SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ZAVOD ZA ELEKTRONIČKE SUSTAVE I OBRADBU INFORMACIJA

Podatkovni višemedijski prijenos i računalne mreže

Seminar

Java Applet

Davor Perišić

21. sječanj 2005.

Sadržaj

1. Uvod

1.1 Što je *Java*

Java je danas već vrlo poznat i raširen, objektno orijentiran programski jezik visoke razine. Osim što je jednostavna, robusna i sigurna, Java je i neovisna o platformi na kojoj se izvodi. Tako se za Javu kaže: *"write once, run anywhere"*, što znači da kada kompajliramo program generiramo *Java bytecodes* što su u stvari instrukcije za Javinu platformu - Java Virtual Machine (Java VM). Ukoliko na svom operacijskom sustavu (bio to Windows, UNIX, MacOS ili neki Internet browser) imamo Java VM, on će razumjeti *Java bytecodes* te će ih interpretirati svaki puta kada se program izvrši.

Za `Javu` danas možemo reći da je:

1. specifikacija programskog jezika i standardni zbir klasa
2. implementacija navedenog programskog jezika i njegovih pratećih datoteka (*libraries*) u okolici prevođenja i izvođenja (*compile and run time enviroment*) za izradu i izvršavanje aplikacija
3. implementacija navedenog programskog jezika kao podskup ugrađenog koda u `HTML` stranicama (*applet*)
4. implementacija navedenog programskog jezika kao dodatak animaciji i interakciji kod `3D` objekata i scena (VRML 2.0)

Možemo tvrditi da svaku od danih podstavaka prezentira drugačija implementacija `Jave`. Svaka od njih ima svojih prednosti kao i ograničenja.

1.2 *Java* Platforma

Platforma je hardversko ili softversko okruženje na kojem se pokreće program. To su npr. Windows, Linux, Solaris i MacOS. Većina platformi može se opisati kao kombinacija operacijskog sustava i hardvera. Java platforma se razlikuje od drugih po tome što je ona samo softverska platforma koja se izvršava na nekoj od drugih platformi baziranih na hardveru.
Java platforma se sastoji od dvije komponente:

- The Java Virtual Machine (Java VM)
- The Java Application Programming Interface (Java API)

Java VM - je glavni dio Java platforme i prenosi se na različite platforme bazirane na hardveru.

Java API - je velika kolekcija već gotovih softverskih komponenti koje pružaju mnoge mogućnosti. Java API je grupiran u biblioteke povezanih klasa i sučelja - pakete (engl. packages).

1.3 Aplikacije, apleti i skriptni jezici

Najuobičajeniji Java programi su appleti i aplikacije.
Applet općenito predstavlja mali klijent (client-side) program koji se preuzima (download) i izvršava na klijentovom (korisnikovom) računalu, obično iz web čitača, a ne na serveru.

Razlika između apleta i standardnog programa (aplikacije) svodi se prvenstveno na način kako se oni izvode:

- standardni programi izvode se pomoću Java interpretora (Java VM),
- apleti se izvode u pretraživaču koji podržava Javu (Netscape Navigator, Internet Explorer, HotJava).

Appleti su zapravo mali djelovi Java koda, mali moduli koji se pozivaju iz nekog višeg programskog modula (više klase), tj. iz web starnice unutar web čitača.
Za razliku od Java Appleta, Java samostalne aplikacije moraju obavezno sadržavati metodu **main()** s kojom uvijek počinje izvođenje programa. Apleti ne moraju imati metodu **main()**. Umjesto nje u apletima postoji nekoliko drugih metoda koje mogu biti pozvane u različitim trenucima izvođenja apleta.

S druge strane skriptni jezici su ugrađeni u web čitače uz postojeći HTML, te predstavljaju njegovo svojevrsno proširenje. Skriptni jezici dolaze u čitač u izvornoj formi, u obliku teksta zajedno s ostalim HTML kodom, za razliku od Java applet-a koji dolaze u prevedenoj formi (bytecode).
Tako uz Java aplikacije i Java aplete postoji i JavaScript. JavaScript je Netscape-ova dopuna osnovnog HTML jezika i osim imena nema prevelike veze s Javom. To je skriptni jezik, koji je ugrađen u HTML dokument. Iz te činjenice slijedi velika brzina izvođenja rutina pisanih pomoću skriptnih jezika, ali i manji nedostatak, a to je da je kod vidljiv svima.
JavaSkript kao i ostali skriptni jezici manjih su mogućnosti od apleta i aplikacija, ali zato omogućuju jednostavniji i brži razvoj interaktivnih grafičkih korisničkih sučelja.

2. *Java* kao jezik za pisanje aplikacija

Osim kao dodatak vlastitim *Web HTML* stranicama u svrhu obogaćivanja stranica animacijama i raznim interaktivnim detaljima, *Javu* možemo koristiti isto kao i *C++* za pisanje *stand-alone* aplikacija. Postupak je sljedeći:

1. prevodimo Java izvorni kod u tzv. *bajtni kod*
2. izvršavamo *bajtni kod* interpretiranjem unutar *JVM* (*Javine Virtualne Mašine*)

2.1 Java Virtual Machine

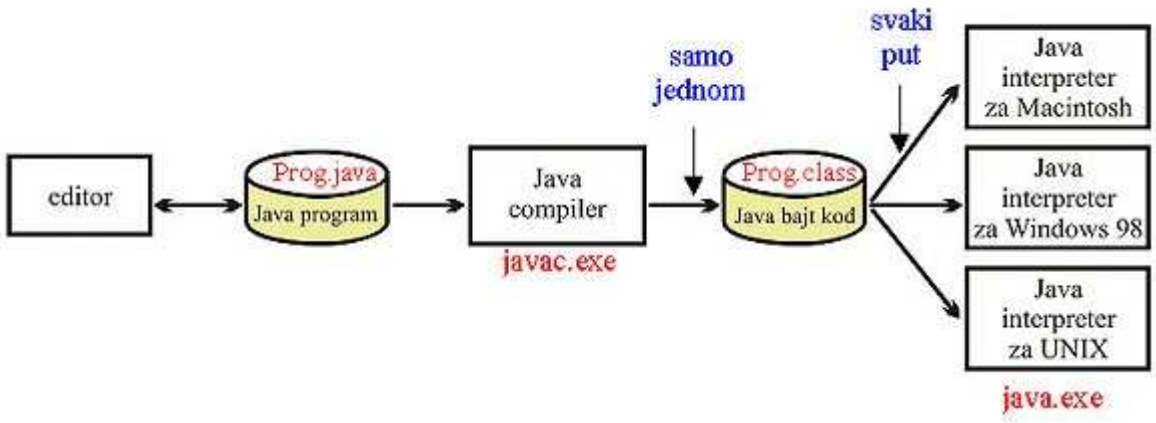
U pozadini svake Javine aplikacije nalazi se JVM (Java Virtual Machine). Kad se jednom prevede Javin kod u *.class* datoteke i ukljuci se u JAR (Java Archive file) datoteku, JVM prevodi *class* datoteku (tocnije *bytecode Java class* datoteke) u strojni jezik prilagoden platformi na kojoj radi JVM. JVM je odgovoran za osiguravanje sigurnosti, alociranje i oslobadanje memorije. Ukratko, omogucava Java programu da radi.

Projektanti Jave su se odlučili na korištenje kombinacije kompilacije i interpretiranja. Programi pisani u Javi se prevode u strojni jezik, ali u strojni jezik računala koje zapravo ne postoji. Ovo takozvano prividno (*virtual*) računalo se zove "*Java Virtual Machine*".

Strojni jezik za Java Virtual Machine se zove Java bytecode. Nema razloga zbog kojega Java bajt kod ne bi mogao biti korišten kao strojni jezik i nekog stvarnog računala, osim ovog prividnog. Zapravo, Sun Mycrosystems, začetnik Jave, razvio je CPU-ove koji izvršavaju Java bajt kod kao svoj strojni jezik.

Ipak, jedna od glavnih prednosti Jave je da zapravo može biti korištena na bilo kojem računalu. Sve što je na tom računalu potrebno je interpreter za Java bajt kod. Takav interpreter oponaša Java virtual machine na isti način kao što prividno računalo oponaša osobno računalo.

Naravno, Java interpreter je potreban za svaku vrstu računala, ali nakon što računalo dobije Java bajt kod interpreter, može izvršavati bilo koji Java bajt kod program. A isti taj Java bajt kod program može biti izvršen na bilo kojem računalu koje ima takav interpreter. Ovo je jedna od glavnih osobina Jave: isti kompilirani program se može izvršavati na više različitih vrsta računala.



Slika 2.1.1 Java VM

Postavlja se pitanje zašto uopće koristiti prijelazni Java bajt kod? Zašto se ne bi isporučivao izvorni Java program pa da ga svako kompilira za sebe u strojni jezik računala na kojem ga želi koristiti?
Prvi od mnogo razloga je da kompiler mora razumijeti Javu, složeni jezik više razine. Kompiler, sam po sebi, vrlo je složen program, dok je, za razliku od njega, Java interpreter prilično mali, jednostavan program. Zbog toga je jednostavnije napisati interpreter za novu vrstu računala, a jednom kad je urađeno, to računalo može pokretati bilo koji kompilirani Java program. Sa druge strane, bilo bi mnogo složenije napisati Java kompiler za isto računalo.
Osim toga, mnogi Java programi su namijenjeni dohvaćanju preko mreže. Ovo vodi do očitih sigurnosnih pitanja: korisnik ne želi spustiti i pokrenuti program koji bi mogao nanijeti štetu njegovom računalu ili podacima. Java interpreter služi kao međuspremnik između korisnika i spuštenog programa. Korisnik zapravo pokreće interpreter koji neizravno izvršava dohvaćeni program. Interpreter može zaštititi korisnika i računalo od moguće opasnih radnji tog programa.

2.2 Kako rade *Java* aplikacije

Primjer aplikacije koja ispisuje string "Ovo je prva alikacija":

Izvorni kod:

```
class JavaAplikacija {

       public static void main (String arg[] ) {
          System.out.println ("Ovo je prva aplikacija");
       }
}
```

Struktura programa:

```
class JavaAplikacija{
}
```

Deklaracija nove klase JavaAplikacija - osnovnog objekta koji čini jedan program. Tijelo klase omeđeno je vitičastim zagradama { }.

```
class JavaAplikacija extends Object {
```

Sve klase nasljeđuju svoja svojstva od osnovnog objekta, Object s mogućnosti dodavanja novih. Nije potrebno pisati jer se podrazumjeva!

```
public static void main (String arg[] ){
}
```

Deklaracije osnovne metode. Kod samostalnih aplikacija metoda main mora obavezno postojati! U Java programu, **main()** je prva metoda koja se izvršava kod pokretanja programa.

Modifikatori:

public – sve metode iz svih ostalih klasa u programu mogu pristupati ovoj metodi
static – metoda je nepromjenjiva, tj. niti jedna metoda ju ne može zamjeniti svojom deklaracijom – u nekoj drugoj klasi nije moguće ponovno definirati metodu istog imena i korisititi ju umjesto ove
void – metoda ne vraća nikakve vrijednosti metodi iz koje je pozvana; Ispis na ekran ne smatra se vraćanjem vrijednosti.

```
(String arg[])
```

Parametri koje metoda prima – u ovom primjeru prima jedan parametar tipa string

```
System.out.println ("Ovo je aplikacija");
```

Poziva se metoda println iz klase System.out. System je glavni objekt, a out njegov podobjekt koji sadrži metodu println. Točkice se koriste za pristup varijablama i metodama sadržanim u objektima

`{ }` - početak i kraj programskog bloka
`;` - sve naredbe (osim iznimaka) završavaju točkom-zarez- ona označava kraj naredbe

Izvorni kod sprema se u datoteku čije ime mora biti jednako imenu prve javne klasa, dakle za ovaj primjer **JavaAplikacija.java**. Program treba kompajlirati - prevesti u baytecode zapis, pri čemu se za svaku deklariranu klasu u izvornom kodu formira datoteka koja će nositi ime **ime_klase.class** (u ovom primjeru stvoriti će samo jednu klasu **JavaAplikacija.class**). Kod pokretanja interpreter poziva aplikaciju u bytecode zapisu (*.*class*) i izvršava je.

Vidimo da je po strukturi implementacije programskog jezika *Java* negdje na prijelazu između dosad kristalno jasnih definicija pojmova *kompajler* i *interpreter*. Ona spada u obje klasifikacije dok uistinu nije potpuno niti jedna od njih.
Sâm *bajtni kod* je dosta manji od ekvivalentnog izvršnog koda, recimo, *C*-a, ali brzina njegovog interpretiranja daleko zaostaje naspram brzine izvršavanja ekvivalentnih programa pisanih u *C*-u. Alternativno rješenje je u korištenju tzv. *just-in-time* interpretiranja, gdje *Javina Virtuelna Mašina* prebacuje *bajtni kod* u *native kod* prije samog izvršavanja. To za manje aplikacije pruža razumnu performansu uz zadržanu prenosivost izvršnog koda.

Međutim, čak i toliko naglašavana prenosivost možda ne počiva na čvrstim temeljima. Iole veće aplikacije će vjerojatno uslijed ograničenja nametnutih standardnim *Java* okruženjem biti prisiljene koristiti vlastita korisnička sučelja u vidu *native library* datoteka (*JNI*) čijim se funkcijama proširuju mogućnosti *Jave*. U tom slučaju mada je osnovna aplikacija prenosiva, morat će se prilagođavati od operativnog sistema do različitih arhitektura računala *library* datoteke pisane u *C*/*C++*-u bez kojih osnovna aplikacija neće moći raditi.

3. *Java* kao HTML proširenje (*Applet*)

Applet je mala aplikacija koja je zamišljena da se ne izvršava samostalno, nego unutar druge aplikacije, tipično unutar web čitača. Aplet se skida sa poslužitelja kao dio web stranice (kao što se primjerice slike skidaju zajedno sa stranicom). U web čitaču aplet se aktivira, te pokreće program. Apleti pružaju mogućnost automatske rapodjele klijentskih programa sa servera baš u trenutku kada ga klijent treba, ne prije. Korisnik dobiva najsviježiju verziju klijentskog programa bez kompliciranе instalacije. Zbog samog načina na koji je Java oblikovana, programer treba napraviti samo jedan program i taj program automatski funkcionira na svim računalima na kojima je instaliran web čitač s ugrađenim Java interpreterom (većina računala danas). Isto tako, na klijentu se obavlja što je moguće više posla, prije i nakon slanja zahtjeva na poslužitelj.

Jedna prednost koju Java aplet ima nad skriptnim programom je to što je aplet u prevedenom (kompajliranom) obliku, pa izvorni kod (*source code*) nije dostupan klijentu. S druge strane, aplet je moguće vratiti u izvorni oblik (dekompajlirati) bez prevelike muke, pa to i nije toliko velika prednost. Druga dva nedostatka pred skriptnim jezicima su dulje vrijeme skidanja zbog veličine kompajliranog koda, te dulje vrijeme učenja samog jezika, jer u 80% slučajeva jednostavniji, skriptni jezik, će u potpunosti zadovoljiti primjenu.

Ipak, skriptni jezici ne zadovoljavaju 20% slučajeva i upravo zbog toga postoji Java Applet.

3.1 Upotreba *Java appleta*

Javina vizualno i funkcionalno vjerojatno najzanimljivija primjena se manifestira u Java Appletima, malim djelićima Jave koji se automatski dobavljaju sa mreže i izvršavaju na Internet pretraživaču.

Applet ugrađujemo u *HTML Web* stranicu korištenjem oznake *APPLET*. Također moramo naznačiti Internet pretraživaču koliko će mjesta morati rezervirati za grafičku prezentaciju *appleta*. Proizvoljno još možemo ubaciti *PARAM* oznake ukoliko su *appletu* potrebni ulazni parametri. Na kraju taj dio izgleda nekako ovako:

```
<APPLET CODEBASE="PGP/Client" CODE="MainApplet.class" WIDTH=463 HEIGHT=360>
<param name=hostname value="localhost">
<param name=port value="4444">
<param name=KeyID value="0x9F9B08EB36329519">
<param name=UserID value="Pero Peric <pp@imagine.cc.fer.hr>">
<param name=debug value="true">
<H1 align=center>Your browser does not support <b>APPLET</b> tag</H1>
</APPLET>
```

Primijetite da se na pretraživačima koji su u stanju prikazivati *applete* sve unutar *APPLET* oznake zamjenjuje grafičkim prikazom *appleta*, dok na tekstualnim pretraživačima koji ne mogu pokazivati *applete* kao što je popularni *lynx* grafički prikaz *appleta* obično zamjenjuje alternativnim *HTML* oznakama.
Applet je odgovoran za sve manifestacije unutar njemu dodijeljenog prostora. On ne može mijenjati ostali sadržaj *HTML* dokumenta, ali zato može komunicirati sa drugim *appletom* pod uvjetom da se taj nalazi na istoj *HTML* stranici. Također je u stanju komunicirati s pretraživačem i naložiti mu da dohvati

neku novu *HTML* stranicu. *Applet* u pravilu svoje vizualno djelovanje ograničava na njemu pridijeljen prostor.
U neku ruku, *applet* je kao slika. Isto kao i sliku moraju mu se zadati dimenzije opisujući pretraživaču koliki pravokutan prostor treba rezervirati za *applet*. U jednoj stvari su slike čak i u prednosti nad *appletima*. One nasljeđuju informaciju o boji pozadine pretraživača tako da se primjerice prozirni dijelovi *GIF* slika prikazuju korektno. *Applet* na žalost ne nasljeđuje tu informaciju tako da je upravo njegov zadatak koju će boju pozadine postaviti ispod slika te koji će *font* (oblik slova) pridijeliti tekstovima i na koliko primjetan način se uklopiti u okolinu *HTML* dokumenta.

Kad zatražimo putem pretraživača prikaz *HTML* stranice koji u sebi sadrži *applet*, po jednakim principima kao i slike, *bajtni kod appleta* se dohvaća zajedno sa samim dokumentom. Ukoliko *applet* nije uobičajena objektna klasa bazirana na *java.applet.Applet* klasi nego se referencira na neke druge klase, svaka od referenciranih klasa se opet zasebno dohvaća. Izuzetak čine standardne klase koje su obično dio programskog paketa pretraživača tako da se dohvaćaju lokalno. Nakon toga, kada smo dobavili sve dijelove *applet* klase koji sada egzistiraju u istom adresnom prostoru, *bajtni kod appleta* se provjerava zadovoljava li sigurnosne norme. Tek nakon te provjere *applet* se počinje izvršavati. Gledano sa programerske strane, aplikacije i *appleti* pisani u *Javi* su vrlo slični. Međutim, sigurnosni zahtjevi sprečavaju *applete* u mnogo čemu što su aplikacije bez problema u stanju napraviti.

3.2 Sigurnost JavaAppleta

Sigurnost je vrlo važna stavka *Javinog* programskog okruženja. Lagano je izvedivo da korisnik zajedno sa *HTML* stranicom dohvati i *applet* a da i ne shvati kakav je "poklon" usput dobio, niti primijeti da se *applet* počeo izvršavati.
Kad bi *appleti* bili u stanju nekontrolirano pisati po čvrstom disku, kad bi mogli uništiti svaku datoteku kojoj mogu pristupiti, pokretati raznorazne naredbe kao *C:\DOS>format*, samoinicijativno dohvaćati *HTML*-stranice, koristiti *Web*-usluge umjesto vas (*mailto: franjo@vlada.hr* ☺) ili slati vaše privatne informacije natrag na *Web* poslužioc, korištenje *appleta* bi donijelo više štete nego koristi. Iz tog razloga što *appleti* mogu, a što ne mogu raditi striktno je unaprijed određeno. Kako trenutno stvari stoje, po standardima koje je definirao *Netscape*, *appleti*...

- ne mogu čitati niti pisati u lokalne datoteke,
- ne mogu pokretati druge programe na korisničkoj strani niti manipulirati *library* datotekama i
- ne mogu se putem mreže priključiti na bilo koje mjesto (*IP* adresu) osim na računalo sa kojega su dohvaćeni.

Naravno, sigurnosne rupe uvjek postoje.

3.3 Ograničenja, mane i prednosti

Razumljivo je da striktna ograničenja iz sigurnosnih razloga doprinose drastičnom smanjenju iskoristivosti punog potencijala *appleta*. Primjerice, možemo dohvatiti *applet* program koji predstavlja tablični kalkulator (tvrtka *AppliX*), ali sve naše kalkulacije ne možemo pohraniti na tvrdom disku našeg računala, već eventualno na računalu sa kojeg smo *applet* dohvatili. Ne bi bili u stanju koristiti ogromne rječnike koji su nam na raspolaganju na Internetu za pravopisnu i gramatičku analizu naših tekstova jer ne možemo natjerati *applet* da dohvati datoteku sa našeg računala i istu poslije obrade pohrani natrag.

Što nam ostaje? Jesu li uopće *appleti* korisni? Pored zanimljivih animacija i grafičkih efekata tu je uvijek i daleko veća interaktivnost koju možemo ostvariti upotrebom *appleta*.

Izmislimo jedan konkretan primjer: recimo da Nacionalno Sveučilišna Knjižnica želi prenosivo i jednostavno riješiti narudžbu i predbilježbu na pojedine knjige. Iskoristimo dosad standardno rješenje preko *FORM* oznake unutar *HTML* dokumenta te korištenje tzv. *formi* koje bi studenti popunjavali i slali putem pretražioca natrag. Ukoliko imamo velik broj interesanata za taj mrežni servis, lako bi dolazilo do zagušenja prometa poruka. Zadatak poslužioca je da osigura dohvaćanje *HTML* dokumenta korisniku. Unutar dokumenta korisnik popuni svoj zahtijev uz vlastiti identifikacijski broj. Tada putem *submit* metode popunjeni podaci sa *forme* putuju nazad do poslužitelja gdje se kontroliraju, te provjerava ispravnost unesenih podataka. Na kraju ako podaci nisu ispravo unijeti, poslužitelj ponovo mora kontaktirati korisnika i zahtijevati novi unos podataka u *formu*. Prezentirano rješenje nije previše efikasno jer s jedne strane zagušuje informacijski promet mrežom, dok s druge strane neravnomjerno raspoređuje resurse: dok studentski *moćni Pentiumi* vrše funkciju neinteligentnog grafičkog terminala, poslužioc mora obrađivati i kontrolirati hrpu zahtijeva za obradom.
Sad pretpostavimo da koristimo *applet* za unos podataka. *Applet* se izvršava na korisničkoj strani, što znači da niti najmanje ne smeta radu poslužioca. On može biti dovoljno "inteligentan" da sam prekontrolira valjanost unesenih podataka. Ukoliko se radi o većoj količini informacija, možemo čak dodati mogućnost kompresije podataka prije slanja. Na kraju, kad podaci stignu natrag do poslužioca možemo ih tako gotove, bez naknadnog kontroliranja pohraniti u zbirnu datoteku.

Valja napomenuti da nismo samo ograničeni na *Javu*. Ovaj problem bi se jednako efikasno mogao riješiti i korištenjem *Netscape*ovog *JavaScript* jezika koji je također podržan od strane popularnih Internet pretraživača. Međutim, u korist *Jave* mogli bi navesti činjenicu da je cijelo korisničko sučelje prezentirano unutar jednog jedinog entiteta, dakle jednom datotekom klase. Ostale prednosti *Jave* pred *JavaScript*om postaju sve izraženije kako problem koji trebamo programski riješiti postaje kompleksniji.

Nadalje, *HTML* ima još manje sposobnosti prikazivanja promjenljivih slika (ne nužno animacija) nego što ima mogućnosti interakcije. Možemo, primjerice, izvesti da poslužioc svako malo mijenja sadržaj *HTML* dokumenta (*refresh*), u našem konkretnom slučaju slike, te prisiliti pretraživač da svako malo dohvaća novu sliku. Možete li pretpostaviti kolika je tek ovdje iskoristivost i opterećenje mrežnih resursa? *Javinim appletom* možemo postići glatku animaciju bez konstantnog dohvaćanja sadržaja sa poslužioca. Možda se neki neće složiti ovdje s ovim, napominjući primjere još brže i glatkije animacije putem animiranih slika u specijalnom *GIF* formatu. Tu bi bili u pravu, jer ne postoji jednostavan način poboljšanja glatkoće animacije u *Javi* jer starije verzije *Jave* imaju velikih problema pri sinkroniziranju. No, cijela ova usporedba je kao usporedba igranog filma i video igrice. Možete li pretpostaviti kolika bi razlika bila da snimite na neki medij (primjerice CD) igricu sa svojih pola sata animacije i animirani film također od pola sata. Da ne pričamo o interaktivnosti jednog naspram drugog.

Treba primjetiti da jedino putem *just-in-time* načina izvođenja *bajtnog koda* možemo ubrzati *applete*. Primjena *native* izvršavanja ovdje ne dolazi u obzir. Čak i da teoretski izvedemo da poslužioc na neki način prepozna arhitekturu i operativni sistem koji korisnik upotrebljava, čak i ako u svojoj bazi podataka nađe i dohvati ispravnu verziju *native* programa te ga proslijedi korisniku, na korisnikovoj strani ne bi bilo moguće prekontrolirati osnovne norme sigurnosti analizom dobivenog koda, barem ne u nekom prihvatljivom vremenu.

3.4 Kako radi *Java Applet*

Klasa *java.applet.Applet* osigurava standardno sučelje između appleta i njihove okoline. Evo što se sve naziva appletom:

- mala aplikacija
- sigurni program koji se izvršava unutar web browsera
- podklasa klase `java.applet.Applet`
- instanca podklase od `java.applet.Applet`

Ovo je hijerarhija njenih nadklasa:

```
java.lang.Object
   |
   +---java.awt.Component
          |
          +---java.awt.Container
                 |
                 +---java.awt.Panel
                        |
                        +---java.applet.Applet
```

Aplet je podklasa klase **Applet** koja je sadržana u paketu **java.applet** i od koje nasljeđuje ponašanje. Mora biti deklariran kao **public** jer je podklasa klasa **Applet** koja je javna klasa.

```
public class Prvi Aplet extends java.applet.Applet {
 ..... // kod
}
```

Aplet prolazi kroz pet osnovnih faza:

1. Inicijalizacija

- prva faza koja se događa jednom
- učitavanje apleta u pretraživački program
- stvaranje potrebnih objekata, učitavanje slika, definicija parametara

```
public void init()  {
..... // kod              // Da bi se u apletu definiralo inicijalizaciju, treba nadjačati metodu init()
}
```

2. Pokretanje

- aplet se pokreće prilikom pristupa web stranici na kojoj se nalazi
- ova faza može se zbiti više puta za vrijeme trajanja životnog ciklusa jednog apleta

```
public void start()  {
..... // kod              // Da bi se definirao način pokretanja apleta, treba nadjačati metodu start()
}
```

3. Ispisivanje na ekran

- nastupa svaki put kada aplet želi ispisati/nacrtati nešto u prozor pretraživača tj. obnoviti sadržaj ekrana
- ova faza može se zbiti više puta za vrijeme trajanja životnog ciklusa jednog apleta, a aktivira se odmah nakon pokretanja apleta i kod svake obnove sadržaja ekrana

```
public void paint(Graphics g)  {
…… // kod              // Da bi se na ekranu nešto prikazalo treba nadjačati metodu paint()
}
```

Argument metode **paint()** je instanca klase **Graphics koju** kreira browser koji izvršava aplet. Da bi pretraživač to mogao učiniti, klasa **Graphics** koja je dio **java.awt** paketa mora biti importirana na početak datoteke koja sadrži izvorni kod apleta naredbom: **import java.awt.Graphics;**

4. Zaustavljanje

- nastupa svaki put kada korisnik napusti web stranicu koja sadrži aplet
- ova faza može se zbiti više puta za vrijeme trajanja životnog ciklusa jednog apleta

```
public void stop()  {
…… // kod              // Da bi se definirao način zaustavljanja apleta, treba nadjačati metodu stop(
}
```

5. Uništavanje

- faza koja se događa samo jednom kada pretraživač završava sa izvršenjem aplikacije
- metoda uklanja suvišne objekte, čisti memoriju,…

```
public void destroy()  {
…… // kod              // Nije potrebno nadjačavati metodu destroy(), ali može se učiniti
}
```

Faze i odgovarjuće metode:

```
import java.applet.*;
import java.awt.*;

public class KosturApleta extends Applet {
      public void init() {            // korisnik pristupa web stranici s apletom
      // inicijalizacijske naredbe
      }
      public void start() {           // aplet je učitan i spreman za pokretanje
      // naredbe koje se izvršavaju prilikom pokretanja
      }
      public void paint(Grafics g) {  // aplet je aktivan
      // naredbe koje utječu na ispis na ekranu
      }
      public void stop() {            // korisnik napušta stranicu ali ne i
                                     //pretraživački program
      // naredbe koje se izvršavaju prilikom zaustavljanja
      }
      public void destroy() {         // korisnik napušta pretraživački program
      // naredbe koje se izvršavaju na samom kraju života apleta
      }
```

Ugrađivanje apleta u Web stranicu

Kompajliranjem izvornog koda apleta dobiju se datoteke sa ekstenzijom *class* (*class file*) za svaku definiranu klasu. U HTML stranicu dodaje se poziv apleta pomoću naredbe **<APPLET>.** Sve što se nalazi između naredbe **<APPLET>** i **</APPLET>** predstavlja blok kojim se određuje način izvršenja java apleta.

Primjer: applet «Hello World» :

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApplet extends Applet {

   public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

Applet Hello World je nešto složeniji nego aplikacija, a ima i nešto dodatnog posla da bi ga se pokrenulo. Najprije se spremi izvorni kod u datoteku *HelloWorldApplet.java*. koja se kompilira na uobičajeni način. Dobije se klasa *HelloWorldApplet.class*. Da bi se pokrenula, potrebno je napraviti HTML dokument koji će sadržavati kreirani applet. Npr:

```
<HTML>
<HEAD>
<TITLE> HelloWorldApplet</TITLE>
</HEAD>

<BODY>
Ovo je applet HelloWorld:<P>
<applet code="HelloWorldApplet.class" width="150" height="50">
</applet>
</BODY>
</HTML>
```

Rezultat izvođenja ovog apleta je string “Hello World” unutar pravokutnika veličine 150x50 točaka.

4. *Java* na mobilnim uređajima (*Midlet*)

Midlet je izvedenica iz riječi MID - Mobile Information Device i applet. Svaki Java MIDlet sastoji se, zapravo, od dvije datoteke: jedna je sažeta Java arhiva sa samom aplikacijom (*.*jar*), što znači da na memorijskoj kartici zauzima manje mjesta od cijele aplikacije, a druga je "Java definition" datoteka (*.*jad*) s ostalim informacijama o Java MIDletu. U toj, drugoj datoteci, pohranjene su informacije o imenu MIDleta, njegovom autoru, te adresi na kojoj se može pronaći na Internetu.

Midleti se izvršavaju na sličan način kao i standardni Java Apleti. U pozadini svake Javine aplikacije nalazi se JVM (Java Virtual Machine). Kad se jednom prevede Javin kôd u .class datoteke i ukljuci se u JAR (Java Archive file) datoteku, JVM translatira class datoteku (tocnije bytecode Java class datoteke) u strojni jezik prilagoden platformi na kojoj radi JVM.
Mikro uređaji koriste razlicite vrste virtualnih mašina. CDC (*Connected Device Configuration*) koristi istu virtualnu mašinu kao i J2SE, dok je Sun za CLDC razvio potpuno novu specifikaciju Virtualne mašine znanu kao *K* virtualna mašina (engl. *Kilobyte Virtual Machine, KVM*). Ova virtualna mašina je razvijena kako bi omogućila rad s ograničenim (limitiranim) uređajima. Zbog toga ona i nije "normalna" virtualna mašina, već ima sljedeće zahtjeve:

- Virtualna mašina za sebe zahtjeva samo 40 do 80 kilobyte-a memorije
- Zahtjeva svega 20 do 40 kilobyte-a dinamičke memorije
- Može raditi na 16 bitnim procesorima s taktom od svega 25 MHz.

Dakle, KVM je Sun-ova implementacija JVM-a koja je prilagođena osnovnim značajkama CLDC-a. Bitno je napomenuti da to nije jedina virtualna mašina za mikro uređaje. Danas veći proizvođači mikro uređaja imaju svoje virtualne mašine.

J2ME (engl. *Java 2 Micro Edition*) je verzija programskog jezika Java razvijenog za uređaje ograničene memorije, malih zaslona i slabijih procesora.

4.1 Connected, Limited Device Configuration (CLDC)

CLDC ima dvije glavne zadaće. To su definiranje specifikacije za Javinu virtualnu mašinu (JVM) i definiranje skupa javinih klasa (biblioteka). Trenutačno je u razvoju druga generacija CLDC-a koja će imati ugrađenu podršku za brojeve s pomičnim zarezom, dodatna rukovanja pogreškama itd.

4.1.1 *Hardware*-ski zahtjevi

Kako mobilni uređaji nisu jednaki, točnije, svaki je drukčiji (imaju različite procesore, memorije, operacijske sustave itd.), usklađeni su neki osnovni zahtjevi koji su većinom vezani za memoriju. Minimalni zahtjevi memorije su:

- 128 kilobyte-a memorije za rad JVM i CLDC biblioteka. Ta memorija u svakom trenutku mora čuvati svoj sadržaj, bez obzira da li je mobilni uredaj ugašen ili ne. Ona se još naziva i nepromjenjiva memorija.
- 32 kilobyte-a memorije dostupnih tijekom rada aplikacije za alociranje objekata. Takva vrsta memorije se još naziva promjenjiva memorija ili gomila (engl. *heap*).

Ako mobilni uređaj nema mogućnosti stavljanja dodatnih programa (neki mobilni uređaji imaju tvornički ugrađene aplikacije, koje se ne mogu brisati), onda je dovoljno i manje memorije.

4.1.2 *Software*-ski zahtjevi

Software-ski zahtjevi se ne razlikuju previše od hardware-skih. Ostaje minimalni skup zahtjeva od strane CLDC-a. Operacijski sustav mora moći vrtjeti JVM i upravljati javinim aplikacijama na uređaju, što uključuje:

- Izbor i pokretanje aplikacija
- Mogućnost uklanjanja Javinih aplikacija iz uređaja

4.2 Mobile Information Device Profile (MIDP)

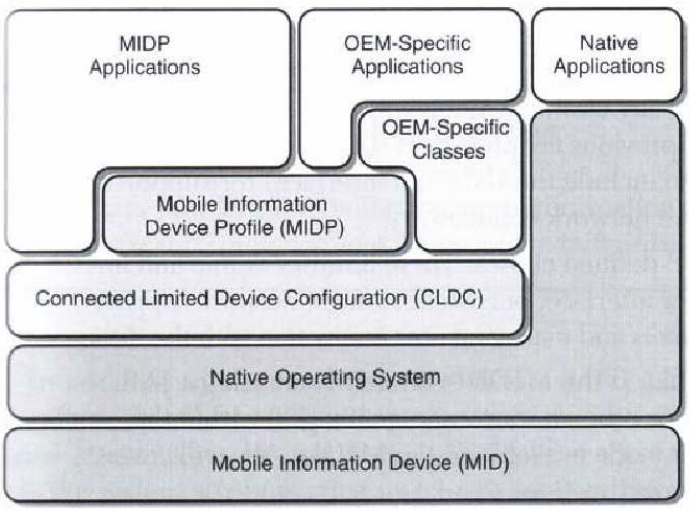
Mobile Information Device Profile (MIDP) definira programsko okruženje za *Mobile Information Devices (MIDs).*
Da bi se pobliže upoznali s MIDP-om moramo znati koji su osnovni hardware-ski i software-ski zahtjevi uređaja koji namjeravaju implementirati MIDP.
Tako su osnovni hardware-ski zahtjevi:

- Ekran uređaja mora podržavati rezoluciju od najmanje 96*54 točaka.
- Mora biti dostupan barem jedan korisnički ulaz. To su telefonska tipkovnica, «Qwerty» tipkovnica ili «Touch screen».
- 128 Kilobyte-a neizbrisive memorije za pokretanje Mobile Information Device (MID) komponenti.
- Barem 8 Kilobyte-a neizbrisive memorije za trajne podatke aplikacija, kao što su postavke programa i slično.
- 32 Kilobyte-a izbrisive memorije za pokretanje Jave.
- Bežična mrežna povezanost (engl. *Wireless network connectivity*)

Operacijski sustavi koji pokreću mobilne uređaje variraju od uređaja do uređaja. Zbog toga je potrebno definirati i minimalni skup software-skih zahtjeva kako bi širok krug uređaja mogao ugraditi MIDP.

- Operacijski sustav koji pokreće mobilni uredaj mora omoguciti «*minimal scheduling*», rukovanje iznimkama, te procesuiranje prekida. Takoder OS mora moci pokrenuti i vrtjeti JVM.
- Software mora podržavati bitmap grafiku na ekranu.
- Koristeći bilo koju od tri navedene ulazne jedinice, software mora prihvatiti ulaz i proslijediti ga do JVM.
- Zbog trajnih podataka, software mora imati mogućnost čitanja i pisanja u odnosno iz neizbrisive memorije.
- Mora imati pristup obilježjima mreže na uređaju (pogotovo čitanja i pisanja podataka kroz bežičnu mrežu).

4.2.1 *MIDP* arhitektura



Slika 4.2.1.1 MIDP arhitektura

Kako bi se stekao bolji dojam o MIDP-u, najbolje je cijeli MIDP prikazati njegovom arhitekturom. Zbog vizualnog prikaza najbolje je početi sa samim uređajem, zato se on i nalazi na dnu. Korak iznad uređaja nalazi se operacijski sustav uređaja. Odmah potom (gornji desni kut) dolazi prvi aplikacijski nivo (tzv. *Native Applications*) na MID-u. Do pojave Jave to su bili jedini dostupni programi. To su programi koje su instalirani u mobitele u tvornicama, poput namještanja melodija, jačine zvona, datuma, vremena i drugih stvari. CLDC je instaliran na operacijski sustav i to je temelj MIDP-a. Bez njega nema ni MIDP-a. Treba primijetiti da MIDP programi imaju pristup bibliotekama i CLDC-a i MIDP-a. OEM (engl. *Original Equipment Manufacturer*) klase su instalirane od strane proizvođača uredaja i one omogućuju pristup dodatnim funkcijama koje će uređaj imati. Važno je napomenuti da su te klase posebne i razlikuju se od uređaja do uređaja.