti uložený na kontrolovanom mieste celý obrázok a môžme ho napríklad ľahko zapísať na disk ako obrazový súbor. Druhá vec spadá do oblasti veľmi pokročilého programovania, takže asi mnohí nebudete rozumieť, ale

nevadí. Keďže grafickými operáciami nemanipulujeme s vizuálnymi komponentami, sú tie operácie thread-safe, teda bezpečné pre použitie v paralelnom prostredí viacerých vlákien. Operácia repaint() je tiež v poriadku, lebo je systémovo zariadené, že ju vykoná v odloženom čase event-dispatching thread. Uf! (To je akože koniec nezrozumiteľného textu, v ktorom autor vystavuje na obdiv všetkých päť cudzích slov, ktoré pozná. Ale seriózny záujemca o programovanie v Jave by si mal o tom niečo prečítať napríklad v odseku How To Use Threads z dokumentu "Creating a GUI with JFC/Swing" od Sunu (na sprievodnom CD v direktóriu `JavaBooksAndTutorials\Java_tutorial`.)

Rozšírenie triedy SimpleGraphics na jednoduché 3D zobrazovanie v paralelnom premietaní predstavuje trieda SimpleGraphics3D. Podrobnosti použitia sú uvedené v API dokumentácii ku knižnici pocprak.

[1] Že nerozumiete, čo hovorím!? Hovorím o lejblovaných tikoch na osi. Keď neviete, ako niečo počítačové pomenovať po slovensky, kliknite si na help v slovenskej verzii balíka Microsoft Office (Točí sa to aj na webe, napríklad terminológia o grafoch na stránke `http://office.microsoft.com/sk-sk/assistance/HP051991411051.aspx`.) Vraj keď sa robila lokalizácia do slovenčiny, jazykovedci to strážili veľmi zaťato a nedali súhlas, kým sa všetky lomítka nepremenovali na lomky. Ale ušlo im na jednom mieste slovo pravítko (!), aká to hrôza. Je pravda, že niekedy sa podarí navrhnúť aj pekný slovenský termín, ako napríklad rozhranie namiesto interfejsu, ktorý tu aj ja používam. (Dalo by sa hovoriť aj medziksicht, ale rozhranie je krajšie). V každom prípade na zistenie frekventnosti nejakého slova alebo zvratu v reálnom živote netreba konzultovať databázu jazykovedného ústavu, stačí sa spýtať Googla. A ten hovorí, že Slováci používajú slovo lomítko na web-stránkach desaťkrát častejšie ako slovo lomka. Nemám patričné vzdelanie, aby som sa k tomu mohol fundovane vyjadrovať, celou touto poznámkou sa vlastne zhadzujem a implicitne vystavujem svoju nekultúrnosť, ale ja som Záhorák, takže čo by ste aj iné čakali. Ale vážne: túto poznámku píšem aj preto, aby som predsa len ospravedlnil malú jazykovú starostlivosť tohto textu a používanie neestetického žargónu aj tam, kde to naozaj nie je nutné. Píšem to vo veľkej časovej tiesni.

Kapitola 5

JAIDA

JAIDA je balík utilít implementovaných v jazyku Java pre vizualizáciu a spracovanie dát orientovaný najmä na potreby fyziky vysokých energií (HEP, High Energy Physics), ale je dostatočne všeobecný, takže sa hodí i mimo oblasti HEP.

5.1 Trocha histórie

Tento odsek je napísaný najmä pre tých, ktorí chcú niekedy spracovávať dáta profesionálnejšie.

Vo fyzike sú dáta denný chlebík, ale asi je to naozaj tak, že najväčšie objemy dát vo fyzike majú a spracúvajú fyzici vysokých energií (teda fyzici, zaoberajúci sa elementárnymni časticami a ich zrážkami pri vysokých energiách). Preto v tomto prostredí prirodzene vznikajú veľké softvérové systémy na spracovanie dát. Asi sa to deje i v iných oblastiach fyziky, ale tam to nepoznám, takže tieto praktiká prirodzene uprednostňujú folklór fyziky vysokých energií.

Fyzika vysokých energií začala používať "veľké" počítače medzi prvými a už hodne dávno, preto tam dlho kraľoval FORTRAN a dodnes ešte veľká časť softu beží vo mierne vynovenom FORTRANe. Vedúce postavenie mal CERN a jeho knižnica aplikačných programov dlho určovala štandard vo svete vedeckého počítania. Súčasťou CERN Library boli aj HBOOK a MINUIT, dva piliere klasickej cernovskej slávy. (Dnes CERNu na nesmrteľnosť stačí, že tam vymysleli www, HBOOK a MINUIT sú proti tomu šepleta).

MINUIT je balík minimalizačných procedúr, teda systém pre numerické nachádzanie minima funkcie viac premenných. Je to v skutočnosti pomerne komplikovaný systém rôznych minimalizačných postupov a pomocných utilít. Jeho expertské používanie vyžaduje preštudovať si manuál a potom získať skúsenosti na veľa príkladoch. Našťastie existuje čosi ako defaultový režim používania systémom cvičená opica, ktorý sa navonok prejavuje slovami ako "MINUIT našiel toto minimum". Znamená to, že príslušný používateľ nie celkom vie, čo ten MINU-

IT vlastne vyvádzal a často ani to nie, či MINUIT samotný považuje nájdený výsledok za minimum a už tobôž nie, či je to naozaj minimum. Balík JAIDA obsahuje akúsi Java implementáciu MINUITu, ale priznám sa, že mi nie je jasné, čo je to za implementáciu a koľko je toho z "veľkého MINUITu" naozaj implementované. Spúšťam to v JAIDe ako opica, ale zdá sa mi, že oproti klasickému MINUITu treba starostlivejšie nastaviť štartovací bod, aby tento JAIDA MINUIT skonvergoval.

HBOOK je klasický cernovský balík na prácu s histogramami, vznikol v dobe, keď neexistovala grafika a histogramy sa tlačili na tzv. riadkovej tlačiarni, ktorá vedela tlačiť len ASCII znaky, bola veľká ako písací stôl tie ASCII znaky v nej behali dokola na čomsi, čo sa ponášalo na bicyklovú reťaz. Takže histogramy sa tlačili pomocou stĺpčekov vytvorených veľkým písmenom X. HBOOK zaviedol terminológiu a základnú koncepciu, ktorej stopy vidno dodnes.

Keď sa od diernych štítkov a riadkových tlačiarní prešlo na obrazovkové terminály schopné pracovať nielen v ASCII ale aj v grafickom režime, zrodil sa PAW, čo je skratka pre Physics Analysis Workstation s interaktívnou filozofiou. Už si nebolo treba rozmyslieť tri dni dopredu aký histogram chcem nakresliť. Ľudia, ktorí robili s PAW, získali isté návyky, žargón a štýl práce. Žiaľ, nie všetci autori ďalších softvérov si uvedomili, že sú na svete ľudia, ktorí sa narodili dlho po tom, čo sa zrodil PAW, a nenarodili sa s príslušnými návykmi a žargónom. A že teda nebudú rozumieť manuálom k novým programovým balíkom, ktoré sú písané pre PAW-kovbojov, ktorí sa na staré kolená začali učiť C++.

Takže ak otvoríte manuál a nebudete rozumieť, netrápte sa. Robte príklady z tutoriálov, dumajte nad ich zdrojákmi, dedukujte, po čase vám svitne. Je to niečo, ako keby ste mali rezbársky manuál, kde sa podrobne vysvetľuje, pod akým uhlom sa prikladá dláto k lipovému drevu a pod akým k javorovému, ale zabudnú vám povedať, že účelom je urobiť sochu. A nepredpokladajú, že nerozoznáte, čo je kladivo a čo je dláto. Začnite tak, že sa na niekoho dívate, keď sochá.

Praktiká, ku ktorým toto má byť sprievodný text, majú ambíciu predovšetkým ukázať, ako funguje fyzika. Ale vedľajším produktom by mali byť aj základy remesla, čo je kladivo, čo dláto a čo socha v odbore "spracovanie fyzikálnych dát".

PAW sa ešte stále používa, ale objektové programovanie a C++ priniesli zo sebou inováciu: spracovanie dát preoblečené do objektového myslenia. V CERNe sa zrodil ROOT, veľký systém v C++. Vlastne je to takmer úplný operačný systém, ktorý používa interpretované C++ ako skriptovací jazyk (CINT, je to krásne). ROOT má bohatý grafický user interfejs a k nemu i príslušný API, takže v ňom možno grafické interfejsy programovať.

Nevýhodou ROOTu je platformová závislosť. Primárne je orientovaný na LINUX/UNIX, pod Windowsami potrebuje VisualC++ alebo prostredie CYGWIN. CYGWIN je zdarma, ale nie každý sa pustí do jeho inštalácie. I pod LINUXom je prenositeľnost binárnych distribúcií obmedzená pre prílišnú pestrosť verzií knižníc i kompilátorov, takže spravidla treba inštaláciu skompilovať zo zdrojákov. Nie je to ťažké, ibaže príslušný Makefile je príliš zložitý, kompiluje sa to dlho a človek sa celú dobu bojí, že to na niečom spadne a potom už nič.

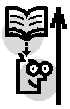
Na záver exkurzie po histórii CERNu ešte dve mená. Autor MINUITu a jeden s počítačových otcov v CERNe je Fred James. Hlavným protagonistom na trase HBOOK→PAW→ROOT je René Brun.

V snahe o univerzálnosť a prenositeľnosť vznikla v komunite fyzikov vysokých energií iniciatíva FreeHEP (`www.freehep.org`) orientovaná na tvorbu a distribúciu voľne šíriteľného softvéru pre potreby fyziky vysokých energií (HEP je High Energy Physics). Iniciatíva podporuje viacero programovacích jazykov a operačných systémov. Jedna z vetiev je zaujímavá pre Javu: FreeHEP Java library (`java.freehep.org`), čo je viacero balíkov užitočného softu implementovaného do Javy. V rámci FreeHEP nás špeciálne zaujíma projekt AIDA (Abstract Interfaces for Data Analysis) Webová stránka `aida.freehep.org`. AIDA zaviedla jednotnú platformovo nezávislú terminológiu i definíciu univerzálnych interfejsov pre všeobecným spôsobom formulované metódy na spracovanie dát. Ide o akýsi abstraktný objektovo orientovaný prístup, ktorý by mal byť v zásade implementovateľný v rámci ľubovoľného objektovo orientovaného jazyka. Praktické jazyky však spravidla takú vysokú úroveň univerzality nedosahujú. V podstate ale platí, že kto sa naučí používať základné interfejsy v nejakej implementácii (napríklad JAVA) potom by ich mal akosi automaticky vedieť používať i v implementácii C++.

Java implementácia systému AIDA sa volá JAIDA. JAIDA využíva knižnice Java FreeHEP Library, ale v distribúcii JAIDA sú potrebné časti knižnice JAVA FreeHEP Library distribuované automaticky. Istú výnimku predstavuje používanie balíku MINUIT v systéme AIDA, ktorý je realizovaný ako volanie natívnych (v C++) napísaných metód. Príslušnú natívnu dynamickú knižnicu treba stiahnuť a inštalovať osobitne. V balíku JAIDA je ale i verzia MINUITU, ktorá sa volá Jminuit a ten nepotrebuje žiadnu natívnu podporu.

Autori systému JAIDA zašli aj ďalej a pripravili čosi ako "integrované grafické prostredie" pre spracovanie dát doplnené o univerzálny "graphical user interface". Systém sa volá JAS – Java Analysis Studio (jas.freehep.org). JAS nie je až taký veľký kombajn ako ROOT, kto chce, môže sa s ním zoznámiť. Pre účely tohto praktika by však detaily práce z JASom boli získavaním príliš špecializovaných zručností.

Naším cieľom nie je vyrábať "hogo-fogo" aplikácie ale kedy-tedy jednoducho nakresliť alebo vyhodnotiť dáta nasimulované v jednotlivých úlohách praktika. JAIDA a jej použitie je pomerne jednoduché a najmä použitie typu "okopírujem a mierne upravím vzorové príklady" je dostatočne transparentné.



5.2 Štruktúra systému AIDA–JAIDA

Ako sme povedali AIDA je obecný protokol interfejsov pre spracovanie dát, JAIDA ie konkrétna implementácia do jazyka Java. V ďalšom však nebudeme striktne rozlišovať, čo je AIDA a čo už je JAIDA a budeme používať spoločný termín JAIDA.

Už sme na inom mieste hovorili o neúplných manuáloch a učení sa pokusmi a omylmi. Text tohto odseku nijako nemá ambíciu byť manuálom k balíku AIDA. Je to pokus povedať pár základných informácií pre začiatočníka v oblasti spracovania dát, aby sa mu zvýšila šanca porozumieť niekoľko vzorových príkladov a prípadne čítať dokumentáciu k AIDA API. API značí Application Programming Interface a je to zase žargón. Myslí sa tým zhruba toto.

Knižničné softvérové balíky (akým je i AIDA) nie sú samostatne funkčné programy. Sú určené na to, aby ich ako externé služby využil programátor pri písaní programu pre nejakú

vyvíjanú aplikáciu. Užívateľ-programátor nepracuje tak, že by excerpoval nejaké časti kódu knižnice prípadne ich modifikoval. Jeho vlastný kód proste volá služby knižnice. V prípade objektového dizajnu je to spravidla cestou vytvorenia potrebných objektov tried definovaných v knižnici a potom volania ich metód. Z hľadiska užívateľa je dôležitá funkčnosť jednotlivých metód, nie to, ako je táto funkčnosť vnútorne dosahovaná. Takej deklarovanej funkčnosti sa hovorí **kontrakt**. Je to naozaj čosi ako zmluva: tvorca knižnice sľubuje, že dodrží funkčnosti, ktoré v kontrakte sľubuje. Užívateľ zasa má využívať služby knižnice len v rámci ponúkaného kontraktu. Súčasťou oného kontraktu je i formálny popis metód a ich parametrov, ktoré pre dosiahnutie funkčnosti má užívateľ využívať. Tieto metódy vlastne tvoria komunikačný kanál medzi užívateľovou aplikáciou a službami poskytovanými knižnicou. Súhrnu prostriedkov (metód a premenných) tohto komunikačného kanála sa hovorí API (Application Programming Interface). Slovo interface v tomto spojení chápeme všeobecne: označuje, že aplikácia a objekty knižnice navzájom interagujú. V balíku JAIDA je tento všeobecne chápaný interface realizovaný pomocou viacerých interface-ov v užšom zmysle. V tomto užšom zmysle ide o prvok "interface" jazyka Java ako sme sa o tom zmieňovali v odseku 2.8.

Poznamenajme ešte, že nedodržanie kontraktu zo strany užívateľa sa volá hack. Je to využívanie softvéru spôsobom, ktorý jeho dizajnér nepredpokladal. Slová hack a hacker nie vždy majú hanlivý význam: často označujú spôsob ako obísť dizajnérovu chybu, keď dizajnér vlastne nedodržal kontrakt a hacker prišiel na to, ako získať sľubovanú funkčnosť novým spôsobom. Niekedy dobrý hack významne obohatí služby softu o nové, dizajnérom nepredpokladané pozitívne funkčnosti. Žiaľ, pri hackovaní niekedy ide o zlovoľné rozšírenie funkčnosti softu mimo rámec kontraktu s cieľom spôsobiť škodu alebo získať nelegálny prospech pre hackera. Dobrý dizajn je preto taký, ktorý nielen dodrží deklarovaný kontrakt ale súčasne zabráni tomu, aby sa vytvorený kód mohol používať mimo deklarovaný kontrakt. Nedobrý dizajn vedie k možnosti používať kód nepredpokladaným spôsobom, čomu sa hovorí "bezpečnostná diera".

Dokumentácia API balíka JAIDA (popis tried a ich metód) je na sprievodnom CD. Na túto dokumentáciu odkazujeme záujemcu o úplnú informáciu o kontraktoch služieb poskytovaných balíkom JAIDA. V tomto texte sa obmedzíme na minimum informácií pre tých, ktorí nechcú vnikať do podrobností a budú používať najjednoduchšie služby metódu drobných modifikácií tutoriálnych príkladov. Aby aspoň trocha rozumeli, čo sa deje.

Zaprisahaní naturisti nemusia čítať ani toto a spoľahnú sa na intuíciu pri používaní technológie copy-paste.

Makroskopická štruktúra balíka JAIDA je typu MVC (Model–View–Controler, pozri odsek 2.9). Definuje niekoľko typov dátových modelov

- Histograms
- Clouds
- Data-point-sets
- Tuples
- Trees
- Functions

V rámci každého z týchto typov je definovaných niekoľko dátových modelov. V tomto texte si všimneme iba štruktúry typu histogram, cloud, data-point-set a function. Záujemcov o ďalšie detaily odkazujeme na dokumentáciu JAIDA, napríklad na stránky AIDA User Guide (off line verzia je na sprievodnom CD).

Modul View zo štruktúry MVC je v balíku JAIDA reprezentovaný interfejsom IPlotter. Modul Controler je pomerne obmedzený. Je implementačne skrytý, nedá sa samostatne využívať (nie je to Java Bean), je prístupný ako pop-up menu (pravé myšacie tlačítko) v rámci vizuálnej prezentácie vyvolanej cez interfejs IPlotter.

Okrem tejto MVC štruktúry ešte balík JAIDA obsahuje dva pomocné moduly

- factories
- fitters

Modul factories slúži na vytváranie "interface-objektov" (pozri diskusiu v odseku 2.8), presnejšie je tam factory na kreovanie factory-interfejsov, ktoré slúžia na kreovanie interfejs-objektov k dátovým štruktúram.

Modul fitters slúži na fitovanie dát (najčastejšie histogramov) zvolenými funkciami. "Fitovanie" je žargónové slovo a značí "optimálnym spôsobom preložiť nejakú krivku danými bodmi". Vzorec krivky má niekoľko voľných parametrov a úlohou "optimálneho preloženia" je nájsť hodnoty tých parametrov. Čo presne značia poetické vyjadrenia "optimálnym spôsobom" a "danými bodmi" závisí od prípadu k prípadu. Po technologickej stránke ide spravidla o numerickú optimalizáciu, v prípade balíka JAIDA tu funguje klasika z CERNu: MINUIT.

5.3 Factories

Programy využívajúce balík JAIDA spravidla začínajú príkazmi, v ktorých sa kreujú potrebné factories a pomocou nich potrebné objekt-interfejsy. V listingu 3.4 sú to riadky 11 až 14

Listing 5.1: kódový snippet z listingu 3.4

```
11      IAnalysisFactory af = IAnalysisFactory.create();
12      IHistogramFactory hf = af.createHistogramFactory(null);
13      cl2D = hf.createCloud2D("Obezita");
14      plotter = af.createPlotterFactory().create("Plot");
```

V riadku 11 je kreovaná hlavná factory `IAnalysisFactory af`. Pomocou nej sa najprv v riadku 12 kreuje factory na kreovanie histogramov `IHistogramFactory hf`. V riadku 13 sa potom kreuje dátová štruktúra `ICloud2D cl2D`. Riadok 14 je o niečo rafinovanejší. Transparentnejšie by bolo rozpísať jeho funkčnosť do dvoch riadkov

Listing 5.2: Úprava riadku 14

```
IPlotterFactory pf = af.createPlotterFactory();
plotter = pf.create("Plot");
```

V pôvodnom riadku 14 je kreovaný objekt typu IPlotterFactory len implicitne a konštrukcia pokračuje bodkou a volaním metódy create.

Nepohodlie celej konštrukcie, teda že najprv musíme kreovať IAnalysisFactory, potom IHistogramFactoty a až potom želanú dátovú štruktúru, teda ICloud2D je daň za to, že chceme mať univerzálne prenositeľný kód. Netreba nad tým príliš dumať, treba proste nejaké také tri riadky systémom copy-paste na začiatok svojho kódu zapísať.

5.4 Dátové modely

5.4.1 Histograms

V tomto odseku chápeme pod histogramom určitý dátový model, povieme si aký. Štruktúru histogram vydumali fyzici, keď bolo treba spracúvať dáta o meraní spojitej náhodnej veličiny (slovo spojitý tu má len obrazný význam ako protipól diskrétnej náhodnej veličiny). Nezáleží na tom, či náhodnosť vzniká v dôsledku náhodných chýb merania alebo veličina samotná má náhodnú povahu. Poeticky povedané, úlohou je vyhodnotiť, ako často "padá" tá ktorá hodnota náhodnej veličiny. Ak tie hodnoty sú čísla zo "spojitej množiny" (kontínua), potom síce pri každom meraní "padne" nejaké konkrétne reálne číslo, ale už sa nikde presne také isté (na nekonečný počet desatinných miest) nezopakuje. Preto nemá zmysel ani otázka typu "Aká je pravdepodobnosť že (nekonečne ostrá) šípka sa zapichne presne do stredového bodu terča. Akákoľvek odpoveď na takú otázku sa totiž nedá experimentálne verifikovať. Dobrý zmysel majú otázky typu "Aká je pravdepodobnosť, že sa šípka zapichne vo vzdialenosti menšej ako 3 cm od stredu terča?". Trik je zrejmý, musíme "priestor dopadov" diskretizovať, rozdeliť na konečný počet diskrétnych chlievikov. Potom už je šanca, že do jedného chlievika padne hodnota veličiny aj viackrát v priebehu pokusu. Presnejšie, potrebujeme, aby do každého chlievika padla hodnota veľakrát, aby sme mohli dostatočne presne stanoviť "pravdepodobnosť dopadu do chlievika".

Chlieviková štruktúra dát, to je histogram.

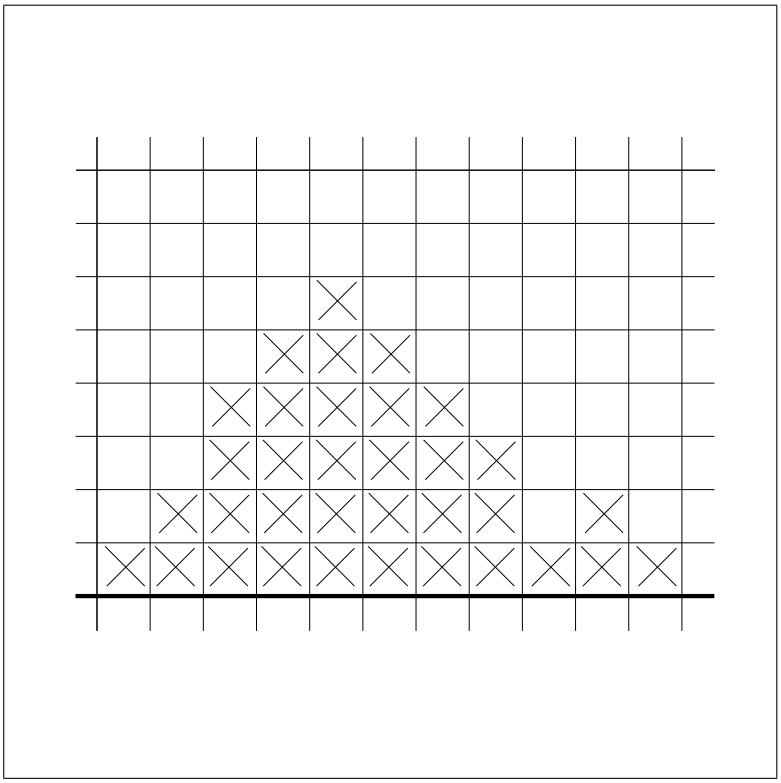
Ak jedna meraná udalosť (event) je daná jediným číslom, potom príslušná štruktúra je jednodimenzionálny histogram. Najčastejšie to robíme tak, že zvolíme na osi príslušnej veličiny nejaký interval (teda zadáme jeho dolnú a hornú hranicu) a ten interval rozdelíme na určitý počet rovnakých[1] dielov - "chlievikov". Tým chlievikom sa v odbornom žargóne hovorí bin. Tak, ako prichádzajú výsledky meraní (prichádzajú eventy) zistíme vždy nameranú hodnotu, určíme bin, do ktorého tá hodnota padne a obsah bunky v pamäti, ktorá reprezentuje onen bin zväčšíme o jednotku[2]. Tejto procedúre sa hovori plnenie (napĺňanie,

[1] V princípe nie je nutné, aby chlieviky mali rovnakú veľkosť. JAIDA umožňuje použiť nerovnomerné delenie na biny, kreovanie histogramu je potom zložitejšie. Záujemcov odkazujeme na AIDA API na sprievodnom disku

[2] Niekedy pracujeme s takzvanými váženými eventami. Býva to v simuláciách v prípadoch, keď nedokážeme presne napodobiť prírodu. Príroda generuje eventy navzájom rovnakej dôležitosti. Stredné hodnoty počítame ako prosté aritmetické priemery, každá nameraná hodnota zaváži v priemere rovnako. Počítač pri simulácii niekedy generuje eventy, ktoré nemajú navzájom rovnakú dôležitosť a reparuje to tým, že spolu s eventom vygeneruje aj jeho relatívnu váhu. Keď počítame v takej simulácii nejakú strednú hodnotu, počítame ju ako vážený priemer. Ak v takom prípade vytvárame histogram, potom do príslušného chlievika pridáme nie hodnotu 1 ale hodnotu váhy eventu. Na konci teda v každom bine bude uložená suma váh eventov, ktorých

filling) histogramu. Po skončení plnenia bude v každom bine uložené číslo, udávajúce počet eventov, ktorých hodnoty padli do daného binu. Pri plnení sa môže stať, že hodnota nepadne do žiadneho binu, pretože je menšia ako dolná hranica alebo väčšia ako horná hranica uvažovaného intervalu. O takom evente hovoríme že podtiekol (underflow) alebo pretiekol (overflow). Dátová štruktúra histogram preto pridáva k užívateľom požadovaným binom ešte dva biny navyše, jeden pod dolnou hranicou a druhý nad hornou hranicou intervalu a do nich ukladá jednotky za každý podtečený resp pretečený event.

Malú poznámku. Videl som starého machra, ktorý sa naučil spracovávať dáta ešte v predpočítačovej ére. A ten si vyrábal dátovú štruktúru histogram na štvorčekovom papieri. Nakreslil si os, interval na nej tak, aby šírka jedného binu bol práve jeden štvorček. Jeden bin bol potom stĺpček štvorčekov. A ten stĺpček postupne zdola zapĺňal tak, že zakrížikoval vždy jeden štvorček v stĺpčeku v tom bine do ktorého padla hodnota eventu, ktorý si práve prečítal. Vznikol tak grafický záznam eventov (čo event, to krížik), ale súčasne aj vizualizácia histogramu ako na obr.5.1 Onen experimentálny guru zjavne nedodržiaval modulárny princíp



Obrázok 5.1: Histogram z dávnoveku

Model-View-Controler, ale histogram mal nakreslený v okamihu. Teda pokiaľ eventov nie je cez stovku.

Z popisu "plnenia histogramu" (aj z tej "krížikovej" reprezentácie) je zrejmé, že uložením dát do histogramu (rozumej do dátovej štruktúry histogram) sa stráca značná časť informácie.

hodnoty padli do daného binu.

O dátových bodoch nevieme v akom poradí prišli, ani aká bola presná hodnota nameranej veličiny: vieme iba, do ktorého binu event padol. Ak po naplnení zbadáme, že sme veľkosti binov nestanovili vyhovujúco, histogram nemôžeme jednoducho "prebinovať". Niečo málo sa urobiť dá, napríklad pozlučovať vždy dva susedné biny do jedného nového, teda zmenšiť počet binov na polovicu. Inak musíme vytvoriť nový prázdny histogram a naplniť ho odznova čítajúc dáta v originále.

Jednodimenzionálny histogram je v balíku JAIDA realizovaný pomocou interface

```
IHistogram1D
```

Príklady najjednoduchšieho kreovania a naplnenia histogramu uvedieme v odseku 5.7.

Treba ešte povedať, že v balíku JAIDA sa počas plnenia (filling) histogramu automaticky vykonávajú nejaké priebežné sumácie, takže po naplnení histogramu sú k dispozícii elementárne štatistiky. Slovo štatistika je pojem z matematickej štatistiky a označujú sa ním akýmsi algoritmom z dát vypočítané čísla. V našom prípade sa nemusíme tváriť príliš vedecky, to čo sa počíta, sú proste štandardné stredné hodnoty. Teda prostý aritmetický priemer (mean), stredná kvadratická odchýlka (RMS - Root Mean Square) teda odmocnina zo stredného kvadrátu. Okrem toho sa vypočítajú štandardné odchýlky (štandardné chyby, errors) každého binu. Spomíname to preto, lebo pri defaultovej vizualizácii sa tieto údaje objavia v rámčekoch v rohu zobrazeného histogramu s titulkom "statistics". Tieto štatistické údaje sú programátorsky k dispozícii v rámci JAIDA API

Stretneme sa ešte aj s použitím dvojdimenzionálneho histogramu. To vtedy, keď event je charakterizovaný dvoma hodnotami nejakých dvoch meraných veličín. Dáta teda prichádzajú ako dvojice (podobne ako sme to videli v príklade o obezite v listingu 3.1). Poznamenajme, že pre vizualizáciu dvojdimenzionálnych dát typu scatter-plot nie je nutná diskretizácia (binovanie). Ale často sa robí: vtedy sa vytvára dátová štruktúra dvojdimenzionálny histogram. Vytvára sa tak, že sa zvolia vhodné intervaly v oboch premenných (na osi x aj na osi y) a oba tieto intervaly sa rozdelia na určitý počet binov. Tým v rovine vzniknú obdĺžnikové biny, do ktorých "dopadajú" jednotlivé eventy. Pri plnení dvojdimenzionálneho histogramu sa do príslušných binov ukladajú jednotky (alebo váhy pri generovaní vážených eventov).

Dvojdimenzionálny histogram je v balíku JAIDA realizovaný pomocou interface

```
IHistogram2D
```

5.4.2 Clouds

Cloud je v AIDA termín pre dátovú štruktúru, ktorá obsahuje nebinovanú (nediskretizovanú) množinu dát o eventoch. Dá sa to predstaviť tak, že eventy sú uložené tak ako prichádzajú bez straty informácie. Cloud namiesto histogramu použijeme vtedy ako potrebujeme dynamicky rebinovateľné histogramy. Zo štruktúry cloud sa totiž vždy dá vygenerovať príslušná štruktúra typu histogram pre ľubovoľne zvolenú definíciu binov.

Niekedy sú objemy dát veľmi veľké, a to znamená že aj štruktúry typu cloud môžu zaberať priveľké miesto v pamäti. Preto balík JAIDA automaticky skonvertuje vnútornú reprezentáciu štruktúry cloud do implicitnej štruktúry typu histogram (teda automaticky binuje dáta) po prekročení určitej defaultovej hodnoty rozsahu dát. Štruktúra navonok síce vystupuje ako cloud ale vnútorne je to histogram sa všetkými dôsledkami, teda stratou schopnosti dynamického rebinovania. Ak chceme takejto autokonverzii zabrániť, musíme zrušiť defaultové chovanie. Podrobnosti nájdete v dokumentácii k API.

Dvojdimenzionálnu štruktúru cloud používame i v prípade, že chceme použiť vizualizáciu typu scatter-plot.

Štruktúry typu clouds v balíku JAIDA realizované pomocou interface

```
ICloud1D
ICloud2D
```

Príklad použitia sme videli v listingu 3.4 v odseku 3.3.

5.4.3 Data-point-set

Dátová štruktúra s prístupom

```
interface IDataPointSet
```

je určená pre uloženie "dátových bodov". Ide, podobne ako u štruktúr typu cloud, o n-tice hodnôt nejakých veličín. Rozdiel je v tom, že DataPointSet sa používa nie na ukladanie hodnôt veličín charakterizujúce jednotlivé eventy ale na ukladanie už "spracovaných výsledkov", teda experimentálnych výsledkov spolu s údajmi o chybách merania (niekto tu uprednostňuje slovenský termín neistota, JAIDA používa anglický termín error).

Ak si predstavíme klasickú praktikovú úlohu "meranie priemeru valčeka mikrometrom" a valček meriame 10-krát, potom dostaneme 10 eventov, každý z nich udávajúci jediné číslo: priemer valčeka získaný jedným priložením mikrometra. Tých 10 eventov potom klasicky spracujeme, získame strednú hodnotu a strednú kvadratickú odchýlku. Toto predstavuje jeden jednodimenzionálny dátový bod. Tento dátový bod je ale reprezentovaný dvoma "jednoticami": jednoticou hodnoty a jednoticou "chyby", teda strednej kvadratickej odchýlky. Ak zmeriame 8 valčekov, každý z nich 10-krát, potom dostaneme 10 dátových bodov, každý z nich charakterizujúci iný valček. Stručne povedané, ak máme uložiť množinu n-tíc hodnôt veličín, použijeme štruktúru cloud. Ak máme množinu párov n-tíc, kde prvá n-tica z páru sú hodnoty veličín a druhá n-tica z páru sú príslušné chyby tých hodnôt z prvej n-tice, použijeme štruktúra DataPointSet.

Príklad použitia je v odseku 5.7

5.5 Plotter

Ploter je ralizovaný ako interface

```
IPlotter
```

Všeobecne sa dá charakterizovať tak, že je to objekt ktorý poskytuje metódy, do ktorých sa dá dosadiť dátová štruktúra (objekt reprezentujúci dátový model) a vyrobí vizualizáciu tohto objektu. Podrobnosti vizualizácie (farby, hrúbky čiar, texty a podobne) sa dajú nastaviť buď pri kreovaní objektu plotter vo fabrike

```
IPlotterFactory
```

alebo až dodatočne volaním rôznych "set" metód[3] objektu IPlotter. Plotter sa dá používať dvoma spôsobmi

- v užívateľom vytvorenom okne, teda ako vložený (embedded) do swing komponenty napr. JPanel
- v automaticky generovanom vlastnom okne, o ktoré sa nemusí starať užívateľ (takže užívateľ nemusí ovládať GUI programovanie - swing)

IPlotter spája v sebe (v terminológii MVC) úlohy modulu View aj modulu Controler. Controler je realizovaný formou polohovo citlivého pop-up menu. Znamená to, že ak sa myšací kurzor nachádza v okne plottera a podržíme pravé tlačidlo, objaví sa na obrazovke menu pomocou ktorého môžme interaktívne meniť v značnom rozsahu vzhľad vizualizácie zobrazovanej plotterom. Obsah menu závisí (veľmi jemne!) na polohe kurzora: iné menu sa spustí. keď je kurzor na okraji plochy, iné keď ukazuje na zobrazovanú os, iné keď ukazuje na čiaru histogramu. Príklad pop-up menu sme už videli na obr 3.8

Osobitne upozornime, že jednou z položiek v pop-up menu je **Save as**. Umožňuje uložiť do grafického súboru prezentovaný obrázok. Budeme to potrebovať pre napísanie protokolu z praktických cvičení. Pop-up menu Save as neponúka však veľa možností ako uložiť obrázok, praktickejšie možnosti voľby sú k dispozícii priamo v kóde aplikácie volaním metódy

```
IPlotter.writeToFile(String filename) throws IOException
```

Niekoľko príkladov je v listingu 5.4

IPlotter umožňuje jedno okno (presnejšie jeden panel) rozdeliť na viacero zobrazovacích plôch, v JAIDA terminológii sa tie plochy nazývajú regions. Na každej ploche môžeme potom

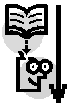
[3]Toto je štandardné názvoslovie objektového programovania. Objekty mávajú nejaké parametre, ktoré špecifikujú ich chovanie. Hodnoty týchto parametrov sa spravidla nenastavujú nejakým priraďovacím príkazom "=" ale volaním osobitnej metódy, ktorá máva v názve slovíčko "set", aby sa naznačilo, že úlohou tej metódy je niečo nastaviť. Takáto metóda sa nazýva "setter". Naopak, existujú metódy, ktoré po zavolaní vrátia hodnotu, ako je dopytovaný parameter nastavený. Takáto metóda má spravidla v názve slovíčko "get" a nazýva sa getter. V balíku JAIDA gettre nemajú slovíčko "get" ale "Value", čo je trocha mimo dobré zvyklosti. Dobre napísané objekty mávajú i metódy, ktoré na požiadanie vrátia aj možné hodnoty, ktoré môže ten-ktorý parameter nadobúdať. V balíku JAIDA sa poznajú tieto metódy podľa slovíčka "available".

zobraziť inú dátovú štruktúru prípadne inú vizualizáciu tej istej štruktúry. Rozdelenie na zobrazovacie plochy sa dá urobiť viacerými spôsobmi, jednoducho napríklad volaním metódy

```
IPlotter.createReginos(int columns,int rows) throws IllegalArgumentException
```

Volanie tejto metódy rozdelí celú plochu na "šachovnicu" so zvoleným počtom stĺpcov a riadkov. Každé políčko tejto "šachovnice" je jednou zobrazovacou plochou. Políčka sú indexované priebežne po riadkoch zľava doprava a zhora dolu, počínajúc indexom 0 označujúcim políčko v ľavom hornom rohu. Uvidíme na príkladoch ako sa to používa.

5.6 Dokumentácia AIDA API



V predchádzajúcom odseku sme krátko popísali jednu z metód interface IPlotter. Naučte sa hľadať si takúto informáciu v dokumentácii, v tomto prípade dokumentácii k AIDA API. Jazyk Java má dokumentačnú technológiu štandardizovanú priamo v štruktúre jazyka, preto dokumentácie ku všetkým softvérovým balíkom vyzerajú rovnako.

Dokumentácia k AIDA API je na sprievodnom CD v direktóriu `Jaida\aida_api`. Jeho obsah je ukázaný na obr.5.2. Možno, že ďalší výklad je dnes už zbytočný, ale pre prípad, že niekto nemá vrodené reflexy používania html-súborov, predsa len pár slov. V direktóriu je zrejme štruktúra html-súborov. To sú súbory, ktoré sa obvykle vyskytujú ako http-adresa (url) na webovských serveroch. Súbory boli naozaj stiahnuté zo servera projektu AIDA. Dajú sa prezerať webovským prehliadačom. Súbor, kde treba začať, je (to je štandardná dohoda pre webovské stránky) `index.html`. Poklikanie na tento súbor naštartuje webovský prehliadač a zobrazí sa stránka typická pre Java dokumentáciu, viď obr.5.3. Na ľavom okraji je zoznam všetkých tried a interfejsov z balíka AIDA. Metódu `IPlotter.createRegions`, o ktorú sme sa zaujímali v predchádzajúcom odstavci nájdeme tak, že v zozname tried nájdeme `IPlotter`, klikneme na tú referenciu a objaví sa nám dokumentácia k `interface IPlotter`. Tam sa už doscrolujeme k popisu kontraktu metódy `IPlotter.createRegions`, viď obr.5.4. Poznamenajme, že namiesto lokálnej kópie dokumentácie k AIDA API môžme ju vyhľadať priamo na web-serveri na adrese

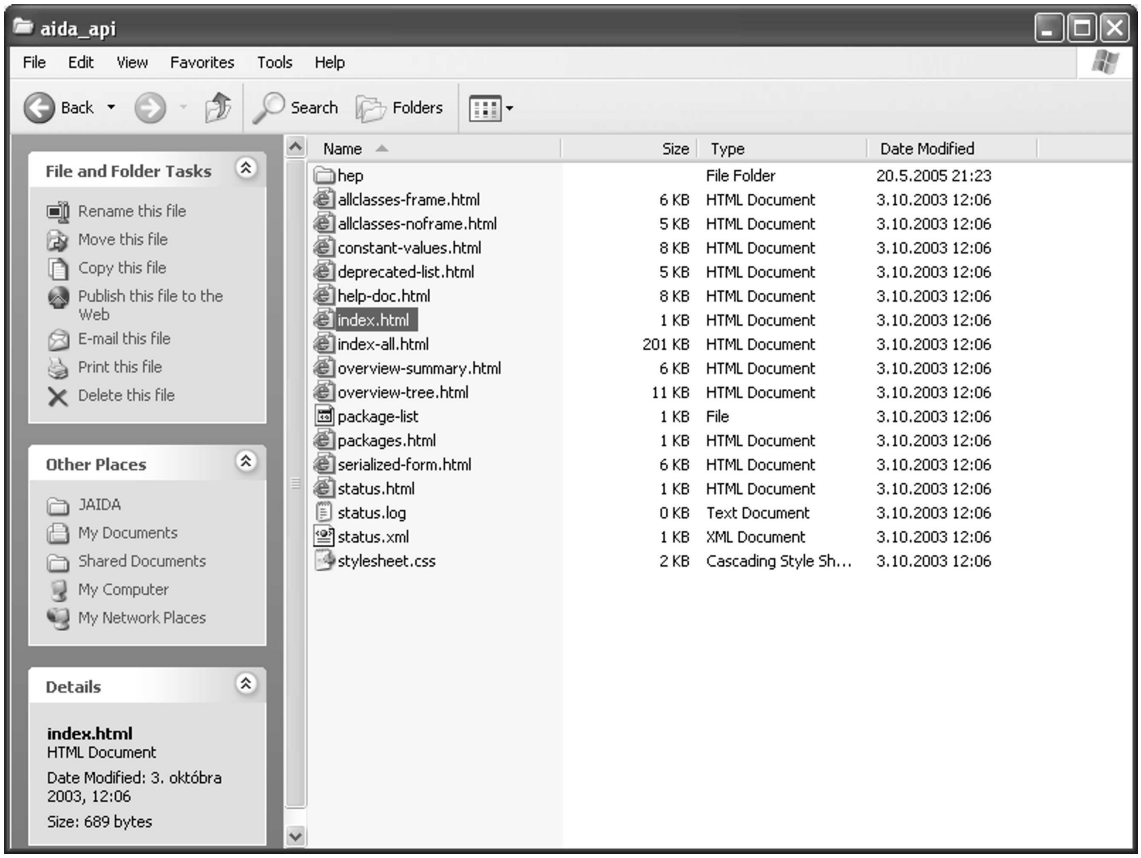
```
http://aida.freehep.org/doc/v3.2.1/api/index.html
```

5.7 Príklady

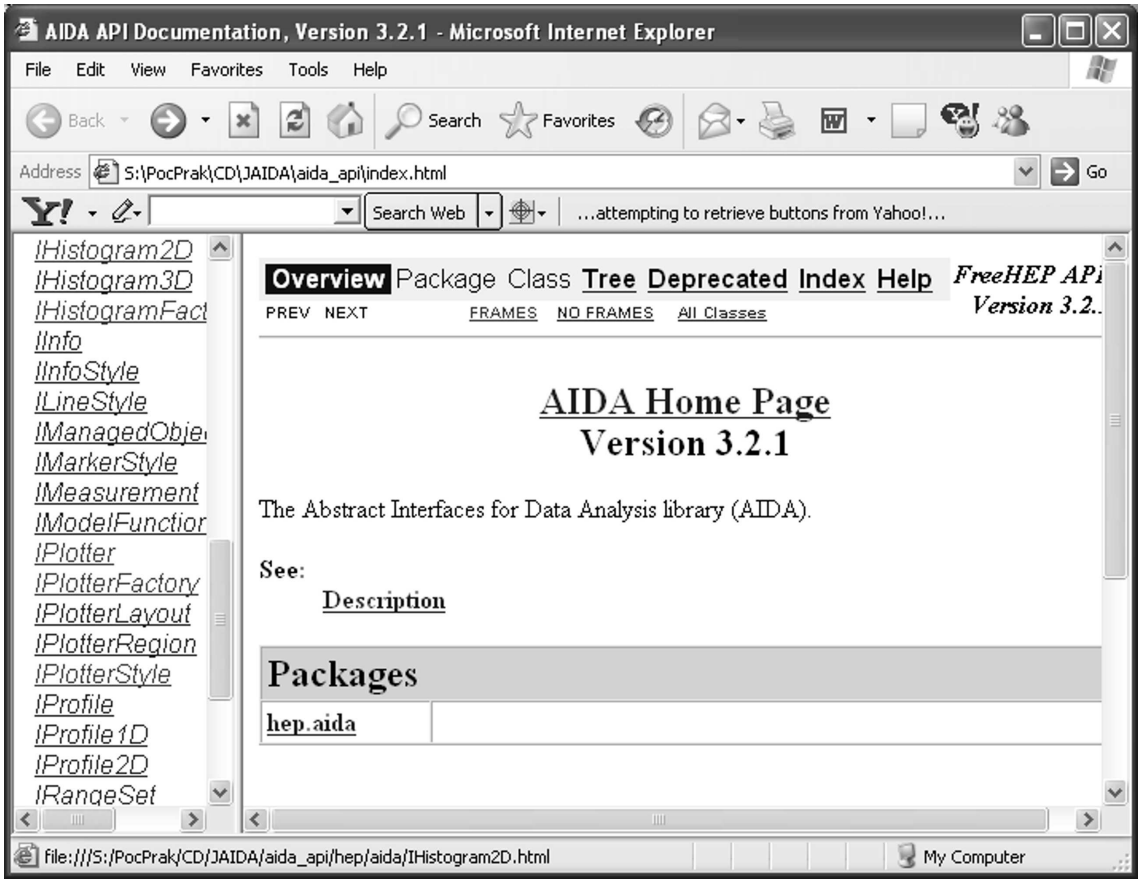
Teraz niekoľko príkladov použitia histogramov a plottera. Pre jednoduchosť budeme histogramy plniť generovanými náhodnými číslami, aby sme si nekomplikovali život čítaním dátových súborov. Začneme nejednoduchším zobrazením jednodimenzionálneho histogramu.

Listing 5.3: Example1.java

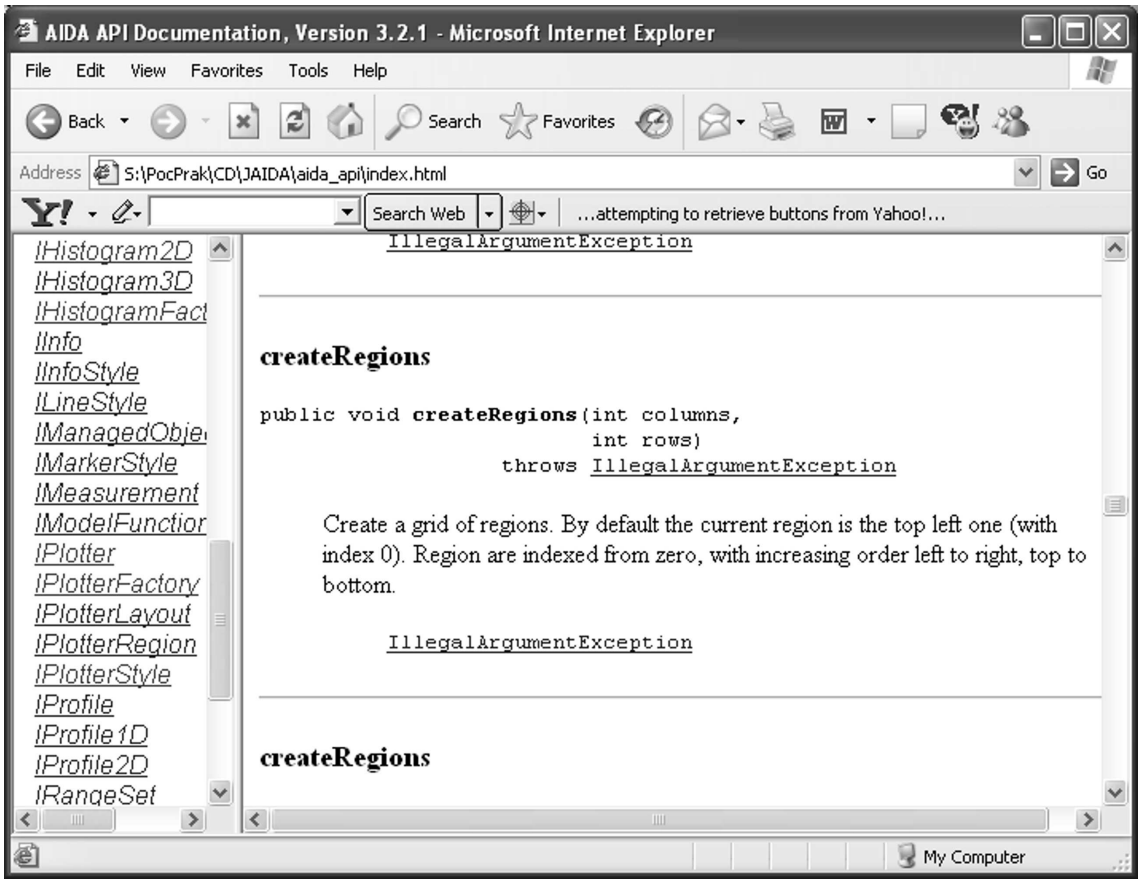
```
1 import hep.aida.*;
2 import java.util.Random;
```



Obrázok 5.2: Directory aida_api



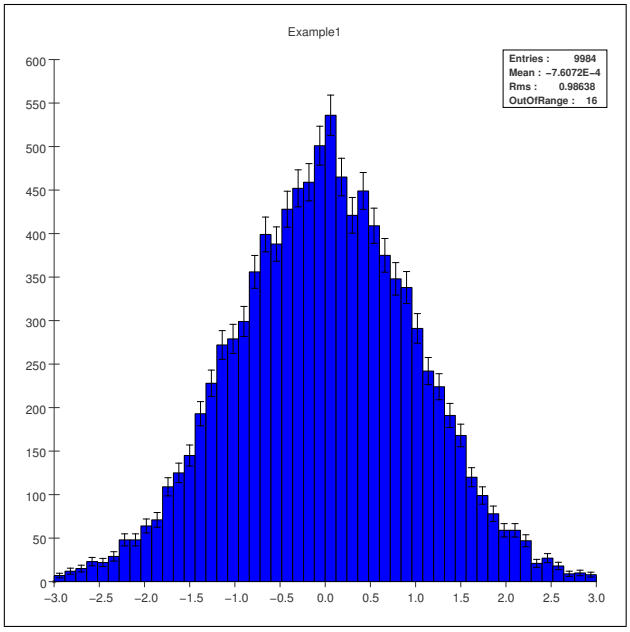
Obrázok 5.3: AIDA API



Obrázok 5.4: Metóda IPlotter.createRegions

```
3
4  public class Example1 {
5
6    public static void main(String[] args){
7      IAnalysisFactory af = IAnalysisFactory.create();
8      IHistogramFactory hf = af.createHistogramFactory(null);
9      IHistogram1D h = hf.createHistogram1D("Example1", 50, -3, 3);
10
11     Random r = new Random(1234567);
12     for (int i = 0; i < 10000; i++) {
13       h.fill(r.nextGaussian());
14     }
15
16     IPlotterFactory pf = af.createPlotterFactory();
17     IPlotter plotter = pf.create();
18     plotter.currentRegion().plot(h);
19     plotter.show();
20   }
21 }
```

Uvedený program vykreslí histogram tak, ako ukazuje obr.5.5



Obrázok 5.5: Example1

Komentár k Listingu 5.3

- Riadky 7,8,16,17 slúžia na kreovanie factories, ako sme to diskutovali v odseku 2.8.
- V riadku sa kreuje (prázdny) jednodimenzionálny histogram s titulkom "Example1", ktorý bude mať 50 binov v intervale $(-3, 3)$.
- Riadok 11 inicializuje náhodný generátor hodnotou seed = 1234567, ako sme to diskutovali v odseku 2.7

- V riadkoch 12-14 sa naplní histogram 10000 hodnotami náhodnej gaussovskej premennej so strednou hodnotou 0 a varianciou 1.
- V riadku 17 sa kreuje plotter. Všimnime si, že ďalej nepožadujeme jeho rozdelenie na viacero zobrazovacích panelov (regions), celé okno plottera bude teda jedinou zobrazovacou plochou.
- V riadku 18 sa histogram virtuálne zobrazí. Všimnime si, že to nie je plotter "kto vie kresliť", ale jeho región. Celý plotter je jediným regiónom, ale i tak o nakreslenie histogramu musíme požiadať aktuálny (current) región. Virtuálne nakresliť hovoríme preto, že tento príkaz ešte nič nezobrazuje na obrazovku, celý akt kreslenia sa odohráva v pamäti, v štruktúrach plottera. Zdôraznime ešte, že sme nijako nešpecifikovali, akým spôsobom chceme histogram zobraziť. Plotter teda použije defaultové hodnoty parametrov špecifikujúcich kresliaci štýl. Iné možnosti uvidíme v nasledujúcom príklade.
- Riadok 19 zabezpečí, aby sa na obrazovke vytvorilo nové okno a v ňom sa fyziky vykreslil doteraz virtuálny obraz
- Riadok 20 uzatvára kód metódy main, ale to neznamená, že sa beh programu končí. Príkaz show v riadku 19 totiž otvoril zobrazovacie interaktívne okno, ktoré ostáva komunikovať s užívateľom[4] i po ukončení metódy main. Komunikácia je prostá: program čaká, či klikneme na pravé myšacie tlačidlo a vyvoláme tak pop-up menu, alebo čaká, kým neklikneme na zatváracie tlačidlo (close button) v pravom hornom rohu okna. To vyvolá príkaz na zatvorenie okna, ukončenie komunikácie ako aj celého behu programu.
- Všimnime si na obr.5.5 v boxe štatistiky že 16 eventov bolo "OutOfRange", teda ich hodnoty nepadli do intervalu $(-3, 3)$. Tieto eventy padli do underflow a overflow binov a nie sú znázornené na obrázku. Celkový počet eventov je $9984 + 16 = 10000$ v zhode s tým, ako sme ich generovali.

Ďalší príklad vychádza z predchádzajúceho, ale dopĺňa ho o príkazy detailizujúce naše grafické požiadavky.

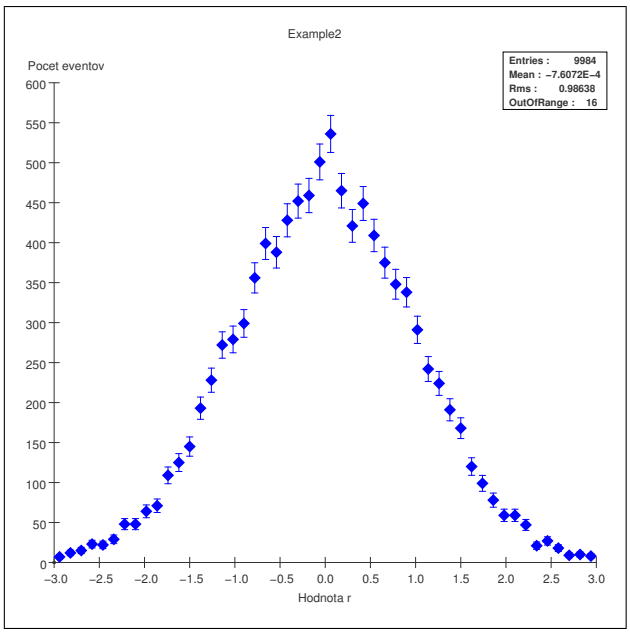
Listing 5.4: Example2.java

```
1  import hep.aida.*;
2  import java.util.Random;
3
4  public class Example2 {
5
6    public static void main(String[] args)throws Exception {
7      IAnalysisFactory af = IAnalysisFactory.create();
8      IHistogramFactory hf = af.createHistogramFactory(null);
9      IHistogram1D h = hf.createHistogram1D("Example2", 50, -3, 3);
10
11     Random r = new Random(1234567);
12     for (int i = 0; i < 10000; i++) {
13       h.fill(r.nextGaussian());
14     }
```

[4]Pre pokročilých niekoľko poznámok navyše. Príkaz show vyvolá nielen vykreslenie okna, ale spustí nové paralelné vlákno (Event dispatching thread). Úlohou toho vlákna je spracovanie eventov prichádzajúcich z vonkajšieho prostredia (klávesnica, myš) a dispečing týchto eventov príslušným event-procesujúcim metódam typu actionListener. Technicky ide o nekonečnú čakaciu slučku, ktorá čaká a reaguje na eventy. Pôvodné inicializačné vlákno (thread) je ukončené vykonaním posledného riadku metódy main. Event dispatching thread ale nezávisle na tom pokračuje v svojej nekonečnej čakacej slučke, až kým táto nie je násilím (break) ukončená reakciou programu napríklad na event "kliknutie myši na uzatváracie tlačidlo okna".

```
15
16      IPlotterFactory pf = af.createPlotterFactory();
17      IPlotter plotter = pf.create();
18      IPlotterStyle style = pf.createPlotterStyle();
19      style.setParameter("showStatisticsBox","true");
20      style.dataStyle().setParameter("fillHistogramBars","false");
21      style.dataStyle().setParameter("showHistogramBars","false");
22      style.dataStyle().setParameter("showDataPoints","true");
23      style.dataStyle().setParameter("showErrorBars","true");
24      style.dataStyle().markerStyle().setParameter("size","12");
25      style.dataStyle().markerStyle().setParameter("shape","3");
26      style.dataStyle().markerStyle().setParameter("color","blue");
27      style.xAxisStyle().setLabel("Hodnota r");
28      style.yAxisStyle().setLabel("Pocet eventov");
29      plotter.currentRegion().plot(h,style);
30      plotter.writeToFile("Example2.gif");
31      plotter.writeToFile("Example2.eps");
32      plotter.writeToFile("Example2.jpg");
33      plotter.writeToFile("Example2.pdf");
34
35      plotter.show();
36    }
37  }
```

Uvedený program vykreslí histogram tak, ako ukazuje obr.5.6



Obrázok 5.6: Example2

Komentár k Listingu 5.4

- V riadku 18 pomocou IPlotterFactory kreujeme IPlotterStyle. V ďalších riadkoch potom nastavíme jeho parametre a použijeme ho v riadku 29 ako špecifikátor grafického štýlu pre kreslenie histogramu. Objekt IPlotterStyle jednak obsahuje nejaké parametre, ktoré sa dajú nastavovať a jednak obsahuje "štýly nižšej úrovne" ako napríklad dataStyle a xAxisStyle, ktoré potom obsahujú parametre, ktoré sa opäť dajú nastavovať

- Riadok 19 ukazuje nastavenie parametra volaním metódy setParameter(String parameterName, String parameterValue). Konkrétne tu nastavujeme, že na obrázku sa má zobraziť "Statistics box"
- V riadkoch 20-23 sa nastavujú parametre "podštýlu" dataStyle. Význam je z kontextu zrejmý.
- V riadkoch 24-26 sa nastavujú parametre "podštýlu podštýlu" markerStyle, ktoré nastavujú aký má byť tvar, veľkosť a farba "dátových bodov". Význam možných hodnôt pre parametre size a shape pozri v tabuľkách na obr.5.7 a 5.8
- V riadkoch 27,28 nastavujeme popis osí.
- V riadku 29 vyvoláme virtuálne vykreslenie histogramu s použitím nastaveného štýlu.
- Riadky 30-33 demonštrujú schopnosť plottera uchovať vytvorený obrázok ako grafický súbor na disku. Koncovka (extension) mena súboru určuje ktorý grafický protokol bude použitý. Grafické súbory GIF, JPG a PDF sú všeobecne známe. EPS značí Encapsulated Post Script a je obľúbený vo vedeckej komunite, lebo je dobre vkladateľný do súborov vytvorených v TeXu. Upozornime na drobný bug v implementácii JAIDA, ktorý vyžaduje, aby metóda `writeToFile` bola volaná skôr ako metóda show. Nie je na to logický dôvod, len chyba v implementácii.

Pre účely tohto praktika vystačíme s nastavovaním štýlu v tej miere ako to ukazuje uvedený listing. Kompletný zoznam parametrov štýlu a ich hodnôt je uvedený v tabuľkách na obrázkoch 5.7 a 5.8.

AIDA Interface	Parameter name	Method	Description	Allowed values
IBrushStyle	color	setColor, color	Set the color of the brush stroke	Colors rotate
	opacity	setOpacity, opacity	Set the opacity of the brush stroke	-1, not set
ILineStyle	type	setLineType, lineType	Set the line's type	Not supported
	thickness	setLineThickness, lineThickness	Set the line's thickness	Not supported
ITextStyle	font	setFont, font	The text's font.	Default "SanSerif"
	fontSize	setFontSize, fontSize	The text's size.	A number, default is 12
	bold	setBold, isBold	Make the text bold.	true, false
	italic	setItalic, isItalic	Make the text italic.	true, false
	underlined	setUnerlined, isUnderlined	Make the text underlined.	true, false
IPlotterStyle	showStatisticsBox		Display the statistics box for all the data	true, false
	statisticsBoxFont		The font type used in the statistics box	SanSerif, Times, Comics, etc
	statisticsBoxFontStyle		The font style used in the statistics box	plain or 0, bold or 1, italic or 2, boldItalic or 3
	statisticsBoxFontSize		The font size used in the statistics box	a positive integer
	showLegend		Display the legend on the plot	true, false
	legendFont		The font type used in the legend	SanSerif, Times, Comics, etc
	legendFontStyle		The font style used in the legend	plain or 0, bold or 1, italic or 2, boldItalic or 3
	legendFontSize		The font size used in the legend	a positive integer
	backgroundColor		The plot's background color	a color, see below
	foregroundColor		The plot's foreground color	a color, see below
	dataAreaColor		The data area's background color	a color, see below
	dataAreaBorderType		The data area's border type	bevelIn or 0, bevelOut or 1, etched or 2, line or 3, shadow or 4
	hist2DStyle		The way a histogram2D is represented	box or 0, ellipse or 1, colorMap or 2
	showAsScatterPlot		For scatter plots only to switch between binned and unbinned representation	true, false. The default is true.
	showTitle		Display the title.	true, false. The default is true.

Obrázok 5.7: Mená a hodnoty paramterov pre IPlotter

AIDA Interface	Parameter name	Method	Description	Allowed values
IDataStyle	showHistogramBars		Display the line around the histogram's bars	true, false
	fillHistogramBars		Fill the histogram's bars	true, false
	showErrorBars		Show the error bars	true, false
	errorBarsColor		The error bars color	a color, see below
	profileErrors		The type of error bars for profile plots	spread or 0, errorOnMean or 1. The default is spread.
	showDataPoints		Show data points on top of the histogram's bars	true, false
	connectDataPoints		Connects the data points with a line	true, false
	lineBetweenPointsColor		The color of the line between the points	a color, see below
	functionLineColor		The color of a function	a color, see below
	showStatisticsBox		Display the statistics box for a given data set	true, false
IAxisStyle	label		The axis label	a string
	scale		The scale of the axis	lin, linear, log, logarithmic
	type		The axis type	double, int, string, date
	allowZeroSuppression		To allow zero suppression	true, false. The default is false.
IFillStyle	pattern	setPattern, pattern	Set the pattern of the filling	Not supported
	colorMapScheme		The type of color map for 2D histograms	warm or 0, cool or 1, thermal or 2, rainbow or 3, grayscale or 4
IMarkerStyle	shape	setShape, shape	The marker's shape	See below
	size		The size of the marker	a positive integer

The colors can be passed to the styles in the following formats:

- by name: "yellow" , where alpha is always 1.0
- by int r,g,b,a: "128, 255, 64, 255" , where alpha (a) is optional
- by float r,g,b,a: "0.5, 1.0, 0.25, 1.0" , where alpha (a) is optional
- by single number: "64637" or "0x0FFF08" , where alpha is always 1.0

The marker's shape can be either a string or a number (a String number); the available shapes are: "dot" or "0", "box" or "1", "triangle" or "2", "diamond" or "3", "star" or "4", "verticalLine" or "5", "horizontalLine" or "6", "cross" or "7", "circle" or "8" and "square" or "9".

Obrázok 5.8: Mená a hodnoty paramterov pre IPlotter pokr.

V ďalšom príklade si ukážme možnosti vizualizácie dvojdimenzionálneho histogramu

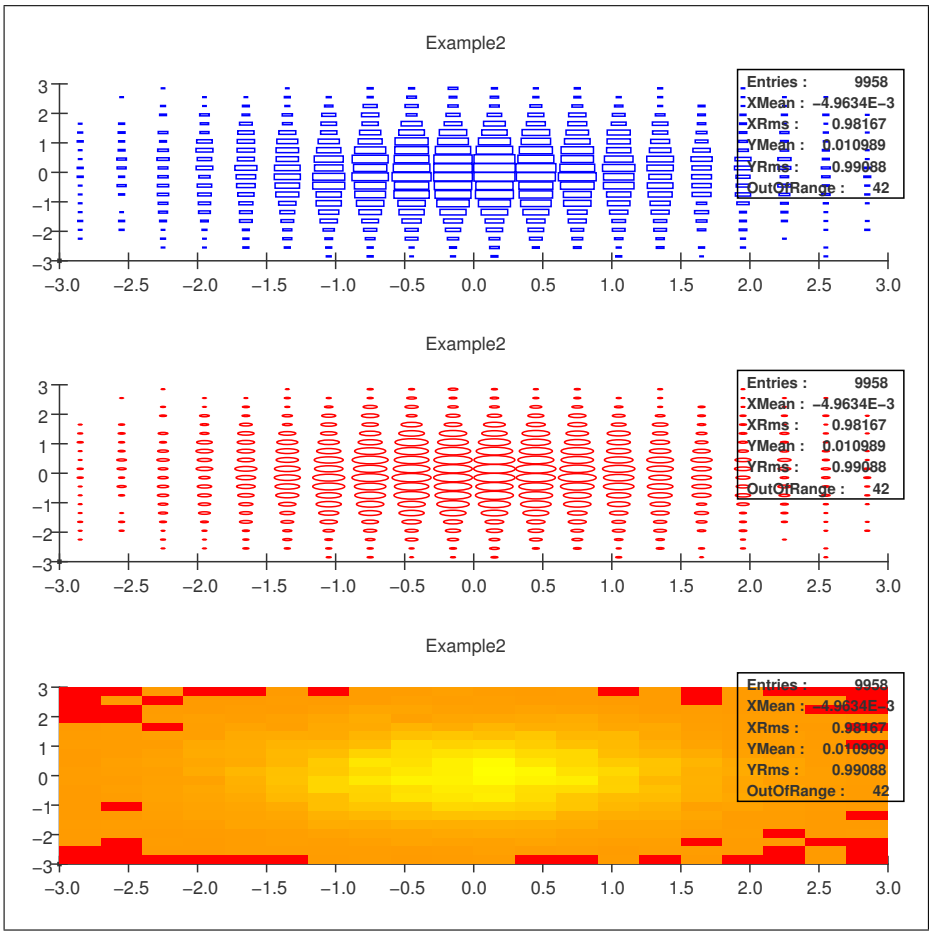
Listing 5.5: Example3.java

```
1  import hep.aida.*;
2  import java.util.Random;
3
4  public class Example3 {
5
6    public static void main(String[] args)throws Exception {
7      IAnalysisFactory af = IAnalysisFactory.create();
8      IHistogramFactory hf = af.createHistogramFactory(null);
9      IHistogram2D h2 = hf.createHistogram2D("Example2", 20, -3, 3, 20,-3,3);
10
11     Random r = new Random(1234567);
12     for (int i = 0; i < 10000; i++) {
13       h2.fill(r.nextGaussian(),r.nextGaussian());
14     }
15
16     IPlotterFactory pf = af.createPlotterFactory();
17     IPlotter plotter = pf.create();
18     plotter.createRegions(1,3);
19
20     IPlotterStyle regionStyle0 = plotter.region(0).style();
21     IPlotterStyle regionStyle1 = plotter.region(1).style();
22     IPlotterStyle regionStyle2 = plotter.region(2).style();
23     regionStyle0.setParameter("hist2DStyle","box");
24     regionStyle1.setParameter("hist2DStyle","ellipse");
25     regionStyle2.setParameter("hist2DStyle","colorMap");
26
27     plotter.region(0).plot(h2);
28     plotter.region(1).plot(h2);
29     plotter.region(2).plot(h2);
30
31     plotter.writeToFile("Example3.eps");
32     plotter.show();
33   }
34 }
```

Uvedený program vykreslí tri rozličné vizualizácie toho istého histogramu tak, ako ukazuje obr.5.9

Komentár k Listingu 5.5

- V riadku 9 kreujeme dvojdimenzionálny histogram, na osi x i y volíme rovnaké binovanie a to interval $(-3, 3)$ rozdelený na 20 binov
- V riadkoch 11-14 plníme tento histogram dvojicami gaussovských náhodných čísel
- V riadku 18 rozdelíme okno plottera na "šachovnicu" zobrazovacích polí, ktorá bude mať 1 stĺpec a tri riadky, teda tri políčka nad sebou
- V riadkoch 20-22 využívame fakt, že každé políčko, teda zobrazovací región plottera má samostatne nastavovateľný štýl.V týchto riadkoch sprístupňujeme štýly všetkých troch regiónov.
- V riadkoch 23-25 nastavujeme tri rozličné spôsoby vizualizácie dvojdimenzionálneho histogramu. Prvému spôsobu sa hovorí box-diagram, druhému elipsový diagram a tretiemu kódovanie farebnými odtieňmi. Vizualizácia je intuitívne zrejmá, bin v ktorom je relatívne viac zásahov znázorňujeme väčším obdĺžnikom, väčšou elipsou alebo svetlejším odtieňom farby.



Obrázok 5.9: Example3

- V riadkoch 27-29 potom virtuálne vykreslíme ten istý histogram (tie isté dáta) troma uvedenými vizualizačnými spôsobmi.
- V riadku 31 zapíšeme celé okno plottera do súboru na disku. Tak vznikol obrázok 5.9 ktorý tu bol prezentovaný cestou TeXu.

Osobitný príklad na použitie dátovej štruktúry typu cloud tu nebudeme uvádzať, videli sme jedno použitie na vykreslenie scatter-plotu v listingu 3.4 v odseku 3.3

5.8 Fitovanie histogramov

Histogramy si fyzik nekreslí len tak, pre potešenie. Výsledky experimentu si zaznamenáva do histogramu, aby zistil nejaké závislosti alebo súvislosti, prípadne aby popísal tvar toho histogramu. Popísať tvar znamená v matematickej reči preložiť tým histogramom nejakú krivku a napísať vzorec tej krivky.

5.8.1 Interface IFunction

Balík JAIDA umožňuje definovať funkciu jednej alebo aj viac premenných, ktorá obsahuje okrem premenných aj ďalšie nastaviteľné parametre, ktoré určujú konkrétny tvar tej funkcie. Pozrime si nasledovný príklad

Listing 5.6: Example4.java

```
1  import hep.aida.*;
2  import java.util.Random;
3  import hep.aida.ref.function.*;
4  public class Example4 {
5
6    public static void main(String[] args){
7      IAnalysisFactory af = IAnalysisFactory.create();
8      IHistogramFactory hf = af.createHistogramFactory(null);
9
10     IHistogram1D h = hf.createHistogram1D("Example1", 100, 0, 1);
11
12     Random r = new Random(1234567);
13     for (int i = 0; i < 16000; i++) {
14       double x = Math.sqrt(r.nextDouble());
15       h.fill(x);
16     }
17     IFunction f1 = new AbstractIFunction("Linear", 1, 2) {
18             public double value(double[] v) { return p[0]*v[0]+p[1]; }
19       };
20     f1.setParameters(new double[] {320.,0. } );
21
22     IPlotterFactory pf = af.createPlotterFactory();
23     IPlotterStyle style = pf.createPlotterStyle();
24     style.dataStyle().setParameter("fillHistogramBars","false");
25     style.dataStyle().setParameter("functionLineColor","red");
26     IPlotter plotter = pf.create();
27
28     plotter.currentRegion().plot(h,style);
29     plotter.currentRegion().plot(f1,style);
30
31     plotter.show();
32   }
33 }
```

Uvedený listing vyrobí obrázok 5.10 len s tým rozdielom, že "Statistics box"je vykreslený v pravom hornom rohu teda cez histogram, a je preto nečitateľný. Dá sa naň ale kliknúť a potom ho pretiahnuť myšou na voľné miesto, a až tak vznikol obrázok 5.10.

Komentár k Listingu 5.6

- V riadkoch 13-16 plníme histogram náhodnými číslami, generovanými v riadku 14. Dokážte si, že takto generované náhodné čísla sú z intervalu $(0, 1)$, a sú generované nerovnomerne, a to s hustotou pravdepodobnosti $\varrho(x) = 2x$.

- V riadkoch 17 až 19 je deklarovaná aj definovaná funkcia f1. Je deklarovaná ako inštancia interface IFunction, ale ie to objekt z tzv. anonymnej vnútornej triedy, odvodenej z abstraktnej triedy AbstractIFunction. Trieda AbstractIFunction je abstraktná lebo má len deklarovanú ale nie definovanú metódu

  `public double value(double []v)`.

  Úryvok zo zdrojáku tejto abstraktnej triedy je uvedený nižšie ako listing 5.7. Z abstraktnej triedy sa nedá vytvoriť objekt, musíme najprv vytvoriť podtriedu, tam doplniť chýbajúcu definíciu metódy a potom vytvoriť objekt. Tento celý dej je skrátený osobitnou konštrukciou jazyka Java v riadkoch, ktorý dovoľuje v krútených zátvorkách za operátorom new (riadok 17) doplniť chýbajúcu definíciu. Tu sa nám to podarilo na jedinom riadku, to nie je podmienka, môže to byť aj komplikovaný kód. Touto definíciou vlastne vzniká podtrieda triedy AbstractIFunction, táto podtrieda ale nemá meno preto sa volá anonymná. Objekt f1 ktorý vytvárame v riadku 17 je vlastne objektom tejto anonymnej podtriedy, ale v ďalšom kóde sa naň dívame iba ako na interface-objekt typu IFunction. Kto nechce, nemusí vnikať takto hlboko do kuchyne Javy, podľa uvedeného listingu si v analógii vytvorí sám potrebný kód bez hlbšieho porozumenia. Kto naopak chce vniknúť do problematiky hlbšie, odporúčam knihu B.Eckel:Thinking in Java (Jej tretie vydanie je ako html verzia je na sprievodnom CD, je tiež verejne prístupné na webe).

- Všimnime si v riadku 17 constructor: má tri parametre (porovnaj aj s riadkom 22 listingu 5.7). Prvý je typu String, je to "meno" funkcie a pre nás nemá význam, druhý je typu int a je to počet premenných funkcie a tretí parameter je opäť typu int a udáva počet parametrov funkcie. V kóde, ktorým vypočítavame funkčnú hodnotu sú premenné prvky poľa `double [] v` a parametre prvky poľa `double [] p`. V našom prípade sme definovali funkciu jednej premennej s dvoma parametrami, je to lineárna funkcia. Z riadku 19 by malo byť všetko jasné.

- V riadku 20 je ukázané, ako sa nastavujú hodnoty parametrov.

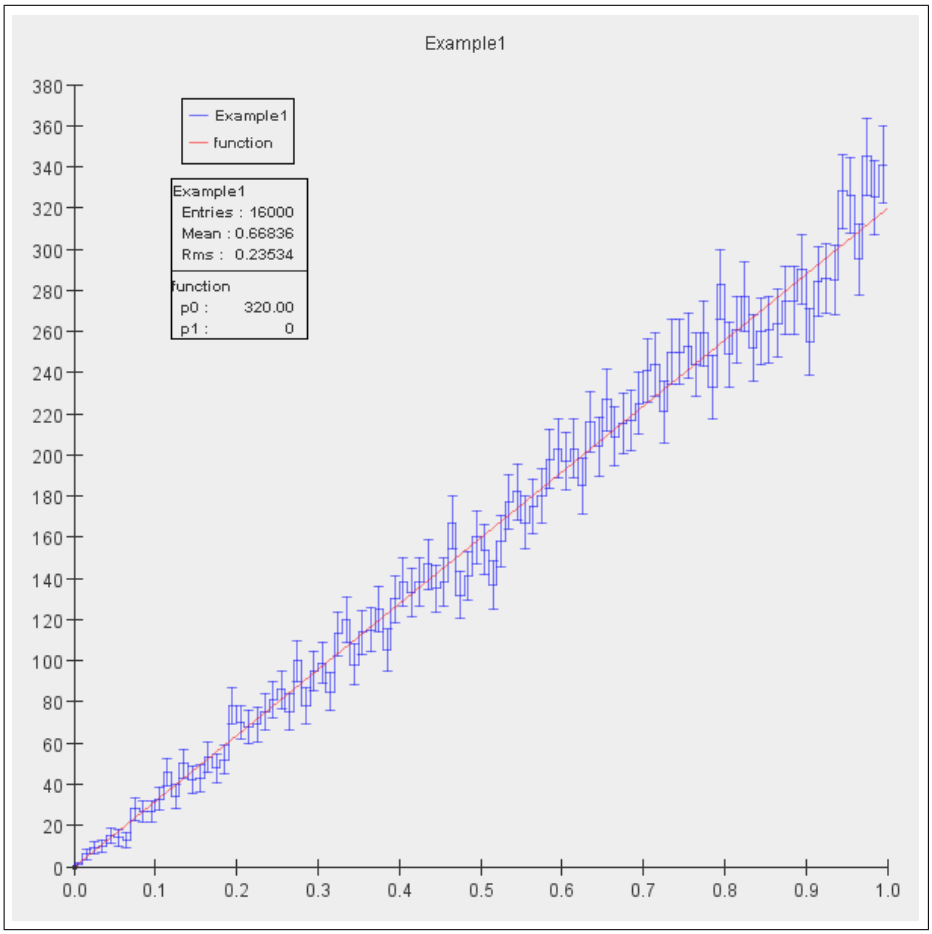
- V riadku 28 najprv necháme vykresliť histogram a potom v riadku 29 v tom istom regióne i funkciu f1

Poznámka. Všimnime si, že voľbou parametrov p[0]=320, p[1]=0, sme sa "trafili": preložili sme histogramom peknú priamku. Podumajte, ako sa to tak dobre podarilo, odkiaľ sme uhádli číslo 320. Malá pomoc: histogram ukazuje, že naozaj hodnoty x sú generované nerovnomerne podľa lineárneho zákona hustoty. Histogram je v podstate diskretizáciou tej funkcie hustoty, ibaže nie je správne normalizovaný. Jeho normalizácia je daná tým, že sme

celkovo urobili 16000 eventov. S tým dajako súvisí to číslo 320. Podumajte aj, ako by sme ten histogram mali preškálovať, aby sa ním pekne preložila priamka 2x.

Upozornenie!

Nie je možné zobraziť plotterom samostatne nejakú funkciu bez toho, aby sme a najprv nezobrazili nejaký histogram alebo inú podobnú dátovú štruktúru. Je to chyba balíka JAIDA, ale dá sa oklamať tak, že necháme zobraziť prázdny histogram, ktorý ale je definovaný, takže nastaví napríklad škálu na asi x.



Obrázok 5.10: Example4

Listing 5.7: Úryvky zo zdrojáku AbstractIFunction.java

```
1 package hep.aida.ref.function;
2 /**
3  * AbstractIFunction is implementation of the IFunction.
4  * User has to implement "value" method.
5  */
6 public abstract class AbstractIFunction  implements IModelFunction,
      //FunctionDispatcher {
7       .
8       .
9       .
```

```
10
11      protected String[] variableNames;
12      protected String[] parameterNames;
13
14      protected double[] p;
15      .
16      .
17      .
18      /** Creates a new instance of AbstractIFunction
19       *  with default variable names (x0, x1, ...)
20       *  and default parameter names (p0, p1, ...)
21       */
22      public AbstractIFunction(String title, int dimension, int numberOfParameters) {
23          variableNames = new String[dimension];
24          for (int i=0; i<dimension; i++) { variableNames[i] = "x"+i; }
25
26          parameterNames = new String[numberOfParameters];
27          for (int i=0; i<numberOfParameters; i++) { parameterNames[i] = "p"+i; }
28
29          init(title);
30      }
31      .
32      .
33      .
34
35      /** Provide value for your function here. Something like:
36       *  return p[0]*v[0]*v[0]+p[1]*v[0]+p[2];
37       */
38      public abstract double value(double[] v);
39      .
40      .
41      .
42  }
```

Niektoré funkcie sa vyskytujú pri spracovaní veľmi často, preto namiesto zložitého postupu pomocou rozširovania triedy AbstractIFunction môžme najprv vykreovať factory na kreovanie funkcií a potom touto fabrikou vykreovať nasledujúce funkcie

- polynóm n-tého stupňa ("pn")
- exponenciálnu funkciu ("e")
- jednodimenzionálny gauss ("g")
- dvojdimenzionálny gauss ("g2")

Príslušné vzorce a definícia parametrov vyzerajú nasledovne

$$\begin{aligned}
pn &= p[0](v[0])^n + p[1](v[0])^{n-1} + \cdots + p[n-1]v[0] + p[n] \\
e &= p[0]\exp(p[1]v[0]) \\
g &= p[0]\exp(-\frac{(v[0]-p[1])^2}{2(p[2])^2}) \\
g2 &= p[0]\exp(-\frac{(v[0]-p[1])^2}{2(p[2])^2})\exp(-\frac{(v[01]-p[3])^2}{2(p[4])^2})
\end{aligned}$$

Všimnime si, že gaussovské funkcie nie sú normalizované na jednotku, normalizačná konštanta je voľným parametrom. Je to tak preto, že histogramy, ktorými chceme "preložiť gaussovskú

krivku" nebývajú v praxi normalizované na jednotku ale "na počet eventov". Niečo podobné sme videli v listingu 5.6 pri "magickej konštante 320" v riadku 21.

Riadky 18-20 listingu 5.6 tak môžeme nahradiť alternatívou

Listing 5.8: Náhrada riadkov 18-20 listingu 5.6

```
18 \\vyssie sme kreovali af prikazom IAnalysisFactory af = IAnalysisFactory.create();
19 IFunctionFactory functionFactory = af.createFunctionFactory(null);
20 IFunction f1 = functionFactory.createFunctionByName("Linear", "p1");
```

Podobne môžme kreovať exponenciálnu funkciu a gaussovské funkcie príkazmi typu

```
IFunction f1 = functionFactory.createFunctionByName("Exponencial", "e");
IFunction f2 = functionFactory.createFunctionByName("Gauss", "g");
IFunction f3 = functionFactory.createFunctionByName("2-dim Gauss", "g2");
```

5.8.2 Jednoduchý fit

V príklade uvedenom v listingu 5.6 sme si potrápili hlavu a zistili sme že generovaným histogramom by sa mala dať preložiť priamka $y = 320x$. Fit bol úspešný, lebo sme mali k dispozícii teóriu, podľa ktorej bol fitovaný histogram vytvorený. Úlohou fyziky je objavovať nové teórie: namerať dáta, urobiť z nich histogram a potom uhádnuť šikovný vzorec tak, aby podľa neho nakreslená krivka fitovala nameraný histogram. Dobré uhádnutie rovná sa nový objav.

Slovo uhádnuť je naozaj primerané: nemáme žiadnu jednoznačnú technológiu ako sa optimálna fitovacia krivka zostrojí. Prinajmenej musíme podľa "fyzikálnej intuície" určiť nejakú triedu funkcií, v rámci ktorej už optimálnu konkrétnu funkciu nájde počítač. Ak si napríklad povieme, že náš tip je lineárna funkcia tvaru

$$y = ax + b$$

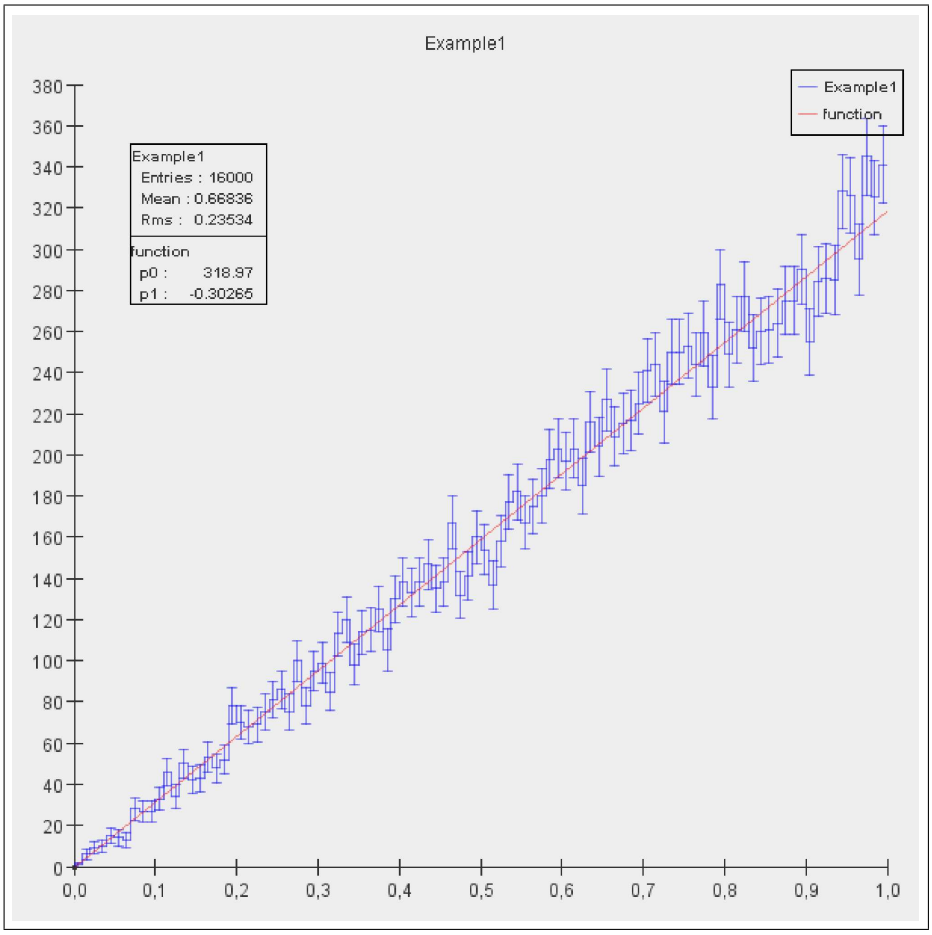
kde a a b sú zatiaľ neznáme voľné parametre, potom počítač už zbytok obvykle zvládne a optimálne hodnoty parametrov a a b určí. Presnejšie hodnoty parametrov nájde program, ktorému JAIDA hovorí fitter. Ako sa to robí si ukážeme na nasledujúcom príklade.

Listing 5.9: Eample5.java

```
1  import hep.aida.*;
2  import java.util.Random;
3  import hep.aida.ref.function.*;
4  public class Example5 {
5
6    public static void main(String[] args){
7      IAnalysisFactory af = IAnalysisFactory.create();
8      IHistogramFactory hf = af.createHistogramFactory(null);
9
10     IHistogram1D h = hf.createHistogram1D("Example1", 100, 0, 1);
11
12     Random r = new Random(1234567);
13     for (int i = 0; i < 16000; i++) {
14       double x = Math.sqrt(r.nextDouble());
15       h.fill(x);
16     }
17     IFunction f1 = new AbstractIFunction("Linear", 1, 2) {
18             public double value(double[] v) { return p[0]*v[0]+p[1]; }
```

```
19          };
20      f1.setParameters(new double[] {2.,0. } );
21
22      IFitFactory fitFactory = af.createFitFactory();
23      IFitter fitter = fitFactory.createFitter("chi2","jminuit","noClone=\"true\"");
24      IFitResult result = fitter.fit(h,f1);
25      double[] params = result.fittedParameters();
26      System.out.println(params[0]+" "+params[1]);
27
28      IPlotterFactory pf = af.createPlotterFactory();
29      IPlotterStyle style = pf.createPlotterStyle();
30      style.dataStyle().setParameter("fillHistogramBars","false");
31      style.dataStyle().setParameter("functionLineColor","red");
32
33      IPlotter plotter = pf.create();
34      plotter.currentRegion().plot(h,style);
35      plotter.currentRegion().plot(result.fittedFunction(),style);
36
37      plotter.show();
38    }
39  }
```

Uvedený program vykreslí obrázok 5.11



Obrázok 5.11: Example5

Komentár k listingu 5.9

- Listing 5.9 je až po riadok 19 zhodný s listingom 5.6
- Riadok 20 sa odlišuje, tvárime sa tu, že nepoznáme správne hodnoty parametrov fitu, strelili sme len tak náhodne hodnoty 2 a 0
- V riadku 22 kreujeme factory na kreovanie fitterov.
- V riadku 23 potom kreujeme fitter. K významu parametrov metódy, ktorú tu používame sa vrátime neskôr, zatiaľ to vnímame len ako "robí sa to takto".
- V riadku 24 fitter spustíme, povediac mu, že má urobiť fit histogramu h pomocou funkcie f1.
- Riadok 25 ukazuje, ako vydolujeme výsledky fitu, teda hodnoty nájdených optimálnych parametrov fitovacej funkcie, aby sme ich mohli v ďalšom nejako použiť.
- Riadok 26 ukazuje jednoduché použitie nájdených parametrov, prostý výstup.
- V riadkoch 33-37 je potom výstup výsledkov v grafickom tvare. Všimnite si osobitne riadok 35, ktorý ukazuje, ako sa vytvorí nájdený fit, teda funkcia, ktorá je výsledkom fitovania.
- Všimnite si na obrázku 5.11, že nájdené optimálne hodnoty parametrov lineárneho fitu sú uvedené aj na obrázku ako súčasť obsahu "štatistického boxu". Nájdené optimálne hodnoty $(318.97, -0.30)$ sa nezhodujú s teoretickými hodnotami $(320, 0)$. Je to tým, že vygenerovaný histogram nie je "ideálny", vidno na ňom štatistické fluktuácie. Tie spôsobia, že optimálna priamka sa im prispôsobí. Keby sme generovali nový histogram štartujúc s inou inicializáciou náhodného generátora, dostali by sme inú optimálnu priamku. Teoretická priamka by sa získala tak, že veľmi veľa pozorovateľov by vygenerovalo svoje histogramy, urobili by sme priemer z týchto histogramov a výsledným histogramom by sme preložili priamku.

5.8.3 Rukoväť fitovača

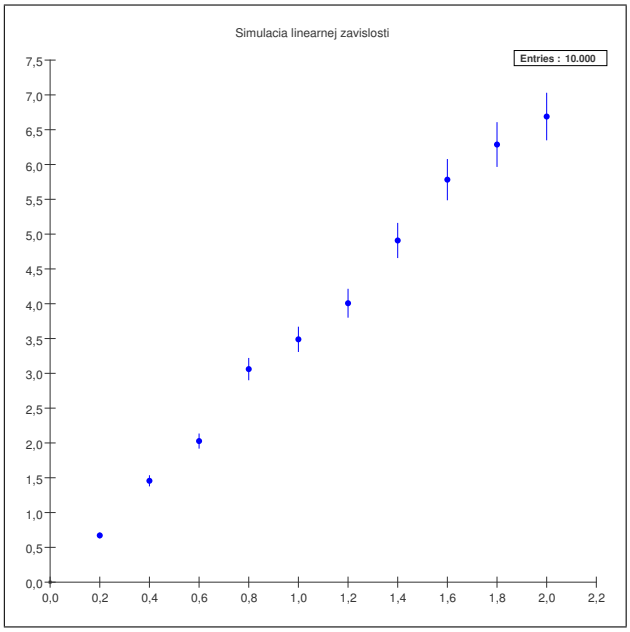
V tomto odseku si povieme niekoľko slov z "praktickej teórie" o fitovaní experimentálnych dát. Na vysvetlenie si najprv pripravíme nasimulované dáta. Môžeme si predstaviť niečo ako meranie elektrického odporu pomocou Ohmovho zákona. Pre niekoľko hodnôt odporom prechádzajúceho prúdu zmeriame prislúchajúce napätie na odpore. Predpokladajme, že máme veľmi presný ampérmeter (meradlo nezávislej premennej), zato voltmeter (meradlo závisle premennej dáva hodnoty zaťažené 5-percentnou chybou. Simulácia takého meranie je v nasledujúcom listingu.

Listing 5.10: Example6.java

```
1 import hep.aida.*;
2 import java.util.Random;
3 public class Example6 {
4
5    public static void main(String[] argv) {
6
7       IAnalysisFactory      af     = IAnalysisFactory.create();
```

```
8         IDataPointSetFactory dpsf    = af.createDataPointSetFactory(null);
9         IDataPointSet dataPointSet = dpsf.create("dummy","Simulacia linearnej
              //zavislosti",2);
10
11        Random r = new Random(1234567);
12        for ( int i = 0; i<10; i++ ) {
13          double x = 0.2*(i+1);
14          double y = (1+0.05*r.nextGaussian()) * (3.5*x);
15          double error =0.05*Math.abs(y);
16          dataPointSet.addPoint();
17          dataPointSet.point(i).coordinate(0).setValue(x);
18          dataPointSet.point(i).coordinate(1).setValue(y);
19          dataPointSet.point(i).coordinate(1).setErrorPlus(error);
20        }
21
22        IPlotter plotter = af.createPlotterFactory().create("Plot IDataPointSets");
23        plotter.region(0).setXLimits(0.,2.2);
24        plotter.region(0).plot(dataPointSet);
25        plotter.show();
26    }
27  }
```

Uvedený program vykreslí obrázok 5.12



Obrázok 5.12: Example6

Komentár k listingu 5.10

- V tomto príklade využívame dátovú štruktúru IDataPointSet, s ktorou sme sa stretli v odseku 5.4.3. V riadku 8 najprv kreujeme príslušnú factory a potom v riadku 9 samotnú dátovú štruktúru IDataPointSet. Volanie factory ma tri parametre, prvý je typu String a pre nás nemá význam, druhý je typu String a znamená titulok dát pri kreslení plotterom, tretí parameter je typu int a má význam dimenzionality dátových bodov. V našom prípade ide o dátové body v rovine (x,y), takže parameter má hodnotu 2.

- V riadkoch 12-20 nasimulujeme 10 dátových bodov. Vymyslené je to tak, že body by "podľa akejsi teórie" mali ležať na priamke $y = 3.5x$ ale vygenerujeme ich so simulovanou 5-percentnou gaussovskou odchýlkou merania. Deje sa to v riadku 14.

- Ku každému bodu pridáme i "akoby odhadovanú" neistotu merania o veľkosti 5% z nameranej hodnoty[5]. Hodnotu neistoty pridáme k dátovému bodu v riadku 19. Štruktúra IDataPointSet vie pracovať i s nesymetrickými neistotami, teda takými že neistota v kladnom smere je iná ako neistota v zápornom smere. My sme priradili neistotu ako kladnú neistotu, ale keďže sme nepriradili žiadnu zápornú neistotu, štruktúra IDataPointSet to chápe ako symetrickú neistotu, ako je ostatne vidno i s vytvoreného grafu.

- V riadku 23 sme explicitne nastavili rozsah vykresľovanej osi x. Dôvodom je, že chceme na obrázku vidieť aj nulový bod osi x. Keďže dátové body začínajú až pri $x = 0.2$, automaticky generovaný rozsah osi x by hodnotu $x = 0$ neobsahoval

Ako vidíme dátové body neležia presne na priamke. Samozrejme, veď sme ich z ideálnej polohy posunuli o hodnotu gaussovsky generovanej odchýlky. Čo to teda znamená, preložiť experimentálnymi bodmi najlepšiu priamku. Zjavne treba nejaké kritérium, ktoré pre každú možnú priamku vyčísli, nakoľko dobre popisuje dáta. Vari najčastejšie sa vo fyzike používa "chikvadrát test". Definovaný je nasledovne. Ak experimentálnymi bodmi (x_i, y_i) prekladáme funkciu $f(x)$, potom vyčíslime hodnotu

$$\chi^2 = \sum_i \frac{(y_i - f(x_i))^2}{\varepsilon_i^2}$$

kde ε_i je neistota (stredná kvadratická odchýlka) i-teho merania.

Čím nižšia hodnota χ^2 vyjde, tým lepšie funkcia popisuje dáta. Filozofia je zrejmá, minimalizujeme sumu normalizovaných kvadrátov odchýlok hodnôt funkcie $f(x_i)$ od nameraných hodnôt y_i. Kľúčové je pritom slovo "normalizovaných". Príslušný kvadrát odchýlky je predelený kvadrátom chyby (neistoty). Ak je hodnota ε_i odhadnutá (alebo odmeraná) dobre, potom očakávaná (typická) hodnota kvadrátu odchýlky v čitateli je rovnako veľká ako kvadrát chyby v menovateli a každý člen typicky prispieva hodnotou 1. Očakávame teda, že pre správnu funkciu $f(x)$ dostaneme hodnotu

$$\chi^2 \approx n$$

[5]Zopakujme si, že keby nešlo o simuláciu, hodnoty y by mali byť získané ako stredné hodnoty opakovaného merania v tom istom bode a hodnoty neistôt potom príslušné stredné kvadratické odchýlky. V praxi to často tak nerobíme. Každý bod sa ampérmetrom zmeria len raz. Ten ampérmeter neukáže presnú hodnotu, ale opakované meranie tým istým ampérmetrom by nemalo zmysel, namerali by sme rovnako nepresnú hodnotu. Spravidla totiž ide o systematickú chybu meracieho prístroja v danom bode, ale tú systematickú chybu nemáme dobre pod kontrolou: dalo by sa povedať, že veľkosť systematických chýb je náhodne rozdelená pozdĺž meracieho rozsahu prístroja. Iný kus rovnakého meracieho prístroja by v tom bode nameral inú hodnotu. Nebolo by účelné kupovať na opakované merania veľa rovnakých prístrojov. Namiesto toho určíme hodnotu neistoty odhadom, na základe údajov od výrobcu prístroja, ktorý v certifikáte napríklad hovorí že presnosť prístroja je 5%. Hodnota neistoty potom nepredstavuje strednú kvadratickú odchýlku meraní, ale mieru našej nevedomosti o veľkosti systematickej chyby prístroja v danom bode. Zdalo by sa, že takýto subjektivistický výraz "miera našej nevedomosti" nemá čo hľadať vo vede. Nie je tu miesto na podrobnejšiu diskusiu, kľúčové heslo pre záujemcov je "Bayesovská štatistika", ktorá v posledných rokoch nadobúda na význame pri spracovaní experimentálnych dát. Pravdou ale je, že dve štatistické školy "Bayesiáni" a "frekvencionisti" sú úhlavní nepriatelia a navzájom sa na konferenciách častujú ironickými až zlostnými poznámkami.

kde n je počet dátových bodov. Ak by sme dostali hodnotu dosť väčšiu ako n, znamenalo by to, že funkcia $f(x)$ nie je "tá pravá". Naopak, ak by sme dostali hodnotu oveľa menšiu ako n, znamenalo by to, že niekde sme prestrelili. Buď sme chyby odhadli zle, alebo sme príliš znásilnili funkciu $f(x)$. Napríklad je jasné, že 10 bodmi možno **presne** preložiť polynóm deviateho stupňa. Pre takú voľbu $f(x)$ dostaneme $\chi^2 = 0$, ale zjavne to nie je "to pravé orechové".

Práve popísaným spôsobom môžme testovať, či nejaká teoreticky očakávaná funkcia fituje namerané dáta. Častejšie mávame iný problém, nepoznáme teoretické parametre funkcie, ktorá má fitovať dáta, práve tie parametre máme určiť z experimentu. Potom postupujeme tak, že vo vzorci pre fitovaciu funkciu necháme niekoľko voľných parametrov. Bude teda

$$f(x, p_0, p_1, ...p_k)$$

Pre každú konkretizáciu hodnôt tých parametrov vieme určiť príslušnú hodnotu χ^2. Chikvadrát sa tak vlastne stal funkciou tých voľných parametrov

$$\chi^2(p_0, p_1, ...p_k)$$

Optimálne hodnoty parametrov určíme tak, že minimalizujeme hodnotu $\chi^2(p_0, p_1, ...p_k)$. Pozrime sa teraz, akú výslednú hodnotu χ^2 možno očakávať po minimalizácii. Ak máme n experimentálnych bodov, potom funkcia $f(x, p_0, p_1, ...p_k)$, ktorá má k voľných parametrov sa spravidla dá znásilniť tak, aby presne prechádzala k zvolenými bodmi. Argumentácia to nie je veľmi silná, ale cítime, že na testovanie kvality nám ostáva teda len $(n - k)$ ostatných experimentálnych bodov. Očakávame teda, že po minimalizácii dostaneme

$$\chi^2 \approx n - k$$

Hodnotu $(n-k)$ zvykneme nazývať "počet stupňov voľnosti fitu". Zavádza sa preto kritérium kvality fitu v tvare

$$\frac{\chi^2}{n-k}$$

Kvalitný fit by teda mal dosiahnuť hodnotu

$$\frac{\chi^2}{n-k} \approx 1$$

Kľúčom k fitovaniu je teda schopnosť nachádzať minimum funkcie viacerých premenných (parametre $p_0, p_1, ...p_k$ funkcie f sú premenné funkcie $\chi^2(p_0, p_1, ...p_k)$). Len málokedy vieme nájsť príslušné minimum analyticky, spravidla sa to robí numericky. Balík JAIDA obsahuje viacero numerických minimalizačných procedúr, my budeme používať procedúru JMINUIT čo je v Jave sprístupnený cernovský klasický balík MINUIT. Nebudeme tu vnikať do tajov numerickej optimalizácie, pre záujemcov je na sprievodnom CD článok klasika Freda Jamesa Function Minimization, kde sú zrozumiteľne vysvetlené základné numerické techniky minimalizácie.

Ukážme si teraz ako sa to prakticky robí. Doplňme kód uvedený ako listing 5.10 o príkazy definujúce fitovaciu funkciu a numerickú optimalizáciu.

Listing 5.11: Example7.java

```
1 import hep.aida.*;
```

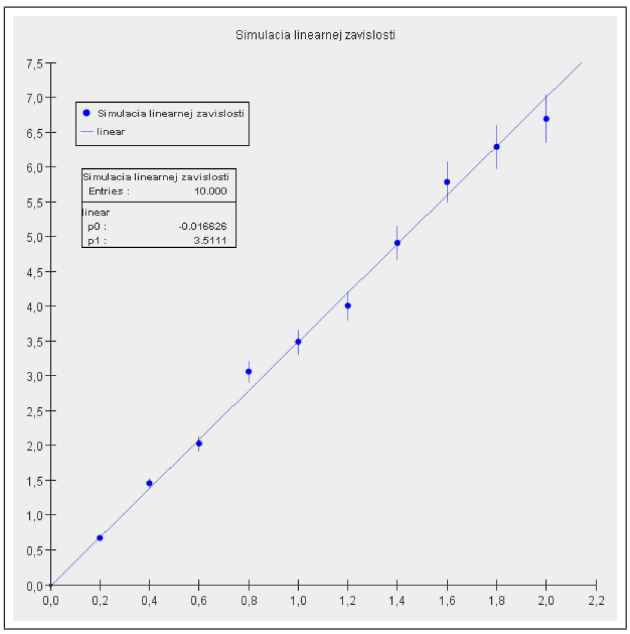
```
2  import java.util.Random;
3  import hep.aida.ref.function.*;
4  public class Example7 {
5
6     public static void main(String[] argv) {
7
8        IAnalysisFactory      af     = IAnalysisFactory.create();
9        IDataPointSetFactory dpsf    = af.createDataPointSetFactory(null);
10       IDataPointSet dataPointSet = dpsf.create("dummy","Simulacia linearnej
             //zavislosti",2);
11
12       Random r = new Random(1234567);
13       for ( int i = 0; i<10; i++ ) {
14         double x = 0.2*(i+1);
15         double y = (1+0.05*r.nextGaussian()) * (3.5*x);
16         double error =0.05*Math.abs(y);
17         dataPointSet.addPoint();
18         dataPointSet.point(i).coordinate(0).setValue(x);
19         dataPointSet.point(i).coordinate(1).setValue(y);
20         dataPointSet.point(i).coordinate(1).setErrorPlus(error);
21       }
22       IFunctionFactory ff = af.createFunctionFactory(null);
23       IFunction f1 = ff.createFunctionByName("linear","p1");
24       f1.setParameters(new double[] {1.,0. } );
25
26       IFitFactory fitFactory = af.createFitFactory();
27       IFitter fitter = fitFactory.createFitter("chi2","jminuit","noClone=\"true\"");
28       IFitResult result = fitter.fit(dataPointSet,f1);
29       System.out.println("quality= "+result.quality());
30       System.out.println("p[0]= "+result.fittedParameters()[0]);
31       System.out.println("p[1]= "+result.fittedParameters()[1]);
32       IPlotter plotter = af.createPlotterFactory().create("Plot IDataPointSets");
33       plotter.region(0).setXLimits(0.,2.2);
34       plotter.region(0).plot(dataPointSet);
35       plotter.currentRegion().plot(result.fittedFunction());
36       plotter.show();
37    }
38 }
```

Uvedený program vykreslí obrázok 5.13

Komentár k listingu 5.11

- V riadkoch 22-24 definujeme fitovaciu funkciu ako polynóm prvého stupňa a zadávame počiatočné hodnoty jeho parametrov. Lineárny fit je pre optimalizačnú procedúru ľahký, na počiatočných hodnotách prakticky nezáleží.

- V riadkoch 26 a 27 kreujeme najprv príslušnú factory a potom "fitovací nástroj" IFitter. Parametre s ktorými vytvárame fitter sú pre naše použitie prakticky nemenné. Všetky parametre sú typu String. Prvý má hodnotu "chi2" a vyberá typ štatistického testu ktorý chceme použiť. My budeme používať len chi2, ostatné možnosti a príslušnú teóriu si môže záujemca doplniť z dokumentácie k balíku JAIDA a zo štatistickej literatúry. Druhý parameter má hodnotu "jminuit" a vyberá konkrétny fitovací nástroj. JAIDA poskytuje i iné minimalizátory, ale nebudeme ich tu spomínať. Tretí parameter treba pozorne opísať presne tak ako je uvedené v príklade (najmä pozor na úvodzovky a backslashe!). Nebudeme sa trápiť s jeho významom, v dokumentácii nie je o ňom nič a trvalo mi dlho nájsť v zdrojákoch prečo ho v určitých prípadoch treba použiť presne takto. V danom príklade by nemusel byť uvedený, ale nepoškodí, inokedy ho vyslovene treba. Je to jeden z príkladov, ktorý som už spomínal, keď dokumentácia zlyháva a nestačí za vývojom programov.



Obrázok 5.13: Example7

- Riadok 28 spúšťa numerickú minimalizáciu. Hovorí, že treba fitovať dataPointSet pomocou funkcie f1. Výsledky minimalizáciu sú uložené do štruktúry result typu IFitResult.
- Riadky 29-31 ukazujú, ako vytiahnuť výsledky zo štruktúry result pre ďalšie použitie, v tomto prípade pre výpis na štandardnom výstupe (termináli)
- Riadok 35 ukazuje ako vyplotovať nájdenú optimálnu funkciu

Ešte niekoľko poznámok k numerickej minimalizácii. Spúšťame tu automat MINUIT spôsobom "cvičená opica", netušiac ako to ten MINUIT robí. Väčšinou to dopadne dobre a nafitujeme dáta. Niekedy sa ale môže stať, že MINUIT "spadne". Vyhodí nejakú chybovú hlášku. Najčastejšie, že sa mu nepodarilo skonvergovať a nájsť dobré minimum. V tom prípade sa môžme ešte pokúsiť zadať iné počiatočné hodnoty parametrov a dúfať vo väčšie šťastie. Ak to nepomôže, najjednoduchšie je vzdať to. Druhá možnosť je stať sa expertom na MINUIT, prečítať si dokumentáciu k jeho fortranovskej verzii v cernovskej knižnici, potom dokumentáciu k JAIDA a Java FreeHEP Library, ktorá je vyslovene slabá, potom lúštiť zdrojáky. Alebo si nainštalovať cernovskú knižnicu a skúsiť to vo FORTRANe. Alebo vypátrať, v akom stave je implementácia do C++ v CERNe, a urobiť to v C++.

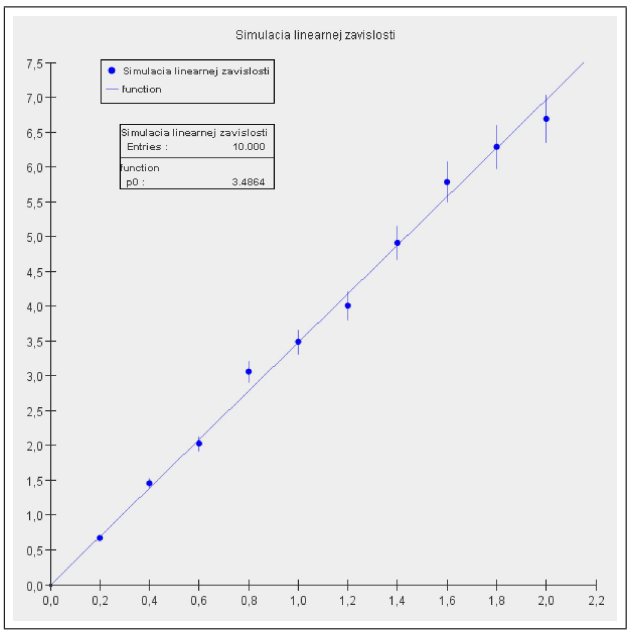
Začali sme kontextuálne ladenou diskusiou o Ohmovom zákone. Poznamenajme preto, že kód uvedený v listingu 5.11 tomu nezodpovedá. Ohmov zákon je priama úmernosť (teda jednoparametrický fit), my sme použili lineárnu funkciu, teda dvojparametrický fit, nájdená priamka neprechádza nulou. Ak chceme naozaj priamu úmernosť, musíme fitovaciu funkciu napísať inak, napríklad takto.

Listing 5.12: Example8.java

```
1  import hep.aida.*;
2  import java.util.Random;
3  import hep.aida.ref.function.*;
4  public class Example8 {
5
6     public static void main(String[] argv) {
7
8        IAnalysisFactory      af     = IAnalysisFactory.create();
9        IDataPointSetFactory dpsf    = af.createDataPointSetFactory(null);
10       IDataPointSet dataPointSet = dpsf.create("dummy","Simulacia linearnej
             //zavislosti",2);
11
12       Random r = new Random(1234567);
13       for ( int i = 0; i<10; i++ ) {
14         double x = 0.2*(i+1);
15         double y = (1+0.05*r.nextGaussian()) * (3.5*x);
16         double error =0.05*Math.abs(y);
17         dataPointSet.addPoint();
18         dataPointSet.point(i).coordinate(0).setValue(x);
19         dataPointSet.point(i).coordinate(1).setValue(y);
20         dataPointSet.point(i).coordinate(1).setErrorPlus(error);
21       }
22       IFunction f1 = new AbstractIFunction("Proportionality", 1, 1) {
23             public double value(double[] v) { return p[0]*v[0]; }
24       };
25       f1.setParameters(new double[] {1. } );
26
27
28       IFitFactory fitFactory = af.createFitFactory();
29       IFitter fitter = fitFactory.createFitter("chi2","jminuit","noClone=\"true\"");
30       IFitResult result = fitter.fit(dataPointSet,f1);
31       System.out.println("quality= "+result.quality());
32       System.out.println("p[0]= "+result.fittedParameters()[0]);
33       IPlotter plotter = af.createPlotterFactory().create("Plot IDataPointSets");
34       plotter.region(0).setXLimits(0.,2.2);
35       plotter.region(0).plot(dataPointSet);
36       plotter.currentRegion().plot(result.fittedFunction());
37       plotter.show();
38    }
39 }
```

Uvedený program vykreslí obrázok 5.14

Všimnime si, že koeficient pri lineárnom člene v priamej úmernosti je jemne iný než koeficient pri lineárnom člene v dvojparametrickom fite. Pri dvojparametrickom fite sa lepší súhlas s dátami dosiahol malým konštantným členom. Zato kvalita fitu sa zhoršila pri dvojparametrickom fite, lebo klesol počet stupňov voľnosti: hodnota χ^2 je pri dvojparametrickom fite nižšia ako pri jednoparametrickom, ale χ^2 pripadajúci na jeden stupeň voľnosti je lepší pre jednoparametrický fit.



Obrázok 5.14: Example8

Kapitola 6

Písanie zápisov na praktikách

Z každého praktického cvičenia je treba odovzdať zápis (protokol, report) v elektronickej forme. Odporúčame použiť Microsoft Word, OpenOffice Writer alebo TeX. V každom prípade word procesor umožňujúci písanie matematických vzorcov a vkladanie obrázkov. Všetky praktiká sú sčasti negatívne vnímané preto, lebo sa vyžaduje písanie protokolov. Je to fakt otrava. Ale nie je pravda, že sa netreba učiť, "lebo keď to budem naozaj potrebovať, tak to nie je kumšt urobiť".

Z vlastnej skúsenosti hovorím, že vážne trpím nedostatkom sebadisciplíny pri robení si vlastných pracovných poznámok. Potom musím všetko robiť veľakrát a znovu a znovu si čarbať tie isté odvodzovačky ako prípravu na prednášku, riešiť tie isté problémy ako sa spúšťa nejaká programová utilita a opakovať experimenty (ja teda najmä počítačové), lebo zo starých poznámok už neviem zistiť, grafom čoho je ten obrázok a s akými hodnotami parametrov to bolo spočítané.

Som obézny, ale dobre vám radím, neprejedajte sa. Som neporiadny, ale dobre vám radím, veďte si "palubný denník" (log book), kde si zapisujte všetko to, čo na znovuzískanie potrebuje podstatne viac času ako na ten zápis do log-booku. Keď sa vám po desiatkach márnych pokusov podarí nainštalovať nejaký soft, zapíšte si, ako ste to dosiahli, lebo to určite zabudnete. Pre experimentálneho fyzika je log-book absolútnou podmienkou života[1].

Podrobnosti o spôsobe odovzdávania protokolov a termínoch budú oznámené na cvičeniach.

Tu sa obmedzíme len na niekoľko technických poznámok.

Forma protokolu bude pre niektoré cvičenia predpísaná, tak, že budú zadané otázky, na ktoré bude treba napísať odpoveď vo forme číselnej hodnoty, voľne formulovaného textu, matematického vzorca alebo obrázku získaného grafu a podobne. Nie úplne triviálne je písanie vzorcov, musíte sa to naučiť. Podobne vkladanie obrázkov nemusí byť vždy priamočiare.

[1] Môj študent Mišo Kreps si získal dôveru šéfa experimentálnej skupiny v CERNe nielen tým, že bol šikovný, ale aj tým, že si viedol mimoriadne poriadne log-book. A dôvera znamená zverenie dôležitých vecí a napísanie dobrého posudku, keď sa človek posúva na ďalšie miesto. Tvrdím to veľmi zodpovedne, rozprával som sa s tým šéfom.

Niekedy sa dá použiť primitívna forma copy-paste. Napríklad pod Windowsami stlačenie klávesy PrintScreen vyvolá uloženie aktuálnej obrazovky vo forme rastrového obrazu do clipboardu. Ak chcem na clipboard dostať len obrázok aktuálneho okna, použije sa kombinácia Alt-PrintScreen. V mnohých aplikáciách potom príkaz paste (Ctrl-V) vloží takto získaný obraz to dokumentu. Ak automaticky navrhnutá veľkosť nie je optimálna, dajú sa väčšinou použiť dodatočné formátovacie príkazy. Vedieť sa v takejto situácii zorientovať patrí k elementárnej počítačovej gramotnosti, ktorá sa na tomto praktiku predpokladá. To, samozrejme, neznamená, že sa nemôžte pri problémoch učiteľa spýtať na radu.

Pri word procesoroch je niekedy problém, že pri použití obyčajného Paste príkazu (ako Ctrl-V) sa správajú čudne. Je to vtedy, ak zdrojom, ktorý kopírujeme je tiež nejaká formátovania schopná aplikácia, ktorá používa vlastný formátovací protokol, odlišný od nášho word procesora. Word procesor sa pri Paste príkaze snaží prevziať aj formátovanie originálu, a niekedy to dopadne zle. Konkrétne pri kopírovaní zo SciTE do OpenOffice Writera tento na dlhú dobu stvrdne. V takých prípadoch je treba používať príkaz typu Paste-Special a potom v dialógu vybrať možnosť typu unformatted text. Originál sa prevezme len ako čistý text. Potom ho môžme ručne sformátovať podľa potreby.

Alternatívou k použitiu kopírovania obrazovky systémovým prostriedkom je použite grafickej aplikácie, ktorá takú funkciu má. Napríklad GIMP2 vie skopírovať obrazovky alebo okna a potom umožňuje takto získaný obraz ďalej spracovať štandardnými technikami spracovania obrazu. Toto tiež patrí ku gramotnosti budúceho fyzika. Odborné články, to nie sú len texty ale aj veľa obrázkov, náčrtkov a grafov a ich vkladanie do vytváraných dokumentov.

Najkvalitnejší spôsob práce s obrázkami je taký, ak aplikácia, ktorá obrázok vytvorí, ho vie aj uložiť na disk ako obrazový súbor vo vhodnom grafickom protokole (tiff, gif, bmp, jpeg, png, eps). Problém býva a v tom, že aplikácia, ktorou vytvárate cieľový dokument vie spravidla vložiť do dokumentu obrázky získané prečítaním grafického súboru, ale nepozná práve ten typ súboru, ktorý zdrojová aplikácia vytvorila. Vtedy treba použiť nejaké "manuálne rozhranie", teda otvoriť univerzálny grafický nástroj ako GIMP2, načítať zdrojový obrazový súbor, podľa potreby ho upraviť a potom uložiť ako cieľový obrazový súbor v cieľovom grafickom protokole.

Nakoniec ešte poznamenajme, že v tomto praktiku sa nebude vyžadovať znalosť TeXu. Ale veľmi odporúčam každému, kto uvažuje o kariére vo fyzike zvládnuť LaTeX. Zatiaľ je to stále hlavné prostredie pre prípravu fyzikálnych dokumentov. I keď rôzne primárne pdf-orientované nástroje sú na postupe. V každom prípade TeX je zadarmo.

Časť II

Úlohy pre praktikum

Kapitola 7

Prehľad úloh a cieľov

Tu je prehľad úloh, ktoré by sa mali cvičiť v priebehu semestra. Je to zatiaľ len rámcový plán, lebo nemám dosť dobrý odhad, ako to pôjde, koľko sa bude dať v priebehu cvičenia urobiť, čo bude študentov viac zaujímať. V priebehu semestra sa teda bude plán dynamicky upravovať.

Prosba

Podieľajte sa sa aktívne na tvorbe tohto predmetu. Prichádzajte s nápadmi, kritikou, otázkami. Dajte vedieť, že nerozumiete. Dajte vedieť, že vás to nebaví. Predmet beží prvýkrát. Cvičiaci aj študenti to nebudú mať ľahké. Ale môžme sa spolu pokúsiť pripraviť zaujímavý predmet, v ktorom sa cvičiaci aj študenti niečo naučia teraz ale najmä v budúcnosti. Vyskúšajme to a potom zhodnoťme, či to stálo za to, zapísať sa na taký predmet. Možno celú ideu bude treba opustiť ako neprinášajúcu adekvátne výsledky. Možno sa ukáže, že to je naozaj len pre geekov. A možno to bude zaujímavé.

1. Šikmý vrh v spreadsheete. Práca v spreadsheete. Kreslenie grafov. Eulerova metóda. Triafanie loptou do koša
2. Obezita. Analýza rozloženia pravdepodobnosti výšky. Fitovanie gaussami. Korelácia výšky a váhy.
3. Fázový priestor harmonického oscilátora. Smernice dotyčníc. Eulerova metóda. Animácia v grafe. Liouvilleova veta. Liouvilleova veta pre sínusový oscilátor
4. Harmonický oscilátor v kontakte s plynom. Termalizácia. Fitovanie gaussom. Teplota.
5. Biliardový stôl s jednou molekulou. Zmena hybnosti pri náraze na stenu. Boyle Marriotov zákon
6. Siločiary a ekvipotenciálne hladiny sústavy dvoch nábojov. Krivka. Interpolácia krivkou cez štyri body. Krivkový integrál

7. Adiabata ako riešenie diferenciálnej rovnice. Smernice v pV diagrame pre adiabatický dej. Fitovanie kappa.
8. Fourierova analýza. Rozklad jednoduchých kriviek. rozklad vlastnej krivky. Riešenie vlnovej rovnice cez rozklad.
9. Tlmený oscilátor. Fourierov rozklad. Princíp neurčitosti. Riadenie hodín.
10. Huyghensov princíp. Skladanie vĺn na zrkadle. Zákon odrazu.
11. Usmerňovač s filtrom, prechodový jav. Návrh parametrov obvodu aby brum bol menší ako zadaná hodnota.
12. Rozpad neutrálnej častice na dve nabité častice v magnetickom poli. Určenie hybností. Doba letu. Experimentálne určenie hmotnosti materskej častice
13. Ray tracing na spojnej šošovke, hĺbka ostrosti v závislosti na svetelnosti.
14. Relaxačná metóda pre Poissonovu rovnicu, potenciál náboja pri vodivej doske.Vykreslenie siločiar.

7.1 Všeobecné poznámky

7.2 Úloha. Šikmý vrh

Zopakovanie "teórie" šikmého vrhu. Tvar dráhy, dolet, maximálna výška, maximálny možný dolet

Zoznámenie sa s tabuľkovým kalkulátorom. Zadávanie hodnôt a funkcií. Vypĺňanie stĺpca ťahom. Nakreslenie grafu funkcie sinus. Pozorovanie zmien grafu pri zmene frekvencie a amplitúdy a fázy. Práca s pop-up menu zmeny popisu, farby, veľkosti dátových bodov. Dva grafy v jednom obrázku, sínus a kosínus.

Preštudovanie polotovaru "voľný pád".

Napísanie "programu" v tabuľkovom kalkulátore pre šikmý vrh. Graf dráhy. Závisloti súradníc a priemetov rýchlosti na čase.

Riešenie okrajovej úlohy triafaním.

Pridanie odporu prostredia, nakreslenie balistickej krivky.

Pozorovanie škálovania: vykresliť dĺžku vrhu v závislosti na uhle pre rôzne hodnoty počiatočnej rýchlosti, body si nesadnú an jednu čiaru. Potom podumať, čo vynášať v závislosti na uhle, aby si všetky body sadli na jednu čiaru. Aký bude tvar tej čiary pre pohyb bez odporu vzduchu. Pozorujte ako bude vyzerať tá čiara pre vrh v odporujúcom prostredí.

Námet na žolík:

- Vyriešiť úlohu triafanie do koša basketbalovou loptou v prostredí s odporom vzduchu, vyhľadať na internete relevantné údaje a dať pozor, že lopta letí cez obruč šikmo: úlohou je aby žiadnym bodom svojho povrchu nezasiahla obruč, má preletieť "hladko"

7.3 Úloha. Obezita

Domáca príprava. Prečítanie častí o balíkoch pocprak a JAIDA a o náhodných generátoroch.

Zopakovanie prostredie Java, kompilácia, run, classpath.

Štúdium programu ObeziataData.java, jeho modifikácia, pripravenie vlastných dát do súboru obezita.txt.

Vykreslenie scatterplotu v balíku JAIDA s polotovarom ObezitaJaida.java. Zakreslenie iného scatterplotu inou farbou do toho istého grafu. Zakreslenie "funkcie" ideálna váha ako scatterplotu z veľa bodiek.

Štúdium histogramu gaussovského rozdelenia a jeho fitovanie, výpočet variancie a porovnanie s fitom

Nakreslenie histogramu hmotností, jeho fitovanie jedným a dvoma gaussami

Námet na žolík:

- preloženie najlepšej priamky cez scatterplot

7.4 Úloha. Harmonický oscilátor. Fázový priestor

Riešenie harmonického oscilátora Eulerovou metódou, formou oprava chyby v dodanom polotovare.

Animácia pohybu na osi x.

Fázový priestor ako priestor stavov. Vykreslenie smerníc dotyčníc, geometrická interpretácia Eulerovej metódy. Animácia vo fázovom priestore jedného oscilátora.

Animácia veľkého počtu harmonických oscilátorov vo fázovom priestore, pozorovanie Liouvilleovej vety. Pozorovanie pre sínusovú silu.

Námet na žolík:

- Dvojné kyvadlo metódou Runge-Kutta a jeho animácia. Vyšetrovanie potrebnej presnosti.

7.5 Úloha. Termalizácia harmonického oscilátora

Stredná doba medzi zrážkami.

Zrážka oscilátora s molekulou. Počítačová realizácia zrážky v náhodnom čase. Animácia množiny oscilátorov, pozorovanie termalizácie.

Analýza termalizovaného stavu v priestore kinetických aj potenciálnych energií. Fitovanie Gaussovým rozdelením, meranie teploty.

Pozorovanie relaxačnej doby v závislosti na rozdelení narážajúcich molekúl

Námet na žolík:

- Modelovanie preletu častice plynom, multiscattering.

7.6 Úloha. Kinetická tória plynov

Animácia biliardového stola s jednou časticou. Oprava chyby v polotovare. Biliard v tvare obdĺžnika a lichobežníka.

Registrácia nárazov na vybranú stenu.

Analýza normálovej zložky rýchlosti.

Analýza impulzov sily. Hodnota priemerného impulzu sily. Hodnota tlaku v závislosti na ploche biliardového stola. Boyle-Marriottov zákon.

Analýza tlaku v závislosti na veľkosti počiatočnej rýchlosti častice.

7.7 Úloha. Filter usmerňovača

Riešenie obvodu s kondenzátorom. Voľba správneho znamienka v člene s kondenzátorom.

Numerické riešenie ako integrálna rovnica.

Riešenia vo viacerých situáciách (prechodový jav pri jednosmernom budení.

Striedavé budenie, závislosť na frekvencii, funkcia filtra.

Návrh filtra na výstupe zo štvorcestného usmerňovača.

Pozorovanie brumu nad jednosmernou úrovňou. Voľba parametrov zabezpečujúcich povolenú hodnotu brumu

7.8 Úloha. Fourierova analýza

Fourierov rad na úsečke. Numerický výpočet Fourierových koeficientov pre jednoduché krivky. Vlastné zadanie krivky typu spojitý hrb. Výpočet jej Fourierových koeficientov.

Animácia, znázornenie časového vývoja normálnych módov na úsečke, vlnová rovnica s pevnými koncami.

Animácia, znázornenie časového vývoja vlnového balíka.

Pozdĺžne kmity tyče, animácia pomocou posunutí vybraných zvislých rezov.

Sledovanie zvoleného objemu medzi dvoma zvislými rezmi, pulzácia objemu, sledovanie napätia v tyči.

Okrajová podmienka pre voľný koniec

Normálne módy pre tyč s jedným voľným a jedným pevným koncom

Odraz vlny na pevnom a voľnom konci.

7.9 Úloha. Adjabatický dej

Analytická úloha: nájsť smernicu adiabaty na pV diagrame z podmienky nulovosti tepla

Vykresliť smernice adaibát a izoterm na pV diagrame

Nakresliť sieť izoterm a adiabát.

Krivka zadaná štyrmi bodmi, výpočet práce a tepla na rôznych krivkách so spoločnými koncami. Výpočet prírastku $\delta Q/T$ na tých krivkách.