3. *JAVA* TECHNOLOGIJOS

3.1. Klientinės programos. *Swing* technologija

3.1.1. Bendrosios žinios apie klientines programas

Pagrindinė klientinės programos (*applet*) paskirtis – praturtinti įprastus interneto naršyklės rodomus www puslapius. Kadangi klientinės programos įkraunamos į kliento kompiuterį iš nutolusio kompiuterio, saugumo sumetimais joms apribota prieitis prie kliento kompiuterio kietojo disko tiek rašyti, tiek skaityti. Tačiau patikimoms (*trusted*) klientinėms programoms, pasirašytoms skaitmeniniu parašu, tokia prieiga galima. Klientinės programos veikia naršyklės viduje – jos adresinėje erdvėje. Naršyklė klientinės programos atžvilgiu atlieka maždaug tas pat funkcijas, kaip OS savarankiškos aplikacijos atžvilgiu.

Klientinių programų įkrovai reikia laiko. Todėl rekomenduojama jas įdėti į archyvines JAR rinkmenas: suspaustame formate į vieną rinkmeną sudedamos visos reikiamos klientinių programų darbui klasės ir visos reikiamos daugialypės terpės rinkmenos.

Klientinių programų aplikacijos karkasas – klasė *Applet* (jei klientinėms programoms kurti naudojamas pasenęs paketas *java.awt*) arba klasė *JApplet* (jei naudojamas dabar rekomenduojamas paketas *javax.swing*. Beje, visi kalbą papildantys *Java 2* paketai pradedami vardu *javax* – nuo žodžio *eXtension* – *išplėtimas*). Mes visas klientines programas kursime remdamiesi *JApplet* klase. Programuotojo rašoma klientinės programos startinė klasė turi paveldėti iš šios klasės metodus ir savybes. Nors AWT (*Abstract Windowing Toolkit*) paketas skelbiamas pasenusiu, tebesinaudojama jo konteinerių ir įvykių klasėmis bei metodais, todėl klientinėje programoje teks įsikelti klases iš paketų *javax.swing, java.awt* ir *java.awt.event*.

AWT nėra daug grafinių elementų, o naujo grafinio elemento kūrimas yra keblus: būtina paveldėti pasirinktojo elemento savybes ir perrašyti metodus, tuo suteikiant naujajai klasei reikiamą funkcionalumą. Visi paketo elementai yra vadinamieji „sunkiasvoriai“ (*heavyweight*). Kad paketas būtų nepriklausomas nuo platformos, *Sun* pasirinko tokį elementų kūrimą: faktiškai naudojami gimtieji operacinės sistemos grafiniai įrankių komplektai. Todėl kuriant, pavyzdžiui, mygtuką *Windows OS*, sukuriamas *Windows* mygtukas, o jei mygtukas kuriamas iš *Linux* – JVM sukuria *Linux* mygtuką. Tokia technika dar vadinama lygiaverčių šablonų (*peer pattern*) technika. Ji menkai tinka sudėtingoms grafinėms sąsajoms kurti, kadangi kuri nors OS gali neturėti kurio nors sudėtingesnio grafinio elemento.

Swing pakete atsisakyta naudoti gimtuosius OS grafinių įrankių atitikmenis; visi elementai 100 % sukurti *Java* – jie yra „lengvasvoriai“ (*lightweight*). Todėl tokie elementai visose OS vaizduojami lygiai taip pat. Tačiau dėl tos pačios priežasties tokių komponentų klasės yra didelės apimties (dažniausiai – apie 2 MB). Paketas yra daug turtingesnis, o paketo elementus galima žymiu mastu tinkinti vartotojo reikmėms.

Klientinių programų paleidimo tokiu būdai:

1. Naršykle: peržiūrėti www puslapį, kurio HTML kode yra sakiniai <*applet*> arba <*embed*>, nurodantys, iš kur reikia parsisiųsdinti klientinės programos kodą.

Atsiųsta klientinė programa automatiškai paleidžiama. Dabar, esant daugeliui naršyklių versijų ir keletui *Java* JDK ar SDK versijų, yra didelė tikimybė, kad klientinė programa naršyklėje neveiks. Gali prireikti plėtinių (*plug-ins*) *Netscape Navigator* naršyklei arba *Active-X* modulių *Internet Explorer* naršyklei. Kompanija *Javasoft* garantuoja tokius plėtinius ir nemokamai juos platina. Į HTML kodą galima įdėti sakinius <*object*> ir <*embed*> su informacija apie tuos plėtinius. Reikiamų plėtinių vardai įvairioms JVM versijoms pateikti java.sun.com/products/plugin/version.html.

Sakiniuose <*applet*> arba <*embed*> turi būti trys privalomi parametrai :

code = rinkmenos **.class* vardas
width = klientinės programos lango plotis vaizdo taškais
height = klientinės programos lango aukštis vaizdo taškais

ir gali būti neprivalomi parametrai

codebase = klientinės programos kodo URL adresas; jei parametras nenurodytas – klientinės programos įkrovai naudojamas HTML dokumento URL adresas
align = nurodo klientinės programos lango lygiavimą naršyklės lange. Galimos parametro reikšmės *left, rigth, top, bottom, middle, baseline, ...* .

Visas HTML dokumento pavyzdys yra 1.9 skyriuje. Beje, nesvarbu, ar HTML sakiniai rašomi didžiosiomis, ar mažosiomis raidėmis. Jų parametrų išdėstymo tvarka sakinyje-konteineryje yra nesvarbi. Sakinių parametrai skiriami vienas nuo kito tuščiais tarpais. Parametrų reikšmės teikiamos arba tarp kabučių " ", arba be jų, o klientinės programos sukompiliuotai rinkmenai priesagą *class* galima praleisti. Daugelis dabartinių naršyklių HTML dokumentą atpažįsta ir be startinių sakinių <*HTML*>, <*HEAD*> ir <*BODY*>, taigi galima jų ir nerašyti. Platesnius paaiškinimus apie kai kuriuos HTML sakinius rasite šioje dalyje, serverinių programų skyriuje, viename iš serverinių programų pavyzdžių.

2. Įrankiu *appletviewer*. Ši galimybė rekomenduojama testuojant klientines programas. Klientinei programai paleisti įrankiu iš pulto reikia sukompiliuotos klientinės programos rinkmenos ir HTML dokumento su sakiniu <*applet*>, kuriame nurodytas sukompiliuotos rinkmenos vardas, rinkmenos; abi rinkmenos turi būti viename aplanke:

appletviewer HTML_rinkmenos_vardas

Testuoti klientines programas vis tik daug patogiau naudojantis *appletviewer* savybe kreipti dėmesį tik į sakinį <*applet*>. Galima tiesiai į klientinės programos kompiliacijos vieneto tekstą įdėti užkomentuotą sakinį

// <applet code = ... width = … height = …
// </applet>

ir įrankiu paleisti klientinę programą taip:

appletviewer klientinės programos_pradinė_rinkmena.*java*

Dabar, keičiant, pavyzdžiui, klientinės programos lango pradinius matmenis, klientinės programos nereikės ir perkompiliuoti, teks tik pakeisti sakinio tekstą komentare.

3. Galima pritaikyti klientinę programą paleisti ir iš naršyklės, ir iš komandinės eilutės pulte. Tam klientinės programos tekstą dar reikėtų papildyti metodu *main* su išreikštu visų klientinės programos karkaso metodų kvietimu (žr. B.Eckelio knygą).

3.1.2. Klientinės programos architektūra

Klientinė programa – programa, dirbanti su langu ir jame išdėstomais valdymo elementais, todėl jos architektūra skiriasi nuo kitų programų – tai „įvykių valdoma sistema“ (*event-driven system*). Iš esmės tai panašu į išimčių apdorojimo mechanizmą: klientinė programa laukia kokio nors įvykio, jį „sugauna“ įvykio apdoroklis, tada klientinė programa atlieka reikiamus veiksmus, o paskui klientinė programa laukia kitų įvykių. Šie veiksmai turi būti atlikti greitai, realiame laike, todėl kartais juos tenka vykdyti atskiru vykdymo srautu. Kitas klientinės programos esminis skirtumas nuo įprastos programos – darbą su klientine programa inicializuoja vartotojas, kai jam to reikia (paspausdamas pelės klavišą ant reikiamo grafinio lango elemento ar pan.). Toks vartotojo veiksmas generuoja įvykį, į kurį reaguoja klientinė programa.

Klientinė programa paveldi *JApplet* klasės savybes ir modifikuoja reikiamus metodus. Metodai, valdantys klientinės programos gyvavimą www puslapyje, yra (visi – *public void*) šie:

init( ) – automatiškai kviečiamas įkraunant klientinę programą į naršyklę. Perrašant metodą, jame reikėtų inicializuoti duomenis, nustatyti grafinių komponentų išdėstymą lange. Šis metodas visada perrašomas.

start( ) – kviečiamas po metodo *init* ir visada automatiškai kviečiamas, kai klientinės programos langas patenka į matomą www puslapio zoną (pavyzdžiui, atkuriant klientinės programos langą, anksčiau sutrauktą į piktogramą, atitinkamu mygtuku).

stop( ) – automatiškai kviečiamas kiekvienąkart, klientinės programos langui paliekant matomą puslapio zoną (pavyzdžiui, sutraukiant klientinės programos langą į piktogramą mygtuku). Perrašant metodą, jame rekomenduojama sustabdyti daug išteklių reikalaujančius veiksmus.

destroy( ) – automatiškai kviečiamas, kai klientinė programa iškraunama iš naršyklės. Metodas išlaisvina visus klientinės programos naudotus kompiuterio išteklius.

Taigi iš šių klientinės programos gyvavimą palaikančių metodų dažnai užteks tik perrašyti *init* metodą, o kitus palikti tokius, kokie jie apibrėžti *JApplet* klasėje. Dar keli metodai, kuriuos teks dažnai perrašyti savo klientinėse programose – iš metodo

void repaint( ) kviečiamas metodas *void paint(Graphics g).* Šiaip *paint* automatiškai kviečiamas, kai reikia perpiešti klientinės programos generuojamą langą. Taigi, jei klientinės programos darbo metu, pavyzdžiui, reaguojant į kurio nors mygtuko paspaudimą, reikia kaip nors pakeisti klientinės programos lango vaizdą, – teks išreikštai kviesti metodą *repaint,* šis automatiškai kvies *paint,* o perrašytame metode *paint* teks nurodyti, ką ir kaip nupiešti. Metodas *paint* turi vieną *Java* klasės *Graphics* tipo argumentą, per kurį gauna visą grafinį klientinės programos kontekstą.

Klientinės programos, kuri tik lange sukuria du mygtukus, bet niekaip nereaguoja į jų paspaudimus, pavyzdys. Visi metodai ir klasės bus paaiškinti vėliau.

```
// Applet shows two buttons
// <applet   code = Buttons    width =  200   height = 100
// </applet>

import javax.swing.*;
import java.awt.*;

public class Buttons extends JApplet{
        JButton b1 = new JButton( “Button 1” ),
                 b2 = new JButton( “Button 2” );
        public void init( ){
                Container c = getContentPane( );
                c.setLayout( new FlowLayout( ) );
                c.add( b1 );
                c.add( b2 );
        }
}
```

Klientinę programą paleisti galima sukompiliavus šią programą ir komandinėje eilutėje surinkus:

```
appletviewer   Buttons.java
```

3.1.3. Įvykių apdorojimo sistema

Įvykių klasės yra pakete *java.awt.event*, o jų klausytojų klasės – pakete *javax.swing*.

Klientinėse programose naudojamas vadinamasis įvykių delegavimo modelis: „įvykio šaltinis“ generuoja įvykį ir siunčia pranešimą apie tai vienam ar keliems „įvykių klausytojams“. Įvykio šaltinis – kuris nors klientinės programos vartotojo veiksmas grafinėje sąsajoje, pavyzdžiui, pelės klavišo nuspaudimas pelės žymekliui esant ant mygtuko. Klausytojo funkcijos – laukti pranešimo apie įvykį. Gavęs tokį pranešimą, klausytojas reikiamu būdu (tą užprogramuoja programuotojas) apdoroja įvykį ir grąžina valdymą klientinei programai. Šis laukia kitų vartotojo veiksmų.

Taigi šiame modelyje įvykius generuojanti vartotojo sąsaja yra griežtai atskirta nuo įvykių apdorojimo mechanizmo. Svarbi modelio savybė, įvesta tik *Java 2,* yra tai, kad įvykių klausytojai, norėdami gauti pranešimus apie įvykį, privalo „užsiregistruoti“ įvykių šaltinyje. Ankstesnėje *Java* versijoje pranešimai apie įvykius buvo siunčiami visiems klausytojams.

Dabar smulkiau apie įvykių delegavimo modelio elementus.

Įvykiai – objektai, aprašantys šaltinio būklės pokyčius. Kaip minėta, tai gali būti vartotojo veiksmų sukelti įvykiai grafinėje sąsajoje arba laikmačio nurodymu generuojamas įvykis, arba tam tikros skaitiklio reikšmės pasiekimas veikiant klientinei programai, ir pan. Programuotojas pats nustato įvykius, kuriuos apdoros klientinė programa. Įvykių klasių vardai paklūsta griežtai įvardijimo schemai: *TypeEvent*, kur vietoje pirmosios vardo dalies *Type* yra žodžiai *Mouse* – pelės įvykiams, *Key* – klaviatūros įvykiams, *Adjustment* – slankiklio įvykiams, ir t. t. Visos šios įvykių klasės yra pakete *java.awt.event.* Visų jų superklasė – *EventObject*, kurioje apibrėžti tik du metodai:

Object getSource( ) – grąžina įvykio šaltinį;
String toString( ) – grąžina įvykio aprašą.

Įvykių šaltiniai – objektai, generuojantys įvykius. Vienas šaltinio objektas gali generuoti ir kelis įvykius: pavyzdžiui, pelė generuoja pavienius įvykius klavišui nuspausti, klavišui atleisti, klavišui dukart nuspausti, pelės žymekliui traukti nuspaudus klavišą, nenuspaudus klavišo. Šaltinis registruoja įvykių klausytojus, kuriems siųs pranešimus. Kiekvienas įvykio tipas turi atskirą registracijos metodą, kurio aprašas paklūsta griežtai schemai

public void addTypeListener(TypeListener tl),

čia *Type* – įvykio objekto vardas.

Kai kurie šaltiniai leidžia registruoti tik vienintelį klausytoją; jie sukelia išimtį *java.util.TooManyListenerException*.

Šaltiniai, kaip ir registracijai, turi analogiškus klausytojus pašalinančius metodus

public void removeTypeListener( )

Visi registracijos ir pašalinimo metodai yra atitinkamose įvykių klasėse.

Įvykių klausytojai – objektai, gaunantys pranešimus apie įvykius. Be minėtos įvykių registracijos, klausytojas privalo realizuoti pranešimų priėmimo ir apdorojimo metodus, nurodomus klausytojus atitinkančiose sąsajų klasėse, esančiose pakete *java.awt.event*. Šių sąsajų klasių vardai suformuoti taip pat griežtai: *TypeListener*.

3.1 lentelėje pateikiamas sąrašas grafinėse sąsajose dažniausiai naudojamų įvykių klasių, klausytojų sąsajų klasių, klausytojų registracijos ir pašalinimo metodų ir įvykių šaltinių.

3.1 lentelė. Įvykių sistema

Įvykio klasė Klausytojo sąsajos klasė Klausytojo registracijos ir pašalinimo metodai	Įvykio apibūdinimas	Įvykių šaltinių klasės
ActionEvent *ActionListener* *addActionListener* *removeActionListener*	Generuojamas, kai nuspaudžiamas pelės klavišas arba klavišas dukart nuspaudžiamas sąsajos elemente	*JButton, JList, JTxtField, JMenuItem, JCheckBoxMenuItem, JMenu, JPopupMenu*
AdjustmentEvent *AdjustmentListener* *addAdjustmentListener* *removeAdjustmentListener*	Generuojamas, kai slankikliu pakeičiama reikšmė	*JScrollBar* ir visos klasės, realizuojančios sąsajų klasę *Adjustable*
ComponentEvent *ComponentListener* *addComponentListener* *removeComponentListener*	Generuojamas, kai komponentas paslepiamas, pasidaro matomas, perstumiamas, pakeičiami jo matmenys.	*Component* ir visos jos subklasės
ContainerEvent *ContainerListener* *addContainerListener* *removeContainerListener*	Generuojamas, kai komponentas įdedamas ar pašalinamas į/iš konteinerio	*Container* ir visos jos subklasės
FocusEvent *FocusListener* *addFocusListener* *removeFocusListener*	Generuojamas, kai komponentas patenka į fokusą ar jo netenka	*Component* ir visos jos subklasės
KeyEvent *KeyListener* *addKeyListener* *removeKeyListener*	Generuojamas, kai klavišas nuspaudžiamas, atleidžiamas, kai spausdinamas simbolis	*Component* ir visos jos subklasės
MouseEvent *MouseListener* ir *MouseMotionListener* *addMouseListener* ir *addMouseMotionListener* *removeMouseListener* ir *removeMouseMotion-Listener*	Generuojamas, kai objektas traukiamas pele; kai pelė judinama; kai pelės klavišas paspaustas ir atleistas; kai klavišas paspaustas; kai klavišas atleistas; kai žymeklis įeina į komponentą; kai žymeklis palieka komponentą	*Component* ir visos jos subklasės

WindowEvent *WindowListener* *addWindowListener* *removeWindowListener*	Generuojamas, kai langas aktyvuojamas, deaktyvuojamas, atidaromas, uždaromas, sutraukiamas į piktogramą, išplečiamas iš piktogramos	*Window* ir visos jos subklasės
ItemEvent *ItemListener* *addItemListener* *removeItemListener*	Generuojamas, kai pažymimas sąrašo ar meniu elementas arba vėliavėlė	*JCheckBox, JCheckBoxMenuItem, JComboBox, JList* ir visos sąsajų klasę *ItemSelectable* realizuojančios klasės
TextEvent *TextListener* *addTextListener* *removeTextListener*	Generuojamas, kai komponente pakinta tekstas	*JTextComponent* ir visos jos subklasės

Žinant 3.1 lentelėje išvardytus įvykių klausytojų vardus, nesunku internete kalbos dokumentacijoje susirasti visus sąsajų klasėse deklaruotus metodus. Pavyzdžiui, *ActionListener* sąsajų klasė turi tik vieną metodą

actionPerformed(ActionEvent ae)

Sąsajų klasės pelės įvykiui *MouseEvent* – *MouseListener ir MouseMotionListener* – turi atitinkamai metodus

mouseClicked(MouseEvent me)
mouseEntered(MouseEvent me)
mouseExited(MouseEvent me)
mousePressed(MouseEvent me)
mouseReleased(MouseEvent me)

ir metodus

mouseDragged(MouseEvent me)
mouseMoved(MouseEvent me)

Kadangi klausytojų sąsajų klasės dažniausiai turi po kelis paskelbtus metodus, o kuriant grafinę sąsają dažnai tereikia tik vieno kurio metodo – realizuoti sąsajų klases ne visada patogu. Tokiems atvejams skirtos klasės-adapteriai, turinčios visus atitinkamose sąsajų klasėse paskelbtus, tačiau „tuščius“ metodus. Tada geriau ne realizuoti sąsajų klasę, o plėsti klasę-adapterį perrašant tik dominantį metodą ar kelis metodus. Šių klasių vardai konstruojami pagal tą pačią vardų schemą: pavyzdžiui, sąsajų klasę *MouseListener* atitinka klasė *MouseAdapter* ir t. t.

Pateiksime įvykių klasės pavyzdį – kad būtų aišku, ką galima gauti iš įvykio klasės objekto, kaip išplėsti klasę, rašant savą analogišką klasę. Taigi klasė *ActionEvent* turi konstruktorius:

ActionEvent(Object source, int type, String cmd)
ActionEvent(Object source, int type, String cmd, int modifiers)

Čia *source* – rodyklė į objektą, sukėlusį įvykį, *type* – konstanta – įvykio tipas, *cmd* – įvykio komandinė eilutė, *modifiers* – konstanta, kurios reikšmę atitinka klavišas-modifikatorius, buvęs nuspaustas generuojant įvykį (Alt, Ctrl ir/arba Shift).

Klasė turi metodus

String getActionCommand( )
String setActionCommand( )
int getModifiers( )

Pirmojo metodo prasmė: pavyzdžiui, kai grafinėje sąsajoje nuspaustas mygtukas, metodo rezultatas yra mygtuko tekstas – *JLabel* objektas. Kitų dviejų metodų paskirtis aiški iš jų pavadinimų.

3.1.4. Klientinių programų pavyzdžiai

1 pavyzdys. Klientinės programos lange, naudojant komponentų išdėstymo tvarkyklę *FlowLayout* (žr. kitą skyrių), išdėstomi du mygtukai (*JButton* klasės objektai) ir vienas tekstinis laukas (*JTextField*). Mygtukai turi užrašus *Button 1* ir *Button 2*, kurie, nuspaudus atitinkamą mygtuką, pasirodys tekstiniame lauke.

```
// 1st example of applet: buttons and event
//
//<applet code = Button2 width = 300 height = 200>
//</applet>
//
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button2 extends JApplet{
        JButton b1 = new JButton("Button 1"),
                b2 = new JButton("Button 2");
        JTextField tf = new JTextField( 20 );
        class ALC implements ActionListener{
                public void actionPerformed( ActionEvent e){
                    String buttonName = ( (JButton)e.getSource() ).getText();
                    tf.setText( buttonName );
                }
        }
        ALC al = new ALC( );
         public void init( ){
                b1.addActionListener( al );
                b2.addActionListener( al );
```

```
                Container c = getContentPane( );
               c.setLayout( new FlowLayout( ) );
                c.add( b1 );
                 c.add( b2 );
                c.add( tf );
        }
    }
```

Taigi klientinėje programoje reikia realizuoti klausytojo sąsajos klasę *ActionListener*, nes būtent joje yra metodas *actionPerformed*, reaguojantis į pelės klavišo nuspaudimo įvykį, esant žymekliui ant kurio nors komponento, *ActionEvent*. Sąsajų klasės tipo objektas konstruojamas vidine vardine klase, perrašant joje metodą *actionPerformed*. Metode, norint identifikuoti, kuris mygtukas nuspaustas, pasitelkiamas anksčiau minėtas visų įvykių superklasės *ObjectEvent* metodas *getSource*. Kadangi šis grąžina *Object* tipo rodyklę – būtinas besileidžiantis tipų pertvarkymas į *JButton* tipą. Toliau kviečiamas *JButton* metodas *getText*, grąžinantis mygtuko tekstą – *String* tipo duomenį. Vėliau klasės *JTextField* metodu *setText* šis duomuo kopijuojamas į tekstinį lauką *tf*.

Klientinėje programoje perrašomas vienintelis klientinės programos karkaso metodas *init*. Jame abiems mygtukams užregistruojamas tas pats įvykių klausytojas *al*. Klientinėje programoje visi grafiniai komponentai išdėstomi konteineryje, kuris gaunamas metodu *getContentPane*. Išdėstymą valdo išdėstymo tvarkyklės, nustatomos konteineriui metodu *setLayout*. Vėliau į konteinerį norima tvarka sudedami visi reikiami grafiniai komponentai.

Jei paleisime šią klientinę programą, bus parodytas klientinės programos langas su dviem mygtukais ir tuščiu tekstiniu lauku. Klientinės programos grafinė sąsaja reaguos tik į pelės klavišo nuspaudimą ant mygtuko – įvykį *ActionEvent*. Nuspaudus kurį nors mygtuką, šį įvykį „sugaus“ įvykio klausytojas ir atitinkamai reaguos: tekstiniam laukui bus nustatytas tekstas *Button 1* arba *Button 2*.

2 pavyzdys. Žemiau pateikiama klientinė programa realizuoja tą pačią grafinę sąsają, demonstruodama skirtingas galimas sintakses klausytojui konstruoti: bevardę vidinę klasę ir klausytojo objekto konstravimą toje kodo vietoje, kur jis registruojamas. Ši forma būtų patogi tada, kai kuriam nors komponentui klientinėje programoje registruojamas vienintelis toks klausytojas. Demonstruojamos ir skirtingos komponentų išdėstymo tvarkyklės:

```
// 2nd example: buttons and event, inner classes, other layouts
//
//<applet code = Button3 width = 300 height = 200>
//</applet>
//
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button3 extends JApplet{
  JButton b1 = new JButton("Button 1"),
           b2 = new JButton("Button 2");
  JTextField tf = new JTextField( 20 );
```

```
 ActionListener al = new ActionListener( ){       // 2nd syntax' alternative;
  public void actionPerformed( ActionEvent e ){  // to be preferred
   String buttonName = ( (JButton)e.getSource( ) ).getText( );
   tf.setText( buttonName );
  }
 };       // ;!   ":" inherited from constructor
 public void init( ){
  b1.addActionListener( al );
  b2.addActionListener( new ActionListener( ){     // 3rd syntax' alternative; to be
   public void actionPerformed( ActionEvent e ){ // preferred for a single addition
    String buttonName = ( (JButton)e.getSource( ) ).getText( );
    tf.setText( buttonName );
  }
 } );
  Container c = getContentPane( );
  c.add( BorderLayout.EAST, b1 );
  c.add( BorderLayout.WEST, b2 );
  c.add( BorderLayout.CENTER, tf );
 }
}
```

3 pavyzdys. Realizuojama ta pati grafinė sąsaja. Šis klientinė programa demonstruoja iš principo kitokią klientinės programos schemą: klausytoju padaroma pati klientinė programa, todėl jos klasė turi realizuoti atitinkamą klausytojo sąsajos klasę, t. y. joje turi būti perrašyti sąsajų klasės metodai.

```
// 3rd example: buttons and event, alternative listening to the events
//
//<applet code = Button31 width = 300 height = 200>
//</applet>
//
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button31 extends JApplet implements ActionListener{
        JButton b1 = new JButton( "Button 1" ),
                b2 = new Jbutton( "Button 2" );
        JTextField tf = new JTextField( 20 );
         public void init( ){
                b1.addActionListener( this ); // ie, adding applet as a listener
                 b2.addActionListener( this );
                Container c = getContentPane( );
                 c.add( BorderLayout.NORTH, b1 );
                 c.add( BorderLayout.SOUTH, b2 );
                  c.add( BorderLayout.CENTER, tf );
         }
        public void actionPerformed(ActionEvent e){
                String buttonName = ( (JButton)e.getSource( ) ).getText( );
                tf.setText( buttonName );
```

```
        }
}
```

4 pavyzdys. Kelių įvykių apdorojimo pavyzdys: grafinė sąsaja reaguoja į visus pelės įvykius. Naudojama 3-iame pavyzdyje parodyta klientinės programos schema: pačioje jos klasėje realizuojamos abi pelės įvykių klausytojų sąsajų klasės. Klientinė programa spausdina pranešimą apie įvykį *message*, o pelės žymeklio įėjimo į langą ir išėjimo iš lango įvykiams – ir įėjimo bei išėjimo taško koordinates vaizdo taškais. Pelę traukiant su nuspaustu klavišu, lange spausdinamas ir žymeklio pėdsakas. Reaguojama į bet kurį pelės klavišą.

```
// 4th example: mouse and multiple events
//
//<applet code = Mouse1 width = 800 height = 500>
//</applet>
//
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Mouse1 extends JApplet
      implements MouseListener, MouseMotionListener{  // applet listens to all
                                                      // mouse events
 String message = "";
  int x=0, y=0;          // where the message is printed
  int xa=0, ya=0;     // where the mouse enters and leaves window
  int i=0;                // counts events "entered" and "exited"

  public void init( ){
    addMouseListener( this );    // ie, adding applet as a listener
    addMouseMotionListener( this );
  }

  public void mouseClicked( MouseEvent me ){
    x = me.getX( )+100;  // to separate messages "clicked" and "pressed"
    y = me.getY( );
    message = "Mouse clicked";
    repaint( );
  }

  public void mouseEntered( MouseEvent me ){
    i++;
    x = 10;
    y = i*10;
    xa = me.getX( );
    ya = me.getY( );
    message = "Mouse entered" + xa + ya;
    repaint( );
  }
```

```
 public void mouseExited( MouseEvent me ){
  i++;
  x = 10;
  y = i*10;
  xa = me.getX( );
  ya = me.getY( );
  message = "Mouse exited" + xa + " " + ya;
  repaint( );
 }

 public void mousePressed( MouseEvent me ){
  x = me.getX( );
  y = me.getY( );
  message = "Mouse pressed";
  repaint( );
 }

 public void mouseReleased( MouseEvent me ){
  x = me.getX( ) + 100;
  y = me.getY( );
  message = "Mouse released";
  repaint( );
 }

 public void mouseDragged( MouseEvent me ){
  x = me.getX( );
  y = me.getY( );
  message = "*";
  showStatus( "Dragging at " + x + ", " + y );  // message at the bottom of window
  repaint( );
  }

 public void mouseMoved( MouseEvent me ){
  x = me.getX( );
  y = me.getY( );
  message = " ";
  showStatus( "Moving at " + x + ", " + y );
  repaint( );
 }

 public void paint( Graphics g) {
  g.drawString( message, x, y );
 }
}
```

Šioje klientinėje programoje būtina išreikštai kviesti klientinės programos perpiešimo metodą *repaint*. Minėta, kad šis automatiškai kviečiamas klientinei programai startuojant po *start* metodo ir kiekvienąkart, kai jos langas tampa matomas (pavyzdžiui, sutraukus langą – išplėtus langą). Kitais atvejais, ką nors pakeitus lange

ir norint iškart matyti pokyčius, būtinas raiškus kvietimas. Visa piešimo metodų kvietimo grandinė yra: *repaint* kviečia *update*, o šis – *paint*.

Taigi sprendimas klientinės programos langui perpiešti akimirksniu yra toks: suformuoti ir išsaugoti identifikatoriuje objektą, kurį reikės perpiešti; kai reikalingas objekto perpiešimas – kviesti *repaint*; perrašyti metodą *paint*, numatant reikiamų objektų išvedimą. Metodas *paint* turi vieną klasės *Graphics* tipo argumentą, kuriame saugomas visas grafinis klientinės programos kontekstas.

Metodo *repaint* yra keturi variantai:

void repaint( ) – perpiešia visą langą.

void repaint(int left, int top, int width, int height) – perpiešia nurodytą (vaizdo taškais) lango dalį. Šiems variantams galimi keblumai, kai metodas kviečiamas per dažnai. Per dažni kvietimai bus atmesti įvykių apdorojimo mechanizmo, ir *update* bus kviečiamas tik laiko tarpais pagal nutylėjimą. Tai netinka judantiems vaizdams perteikti, todėl tam yra šie *repaint* variantai:

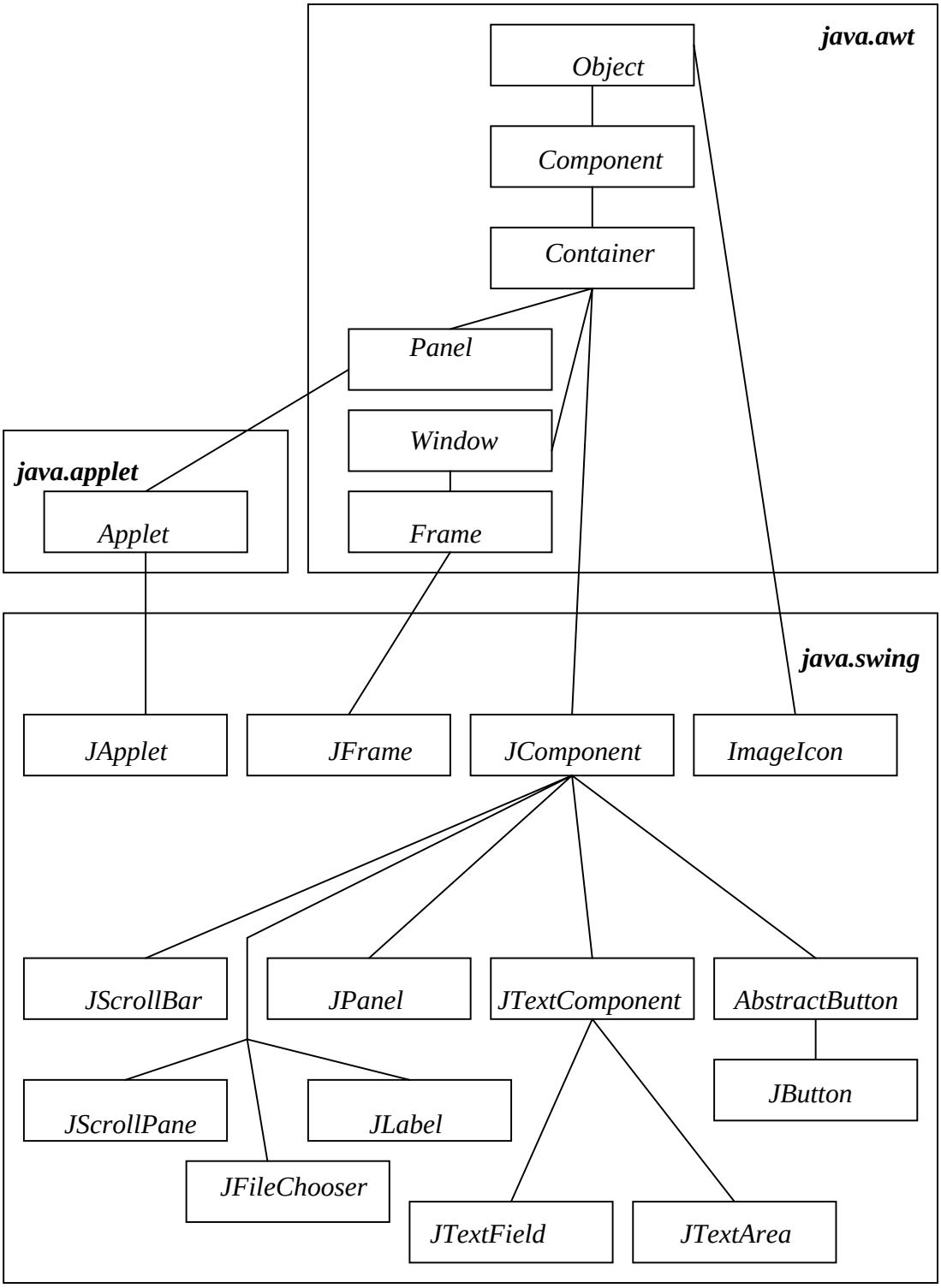
void repaint(long maxDelay) – čia *maxDelay* yra maksimalus laiko tarpas tarp *update* kvietimų milisekundėmis.

void repaint(int left, int top, int width, int height, long maxDelay)

Tačiau jei dėl sisteminių priežasčių metodas *update* negali būti kviečiamas nurodytais intervalais, metodas išvis nekviečiamas ir negeneruojama jokia išimtis – klientinės programos veikimas šiuo atveju tampa visiškai nesuprantamas.

3.1.5. Baziniai grafiniai komponentai

Rašant klientines programas, be *Swing* paketo komponentų klasių, reikalingi ir AWT pakete esantys komponentai. Pagrindinių klasių (*Swing* paketas yra labai didelis; čia apsiribojama tik keliomis jo klasėmis) hierarchija parodyta 3.1 pav.



3.1 pav. Paketų klasių hierarchija

Taigi *Swing* hierarchijos viršuje yra *JComponent* klasė. Ji paskelbta *abstract*, bet turi daugelį metodų. Visos jos subklasės yra trijų tipų:

1. Grafiniai komponentai (*graphical widgets*);
2. Tekstiniai komponentai;
3. Konteineriai.

Grafinių komponentų pavyzdžiai – mygtukas *JButton*, slankiklis *JScrollBar*, rinkmenų medis *JFileChooser*. Jiems panaudoti faktiškai pakanka žinoti metodus-konstruktorius. Viena išimtis čia – piktogramos užrašas *ImageIcon*, plečiantis tiesiai *Object* klasę.

Tekstiniai komponentai – tekstinis laukelis *JTextField*; jame galima pateikti tekstą ir įvesti duomenis, tekstinė sritis iš kelių eilučių *JTextArea*.

Konteineriai – juose privalo būti išdėstyti komponentai. Yra dviejų tipų konteineriai:

1. Aukštesniojo lygio (*JFrame, Window*);
2. Žemesniojo lygio (*JPanel, JScrollPane*).

Aukštesniojo lygio konteineriai negali būti įdedami į kitus konteinerius. Žemesniojo lygio konteineriai – gali, vieną *JPanel* konteinerį galima dėti į kitą, o šį dar į kitą ir t. t. Taip galima suformuoti bet kokio grafinio vaizdo vartotojo sąsajas. Komponentų įdėjimas į konteinerius – sudėtingas procesas. Dažniausiai taikomas toks būdas: visi žemo lygio konteineriai ir grafiniai bei tekstiniai komponentai metodu *add* sudedami į *Container* egzempliorių, kuris gaunamas kaip parodyta ankstesniuose klientinių programų pavyzdžiuose:

Container c = getContentPane( )

Konteineriai *Window, Frame, JFrame* pirmiausia skirti savarankiškoms kalbos aplikacijoms, leidžiamoms savo kompiuteryje, kiti – klientinėms programoms, leidžiamoms kituose kompiuteriuose. Skiriasi šių konteinerių saugumo galimybės, todėl ankstesnėse *Java* versijose, klientinėje programoje sukūrus, pavyzdžiui, *JFrame* langą, net buvo išvedamas įspėjimas "*Warning: This is an applet window*".

3.1.6. Komponentų išdėstymo konteineryje tvarkyklės

Pati tvarkyklė sprendžia, kaip išdėstyti konteineryje grafinius komponentus. Komponentų dydis, forma, išdėstymo būdas skirsis naudojant skirtingas tvarkykles. Pakeitus klientinės programos lango matmenis, pakis konteinerio matmenys – tvarkyklė pakeis ir komponentų išdėstymą. Yra tokios išdėstymo tvarkyklės:

BorderLayout
BoxLayout
CardLayout
FlowLayout
GridBagLayout
GridLayout
OverlayLayout
ScrollPaneLayout
ViewportLayout

Klasė *Container* turi metodą *setLayout*, nustatantį konteineriui norimą tvarkyklę. Rodyklė į *Container* objektą gaunama metodu *getContentPane*. Grafiniai komponentai leidžia hierarchinę jų sandarą: pavyzdžiui, *JPanel* – objekte galima sudėti kitus žemesniojo lygio *JPanel* objektus ir t. t. Šį išdėstymą taip pat valdo tvarkyklės.

Pagal nutylėjimą klientinės programos naudoja tvarkyklę *BorderLayout.* Ji komponentą deda į konteinerio centrą ir „ištempia“ iki konteinerio matmenų. Komponentas įdedamas į konteinerį metodu *add(Component c)* arba *add(BorderLayout.CONSTNAME, Component c*); čia *static final int* konstanta *CONSTNAME* nustato vietą, kur padedamas komponentas. Yra tokios konstantos reikšmės (fizinė jų prasmė aiški iš pavadinimų): *NORTH, SOUTH, EAST, WEST, CENTER*.

Tvarkyklė *FlowLayout* dėsto komponentus eilute iš kairės į dešinę, paskui į kitą eilutę ir t. t. Naudojamas vienodas (arba reikiamas – jei tą nustato, pavyzdžiui, užrašas ant mygtuko) komponentų dydis.

Tvarkyklė *GridLayout* išdėlioja komponentus lentelės forma. Lentelės eilučių ir stulpelių kiekis nustatomas konstruktoriuje: *GridLayout(int rows, int columns).*

BoxLayout atskirai skirsto komponentus vertikaliąja ir horizontaliąja kryptimis. Galima specialiais metodais nurodyti reikiamus tarpelius tarp komponentų (vadinamieji *strut* ir *glue*).

Pati sudėtingiausia tvarkyklė – *GridBagLayout*. Jos aprašas pateikiamas tik specializuotose *Swing* paketui skirtose knygose.

3.1.7. Kai kurie *Swing* komponentai

Tekstinis laukas *JTextField*, tekstinė sritis *JTextArea*. Abi klasės plečia *JTextComponent* klasę. Tekstinio lauko klasė leidžia perteikti ir redaguoti vieną teksto eilutę. Galimi konstruktoriai, suteikiantys laukui reikšmę ir jos nesuteikiantys, nurodantys lauko ilgį ir jo nenurodantys:

JTextField( )
JTextField(int columns)
JTextField(String s, int columns)
JTextField(String s).

JTextArea leidžia redaguoti kelias teksto eilutes. Tekstinės srities klasės konstruktoriai – panašūs. Tekstinę sritį galima „įvilkti“ į vertikaliuosius ir horizontaliuosius slankiklius. Tai atliekama konstruojant slankiklio objektą konstruktoriais

JScrollBar(Component c)
JScrollBar(Component c, int VERTSB, int HORSB).

Sąsajų klasėje *ScrollPaneConstants* nustatytos tokios šių *static final int* konstantų, valdančių vertikalųjį ir horizontalųjį slankiklius, reikšmės (fizinė jų prasmė aiški iš pavadinimų):

VERTICAL_SCROLLBAR_ALLWAYS
VERTICAL_SCROLLBAR_AS_NEEDED

VERTICAL_SCROLLBAR_NEVER
HORIZONTAL_SCROLLBAR_ALLWAYS
HORIZONTAL_SCROLLBAR_AS_NEEDED
HORIZONTAL_SCROLLBAR_NEVER

Į konstantas galima kreiptis ir sąsajų klasę realizuojančios klasės vardu, pavyzdžiui, *JScrollPane.VERTICAL_SCROLLBAR_ALWAYS* – vertikalusis slankiklis bus visada rodomas.

1 pavyzdys. Lange išdėstomi du mygtukai ir teksto sritis. Vienas mygtukas į tekstinę sritį įkelia tekstą, esantį *hash*-lentelėje (žr. kolekcijų skyrių), kitas – tekstą trina. Tekstinė sritis įvilkta į slankiklį; tačiau slankiklis bus parodytas tik tada, kai reikės – jei viso teksto tekstinėje srityje jo parodyti negalima.

```
// Example:  JTextArea, collection HashMap
//
//<applet code = TextArea width = 500 height = 500>
//</applet>
//
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;  //all collections are saved here

public class TextArea extends JApplet {
  JButton b1   = new JButton( "Add data" ),
          b2   = new JButton( "Delete data" );
  JTextArea ta = new JTextArea( 20,40 );
  HashMap hm   = new HashMap( );
  public void init( ){
    hm.put( "a",new Double( 1. ) );
    hm.put( "b",new Double( 2. ) );
    hm.put( "c",new Double( 3. ) );
    hm.put( "d",new Double( 4. ) );
    hm.put( "e",new Double( 5. ) );
    hm.put( "f",new Double( 6. ) );
    b1.addActionListener( new ActionListener( ){
      public void actionPerformed( ActionEvent ae ){
        Set s = hm.entrySet( ); //all entries (pairs key/value) of hashtable
                                // rendered to a set; iterator is available for a set
        Iterator i = s.iterator( ); //method returns iterator's object i
        while( i.hasNext( ) ){      //hasNext - method of class Iterator
          Map.Entry me = ( Map.Entry ) i.next( ); //method next returns object of
                                                  //type Object; downcasting
                                                  //Entry - inner class of Map
          ta.append( me.getKey( ) + " : "
                  +me.getValue( ) + "\n" ); //method append (JTextArea) appends
                                            //current text to the existing text
        }
      }
```

```
    } );
    b2.addActionListener( new ActionListener( ){
      public void actionPerformed( ActionEvent ae ){
        ta.setText( "" );      //ie, deletes text
      }
    } );
    Container c = getContentPane( );
    c.setLayout( new FlowLayout( ) );
    c.add( new JScrollPane( ta ) );       //text area wrapped into a scrollbar
    c.add( b1 );
    c.add( b2 );
  }
}
```

Etiketės *JLabel*: grafinis objektas su tekstu. Galima sukurti etiketę su ženkleliu *Icon*.

Vėliavėlės *JCheckBox*: keturkampis ir etiketė šalia. Vėliavėlė gali būti „parinkta“: joje bus pavaizduotas ženklas „X“.

Perjungikliai *JRadioButton*: jie grupuojami į grupes; joje įjungtas gali būti tik vienas jungiklis. Parenkant vieną iš jungiklių, buvęs įjungtas automatiškai išjungiamas.

„Krentantieji“ sąrašai *JComboBox*: leidžia parinkti vieną sąrašo elementą, o **sąrašai JList** – keletą, klavišų *Shift* ar *Ctrl* pagalba. Šių sąrašų elementai yra visą laiką matomi.

Skirtingi meniu tipai: paprastasis meniu *JMenu*, „išsiskleidžiantis“ meniu *JPopupMenu*, „lentelinis“ meniu *JTabbedPane*.

Sudėtingiausi grafiniai objektai yra **medžiai *JTree***, realizuojantys tokią grafinę struktūrą, kaip OS aplankų medis ir lentelė *JTable*.

Bet kuriam komponentui galima priskirti paaiškinamąjį tekstą, pasirodantį užlaikius pelės žymeklį virš komponento: metodas *setToolTipText(String s)*. Bet kurį komponentą galima įrėminti metodu *setBorder( )* su daugybe nustatytų *static final int* konstantų.

2 pavyzdys: klientinė programa, realizuojanti *JTable* klasę. Čia minimos tik kelios lentelės savybės. Lentelei taikomų metodų sąrašas siekia per 400. Klientinė programa iš rinkmenos, esančios tame kompiuteryje, iš kurio pati įkraunama (kompiuterių ryšio galimybes žr. skyriuje apie bazinį tinklinį programavimą) nuskaito dvimatį masyvą, kuris bus dedamas į lentelę. Vienmatis masyvas nustato lentelės stulpelių pavadinimus. Lentelės ląstelėms priskiriamas aiškinamasis tekstas, pasirodantis virš ląstelės užlaikius pelės žymeklį. Lentelės eilutei išskirti iš lentelės pasinaudota metodu *getPoint*, grąžinančiu *Point* klasės objektą *p* – fizines taško koordinates vaizdo taškais, o paskui metodu *rowAtPoint(p)* gaunamas eilutės indeksas.

```
/*Applet with some new components:
  JTable, JPanel, tool-tip-text.
  Input file on the same directory as code:
  collection of lines, each of 3 String-type
  elements divided by space. Later on first two elements
  of each line are put to the array1 while the third one to array2.
  array1 makes the model for the table; array2 plays role of
  explanatory text.
  Name of file may be supplied in the text field.
  All errors from error stream are put to the text area. */

//<applet code = Table width =500 height=550>
//</applet>

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class Table extends JApplet{
  String array1[ ][ ] = new String[ 20 ][ 2 ],
           array2[ ]    = new String[ 20 ],
          colHeads[ ] = { "Number", "Name" };
  JButton b1    = new JButton( "Add data" ),
            b2    = new JButton( "Delete data" );
  JTextField tf = new JTextField( "Input.txt", 20 );
  JTextArea ta  = new JTextArea( 3, 40 );
  JTable table  = new JTable( array1, colHeads ){
    public String getToolTipText( MouseEvent me ){ //overriding method
       String tip = "";                             //that sets "tip-text"
       Point p = me.getPoint( );                    //to the table
       int row = table.rowAtPoint( p );
       tip = array2[ row ];
       return tip;
    }
  };

  class ReadingFile implements ActionListener{
   public void actionPerformed( ActionEvent ae ){
    try{
     URL url = new URL( getCodeBase( ), tf.getText( ) );
     InputStream is = url.openStream( );
     BufferedReader in = new BufferedReader( new InputStreamReader( is ));
     String line = "";
     int count = 0;
       while( (line = in.readLine( )) != null ){
         array2[count] = line.substring( line.lastIndexOf( " " )+1, line.length( ) );
         array1[count][0] = line.substring( 0, line.indexOf( " " ) );
```

```
        array1[count][1] = line.substring( line.indexOf( " " )+1,
                                    line.lastIndexOf( " " ) );
        count++;
       }
    } catch( ArrayIndexOutOfBoundsException e ){
       ta.append( "Dimensions of table only until 20 rows allowed" );
    } catch( Exception e ) {
       ta.append( "Error reading data: "+ e );
    }
    table  = new JTable( array1, colHeads );
    showStatus( "Data added" );
    repaint( );
   }
  }

  class DeletingData implements ActionListener{
   public void actionPerformed( ActionEvent ae ){
    for( int i=0;  i<20;  i++ ){
      array1[i][0] = "";
      array1[i][1] = "";
      array2[i]    = "";
    }
    table  = new JTable( array1, colHeads ); //modifying table
    showStatus( "Data deleted" );
    repaint( );
   }
  }

  public void init( ){
    b1.addActionListener( new ReadingFile( ) );
    b2.addActionListener( new DeletingData( ) );
      Container c = getContentPane( );
      JScrollPane spt = new JScrollPane( table );
        JPanel p1 = new JPanel( );
        p1.setLayout( new FlowLayout( ) );
        p1.add( tf );
        p1.add( b1 );
        p1.add( b2 );
          JPanel p2 = new JPanel( );
          JScrollPane spta = new JScrollPane( ta );
          p2.add( spta );
            c.add( spt,BorderLayout.NORTH );
            c.add( p1,BorderLayout.CENTER );
            c.add( p2,BorderLayout.SOUTH );
   }
  }
```

Langai *JFrame, JDialog*. Jei klientinėje programoje reikia papildomo, šalia klientinės programos lango pasirodančio lango, paprasčiausia sukurti *JFrame* objektą (nors, kaip minėta, šios klasės objektai pirmiausia skirti savarankiškoms *Java* aplikacijoms). *JFrame* langas jau nepriklauso nuo naršyklės. Tokiems langams galima priskirti savus, atskirus įvykius. Tokius langus uždarant, klientinė programa darbo nebaigia, tik atlaisvinami šiam langui buvę paskirti ištekliai (metodas *windowClosing*, o jame kviečiamas metodas *dispose*). Galima tokius langus padaryti matomus arba nematomus metodu *setVisible* su argumentais atitinkamai *true* arba *false*.

Panašiai darbas organizuojamas ir su dialogo langais – klasės *JDialog* objektais.

3 pavyzdys: klientinė programa su *JFrame* langu. Klientinė programa sukuria papildomą langą (klasė *Boxes*), padalytą į atskirus laukelius, ir kiekvieną laukelį nuspalvina atsitiktine spalva (metodas *gettingColor*, o šiame jau naudojamas vienas iš *Java* spalvos kūrimo konstruktorių). Laukelis spalvinamas *Graphics* klasės metodu *fillRect*, nurodyta spalva užpildančiu nurodyto dydžio stačiakampį. Nurodytu intervalu visi laukeliai keičia savo spalvą. Kiekvienas toks laukelis paleidžiamas gyvuoti atskiru srautu, todėl šia klientine programa galima nustatyti, kiek kompiuteris realiai gali palaikyti atskirų srautų. Duomenų (laukelių kiekis, nurodomas dviem dydžiais: eilučių ir stulpelių skaičiais; mirksėjimo dažnis, nurodomas pauze milisekundėmis) įvedimo operacijos atliekamos atidalytoje nuo likusios programos dalies grafinėje sąsajoje jau žinomomis technologijomis.

```
//Blinking random-colored boxes. <grid*grid> boxes in all.
//Blinking rate: <pause>, in milisec.
//Applet launches individual thread for each box.
//Applet illustrates the efficiency of multitasking.
//
//<applet code = MThreads width = 500 height = 500>
//</applet>
//

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Box extends JPanel implements Runnable{ //separate colored box.
  private Thread t;                                  //Rendered in separate thread.
  private int pause;
  boolean running = true;
  private Color clr = gettingColor( );

  private Color gettingColor(){     //random color for the box
    int red    = ( int ) ( Math.random( ) * 255 ),
        green = ( int ) ( Math.random( ) * 255 ),
        blue   = ( int ) ( Math.random( ) * 255 );
    return new Color( red,green,blue );
  }
```

```
 public void paintComponent( Graphics g ){  //painting the colored box
   super.paintComponent( g );
   g.setColor( clr );
   Dimension d = getSize( );
   g.fillRect( 0, 0, d.width-1, d.height-1 );
  }

  public Box( int pause ){
   this.pause = pause;
   t = new Thread( this );
   t.start();
  }

  public void run( ){
   while( running ){ //will be stopped by closing frame-window for all boxes
     clr = gettingColor( );  //re-coloring the box
     repaint( );
     try{
       t.sleep( pause );    //stops thread for <pause> milisec
     } catch( InterruptedException ie ){ }
   }
  }
}

class Boxes extends JFrame{  //panel for <grid*grid> boxes
  int grid, pause;
  public Boxes( int grid, int pause ){
   this.grid = grid;
   this.pause = pause;
   setSize( 300,300 );
   Container cf = getContentPane( );
   cf.setLayout( new GridLayout( grid, grid ) );
   for( int i=0; i<grid*grid; i++ )
      cf.add( new Box( pause ) );
   addWindowListener( new WindowAdapter( ){
     public void windowClosing( WindowEvent we ) {
       dispose( );      //responce to the frame-window buttons: close the window
     }
   } );
  }
}

public class MThreads extends JApplet{  //monitor program
  private int grid = 5;
  private int pause = 100;
  JLabel l1 = new JLabel( "Enter number of rows and columns" ),
         l2 = new JLabel( "Enter delay time (milisec)" ),
         l3 = new JLabel( "Messages" );
  JTextField t1 = new JTextField( 10 ),
             t2 = new JTextField( 10 );
```

```
JTextArea  ta = new JTextArea( 3,20 );
JButton b1 = new JButton( "Start" ),
          b2 = new JButton( "Reenter" );
JFrame boxes;

public void init( ){
  Container c = getContentPane( );
  JPanel p11 = new JPanel( ),
           p12 = new JPanel( );
  p11.setLayout( new GridLayout( 3,3 ) );
  p12.setLayout( new FlowLayout( ) );
  t1.addActionListener( new ActionListener( ){
    public void actionPerformed( ActionEvent ae ){
      ta.setText( "" );
      String s = t1.getText( );
      try{
        grid = Integer.parseInt( s );
      } catch( NumberFormatException nfe ){
        ta.append( "Erroneous input for number of rows." +
                     " Reenter or use default: 5 \n" );
        grid = 5;
      }
      if( grid<1 ) {
        ta.append( "Erroneous input for number of rows \n" +
                     "Using default: 5  \n" );
        grid = 5;
      }
      ta.append( "Entered: number of rows  " + grid + "\n" );
    }
  });
  t2.addActionListener( new ActionListener( ){
    public void actionPerformed( ActionEvent ae ){
      ta.setText( "" );
      String s = t2.getText( );
      try{
        pause = Integer.parseInt( s );
      } catch( NumberFormatException nfe ){
        ta.append( "Erroneous input for pause." +
                 " Reenter or use default: 100 milisec \n" );
        pause = 100;
      }
      if( pause<1 ) {
        ta.append( "Erroneous input for pause \n" +
                     "Using default: 100 milisec \n" );
         pause = 100;
      }
      ta.append( "Entered: pause (milisec)  "+ pause + "\n" );
    }
  });
  p11.add( l1 ); p11.add( t1 ); p11.add( l2 ); p11.add( t2 );
```

```
   p11.add( l3 ); p11.add( ta );

   b1.addActionListener( new ActionListener( ){
    public void actionPerformed( ActionEvent ae ){
      boxes = new Boxes( grid, pause );
      boxes.setVisible( true );
    }
   });
   b2.addActionListener( new ActionListener( ){
    public void actionPerformed( ActionEvent ae ){
      t1.setText( "" );
      t2.setText( "" );
      ta.setText( "" );
      boxes.dispose( );
      repaint( );
    }
   });
   p12.add( b1 );
   p12.add( b2 );

    c.add( p11,BorderLayout.NORTH );
    c.add( p12,BorderLayout.SOUTH );
  }
}
```

3.1.8. Klientinių programų archyvavimas

Kaip matyti iš ankstesnių pavyzdžių, klientinės programos gali būti nemažos apimties, apimti daugialypės terpės elementus. Tokias klientines programas reikėtų archyvuoti į *Java* archyvus – *jar* rinkmenas. Juose yra visos reikiamos klasės kartu su daugialypės terpės rinkmenomis.

Archyvavimo komandos paprasčiausia forma:

```
jar   cf   archiveName.jar   *.class
```

čia *jar* – pagalbinės programos-archyvatoriaus vardas, yra pakete *java.util*.
c – komanda: sukurti naują archyvą.
f – komanda, nurodanti, kad pirmasis vardas toliau bus archyvo vardas.
**.class* – reiškia, kad pakuojamos rinkmenos yra darbiniame aplanke.

Ruošiant HTML dokumentą suarchyvuotai klientinei programai, sakinyje <*applet*> dabar tektų nurodyti tokią papildomą informaciją:

```
<applet code = publicClassName.class   archive = archiveName.jar
                width = …   height = …
</applet>
```

3.2. *Java Beans* technologija

Technologija leidžia formuoti sudėtingas programines sistemas iš atskirų programinių komponentų, vadinamųjų *beans*. *Java Beans* – architektūra, nusakanti, kaip šie komponentai turi kartu dirbti. Technologija pirmiausia naudojama „vizualiniam programavimui“, labai pagreitinančiam kodo kūrimą. Tipinis vizualinio programavimo įrankis turi paletę su komponentų sąrašu, paletę, kurioje bus dedami komponentai, ir paletę, kurioje komponentas bus konfigūruojamas – pritaikomi įvykiai, charakteristikos. Komponento konfigūracija gali būti išsaugota atmintyje ir vėliau atkurta. Programuojant vizualiniu įrankiu, pavyzdžiui, grafinę sąsają, tereikia į darbinę paletę pele „pertempti“ reikiamus komponentus, juos reikiamai išdėlioti ir sukonfigūruoti. Programos kodas generuojamas automatiškai. Vienas tokių įrankių – *Bean Development Kit* (BDK), www.javasoft.com.

Komponentas *bean* yra paprasta klasė, užprogramuota laikantis tam tikrų taisyklių. Pirmiausia visi klasės laukai turi būti paskelbti privačiais. Laukų reikšmes galima gauti ir nustatyti tik metodais

public type getField( ) ir
public void setField( ),

čia *Field* – lauko identifikatorius, o *type* – lauko tipas. Indeksuoti laukai privalo turėti po du metodus:

public type getField(int index)
public type[ ] getField( )

ir analogiškai metodus *set*.

Kitiems metodams ši vardų taisyklė negalioja. *private* ir *protected* metodai grafiniams programavimo įrankiams neprieinami – *Java* savianalizės sistema jų neranda. Įvykių sistema lygiai tokia, kokia naudojama *Swing* pakete. Kur tik įmanoma, *public* metodai turi būti skelbiami *synchronized*, kad komponentą būtų galima saugiai naudoti daugiasrautėje aplinkoje.

Kad komponento būklė būtų išsaugota, jis turi realizuoti sąsajų klasę *Serializable*. Šoje sąsajų klasėje nėra nė vieno metodo; jis tik nurodo galimybę užrašyti objektą baitinio srauto pavidalu, pavyzdžiui, rinkmenoje. Objekto užrašymo baitiniu srautu procesas vadinamas serializavimu. Vėliau, atkuriant objektą, jį galima grąžinti į pradinį pavidalą. Serializacijos metu užrašomas ir pats objektas, ir visi subobjektai, į kuriuos objekte buvo rodyklės. Tai dar vadinama „giliuoju kopijavimu“. Serializuojant neišsaugomos *static* ir *transient* laukų reikšmės.

Komponento pavyzdys (P. Naughtonas): piešiamas keturkampis arba ovalas ir nuspalvinamas atsitiktine spalva. Iš pradžių nupiešiamas ovalas. Mygtuku, kuris būtų susietas su metodu *change*, keičiama spalva. Spalvą taip pat galima keisti, nuspaudus pelės klavišą panelio ribose. Toliau, ar vaizduoti stačiakampį, ar ovalą – būtų galima nurodyti konfigūruojant komponentą *Figure* grafinio programavimo įrankio galimybėmis. Konfigūravimo paletėje būtų automatiškai suformuoti mygtukai lauko *rectangular* reikšmei keisti, taip pat foninei spalvai ir panelio spalvai nustatyti.

```
// Java Bean: figure of random color
//
package figureBean;  //CLASSPATH must point to the directory of figureBean!
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Figure extends JPanel implements Serializable{
        transient private Color color;  //this will not be kept during serialization
        private boolean rectangular;

        public Figure( ){
                addMouseListener( new MouseAdapter( ){
                        public void mousePressed( MouseEvent me ){
                                change( );
                        }
                } );
                rectangular = false;  //the oval will be drawn for the first
                setSize( 200, 100 );
                change( );              //for the first coloring
        }

        public synchronized boolean getRectangular( ){
                return rectangular;
        }

        public synchronized void setRectangular( boolean flag ){
                rectangular = flag;
                repaint( );  //to see changes immediately
        }

        public synchronized void change( ){
                int red      =( int ) ( 255 * Math.random( ) ),
                    green  = ( int ) ( 255 * Math.random( ) ),
                    blue    = ( int ) ( 255 * Math.random( ) );
                color = new Color( red, green, blue );
                repaint( );
        }

        public void paint( Graphics g ){     //not synchronized! Metod paint is set
                Dimension d = getSize( );  //as non-synchronized in the  language
                int h = d.height;
                int w = d.width;
                g.setColor( color );
                if( rectangular )
                        g.fillRect( 0, 0, w-1, h-1 );
                else
                        g.fillOval( 0, 0, w-1, h-1 );
        }
}
```

Taigi komponentas – nesavarankiška programa, jame nėra nei metodo *init*, nei metodo *main*. Komponento paleidimu pasirūpina grafinio programavimo įrankis.

Tolesnė darbo su komponentu eiga:

1. Kompiliavimas.
2. Vadinamosios manifesto rinkmenos sudarymas komponentui:

Manifest – Version: 1.0
Name: figureBean/figure.class
Java – Bean: True

Ši rinkmena vardu *Figure.mf* įrašoma į aplanką laipteliu aukštesnį nei *figureBean*.

3. Visų komponento rinkmenų archyvavimas:

jar cfm Figure.jar Figure.mf figureBean

čia *c* – komanda sukurti naują archyvą;
f – komanda, nurodanti, kad pirmasis vardas bus archyvo vardas;
m – kad antrasis vardas yra manifesto rinkmenos vardas.
figureBean – aplanko, kuriame reikia ieškoti visų kitų reikiamų klasių, vardas.
Archyvatorius supakuos į archyvą visas rinkmenas, kurias ten ras.

4. *jar* rinkmena įrašoma į vizualinio programavimo įrankio reikalaujamą rinkmeną. Tolesnis darbas priklauso nuo įrankio komandų.

Grafiniai įrankiai komponentų savybėms nustatyti naudoja savianalizės sistemą – tiria komponento *class* objektus. Panašiems dalykams anksčiau parašėme savo programą. Pakete *java.beans* tam yra ir *Java* įrankis – klasė *Introspector*, o ši turi metodą *static BeanInfo getInfo(Class c)*, kur klasės *BeanInfo* objektas turi išsamią informaciją apie komponento *class* objektą *c*.

3.3. Bazinis tinklinis programavimas

3.3.1. Ryšys tarp dviejų kompiuterių tinkle

Visos bazinio tinklinio programavimo galimybes realizuojančios klasės yra pakete *java.net*.

Java realizuojamas ryšys tarp dviejų kompiuterių tinkle yra labai panašus į įprastą darbą su įvedimo bei išvedimo srautais, tik tie srautai gaunami ar nukreipiami iš arba į jungtį tarp kompiuterių – vadinamųjų lizdų (*sockets*).

Kad vienas serveris galėtų aptarnauti kelis klientus, tenka naudoti *Java* daugiasrautiškumo galimybes.

Kompiuteris-serveris laukia, kol prie jos pasijungs kompiuteris-klientas, „klausydamas“ tinklo tam tikrame prievade. Laukimo ir tinklo klausymo funkcijas

atlieka speciali *Java* klasė *ServerSocket,* kurios darbo rezultatas (faktiškai – jos objekto metodo *accept* darbo rezultatas) gali būti jungtis tarp mašinų – *Socket*. Iš kliento pusės jungtį stengiasi sukurti klasės *Socket* objektas. *Socket* konstruktoriui reikia nurodyti serverio IP adresą ir prievado numerį. Kai jungtis sėkmingai sudaryta, skirtumo tarp serverio ir kliento nebelieka: atliekami paprasti duomenų mainai. Tam naudojami metodai *getInputStream* ir *getOutputStream,* sukuriantys baitinius srautus *InputStream* ir *OutputStream*, norint galima juos įvilkti į simbolinius srautus.

Kaip minėta, klientas, kad prisijungtų prie serverio, turi žinoti serverio IP adresą. Tai – 4 skaičiai nuo 0 iki 255, atskirti taškais. Šį adresą, jei žinomas kompiuterio-serverio vardas arba jo DNS adresas, galima gauti statiniu klasės *InetAddress* metodu *getByName(String computerName)* arba *getByName(String DNSAddress).*

Savo kompiuterio IP adresą ir vardą galima sužinoti naudojantis OS priemonėmis. Tinklinėms programoms testuoti numatyta galimybė simuliuoti tinklą vien tik savo kompiuteriu. Tam naudojamas specialus IP adresas, atitinkantis vadinamąją *localhost* mašiną. Šis adresas gali būti gautas tokiais būdais: *InetAddress.getByName("localhost")* arba *InetAddress.getByName(null).*

ServerSocket (serverio pusėje) ir *Socket* (kliento pusėje) objektams kurti dar reikia žinoti jungties prievado numerį. Prievadas – programinė abstrakcija, naudojama atliekant žemesnį, antrojo lygio adresavimą: juk kiekvienas kompiuteris gali palaikyti daugelį serverių, todėl vien tik jos IP adreso jungčiai sukurti nepakanka. Kiekvienas serveris tam tikrame prievade gali teikti tam tikras paslaugas klientui. Prievadai nuo 1 iki 1024 rezervuoti OS reikmėms: pavyzdžiui, 21-as prievadas skirtas FTP protokolui, 25-as – elektroniniam paštui, 80-as – HTTP protokolui.

Kaip pavyzdį pateiksime paprasčiausio serverio ir paprasčiausio kliento programas. Abu sukuriami tame pat kompiuteryje, naudojant kompiuterį *localhost* atitinkantį IP adresą.

```
//The simplest server: getting messages from
//client and printing them
//

import java.io.*;
import java.net.*;

public class SimpleServer{
  public static final int PORT = 2000;    //number is our arbitrary choice
  public static void main( String args[ ] ) throws IOException{
    ServerSocket ss = new ServerSocket( PORT );
    System.out.println( "Launched: " + ss );
    try{
      Socket s = ss.accept( );    //waiting for connection
      try{
        System.out.println( "Connection got: " + s );
        BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                        s.getInputStream( ) ) );
        PrintWriter out =   new PrintWriter(
```

```
                    new BufferedWriter(
                    new OutputStreamWriter(
                    s.getOutputStream( ) ) ),true );
      while( true ){                    //server will be closed after
       String str = in.readLine();      //client's message "END"
       if( str.equals( "END" )) break;
       System.out.println( "Got from client: " + str );
       out.println( str );
      }
     } finally{                         //socket must be closed!
       System.out.println( "Disconnecting client" );
       s.close( );
     }
    } finally{                          //socket must be closed!
      System.out.println( "Closing server" );
      ss.close( );
    }
  }
}
```

Serveris laukia kliento prisijungimo. Prisijungus klientui, skaito kliento siunčiamas eilutes ir persiunčia jas atgal klientui. Toks siuntinėjimas vyksta, kol kliento siunčiamame duomenų sraute aptinkama eilutė *END*. Serverio darbo rezultatus parodysime tik po analogiškos kliento programos analizės, nes kartu funkcionuoti privalo abi šios programos.

Kaip matyti, *ServerSocket* objektas kuriamas nurodant konstruktoriuje tik prievado numerį (jį pasirenkame didesnį nei 1024), o IP adresas nereikalingas – juk serveris kuriamas mūsų kompiuteryje. *Socket* objektas gaunamas *ServerSocket* metodu *accept*. Kol *accept* negrąžina *Socket* objekto – programos veikimas blokuojamas, ji laukia, kol pasijungs klientas.

Kai tik *ServerSocket* ir *Socket* objektai sukuriami, spausdinama informacija apie šiuos objektus į pultą, naudojant neišreikštą metodo *toString* kvietimą.

Iš *Socket* objekto gaunamas baitinis įvedimo srautas *in*, kuris pertvarkomas į simbolinį buferizuotą srautą. Į *Socket* objektą taip pat nukreipiamas simbolinis buferizuotas srautas *out*; čia reikalingas dar vienas papildomas antstatas *PrintWriter*, nes tik šioje klasėje yra mums reikalingas metodas *println*. *PrintWriter* objektui kurti būtinas ir antrasis *boolean* tipo argumentas, kurio reikšmė turi būti *true*. Tik tokiu atveju iš buferio į srautą bus perkelti duomenys kiekvienąkart suveikus metodui *println*, t. y. suradus eilutės laužimo simbolį *\n*. Jei antrojo argumento reikšmė būtų *false* arba būtų naudotas konstruktorius be antrojo argumento – prieš įkeliant duomenis į srautą, t. y. formuojant duomenų paketą siuntimui – būtų laukiama, kol buferis prisipildys. Tokiai programai, kaip aprašytoji, tai netinka: duomenų nedaug, buferis neprisipildys, duomenys nebus išsiųsti. Jei būtų siunčiami kitokie duomenys, pavyzdžiui, dideli baitiniai srautai – efektyviau naudoti automatinį duomenų perkėlimą į buferį. Taip būtų suformuoti didesni siuntimo paketai, procesas vyktų greičiau.

Dėl *try* blokų. *ServerSocket* kūrimo nebūtina įdėti į tokį bloką. Jei nepavyktų sukurti *ServerSocket* objekto konstruktoriumi, metodas *main* tiesiog generuotų išimtį *IOException* ir programos darbas būtų baigtas. *Socket* objekto kūrimas įvilktas į *try*-

finally konstrukcijos bloką, nes, serveriui baigus darbą su klientu, būtina jį atjungti – užverti jungtį. Tada išjungiamas ir serveris – užveriamas *ServerSocket* objektas.

Klientas:

```
//The simplest client: sending messages to
//server and getting messages from server
//

import java.io.*;
import java.net.*;

public class SimpleClient{
  public static void main( String args[ ] ) throws IOException{
    InetAddress ia = InetAddress.getByName( null ); //ie, server is on the
                                                    //same machine
    System.out.println( "Address of server: " + ia );
    Socket s = new Socket( ia, SimpleServer.PORT ); //code "SimpleServer" must
                                                    //be in the same package
    System.out.println( "Launched socket: " + s );
    try{
      BufferedReader in = new BufferedReader(
                    new InputStreamReader(
                    s.getInputStream( ) ) );
      PrintWriter out =   new PrintWriter(
                    new BufferedWriter(
                    new OutputStreamWriter(
                    s.getOutputStream( ) ) ),true );
      for( int i=0; i<10; i++ ){
        String msg = "Client's message " + i;
        out.println( msg );
        System.out.println( "Sent to server: " + msg );
        String str = in.readLine( );
        System.out.println( "Got from server: " + str );
      }
      out.println( "END" );              //"END" appointed for the finish
    } finally{
      System.out.println( "Closing client" );
      s.close( );
    }
  }
}
```

Klientas bando prisijungti prie serverio, esančio tome pačiame kompiuteryje, kaip ir serveris – *localhost*, todėl jo adresas nustatomas metodui *getByName* teikiant argumentą *null*. *Socket* objektui formuoti būtinas kitas argumentas – prievado numeris nurodomas *static final int* lauku *PORT*, apibrėžtu serverio programoje. Kad ši reikšmė būtų prieinama kliento programai, abi programos turi būti tame pačiame pakete. Nors joks paketas nedeklaruotas, taip ir bus, jei abi programas įdėsime į tą patį aplanką.

Kaip žinia, visos aplanke esančios rinkmenos priklauso bevardžiam paketui pagal nutylėjimą.

Prisijungęs prie serverio klientas nusiunčia serveriui 10 pranešimų ir eilutę *END*. Serveris klientui atgal persiunčia tokius pat pranešimus.

Visos kliento funkcijos įvilktos į *try-finally* bloką, kad bet kokiu atveju baigus darbą kliento objektas būtų uždarytas.

Abi šias programas reikia paleisti iš atskirų pultų. Pirma paleidžiamas serveris, vėliau – klientas. Pirmiau paleidus klientą, bus sugeneruota išimtis *java.net.ConnectException*.

Po serverio ir kliento paleidimo serverio pulte mūsų kompiuteryje matėme pranešimus:

Launched: ServerSocket[addr = 0.0.0.0./0.0.0.0., port = 0, localport = 2000]
Connection got: Socket[addr=/127.0.0.1, port = 4496, localport = 2000]
Got from client: Client's message 0
Got from client: Client's message 1
Got from client: Client's message 2
.
.
.
Got from client: Client's message 9
Disconnecting client
Closing server

Kliento pulte:

Address of server: localhost /127.0.0.1
Launched socket: Socket[addr = localhost /127.0.0.1, port = 2000, localport = 4496]
Sent to server: Client's message 0
Got from server: Client's message 0
Sent to server: Client's message 1
Got from server: Client's message 1
Sent to server: Client's message 2
Got from server: Client's message 2
.
.
.
Sent to server: Client's message 9
Got from server: Client's message 9
Closing client

Abiejų programų pranešimai pultuose pasirodys suderinti, klientui nusiuntus pranešimą serveriui – gavus iš jo atsakymą.

Kelios pastabos. Taigi jungtis tarp kompiuterių apibūdinama 4 dalykais: kliento IP adresu ir prievado numeriu, serverio IP adresu ir prievado numeriu. Kai paleidžiamas serveris, jis užima nurodytą prievadą (*2000*). Kai grąžinama jungtis su klientu (objektas *s*), jame yra informacija apie serverio IP adresą, jo prievadą

(*localport = 2000*) ir prievado, kurį užima klientas (*4496*), numerį. Šis adresas nuolat didinamas pradedant nuo 1025 iki kol bus perkrauta OS. Tokią pačią informaciją apie sukurtą jungtį spausdina ir kliento programa, tik kitokia tvarka: juk nagrinėjamai programai *port* yra taikinys, o *localport* – šaltinis. Tai, kas klientui yra *port*, serveriui bus *localport*, ir atvirkščiai.

Serveris privalo aptarnauti ne vieną, bet dešimtis klientų, todėl ankstesnė programa, aišku, netinkama. Ji tik demonstravo, kaip siųsti ir gauti pranešimus į kitą mašiną ir iš kitos mašinos. Realesnis serveris: sukuriamas vienas *ServerSocket* objektas ir kviečiamas metodas *accept*. Kai tik jis grąžina *Socket* objektą – sukuriamas srautas šio kliento aptarnavimui, o *ServerSocket* toliau laukia kito kliento, ir t. t. Veiksmai tarp serverio ir klientų atliekami lygiai tokie pat, kaip ir ankstesniame pavyzdyje.

Serverio programa:

```
//The simplest multi-client server: getting messages from
//clients and printing them
//One server-thread for one client
//

import java.io.*;
import java.net.*;

class ServeOneClient extends Thread{
  private Socket s;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
    public ServeOneClient( Socket s ) throws IOException{
      this.s = s;
      in = new BufferedReader(
           new InputStreamReader(
           s.getInputStream( ) ) );
      out = new PrintWriter(
            new BufferedWriter(
            new OutputStreamWriter(
            s.getOutputStream( ) ) ),true );
      start( );
    }
    public void run( ){
      try{
        while( true ){
          String str = in.readLine( );
          if( str.equals( "END" ) ) break;
          System.out.println( "Got from client: " + str );
          out.println( str );
        }
      } catch( IOException ioe ){
          System.err.println( "I/O error" );
```

```
    } finally{
       try{
         System.out.println( "Disconnecting client " + id );
         s.close( );
       } catch( IOException ioe ){
          System.err.println( "Error while closing socket" );
       }
    }
   }
}

public class SimpleMultiServer{
  public static final int PORT = 2000;
  public static void main( String args[ ] ) throws IOException{
    ServerSocket ss = new ServerSocket( PORT );
    System.out.println( "\n Press Ctrl/C for termination \n" );
    System.out.println( "Launched: " + ss );
    try{
      while( true ){                //endless loop!
        Socket s = ss.accept( ); //waiting for connection
        try{
          new ServeOneClient( s );
        } catch( IOException ioe ){
           s.close( );
        }
      }
    } finally{
       ss.close( );
    }
  }
}
```

Ir šioje programoje jungčių objektai privalo būti uždaryti bet kokiu atveju. Jei konkrečiam klientui nepavyksta sukurti srauto, – nesuveikia *ServeOneClient* konstruktorius – jungtis iškart uždaroma metodo *main* viduje. Jei sukuriamas, – jungtį uždaro srauto metodas *run*.

Kliento programa:

```
//Code generates simplest clients: sending messages to
//server and getting messages from server.
//Until MAX_CLIENTS clients may exist simultaneously.
//One thread is created for one client.
//
//Results of code depend exclusively on the period
//of pause. Experiment on it!
//

import java.io.*;
import java.net.*;
```

```
class OneClient extends Thread{
  private Socket s;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;              //id-number of client
  private static int threadCount = 0;   //number of "living" clients
  public static int threadCounting(){
    return threadCount;
  }
  public OneClient( InetAddress ia,int PORT ){
    System.out.println( "Creating client No " + id );
    threadCount++;           //count in this client
    try{
      s = new Socket( ia, PORT );
    } catch( IOException ioe ){
       System.err.println( "Error while creating socket No " + id );
    }
    try{
      in =  new BufferedReader(
                  new InputStreamReader(
                  s.getInputStream( ) ) );
      out = new PrintWriter(
                   new BufferedWriter(
                   new OutputStreamWriter(
                   s.getOutputStream( ) ) ),true );
      start();
    } catch( IOException ioe1 ){
       try{
         s.close( );
       } catch( IOException ioe2 ){
          System.err.println( "Unable to close socket No " + id );
       }
    }
  }
  public void run( ){
    try{
      for( int i=0; i<10; i++ ){
        String msg = "Client's No " + id + " message " + i;
        out.println( msg );
        System.out.println( "Sent to server: " + msg );
        String str = in.readLine( );
        System.out.println( "Got from server: " + str );
      }
      out.println( "END" );
    } catch( IOException ioe ){
       System.out.println( "I/O error" );
    } finally{
       try{
         s.close( );
```

```
        System.out.println( "Disconnecting socket No " + id );
      } catch( IOException ioe ){
        System.out.println( "Unable to close socket No " + id );
      }
      threadCount--;          //client is closed; decrease the number of clients
    }
  }
}

public class SimpleMultiClient{
  static final int MAX_CLIENTS = 3;
  public static void main( String args[ ] ) throws IOException, InterruptedException{
    InetAddress ia = InetAddress.getByName( null );
    System.out.println( "Address of server: " + ia );
    for( int i=0; i<20; i++ ){
      if( OneClient.threadCounting( ) < MAX_CLIENTS )
        new OneClient( ia, SimpleMultiServer.PORT );
      Thread.currentThread( ).sleep( 10 );
    }
  }
}
```

Vienu metu ši programa leidžia gyvuoti iki *MAX_CLIENTS* klientų. Programa bando kurti naujus klientus 20 kartų. Pagrindinis programos vykdymo srautas pristabdomas kuriam laikui metodu *sleep*, kad per šį laiką dalis klientų suspėtų atlikti savo darbą ir būtų uždaryti, o tada vietoje jų bus kuriami nauji klientai.

Vėl pirmiausia reikia paleisti serverio programą, paskui – kliento. Kad būtų paprasčiau, serverio programa yra begalinė, ir ją sustabdyti galima tik klavišais *Ctrl-C*. Programų leidimo rezultatai priklausys nuo vienu metu „gyvų“ klientų skaičiaus, pauzės laiko ir kompiuterio pajėgumo. Paeksperimentavus su šiais duomenimis, programų veikimas taps visiškai aiškus.

3.3.2. Ryšys tarp HTTP serverio ir kliento

Šiame skyriuje trumpai apžvelgiamos *Java* teikiamos galimybės atskiram serverio tipui – HTTP serveriui.

Kliento programos gali jungtis tik su kompiuteriu, iš kurio įkrautas kliento programos kodas. Aplikacijos gali jungtis su bet kuria tinklo mašina. Bet kuriuo atveju reikalingas URL adresas. URL adresą sudaro 4 komponentai:

1. Protokolas (*http, ftp, gopher, file*). Po jo seka 3 simboliai :// .
2. Kompiuterio IP adresas (arba jo DNS forma). Po jo seka 1 simbolis / arba : .
3. Prievado numeris.
4. Kelias į norimą rinkmeną.

Dažniausiai protokolo pavadinimą ir prievado numerį galima praleisti, ir lieka, pavyzdžiui, `www.nba.com/index.html` .

Yra keli URL adresų kūrimo konstruktoriai. Visi jie gali sukelti išimtį *MalformedURLException.* Serveryje esančio norimo aplanko ir norimos rinkmenos adresui kurti reikėtų naudoti šias konstruktoriaus formas:

```
URL url = new URL( getDocumentBase( ) ), String fileName )
URL url = new URL( getCodeBase( ) ), String fileName )
```

Pirmoji forma nustato URL adreso dalį serverio aplankui, kuriame yra HTML dokumentas su nuoroda į kliento programą, o antroji – aplankui, kuriame yra pats kliento programos kodas.

Toliau gali būti atidarytas baitinis įvedimo iš serverio srautas, o jis – įvilktas į buferizuotą simbolinį srautą, kaip matėme viename iš ankstesnių kliento programų pavyzdžių:

```
. . .
InputStream is      = url.openStream( );
BufferedReader in = new BufferedReader(
                            new InputStreamReader( is ) );
. . .
```

Java aplikacijoms, kurios gali jungtis su bet kuria tinklo mašina, derėtų naudoti kitą konstruktoriaus formą:

```
URL url  = new URL( String urlSpecifier )
```

Čia *urlSpecifier* – pilnas DNS adresas.

Prieš parsisiunčiant iš nutolusio kompiuterio rinkmeną, galima iš anksto nustatyti visas jo savybes klasės *URLConnection* metodais *Date getDate( ), Date getLastModified( ), String getContentType( ), int getContentLength( ),* kurių paskirtį aiškiai nusako jų pavadinimai. Šioje klasėje taip pat yra metodas *getInputStream( )* – tai alternatyva *URL* klasės metodui *openStream( ).* Klasės objektas konstruojamas ir įvedimo srautas gali būti gaunamas taip (metodas *openConnection* gali sukelti išimtį *IOException*):

```
. . .
URL url = . . .
URLConnection urlcon = url.openConnection( );
System.out,println( “Date” + new Date( urlcon.getDate( ) ) );
System.out,println( “Type of content” + urlcon.getContentType( ) );
System.out,println( “Period of storage” + urlcon.getExpiration( ) );
System.out,println( “Last modified” + new Date( urlcon.getLastModified() ) );
System.out,println( “Length, bytes” + urlcon.getContentLength( ) );
InputStream is =urlcon.getInputStream( );
. . .
```

Kitas dažnai reikalingas dalykas – parodyti iš kliento programos kitą HTML dokumentą – peradresuoti naršyklę į kitą puslapį. Tam reikia atlikti veiksmus:

1. Suformuoti norimo dokumento URL adresą *url* jau žinomais metodais.

2. Kviesti metodą *showDocument*:

getAppletContext( ).showDocument(url)

Pirmasis metodas grąžina grafinį kliento programos aplinkos kontekstą – klasės *AppletContext* objektą, o šioje klasėje yra perkrauti *void showDocument* metodai:

void showDocument(URL url)
void showDocument(URL url, String where)

Abu metodai parodo HTML rinkmeną, esančią nurodytu adresu, o antroji forma dar papildomai nusako, kurioje naršyklės lango vietoje parodyti dokumentą. *String* duomens reikšmė *_self* nurodo dokumentą rodyti dabartiniame kliento programos lange, *_blank* – naujame naršyklės lange ir t. t.

3.4. Serverinės programos

3.4.1. Bendrosios žinios apie serverines programas

Serverinės programos (*servlets*)– mažos programos, vykdomos www jungties serverio pusėje. Panašiai kaip klientinės programos dinamiškai išplečia naršyklių funkcines galimybes, serverinės programos dinamiškai išplečia www serverio funkcines galimybes. Šios programos įdiegiamos į tinklo serverį, palaikantį *Java*.

Instrumentai serverinėms programoms kurti yra pakete JSDK (*Java Servlet Development Kit*). Tarp instrumentų yra ir įrankis serverinei programai testuoti *servletrunner*. Visas instrukcijas JSDK instaliuoti kompiuteryje žr. www.javasoft.com. Šiuo metu šis nesudėtingas naudoti paketas jau įtrauktas į archyvų tarpą; vietoj jo rekomenduojama parsisiųsti tiesiog *Java* tinklo aplikacijų serverį *Application Server*, esantį pakete J2EE (*Java 2 Enterprise Edition*). Serveris nemokamas. *Windows* platformai jo apimtis truputį viršija 28 MB. Instaliavimo instrukcijos ateina kartu su paketu. Serverinių programų testavimas serveriu, naudojant *localhost* IP adresą, yra sudėtingesnis nei *servletrunner* įrankiu.

Abiejuose siuntos paketuose yra vadinamasis *Servlet API* (*Applied Programming Interface*), kuriame sudėti du paketai su visomis klasėmis, palaikančiomis visas serverinių programų galimybes: *javax.servlet* ir *javax.servlet.http*.

Kas lėmė serverinių programų technologijos sukūrimą? Tradicinė www tinklo veikimo schema informacijai gauti yra tokia.

1. www-serveris gauna informaciją iš kliento-naršyklės. Reikiamo serverio URL adresą nurodo naršyklės vartotojas, o naršyklė sugeneruoja serveriui užklausą HTML kalba. Reikiama informacija užklausoje yra sakiniuose <*form*>. Užklausa siunčiama protokolu HTTP, todėl dažnai dar vadinama HTTP užklausa.

2. Serveris analizuoja užklausą ir apdoroja gautą informaciją. Šiuos veiksmus atlieka serverio dalis CGI (*Common Gateway Interface*), galinti sąveikauti su kitomis programomis serverio kompiuteryje (duomenų bazėmis, elektroninėmis skaičiuoklėmis ir pan.). CGI gali būti realizuota bet kuria programavimo kalba, tačiau

dažniausiai tam naudojama pirmiausia tekstui apdoroti skirta kalba *Perl*. Programa *Perl* kalba yra interpretuojama, todėl dažnai CGI sąsajos dar vadinami CGI scenarijais arba tiesiog CGI *scripts*. Jei CGI programa negali reikiamai apdoroti gautos užklausos, ji savo ruožtu generuoja užklausą kompetentingai programai, pateisindama savo pavadinimą *gateway* – šliuzas.

3. Sukuriamas (ar tiesiog kopijuojamas – jei reikalingas statinis dokumentas) kliento prašytas HTML dokumentas ir persiunčiamas klientui-naršyklei. Šio HTTP atsakymo pradžioje nurodomas atsakymo turinio tipas MIME (*Multipurpose Internet Mail Extension*) formatu. Pavyzdžiui, *text/plain* reiškia ASCII simbolių tekstą, *text/html* – HTML tekstą ir t. t.

CGI programos priklauso nuo kompiuterio architektūros. Be to, jos nėra pakankamai efektyvios, kadangi, aptarnaujant kelis klientus, kiekvienam sukuriamas atskiras procesas (ne srautas!). Pakeitus CGI programas serverinėmis programomis, jų pranašumai būtų:

1. Serverinės programos vykdomosi serverio adresų erdvėje. Kiekvienai atskirai kliento užklausai apdoroti galima sukurti atskirą vykdymo srautą.

2. Serverinės programos nepriklausomos nuo kompiuterio tipo. *Sun, Netscape,* MS tinklo aplikacijų serveriai palaiko *Servlet* API, todėl programos, sukurtos *Java,* gali būti laisvai perkeliamos iš vieno serverio į kitą.

3. Serverinės programos gali naudoti turtingas *Java* klasių bibliotekas.

4. *Java* saugumo mechanizmas palaiko tam tikrus serverinių programų apribojimus, kad būtų apsaugoti serverio mašinos ištekliai.

Taigi serverinės programos yra tarpininkai tarp kliento ir serverio, perimdami daugelį funkcijų tiek iš kliento, tiek iš serverio. Pavyzdžiui, bendraudamos su klientinėmis programomis, serverinės programos gali atlikti jų *proxy*-serverio vaidmenį: jei klientinė programa turi kreiptis į duomenų bazę kitoje mašinoje (kas klientinėms programoms uždrausta), šiuos veiksmus už klientinę programą gali atlikti serverinė programa. Taip pat serverinės programos gali praturtinti serverio galimybes, palaikydamos kokį serveriui nepažįstamą užklausų ir atsakymų pagrindu veikiantį protokolą (pavyzdžiui, SMTP, POP, FTP). Dažniausiai vis tik serverinės programos naudojamos kartu su HTTP protokolu dinaminiams HTML puslapiams generuoti, todėl tam sukurtas ir specialus paketas HTTP protokolui palaikyti *javax.servlet.http*.

Kaip serverinės programos pasiekiamos klientui? Keliais būdais:

1. Kai vartotojas naršyklei teikia URL adresą, kuris yra serverinės programos adresas.

2. Iš HTML dokumento sakinio <*form*>.

3. Iš HTML dokumento sakinio <*servlet*>. Tokius sakinius pažįsta ne visi serveriai.

4. Iš HTML dokumentų specialių komandų – SSI (*Server-Side Includes*).

Serverinių programų instaliavimas yra gana sudėtingas procesas, todėl visų instrukcijų, pateikiamų J2EE 1.4 *Application Server* dokumentacijoje, čia nekartosime. Trumpai tariant reikia paleisti serverį, serverio domeną, paskui įdiegti J2EE aplikacijas iš *jar* rinkmenų tam tikruose aplankuose. Serverinės programos (ir JSP, *Java Server Pages*, apie kuriuos kalbėsime vėliau) turi būti suarchyvuotos į *jar* rinkmenų tipą *war* (*Web Application Archive*). Tokių archyvų rinkmenos turi plėtinius **.war*. Archyvavimo galimybės yra serverio programoje. Serverinės programos gali

būti laikinos arba pastovios (t. y. veiks, kol veiks pats serveris) – tai priklausys nuo jų įdiegties serveryje.

3.4.2. Serverinės programos gyvavimo ciklas

Serverinė programa turi tris metodus: *init, service* ir *destroy*. Metodus iškviečia serveris. Tipiško serverinės programos gyvavimo ciklo pavyzdys. Tinklo serveris, gavęs iš kliento-naršyklės HTTP užklausą URL adresui, kuriuo yra serverinė programa, įkelia į savo adresinę erdvę nurodytąją serverinę programą.

1. Kai serverinė programa įkelta į atmintį, kviečiamas pradinis metodas *init*. Pradinių reikšmių parametrus galima perduoti serverine programai.
2. Kviečiamas metodas *service*, apdorojantis HTTP užklausą: nuskaito perduotus duomenis, formuoja HTTP atsakymą.
3. Serverinė programa lieka serveryje ir gali apdoroti kitas klientų užklausas. Kiekvienai užklausai kaskart iškviečiamas metodas *service*.
4. Tam tikrais specifiniais kiekvienam serveriui algoritmais nustačius, kad serverinė programa nebereikalinga, serverinė programa iškeliama iš serverio atminties – kviečiamas metodas *destroy*. *Java* šiukšlių rinktuvas grąžina serveriui serverinės programos užimtą atmintį.

Paprasčiausios serverinės programos pavyzdys. Serverinė programa tik nusiunčia naršyklei pranešimą paryškintu šriftu *Simplest servlet*.

```
//The simplest servlet
//
import java.io.*;
import javax.servlet.*;

public class SimplestServlet extends GenericServlet{
        public void service( ServletRequest request, ServletResponce responce )
                                                throws Serverinėje programojexception,
IOException{
                responce.setContentType( “text/html” );
                PrintWriter pw = responce.getWriter( );
                pw.println( “<B> Simplest servlet” );
                pw.close( );
        }
}
```

Šioje programoje *javax.servlet* – paketas, kartu su specializuotu paketu HTTP protokolui sudarantis visą *Servlet* API, o *GenericServlet* – jo klasė. *ServletRequest* yra sąsajų klasė kliento HTTP užklausos duomenims skaityti, o *ServletResponce* – sąsajų klasė HTTP atsakymui kurti. Metodas *setContentType* nurodo HTTP atsakymo turinio tipą MIME formatu. Sąsajų klasės *ServletResponce* metodas *getWriter* grąžina *PrintWriter* objektą. Metodas gali sukelti išimtį *IOException*. *PrintWriter* klasės metodas *println* išveda į HTTP atsakymą – sakinį paryškintu šriftu; HTML dokumento titulinė dalis (sakiniai <*HEAD*> ir <*TITLE*>) nebūtina, todėl čia praleista.

Jei serverinė programa bus testuojama įrankiu *servletrunner*, teks atlikti šiuos veiksmus:

1. Serverinę programą sukompiliuoti ir įdėti į aplanką C:\JSDK2.0\examples (tik tada *servletrunner* galės aptikti serverinę programą).
2. Paleisti *servletrunner*.
3. Paleisti naršyklę ir nurodyti jai URL adresą `http://localhost:8080/servlet/SimplestServlet`. Čia *localhost* – mūsų kompiuterio adresas, o *8080* – prievadas, kurio „klauso“ *servletrunner*.

Naršyklė turi parodyti pranešimą ***Simplest Servlet***.

3.4.3. *Servlet* API sąsajų klasės, klasės ir metodai

Kaip minėta, API sudaro du paketai *javax.servlet* ir *javax.servlet.http*. API palaiko *Sun*, MS, *Netscape* firmų serveriai.

Pakete *javax.servlet* yra 6 sąsajų klasės:

Servlet – skelbia minėtuosius 3 serverinės programos gyvavimo ciklo metodus.

ServletConfig – leidžia serverinėms programoms gauti pradinių reikšmių parametrus.

ServletContext – leidžia serverinėms programoms gauti informaciją apie jų vykdymo aplinką bei užtikrina įvykių sistemą.

ServletRequest – skelbia metodus kliento užklausos duomenims skaityti.

ServletResponce – skelbia metodus atsakymo klientui duomenims užrašyti.

SingleThreadModel – uždraudžia daugiasrautiškumo galimybes serverinei programai.

Kai kurios paketo klasės:

GenericServlet – realizuoja sąsajų klases *Servlet* ir *ServletConfig*.

ServletException, UnavailableException – išimčių klasės; antrosios objektas sukuriamas, kai serverinė programa yra neprieinama HTTP užklausai – neįkelta į serverį.

Dabar apie sąsajų klasių ir klasių pagrindinius metodus. Sąsajų klasėje *Servlet* yra trys serverinės programos gyvavimo ciklo ir keli kiti metodai:

void init(ServletConfig sc) throws ServletException – *sc* yra serverinės programos inicializavimo parametrai. Kaip pateikiami šie parametrai, priklauso nuo

serverio. Įrankiui *servletrunner* jie turi būti rinkmenoje *servlet.properties*, o ši – aplanke C:\JSDK2.0\examples. Rinkmenos turiniui keliami tam tikri reikalavimai, kurių čia nenagrinėsim.

void service(ServletRequest req, ServletResponce res) throws ServletException, IOException – apdoroja užklausą *req* ir formuoja atsakymą *res*.

void destroy( ) – iškelia serverinę programą iš serverio atminties ir išlaisvina serverinei programai skirtus išteklius.

ServletConfig getServletConfig( ) – grąžina pradinių reikšmių parametrus.

String getServletInfo( ) – grąžina eilutę, aprašančią serverinę programą. Galima perrašyti metodą, kad jis grąžintų reikiamą informaciją.

Sąsajų klasės *ServletConfig* metodai yra:

ServletContext getServletContext( ) – grąžina serverinės programos kontekstą – savo ruožtu sąsajų klasės *ServletContext* tipo; šioje yra 8 metodai, grąžinantys visą informaciją apie serverinės programos aplinką.

String getInitParameter(String param) – grąžina parametro *param* pradinę reikšmę.

Enumeration getInitParameterNames( ) – grąžina visus pradinių parametrų vardus.

Sąsajų klasę *ServletRequest* realizuoja serveris. Sąsajų klasės metodai leidžia gauti informaciją apie kliento užklausą. Kai kurie iš tų metodų yra:

int getContentLength( ) – grąžina užklausos ilgį; jei negalima nustatyti ilgio, grąžinama reikšmė *–1*.

String getContentType( ) – grąžina užklausos tipą MIME formatu; jei tipo nustatyti negalima – grąžinama reikšmė *null*.

Enumeration getParameterNames( ) – grąžina visus užklausos parametrų vardus.

String getParameterValues( ) – grąžina visas užklausos parametrų reikšmes.

BufferedReader getReader( ) throws IOException – grąžina buferizuotą simbolinį srautą realizuojantį objektą užklausai skaityti.

String getRemoteAddr( ) – grąžina klieto IP adresą.

String getServerName( ) – grąžina serverio vardą.

int getServerPort( ) – grąžina prievado numerį.

Analogiška sąsajų klasė HTTP atsakymams generuoti yra *ServletResponce*. Kai kurie jo metodai:

PrintWriter getWriter( ) throws IOException – grąžina buferizuotą simbolinį srautą realizuojantį objektą atsakymui rašyti. Objektas turi metodus *println()* ir *print()*.

void setContentLength(int size) – nustato atsakymo turinio ilgį.

void setContentType(String type) – nustato atsakymo tipą MIME formatu.

Nesudėtingos serverinės programos pavyzdys. Programa tik nuskaito HTML puslapyje, kur yra šios serverinės programos URL adresas, teikiamus parametrus ir persiunčia juos atgal naršyklei (P. Naughtonas). Tegu tie parametrai yra darbuotojo pavardė (HTML puslapyje bus lentelė, o šioje – laukelis su užrašu *Employee*) ir jo telefono numeris (laukelis su užrašu *Phone*).

Pirmiausia aptarsime HTML dokumentą, kuris kreipsis į serverinę programą. Tegu jame bus lentelė su dviem minėtaisiais laukais bei mygtukas duomenų įvedimui patvirtinti. Dokumentas tegu bus saugomas vardu *PostParameters.htm*:

```
<HTML>
<BODY>
<CENTER>
<FORM       NAME = "FORM"
            METHOD = POST
            ACTION   =
            "http://localhost:8080/servlet/PostParametersServlet">
    <TABLE>
        <TR>
            <TD><B> Employee</B></TD>
            <TD><INPUT  TYPE = TEXTBOX   NAME ="e"
                SIZE = "25"   VALUE = " "></TD>
        </TR>
        <TR>
            <TD><B> Phone </TD>
            <TD><INPUT  TYPE =TEXTBOX   NAME =  "p"
                SIZE = "25"   VALUE = " "></TD>
        </TR>
    </TABLE>
    <INPUT  TYPE = SUBMIT  VALUE = "Submit">
</FORM>
</BODY>
</HTML>
```

Dokumente sakinius rašėme didžiosiomis raidėmis, nors tai neturi jokios įtakos. Kai kurios parametrų reikšmės sakiniuose-konteineriuose rašomos kabutėse – jos nebūtinos. Kabučių nerašėme tais atvejais, kai parametrams priskyrėme standartines reikšmes, pavyzdžiui, *TEXTBOX* ar *SUBMIT*.

Visi dokumento duomenys, skirti persiųsti į serverį, dedami į sakinį *<FORM>*. Vienintelis būtinas sakinio parametras yra *ACTION* – URL adresas programos, kuriai bus atiduodami dokumento duomenys. Tai gali būti serverinė

programa (kaip šiuo atveju) arba kokia nors CGI programa. Parametras *METHOD* nustato duomenų persiuntimo metodą. Metodų yra trys: *GET, POST* ir *HEAD. GET*, kaip sako vien jo vardas, pirmiausia skirtas informacijai iš serverio pareikalauti (pavyzdžiui, kokios nors rinkmenos, serverinės programos ar kitos programos išvesties srauto, serverio kokio nors įrenginio išvesties srauto). *GET* duomenys siunčiami kartu su užklausa maždaug tokiu formatu poromis: *parametro vardas = reikšmė*. Metodas turi griežtus apribojimus: dauguma serverių neleidžia persiųsti juo daugiau kaip kelis šimtus baitų. Į *GET* panašus ir *HEAD* metodas: juo serveris persiunčia tik titulinę informaciją. *POST* metodas pirmiausia skirtas kliento duomenims siųsti į serverį. Duomenys siunčiami kaip įvesties srautas, todėl nėra duomenų ilgio apribojimų.

<TABLE> – lentelės sakinys, *<TR>* – jos atskiros eilutės sakinys, o *<TD>* – eilutės atskiros ląstelės sakinys. Pirmojoje dokumento ląstelėje paryškintu šriftu (tą nustato *<B>*) bus spausdinamas užrašas *Employee*.

Sakinys-konteineris *<INPUT>* turi du būtinus parametrus: *TYPE* nurodo lauko tipą (*TEXTBOX*), o *NAME* – to lauko vardą; naršyklė vardo nerodo. Parametras *SIZE* nustato lauko ilgį, o *VALUE* suteikia laukui reikšmę. Kadangi šis sakinys įdėtas į sakinį *<TD>* – jis nustato lentelės eilutės ląstelės pavidalą. Žemiau tokiame sakinyje parametrui *INPUT* teikiama reikšmė *SUBMIT* – tai sukuria duomenų perdavimo mygtuką, kurį nuspaudus duomenys persiunčiami serveriui. *VALUE* šiuo atveju nustato užrašą ant mygtuko. Panašiai *TYPE = RESET* sukuria duomenų atmetimo mygtuką.

Serverinė programa duomenims iš parašyto HTML dokumento įvesti ir juos persiųsti atgal naršyklei:

```
//Servlet for reading /sending parameters
//from/to client, ie, HTML page
//A HTML file PostParameters must be provided
//from client side

import java io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet extends GenericServlet{
        public void service( ServletRequest request, ServletResponce responce )
                                        throws ServletException, IOException{
                responce.setContentType( "text/html" );
                PrintWriter pw = responce.getWriter( );
                Enumeration e = request.getParameterNames( );
                pw.println( "<H2> Your data </H2>" );
                while( e.hasMoreElements( ) ) {
                        String name = ( String )e.nextElement( );
                        String value = request.getParameter( name );
                        pw.println( name + "= " + value );
                }
                pw.close( );
        }
}
```

Darbas su šia serverine programa:

1. Paleisti *servletrunner*.
2. Parodyti naršyklėje dokumentą *PostParameters.htm*, įvesti formoje prašomus duomenis ir paspausti mygtuką *Submit*.
3. Naršyklė turi parodyti serverinės programos *PostParametersServlet* atsaką: užrašą *Your data* pavadinimo sakinio *<H2>* formatu (pavadinimų sakinių yra nuo *H1* iki *H6*; didžiausias jų – *H1*) ir toliau įvestų duomenų vardus bei reikšmes.

Dirbant su HTTP užklausomis, dažniau naudojama specializuota klasė *HttpServlet*, įeinanti į paketą *javax.servlet.http,* o ne *GenericServlet*. Šiame pakete yra sąsajų klasės *HttpServletRequest, HttpServletResponce, HttpSession* ir kt. Pirmosios dvi sąsajų klasės atitinka nagrinėto paketo *javax.servlet* analogiškas sąsajų klases, o trečioji skelbia metodus, leidžiančius sekti visą kliento – serverinės programos bendrojo darbo sesiją (ar kitaip – seansą). Pagrindinės paketo klasės yra *HttpServlet,* realizuojanti metodus darbui su HTTP užklausomis ir atsakymais, bei *Cookie,* turinti metodus, leidžiančius išsaugoti kliento mašinoje seanso būklės informaciją. *Cookie* saugo visus HTML dokumentų titulinio skyriaus duomenis.

Sąsajų klasė *HttpServletRequest* skelbia 18 metodų, tarp jų:

Cookie [ ] getCookies( ) – grąžina *Cookie* duomenų masyvą.

Enumeration getHeaderNames( ) – grąžina visus dokumento titulinio skyriaus vardus.

String getRemoteUser( ) – grąžina nutolusio kliento vardą.

HttpSession getSession(boolean new) – jei *new* yra *true* – sukuria ir grąžina šios užklausos sesijos objektą; jei *false* – grąžina jau egzistuojančios sesijos objektą.

Sąsajų klasė *HttpServletResponce* turi 12 metodų, tarp jų:

void addCookie(Cookie c) – prie HTTP atsakymo prijungia objektą *c*.

void sendError(int i, String s) throws IOException – įvykus klaidai, klientui siunčia klaidos kodą *i* ir pranešimą *s*.

void sendRedirect(String url) throws IOException – perjungia klientą adresu *url*.

Sąsajų klasėje *HttpSession* yra 10 metodų, leidžiančių gauti informaciją apie HTTP seansą arba įrašyti informaciją apie su seansu susijusią kliento būklę. Seanso sąvoka reikalinga, pavyzdžiui, elektroninėje parduotuvėje. HTTP protokolas seansų nepalaiko, todėl būtinos tokios programavimo priemonės, kurios leistų išsaugoti visą informaciją apie kliento visas užklausas parduotuvei ir visus parduotuvės atsakymus klientui. Objektai *Session* egzistuoja tam tikrą laiką priklausomai nuo serverinės programos serverio (apie 30 minučių). Visi sąsajų klasės metodai gali sukelti išimtį

IllegalStateException – jei metodo kvietimo metu seansas jau pasibaigęs. Sąsajų klasėje yra metodai:

String getID( ) – grąžina seanso identifikatorių.

long getCreationTime( ) – grąžina *Session* objekto sukūrimo laiką pradedant nuo 1970-01-01 GMT (*Greenwich Mean Time*), milisekundėmis.

long getLastAccessedTime( ) – grąžina paskutinės kliento užklausos laiką tuo pat formatu, kaip metodas *getCreationTime*.

void putValue(String name, Object obj) – seanso objektą *obj* susieja su vardu *name*.

Object getValue(String name) – grąžina seanso objekto, turinčio vardą *name*, reikšmę.

void invalidate( ) – užbaigia seansą.

Klasė *Cookie*. Vadinamieji *Cookie* duomenys ar kitaip – elementai leidžia išsaugoti kokius nors duomenis apie klientą jo kompiuteryje. Pavyzdžiui, elektroninės parduotuvės atveju elementas gali saugoti kliento vardą, adresą ir pan. Klientui nereikės kelis kartus įvedinėti šių duomenų, keletą kartų užsakant prekes. Taigi serverinė programa gali į kliento kompiuterį metodu *addCookie* įrašyti *Cookie* duomenis: duomens vardą ir reikšmę, saugojimo laiką, *Cookie* elemento domeninį adresą ir kelią iki jo. Adresas ir kelias iki elemento reikalingi tam, kad būtų galima nustatyti, ar įtraukti *Cookie* elementą į HTTP užklausą ir siųsti serveriui. Jei vartotojo naršyklės nurodytas URL adresas sutampa su *Cookie* elemento adresu – elementas prie užklausos prijungiamas. Jei *Cookie* duomenų saugojimo kliento mašinoje laikas nenurodytas, tai elementas pašalinamas, pasibaigus dabartiniam naršyklės seansui. Klasė turi vienintelį konstruktorių *Cookie(String name, String value),* nustatantį elemento vardą ir reikšmę.

Klasėje tarp kitų yra metodai:

String getName( ) – grąžina *Cookie* elemento vardą.

String getValue( ) – grąžina elemento reikšmę.

String getDomain( ) – grąžina elemento domeninį adresą.

String getPath( ) – grąžina kelią iki elemento.

int getMaxAge( ) – grąžina maksimalų elemento saugojimo laiką sekundėmis.

Klasėje taip pat yra analogiškų metodų reikšmių prieskyrai *set*.

Klasė *HttpServlet* plečia klasę *GenericServlet*. Klasė turi metodus, realizuojančius įvairias HTTP užklausas: užklausai *GET* – metodą *doGet, POST* – *doPost, PUT* – *doPut* ir pan. Bendriausias klasės metodas yra metodui *service* iš klasės *GenericServlet* atitinkantis

void service(HttpServletRequest request, HttpServletResponce responce) throws IOException, ServletException

Darbo su Cookie klase pavyzdys. Iliustruojamas darbas su *Cookie* elementais (P. Naughtonas). Yra HTML dokumentas, saugomas rinkmenoje vardu *AddCookie.htm,* kurio įvedimo laukelyje vardu *data* įvedama elemento reikšmė. Dokumente taip pat yra mygtukas su užrašu *Submit* duomenų persiuntimui į serverį patvirtinti. Duomenys siunčiami metodu *POST* adresu, kuris yra serverinės programos *AddCookieServlet* adresas – taigi bus suaktyvinta ši serverinė programa. Čia, kaip ir ankstesniuose pavyzdžiuose, naudojamasi *localhost* adresu, leidžiančiu imituoti serverio darbą savu kompiuteriu.

```
<HTML>
<BODY>
<CENTER>
<FORM          NAME = “FORM2”
               METHOD = POST
               ACTION  =”http://localhost:8080/servlet/AddCookieServlet”>
     <B> Enter a value for MyCookie: </B>
     <INPUT   TYPE = TEXTBOX     NAME =”data”
                         SIZE = “25”    VALUE = “ “>
     <INPUT   TYPE = SUBMIT    VALUE = “Submit “>
</FORM>
</BODY>
</HTML>
```

Serverinė programa *AddCookieServlet* metodu *getParameter* (metodas priklauso sąsajų klasei *ServletRequest*, tačiau yra prieinamas klasei *HttpServlet*, nes ji plečia klasę *GenericServlet*, o ši realizuoja minėtąją sąsajų klasę) gauna parametro *data* reikšmę, sukuria *Cookie* elementą, kuriam suteikiamas vardas *MyCookie*, o su šiuo vardu susiejama parametro *data* reikšmė. Elemento gyvavimo laikas nenurodomas, taigi elementas bus saugomas tol, kol vyks naršyklės darbo seansas. Vėliau prie serverinės programos HTTP atsakymo metodu *addCookie* prijungiamas šis sukurtasis *Cookie* elementas ir siunčiamas atgal klientui su paaiškinamuoju tekstu *MyCookie has been set to:*

```
import java io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet{
     public void doPost( HttpServletRequest request,
                              HttpServletResponce responce )
                              throws     Serverinėje     programojexception,
IOException{
          String data = request.getParameter( “data” );
          Cookie cookie = new Cookie( “MyCookie”, data );
          response.addCookie( cookie );
          responce.setContentType( “text/html” );
```

```
                PrintWriter pw = responce.getWriter( );
                pw.println( "<B> MyCookie has been set to:"  );
                pw.println( data );
                pw.close( );
        }
}
```

Antroji serverinė programa parodys, kad kliento kompiuteris saugo *Cookie* elementą su ką tik suteiktomis pradinėmis reikšmėmis. *Cookie* elementų masyvas *cookies* gaunamas taip, kaip to reikalauja metodas *getCookies*, kiekvienam masyvo elementui išskiriamos *Cookie* elemento dalys: vardas metodu *getName* bei reikšmė metodu *getValue*, ir siunčiamas klientui.

```
import java io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet{
        public void doGet( HttpServletRequest request,
                                        HttpServletResponce responce )
                                        throws Serverinėje programojexception,
IOException{
                Cookie [ ] cookies  = request.getCookies( );
                responce.setContentType( "text/html" );
                PrintWriter pw = responce.getWriter( );
                for( i = 0; i < cookies.length; i++ ) {
                        String name = cookies[ i ].getName( );
                        String valuee = cookies[ i ].getValue( );
                        pw.println( "name = " + name + " value = " + value  );
                }
                pw.close( );
        }
}
```

Šias serverines programas paleisti taip:

1. Paleisti įrankį *servletrunner*.
2. Naršyklėje parodyti dokumentą *AddCookie.htm*. Įvesti formoje prašomą duomenį ir paspausti mygtuką *Submit*.
   Dabar serverinė programa, kurią suaktyvins dokumentas *AddCookie.htm*, atgal parsiųs įvestą duomens reikšmę su paaiškinamuoju tekstu.
3. Naršyklėje įvesti kitą URL adresą:
   http://localhost:8080/servlet/GetCookiesServlet.

Dabar bus suaktyvintas antroji serverinė programa, kuri naršyklei perduos *Cookie* elemento vardą ir reikšmę.

Kad įsitikintumėte, kiek laiko duomenų elementai saugomi kompiuteryje, paeksperimentuokite su šiomis programomis: pirmojoje serverinėje programoje galima metodu *setMaxAge* nustatyti elemento saugojimo laiką ir pažiūrėti, kiek laiko tie duomenys bus prieinami antrajai serverinei programai.

3.4.4. JSP technologija

JSP (*Java Server Pages*) tikslas – supaprastinti dinaminių www puslapių kūrimą. Technologija paremta serverinių programų klasėmis ir yra J2EE SDK standartinė dalis. JSP technologiją palaiko dauguma tinklo serverių, palaikančių pačią *Java*: JSP puslapiai kompiliuojami į serverinių programų klases.

Kuriant www puslapį, technologija leidžia kartu kombinuoti HTML kodą ir *Java* kodo fragmentus. Šie fragmentai įjungiami į specialius sakinius. JSP konteineriai atpažįsta šiuos sakinius ir naudoja juose įdėtą kodą serverinei programai ar jo daliai generuoti, arba šis kodas gali kviesti *JavaBeans* komponentus. Taigi JSP yra paprastesnis puslapių kūrimo įrankis nei serverinės programos: pakanka vieno vienintelio dokumento dinaminiam puslapiui apibrėžti.

JSP puslapio rinkmenos turi plėtinius *.jsp* arba *.jspx*. Plėtiniai nurodo tinklo serveriui, kad rinkmenos tam tikras dalis turi kompiliuoti JSP kompiliatorius. Kompiliatorius interpretuoja JSP sakinius ir generuoja atitinkamą kodą – serverinės programos kodą. Tokios puslapių rinkmenos yra sukompiliuojamos pirmąkart suaktyvinus puslapį, vėliau sukompiliuotas tekstas saugomas serverio adresinėje erdvėje. Todėl kitos to paties puslapio užklausos atliekamos greitai, o pats puslapis faktiškai yra statinis HTML puslapis. Pakeitus JSP rinkmenos sakinius, rinkmena automatiškai perkompiliuojama ir perkraunama į serverio atmintį.

Paprasčiausios JSP rinkmenos pavyzdys: www puslapio, rodančio esamą laiką sekundėmis, kūrimas. Rinkmenos tekstas:

```
<HTML>
<BODY>
<H1> The time is:
        <% =  System.currentTimeMillis( ) / 1000  %>
</H1>
</BODY>
</HTML>
```

Kaip matyti, JSP sakinys pradedamas ir baigiamas procento ženklu, o lygybės ženklas nustato atskirą sakinio tipą – išraiškos sakinį. Klientui pirmąkart pareikalavus URL adreso – šios rinkmenos adreso, tinklo serveris persiųstų užklausą JSP konteineriui, šis sukompiliuotų JSP puslapį į serverinę programą. Šioje serverinėje programoje būtų kodas, naudojantis klases *HttpServletResponse, PrintWriter* ir *String* (rezultatui – laikui saugoti) ir generuojantis HTTP atsakymą klientui su esamu laiku sekundėmis; atsakymas būtų parodytas formatu *H1*.

Trumpai apie JSP sintaksę. Yra JSP sakiniai-direktyvos, skirti tik JSP konteineriui ir nieko nerašantys į išvesties srautą *PrintWriter,* bei sakiniai-scenarijų elementai. Direktyvos pavyzdys:

```
<% page language = “java”    import = “java.util.*” %>
```

Direktyva JSP konteineriui praneša, kad toliau naudojama *Java* kalba, ir prašo dinamiškai įkelti nurodyto paketo klases. Kitos direktyvos: *extends, buffer, autoFlush, info, isThreadSafe* ir t.t.

Sakinių-scenarijų elementų yra trys tipai: skelbimai, vadinamieji *scriplets* ir išraiškos. Skelbimai aprašo scenarijų elementus, antrieji yra kodo fragmentai, o išraiškos pavyzdį jau matėme ankstesniame pavyzdyje. Jų sakiniai atitinkamai yra:

<%! %>, *<% %>*, *<%= %>*. Sakinys *<%-- --%>* yra JSP komentaras.

JSP kuriamo www puslapio pavyzdys: puslapyje bus parodyta puslapio įkėlimo data, šios dienos data, kiek laiko puslapis rodomas ir kiek kartų puslapis aplankytas. Paskui į puslapį išvedamas tekstas *Good bye*. JSP rinkmenos tekstas:

```
<%@  page language = "java"  import = "java.util.*"  %>
<%!
        long loadTime = System.currentTimeMillis( );
        Date loadDate = new Date( );
        int hitCount = 0;
%>
<HTML>
<BODY>
<H1> This page is loaded on  <%= loadDate%> </H1>
<H1> Today's date is  <%=  new Date( ) %> </H1>
<H1>  This page is working <%= (System.currentTimeMillis( ) – loadTime )/1000
%>
        seconds </H1>
<H1> This page was visited <%= ++hitCount %> times since
        <%= loadDate %> </H1>
<%--   This scriplet prints message "Good bye" to the www page.
        The PrintWriter object is always named "out" by default.
--%>
<%
        out.println( "Good bye" );
%>
</BODY>
</HTML>
```

3.5. RMI technologijos pagrindai

RMI (*Remote Method Invocation* – nutolusių metodų kvietimas) technologija leidžia objektui, gyvuojančiam viename kompiuteryje, kviesti metodus objekto, esančio kitame tinklo kompiuteryje. Visus technologinius reikalavimus atitinkantis nutolęs metodas traktuojamas lygiai taip pat, kaip lokalusis šiame kompiuteryje esantis metodas.

Technologiją realizuoja paketai *java.rmi, java.rmi.registry, java.rmi.server*.

Technologijai realizuoti teks sudaryti keturias rinkmenas. Pirmiausia turi būti parašyta nutolusi sąsajų klasė, skelbianti reikiamus metodus. Šią sąsajų klasę realizuoja serveris. Ši sąsajų klasė turi būti paskelbta *public*, turi plėsti sąsajų klasę *java.rmi.Remote*, o kiekvienas jos metodas gali sukelti išimtį *java.rmi.RemoteException*. Antra rinkmena – serverio programa, realizuojanti nutolusią sąsają. Ši programa dar privalo plėsti klasę *UnicastRemoteObject* – ši superklasė palaiko objektų, esančių nutolusiame kompiuteryje, funkcionalumą. Programoje būtina perrašyti klasės konstruktorių, kad šis galėtų sukelti išimtį *RemoteException*. Konstruktorius gali būti „tuščias": superklasės konstruktorius

kviečiamas automatiškai. Trečioji rinkmena – taip pat serverio programos, turinčios startinį *main* metodą. Ši programa taip pat privalo užregistruoti sukurtąjį nutolusį objektą vadinamajame RMI rejestre serveryje metodu *Naming.rebind*. Ketvirtoji rinkmena – programa kliento kompiuteryje. Joje reikia gauti iš serverio rodyklę į nutolusią sąsajų klasę tos pačios klasės *Naming* metodu *lookup*, o toliau jau su šiuo objektu elgiamasi įprastai, lyg su esančiu lokalioje mašinoje. Be šių programų kompiliavimo ir paleidimo, dar teks pasitelkti įrankį RMI rejestrui paleisti serveryje bei įrankį programoms-kamščiams (*stubs*), turinčioms lygiai tokias pačias klases, kaip nutolusios sąsajų klasės, ir esančioms kliento kompiuteryje, taip pat programoms-karkasams (*skeletons*), esančioms serverio kompiuteryje, generuoti. Programos-kamščiai ir programos-karkasai rūpinasi visais ryšiais tarp kliento ir serverio; programuotojui tuo užsiimti nereikia. Jei reikia persiųsti objektus, šie siunčiami serializuoti, o paskirties vietoje automatiškai atkuriami.

Visus šiuos reikalavimus iliustruosime paprasčiausiu uždaviniu: tarkime, norime sudėti du *double* formato skaičius. Parašysime sudėties metodą *add*, o šis metodas bus vykdomas nutolusiame kompiuteryje. Duomenis – abu sudedamus skaičius, taip pat serverio URL adresą, įvesime klientinėje programoje per komandinės eilutės argumentus.

1. Nutolusi sąsajų klasė, atitinkanti minėtuosius reikalavimus:

```
import java.rmi.*;
public interface ServerInterface extends Remote{
        double add( double d1, double d2) throws RemoteException;
}
```

2. Serverio programa, realizuojanti nutolusią sąsajų klasę:

```
import java.rmi.*;
import java.rmi.server.*;

public class ServerImplementation extends UnicastRemoteObject
                                        implements ServerInterface{

        public ServerImplementation( ) throws RemoteException{ }

        public double add( double d1, double d2 ) throws RemoteException{
                return d1 + d2;
        }
}
```

3. Startinė serverio programa, sukurianti bent vieną parašytąjį metodą *add* turinčios klasės objektą (t. y. *ServerImplementation* objektą) ir registruojanti šį nutolusį objektą RMI rejestre paketo *java.rmi* klasės *Naming* statiniu metodu *rebind*. Metodo pirmasis argumentas – startinės programos vardas, teikiamas kaip *String* duomuo, o antrasis – sukurtasis *ServerImplementation* klasės objektas:

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
```

```
public class ServerProgram{

        public static void main( String args[ ] ){
                try{
                    ServerImplementation si = new ServerImplementation( );
                    Naming.rebind( “ServerProgram”, si );
                } catch( Exception e ){
                        System.out.println( “Exception: “ + e );
                }
        }
}
```

4. Klientinė programa. Esminis dalykas šioje programoje – gauti rodyklę į nutolusios sąsajų klasės (ne klasės!) tipo objektą, esantį serveryje. Tai atlieka klasės *Naming* statinis metodas *lookup*, kuriam teikiamas vienas argumentas – serverio programos URL adresas su specialiu protokolu *rmi*. Šis adresas suformuojamas kaip *String* tipo duomuo iš pirmojo komandinės eilutės argumento. Turint rodyklę į nutolusį objektą, jam jau galima kviesti metodą *add*.

```
import java.rmi.*;
import java.rmi.registry.*;

public class ClientProgram{

        public static void main( String args [ ] ){
                try{
                    String serverURL = “rmi://”+args[0]+ “/ServerProgram”;
                    ServerInterface si = ( ServerInterface ) Naming.lookup(
                                                            serverURL );
                    System.out.println( “Numbers to be added “ + args[1] +
                                                        “   “ +  args[2] );
                   double d1 = Double.valueOf( args[1] ).doubleValue( );
                   double d2 = Double.valueOf( args[2] ).doubleValue( );
                   System.out.println( “Result “ + si.add( d1,d2 ) );
                } catch( Exception e ) {
                        System.out.println( “Exception “ + e );
                }
        }
}
```

Visas šis programų paketas paleidžiamas taip:

1. Visos programų rinkmenos sukompiliuojamos.
2. Suformuojamos programos-kamščiai ir programos-karkasai įrankiu *rmic*:

```
rmic ServerImplementation
```

Įrankio darbo rezultatai – rinkmenos *ServerImplementation_Stub.class* (programai-kamščiui) ir *ServerImplementation_Skel.class* (karkasui).

3. Aplanke kliento kompiuteryje sudedamos rinkmenos *ClientProgram.class, ServerInterface.class, ServerImplementation_Stub.class*.

Aplanke serverio kompiuteryje turi būti rinkmenos *ServerProgram.class, ServerInterface.class, ServerImplementation.class, ServerImplementation_Stub.class, ServerImplementation_Skel.class*.

Taigi dalis rinkmenų turi būti ir serveryje, ir kliento kompiuteryje.

4. Serveryje paleidžiamas RMI rejestro įrankis *rmiregistry*:

start rmiregistry

5. Paleidžiamas serveris:

java ServerProgram

6. Paleidžiamas klientas. Jei norime patikrinti visų programų veikimą be serverio, t. y. kad tas pats kompiuteris atliktų ir serverio, ir kliento vaidmenis, vietoje serverio URL adreso komandinėje eilutėje nurodome specialų IP adresą 127.0.0.1 (žr. skyrių apie bazinį tinklinį programavimą). Be to, būtini dar du komandinės eilutės argumentai: antrasis bus pirmas sudedamas skaičius, o trečiasis – antras:

java ClientProgram 127.0.0.1 1 2

Programų ūkio spausdinimai bus tokie:

Numbers to be added 1 2
Result 3

3.6. Bendrosios žinios apie CORBA technologiją

Jei RMI biblioteka realizuoja sąveiką tarp nutolusių *Java* objektų, tai CORBA tą patį leidžia padaryti tarp bet kokiomis programavimo kalbomis parašytų objektų. CORBA – tai integracinė technologija, kurios specifikaciją parengė OMG komitetas (*Object Management Group*, www.omg.org).

Specifikacija susideda iš dviejų dalių: pagrindinio objektų modelio (*Core Object Model*) ir objektų valdymo architektūros (*Object Management Architekture*, OMA). OMA nustato būtinas objekto charakteristikas, sąsajos, operacijos savybes ir pan. – nustato mechanizmų, leidžiančių objektams sąveikauti, infrastruktūrą. Joje taip pat apibrėžtos objektams teikiamos paslaugos (*Object Services*) ir vadinamasis užklausų objektams tarpininkas (*Object Request Broker*, ORB).

Java CORBA galimybės realizuotos klasių paketuose CORBA (faktiškai jame yra klasė ORB), *org.omg. CORBA, org.omg. CosNaming, org.omg.CosNaming.NamingContextPackage* ir keliuose kituose CORBA paketuose ir subpaketuose.

Klientinėje šios technologijos programoje metodai kviečiami įprastu būdu, tačiau pats kvietimo mechanizmas sudėtingas: reikia pasitelkti ORB, kad būtų susijungta su serveriu, paskui metodų argumentai supakuojami į dvejetainius duomenis ir persiunčiami žemojo lygio protokolu; serveryje duomenys dekoduojami į reikiamą formatą ir kviečiami reikiamo proceso reikiami metodai. Visos šios paslaugos apibrėžtos OMG komiteto nustatytose sąsajose.

Kadangi įmanoma skirtingomis kalbomis parašytų objektų sąsaja, būtina dar viena standartizuota kalba minėtosioms sąsajoms pateikti. Tai sąsajų apibrėžimo kalba IDL (*Interface Definition Language*). Kalba nustato, kaip turi būti aprašyti duomenų tipai, operacijos, argumentai, sąsajų klasės, metodai. Kalbos sintaksė panaši į *Java* sintaksę, tik vietoje *Java* paketo sąvokos IDL kalboje naudojama modulio (*module*) sąvoka.

IDL kalba parašyta programa kompiliuojama kompiliatoriumi IDL/*Java* į *Java* kodą. Kompiliatorius taip pat sugeneruoja programas-kamščius ir programas-karkasus (žr. RMI technologiją).

Kad susidarytų įspūdis apie technologiją, nesigilindami į detales, pateiksime vieną pavyzdį (B. Eckelis): kuriamas nutolęs serveris, teikiantis klientams tikslų laiką. Abi programos dalys čia parašytos *Java*, nors galima būtų naudoti ir skirtingas kalbas. Pavyzdžiui realizuoti pasitelkti J2EE instaliaciniame pakete esantys instrumentai *Java*DL, ORB, rodyklių-objektų identifikavimo tarnyba.

Pirmiausia IDL kalba reikia parašyti būtinas sąsajas:

```
module remotetime{
        interface ExactTime{
                string getTime( );
        };
};
```

Rinkmena išsaugoma su priesaga *.idl* ir kompiliuojama IDL/*Java* kompiliatoriumi (pakete – *idltojava.exe* programa), gaunant programas-kamščius ir programas-karkasus:

```
idltojava remotetime.idl
```

Suformuojamos minėtosios programos *_ExactTimeStub.java* ir *_ExactTimeImplBase.java*, bei sąsajų klasė *Java* kalba *ExactTime.java*.

Dabar į vieną programą įdedamas serveris *ExactTimeServer* ir jo objektą realizuojanti klasė *RemoteTimeServer*. Ši dar turi tą objektą, panašiai kaip RMI technologijoje, užregistruoti, tik čia – su ORB. Tarpininko ORB metodų čia neaiškinsime, tai galima rasti *Javasoft* ir OMG internetiniuose puslapiuose.

Programa:

```
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;
```

```
// Server
class ExactTimeServer extends _ExactTimeImplBase{
        public String getTime( ){
                return DateFormat. getTimeInstance( DateFormat.FULL ).
                        format( new Date( System.currentTimeMillis( )));
        }
}
// Implementation and registry of server
public class RemoteTimeServer{
        public static void main( String args[ ] ) throws Exception{
                ORB orb = ORB.init( args, null );
                ExactTimeServer ets = new ExactTimeServer( );
                orb.connect( ets );
                org.omg.CORBA.Object obj = orb.resolve_initial_references(
                                                                "NameService" );
                NamingContext nct = NamingContextHelper.narrow( obj );
                NameComponent nc = new NameComponent( "ExactTime", " " );
                nct.rebind( path, ets );
                java.lang.Object sync = new java.lang.Object( );
                synchronized( sync ){
                        sync.wait( );
                }
        }
}
```

Jei ši programa būtų paleista, būtų sukurtas serveris ir jis lauktų klientų užklausų.

Kliento programa. Sąsają į objektą serveryje programa gauna taip pat per tarpininką ORB:

```
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient{
        public static void main( String args[ ] ) throws Exception{
                ORB orb = ORB.init( args, null );
                Org.omg.CORBA.Object obj = orb.resolve_initial_references(
                                                                "NameService" );
                NamingContext nct = new NamingContextHelper.narrow( obj );
                NameComponent nc = new NameComponent( "ExactTime", " " );
                NameComponent [ ] path = { nc };
                ExactTime et = ExactTimeHelper.narrow(  nct.resolve( path ));
                String exactTime = et.getTime( );
                System.out.println( exactTime );
        }
}
```

Kad visas programinis ūkis veiktų, būtina, panašiai kaip RMI technologijoje, suaktyvinti serverį, klientą, sąsajų klasę, programą-kamštį ir programą-karkasą bei dar *Java*DL identifikavimo tarnybą, kuri pagal nutylėjimą klauso 900-ojo prievado.

Šiame pavyzdyje ir serveris, ir klientas yra tame pačiame kompiuteryje, tačiau jie realizuojami dviejų skirtingų JVM.