2. *JAVA* SINTAKSĖ IŠSAMIAU

2.1. Sąlygos operatoriai

Java sąlygos operatoriai – lygiai tokie pat, kaip kitose kalbose:

```
if( reiškinys )
        { operatorių grupė 1 }
else
        { operatorių grupė 2 }
```

Reiškinio reikšmė privalo būti loginė. Jei grupėse yra tik po vieną operatorių, figūriniai skliaustai nebūtini. *else* ir antroji operatorių grupė taip pat nebūtini. Jei reikia, prie *else* galima prirašyti papildomą sąlygą: *else if(reiškinys) { operatorių grupė }*.

Pavyzdys: skaičiaus įvedimas iš klaviatūros ir nustatymas, ar skaičius lyginis, ar nelyginis.

```
. . .
int number = System.in.read( ); // įvedamas vienas simbolis iš klaviatūros;
if( number %2 = = 0 )           // jei įvesti keli simboliai – imamas paskutinis
        System.out.println( "Even number entered" );
else
        System.out.println( "Odd number entered" );
. . .
```

2.2. Selektorius

```
switch( reiškinys ) {
        case value1:
                { operatorių grupė 1 }
                break;
        case value2:
                { operatorių grupė 2 }
                break;
        . . .
        default:
                { operatorių grupė d }
                break;
}
```

Operatoriaus reiškinio reikšmė turi būti tik *byte*, *char*, *short* arba *int* tipo. Operatorių *break* gali nebūti; tada bus vykdoma kita iš eilės operatorių grupė.

Pavyzdys: programa nustato, ar iš klaviatūros įvestas simbolis yra balsė, ar priebalsis (pagal anglų kalbos taisykles).

```
public class VowelsAndConsonants{
        public static void main( String args[ ] ){
                char c = ( char ) System.in.read( );
                switch( c ) {
                        case ‘a’:
                        case ‘e’:
                        case ‘i’:
                        case ‘o’:
                        case ‘u’:
                                System.out.println( “Vowel” );
                                break;
                        case ‘y’:
                        case ‘w’:
                                System.out.println( “Sometimes vowel” );
                                break;
                        default:
                                System.out.println( “Consonant” );
                }
        }
}
```

2.3. Ciklai

1.

while (reiškinys) { operatorių grupė }

Operatoriaus reiškinio reikšmė turi būti loginė. Galimas atvejis, kai ciklas iš viso nebus vykdomas – kai reiškinio reikšmė *false*; galimas begalinio ciklo atvejis – kai reiškinio reikšmė pastovi – *true*.

Pavyzdys: tarkim metodas *processChar* kažkokiu būdu apdoroja iš klaviatūros įvestą simbolį; darbas nutraukiamas įvedus simbolį *q*.

```
. . .
char c = (char) System.in.read( );
while( c != ‘q’ ) {
        processChar( c );
        c = (char) System.in.read( );
}
. . .
```

2.

```
do { operatorių grupė }
while( reiškinys );
```

Šio operatoriaus skirtumas nuo pirmojo – jis bent vieną kartą visada bus vykdomas. Dabar ankstesnis pavyzdys su šiuo operatoriumi:

```
. . .
do {
        char c = (char) System.in.read( );
        processChar( c );
}
while ( c != ‘q’ );
. . .
```

3.

for(reiškinys1; reiškinys2; reiškinys3) { operatorių grupė }

reiškinys1 suteikia pradinę ciklo skaitiklio reikšmę, *reiškinys2* nurodo, kaip ciklo skaitiklio reikšmę modifikuoti, o loginis *reiškinys3* – iki kada ciklą vykdyti. Aišku, *reiškiniai* gali būti supaprastėję iki kintamojo ar konstantos.

Vėl perrašysime tą patį pavyzdį:

```
. . .
for( char c = (char) System.in.read( );  c != ‘q’;  c = (char) System.in.read( ) )
        processChar( c );
. . .
```

for cikluose galima naudoti operatorių-kablelį. Pavyzdžiui:

```
. . .
for( int i = 1, j = i+10;  i < 5;  i++, j = i*2 ) {
        System.out.println( “i= “+i+” j= “+j );
}
. . .
```

Spausdinimo rezultatai:

```
i = 1  j = 11
i = 2  j = 4
i = 3  j = 6
i = 4  j = 8
```

2.3.1. Ciklų žymėjimas. Ciklų nutraukimas

Nors *Java* ir yra griežtai objektiškai orientuota kalba, vis tik ir joje faktiškai yra programavimo puristų smerkiamas operatorius *go to*. *Java* jo funkcijas atlieka ciklų žymėjimas ir ciklo nutraukimo operatorius *break* bei valdymo perdavimo į žymę operatorius *continue*. Šios konstrukcijos turi prasmę kartotiniuose cikluose.

Tarkime, apdorojamas kvadratinis masyvas *array*, turintis *arraySize* eilučių ir stulpelių. Masyvo eilinis elementas apdorojamas metodu *processArrayElement.* Jei randamas bent vienas neigiamas elementas, – tolesnis masyvo apdorojimas nutraukiamas: nutraukiami abu ciklai.

```
. . .
err:  // žymė žemiau esančiam operatoriui – for
for( i = 0; i < arraySize; i++ ) {
        for( j = 0; j < arraySize; j++ ) {
                if( arry[ i ] [ j ]  <  0 ) {
                    System.out.println( "Error: negative number encountered"
);
                    break err; // nutraukiamas žymėtasis - išorinis ciklas
                }
                processArrayElement( array[ i ] [ j ] );
        }
}
. . .
```

2.4. Metodas-konstruktorius. Konstruktoriaus ir metodo perkrovimas

Metodas-konstruktorius garantuoja kiekvieno programos objekto pradinės reikšmės(-ių) prieskyrą. Jis automatiškai kviečiamas sukuriant objektą operatoriumi *new*. Konstruktorių bet kuriai programuotojo parašytai klasei automatiškai sukuria *Java*. Esant reikalui (pavyzdžiui, klasės laukams priskirti ne nulines pradines reikšmes) tokį metodą gali parašyti ir pats programuotojas. Konstruktoriaus vardas, skirtingai nuo įprastų metodų, pradedamas didžiąja raide. Konstruktorius gali turėti argumentų. Konstruktorius negrąžina jokios reikšmės.

Pavyzdžiui, „tuščios" klasės konstruktorius gali būti parašytas taip:

```
class AClass{
        float x;
        AClass( ) {
                System.out.println( "Creation of AClass object" );
                 x = 1.23f;
        }
}
```

Dabar *new AClass( );* kviečia konstruktorių, išskiria atmintį objektui – klasės laukams ir sukuria vieną klasės objektą.

Konstruktorius su argumentais:

```
class AClass2{
        AClass2( int i ){
                System.out.println( "AClass object No "+ i );
        }
}
public class ObjectCreation{
        public static void main( String args [ ] ){
                for( int i = 0; i < 10; i++ ) {
                        new AClass2( i );
                }
        }
}
```

sukuria 10 objektų su numeriais 0, 1, 2, …, 9. Aišku, šie objektai neturi jokių į save nukreiptų rodyklių, todėl juos sunaikins šiukšlių rinktuvas.

Tos pačios klasės objektus galima sukurti skirtingais būdais. Tam tektų metodą-konstruktorių, panašiai kaip paprastą metodą, „perkrauti“ (*overload*). Perkraunant metodus, gaunama visa bendravardžių metodų grupė. Metodas nuo metodo šioje grupėje turi skirtis argumentų sąrašu (gali skirtis net ir argumentų išdėstymo tvarka). Kompiliatorius iš metodų grupės reikiamą metodą atsirenka pagal kvietimo metu nurodytą argumentų sąrašą.

Pavyzdys su metodų-konstruktorių ir metodų perkrovimu:

```
class AClass3{
        int number;
        AClass3( ){
                System.out.println( “Simple creation” );
                number = 0;
        }
        AClass3( int i ){
                System.out.println( “Numbered creation. No ”+i );
                number = i;
        }
        void info( ){
                System.out.println( “Object “+ number );
        }
        void info( String s ){
                System.out.println( s + “ object “ + number );
        }
}
public class Overloading{
        public static void main( String args [ ] ){
                AClass3 ac = new AClass3( );
                ac.info( );
                ac.info( “Overloaded” );
                for( int i = 0; i < 10; i++ ) {
                        ac = new AClass3( i );
                }
        }
}
```

Skiriasi metodų su argumentais – primityvais perkrovimas. Taip yra dėl to, kad primityvai automatiškai gali būti pervesti į bendresnįjį tipą. Išimtis čia yra tik *char* primityvas: jis pervedamas iškart į *int* tipą, apeinant mažiau už *int* bendrus *byte* ir *short* tipus. Visi galimi atvejai parodyti šiame pavyzdyje:

```
public class PrimitiveOverloading{
        static void prt(String s){
                System.out.println(s);
        }

        void m1(char x){ prt("m1(char)"); }
        void m1(byte x){ prt("m1(byte)"); }
```

```
void m1(short x){ prt("m1(short)"); }
void m1(int x){ prt("m1(int)"); }
void m1(long x){ prt("m1(long)");}
void m1(float x){ prt("m1(float)"); }
void m1(double x){ prt("m1(double)"); }

 void m2(byte x){ prt("m2(byte)"); }
 void m2(short x){ prt("m2(short)"); }
void m2(int x){ prt("m2(int)"); }
void m2(long x){ prt("m2(long)");}
void m2(float x){ prt("m2(float)"); }
void m2(double x){ prt("m2(double)"); }

void m3(short x){ prt("m3(short)"); }
void m3(int x){ prt("m3(int)"); }
void m3(long x){ prt("m3(long)");}
void m3(float x){ prt("m3(float)"); }
 void m3(double x){ prt("m3(double)"); }

 void m4(int x){ prt("m4(int)"); }
void m4(long x){ prt("m4(long)");}
void m4(float x){ prt("m4(float)"); }
 void m4(double x){ prt("m4(double)"); }

void m5(long x){ prt("m5(long)");}
 void m5(float x){ prt("m5(float)"); }
void m5(double x){ prt("m5(double)"); }

 void m6(float x){ prt("m6(float)"); }
void m6(double x){ prt("m6(double)"); }

void m7(double x){ prt("m7(double)"); }

void testChar( ){
          char x ='x';
           prt("char argument");
            m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
 }

void testByte( ){
           byte x =1;
          prt("byte argument");
          m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
}

 void testShort(){
          short x =1;
           prt("short argument");
            m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
 }
```

```
        void testInt(){
                int x =1;
                prt("int argument");
         m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
        }

        void testLong(){
                 long x =1L;
                prt("long argument");
                m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
       }

        void testFloat(){
                float x =1F;
                prt("float argument");
                 m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
        }

         void testDouble(){
                double x =1;
                prt("double argument");
                m1(x); m2(x); m3(x); m4(x); m5(x); m6(x); m7(x);
         }

        public static void main(String args[]){
                PrimitiveOverloading pl = new PrimitiveOverloading();
                pl.testChar();
                 pl.testByte();
                 pl.testShort();
                 pl.testInt();
                 pl.testLong();
                 pl.testFloat();
                 pl.testDouble();
        }
}
```

Šios programos spausdinimai akivaizdžiai rodo, kuris metodas pasirenkamas iš visų perkrautų metodų grupės, kai trūksta metodo su nurodytu kvietimo metu argumento tipu: pasitelkiamas minėtasis automatinis pervedimas į bendresnįjį tipą. Rezultatus pateiksime lentelės forma:

char argument	*byte argument*	*short argument*	*int argument*
m1(char)	*m1(byte)*	*m1(short)*	*m1(int)*
m2(int)	*m2(byte)*	*m2(short)*	*m2(int)*
m3(int)	*m3(short)*	*m3(short)*	*m3(int)*
m4(int)	*m4(int)*	*m4(int)*	*m4(int)*
m5(long)	*m5(long)*	*m5(long)*	*m5(long)*
m6(float)	*m6(float)*	*m6(float)*	*m6(float)*
m7(double)	*m7(double)*	*m7(double)*	*m7(double)*

(lentelės tęsinys)

long argument	*float argument*	*double argument*
m1(long)	*m1(float)*	*m1(double)*
m2(long)	*m2(float)*	*m2(double)*
m3(long)	*m3(float)*	*m3(double)*
m4(long)	*m4(float)*	*m4(double)*
m5(long)	*m5(float)*	*m5(double)*
m6(float)	*m6(float)*	*m6(double)*
m7(double)	*m7(double)*	*m7(double)*

Ir paskutinė pastaba apie metodų-konstruktorų perkrovimą. Metodas-konstruktorius, kurį sukuria *Java* pagal nutylėjimą – be argumentų. Jei programuotojas sukuria savo konstruktoriaus variantą su argumentais, tai dabar kreipinys į metodą-konstruktorių be argumentų tampa negalimas: kompiliatorius automatiškai konstruktoriaus nebekuria. Generuojamas pranešimas „nėra tokio konstruktoriaus“.

2.5. Raktažodis *this*

Kompiliatorius, atlikdamas kurio nors metodo instrukcijas, dabartiniu momentu apdorojamam objektui sukuria rodyklę *this*. Šią rodyklę galima naudoti kaip ir bet kurią kitą. Dažniausiai šis raktažodis naudojamas konstruktoriuose.

Aptarsime situacijas su *this* naudojimu.

1.

```
. . .
class AClass4{
        void m1( ) { . . .}
        void m2( ) { this.m1( ); }
}
. . .
```

Šiuo atveju, kai vienas metodas kviečia kitą tos pačios klasės viduje, *this* nėra būtinas. Čia jis paprasčiausiai pabrėžia, kad metodas *m1* kviečiamas dabar apdorojamam objektui.

2. Raktažodis būtinas, kai paprastas metodas grąžina klasės objektą:

```
public class AClass5{
        int i = 0;
        AClass5 incr( ) {
                i++;
                return this;
        }
        public static void main( String args [ ] ) {
                AClass5 ac = new ACLass5( );
                ac.incr( ); // kadangi incr grąžina rodyklę į darbinį objektą,
        }                   // jam toliau galima tęsti inkremento operacijas:
}                           // ac.incr ( ).incr( ).incr( ) . . .;
```

3. Raktažodis būtinas, konstruktoriuje kviečiant kitą, mažiau bendrą metodo-konstruktoriaus variantą. Čia pat ir pavyzdys (pirmasis *AClass6* konstruktoriaus variantas), kai be *this* kiltų vardų konfliktas. Be abejo, konflikto lengva išvengti parenkant argumentui kitokį vardą (tą demonstruoja antrasis konstruktoriaus variantas). Tačiau daugelis autorių propaguoja programavimo stilių kaip pirmajame konstruktoriaus variante – su *this* naudojimu.

```
public class AClass6{
        int i = 0;
        String s = "null";
        AClass6( int i ) {
                this.i = i;
                System.out.println( "Constructor1: only int" );
        }
        AClass6( String ss ) {
                s = ss;
                System.out.println( "Constructor 2: only String" );
        }
        AClass6( int ii, String ss ) {
                this( ii );
                // this( ss ); // klaida: leidžiamas tik vienas toks kvietimas
                s = ss;
                System.out.println( "Constructor3: int and String" );
        }
        AClass6( ) {
                this( 1, "string" );
                System.out.println( "Constructor4: no arguments" );
        }
        public static void main( String args [ ] ) {
                AClass6 ac = new AClass6( );            // kviečiamas 4-asis
                ac.System.out.println( "Object: " + i + s ); // konstruktorius
        }
}
```

2.6. Objektų naikinimas

Programuotojui nereikia pačiam rūpintis *Java* programose objektams skirtos atminties išlaisvinimu, tą darbą atlieka „šiukšlių rinktuvas". Bendrasis objektų naikinimo mechanizmas yra labai paprastas: kai nebėra nė vienos į objektą nutaikytos rodyklės, jam skirta atmintis surenkama.

Šiukšlių surinkimas išlaisvina tik objektams skirtą atmintį ir neliečia kitų objektams skirtų resursų. Jei objektas, pavyzdžiui, kažką nupiešė kompiuterio ekrane, tai, sunaikinant objektą, piešinio panaikinimu tenka rūpintis pačiam programuotojui.

Šiukšlės surenkamos ne visada. Jei programa naudoja nedaug atminties ir nepasiekia tam tikros kritinės atminties panaudojimo ribos, atminties surinkimo mechanizmas nepaleidžiamas. Šiuo atveju atmintis operacinei sistemai grąžinama programai baigus darbą.

Klasėje *System* yra specialus metodas, priverstinai iškviečiantis darbui atminties rinktuvą:

System.gc( );

Metodo pavadinimas kilęs nuo žodžių *garbage collection* pirmųjų raidžių, kas šiaip jau nebūdinga *Java* vardų sistemai.

Šiukšlių rinktuvo darbo algoritmai yra pakankamai sudėtingi: naudojama schema „sustoti – ir – perkopijuoti“ programai skirtą atmintį (*heap*) į kitą atminties sritį; perkopijuojami tik objektai, turintys į save nukreiptas rodykles (rodyklių pradedama ieškoti nuo pačiame klasių hierarchijos viršuje esančios klasės *Object*). Be to, įvairios JVM realizacijos šią schemą dar vienaip ar kitaip patobulina.

2.7. Pradinių reikšmių skyrimas

Primename: klasių kintamiesiems automatiškai skiriamos nulinės reikšmės (loginiams laukams – reikšmė *false*). Matyt, geresnis programavimo stilius vis tik yra laukų reikšmes išreikštai skirti prieskyros operatoriuose arba tai perkelti į metodus-konstruktorius.

Skiriant pradines reikšmes išreikštai klasės viduje, operatorių vykdymo tvarka yra gana neįprasta: pirmiau, nepaisant, kur jie yra, atliekami pradinių reikšmių prieskyros operatoriai, o tik paskui visi kiti metodai. Tą iliustruoja šis pavyzdys:

```
class Tag{
        Tag( int i ){
                System.out.println( "Tag( " + i + " )" );
        }
}

class Card{
        Tag t1 = new Tag( 1 ); // prieš konstruktorių !
        Card( ){
                System.out.println( "Inside constructor Card" );
                t3 = new Tag( 33 );
        }
        Tag t2 = new Tag( 2 );
        void m( ){
                System.out.println( "Finish" );
        }
       Tag t3 = new Tag( 3 ); // after constructor !
}

public class Initialization{
        public static void main( String args[ ] ) {
                Card c= new Card( );
                c.m( );
        }
}
```

Programos spausdinimai:

Tag(1)
Tag(2)
Tag(3)
Inside constructor Card
Tag(33)
Finish

Statiniams laukams pradinės reikšmės skiriamos tik tada, kai programa sukuria objektus su statiniais laukais.

Taigi visa pradinių reikšmių skyrimo tvarka yra:

1. Interpretatorius ieško klasės sukompiliuotos rinkmenos vardu *klasėsVardas.class*, kai bandoma sukurti šios klasės objektą arba prieiti prie šios klasės statinių laukų ar metodų. Informacija, kur interpretatorius turi ieškoti, nurodoma aplinkos kintamuoju *CLASSPATH*.
2. Įkėlus rinkmeną *klasėsVardas.class*, klasės statiniams laukams paskiriamos pradinės reikšmės, ir tik vieną kartą.
3. Kuriant naują klasės *klasėsVardas* objektą operatoriumi *new*, išskiriama atmintis klasės nestatiniams laukams ir jiems skiriamos nulinės reikšmės.
4. Jei yra, atliekama išreikštinė šių laukų reikšmių prieskyra.
5. Vykdomi metodai-konstruktoriai.

2.8. Pradinių reikšmių prieskyra masyvams

Masyvas – vienodo tipo objektų seka vienu vardu. Gali būti vienmačiai, dvimačiai ir t. t. Vienmatis masyvas apibrėžiamas, pavyzdžiui, taip:

int [ ] m; arba *int m[ ];*

Čia negalima nurodyti elementų kiekio masyve, nes kol kas sukurta tik nuoroda, o atmintis masyvui dar nepaskirta. Atmintis masyvui paskiriama jam skiriant pradines reikšmes. Tai daroma keliais būdais:

1.
```
int [ ] m1 = { 1, 2, 3, 4, 5 }; // supaprastinta  sintaksė
                                // primityvams ir …
Integer [ ] m2 = { new Integer( 1 ), new Integer( 2 ), new Integer( 3 ),
                   new Integer( 4 ), new Integer( 5 ) }; // …objektams
```

2.
```
int [ ] m3;
. . .
m3 = new int[ 5 ];  // ir primityvų masyvams galimas „new“ naudojimas;
                    // visiems 5 masyvo elementams skiriamos reikšmės 0
```

Pradinių reikšmių skyrimo daugiamačiams masyvams abiem būdais pavyzdžiai:

```
int [ ]  [ ] m4 ={                  // dvimatis  masyvas 2x3
                  { 1, 2, 3 },
                  { 4, 5, 6 },
                };
int [ ]  [ ]  [ ]  m5 = new int[ 2 ] [ 3 ] [4 ];  // trimatis nulinis masyvas 2x3x4
```

Kiekvienas dvimačio masyvo elementas *m4[i]* yra vienmatis masyvas; į šį masyvą galimas toks kreipinys kaip parašyta. Be to, kiekvienam indeksui *i* vienmačiai masyvai gali būti skirtingo ilgio. Pavyzdžiui, trimačiam masyvui *m5* galima tokia pakopinė pradinių reikšmių prieskyra:

```
m5 = new int[ 2 ] [ ] [ ];
. . .
int i = . . . ;
int j = . . . ;
m5[ i ] = new int[ j ] [ ];
. . .
int k = . . . ;
m5[ i ] [ j ] = new int[ k ];
. . .
```

Kiekvienas masyvas turi ilgio lauką *length* kiekvienai dimensijai. Pavyzdžiui, tam pačiam trimačiam masyvui jo atskiras dimensijas galima būtų nustatyti:

```
m5.length;              // pirmoji dimensija
m5[ i ].length;         // i elemento – vienmačio masyvo, kurio elementai irgi
                        // vienmačiai masyvai, ilgis
m5[ i ] [ j ].length;   // ij elemento – vienmačio masyvo –  ilgis
```

Java, kaip ir C kalbose, masyvų elementų indeksai kiekvienoje dimensijoje pradedami skaičiuoti nuo nulio.

2.9. Vardų, laukų, metodų ir klasių slėpimas

2.9.1. Paketai

Paketai visų pirma skirti sutvarkyti *Java* rinkmenų sistemai, panaudojant OS rinkmenų hierarchiją. Kartu tai patogi priemonė vardų matomumo sritims valdyti.

Pradinis programos tekstas (kuriame gali būti kelios klasės) vadinamas kompiliacijos vienetu. Tokiame vienete gali būti tik viena viešoji klasė (*public*), o jį apimančios rinkmenos vardas turi sutapti su viešosios klasės vardu ir turėti plėtinį **.java*. Jei kompiliacijos vienete nė viena klasė nėra paskelbta *public,* rinkmenos vardas gali būti bet koks. Sukompiliavus tokį vienetą, bus sukurtos rinkmenos **.class* kiekvienai klasei. Vykdant programą, šias rinkmenas suranda, įkrauna ir interpretuoja interpretatorius – *Java* virtuali mašina (JVM). Beje, **.class* rinkmenas galima suspausti į *Java* archyvo rinkmeną **.jar* įrankiu *jar*.

Norint sugrupuoti kelis kompiliacijos vienetus į visumą – paketą, prieš *public* klasę būtinai pirmojoje eilutėje (neskaitant komentarų) rašomas raktažodis

package packageName;

Skirtingai nei klasei, paketo vardą įprasta pradėti mažąja raide. Dabar, norint kreiptis į *public* klasę (tarkime, jos vardas yra *AClass*), jau teks naudoti pilnąjį jos vardą *packageName.AClass* arba importuoti paketo klases operatoriumi

import packageName;

Šiuo atveju kreipinyje vėl galimas trumpasis klasės vardas. Operatorius

import packageName.;*

importuoja tik reikiamas programai paketo klases – vadinamasis dinaminis importavimas.

Kadangi *Java* programos pirmiausia skirtos internetui, tinkle neišvengiamai gali kilti ir paketų vardų konfliktas. Šiai problemai išspręsti rasta tokia išeitis: paketo vardui naudojamas atvirkščias domeninis kompiuterio adresas ir kompiuterio rinkmenų hierarchija, skiriant domenus ir aplankų vardus taškais. Kartu tai sutvarko ir rinkmenų ūkį: visos paketo rinkmenos **.class* yra viename aplanke. Kur kompiuteryje ieškoti reikiamų klasių, kai kuriose OS nurodoma aplinkos kintamuoju, pavyzdžiui,

CLASSPATH = .; C:\A\B\C; C:\D\e.jar

Čia taškas reiškia darbinį aplanką. Archyvinėms rinkmenoms, kaip matyti iš pavyzdžio, reikia nurodyti ir rinkmenos vardą.

Kitas pavyzdys: tarkim, kompiuterio internetinis adresas yra *domeninisAdresas.com*, o kompiuteryje jis „rodo“ į *C:\ABC\com.domeninisAdresas*, šiame aplanke yra aplankas *def*, kuriame ir bus paketas. Jei kompiuterio aplinkos kintamasis nustatytas *CLASSPATH = C:\ABC*, tai paketo vardą reikia skelbti

package com.domeninisAdresas.def;
public class AClass{ . . . }
. . .

Vėliau, importavus paketą:

import com.domeninisAdresas.def.;*

bus galima naudoti trumpąjį klasės vardą *AClass*. Beje, importuojamos tik viešosios klasės.

Jei *Java* programoje nėra deklaruotas paketas, tačiau kelios klasės yra viename kompiuterio rinkmenų aplanke, tai šios klasės vis tiek laikomos paketo „pagal nutylėjimą“ nariais.

2.9.2. Laukų ir metodų slėpimas

Paketai ir klasės palaiko įvairias vardų, arba matomumo, sritis. Paketai faktiškai yra konteineriai klasėms ir kitiems paketams. Klasės yra duomenų ir kodo konteineriai. Tai leidžia objekto duomenis, kuriuos nebūtinai turi žinoti visi kiti programos objektai, paslėpti (*incapsulate*) objekte arba keliuose susijusiuose objektuose.

Iš viso yra 5 prieities lygiai klasės elementams:

a) prieitis iš tos pat klasės;
b) iš klasės subklasių šiame pakete;
c) iš ne subklasių šiame pakete;
d) iš subklasių skirtinguose paketuose;
e) iš ne subklasių kituose paketuose.

Terminas *subklasė* čia reiškia, kad ji *paveldi* nagrinėjamos klasės savybes (žr. 2.10 skyrių).

Prieš klasės elementus galima įrašyti prieities modifikatorius *public*, *private* arba *protected*. Modifikatorius veikia tik tą objektą, prieš kurį jis įrašytas. Jei joks modifikatorius neįrašytas, prieitis prie lauko yra *friendly* (tačiau tokio raktažodžio nėra). Modifikatorių reikšmė iliustruojama 2.1 lentelėje: prieitis galima arba negalima.

2.1 lentelė. Modifikatorių reikšmės

Modifikatorius / Prieities lygis	*private*	*„friendly“*	*protected*	*public*
a	taip	taip	taip	taip
b	ne	taip	taip	taip
c	ne	taip	taip	taip
d	ne	ne	taip	taip
e	ne	ne	ne	taip

Taigi elementai, paskelbti *public*, yra prieinami iš visur. Elementai *private* prieinami tik iš pačios klasės. Elementai be modifikatoriaus matomi klasėje, jos subklasėse ir kitose to paties paketo klasėse. Elementai, kurie turi būti matomi ir kitų paketų klasėse – nagrinėjamos klasės subklasėse, skelbiami *protected*.

Programos su klaidinga prieitimi pavyzdys:

```
package aPackage;
public class Class1{
        int a = 0;
        void m( ){
                System.out.println( “AClass method” );
        }
}
import aPackage.*;
public class Class2{
        public static void main( String args[ ] ){
```

```
            Class1 c1 = new Class1( );
            c1.a = 1; // Klaida!
            c1.m( );  // Klaida!
        }
}
```

Norint šią programą sukompiliuoti be klaidų, teks arba skelbti pirmosios klasės lauką ir metodą viešais, arba *protected* ir susieti antrąją klasę su pirmąja paveldimumu (žr. 2.10 skyrių).

2.9.3. Klasių slėpimas

Klasei galimas tik modifikatorius *public* ir prieitis pagal nutylėjimą *friendly*. Norint vienaip ar kitaip uždaryti prieigą prie klasės, reikia pasitelkti tam tikras technologijas.

Pavyzdys. Technologija vieninteliam klasės objektui sukurti: perrašyti metodą-konstruktorių ir paskelbti jį *private* – iš už klasės ribų nebus įmanoma sukurti klasės objektų; klasės viduje sukurti vienintelį klasės objektą; sukurti *static* metodą (kad būtų galima kreiptis klasės vardu), grąžinantį šį sukurtąjį klasės objektą.

```
class AClass{
        private AClass( ){ . . . }
        private static AClass ac = new AClass( );
        public static AClass access( ){
                return ac;
        }
        public void m( ){ . . . }
}

. . .
AClass.access( ).m( );
. . .
```

2.10. Kodo fragmentų kartojimas: kompozicija ir paveldimumas

2.10.1. Sintaksė. Konstruktorių vykdymas

Kad būtų greičiau rašomas programos kodas, kiekviena programavimo kalba siūlo tam tikrus mechanizmus, leidžiančius pakartoti reikiamus kodo fragmentus. *Java* tokie mechanizmai yra klasių kompozicija ir paveldimumas.

Kompozicija taikytina, kai naujai rašomoje klasėje reikia panaudoti jau parašytų klasių galimybes ir charakteristikas. Supaprastintai kompoziciją galima apibūdinti teiginiu „tai yra to dalis“. Klasikinis pavyzdys, kuriuo kompozicija iliustruojama objektinio programavimo vadovėliuose – klasė „mašina“, susidedanti iš klasės „variklio“ objekto, kelių klasės „ratai“ objektų ir t. t.

Paveldimumas naudotinas tada, kai reikia panaudoti motininės, arba superklasės, sąsają; subklasė gali šią sąsają išplėsti. Teiginys, kurį įgyvendina paveldimumas – „šitai yra panašu į tai“.

Lankstesnis programavimo mechanizmas yra kompozicija, o paveldimumą reikėtų taikyti ten, kur reikalingas „kylantysis tipų pertvarkymas“ (žr. tolesnį skyrių).

Kompozicijos sintaksė yra aiški: rašomoje klasėje kuriami jau egzistuojančių klasių objektai, todėl dėmesį sutelksime į paveldimumo mechanizmą.

Kad subklasė paveldi superklasės duomenis ir metodus, išreiškiama raktažodžiu *extends* ir superklasės vardu. Pavyzdys:

```
// Inheritance
//
class Class1 {
        private String s = "Class1:  ";
        public void append( String ss ) { s += ss; }
        public void m1( ) { append( "m1()  " );}
        public void m2( ) { append( "m2()  " );}
        public void m3( ) { append( "m3()  " );}
       public void print( ) { System.out.println( s );}
        public static void main( String [ ] args ) {
                Class1 c1 = new Class1();
                c1.m1(); c1.m2(); c1.m3(); c1.print();
         }
}
public class Class2 extends Class1 {
        public void m3( ){   // overriding method m3
                append( "Class2.m3( )  " );
                super.m3( );       // calling m3 of superclass
        }
        public void m4( ) { append( "   m4( )" ); } // extension of interface
       public static void main( String [ ] args ) {
                Class2 c2 = new Class2( );
                c2.m1( ); c2.m2( ); c2.m3( ); c2.m4( ); c2.print( );
               System.out.println( "Test for the superclass Class1:" );
               Class1.main( args ); // args already defined
        }
}
```

Programos rezultatai, kai kviečiamas antrosios klasės metodas *main*:

```
Class1:   m1( )   m2( )   Class2.m3( )m3( )   m4( )
Test for the superclass Class1:
Class1:   m1( )   m2( )   m3( )
```

Šiame pavyzdyje yra iškart keli nauji programavimo mechanizmai ir technologijos:

1. Žingsniniam klasių testavimo technologija: į kiekvieną klasę įdedamas metodas *main*. Dabar interpretatoriui galima nurodyti, nuo kurios klasės

pradėti programos vykdymą: komanda *java Class2* paleis visos programos vykdymą, o *java Class1* leistų testuoti tik pirmąją klasę.

2. Kai klasės susietos paveldimumu, kuriant subklasės objektą, automatiškai bus sukurtas superklasės objektas. Todėl kreipinys *c2.m1( );* spausdins *Class1: m1( )*, kadangi konstruojant superklasės objektą, eilutei *s* priskiriama reikšmė *Class1:* .
3. Superklasės metodą subklasėje galima perrašyti arba kitaip modifikuoti (*override*): kreipinys į metodą lygiai toks pat – *public void m3( )*, o metodo kūnas perrašomas kitaip. Šiame perrašytame kūne galima kreiptis ir į bet kurį superklasės metodą raktažodžiu *super*: *super.m3( );* . Beje, jei šioje eilutėje raktažodžio *super* nebūtų, būtų rekursyviai kreipiamasi į subklasės metodą *m3* .

Dabar truputį plačiau apie paveldimumu susietų klasių konstruktorių vykdymą. Kadangi subklasėje yra „įdėtas“ ir praplėstas superklasės objektas, tai, konstruojant subklasės objektą, automatiškai prieš tai sukonstruojamas superklasės objektas. Tą matėme iš ankstesnio pavyzdžio. Jei konstruktoriai turi argumentus, tai iš subklasės konstruktoriaus juos reikia raiškiai perduoti „į viršų“ panaudojant raktažodį *super* su argumentais, kaip parodyta šiame pavyzdyje:

```
// Inheritance: constructors
//
class Class3 {
        Class3( int i ) {
                System.out.println( "Constructor of Class3  "+i );
        }
}
class Class4 extends Class3 {
        Class4( int i ) {
                super( i ); // calling constructor of superclass
                            // Class3
                i++;  // scope of "i" is limited within constructor's
                      // area
                System.out.println( "Constructor of Class4 " +i );
        }
}
public class Class5 extends Class4 {
        Class5( int i ) {
                super( i ); // calling constructor of superclass
                                  // Class4
                i++;
                System.out.println( "Constructor of Class5" +i );
        }
        public static void main ( String [] args ) {
                 Class5 c5 = new Class5( 123 );
        }
}
```

Programos spausdiniai:

Constructor of Class3 123
Constructor of Class4 124
Constructor of Class5 124

Jei jokių argumentų perduoti nereikėtų, superklasės konstruktoriaus kvietimui žodžio *super* nereikėtų – jis būtų kviečiamas automatiškai.

Kreipinys *super( )* subklasės konstruktoriuje privalo būti pats pirmas operatorius, o tai neleidžia perimti jokių išimčių (žr. skyrių apie klaidų apdorojimą), kylančių konstruktoriuje.

2.10.2. Kylantysis tipų pertvarkymas

Tai svarbiausias paveldimumo aspektas, kurį galima išreikšti teiginiu „ši nauja klasė yra superklasės tipo“. Iliustracijai – B. Eckelio pavyzdys:

```
class Instrument{
        public void play( ){ . . . }
        static void tune( Instrument i ){
                i.play( );
                . . .
        }
}
class Wind extends Instrument{
        public static void main( String args[ ] ) {
                Wind flute = new Wind( );
                Instrument.tune( flute ); // upcasting here!
        }
}
```

Kaip matyti, metodo *tune* argumentas yra *Instrument* tipo, tačiau kreipinyje jam atiduodamas *Wind* tipo argumentas. Tokia sintaksė teisinga, nes *Wind* klasė yra subklasė *Instrument* klasei, o vietoje „žemesniojo“ klasės tipo visada leidžiama naudoti „aukštesniosios“ klasės tipą (todėl ir terminas toks – „kylantysis tipų pertvarkymas“ (*upcasting*)). Toks tipų pertvarkymas visada yra įmanomas ir saugus, nes subklasė turi visus superklasės metodus (gali turėti ir daugiau metodų), todėl superklasės metodą visada bus įmanoma atlikti.

Toks tipų pertvarkymas atveria polimorfizmo galimybę, kurią vėliau išsamiai nagrinėsime.

2.10.3. Raktažodis *final* ir paveldimumas

Raktažodis *final* visada reiškia kažką nekintančio. Taikomas klasėms, duomenims arba metodams.

Jei modifikatorius taikomas klasei, jos subklasių sukurti (t. y. paveldėti jos savybių) neleidžiama. Tokios klasės laukų reikšmes keisti galima, metodus – taip pat.

Pritaikytas duomenims raktažodis priklausomai nuo konteksto gali reikšti duomens nekintamumą arba objektui, arba visiems klasės objektams.

Raktažodis prieš metodą reiškia, kad šis metodas negali būti jokioje klasės subklasėje nei perkrautas, nei perrašytas. Iš esmės metodams raktažodis naujų galimybių nesuteikia, nes *private* metodas savaime yra nekintantis; jo modifikuoti kitoje klasėje negalima – metodas paprasčiausiai ten nematomas.

final įtaka duomenims parodoma šiame pavyzdyje:

```
class Value{
        int i = 1;
}
class FinalData{
        final int i1 = 1; // constant for object
        static final int i2 = 2; // constant for class
        public static final int I3 = 3; // same; standard declaration
        final Value v1 = new Value( ); // constant pointer
        public static void main( String args[ ] ){
                FinalData fd = new FinalData( );
                fd.i1++;  // Error!
                fd.v1 = new Value( ); // Error!
                fd.v1.i++;
        }
}
```

Įprasta pastovų duomenį visai klasei skelbti taip, kaip laukui *I3*. Tokio lauko vardą įprasta rašyti didžiosiomis raidėmis. Kaip matyti, *final Value v1* padaro pastovią pačią rodyklę, todėl nebegalima jos nutaikyti į kitą atminties sritį, tačiau leidžiama keisti patį atminties turinį.

2.11. Polimorfizmas

Polimorfizmas – vienas iš pagrindinių objektinio programavimo principų. Dar vadinamas dinaminiu, arba vėlyvuoju, susiejimu (*dynamic, late binding*) – metodo vardo vėlyvuoju susiejimu su vykdomu kodu.

Dar vienas pavyzdys, panašus į jau nagrinėtąjį:

```
class Instrument{
        public void play( ){
                System.out.println( “Instrument.play( )” );
        }
}
class Wind extends Instrument{
        public  void play( ) {
                System.out.println( “Wind.play( )” );
        }
}
public class Music{
        public static tune( Instrument i ) {
                i.play( );
        }
        public static void main( String args [ ] ){
```

```
            Wind flute = new Wind( );
            tune( flute ); // upcasting here!
        }
}
```

Programos rezultatas:

Wind.play( )

Taigi, kviečiant metodą *tune,* ignoruojamas tikslus argumento tipas. Atrodo, paprasčiau metodą *tune* parašyti taip:

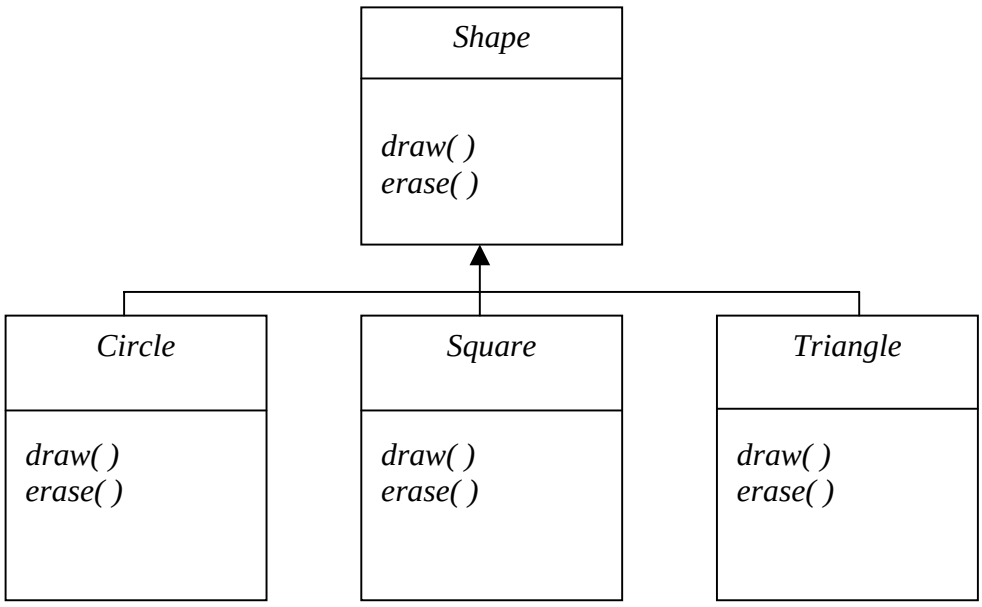
. . . tune(Wind i) { . . . }

Tačiau dabar, plečiant programą naujais instrumentų tipais *Brass, Horn, …*, kiekvienam naujam tipui klasėje *Music* teks modifikuoti metodą *tune*:

. . . tune(Brass i){ . . . } , . . . tune(Horn i){ . . . }, ...

Iš kur kompiliatorius žino, kad ankstesniame pavyzdyje būtent *Wind* klasės metodą reikia kviesti? Tą žino ne kompiliatorius, o JVM vėlyvojo susiejimo mechanizmas. Tam į sukompiliuotus *Java* objektus **.class* automatiškai įdedama kai kuri paslėpta informacija.

Kitas klasikinis polimorfizmui aiškinti naudojamas pavyzdys: programa figūroms, pavyzdžiui, *Circle, Square* ir *Triangle* braižyti ir trinti (metodai *draw* ir *erase*). Sukuriama apibendrinta abstrakti klasė „figūra“ (*Shape*), ir toliau visas kodas (klasė *Shapes*) rašomas šiam apibendrintam duomenų tipui, nekreipiant dėmesio į konkrečius tipus. Įprasta visą klasių hierarchiją vaizduoti UML (*Unified Modeling Language*) schemų pavidalu (2.1 pav.), kur nurodomi ryšiai tarp klasių ir klasių sąsajos – atvirų metodų sąrašai.



2.1 pav. UML schema klasių hierarchijai

Programos tekstas:

```
// Polymorphous methods: 1th example
class Shape {
        void draw( ){ }
        void erase( ){ }
}

class Circle extends Shape {
        void draw( ) {
                System.out.println(  "Circle.draw( )"  );
         }
         void erase( ) {
                System.out.println( "Circle.erase( )" );
        }
}

class Square extends Shape {
         void draw( ) {
                 System.out.println(  "Square.draw( )" );
         }
         void erase( ) {
                System.out.println( "Square.erase( )" );
         }
}

class Triangle extends Shape {
         void draw( ) {
                System.out.println( "Triangle.draw( )" );
        }
         void erase( ) {
                 System.out.println( "Triangle.erase( )" );
         }
}

public class Shapes {
        public static void main (  String [ ] args ) {
                Shape[ ] s = {
                                new Circle( ),
                                new Square( ),
                               new Triangle( )
                };  // upcasting toward Shape
                 for(  int i=0;  i<s.length;  i++  ) {
                        s[i].draw( );  // polymorphous calling
                        s[i].erase( ); // same
                 }
         }
}
```

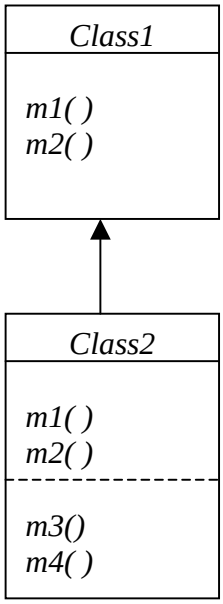
Programos rezultatai:

Circle.draw( )
Circle.erase( )
Square.draw( )
Square.erase( )
Triangle.draw( )
Triangle.erase( )

Taigi klasė *Shapes* nekreipia dėmesio į konkretų figūros tipą ir operuoja tiesiog baziniu *Shape* tipu. Šioje klasėje, kad būtų dar įtikinamiau, galima masyvą *Shapes* užpildyti figūromis ir atsitiktiniu būdu (atsitiktinių skaičių generatoriaus, kurio rezultatas yra normalizuotas *double* skaičius iš intervalo [0, 1], *Math.random( )*: *(int)(Math.random( )*3),* ir naudojant operatorių *switch*). Galima sukurti daug naujų figūrų tipų, tačiau klasės *Shapes* teksto (jei masyvo užpildymas figūromis būtų atsitiktinis) keisti nereikės. Tai ir yra pagrindinis polimorfizmo pranašumas: galima atskirti kintančią ir nekintančią programos dalį, rašant kodą tik baziniams duomenų tipams. Taip galima daug ekonomiškiau užrašyti programos tekstą.

2.12. Besileidžiantysis tipų pertvarkymas

Su tokiu duomenų tipų pertvarkymu susiduriama paveldimumo schemose, kai subklasės išplečia superklasės sąsają, o programose operuojama bendruoju superklasės tipu. Pavyzdys:



2.2 pav. Klasių schema

Dabar, siunčiant pranešimus bendrajam tipui *Class1*, vis tik reikės tiksliai žinoti tikrąjį klasės egzemplioriaus tipą: jei tai yra *Class1* egzempliorius – jis negali priimti pranešimų *m3( )* ar *m4( )*, jei *Class2* egzempliorius – gali priimti visus pranešimus *m1( ), m2( ),m3( ), m4( ).* Todėl *Java* tokiose situacijose naudoja vadinamąjį vykdymo laiko tipų nustatymo (*Run-Time Type Identification*, RTTI) mechanizmą. Jei tipas pasirodo netinkamas – generuojama tipo pervedimo išimtis *ClassCastException* (žr. skyrių, skirtą klaidų apdorojimo mechanizmui).

Klaidingos programos parodytai schemai pavyzdys:

```
class Class1{
        public void m1( ){ }
        public void m2( ){ }
}
class Class2 extends Class1{
        public void m1( ){ }  // overriding
        public void m2( ){ }  // overriding
        public void m3( ){ }
        public void m4( ){ }
}
public class Class3{
        public static void main( String args[ ] ){
                Class1[ ] c = {
                                  new Class1( ), new Class2( )
                               };
                c[0].m1( );
                c[1].m2( );
                c[1].m3( );  // compilation error !
                ( ( Class2 ) c[1] ).m3( );  // downcasting here
                ( ( Class2 ) c[0] ).m3( );  // error: ClassCastException !
        }
}
```

Pranešimas *c[1].m3( );* klaidingas, nes masyvas paskelbtas superklasės tipo – *Class1*, o toks tipas gali priimti tik pranešimus *m1( )* ir *m2( )*. Problemą galima išspręsti tik išreikštiniu būdu pervedant superklasės tipą į konkretų subklasės tipą: *((Class2) c[1]).m3( );* . Operatorius *((Class2) c[0]).m3( );* klaidingas, nes šis masyvo elementas yra *Class1* tipo.

2.13. Abstrakčiosios klasės ir metodai

2.11 skyriaus paskutiniame pavyzdyje klasės *Shape* metodai buvo fiktyvūs. Vienintelė prasmė juos pateikti – pranešti, kad visos subklasės privalo turėti tokią pat sąsają (aišku, ją galima dar praplėsti). Taigi *Shape* faktiškai yra abstrakčioji klasė, kurios tipu naudojamasi, manipuliuojant įvairiomis jos subklasėmis. Visi subklasių metodai, įeinantys į superklasės sąsają, kviečiami vėlyvojo susiejimo mechanizmu.

Abstrakčiosios klasės objektai nereikalingi. Abstrakčiosios klasės metodams reikia turėti tik sąsają, bet ne kūną, todėl tokie metodai skelbiami, pavyzdžiui, taip:

```
abstract void m( );
```

Klasė, kurioje yra bent vienas abstraktus metodas, irgi privalo būti paskelbta su modifikatoriumi *abstract*. Abstrakčiosios superklasės subklasės privalo pateikti kūnus visiems abstraktiesiems metodams. Ankstesniam pavyzdžiui klasę *Shape* galėtume perrašyti taip (visas programos *ShapesAbstract.java* tekstas yra prieduose):

```
abstract class Shape{
        abstract void draw( );
        abstract void erase( );
}
```

Abstrakčiojoje klasėje gali būti ir realūs metodai.

2.14. Sąsajų klasės

Tai dar vienas žingsnis abstraktumo link, palyginti su abstrakčiomis klasėmis: jų kūne jau negali būti jokių realių metodų; kūne gali būti tik kompiliavimo laiko (*static final*, o jei tokie modifikatoriai nenurodyti – jie pritaikomi automatiškai) konstantos. Konstrukcijos paskirtis dvejopa. Sąsajų klasės (*interfaces*) gali būti naudojamos kaip programos projektavimo bei kūrimo įrankis – nurodant klasių bendravimo protokolus. Matyt, dar svarbesnė paskirtis yra sukurti galimybę tam tikram daugybiniam paveldėjimui iš kelių bazinių tipų (visiškai daugybinio paveldėjimo galimybes realizuoja tik vidinės klasės, apie tai kalbėsime vėliau). Sintaksės pavyzdys:

```
interface Interface1{
        int i = 123;  // static final supposed
        float m1( );  // public supposed
        void m2( );  // public supposed
}
interface Interface2{
void m3( );
}
class Class1 implements Interface1, Interface2{
        public float m1( ){ . . . }   // class implementing interface
        public void m2( ){ . . . }    // gives realization of all
        public void  m3( ){ . . . }   // methods
        System.out.println( "i = " + i );  // "i" is accessible
}
. . .
```

Dabar programoje iš 2.11 skyriaus klasę *Shape* galėtume perrašyti kaip sąsajos klasę. Toks programos pavyzdys yra prieduose.

Daugybinio paveldimumo pavyzdys. *Java*, skirtingai nei C++, toks paveldėjimas bus saugus, nes sąsajų klasės paprasčiausiai neturi realizacijos, ir jokių duomenų konfliktų negali būti. Faktiškai schema tokia: paveldima iš abstrakčios ar realios klasės ir dar realizuojamos reikiamos sąsajų klasės. Esminis schemos privalumas yra tai, kad dabar įgalinamas kylantysis tipų pertvarkymas į bet kurį iš

bazinių tipų: abstrakčiosios (ar realiosios) superklasės bei visų realizuotų sąsajų klasių. Pateikiame tokios programos-schemos pavyzdį:

```
interface I1{    // 1th basic type
        void im1( );
}
interface I2{    // 2nd basic type
        void im2( );
}
interface I3{    // 3rd basic type
        void im3( );
}
class C1{
        public void im1( ){ }   //  4th basic type
}
class C2 extends C1 implements I1, I2, I3{
        // im1 inherited from class C1
        public void im2( ){ }
        public void im3( ){ }
}
public class C{
        void cm1( I1 x ) { x.im1( ); }
        void cm2( I2 x ) { x.im2( ); }
        void cm3( I3 x ) { x.im3( ); }
        void cm4( C1 x ){ x.im1( ); }
        public static void main( String args [ ] ){
                C2 cc2 = new C2( );
                cm1( cc2 );     // upcasting of type C2 toward type I1
                cm2( cc2 );     // . . . toward I2
                cm3( cc2 );     // . . . toward I3
                cm4( cc2 );     // . . . toward C1
        }
}
```

Kaip matyti, klasėje *C* kviečiant metodus konkretus tipas *C2* gali būti naudojamas vietoj bazinių tipų *I1, I2, I3, C1* – automatiškai panaudojamas kylantysis tipų pertvarkymas, taip lyg ir realizuojamas paveldėjimas iš šių tipų.

Sąsajų klasės gali paveldėti kitų sąsajų klasių charakteristikas operatoriaus *extends* dėka.

Sąsajų klasės gali būti naudojamos kompiliavimo laiko konstantoms laikyti. Joje suteikus reikšmes konstantoms net ir, pavyzdžiui, atsitiktinių skaičių generatoriumi *Math.random( ),* viso programos vykdymo metu konstantų reikšmė nekis; pradinė reikšmė suteikiama tik vienintelį kartą įkeliant sąsajų klasę. Terminą „įkelti“ anksčiau naudojome tik kalbėdami apie klases, tačiau sąsajų klasei jį lygiai taip pat galime taikyti, nes juk sąsajų klasė yra tam tikra abstrakti klasės rūšis.

Prieduose yra programa *ShapesInterface.java*, realizuojanti jau kelis kartus nagrinėtą 2.11 skyriaus pavyzdį su figūromis, šįkart perrašyta su sąsajų klasėmis. Pridėta ir viena kompiliavimo laiko konstanta, kad būtų iliustruotas tokios klasės kintamojo matomumas.

2.15. Vidinės klasės

Klasę galima įdėti į kitos klasės ar metodo vidų arba į kurią nors matomumo sritį. Tokia klasė vadinama vidine klase. Vidinė klasė gali neturėti vardo.

Pagrindinė vidinės klasės paskirtis – paslėpti klasės realizaciją nuo ją įtalpinančios klasės išorės, arba realizuoti tikrą daugybinį paveldėjimą ne nuo sąsajų klasių, o paveldėjimą nuo kelių vidinių klasių tipų.

Vidinėms klasėms galima taikyti prieities modifikatorius *public, private* ar *protected*.

Pavyzdys: klasė kitos klasės viduje:

```
public class ExternalClass{
        class InnerClass1{
                private int i = 123;
                public int valueInt( ){ return i; }
        }
        class InnerClass2{
                private String s = “abc”;
                public String valueString( ){ return s; }
        }
        public void m( ){
                InnerClass1 ic1 = new InnerClass1( );
                InnerClass2 ic2 = new InnerClass2( );
                int j = ic1.valueInt( );
                String s = ic2.valueString( );
                System.out.println( “j =  “+  j + “ s = “ + s );
        }
        public static void main( String args [ ] ){
                ExternalClass ec = new ExternalClass( );
                ec.m( );
        }
}
```

Programos rezultatai:

```
j = 123 s = abc
```

Esminis skirtumas nuo įprastinių klasių – iš už *ExternalClass* ribų vidinės klasės visiškai nepasiekiamos, rodykles į vidines klases galima būtų sukurti tik naudojant specialius metodus:

```
. . .
public InnerClass1 makeInnerClass1( ){
        return new InnerClass1( );
}
. . .
```

Dabar kitoje nei *ExternalClass* klasėje vidinių klasių objektai būtų sukuriami taip:

```
. . .
ExternalClass ec = new ExternalClass( );
ExternalClass.InnerClass1 ic1 = ec.makeInnerClass1( );
. . .
```

Užrašas *ec.makeInnerClass1( )* rodo, kad vidinių klasių objektai gali egzistuoti tik kaip išorinių klasių objektų dalis.

Kitas pavyzdys rodo kylantįjį tipų pertvarkymą su vidinėmis klasėmis:

```
public interface InnerClass1{
        int valueInt( );
}
public interface InnerClass2{
        String valueString( );
}
class ExternalClass{
        private class PInnerClass1 implements InnerClass1{
                private int i =123;
                public int valueInt( ){ return i; }
        }
        protected class PInnerClass2 implements InnerClass2{
                private String s = "abc";
                public String value String( ){ return s; }
        }
        public InnerClass1 makeInnerClass1( ){ // upcasting from
                return new PInnerClass1( );     // PInnerClass1 to  InnerClass1
        }
        public InnerClass2 makeInnerClass2(){ // upcasting from
                return new PInnerClass2( );     // PInnerClass2 to  InnerClass2
        }
}
public class PExternalClass{
        public static void main( String args [ ] ){
                ExternalClass ec = new ExternalClass( );
                InnerClass1 ic1 = ec.makeInnerClass1( );
                InnerClass2 ic2 = ec.makeInnerClass2( );
        }
}
```

Šiame pavyzdyje vidinės klasės realizuoja sąsajų klases, todėl vėliau (metoduose, grąžinančiuose vidinių klasių objektus) įmanomas kylantysis tipų pertvarkymas į sąsajų tipus. Klasės *ExternalClass* objektai suderina dviejų sąsajų klasių funkcionalumą – realizuotas jų savybių daugybinis paveldimumas. Kitoje klasėje *PExternalClass* vidinės *ExternaClass* klasės yra visiškai nematomos, todėl joje bandymas sukonstruoti objektą taip:

```
ExternalClass.PInnerClass1 ic1 = ec.new PInnerClass1( );
```

kompiliatoriaus būtų pripažintas klaidingu.

Vidinės klasės metodų viduje. Remiantis ankstesniu pavyzdžiu, jei turėtume tik vieną vidinę klasę:

```
public interface InnerClass{
        int valueInt( );
}
public class ExternalClass{
        public InnerClass  makeInnerClass( ){
                 class PInnerClass  implements InnerClass{
                        private int i =123;
                        public int valueInt( ){ return i; }
                }
                return new PInnerClass( ); // upcasting to InnerClass1
        }
        public static void main( String args [ ] ){
                ExternalClass ec = new ExternalClass( );
                InnerClass  ic1 = ec.makeInnerClass( );
        }
}
```

Čia vidinė klasė yra paslėpta dar giliau ir prieinama tik iš metodo *makeInnerClass* vidaus.

Labai panašiai vidines klases galima įdėti į bet kurią matomumo sritį; tik iš ten jos ir būtų prieinamos.

Bevardės vidinės klasės. Pavyzdys:

```
public class ExternalClass{
        public InnerClass makeInnerClass( ){
                return new InnerClass( ){  // here the inner class begins
                        private int i =123;
                        public int valueInt( ){ return i; }
                };                          //  …and here ends
}
        public static void main( String args [ ] ){
                ExternalClass ec = new ExternalClass( );
                InnerClass1 ic1 = ec.makeInnerClass( );
        }
}
```

Šiuo atveju metode *makeInnerClass* konstruojamas *InnerClass* objektas, ir iš karto parodoma, kaip tai turi būti atlikta pasinaudojant vadinamąja bevarde vidine klase. Tai labai trumpa ir aiški sintaksės forma, todėl šio tipo vidines klases plačiai naudosime kurdami grafines vartotojo sąsajas, realizuodami *Swing* paketo įvykių klausytojus (žr. skyrių „*Swing* technologija“). Tokios sąsajos yra „įvykiais valdomos sistemos“. Paketas realizuoja jų karkasą (*application framework*), o funkcionalumą šiam karkasui suteiksime pasinaudodami vidinėmis klasėmis.

Dabar pateiksime principinę schemą, rodančią, kaip naudojantis vidinėmis klasėmis galima realizuoti daugybinį paveldėjimą iš kelių ne sąsajų klasių (tokia schema daugybiniam paveldėjimui iš kelių sąsajų parodyta pirmajame šio skyriaus programos pavyzdyje):

```
class C{ . . . }
abstract class D{ . . . }

class E extends C{
        D makeD( ){
                return new D( ){ . . . };
        }
}

public class CD{
        static void takesC( C c ){ . . . }
        static void takesD( D d ){ . . . }
        public static void main( String args [ ] ){
                E e = new E( );
                takesC( e );              // upcasting to type C
                takesD( e.makeD( ) ); // upcasting to type D
        }
}
```

Čia klasė *E* naudoja klasių *C* ir *D* funkcionalumą. Kai *C* ir *D* yra klasės – panaši schema su vidine (bevarde) klase yra vienintelė galimybė daugybiniam paveldėjimui.

Paskutinės pastabos apie vidines klases. Pirma, jau sakėme, vidinės klasės objektas egzistuoja tik kaip išorinės klasės subobjektas. Todėl vidinė klasė automatiškai turi prieitį prie visų išorinės klasės elementų. Antra, kompiliuojant klases su vidinėmis klasėmis, bus sukurtos ir *class* rinkmenos visoms vidinėms klasėms: pavyzdžiui, priešpaskutinei programai bus sukurtos rinkmenos *ExternalClass.class* – išorinei klasei ir *ExternalClass$InnerClass.class* – vidinei klasei, naudojant būtent tokias taisykles klasės vardui.

2.16. Klaidų apdorojimo mechanizmas

Klaida, arba išimtis (*Java* jas vadina *exception*) – objektas, aprašantis klaidingą situaciją programoje. Šis objektas sukuriamas kilus situacijai ir perduodamas situaciją sukėlusiam metodui. Metodas, reaguodamas į šią išimtį, gali atlikti kokius nors veiksmus – apdoroti išimtį arba perduoti išimtį apdoroti aukštesniam metodui. Išimtį programoje gali sukelti ir pats programuotojas, informuodamas apie kokią nors neleistiną situaciją (pavyzdžiui, jei koks programos kintamasis išeina už algoritmo jam leidžiamų ribų).

Išimtims apdoroti reikalingi raktažodžiai *try, catch, throw, throws* ir *finally*. Raktažodis *try* figūriniais skliaustais apima operatorių bloką, kuris bus kontroliuojamas. Jei šiame bloke kyla išimtis – ją reikia „sugauti“, arba perimti – *catch*, ir apdoroti. Programos vykdymo laiko klaidas (*run-time errors*) sukelia automatiškai *Java*. Programuotojas pats savo išimtis sukelia specialiu operatoriumi *throw*. Metodas, kuriame gali kilti išimtis, bet kuris pats neapdoroja šios išimties, o nori perduoti ją apdoroti aukštesniam metodui, turi būti pažymėtas operatoriumi *throws*. Operatorius *finally* atlieka panašias funkcijas kaip *default* operatoriuje *switch*:

nurodo kodo fragmentą, kuris būtinai turi būti atliktas neatsižvelgiant į tai, kilo išimtis ar ne. Taigi principinė klaidų apdorojimo schema yra tokia:

try{ kontroliuojamas kodo blokas *}*
catch(ExceptionType1 et1) { et1 apdorojimas *}*
catch(ExceptionType2 et2) { et2 apdorojimas *}*
. . .
finally { būtinai atliekamas kodo fragmentas *}*

Visos išimtys yra klasės *Throwable* subklasės. Žemiau yra dvi jos subklasės – *Exception* ir *Error*. *Exception* klasė apima klaidingas programos situacijas, kurias turėtų sugauti ir apdoroti programuotojas. Jo kuriamos išimtys taip pat turi būti šios klasės subklasės. Svarbi *Exception* subklasė yra *RuntimeException*: ji apima tokias klaidas, kaip dalyba iš nulio, masyvo indekso išeitis už masyvo ribų ir pan. Aišku, programuotojas norėdamas gali sukelti tokio tipo išimtis ir pats. Klasės *Error* ir jos subklasių apibrėžiamų išimčių programoje perimti nereikia: tai tokios išimtys, kurių programa negali pati apdoroti – tai sisteminės ir kompiliavimo klaidos.

Jei *RuntimeException* išimties programuotojas neperima, ją automatiškai perima *Java* klaidų apdorojimo mechanizmas: išvedamas pranešimas apie klaidą, metodų kvietimų seka, ir programos darbas nutraukiamas. Seka išvardija visą metodų ir jų klasių vardų seką nuo išimtį sukėlusio metodo iki aukščiausio metodo ir atrodo taip:

at KlasėsVardas, MetodoVardas (RinkmenosVardas.*java*: eilutės numeris)
at KlasėsVardas,AukštesniojoMetodoVardas (RinkmenosVardas.*java*: eilutės numeris)
. . .

2.16.1. Operatoriai *try* ir *catch*

Operatoriai reikalingi, kai programuotojas pats bando perimti *Java* ar savo paties sukurtą išimtį. Pavyzdžiui, dalybos iš nulio perėmimas:

```
class MyException{
        public static void main( String args [ ] ){
                int numerator, denominator, result;
                try{
                        numerator = 123;
                        denominator = 0;
                        result  = numerator/denominator;
                        System.out.println( “This won’t be printed” );
                } catch( ArithmeticException ae ){
                        System.out.println( “Processing of exception” );
                 }
                System.out.println( “After catch of exception” );
        }
}
```

Programos spausdinimai:

Processing of exception
After catch of exception

Taigi, programoje perėmus išimtį, atliekamas to *catch* operatoriaus, kurio argumento tipas sutampa su kylančios išimties tipu, srityje esantis kodo fragmentas, o toliau – vykdomi žemiau *try* srities esantys operatoriai. Taip galima parašyti niekada avariškai nesibaigiančias programas.

Klasė *Throwable* turi modifikuotą metodą *toString*, paveldimą iš *Object* klasės. Kiekviena *Java* išimtis turi su ja asocijuotą paaiškinamąjį tekstą, kurį galima išvesti šiuo metodu (metodo išreikštai kviesti nereikia), pavyzdžiui:

```
. . .
} catch( ArithmeticException ae ){
        System.out.println( “Exception” + ae );
. . .
```

Kai programa po kontroliuojamo bloko bando perimti kelias skirtingas išimtis – reikalingi keli skirtingi operatoriai *catch*. Jie turi būti išdėstomi tam tikra tvarka: pirmiau reikia perimti išimtį-subklasę, o vėliau – išimtį-superklasę. Antraip dalis programos kodo niekada nebūtų vykdoma, o tai jau sintaksės klaida. Pavyzdžiui:

```
. . .
try{
        . . .
} catch ( Exception e ){
        System.out.println( “Exception” + e );
} catch ( ArithmeticException ae ){
        System.out.println( “Exception” + ae );
}
. . .
```

ArithmeticException yra *Exception* subklasė, todėl visada bus perimta pirmoji išimtis – *Exception*, o antrasis blokas niekada nebūtų vykdomas. Todėl toks programos fragmentas negalėtų būti sukompiliuotas.

Minėtieji operatoriai gali būti ir kartotiniai: vieno *try* bloko viduje gali būti kitas *try* blokas ir t. t.

2.16.2. Operatorius *throw*

Sukelia programuotojo išimtį naudojant tokią sintaksę:

. . . throw ThrowableInstance;

čia *ThrowableInstance* yra arba vidinė *Java* išimtis – viena kuri nors *Throwable* klasės subklasė, arba paties programuotojo sukurta išimtis (taip pat turi paveldėti *Throwable* savybes). Šis objektas sukuriamas įprastai, operatoriumi *new*. Visų vidinių *Java* išimčių konstruktoriai yra dviejų tipų: be argumento arba su *String* tipo

argumentu. Pastaruoju galima objektui suteikti paaiškinamąjį tekstą, kurį vėliau spausdintume metodu *toString*.

Įvykdžius operatorių, programos vykdymas pertraukiamas ir valdymas atiduodamas atitinkamam *catch* blokui. Jei toks nerandamas – programą stabdo *Java* klaidų apdorojimo mechanizmas. Pavyzdys-schema: sukeliama vidinė *Java* klaida *NullPointerException* – tam operatoriumi *new* sukuriamas jos objektas; ji sugaunama, paskui sukeliama nauja tokio pat tipo klaida, o jos apdorojimas atiduodamas aukštesniam metodui:

```
public class MyException2{
        static void throwException( ){
                try{
                        throw new NullPointerException( “Example” );
                } catch( NullPointerException npe ){
                        System.out.println( “Caugth inside throwException” );
                        System.out.println( “Exception type: “ + npe );
                        throw npe;
                }
        }
        public static void main( String args [ ] ){
                try{
                        throwException( );
                } catch( NullPointerException npe ){
                        System.out.println( “Caugth repeatedly inside main” );
                }
        }
}
```

Programos spausdinimai:

```
Caugth inside throwException
Exception type: Example
Caugth repeatedly inside main
```

2.16.3. Operatorius *throws*

Tai raktažodis metodui, kuris gali sukelti išimtį, tačiau pats jos neapdoros – tada klaidą privalo apdoroti aukštesnis kviečiantysis metodas (aišku, šis gali peradresuoti apdorojimą dar aukštesniam metodui ir t. t.). Po operatoriaus *throws* galima išvardyti ir kelias išimtis – tada visos jos turi būti perimtos aukštesniame metode. Nereikia apdoroti tik *Error* ir *RuntimeException* išimčių bei jų subišimčių.

Pavyzdys:

```
public class MyException3{
        static void throwException( ) throws Exception{
                System.out.println( “Inside throwException” );
                throw new Exception( “Example” );
        }
        public static void main( String args [ ] ) {
```

```
            try{
                throwException( );
            } catch( Exception e ){
                System.out.println( “Exception “ + e + “ caught” );
            }
        }
}
```

2.16.4. Operatorius *finally*

Šio operatoriaus blokas rašomas po visų *catch* blokų ir vykdomas bet kokiu atveju – kilo išimtis ar ne. Pavyzdžiu galėtų būti duomenų skaitymas iš rinkmenos: skaitant duomenis gali kilti vidinė *Java* išimtis *IOException* – tokiu atveju ją reikia perimti, o po jos apdorojimo – uždaryti rinkmeną; jei duomenys nuskaityti tvarkingai be klaidų – vis tiek po visko rinkmeną reikia uždaryti. Beje, jei yra operatorius *finally*, *Java* kompiliatorius nereikalauja po *try* bloko *catch* operatorių.

Tas pat pavyzdys su šiuo operatoriumi:

```
public class MyException4{
    static void throwException( ) throws Exception{
        System.out.println( “Inside throwException” );
        throw new Exception( “Example” );
    }
    public static void main( String args [ ] ) {
        try{
            throwException( );
        } finally{
            System.out.println( “An exception  caught” );
        }
    }
}
```

2.16.5. Kai kurios vidinės *Java* išimtys

Java vidinės išimtys dar skirstomos į kontroliuojamas ir nekontroliuojamas išimtis. Minėta, kad *RuntimeException* išimtys gali būti programoje neperimamos – todėl jos vadinamos nekontroliuojamomis. Kai kurios iš jų (kada jos kyla – gerai nusako vien jų išsamūs pavadinimai) yra:

ArithmeticException
ArrayIndexOutOfBoundsException
ClassCastException
NullPointerException

Kontroliuojamos išimtys turi būti perimamos arba įtraukiamos į *throws* sąrašą. Jų pavyzdžiai:

ClassNotFoundException

IllegalAccessException
InstantiationException – kai bandoma sukurti *abstract* klasės objektą
InterruptedException – kai bandoma pertraukti vykdomą srautą (žr. tolesnius skyrius)
NoSuchMethodException
NoSuchFieldException

Kuriant savo išimtį, reikia plėsti klasę *Exception*. Ši klasė neturi jokių savo metodų; juos paveldi iš *Throwable*: *String toString( ), String getMessage( )* ir t. t. Šiuos metodus galima perrašyti.

Savos išimties kūrimo pavyzdys. Tarkime, programa turi kviesti skaičiavimo metodą *compute* su nurodyta *int* tipo argumento reikšme. Jei ta reikšmė viršija 10, reikia sužadinti išimtį. Kartu perrašomas ir metodas *toString*, kad būtų išvedamas sugeneruotos išimties pavadinimas ir metodui *compute* atiduota argumento reikšmė.

```
class MyException5 extends Exception{
        private int arg;
        MyException5( int i ){
                arg = i;
        }
        public String toString( ){
                return "MyException5( " + arg + " )";
        }
}

public class ExceptionExample{
        static void compute( int i ) throws MyException5{
                System.out.println( "Method compute( " + i + " )";
                if( i > 10 ) throw new MyException5( i );
                System.out.println( "Regular finish of method  compute" );
        }
        public static void main( String args [ ] ) {
                try{
                        compute( 5 );
                        compute( 15 );
                } catch( MyException5 e ){
                        System.out.println( "An exception  caught" + e );
                }
        }
}
```

Programos spausdiniai:

```
Method compute( 5 )
Regular finish of method compute
Method compute( 15 )
An exception caught  MyException5( 15 )
```

2.17. Įvesties ir išvesties sistemos pagrindai

Diduma sistemos realizuota pakete *java.io*.

Java įvesties ir išvesties sistema funkcionuoja vadinamųjų srautų pagrindu. Tai tam tikra programinė abstrakcija: sukuriamas srauto objektas, susiejamas su kokiu nors fiziniu kompiuterio įrenginiu ir specialiais *Java* metodais srautams (pavyzdžiui, jau mums žinomu metodu *println*) duomenys įvedami iš srauto ar nukreipiami į srautą.

Yra baitiniai ir simboliniai (*Unicode* simbolių) srautai. Pastaraisiais papildyta tik *Java 2*.

Šiame kurse susidursime su tokiais baitiniais srautais:

InputStream *OutputStream*	– abstrakčiosios srautų klasės įvesčiai ir išvesčiai
BufferedInputStream *BufferedOutputStream*	– srautai, naudojantys atminties buferį, kurio talpa gali būti nustatyta pagal nutylėjimą arba išreikštai paties programuotojo. Daug efektyvesni nei nebuferizuoti.
DataInputStream *DataOutputStream*	– primityvių duomenų ir *String* duomenų srautai
FileInputStream *FileOutputStream*	– srautai iš rinkmenos ir į rinkmeną
PrintStream	– srautas metodams *print( )* ir *println( )* laikyti

Atitinkami simboliniai srautai yra:

Reader *Writer*	– abstrakčiosios srautų klasės
BufferedReader *BufferedWriter*	
StringReader *StringWriter*	– tik iš dalies atitinka *DataInputStream* bei *DataOutputStream*: skirtas tik *String* duomenims
FileReader *FileWriter*	
PrintWriter	

Kadangi *Java 2* rekomenduojami simboliniai srautai, yra ir specialios klasės, pervedančios baitinius srautus į simbolinius: *InputStreamReader* ir *OutputStreamWriter*.

Greta šių srautų palaikoma ir vadinamoji standartinė įvestis ir išvestis. Tokia sąvoka reiškia: visi duomenys programai įvedami iš vienintelio standartinės įvesties srauto ir išvedami į vienintelį standartinį išvesties srautą, o visos klaidos užrašomos į vienintelį standartinį klaidų srautą. Šie standartiniai srautai realizuoti ne pakete *java.io,* bet *java.lang.System* klasėje. Šioje klasėje yra *public static* laukai:

in – srautas standartinei įvesčiai,
out – srautas standartinei išvesčiai,
err – srautas standartiniam klaidų srautui.

Laukai *out* ir *err* yra klasės *PrintStream* objektai; jie susieti su pultu. Laukas *in* yra *InputStream* objektas ir taip pat susietas su pultu. Šiuos baitinius srautus galima pertvarkyti į simbolinius srautų-„tiltų“ *InputStreamReader* ir *OutputStreamWriter* pagalba. Šiuos srautus galima nukreipti ir ne į pultą klasės *System* metodais *setIn(InputStream), setOut(PrintStream)* bei *setErr(PrintStream).*

Įvesties ir išvesties sistema yra klaidi, todėl apsiribosime išnagrinėdami kai kuriuos standartiškai reikalingus atvejus.

1. Įvestis iš pulto. Rekomenduojama ne baitinė, bet simbolinė įvestis, ir dar buferizuota, t. y. reikia operuoti ne vienu simboliu, bet keliais iškart – taip efektyviau. Minėta, kad pulto įvestis – įvestis iš objekto *System.in*; šis objektas yra baitinio srauto *InputStream* tipo. Šį baitinį srautą reikia pertvarkyti į simbolinį tiltu *InputStreamReader*. Viską sudėjus kartu, turime:

```
BufferedReader br =  new BufferedReader(
                                new InputStreamReader(
                                        System.in ) );
```

Klasėje *BufferedReader* yra metodai:

int read( ) throws IOException – vienam simboliui įvesti; jis grąžinamas kaip atitinkama *int* reikšmė. Kai randama srauto pabaiga, grąžinama reikšmė *–1*;
String readLine( ) throws IOException – vienai eilutei įvesti. Suradus srauto pabaigą, grąžinama reikšmė *null*.

Pavyzdys: simbolių įvedimas iš klaviatūros, kol nebus įvestas simbolis *q*:

```
import java.io.*;
class CharReading{
        public static void main( String args [ ] ) throws IOException{
                        // “throws” transfers possible exception of method
                        // “read”.Now the “try” block is not needed
                char c;
                BufferedReader br =  new BufferedReader(
                                new InputStreamReader(
                                        System.in ) );
```

```
                do{
                        c =( char )br.read( );
                        System.out.println( c );
                } while( c != 'q' );
        }
}
```

Įvedus, pavyzdžiui, *abcdq* ir nuspaudus įvedimo klavišą, programa spausdins:

```
a
b
c
d
q
```

Taip yra todėl, kad įvedimas buferizuotas – į buferį nuskaitomi iškart visi simboliai, o iš buferio skaitoma jau po simbolį. Nebuferizuoto įvedimo tokiu atveju būtų įvestas tik paskutinis simbolis.

2. Išvestis į pultą. Paprasčiausia naudoti klasės *System* metodus *out.print* arba *out.println*. Rekomenduojama naudoti simbolinius srautus – klasę *PrintWriter*. Jos objektai sukuriami keliais alternatyviais konstruktoriais. Vienas iš jų yra: *PrintWriter(OutputStream os, boolean b)*, t. y. simbolinis srautas gaunamas iš baitinio srauto. Taigi vietoje šio argumento galima teikti *System.out* lauką, kadangi šis yra *PrintStream* tipo, o pastarasis paveldi iš *OutputStream* klasės. Jei *b* yra *true*, srautas išvedamas iš buferio diske automatiškai, kai surandama eilutės pabaiga (simboliai \n). Antraip galimas atvejis, kai nieko nebus išvesta: kol buferis nebus pripildytas, duomenys iš jo nebus išvedami. Taip gali nutikti mūsų programose, išvedant nedidelius kiekius duomenų. Viską sudėjus, turime:

```
PrintWriter pw = new PrintWriter( System.out, true );
```

Klasė *PrintWriter* turi metodus *print( )* ir *println( )*, kurie prireikus pasitelkia metodą *toString( )*.

Pavyzdys: perrašysime ankstesnę programą:

```
import java.io.*;
class CharReading{
        public static void main( String args [ ] ) throws IOException{
                char c;
                BufferedReader br =  new BufferedReader(
                                new InputStreamReader(
                                        System.in ) );
                PrintWriter pw = new PrintWriter( System.out, true );
                do{
                        c =( char )br.read( );
                        pw.println( c );
                } while( c != 'q' );
        }
}
```

3. Duomenų mainai su rinkmenomis. Visos *Java* rinkmenos yra baitinės. Srautams iš rinkmenų savo mašinoje atidaryti paprastai naudojami konstruktoriai

FileInputStream(String fileName) throws FileNotFoundException;
FileOutputStream(String fileName) throws FileNotFoundException;

Atidarius įrašymo rinkmeną, ankstesnė tuo pat vardu buvusi rinkmena sugriaunama. Jei rinkmenos nebuvo – sukuriama nauja. Taigi lieka neaišku, kodėl konstruktorius generuoja nurodytą išimtį. Po rinkmenos įrašymo ar nuskaitymo ją reikia uždaryti. Mums bus reikalingi tokie šių srautų metodai:

```
int read( ) throws IOException;
void write( int i ) throws IOException;
void close( ) throws IOException;
```

Skaitant rinkmeną minėtoji išimtis kyla, kai randama rinkmenos pabaiga (EOF, grąžinama reikšmė *–1*). Rekomenduojami simboliniai buferizuoti srautai, kurie turi metodus *readLine( ),* kelis perkrautus *write( )* metodus, ir kuriuos gautume jau įprastu būdu:

```
BufferedReader br = new BufferedReader(
                                new InputStreamReader(
new FileInputStream( “file.in” )));

PrintWriter pw = new PrintWriter(
                        new BufferedWriter(
                                new FileWriter( “file.out” )));
```

Pavyzdys: rinkmenos kopijavimas į kitą rinkmeną; rinkmenų vardai programai teikiami komandinėje eilutėje:

```
import java.io.*;
class CopyingFile{
        public static void main( String args [ ] ) throws IOException{
                        // transfers exceptions of “write”, “read”, “close”
                String line;
                BufferedReader in;
                PrintWriter out
                try{
                        in = new BufferedReader(
                                new InputStreamReader(
new FileInputStream( args[0]  )));
                        out = new PrintWriter(
                                new BufferedWriter(
 new FileWriter( args[1]  )));
                } catch( FileNotFoundException e ) {
                         // exception may arise when  searching for source file
                        System.out.println( “Source file does not exist” );
```

```
                return;
            } catch( ArrayIndexOutOfBoundsException e ){
                // exception may arise when the program is
                // wrongly  launched from command line
                System.out.println( “To be used:” );
                System.out.println( “java CopyingFile source target” );
                return;
            }
            do{
                line = in.readLine( );
                if( line != null ) {
                    out.write( line+”\n” );  // here the overridden
                }                            // write( String ) is used.
            } while(line != null );          // \n breaks the line.
            in.close( );
            out.close( );
        }
    }
```

Programa turi būti paleista iš komandinės eilutės taip (darbiniame aplanke jau turi būti įrašyta eilutinių duomenų rinkmena *sourceFile*):

java CopyingFile sourceFile targetFile

Darbiniame aplanke programa sukuria rinkmeną *targetFile*.

4. Sisteminės įvesties ir išvesties peradresavimas metodais *setIn, setOut, setErr*.

Sisteminės išvesties ir sisteminio klaidų srauto nukreipimas į rinkmeną (tegu rinkmenos vardas yra *file.out*):

```
. . .
PrintStream out;
try{
    out = new PrintStream(
        new BufferedOutputStream(
            new FileOutputStream( “file.out”  )));
    System.setOut( out );
    System.setErr( out );
. . .
    System.out.println( “This message goes to the file file.out” );
    System.err.println( “This error-message goes to the file file.out, too” );
} catch( Exception e ) {
    System.err.println( “Error “ + e );
}
. . .
out.close( );
. . .
```

Šiame pavyzdyje tenka naudoti srautą *PrintStream*, nes klasės *System* laukas *out*, kurį ir norime peradresuoti, yra *OutputStream* tipo; o *PrintStream* paveldi *OutputStream* funkcionalumą. Toliau dar pridedamas buferis skaitymo iš rinkmenos srautui *FileOutputStream*. Viską sudėjus gaunamas reikiamas programos objektas *out*.

Sisteminės įvesties peradresavimas iš rinkmenos, tarkime, *file.in*, būtų toks (dėl fragmento trumpumo praleisime *try* bloką):

```
. . .
BufferedInputStream in = new BufferedInputStream(
                                    new FileInputStream( "file.in" ));
System.setIn( in );
BufferedReader br = new BufferedReader(
                                new InputStreamReader( System.in ));
. . .
String line = br.readLine( );
. . .
in.close( );
. . .
```

Iš programos matyti, kad sisteminės įvesties peradresavimas iš rinkmenos programos teksto nesutrumpina ir nėra labai naudingas.

Visa tai – tik įvadas į sudėtingą *Java* įvesties ir išvesties sistemą. Galbūt, norint realizuoti kai kurias idėjas, teks savarankiškai išnagrinėti ir kitus srautus:

ZipInputStream, ZipOutputStream – srautai su duomenų automatiniu suspaudimu į *zip* formatą;

PipedInputStream, PipedOutputStream ir analogiški simboliniai srautai *PipedReader, PipedWriter* – srautai bendrauti tarp kelių programos vykdymo gijų (žr. tolesnius skyrius);

RandomAccessFile – visiškai savarankiška klasė šio tipo rinkmenoms ir t. t.

2.18. Savianalizės sistema

Daugelis anksčiau nagrinėtų klasių yra pakankamai sudėtingos. Kad jomis būtų galima sėkmingai naudotis, reikia žinoti jų superklasę, sąsajų klases, konstruktorius, metodus (pačioje klasėje skelbiamus metodus ir metodus, paveldimus iš visų superklasių iki pat *Object* klasės), laukus. Visus šiuos dalykus galima sužinoti naudojantis *Java* savianalizės sistema, kurią įgyvendina paketas *java.lang.reflect* ir jo klasė *Class*.

Klasė *Class* saugo informaciją apie visą objekto būklę programos vykdymo (ne kompiliavimo!) metu. *Class* objektai automatiškai sukuriami įkeliant klases; jų negalima sukurti įprastiniu būdu operatoriumi *new*. Rodyklę į *Class* objektą galima gauti statiniu *Class* metodu:

```
static Class forName( String name ) throws ClassNotFoundException
```

Pavyzdžiui,

```
Class c = Class.forName( “java.lang.String” );
```

Kiekvienam egzistuojančiam bet kurios klasės objektui rodyklę į jo Class objektą taip pat galima gauti metodu *getClass( )*:

```
. . .
ClassX cx  = new ClassX( );
Class c  =  cx.getClass( );
. . .
```

Turint reikiamą rodyklę, metodais *getConstructors( ), getMethods( ),* ... gaunami visi klasei prieinami konstruktoriai, metodai ir kita informacija.

Šis savianalizės mechanizmas reikalingas *JavaBeans* technologijoje (žr. skyrių apie *Swing* technologiją). Čia jį panaudosime kurdami įrankį, kuris sužinos ir į pultą bei rinkmeną *List.out* tame pat aplanke, kaip ir programos kodo rinkmena, išves klasės konstruktorių ir metodų sąrašus. Be tokių sąrašų naują klasę ir jos galimybes išnagrinėti labai keblu. Kadangi pilnuose metodų ir konstruktorių varduose dažnai pasikartoja pradinė vardo dalis *java.lang*, ją iš visų vardų pašalinsime klasės *String* metodais. Duomenys apie norimą ištyrinėti klasę įrankiui perduodami komandinės eilutės pirmuoju argumentu (reikia pilnojo klasės vardo), o antruoju argumentu, jei pageidaujama, perduosime įrankiui informaciją apie būtinai varde turinčią būti subeilutę (pavyzdžiui, jei norime sužinoti tik sąrašą metodų išvesčiai, galime įtarti, kad varde turės būti žodis *write* – tokį *String* duomenį ir teiksime antruoju komandinės eilutės argumentu).

```
//
// Utility: list of methods for the indicated class
//        [ possessing indicated substring ]
//        to the screen    and
//        to the file "List.out" in the same directory
//


import java.lang.reflect.*; //for class Class
import java.io.*;

public class ML{

  static final String usage =
    "Use: java ML packageName.ClassName     or\n"+
    "     java ML packageName.ClassName subnameOfMethod";
  static String [] ls;  //array for the list
  static int count = 0; //counts the methods found

  static String removeName( String s ) { //removes "java.lang." from
    int sl    = s.length();                    //the name of method
    int first = s.indexOf( "java.lang." );  //see class String
    if( first == -1 ) return s;
      int last  = first + 10;
      return removeName(
        s.substring( 0,first ) + s.substring( last,sl ) );  //see class String
```

```
 }

 public static void main( String args[]){
  if( args.length < 1 ) {
   System.out.println( usage );
   System.exit( 1 );   //erroneous exit; see class System
  }
  try {
   Class c           = Class.forName( args[0] );
   Method [] m       = c.getMethods();   //see class Class
   Constructor [] cr = c.getConstructors(); //same
   ls = new String[ m.length+cr.length ];

   if( args.length == 1 ) { //ie, full list of methods and constructors
     for ( int i = 0; i < m.length; i++ ) //for class given in args[0]
        ls[count++] = "method  "+ removeName( m[i].toString() );
     for ( int i = 0; i < cr.length; i++ )
        ls[count++]= "constructor  "+ removeName( cr[i].toString() );
   }else { //ie, length==2; list of methods containing substring given
     count = 0;            //in args[1]
     for ( int i = 0; i < m.length; i++ ) {
       String ms = m[i].toString();
       if( ms.indexOf( args[1] )!=-1 )
          ls[count++]= "method  "+ removeName( ms );
     }
     for ( int i = 0; i < cr.length; i++ ) {
       String ms =  cr[i].toString();
       if( ms.indexOf( args[1] )!=-1 )
          ls[count++]= "constructor  "+ removeName( ms );
     }
   }
  } catch( ClassNotFoundException e ) {
    System.err.println( "Class "+e+" does not exist" ); //errors to the screen
   }
  try{  //output to the screen and file
   PrintWriter out = new PrintWriter(
                              new BufferedWriter(
                                 new FileWriter(  "List.out" ))),
    con = new PrintWriter( System.out, true ); //true: empty
   for( int i=0; i<count; i++)  {                  //the buffer
     out.println( (i+1) +"  "+ls[i] );
     con.println( (i+1) +"  "+ls[i] );
   }
   out.close();
  } catch( Exception e ){
    System.err.println( e.toString() ); //errors to the screen
   }
 }
}
```

Pastabos:

1. Metodas *removeName( )* iš pilnojo konstruktoriaus ar metodo vardo išmeta subeilutę *java.lang*. Tam pasitelkiami klasės *String* metodai *s.length( )* – jie nustato *String* duomens *s* ilgį , *s.indexOf (String sub)* – nustato subeilutės *sub* pirmojo įėjimo į duomenį *s* poziciją, o jei įėjimo nėra – grąžina *–1*; *s.substring(int i1, int i2)* – grąžina *s* subeilutę pradedant nuo simbolio pozicijoje *i1* ir baigiant simboliu pozicijoje *i2*. Kadangi į vieną metodo ar konstruktoriaus aprašą *java.lang.* gali įeiti kelis kartus, metodas *removeName* kviečiamas rekursyviai.

2. Jei įrankis *ML* kviečiamas neteisingai, į pultą išvedamas pranešimas apie tai, kaip naudoti įrankį (*String usage*), ir programa stabdoma klasės *System* metodu *exit*, kuriam teikiama *int* reikšmė. Įprasta, kad programai normaliai baigiant darbą *exit* reikšmė turėtų būti *0*, kitais atvejais – kita, pavyzdžiui, kaip čia – *1*.

3. Metodai *getMethods* ar *getConstructors* grąžina klasių *Method* ir *Constructor* objektus; juos tenka išreikštai pervesti į *String* tipą metodu *toString*, kad būtų galima analizuoti vardus.

2.19. Vykdymo srautai

Java leidžia vienu metu vykdyti kelias programos dalis. Tai efektyvu, pavyzdžiui: atliekant įvedimo ir išvedimo operacijas (jas atliekantys įrenginiai lėti) ir kitas programos operacijas vienu metu; arba grafinėje vartotojo sąsajoje besikartojančius uždavinius (slenkantis vaizdas, besisukantis paveikslėlis ir pan.) paleidžiant atskiru nuo pagrindinio sąsajos srauto srautu. Visais šiais atvejais pilniau išnaudojamas kompiuterio CPU.

Keli *Java* srautai (arba gijos, *threads*) paleidžiami vykdyti vienoje programoje (kitaip – viename procese) ir naudoja tą pačią adresų erdvę ir tuos pačius atminties išteklius. Tuo *Java* srautai skiriasi, pavyzdžiui, nuo C++ srautų, kuriamų funkcija *fork*. Pastarieji yra pilnos pradinio srauto kopijos – atskiri procesai. Perjungimas iš vieno srauto į kitą reikalauja tam tikrų išteklių (šimtų operacijų eilės), tačiau tai daug efektyviau nei perjungimai tarp procesų, pavyzdžiui, daugiaprogramėje OS, reikalaujantys tūkstančių operacijų eilės.

Programoje gyvuojant keliems srautams, galimos situacijos, kai keli srautai konkuruoja dėl to paties kompiuterio išteklio: pavyzdžiui, keli srautai bando vienu metu rašyti duomenis į rinkmeną. Todėl, norint sutvarkyti kelių srautų darbą, gali tekti suderinti jų darbą – sinchronizuoti srautus.

Srautai gali būti grupuojami į grupes. Galima iškart valdyti vienos grupės visus srautus.

Visos *Java* bibliotekos sukurtos įvertinant galimą daugiasrautiškumą. Matyt, daugiasrautiškumo modelis vis tik nėra tobulas, nes daugelis autorių programavimą naudojant kelis srautus vadina menu. Visų pirma, yra ribos, kai didesnis srautų kiekis sumažina programos efektyvumą (ta riba siekia šimtus ir tūkstančius srautų ir priklauso nuo kompiuterio pajėgumo). Antra, pauzės srauto vykdymo metu (gaunamos metodu *sleep( )*) arba savanoriškas valdymo atsisakymas sraute (kviečiant metodą *yield( )*) gali taip pat padidinti programos efektyvumą. Taigi automatinis perjungimo tarp srautų mechanizmas nėra tobulas.

Srauto būklė gali būti tokia:

1. Pasiruošęs darbui (sukurtas, bet nepaleistas).
2. Vykdomas (tiksliau, gali būti vykdomas, jei *Java* JVM srautui skirs CPU išteklių).

3. Pasibaigęs (baigė darbą srauto metodas *run( )*).
4. Blokuotas (pavyzdžiui, laukia išteklio, arba pats srautas iškvietė metodą *sleep( )* – tada jo eilė srautų eilėje praleidžiama ir vykdomas kitas srautas).

Vienas programos srautas yra pagrindinis: jame kuriami kiti srautai, jis turi baigtis pats paskutinis. Jei pagrindinis srautas greičiau baigiasi nei kuris dukterinis srautas – dokumentacijoje nurodoma, kad JVM gali „pakibti“. To išvengti padeda metodas *join( ),* susiejantis pagrindinį srautą su dukteriniu ir leidžiantis pagrindiniam srautui palaukti, kol baigs darbą dukterinis. Srautai metodu *setDaemon(true)* gali būti paskelbti srautais-demonais ir dirbti foniniu režimu. Dabar programa baigiasi, kai baigiasi pagrindinis srautas.

Dabar pateikiama srauto sukūrimo sintaksė, o vėliau – srautų panaudojimo savarankiškose programose ir klientinėse programose pavyzdžiai. Todėl dalis šių pavyzdžių dabar iki galo dar nebus suprantami, prie jų teks grįžti išnagrinėjus *Swing* paketą.

2.19.1. Srauto sintaksė

Srautas saugomas klasės *Thread* objekte. Ši klasė realizuoja sąsajų klasę *Runnable*. Todėl srautą galima suformuoti dviem būdais – plečiant klasę *Thread* arba realizuojant sąsajų klasę *Runnable.* Pagrindiniai *Thread* metodai yra šie:

start( ) – paleidžia srautą;
run( ) – nurodo, ką reikia atlikti sraute. Jam pasibaigus, baigiasi srauto gyvavimas;
sleep(t) – blokuoja srautą laikui *t*, matuojamam milisekundėmis;
yield( ) – atiduoda valdymą šį srautą sukūrusiam srautui – supersrautui;
join ( ) – susieja šį srautą su pagrindiniu;
isAlive( ) – nustato, ar šis srautas dar gyvuoja;
getPriority( ) – nustato srauto prioritetą;
getName( ) – nustato srauto vardą.

Net jei programa nesukuria jokio išreikšto srauto, vienas – pagrindinis srautas jai sukuriamas automatiškai. Vardą ir prioritetą jam suteikia JVM. Visada galima gauti rodyklę į šį srautą statiniu metodu *currentThread( )*:

```
class OneThread{
        public static void main( String args [ ] ){
                Thread t = Thread.currentThread( );
                t.setName( “Thread1” );
                System.out.println( “Current thread: “ + t );
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( i );
                                t.sleep( 1000 );
                        }
                } catch( InterruptedException ie ) { }
        }
}
```

Programa spausdins:

Current thread: Thread[Thread1, 5, main]

Tokią spausdinimo formą nustato automatiškai objektui *t* kviečiamas metodas *toString( )*, o paskui kas sekundę spausdinami cikle nurodyti skaičiai, nes kiekvienas ciklo žingsnis tokiam laikui pertraukiamas. Kaip matyti, srauto prioritetas pagal nutylėjimą yra *5* (prioritetų ribos yra 1–10), o srautas automatiškai įjungiamas į srautų grupę *main*. Sustabdžius srautą, gali kilti išimtis *InterruptedException,* jei šį srautą pauzės metu pertrauktų kitas srautas metodu *interrupt( ).* Teigiama, kad tokia galimybė bus naudinga ir bus naudojama ateities *Java* versijose. Dabar tenka tik perimti išimtį.

Pirmasis būdas sukurti dukterinį srautą – *Thread* klasės plėtimas.

```
class NewThread extends Thread{
        NewThread( ){
                super( “Child-thread” );
                System.out.println( “Child- thread: “  + this );
                start( );
        }
        public void run( ){
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( ”Child-thread: ” + i );
                                this.sleep( 500 );  // or Thread.sleep( 500 );
                        }
                } catch( InterruptedException ie ) { }
                System.out.println( “End of child-thread” );
        }
}
class TwoThreads1{
        public static void main( String args [ ] ){
                new NewThread( );
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( ”Main thread: + i );
                                Thread.sleep( 1000 );
                        }
                } catch( InterruptedException ie ) { }
                System.out.println( “End of main thread” );
        }
}
```

Čia formuojant dukterinį srautą, pirmiausia iškviečiamas *Thread* konstruktorius (viena iš perkrautų jo versijų priima vieną argumentą – *String* duomenį – srauto vardą). Rekomenduojama konstruktoriuje srautą ir paleisti. Programos darbo rezultatai priklauso nuo kompiuterio pajėgumo ir abiejų srautų pauzių trukmės.

Antrasis būdas srautui sukurti – sąsajų klasės *Runnable,* kurioje yra tik vienas metodas *run( ),* realizavimas. Šioje programoje panaudota kita konstruktoriaus versija su dviem argumentais – *Runnable* ir *String* tipų.

```
class NewThread implements Runnable{
        Thread t;
        NewThread( ){
                t = new Thread( this, “Child-thread” );
                System.out.println( “Child-thread: “ + t );
                t.start( );
        }
        public void run( ){
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( “Child-thread: “ + i );
                                Thread.sleep( 500 );
                        }
                } catch( InterruptedException ie ){ }
                System.out.println( “End of child-thread” );
        }
}

class TwoThreads2{
        public static void main( String args[ ] ) {
                new NewThread( );
                        try{
                                for( int i=5; i>0; i-- ){
                                        System.out.println( “Main thread: “+ i
);
                                        Thread.sleep( 1000 );
                                }
                        } catch( InterruptedException ie ){ }
                        System.out.println( “End of main thread” );
        }
}
```

Ši programa parodo, kaip reikia sujungti kelis srautus metodu *join( )*, kad JVM niekada „nepakibtų“, pagrindiniam srautui baigus darbą pirmiau nei dukteriniam.

```
class NewThread implements Runnable{
        Thread t;
        NewThread( ){
                t = new Thread( this,"Child-thread" );
                System.out.println( "Child-thread: "+ t );
                t.start( );
        }
        public void run( ){
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( "Child-thread: "+ i );
                                Thread.sleep( 5000 );
                        }
                } catch( InterruptedException ie ){ }
```

```
                System.out.println( "End of child-thread" );
        }
}

class TwoThreads21{
        public static void main( String args[] ) {
                NewThread nt = new NewThread( );
                try{
                        for( int i=5; i>0; i-- ){
                                System.out.println( "Main thread: "+ i  );
                                Thread.sleep( 1000 );
                        }
                } catch( InterruptedException ie ){ }
                try{
                        nt.t.join( ); //now main thread waits for finish
                } catch( InterruptedException ie ){ } // of thread "nt.t"
                System.out.println( "End of main thread" );
        }
}
```

Pavyzdys, iliustruojantis prioritetų įtaką srautų vykdymui. Rekomenduojama sraute daryti pauzes arba iš srauto kai kuriais momentais savanoriškai atiduoti valdymą supersrautui – tada ir kiti srautai turės šansą gauti valdymą. Kartais, aišku, pauzės kyla natūraliai, pavyzdžiui, kai srautas blokuojamas laukiant įvesties ar išvesties operacijos. Kai kada aukštesnio prioriteto srautas priverstinai blokuoja žemesnio prioriteto srautą.

Yra apibrėžtos *static final* konstantos *MIN_PRIORITY, NORM_PRIORITY* ir *MAX_PRIORITY*, nustatančios žemiausią, normalų ir aukščiausią prioritetus.

Programa paleidžia du srautus, kurie kiekvienas augina savo skaitiklį – *int* duomenį. Kartu čia paaiškinama technologija, kaip srautus paleisti iš grafinės vartotojo sąsajos, susiejant srauto gyvavimą apibrėžiantį *boolean* duomenį *running* su sąsajos mygtukais. Šis duomuo turi dar nematytą modifikatorių *volatile*, kuris neleidžia kompiliatoriui optimizuoti tokio duomens saugojimo. Kitais žodžiais tariant, v*olatile* informuoja, kad kintamojo reikšmė bet kada gali būti programos pakeista ir sąlygą *while(running)* reikia tikrinti kiekvienoje iteracijoje.

Programos rezultatai rodo, kad paleidus programą pakankamai ilgam pasireiškia prioritetų įtaka – aukštesnio prioriteto skaitiklio reikšmė bus akivaizdžiai didesnė. Trumpam paleidus programą, daugiau priaugti gali ir žemesnio prioriteto srauto skaitiklis.

Nuspaudus kelis kartus sąsajos mygtukus „*Start threads*“ ir „*Stop threads*“, skaitiklių reikšmės nebus iš naujo skaičiuojamos nuo nulio. Tokią skaitiklius perskaičiuojančią programos versiją galite pabandyti parašyti patys (abi programos versijos pateikiamos prieduose).

```
//Creating threads with different priorities
//in applet context
//The only evaluation of counters enabled.
//
//<applet code = Threads3 width =600 height=550>
//</applet>
```

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

class Counter implements Runnable{

        int count = 0;
        Thread t;
        volatile boolean running = true;
        public Counter( int priority ){
                t = new Thread( this );
                t.setPriority( priority );
        }

        public void run( ){
                while( running ) count++;
         }

        public void start( ){
                t.start( );
        }

        public void stop( ){
                running = false;
       }

}

public class Threads3 extends JApplet{

  Counter c1 = new Counter( Thread.NORM_PRIORITY+2 ), //priorities for
          c2 = new Counter( Thread.NORM_PRIORITY-2 ); // child-threads

        JButton b1 = new JButton( "Start threads" ),
                b2 = new JButton( "Stop threads " );
        JTextArea ta = new JTextArea( 3,50 );

        public void init( ) {
               Thread.currentThread().setPriority( Thread.MAX_PRIORITY );
                                                        //for current thread!
                b1.addActionListener( new ActionListener( ){
                        public void actionPerformed( ActionEvent ae ){
                                c1.start( );
                                c2.start( );
                                ta.append("   Threads started \n" );
                        }
                } );
                b2.addActionListener( new ActionListener( ){
                        public void actionPerformed( ActionEvent ae ){
```

```
                        c1.stop( );
                        c2.stop( );
                        ta.append( "Low-priority  counter: "+c2.count+ "\n" );
                        ta.append( "High-priority counter: "+c1.count+ "\n" );
                    }
                } );
                try{
                    Thread.sleep( 1000 );
                } catch( InterruptedException ie ){ }

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

Dabar principinė schema, rodanti, kaip klientinėje programoje atskiru srautu paleisti judantį daugialypės terpės elementą. Tokį srautą galima paskelbti demonu. Paprastumo dėlei apsiribosime pačiu paprasčiausiu atveju – judančiąja fraze. Klientinėje programoje yra galimybė sustabdyti ir vėl paleisti judėti frazę. Frazės judėjimas organizuojamas paprastai: imamas pirmasis jos simbolis ir perkeliamas į frazės galą. Frazės slinkimo greitis nustatomas reguliuojant pauzės dydį.

Judančiąją frazę realizuojanti klasė *MovingBanner* yra vidinė – kad būtų patogu prieiti prie grafinių sąsajos elementų.

```
//Applet with moving banner
//
//<applet code = Banner width =500 height=100>
//</applet>

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

public class Banner extends JApplet{

        JTextField tf  = new JTextField( 20 );
        JButton b1     = new JButton( "Start moving"),
                b2     = new JButton( "Stop  moving");
        MovingBanner mb;

        class MovingBanner implements Runnable{

                String message = "   A simple moving banner   ";
                Thread t;
                volatile boolean moving;

                public void start( ){
```

```
                moving = true;
                t = new Thread( this );
                 t.start();
            }

            public void stop( ){
                 moving = false;
            }

            public void run( ){
                char c;
                while( moving ){
                    try{
                        repaint( );
                        Thread.sleep( 100 );
                        c = message.charAt( 0 );
                        message  = message.substring( 1,
                                                message.length( ) );
                        message += c;
                        tf.setText( message );
                    } catch( InterruptedException ie ){ }
                }
            }
        }

        public void init( ){
            b1.addActionListener( new ActionListener( ){
                public void actionPerformed( ActionEvent ae ){
                    mb = new MovingBanner( );
                    mb.start( );
                }
            } );
            b2.addActionListener( new ActionListener( ){
                 public void actionPerformed( ActionEvent ae ){
                    mb.stop( );
                }
            } );
            Container c = getContentPane( );
             c.setLayout( new FlowLayout( ) );
            c.add( tf );
            c.add( b1 );
            c.add( b2 );
        }

}
```

2.19.2. Srautų sinchronizavimas. Tarpsrautiniai ryšiai

Sinchronizavimas leidžia išvengti srautų susidūrimo konkuruojant dėl to paties išteklio. Galima sinchronizuoti visą metodą ar atskirą jo dalį – vadinamąją kritinę sekciją. Sinchronizavimas visada padidina vykdymo laiką. Pavyzdžiui, (B. Eckelis) sinchronizuotas metodo kvietimas vykdomas maždaug keturis kartus ilgiau už įprastąjį kvietimą (čia turimas omenyje pats metodo kvietimas, neįvertinant paties metodo vykdymo).

Metodo sinchronizavimo pavyzdys:

```
synchronized void m1( ){ . . . }
synchronized void m2( ){ . . . }
```

Jei objektui iškviečiamas metodas *m1*, kitas sinchronizuotas (atkreipiame dėmesį, kad šis teiginys galioja tik sinchronizuotam metodui!) metodas *m2* tam pačiam objektui galės būti iškviestas tik pabaigus darbą pirmajam metodui, taigi „objektas blokuojamas“.

Kartais viso metodo blokavimas nereikalingas arba iš viso neįmanomas. Tarkime, programoje *Threads3* metodas

```
public synchronized void run( ){
        while( running ) count++;
}
```

būtų begalinis, kadangi, jį užblokavus, neįmanoma pakeisti *running* reikšmės. Reikėtų blokuoti tik skaitiklio reikšmės keitimą:

```
public void run( ){
        while( running ){
                synchronized( count ){
                        count++;
                }
        }
}
```

Šis pavyzdys nėra prasmingas ir skirtas tik kritinės sekcijos blokavimui paaiškinti: skliaustuose nurodomas blokuojamas objektas, o figūriniuose skliaustuose – visa blokuojamoji sekcija. Toks pavyzdys būtų prasmingas tik tada, jei reikšmei keisti reikėtų kelių operacijų. Čia bloke tėra vienintelė operacija, todėl tokiai paprastai sekcijai tas pat, ar blokuoti ją ar ne.

Sinchronizuotuose srautuose galimi metodai *wait( ), notify( ), notifyAll( )*, padedantys suderinti srautų darbą. Visi šie metodai apibrėžti klasėje *Object*, todėl iš principo prieinami kiekvienai klasei.

final void wait( ) throws InterruptedException – blokuoja savo darbą ir praneša apie tai savo supersrautui.

final void notify( ) – sužadina srautą, kuris tam pačiam objektui buvo iškvietęs *wait( )*.

final void notifyAll( ) – sužadina visus srautus, kurie tam pačiam objektui buvo iškvietę *wait( )*. Pirmasis bus vykdomas aukščiausiojo prioriteto srautas.

Čia parodysime tik srautų darbo suderinimo sintaksę. Pats srautų derinimas išeina už kurso ribų ir yra daugiau lygiagrečiųjų skaičiavimų algoritmų reikalas. Klasikinis tarpsrautinio darbo organizavimo pavyzdys – duomenų struktūros-eilės realizavimas (P. Naughtonas). Programoje yra dvi klasės, palaikančios eilės klasę *Q*: *Producer* „gamina“ objektą – į eilę įdeda eilinį skaičių metodu *void put(int n),* o *Consumer* tą skaičių suvartoja – atspausdina metodu *int get( ).* Jei į programą neįtrauktume *wait* ir *notify* metodų kvietimų, srautų darbas būtų nesuderintas ir, pavyzdžiui, gamintojas galėtų pagaminti iš eilės kelis skaičius, jų nesuvartojus vartotojui, arba vartotojas galėtų suvartoti tą patį skaičių kelis kartus.

Kad programos esmės neuždengtų sąsajos organizavimo ir kiti dalykai, programa parašyta paprasčiausiu būdu ir yra begalinė. Jos darbą galima nutraukti tik klavišais *Ctrl – C*.

```
// Model of queue Q: Producer produces number n, Consumer takes it
// Synchronization. Inter-threads collaboration
//
class Q{

        int n;
        boolean go = false;


        synchronized int get( ){
                if( !go )
                        try{
                                wait( );
                        } catch( InterruptedException ie ){ }
                System.out.println( "Got: "+ n );
                go = false;
                notify( );
                return n;
         }
        synchronized void put( int n ){
                if( go )
                        try{
                                wait( );
                        } catch( InterruptedException ie ){}
                this.n = n;
                System.out.println( "Put: "+ n );
                go = true;
                notify( );
        }
}

class Producer implements Runnable{

        Q q;
        Producer( Q q ){
                this.q = q;
```

```
            new Thread( this,"Producer" ).start( );
        }

        public void run(){
            int i = 0;
            while( true ){
                q.put( i++ );
                try{
                    Thread.sleep( 1000 );
                } catch( InterruptedException ie ){ }
            }
        }
}

class Consumer implements Runnable{

        Q q;
        Consumer( Q q ){
            this.q = q;
            new Thread( this,"Consumer" ).start( );
        }

        public void run( ){
            while( true ) q.get( );
        }
}


class InfiniteQueue{
        public static void main( String [] args ){
            System.out.println( "Press Ctrl-C for finish" );
            Q q = new Q( );
            new Producer( q );
            new Consumer( q );
        }
}
```

2.20. Kolekcijos

Kolekcijos – programinės talpyklos objektų grupėms saugoti (C kalbose vadinamos konteineriais). *Java* kolekcijos aprūpina programuotojus rinkiniu programinių galimybių daugybei dažnai pasitaikančių programavimo uždavinių. Kolekcijos naudotinos visada, kai reikia išsaugoti informaciją kompiuterinėje saugykloje ir vėl ją paimti. Kolekcijos netinkamos tik didelės apimties uždaviniams spręsti; tam yra duomenų bazių sistemos.

Visos kolekcijų klasės ir sąsajų klasės, taip pat klasės įvairiems informacijos įdėjimo ir išėmimo veiksmams, rūšiavimo algoritmams sutelkti pakete *java.util*. Dauguma *Java* kolekcijų pasirodė tik *Java 2* kalboje. Šiame pakete yra ir kitos

pagalbinės *Java* klasės bei sąsajų klasės (manipuliuoti data ir laiku, atsitiktiniams skaičiams generuoti, eilutinių duomenų analizei ir t. t.).

Yra kolekcijos visoms programavime plačiau naudojamoms duomenų struktūroms: dinaminiams masyvams, rinkiniams, sąrašams, susietiesiems sąrašams, medžiams, *hash*-lentelėms. Visi įvairioms kolekcijoms reikalingi algoritmai sutelkti klasėje *Collections* ir įforminti kaip statiniai metodai. Jie yra prieinami visoms kolekcijoms. Metodai, reikalingi duomenims iš kolekcijos išrinkti po vieną iš eilės, kaip jie sudėti į kolekciją, apibrėžti sąsajų klasėje *Iterator* (*Java 1* yra panašias funkcijas atliekanti sąsajų klasė *Enumeration*, tik, aišku, *Java 2* kolekcijose ji nerealizuota). Kiekviena kolekcijų klasė realizuoja sąsajų klasę *Iterator*. Metodai, reikalingi kai kurioms kolekcijoms įvairiai rūšiuoti, apibrėžti sąsajų klasėje *Comparator*.

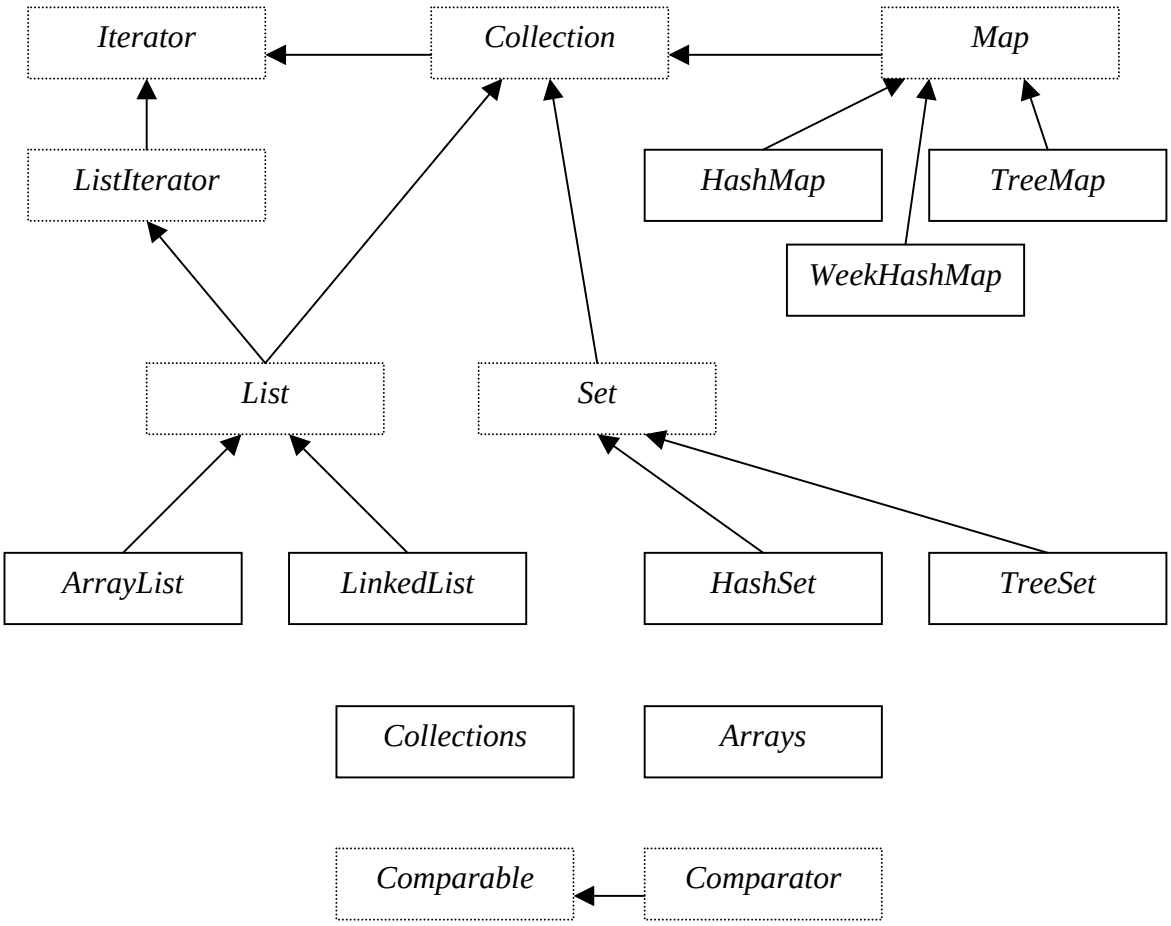
Objektams saugoti skirti ir masyvai, kuriuos nagrinėjome anksčiau ir kuriuos naudojant programose paketas *java.util* nereikalingas. Masyvas – pats efektyviausias duomenų konteineris, tačiau jo matmuo vykdant programą yra pastovus, t. y. programoje turi būti nurodyta, kiek objektų galės tilpti į masyvą, o vėliau masyvas prireikus negalės automatiškai plėstis. Pakete *java.util* yra klasė *ArrayList*, realizuojanti automatiškai besiplečiančius masyvus, bet jos efektyvumas gerokai menkesnis. Pakete taip pat yra klasė *Arrays*, turinti statinius metodus *equals* – dviem masyvams sulyginti, *fill* – masyvui užpildyti norimais duomenimis, *sort* – masyvui rūšiuoti tam tikra tvarka, *binarySearch* – norimam elementui rasti rūšiuotame masyve ir *asList* – masyvui pertvarkyti į sąrašą *List*. Taigi klasė *Arrays* masyvams yra klasės *Collections* kolekcijoms atitikmuo. Masyvams kopijuoti skirtas metodas yra standartinėje *Java* bibliotekoje – klasės *System* statinis metodas *arraycopy* kur kas efektyvesnis nei masyvų kopijavimas pasitelkiant ciklų operatorius. Masyvai skirti tam tikro tipo duomenims, taip pat ir primityvams, saugoti (skyrius 2.8).

Visose kolekcijose yra tik *Object* tipo duomenys. Kadangi *Object* – aukščiausia *Java* klasių hierarchijos klasė, bet kuri tiek *Java*, tiek programuotojo sukurta klasė galės būti įdėta į bet kurią kolekciją (primityvūs duomenys – negalės; tam primityvius duomenis reikia pertvarkyti į atitinkamas klases-kevalus (1.3 skyrius). Tačiau tada prarandamas tikslus kolekcijoje esančių duomenų tipas, todėl vėliau, iš kolekcijos paėmus duomenį, prieš dirbant juo teks pasitelkti besileidžiantįjį duomenų tipų pertvarkymą (2.12 skyrius).

2.20.1. Kolekcijų hierarchija

Jei nenagrinėtume pasenusių *Java 1* kolekcijų, visą hierarchiją galėtume suskirstyti į 3 pagrindinius kolekcijų tipus: *Map, List* ir *Set*. *List* ir *Set* turi bendrą superklasę *Collection*. Sąraše *List* elementai saugomi pagal tam tikrą apibrėžtą tvarką; jame gali būti pasikartojantys elementai. Aibėje, arba rinkinyje, *Set* pasikartojančių elementų nėra. Lentelėje *Map* duomenys saugomi poromis „raktas – reikšmė“; raktai lentelėje nesikartoja. Iš lentelės *Map* galima gauti porų rinkinį, raktų rinkinį ir reikšmių kolekciją *Collection*. Lentelės, panašiai kaip masyvai, gali būti keliamatės (tai lentelės, kurių reikšmės yra kitos lentelės).

Čia pateikiama supaprastinta sąsajų klasių ir klasių hierarchija (B. Eckelis). Sąsajų klasės apvestos punktyriniais rėmeliais, klasės – rėmeliais ištisine linija. Linijos su rodyklėmis rodo paveldimumo schemas.



2.3 pav. Kolekcijų hierarchija

Kaip matyti, sąsajų klasėsi *Set* ir *Map* turi *hash*-subklases *HashSet* ir *HashMap*. Šių klasių dėka galima greita norimo kolekcijos elemento paieška (paieška atitinkamose ne *hash*-klasėse yra pakankamai lėta), panaudojant vadinamąjį *hash*-kodą. Kodas turi sveikuosius skaičius – indeksus, atitinkančius kolekcijos elementus. *Hash*-kodas gaunamas metodu *hashCode*, paveldimu iš klasės *Object*.

2.20.2. *Collection* kolekcijos

Pagrindiniai metodai visoms *List* ir *Set* supertipų kolekcijoms yra:

boolean add(Object o) – įdeda į kolekciją objektą *o* ir grąžina *true*. Grąžinama reikšmė *false* galima tik tada, kai į kolekciją, nepalaikančią kelių vienodų elementų, bandoma įdėti jau toje kolekcijoje esantį objektą.

boolean addAll(Collection c) – įdeda į metodą kviečiančiąją kolekciją visą kolekciją *c*. Grąžinama *boolean* reikšmė – analogiškai metodui *add*.

boolean remove(Object o) – pašalina iš kolekcijos egzempliorių *o* (jei kolekcija palaiko besikartojančius elementus – pašalina tik vieną elementą, lygų *o*). Jei *o* pašalintas, grąžinama reikšmė *true*, jei *o* kolekcijoje nebuvo – niekas nepašalinta – *false*.

boolean removeAll(Collection c) – pašalina iš metodą kviečiančiosios kolekcijos kolekciją *c*. Grąžina *boolean* reikšmes analogiškai metodui *remove*.

void clear( ) – pašalina iš kolekcijos visus elementus.

boolean contains(Object o) – grąžina *true*, jei kolekcija turi egzempliorių *o*; antraip grąžinama *false*.

boolean equals(Object o) – grąžina *true*, jei kviečiančiosios kolekcijos objektas ir objektas *o* yra lygūs; antraip – *false*.

int size( ) – grąžina kolekcijos elementų kiekį.

Object[ ] toArray( ) – grąžina masyvą, kurio elementai yra kolekcijos elementų kopijos.

Iterator iterator( ) – grąžina iteratorių kviečiančiajai kolekcijai.

Sąsajų klasė *List*, kaip matyti iš schemos, išplečia sąsajų klasę *Collection* papildomais metodais. Visi šie papildomi metodai faktiškai leidžia operuoti kolekcijos elementu (ar elementais) tam tikroje kolekcijos vietoje, apibrėžiamoje indeksu. Dažniau iš šių metodų naudojami:

void add(int i, Object o) – įdeda objektą *o* į sąrašo poziciją *i*. Anksčiau šioje pozicijoje buvęs sąrašo elementas ir visi toliau buvę elementai perstumiami tolyn į sąrašo galo pusę per vieną poziciją.

boolean addAll(int i, Collection c) – įdeda visą kolekciją *c* į sąrašą nuo pozicijos *i* panašiai kaip metode *add*. Grąžinama reikšmė *true*, jei kviečiantysis sąrašas pakeičiamas.

Object set(int i, Object o) – perkelia objektą *o* į kviečiančiojo sąrašo poziciją *i*. Grąžinama ankstesnė elemento reikšmė.

Object get(int i) – grąžina objektą, esantį pozicijoje i.

int indexOf(Object o) – grąžina sąraše esančio pirmojo objekto *o* egzemplioriaus indeksą. Jei sąraše tokio objekto nėra, grąžina reikšmę *–1*.

int lastIndexOf(Object o) – veikia panašiai kaip metodas *indexOf*, grąžindamas paskutiniojo objekto *o* egzemplioriaus indeksą.

Object remove(int i) – pašalina pozicijoje *i* esantį sąrašo elementą ir grąžina šio elemento reikšmę. Pertvarkytas sąrašas „sutraukiamas“ – tuščia pozicija užpildoma elementu iš dešinės ir t. t.

List subList(int i1, int i2) – grąžina metodą kviečiančiojo sąrašo dalį – sąrašą, pradedant nuo *i1*-ojo elemento ir baigiant *i2-1*-uoju elementu.

ListIterator listIterator( ) – grąžina sąrašui iteratorių, pasiruošusį perrinkti nuo sąrašo pradžios sąrašo elementus.

ListIterator listIterator(int i) – grąžina sąrašui iteratorių, pasiruošusį perrinkti sąrašo elementus nuo sąrašo *i*-ojo elemento.

Sąsajų klasė *Set*, kaip matyti iš schemos, išplečia sąsajų klasę *Collection*, tačiau neįveda jokių papildomų metodų. Kadangi rinkinys nesaugo besikartojančių elementų, metodas *add* jam grąžintų *false*, bandant įdėti į rinkinį jau egzistuojantį objektą.

Dabar trumpai peržvelgsime klases, realizuojančias sąsajų klasę *List*: *ArrayList* ir *LinkedList*.

Klasė *ArrayList* faktiškai realizuoja dinaminį masyvą, galintį plėstis vykdant programą. Masyve saugomos rodyklės į objektus. Masyvas sukuriamas tam tikro pradinio dydžio. Kai programos vykdymo metu šis masyvo pradinis dydis viršijamas, masyvas automatiškai plečiamas. Jei iš anksto aišku, kad masyvas pasieks tam tikrą dydį, galima iš anksto klasės metodu *void ensureCapacity(int capacity)* nustatyti reikiamą masyvo dydį: taip bus išvengta kelių automatinių masyvo plėtimų perrašant visus masyvo elementus ir sutaupytas programos vykdymo laikas. Klasėje perkrautos 3 konstruktoriaus versijos:

ArrayList( )
ArrayList(Collection c)
ArrayList(int capacity)

Pirmasis konstruktorius sukuria tuščią tam tikro dydžio nuorodų masyvą, trečiasis – tuščią *capacity* dydžio masyvą, o antrasis – kolekcijos dydžio masyvą, turintį kolekcijos elementų reikšmes.

Pavyzdys kai kuriems klasės metodams iliustruoti: sąrašo suformavimas, pradinių *String* tipo reikšmių suteikimas, elemento įdėjimas į sąrašą ir pašalinimas iš sąrašo, šio pertvarkymas į masyvą (jo elementai bus *Object* tipo) ir masyvo elementų sumavimas:

```
import java.util.*;
class MyArrayList{
        public static void main( String args[ ] ){
                ArrayList al = new ArrayList( );
                System.out.println( "Initial size of array " + al.size( ) );
                al.add( "A" );
                al.add( "B" );
                al.add( "C" );
                al.add( "D" );
                al.add( 1,"A1" );
                System.out.println( "Size of array after resizing " + al.size( ) );
                System.out.println( "Array: " + al ); // invokes method toString
                        al.remove( "C" );
```

```
                    al.remove( 1 );
                    System.out.println( "Size of array after removing " +
                            al.size( ) );
                    System.out.println( "Array: " + al );
                    Object array[ ] = al.toArray( ); // transforming list to
                                                     // an array
                    String sum = "";
                    for( int i=0; i<array.length; i++ )
                            sum = sum + ( String )array[ i ]; // downcasting
                    System.out.println( "Sum of array elements: " + sum );
            }
    }
```

Programa, spausdindama kolekcijos elementus, neišreikštai kviečia metodą *toString* (tai pažymėta komentare atitinkamoje eilutėje). Programa spausdina:

```
Initial size of array 0
Size of array after resizing 5
Array: [A, A1, B, C, D]
Size of array after removing 3
Array: [A, B, D]
Sum of array elements: ABD
```

Susietojo sąrašo klasės *LinkedList* objektai kuriami konstruktoriais *LinkedList()* arba *LinkedList(Collection c)*. Klasė turi kelias naujas galimybes, palyginti su *ArrayList* – metodus, kurių paskirtis aiški iš jų pavadinimų:

void addFirst(Object o) – įdeda *o* į sąrašo pirmąją poziciją.
void addLast(Object o) – analogiškai į paskutinę poziciją.
Object getFirst( )
Object getLast( )
Object removeFirst( )
Object removeLast( )

Rinkinio klasė *HashSet* turi 4 perkrautus konstruktorius:

```
HashSet( )
HashSet( Collection c )
HashSet( int capacity )
HashSet( int capacity, float fillRatio )
```

Jų prasmė aiški iš analogijos su *ArrayList* konstruktoriais, išskyrus 4-ąją formą: rinkinio užpildymo koeficientas *fillRatio* nustato ribą, kurią pasiekus rinkinys plečiamas. Koeficiento reikšmės turi būti tarp 0 ir 1, o standartinė koeficiento reikšmė – 0,75. Šios klasės objekto elementai nėra kaip nors papildomai išrikiuojami, tuo tarpu *TreeSet* elementai – išdėstomi natūralia didėjančia tvarka (pavyzdžiui, simboliniai duomenys – pagal lotynų abėcėlę) arba komparatoriumi nustatyta tvarka. Šios klasės objektai gali būti kuriami vienu iš konstruktorių

TreeSet( )
TreeSet(Collection c)
TreeSet(Comparator comp)
TreeSet(SortedSet ss)

Čia *SortedSet* yra dar viena anksčiau neminėta sąsajų klasė rūšiuotam rinkiniui, įsiterpianti tarp *TreeSet* klasės ir *Set* sąsajų klasės.

Pavyzdys:

```
import java.util.*;
class MyTreeSet{
        public static void main( String args[ ] ){
                TreeSet ts = new TreeSet( );
                ts.add( "F" );
                ts.add( "b" );
                ts.add( "B" );
                ts.add( "K" );
                ts.add( "A" );
                ts.add( "a" );
                System.out.println( "Set: " + ts );
        }
}
```

Programa spausdina:

Set: [A, B, F, K, a, b]

Cikliškai po vieną perrinkti bet kurios kolekcijos elementus leidžia vadinamasis iteratoriaus objektas, sukuriamas kolekcijai kviečiamu minėtuoju metodu *Iterator iterator( )*. Sąrašą galima perrinkti galingesniu analogiškai gaunamu *ListIterator* iteratoriumi. Klasės *Iterator* metodai yra:

boolean hasNext( ) – grąžina *true*, jei kolekcijoje yra kitas elementas.

Object next( ) – grąžina kitą elementą. Jei kito elemento nėra, sukeliama išimtis *NoSuchElementException*.

void remove( ) – pašalina dabartinį elementą. Visada turi būti kviečiamas po *next( )*; jei to nepaisysime – kils išimtis *IllegalStateException*.

Klasė *ListIterator* papildomai siūlo metodus:

void add(Object o) – į sąrašą prieš elementą, kuris būtų grąžintas metodu *next( )*, įterpiamas *o*.

boolean hasPrevious( ) – grąžina *true*, jei kolekcijoje yra ankstesnis elementas.

Object previous( ) – grąžina ankstesnįjį elementą. Jei jo nėra – kyla išimtis *NoSuchElementException*.

int nextIndex( ) – grąžina kito elemento indeksą.

int previousIndex( ) – grąžina ankstesniojo elemento indeksą.

void set(Object o) – dabartiniam elementui (t. y. elementui, grąžintam metodais *next( )* ar *previous( )*) suteikiama *o* reikšmė.

TreeSet objektų elementai, kaip minėta, rūšiuojami arba natūraliai, arba komparatoriumi teikiama tvarka. Sąsajų klasėje *Comparator* yra du metodai:

int compare(Object o1, Object o2) – grąžina 0, jei objektai lygūs; teigiamą reikšmę, jei *o1>o2*; neigiamą reikšmę, jei *o1<o2*. Jei objektų tipai nesuderinami, kyla išimtis *ClassCastException*.

boolean equals(Object o) – lygina du komparatorius: kviečiantįjį metodą ir *o*.

Pavyzdys, iliustruojantis iteratoriaus ir komparatoriaus panaudojimą rūšiuotame rinkinyje. Komparatorius nustato atvirkščią elementų išdėstymo tvarką. Metode *compare* kviečiamas metodas *compareTo* priklauso *String* klasei, o jo grąžinamos *int* tipo reikšmės analogiškos grąžinamoms iš metodo *compare*.

```
import java.util.*;

class MyComparator implements Comparator{
        public int compare( Object o1, Object o2 ){
                String s1 = (String) o1,
                        s2 = (String) o2;
                return s2.compareTo( s1 ); // reversing the natural order
        }
}

class MyIteratorComparator{
        public static void main( String args[ ] ){
                TreeSet ts = new TreeSet( new MyComparator( ) );
                ts.add( "F" );
                ts.add( "b" );
                ts.add( "B" );
                ts.add( "K" );
                ts.add( "A" );
                ts.add( "a" );
                Iterator i = ts.iterator( ); // getting iterator
                while( i.hasNext( ) ){
                        Object tse = i.next( );
                        System.out.print( tse + " " );
                }
        }
}
```

Programa spausdina *TreeSet* objekto *ts* elementus, išrikiuotus atvirkščia tvarka:

b a K F B A

2.20.3. *Map* lentelės

Sąsajų klasėje *Map* apibrėžti pagrindiniai metodai:

Object put(Object key, Object Value) – įdeda į lentelę vadinamąjį lentelės įėjimą – raktą *key* ir pagal jį atrenkamą reikšmę *value*. Grąžinama reikšmė *null,* jei teikiamo rakto lentelėje nebuvo; antraip – ankstesnė su raktu susieta reikšmė.

void putAll(Map m) – įdeda į lentelę visą lentelę *m*.

Object get(Object key) – grąžina su raktu *key* susietą reikšmę.

int size( ) – grąžina lentelės įėjimų kiekį.

Set entrySet( ) – grąžina lentelės įėjimų – sąsajų klasės *Map.Entry* tipo objektų rinkinį.

Set keySet( ) – grąžina lentelės raktų rinkinį.

Collection values( ) – grąžina lentelės reikšmių kolekciją.

Object remove(Object key) – pašalina iš lentelės įėjimą, kurio raktas yra *key*, ir grąžina šį objektą.

boolean containsKey(Object key) – grąžina *true*, jei metodą kviečianti lentelė turi raktą *key*; antraip – *false*.

boolean containsValue(Object value) – analogiškas ankstesniam metodas, tik žiūrima lentelėje esančių reikšmių.

Sąsajų klasę *Map* išplečia sąsajų klasės *SortedMap* – rūšiuotai lentelei bei *Map.Entry*, leidžianti manipuliuoti lentelės įėjimais. Šios sąsajų klasės papildo minėtuosius metodus dar keliais naudingais metodais. Dalis jų yra:

Comparator comparator( ) – grąžina lentelės komparatoriaus objektą.

Object firstKey( ) – grąžina rūšiuotos lentelės pirmąjį raktą.

Object lastKey( ) – analogiškai paskutinįjį.

SortedMap subMap(Object start, Object end) – grąžina išrūšiuotą lentelę, turinčią tik įėjimus, kurių raktų reikšmės didesnės arba lygios *start* ir mažesnės už *end*.

Object getKey( ) – grąžina dabartinio lentelės įėjimo raktą.

Object getValue( ) – grąžina dabartinio lentelės įėjimo reikšmę.

Šias sąsajų klases realizuoja klasės *HashMap, WeakHashMap* ir *TreeMap*. *Hash*-lentelė *HashMap* garantuoja vienodai greitą prieitį prie visų lentelės įėjimų. Jos konstruktoriai yra (konstruktoriai analogiški anksčiau nagrinėtiems klasės *HashSet* konstruktoriams, todėl jų prasmės nereikia papildomai aiškinti):

HashMap( )
HashMap(Map m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)

TreeMap lentelė sudaryta naudojant duomenų struktūrą – medį. Lentelė išrūšiuota pagal raktus natūralia arba komparatoriumi teikiama tvarka, arba, jei lentelei suteikiamos *SortedMap* įėjimų reikšmės – *SortedMap* tvarka. Yra konstruktoriai:

TreeMap( )
TreeMap(Comparator comp)
TreeMap(Map m)
TreeMap(SortedMap sm)

Pavyzdys su lentelėmis pateiktas kitame skyriuje, kalbant apie klientines programas.

2.20.4. *Collections* klasės algoritmai

Kolekcijoms ir lentelėms galima taikyti 29 algoritmus, įformintus kaip klasės *Collections* statiniai metodai. Čia paminėti tik keli iš jų:

static void fill(List l, Object o) – užpildo sąrašą *l* objekto *o* kopijomis.

static Object max(Collection c) – grąžina didžiausią pagal natūralią tvarką kolekcijos *c* elementą.

static Object min(Collection c) – analogiškas ankstesniam metodas.

static Object max(Collection c, Comparator comp) – grąžina didžiausią kolekcijos *c* elementą pagal komparatorių *comp*.

static Object min(Collection c, Comparator comp) – analogiškas ankstesniam metodas.

static void reverse(List l) – sąrašo elementus išdėsto atvirkščia tvarka.

static void sort(List l, Com,parator comp) – rūšiuoja sąrašo *l* elementus pagal komparatorių *comp*.

Baigiamosios pastabos. Kolekcijų ir lentelių pasirinkimas gana didelis. Patarsime, kada ir kokią pasirinkti kolekciją:

1. Jei vykdant programą teks daug kartų išrinkti bet kuriuos sekos elementus, efektyviausia naudoti *ArrayList* klasės objektą.
2. Jei teks daug kartų į seką įterpti ar išmesti elementus, naudotinas *LinkedList* objektas.
3. Rinkiniai *Set* saugo tik unikalius sekoje elementus. *HashSet* garantuoja pačią greičiausią elementų paiešką.
4. Jei rinkinys turi būti išrūšiuotas, naudotinas *TreeSet* objektas.
5. Primityvioms duomenų bazėms kurti užtenka lentelių *Map* galimybių.