doi:10.3359/oz1112033

poročilo

KONFERENCA *JAVAONE 2010*

Konferenca JavaOne[1] je največja konferenca s področja jave. V letu 2010 je bila konferenca prelomna. Potem ko je Oracle kupil Sun, so prvič združili konferenci JavaOne in OracleOpenWorld. Če je v prejšnjih letih konferenca JavaOne dajala vtis, da gre za res veliko konferenco (z okoli 12.000 obiskovalci), pa tokrat kot nekakšna podkonferenca konference OracleOpenWorld ni dajala takega vtisa. Konferenco OracleOpenWorld naj bi obiskalo več kot 40.000 obiskovalcev, od teh naj bi jih bilo po nekaterih ocenah približno 5000 namenjenih izključno na JavaOne, kar glede na prejšnja leta kaže, da združitev z OracleOpenWorld ni bila dobro sprejeta.

Združitev obeh konferenc je seveda velik zalogaj in človek se ob obisku ne more načuditi, kako lahko organizatorji izpeljejo nekaj takega. Lokacije posameznih dogodkov so bile razporejene po hotelih v centru San Francisca, glavna predavanja (predvsem konferenca OracleOpenWorld) pa so bila v centru Moscone. Zaradi velikega pretoka udeležencev konference po mestu so celo spremenili promet v samem središču San Francisca. Tako so zaprli štiripasovnico med severnim in južnim delom centra Moscone in tja postavili ogromen šotor. Po mestu so pločnike in ceste opremili z ustreznimi smerokazi. Če smo pred konferenco dvomili, da je možno v odmoru med posameznimi predavanji v nekaj minutah priti z ene lokacije na drugo, pa so zadeve uredili tako, da smo obiskovalci lahko konferenco nemoteno spremljali, čeprav se ni odvijala na eni sami lokaciji. Prav tako kot predavanja so bili tudi konferenčni razstavni prostori na več lokacijah.

Konferenca JavaOne 2010 je ponudila 370 predavanj z veliko področij. Dogodki so potekali v vzporednih sekcijah (tudi do 10 hkrati), zato je bilo seveda nemogoče obiskati vsa predavanja. Iz statistik pa je možno sklepati, da je bila velika večina predavanj namenjena Java EE (80 %), torej strežniškim rešitvam, manj je bilo predstavitev s področja JavaFX in Java ME, torej rešitvam za odjemalce. Vtis je, da bo Oracle na področju jave svojo pozornost usmeril predvsem na strežnik.

Glavna tema konference JavaOne 2010 je bilo računalništvo v oblaku, ki je očitno glavna strateška usmeritev Oracla. Tudi njihov nakup podjetja Sun je verjetno treba gledati s tega vidika, pridobili so znanje o razvoju strežnikov in dobili programski jezik java, v katerem bodo tekle rešitve na teh strežnikih.

Podjetje Oracle je konferenco izkoristilo za to, da so podali jasne strateške in razvojne odločitve za nekaj naslednjih let. Poudarili so, da imajo trenutno največji razvojni potencial v informatiki, saj letno namenijo raziskavam in razvoju celo 4 milijarde dolarjev. Veliko so vlagali v širitev podjetja in v zadnjih letih kupili Sun, PeopleSoft, BEA, Sibel Systems itd. Na ta način so pod eno streho združili večino tehnologij, potrebnih za obvladovanje strežniških tehnologij, in s tem prevzeli vodilno vlogo ponudnika rešitev za računalništvo v oblakih.

Na konferenci so predstavniki Oracla predstavili vizijo razvoja strojne in programske opreme. Z nakupom podjetja Sun so pridobili razvoj strojne opreme, na tej strojni opremi pa bo tekla programska oprema, ki bo v celoti razvita pod njihovo streho. Programska in strojna oprema bosta tako lahko na vseh nivojih optimirani za skupno delovanje. Tako so na konferenci predstavili svoj proizvod Exalogic.[2] Gre za veliko strežniško omaro, v katero je možno namestiti 30 strežnikov s skupno 360 procesorskimi jedri. Programske rešitve temeljijo na javi, optimizacija vseh komponent pa je prinesla izredne karakteristike v zmogljivosti, veliko skalabilnost in seveda možnost virtualizacije. Exalogic stane približno 1 milijardo dolarjev, z dvema pa naj bi bilo možno upravljati z informacijskim sistemom velikosti FaceBooka. Oracle namerava za Exalogic optimirati operacijski sistem Linux, kajti trenutno jedro RedHat jih ne zadovoljuje. Tako razvijajo svoje jedro – Unbreakable kernel, ki naj bi zamenjalo trenutno veljavno.

NOVOSTI V JAVA 7 IN JAVA 8

Zadnja verzija Java JDK, 6.0, je bila izdana leta 2006, medtem ko nove verzije, o kateri se govori že nekaj let, še ni na vidiku. Očitno so spremembe preobsežne, zato se je Oracle odločil, da bo izdal novo verzijo v dveh delih. V letu 2010 je luč sveta uzrla verzija 7.0, ki vključuje vse zaprte projekte, medtem ko bodo nedokončani projekti počakali na verzijo 8.0, ki naj bi izšla predvidoma v letu

2011. Po tem datumu pa naj bi ponovno prešli na 18-mesečni ciklus izdajanja novih verzij. Ključni projekti pri razvoju nove verzije JDK so:

- Projekt Coin – manjše dopolnitve jezika java, katerih namen je povečati produktivnost razvijalcev.
- Projekt Lambda – izrazi Lambda ("closures") so posebni bloki kode, ki jih vključimo v običajno javansko kodo, kar poznamo že v drugih programskih jezikih (npr. Smalltalk).
- Projekt Jigsaw – modularizacija okolja Java (moduli bodo zamenjali datoteke JAR, pot do razredov (angl. *classpath*) ne bo več potrebna).
- Večjedrni procesorji, večji spomin in hitrejše omrežje – gre za izboljšave večnitnega delovanja (npr. Fork/Join Framework), izboljšave v sproščanju pomnilnika (angl. *Garbage Collection*) v primeru zelo velikih kopic (angl. *heap*), izboljšani bodo omrežni vmesniki (Native Infiniband, 10 G Ethernet, SDP & SCTP), na voljo bodo tudi novi vhodni in izhodni (I/O) programski vmesniki.
- Javanski navidezni stroji za različne programske jezike – gre za podporo novim dinamičnim jezikom, povečanje zmogljivosti obstoječih dinamičnih jezikov, boljši JavaScript Engine itd.

PROJEKT COIN

V projektu pod kodnim imenom Coin je potekal razvoj nove sintakse java. Z nekaterimi majhnimi spremembami so poskušali nadgraditi ta programski jezik, tako da bi bil razvoj aplikacij enostavnejši, koda pa bi postala bolj pregledna oz. čitljiva. S temi spremembami naj bi nova sintaksa jave zadovoljila vse razvojne zahteve do leta 2030.

Vnos numeričnih vrednosti

Vnos števil je postal veliko bolj pregleden:

- vrednost celih števil 1346704470 lahko vpišemo kot 1_346_704_470
- heksadecimalno vrednost 0x50451456 lahko vpišemo kot 0x5045_1456 ali 0x50_45_14_56
- binarno vrednost 0b01010000010001010001010001010110 vpišemo kot 0b0101_0000_0100_0101_0001_0100_0101_0110

Stavek "switch"

V stavku za preklop lahko sedaj uporabljamo tudi nize (angl. *string*):

```
int monthNameToDays(String s, int year){
    switch(s){
            case "April":case "June":case
            "September":case "November":
                    return 30;
            case "January":case "March":
            case "May":case "July":
            case "August":case "December":
                    return 31;
            case "February":
                    ...
            default
                    ...
    }
}
```

Inicializacija kolekcij

V novi sintaksi je inicializacija kolekcij preglednejša.

Poenostavitev

```
List<List<String>>monthsInTwoLanguages =
Arrays.asList(Arrays.asList("January",
"February"), Arrays.asList("Gennaio",
"Febbraio"));
```

nadomestimo z

```
List<List<String>>monthsInTwoLanguages=
{{"January", "February"}, {"Gennaio",
"Febbraio"}};
```

V trenutni veziji Jave je inicializacija generičnih kolekcij preveč kompliciranа.

Poenostavitev

```
List<List<List<String>>> list =
new ArrayList<List<List<String>>>();
```

nadomestimo z

```
List<List<List<String>>> list = new
ArrayList<>();
```

Stavek "multi-catch"

Pri lovljenju napak moramo za vsak tip napake vpisati svoj blok kode:

```
try {
    //Reflective operations calling
      Class.forName,
    //Class.newInstance, Class.getMethod,
    //Method.invoke, etc.
} catch (ClassNotFoundException cnfe) {
    log(cnfe);
    throw cnfe;
} catch (InstantiationException ie) {
    log(ie);
    throw ie;
} catch (NoSuchMethodException nsme) {
    log(nsme);
    throw nsme;
} catch (InvocationTargetException ite) {
    log(ite);
    throw ite;
}
```

To lahko poenostavimo z uporabo nadrazreda ***Exception***, kar pa ni priporočljivo. Po novi sintaksi lahko kodo preoblikujemo.

```
try {
    //Reflective operations calling
      Class.forName,
    //Class.newInstance, Class.getMethod,
    //Method.invoke, etc.
} catch (final ClassNotFoundException|
    InstantiationException|
    NoSuchMethodException|
    InvocationTargetException e){
    log(e);
    throw e;
}
```

ali še bolje

```
try {
    //Reflective operations calling
      Class.forName,
    //Class.newInstance, Class.getMethod,
    //Method.invoke, etc.
} catch
    (final ReflectiveOperationException e){
    log(e);
    throw e;//Means ClassNotFoundExcept-
              on or …
}
```

Novost je tudi deklaracija ***final***. Če ni definirana, potem mora biti izjema, ki jo vračamo, istega tipa, kot je definirana v bloku ***catch***, sicer je lahko tudi dedovanega tipa.

AVTOMATSKO UPRAVLJANJE VIROV

Stavek "try-with-resouces"

Pri odpiranju raznih sistemskim virov lahko pride do problemov, ko nam zaradi določene napake ne uspe sprostiti vseh virov. Programska koda za zagotavljanje zanesljivega sproščanja zasedenih virov je zelo zahtevna in nepregledna. Nova verzija jave nam to zelo olajša:

```
try (InputStream in = new
     FileInputStream(src);
          OutputStream out = new
          FileOutputStream(dest)) {
    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) >= 0)
    out.write(buf, 0, n);
}
```

V zgornjem primeru sta znotraj novega stavka ***try*** deklarirana dva vira (***InputStream*** in ***OutputStream***). Oba vira se bosta na koncu zaprla, ne glede na to, ali se bo med izvajanjem zgodila napaka.

PROJEKT JIGSAW

Namen projekta je modularizacija okolja java. Sedaj se ob zagonu enostavne aplikacije, kot je npr. "Hello world", naložijo v pomnilnik vse knjižnice, ki jih aplikacija potrebuje ali ne, kar ima za posledico počasni prenos datotek preko omrežja, počasni zagon ter preveliko porabo pomnilnika (nekaj deset MB). To pa zagotovo ni primerno za majhne naprave (angl. *small devices*), kot so mobilni telefoni itd. Cilji projekta so:

- ukinitev spremenljivke *classpath* in *datoteke JAR,*
- uvedba paketnih modulov za avtomatsko nalaganje in nameščanje,
- označevanje odvisnosti paketov in modulov neposredno v programski kodi,
- prenos (angl. *download*) modulov na zahtevo,
- optimizacija modulov med namestitvijo,
- moduli, primerni tudi za majhne naprave (telefoni ...),
- hitro nalaganje modulov pri zagonu oz. prvi uporabi,
- zagotavljanje predvidljive uporabe paketov (sedaj nam isti razredi v različnih datotekah JAR povzročajo težave),
- kontrola prisotnosti vseh modulov že pri namestitvi in ne šele pri zagonu,
- upoštevanje različnih verzij modulov (odvisnost aplikacije od določene verzije knjižnice),
- vključitev opcijskih ter navideznih modulov.

DATOTEČNI SISTEM

Nova verzija programskega vmesnika za javo bo prinesla veliko izboljšav pri dostopu do datotečnega sistema ***FS*** (angl. *file system*), kar bo odpravilo dosedanje pomanjkljivosti pri uporabi paketa ***java.io***, kot so:

- nekonsistenca med različnimi računalniškimi okolji,
- pri napakah pri dostopu so vržene izjeme neuporabne pri odkrivanju napak,
- ni podpore za osnovne operacije, kot so kopiranje in premikanje datotek,
- omejena podpora za simbolične povezave,
- omejena podpora za atribute datotek, pravice dostopa …,
- problemi zaradi zmogljivosti itd.

Za dostop do datotečnega sistema (*FS*) so dodali nove pakete ***java.nio.file***, ***java.nio.file.attribute*** in ***java.nio.file.spi***. Osnovni razred postaja ***java.nio.file.Path***, ki nadomešča stari ***java.io.File***. Datotečni sistem je predstavljen z vmesnikom ***java.nio.file.FileSystem***, medtem ko ***FileStore*** naslavlja konkreten ***datotečni sistem***, diske ...

Razred Path definiramo podobno kot razred File, dodano pa mu je mnogo novih metod.

Dostop do datoteke za branje:

```
Path file = Paths.get("myFile");
InputStream in = file.newInputStream();
```

Pisanje v datoteko:

```
import static java.nio.file.
StandardOpenOption.*;
Path file = …
OutputStream out = file.
newOutputStream(CREATE, TRUNCATE_
EXISTING);
```

Možno je določiti atribute oz. privilegije pri kreiranju datoteke:

```
Set<PosixFilePermission> perms =
PosixFilePermissions.fromString("rw-r---
--");
Set<OpenOption> opts = EnumSet.
of(CREATE_NEW, WRITE);
OutputStream out = ...
file.newOutputStream(opts,
 PosixFilePermissions.asFileAttribute(perms));
```

Zelo uporabna možnost, ki jo pogrešamo danes, je kopiranje datotek:

```
Path source = …
Path target = …
source.copyTo(target, REPLACE_EXISTING,
COPY_ATTRIBUTES);
source.moveTo(target);
```

Preko razreda FileChanel je omogočeno iskanje po datotekah, asinhrono branje in pisanje, zaklepanje datotek ... Razred ***DirectoryStream*** omogoča zelo hitro sprehajanje po drevesni strukturi, pri čemer porabi malo sistemskih virov. Vgrajeno ima filtriranje po vzorcih ***GLOB*** in ***REGEX*** ter možnost uporabe lastnih filtrov.

Primer sprehajanja z uporabo filtra:

```
Path dir = …
try (DirectoryStream<Path> stream = dir.
     newDirectoryStream("*.java")) {
    for (Path entry: stream) {
     System.out.println(entry.getName());
    }
}
```

Ali:

```
Path dir = …
DirectoryStream.Filter<Path> filter = new
DirectoryStream.Filter<Path>() {
    public boolean accept(Path entry) {
            ...
    }
}
try (DirectoryStream<Path> stream = dir.
     newDirectoryStream(filter)) {
    for (Path entry: stream) {
     System.out.println(entry.getName());
    }
}
```

Files.walkFileTree je vgrajeni iterator, ki omogoča sprehajanje po drevesni strukturi. Določimo mu začetno točko ter poslušalce (angl. *listener*), ki izvedejo operacije glede na trenutni imenik oz. datoteko:

```
interface FileVisitor<T> {
    FileVisitResult preVisitDirectory
    (T dir, BasicFileAttributes attrs);
    FileVisitResult visitFile(T file,
    BasicFileAttributes attrs);
    FileVisitResult vistFileFailed
    (T file, IOException ioe);
    FileVisitResult postVisitDirectory
    (T dir, IOException ioe);
}
```

Močno je razširjena podpora za atribute datotečnega sistema:

```
interface BasicFileAttributes {
    boolean isRegularFile();
    boolean isDirectory();
    boolean isSymbolicLink();
    boolean isOther();
    long size();
    FileTime lastModifiedTime();
    FileTime lastAccessTime();
    FileTime creationTime();
    ...
}
BasicFileAttributes attrs = Attributes.
readBasicFileAttrbutes(file,
 NOFOLLOW_LINKS);
long now = System.currentTimeMillis();
Attributes.setLastModifiedTime(file,
FileTime.fromMillis(now));
```

Podpora za atribute ***POSIX*** (pravice dostopa):

```
interface PosixFileAttributes extends
BasicFileAttributes {
    UserPrincipal owner();
    GroupPrincipal group();
    Set<PosixFilePermission> permissions();
}
PosixFileAttributes attrs = Attributes.
readPosixFileAttrbutes(file);
Set<PosixFilePermission> perms
    = EnumSet.of(OWNER_READ,
OWNER_WRITE, GROUP_READ);
Attributes.setPosixFilePermissions(file,
perms);
```

WatchService omogoča registracijo in spremljanje sprememb v datotečnem sistemu. Ob spremembi registriranih poslušalcev ***Watchable*** se sproži dogodek ***WatchEvent***.

Primer registracije poslušalca za kreiranje oz. brisanje datotek v danem imeniku:

```
WatchService watcher = FileSystems.
getDefault().newWatchService();
Path dir = ...
WatchKey key = dir.register(watcher,
ENTRY_CREATE, ENTRY_DELETE);
```

Spremljanje sprememb:

```
for (;;) {
    WatchKey key = watcher.take();
    for (WatchEvent<?> event: key.
         pollEvents()) {
      if (event.kind() == ENTRY_CREATE){
        Path name = (Path)event.context();
        System.out.format("%s created%n",
        name);
      }
    }
    key.reset();
}
```

Definiramo lahko druge ponudnike datotečnih sistemov. Primer dostopa do arhiva ***ZIP***:

```
Path zipfile = …
Map<String,?> env = Collections.
emptyMap();
FileSystem zipfs = FileSystems.
newFileSystem(zipfile, env, null);
Path top = zipfs.getPath("/");
```

JDK je končno dobila močno orodje za manipulacijo datotečnega sistema, ki je enostavno za uporabo ter razširljivo. Sedanji razred ***java.io.File*** lahko enostavno pretvorimo v novi ***java.nio.file.Path*** z metodo ***File.toPath()***.

OPTIMIZACIJA JAVE

Za dobro uporabniško izkušnjo je zelo pomembno načrtovanje arhitekture aplikacije, saj je pravilna arhitektura ključnega pomena za hitrost delovanja, zagona, porabo pomnilnika in razširljivost (skalabilnost) aplikacije. Ponavadi se posamezne kategorije izključujejo, saj povečanje hitrosti delovanja pomeni hkrati tudi večjo porabo pomnilnika. Obstajajo strategije, kako načrtovati aplikacije v java za dosego dobre uporabniške izkušnje ob hkratni skromni uporabi računalniških virov. Na konferenci se je veliko predavateljev ukvarjalo z vprašanji o optimizaciji jave, o delovanju sprostilca pomnilnika (**garbage collector – GC**), orodjih za nadzor aplikacij in optimizacijo programske kode (Java profiler).

Vhodno-izhodna zmogljivost (I/O Performance)

Branje in pisanje datotek je lahko časovno zelo potratno opravilo. Za te potrebe se v javi uporabljata abstraktna razreda ***java.io.InputStream*** in ***java.io.OutputStream***, ki zagotavljata dostop do različnih tipov podatkov, kot so diski in mreža. Paket ***java.io*** vključuje mnoge filtre, ki nadgradijo osnovne podatkovne tokove. Med pomembnejšimi tipi filtrov sta ***BufferedInputStream*** ter ***BufferedOutputStream***.

Današnji diski so zelo zmogljivi pri branju velikih količin (blokov) podatkov, toda zelo neučinkoviti pri branju velikega števila majhnih kosov podatkov. Za pospešitev branja oz. pisanja je zelo pomembno predpomnjenje podatkov in prav tu se izkažejo filtri za predpomnjenje. Primer branja datoteke v javi:

```
InputStream in = null;
try {
    in = new FileInputStream(fileName);
    while (true) {
           int data = in.read();
           if (data == -1) break;
           ...
    }
} catch (IOException) {
    if (in != null) in.close();
}
```

Z uporabo filtra lahko branje take datoteke pohitrimo za nekaj velikostnih razredov:

```
InputStream in = null;
try {
    InputStream inFile =
     new FileInputStream(fileName);
    in = new BufferedInputStream(inFile);
    ...
```

Še dodatno lahko pospešimo vhodne in izhodne opracije z uporabo lastnega predpomnilnika. Velikost predpomnilnika moramo določiti glede na pričakovano velikost naših podatkov:

```
static final in BUF_SIZE = 100000;
static binal byte[] buffer = new
byte[BUF_SIZE];
InputStream in = null;
try {
    in = new FileInputStream(fileName);
    while (true) {
           synchronized (buffer) {
                  int amountRead =
                  in.read(buffer);
                  if (amountRead == -1)
                  break;
                  ...
```

Serializacija

Pri prenosu objektov preko mreže (angl. *remote method invocation*) ali pri shranjevanju na disk se izvede serializacija oz. deserializacija objektov. Serializirajo se vsi atributi, ki niso tipa ***transient*** oz. ***static***. Po deserializaciji so transientni atributi prazni (privzete vrednosti). Da bi povečali hitrost prenosa, naj bo čim več atributov transientnih, le-te pa inicializiramo pri deserializaciji objekta. To naredimo tako, da povozimo metodo nadrazreda ***java.lang.Object*** in v njej izvedemo inicializacijo teh atributov:

```
public void readObject(ObjectInp
utStream in) throws IOException,
ClassNotFoundException
{
    in.defaultReadObject();
    initTransientAttributes();
}
```

Poraba pomnilnika

Pri delovanju uporablja aplikacija delovni pomnilnik RAM, v katerem so shranjeni vsi podatki in koda, ki jo program potrebuje. Pri slabo napisani aplikaciji se količina porabljenega pomnilnika s časom povečuje, kar nakazuje na to, da se sistemski viri po uporabi ne brišejo. Porabo pomnilnika lahko preverimo z dvema vgrajenima metodama ***Runtime.totalMemory()*** in ***Runtime.freeMemory()***. Prva metoda nam poda celotno razpoložljivo količino pomnilnika, ki ga lahko zasedajo objekti in drugi podatki aplikacije, medtem ko nam druga metoda vrne količino prostega pomnilnika. Pri preverjanju pomnilnika v **Windows Task Manager** opazimo, da je poraba pomnilnika veliko večja, kot nam prikažejo metode v javi; te namreč kažejo samo porabo, ki ga zasedajo objekti.

K celotni porabi pomnilnika prispevajo:

- Objekti
  To je pomnilnik, ki ga prikažejo javine metode ***Runtime.totalMemory()*** in ***Runtime.freeMemory()***. Prav pri tej komponenti lahko največ pripomoremo k optimizaciji porabe pomnilnika. Primeri porabe pomnilnika za tipične razrede java:

Tip	Velikost
Byte	1 bajt
Char	2 bajt
Short	2 bajt
Int	4 bajt
Float	4 bajt
long	8 bajt
double	8 bajt
reference	4 bajt
java.lang.Object	8 bajt
java.util.Hashtable	96 bajt
javax.swing.JTextField	3109 bajt
javax.swing.JTable	4086 bajt

- Razredi
  Tu je vključena prevedena koda v bajtih (datoteke tipa ***.class***). Ko se prevedena koda pri zagonu naloži, se kreirajo metapodatkovne strukture, ki jih uporablja javanski paket ***java.lang.reflect***. Tudi ti zasedejo določeno količino pomnilnika. Vse konstante programa se naložijo v ta pomnilnik. In nazadnje še prevajalnik ***JIT*** doda svojo nativno prevedeno kodo.
- Niti
  Niti uporabljajo določene sistemske vire, toda prednosti njihove uporabe so take, da se s porabo virov ne obremenjujemo.
- Nativne podatkovne strukture
  Java uporablja nativne knjižnice operacijskega sistema *AWT*, ki na primer uporablja ogromno nativnih podatkovnih struktur.
- Nativna koda
  Java uporablja sistemske knjižnice DLL za dostop do mreže, diskov in drugih sistemskih virov.

Razredi se naložijo v pomnilnik, ko jih prvič potrebujemo oz. se na njih sklicujemo. Dejansko pa jih včasih sploh ne potrebujemo, pa se vseeno inicializirajo in trošijo sistemske vire:

```
public static Translator
getTranslator(String fileType) {
    if (fileType.equals("doc")) return
        new WordTranslator();
    else if (fileType.equals("html"))
            return new HTMLTranslator();
    ...
```

Ne glede na tip dokumenta se v prejšnjem primeru naložijo vsi razredi. Ta problem lahko zaobidemo z uporabo paketa ***java.lang.reflect***:

```
public static Translator
getTranslator(String fileType) {
    if (fileType.equals("doc")) {
          return (Translator) Class.
          forName("WordTranslator").
          newInstance();
    } else if (fileType.equals("html")) {
          return (Translator) Class.
          forName("HTMLTranslator").
          newInstance();
    ...
```

Tu se dejansko naloži le tisti razred, ki ga res potrebujemo. Količino naloženih razredov lahko zmanjšamo tudi tako, da zaženemo več aplikacij znotraj istega navideznega stroja (***JVM***), kar pa je po našem mnenju zelo redka uporaba:

```
Class clazz = Class.forName(className);
Class[] argsTypes = new {String[].class};
Object[] args = {new String[0]};
Method method = clazz.getMethod("main",
argsTypes);
method.invoke(clazz, args);
```

Vsako tako aplikacijo je seveda treba zagnati v svoji niti.

Odziven grafični vmesnik

Za prijetno delo je zelo pomemben odziven uporabniški vmesnik, saj dajejo že zakasnitve nad 50 milisekund občutek počasnega vmesnika. Trajajoče operacije je treba izvajati v ločenih nitih, saj tako zakrijemo slabo odzivnost. Lahko pa uporabniku prikažemo potek izvajanja operacije s ***ProgressBar***, s čimer ga tudi nekoliko pomirimo. Pri delu z nitmi znotraj grafičnega vmesnika je treba biti zelo pazljiv, saj se lahko pojavijo nezaželene napake pri prikazu. Celoten grafični vmesnik se v javi izvaja znotraj ene same niti ***event-dispatching thread*** (***EDT***). S tem je zagotovljen lažji nadzor nad risanjem vmesnika in pravilnim zaporedjem izvajanja proženih dogodkov (***event-handling***). Proženje dogodkov oz. risanje izven niti ***EDT*** lahko povzroči nepravilno risanje in osveževanje grafičnih elementov aplikacije. Privede pa lahko tudi do nepravilnega zaporedja sproženih dogodkov. Delo z vmesnikom v drugih nitih moramo izvesti z uporabo metod ***javax.swing.SwingUtilities.invokeLater()*** in ***.invokeAndWait()***, ki poiščeta nit ***EDT*** in zaženeta želeno operacijo znotraj te niti. Prva metoda se izvede, ko ima sistem čas, medtem ko se druga metoda izvede takoj. Primer dela z grafičnim vmesnikom znotraj drugih niti:

```
Runnable doWork = new Runnable() {
    public void run() {
          // do some GUI work
    }
};
SwingUtilities.invokeLater(doWork);
```

Pri morebitnih problemih z osveževanjem oken je najlažje preveriti, ali smo res v niti ***EDT*** z uporabo metode ***SwingUtilities.isEventDispatchThread()***. Ker je delo z nitmi v grafičnem vmesniku zahtevno, so v Javi vključili poseben vmesnik ***javax.swing.SwingWorker***, ki zelo olajša delo z nitmi. Primer uporabe ***SwingWorker***:

```
final JLabel label;
class MeaningOfLifeFinder extends
     SwingWorker<int, Object> {
  @Override
  public int doInBackground() {
      for (int i=1; i<42; i++) {
            publish(i);
```

```
        }
        return 42;
    }
    @Override
    protected void process(List<int>
    numbers) {
        for (int number : numbers) {
                label.setText("Number " +
                number + " is wrong answer");
        }
    }
    @Override
    protected void done() {
        try {
                label.setText(get() + " is
                the answer to the ultimate
                question " + "concerning
                life, the universe, and
                everything");
        } catch (Exception ignore) {
        }
    }
}
(new MeaningOfLifeFinder()).execute();
```

Metoda ***doInBackground()*** je časovno potratna in se izvaja zunaj niti ***EDT***. Z metodo ***publish()*** sprožimo metodo ***process()*** znotraj niti ***EDT***, ki osvežuje oznake grafičnega vmesnika. Ko metoda ***doInBackground()*** zaključi delo, pokliče metoda niti ***EDT done()***. Ta izvede zadnje popravke grafičnega vmesnika.

Časovno potratne operacije lahko preložimo na kasnejši čas (nočni čas) z uporabo časovnikov, ki jih napišemo sami z uporabo niti ali pa uporabimo že pripravljene vmesnike ***javax.Swing.Timer*** in ***java.util.Timer***. Prvi vmesnik se vedno izvaja znotraj niti ***EDT*****,** medtem ko vsaka vključitev drugega vmesnika tvori novo nit.

Primer uporabe prvega vmesnika, ki izvede utripanje kurzorja na vsakih 300 milisekund:

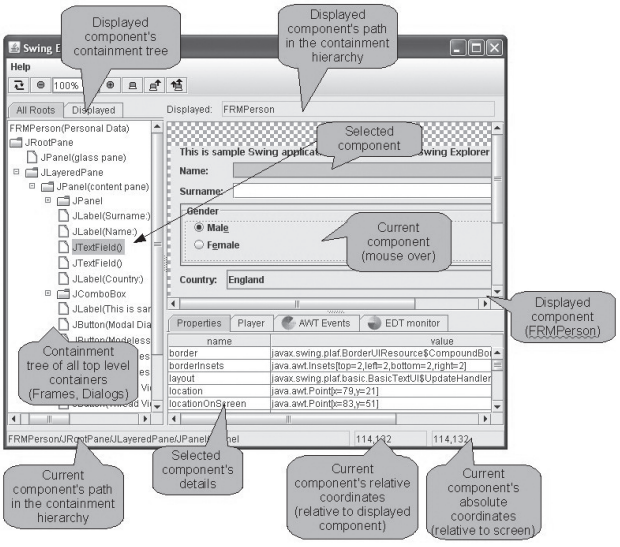
```
int delay = 300; //milliseconds
ActionListener taskPerformer = new
ActionListener() {
    public void actionPerformed(ActionEv
                ent evt) {
            //...Perform redraw cursor...
    }
};
new Timer(delay, taskPerformer).start();
```

Swing Explorer

Swing Explorer je češko odprtokodno orodje za razhroščevanje napak v vmesnikih Swing. Prenesemo si lahko vtičnik za okolje NetBeans ali Eclipse. Stvar deluje dobro (Swing Explorer plug-in 1.4 za Eclipse), le hitrost delovanja je bolj slaba. V Eclipsu se v zagonih "run/debug" pojavi možnost za zagon aplikacije znotraj Swing Explorerja. Za uporabo tega grafičnega orodja je potrebno dobro poznavanje arhitekture Swing. Z orodjem lahko:

- raziskujemo interno delovanje Swinga,
- se sprehajamo po hierarhiji komponent in podlag (angl. *layout*),
- raziskujemo izrisovanje komponent (L&F),
- določimo natančne velikosti in pozicije komponent,
- analiziramo probleme z nitjo Event Dispatch,
- sledimo dogodkom AWT (poslušalci mouse/key/focus …).

Swing Explorer je zelo močno orodje pri razvoju uporabniških vmesnikov. Težje pa ga je izkoristiti pri generični gradnji grafičnega vmesnika, kot na primer našega GDialoga.



Slika 1: Uporabniški vmesnik Swing Explorer

Garbage collector

Sprostilec pomnilnika (garbage collecotr – ***GC***) je zadolžen za zbiranje in odstranjevanje virov (objekti, povezave ...), ki jih aplikacija ne potrebuje več. Na to, kdaj se zaganja in kako deluje, nimamo velikega vpliva. Paziti moramo le na to, da pravilno sproščamo neuporabljene vire, da jih lahko ***GC*** odstrani.
Pri velikih količinah alociranega in sproščenega pomnilnika lahko pride v delovanju aplikacije do opaznih zakasnitev, saj mora ***GC*** preveriti ogromno količino povezav do objektov in ugotoviti, ali lahko

objekt odstrani ali ne. Da bi bili ti zamiki čim krajši, ***GC*** razbije trajajoče opravilo na več majših delov, kar ima za posledico bolj tekoče delovanje aplikacije. Spodaj je opisan življenjski krog nekega objekta:

- kreiran,
- v uporabi,
- neviden,
- nedosegljiv,
- v izboru sprostilca pomnilnika (GC-ja),
- finaliziran,
- dealociran.

Kreiran objekt

Pri ustvarjanju novega primerka razreda se izvedejo naslednji postopki:

- alocira se pomnilnik za novi objekt,
- izvedejo se konstruktorji nadrazredov,
- inicializirajo se spremenljivke,
- izvede se konstruktor.

Na novo kreiran objekt se dodeli neki spremenljivki in tako preide v stanje "V uporabi".

Objekt v uporabi

Objekt je lahko referenciran iz spremenljivke na ***stack***-u ali iz statične spremenljivke na kopici (pomnilnik ***heap***). Povezava iz ***stack***-a se prekine, ko spremenljivka ni več uporabna oz. trenutna nit zaključi delo in se zapre. Povezave iz statičnih spremenljivk prekinemo tako, da to spremenljivko postavimo na ničto vrednost (***null***).

Nevidni objekt

Objekt je neviden, ko je še vedno v uporabi, a ga ni mogoče doseči. Primer:

```
public void run() {
    try {
          Object foo = new Object();
          foo.doSomething();
    } catch (Exception e) {
    }
    while (true) { // do stuff }// loop
          for ever
}
```

Znotraj neskončne zanke objekt "foo" ni viden, sprosti pa se šele po izhodu iz metode. Taki nevidni objekti lahko povzročijo nenadzorovano naraščanje porabe pomnilnika.

Nedosegljivi objekt

Objekt je nedosegljiv, ko na njega ne kaže nobena močna povezava. Na objekt ne kaže nobena začasna spremenljivka iz ***stack***-a, nobena statična spremenljivka in nobena referenca iz nativne kode. To vse so tako imenovane spremenljivke ***GC root***. Na objekt lahko kaže močna referenca iz drugih objektov, toda noben od teh objektov ne sme biti referenciran iz spremenljivk ***GC root***. Tak objekt je kandidat, da ga ***GC*** odstrani. ***JVM*** odstrani objekt, ko potrebuje pomnilnik, ki ga ta objekt zaseda.

Objekt v izboru GC

Objekt je v tem stanju, ko je sprostilec pomnilnika – ***GC*** ugotovil, da je objekt kandidat za brisanje. Če ima objekt metodo ***finalize()***, jo izvede in preide v stanje "finaliziran".

Finaliziran objekt

Objekt je v tem stanju, če je po izvedbi metode ***finalize()*** še vedno nedosegljiv. Metoda ***finalize()*** ni priporočljiva, ker samo podaljša čas obstoja objekta ali ga celo naredi zopet vidnega.

Dealociran objekt

To je zadnje stanje objekta, ko ***GC*** sprosti pomnilnik, ki ga je zasedal ta objekt. Kdaj se to zgodi, pa je odvisno od same izvedbe ***GC***-ja.

Z Java 2.0 so se poleg močnih referenc pojavile v paketu ***java.lang.ref*** še reference ***SoftReference***, ***WeakReference*** in ***PhantomReference***. S temi novimi referencami je omogočeno vsaj minimalno komuniciranje z ***GC***-jem.

GWT (GOOGLE WEB TOOLKIT)

Na področju spletnih odjemalcev je stanje še vedno tako, da obstaja vsaj 100 različnih ogrodij za razvoj spletnih aplikacij na osnovi jave. Google je s svojo ponudbo **GWT** (Google web Toolkit, http://code.google.com/intl/sl-SI/webtoolkit/) predstavil nov pristop k razvoju spletnih odjemalcev. Gre za ogrodje, ki ga lahko uporabimo v okolju Eclipse ali katerem drugem IDE za javo.

GWT naj bi omogočil razvoj kompleksnih spletnih dinamičnih aplikacij, ki temeljijo na tehnologiji AJAX. Pri tem lahko programer aplikacijo v popolnosti razvije v javi, pri čemer naj se ne bi srečeval s kodo HTML in JavaScript. Seveda je v praksi potrebno zelo dobro poznavanje vseh spletnih tehnologij. Programski vmesnik GWT pa je res tak, da se aplikacija razvije v javi. GWT celotno aplikacijo prevede v JavaScript, pri čemer zagotavlja delovanje v vseh obstoječih spletnih odjemalcih.

JAVAFX

Ena glavnih zamer JavaFX[3] je bila ta, da je uvedel nov skriptni jezik. Google je s svojim GWT-jem pokazal, da je možno ponuditi ogrodje za razvoj spletnih aplikacij z javanskim programskim vmesnikom. Tako so tudi pri Oraclu napovedali,[4] da bo aplikacije v JavaFX v bodoče možno v celoti razvijati z javanskim programskim vmesnikom. Novosti se načrtujejo za drugo polovico leta 2011, novi JavaFX pa velja med razvijalci za poskus razvoja nove verzije Swinga. Z JavaFX bi bilo možno razvijati odjemalce tako za splet kot za nativne odjemalce (namizje).

JAVA IN MOBILNE NAPRAVE

Java ME[5] je verzija jave, ki že od leta 1998 pokriva področje mobilnih naprav. Statistike kažejo uporabo Java ME:

- 5 milijard kartic za telefone (SIM),
- 3 milijarde mobilnih telefonov,
- 80 milijonov TV,
- vsi Blue-ray predvajalniki.

Nadaljnji razvoj jave za mobilne naprave naj bi šel v smeri večje povezanosti s spletnimi tehnologijami (HTML, CSS, JavaScript). Dodali bodo vmesnike za dostop do strojne opreme. Na ta način želijo Java ME postaviti ob bok okoljem Android in IPhone, ki v tem trenutku osvajata trg mobilnih aplikacij.

WEB SOCKETS

Med zelo zanimivimi novostmi, ki prihajajo s standardom HTML5, je dvosmerni prenos podatkov preko ene vtičnice TCP. To omogoča obojestransko komunikacijo med spletnim strežnikom in njegovim odjemalcem. Brez dvosmerne komunikacije med strežnikom in odjemalcem je težko graditi dinamične spletne aplikacije. WEB sockets[6] je standard, ki naj bi poenostavil uporabo dvosmerne komunikacije med strežnikom in odjemalcem. Nadomestil naj bi AJAX in še nekatere druge pristope, ki se trenutno uporabljajo (npr. COMET).

NOVOSTI V JPA

JPA (Java Persistence API) je programski vmesnik za dostop do baze podatkov. Zanimive novosti v JPA, 2.0, so:

- razširitev preslikave med objekti in relacijami,
- nov programski vmesnik za povpraševanje,
- standardizacija namigov za iskanje.

Referenčna implementacija JPA, 2.0, je ExlipseLink (http://www.eclipse.org/eclipselink/).

Opombe

1 http://www.oracle.com/us/javaonedevelop/index.html
2 http://www.oracle.com/us/products/middleware/exalogic/index.html
3 http://javafx.com/
4 http://javafx.com/roadmap/#general
5 http://download.oracle.com/javame/
6 http://dev.w3.org/html5/websockets/

Vojko Ambrožič, Robert Vehovec