Paralēlie procesi un valoda Ada

Asociētais profesors

Pāvels Rusakovs

Programmēšanas valoda Ada

Standarti Ada'83 un Ada'95

Literatūras saraksts:

1. *Нарайн Джехани*. Язык Ада.
Москва, “Мир”, 1988.

2. *Юджин Василеску*. Прикладное программирование на языке Ада.
Москва, “Мир”, 1990.

3. *Jānis Osis*. Programmēšanas valoda ADA.
Rīgas Tehniskā universitāte, Rīga, 1993.

4. *John Barnes*. Programming in Ada.
Addison-Wesley, 1995.

5. *John Barnes*. Programming in Ada 2005.
Addison-Wesley, 2006.

6. *Alan Burns*. Real Time Systems & Programming Languages: Ada 95, Real-Time Java and Real-Time C/POSIX. Addison-Wesley, 2001.

7. *Ada Home: The Web Site for Ada*.
Internets: http://www.adahome.com.

8. *Ada Information Clearinghouse*.
Internets: http://www.adaic.com.

9. *Special Interest Group on Ada*.
Internets: http://www.sigada.org.

10. *Ada Tools and Resources*.
Internets: http://www.adapower.com.

11. *AdaCore*.
Internets: http://www.adacore.com.

Modernie Ada projekti

- pasažieru lidmašīnas *Boeing 777* programmatūra (apmēram 99%);
- lidmašīnas-amfībijas *Be-200* programmatūra (Krievija);
- teleskopa programmatūra (*Helsinki University of Technology*);
- Ņujorkas metropolitēna vadības sistēma (*New York City subway*).

Informācijas izvade

Teksta ziņojuma izvade (fails *Out1.adb*)

```
with Text_IO;
procedure Out1 is
begin
   Text_IO.Put("Hello, World !");
end Out1;
```

Komentāri:

1. *Text_IO* – pakotne.
2. *Put( )* – izvades funkcija.
3. Pakotnes vārdu izmanto kā *Put( )* funkcijas prefiksu.

Teksta ziņojuma izvade (fails *Out2.adb*)

```
with Text_IO;
use  Text_IO;
procedure Out2 is
begin
   Put("Hello, World !");
end Out2;
```

Komentāri:

1. Ir viena papild rindiņa: ***use*** *Text_IO*.
2. Tagad pakotnes vārdu neizmanto kā prefiksu.
3. Rindiņās ar ***with*** un ***use*** var norādīt vairākas pakotnes.

Uzdevums: izvadīt veselu skaitli.

```
Put(2);
```

Rezultāts: kompilācijas kļūda. Ziņojuma fragments:

```
possible missing instantiation of Text_IO.Integer_IO
```

Risinājums: patstāvīgi izveidot pakotni veselu skaitļu izvadei.

```
package IO_Integer is new Integer_IO(Integer);
use IO_Integer;
```

1. *Integer_IO* ir *noskaņojamā* pakotne (šablons).
2. *Integer* ir tips-parametrs.
3. *IO_Integer* ir izveidotā pakotne.

Programmas teksts (fails *Out_i_1.adb*)

```
with Text_IO;
use  Text_IO;
procedure Out_i_1 is
   i : Integer := 1;

   package IO_Integer is new Integer_IO(Integer);
   use IO_Integer;

begin
   Put(i, 2);
end Out_i_1;
```

Cita iespēja (fails *Out_i_2.adb*). Eksistē tikai Ada'95 standartā.

```
with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;
procedure Out_i_2 is
   i : Integer := 1;
begin
   Put(i, 2);
end Out_i_2;
```

Komentāri:

1. *Ada.Integer_Text_IO* – sistēmas pakotne.
2. Nevajag veidot savu pakotni veselu skaitļu izvadei.

Ierobežotā tipa vērtību izvade:

```
with Text_IO;
use  Text_IO;
procedure Out_i_3 is
   type Integer_1_100 is new Integer
      range 1..100;
   i : Integer_1_100 := 1;

   package IO_Integer_1_100 is new
      Integer_IO(Integer_1_100);
   use IO_Integer_1_100;

begin
   Put(i, 2);
end Out_i_3;
```

Uzskaitāmā tipa vērtību izvade:

```
with Text_IO;
use  Text_IO;
procedure Out_Fig is
   type Shapes is (Square, Rectangle, Circle);

   package IO_Shapes is new
      Enumeration_IO(Enum => Shapes);
   use IO_Shapes;

begin
   for Shape in Shapes loop
      Put(Shape);
      New_Line;
   end loop;
end Out_Fig;
```

Tipi un apakštipi

Tipi (types): *ievades/izvades pakotnes* un mainīgo deklarēšana.

```
with Text_IO;
use  Text_IO;
...
   type Speed is new Float range 0.0..300.0;
   type Weight is new Float range 0.0..6000.0;

   package Speed_IO is new Float_IO(Speed);
   use Speed_IO;

   package Weight_IO is new Float_IO(Weight);
   use Weight_IO;

   AutoSpeed  : Speed  := 60.0;
   AutoWeight : Weight := 2500.0;
```

Tipi (types): operācijas ar *dažādu tipu* mainīgajiem.

```
Put("Speed: ");
Put(AutoSpeed, 4, 1, 0);
New_Line;

Put("Weight: ");
Put(AutoWeight, 4, 1, 0);
New_Line;

-- AutoSpeed := AutoWeight; (ERROR     !)
-- AutoSpeed := 301.0;      (EXCEPTION !)
```

Komentāri:

1. *Katram tipam* ir nepieciešamas ievades/izvades pakotnes.
2. Nav iespējama *atšķirīgo tipu* vērtību piešķire.

Apakštipi (subtypes):

```
with Text_IO, Ada.Float_Text_IO;
use Text_IO, Ada.Float_Text_IO;
...
   subtype Speed is Float range 0.0..300.0;
   subtype Weight is Float range 0.0..6000.0;

   AutoSpeed  : Speed  := 60.0;
   AutoWeight : Weight := 2500.0;
...
   Put("Speed: ");
   Put(AutoSpeed, 4, 1, 0);New_Line;

   Put("Weight: ");
   Put(AutoWeight, 4, 1, 0);New_Line;

   AutoWeight := AutoSpeed;
```

Neierobežotie masīvi: matricas elementu izvade.

```
with Text_IO, Ada.Integer_Text_IO;
...
   N : constant Integer := 2;
   M : constant Integer := 3;
   type MatrixType is array
      (Integer range <>, Integer range <>)
         of Integer;
   Matr : MatrixType(1..N, 1..M) :=
      ( (1, 2, 3), (4, 5, 6));
...
   for i in Matr'First..Matr'Last loop
      for j in Matr'Range(2) loop
         Put(Matr(i, j), 3);
      end loop;
      New_Line;
   end loop;
```

Citas iespējas organizēt apstrādes ciklus:

```
for i in Matr'First(1)..Matr'Last(1) loop
   for j in Matr'First(2)..Matr'Last(2) loop
      Put(Matr(i, j), 3);
   end loop;
   New_Line;
end loop;
```

```
for i in Matr'Range(1) loop
   for j in Matr'Range(2) loop
      Put(Matr(i, j), 3);
   end loop;
   New_Line;
end loop;
```

Izņēmumu apstrāde

Dalīšana ar nulli:

```
with Text_IO, Ada.Integer_Text_IO,
   Ada.Float_Text_IO;
...
   N : Integer;
   Revrs: Float;
begin
   Put("Enter N:");
   Get(N);
   Revrs := Float(1.0/N);
   Put(Revrs, 4, 2, 0);
exception
   when Constraint_Error =>
      Put("Division By Zero.");
```

Programmētāja izņēmums:

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;
...
   N : Integer;
   NegativeValue:exception;
...
   Put("Enter N:");
   Get(N);
   if (N<0) then
      raise NegativeValue;
   end if;
exception
   when NegativeValue =>
      Put("Exception: negative value.");
```

Vairāki izņēmumu apstrādātāji:

```
begin
   ...
exception
   when NegativeValue =>
      Put("Exception: negative value.");
   when Constraint_Error =>
      Put("Division By Zero.");
```

Vairāku izņēmumu apstrāde *vienā apstrādātājā*:

```
begin
...
exception
   when NegativeValue | Constraint_Error =>
      Put("Exception.");
```

Apakšprogrammas

Funkciju (procedūru) deklarēšana: fails *Main.adb*.

```
procedure Main is
   type VType is array(Integer range <>)
      of Integer;
   function SumNeg(V : in VType)
      return Integer is
      S : Integer := 0;
   begin
      ...
      return S;
   end SumNeg;
begin
   ...
end Main;
```

Formālo parametru deklarēšana

1. **in** – parametri *(pēc noklusējuma)*.

Nevar izmainīt parametra vērtību.

```
function SumNeg(V : in VType)
   return Integer is
begin
   -- V(1) := 1; kompilācijas kļūda
   ...
end SumNeg;
```

2. **out** – parametri. Parametru *inicializē* apakšprogrammā.

```
procedure SumNeg(V : in VType;
    Sum: out Integer) is
begin
    ...
end SumNeg;
Vect:VType(1..3) := (1, 2, 3);
S:Integer;
...
SumNeg(Vect, S);
```

---

3. **in out** – parametri. Parametru var *izmainīt* apakšprogrammā.

Klase - lietotāja pakotne (koordinātu punkts)

1. Pakotnes *interfeisa* fragments (fails *CP_Pack.ads*).

```
package CP_Pack is
   type CoordPoint is private;
   procedure Init(CPoint : out CoordPoint;
      Px : in Integer; Py : in Integer);
   function GetX(CPoint : in CoordPoint)
      return Integer;
   procedure SetX(CPoint : in out CoordPoint;
      Px : in Integer);
   procedure Print(CPoint : in CoordPoint);
private
   type CoordPoint is record
      X,Y : Integer;
   end record;
end CP_Pack;
```

2. Pakotnes *realizācijas* fragments (fails *CP_Pack.adb*).

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;
package body CP_Pack is
   procedure Init(CPoint : out CoordPoint;
      Px : in Integer; Py : in Integer) is
   begin
      CPoint.X := Px;
      CPoint.Y := Py;
   end Init;

   function GetX(CPoint : in CoordPoint)
      return Integer is
   begin
      return CPoint.X;
   end GetX;
```

2. Pakotnes *realizācijas* fragments (fails *CP_Pack.adb*).

```
    procedure SetX(CPoint : in out CoordPoint;
       Px : in Integer) is
    begin
       CPoint.X := Px;
    end SetX;
    procedure Print (CPoint : in CoordPoint) is
    begin
       Put("X: ");Put(CPoint.X, 1);
       Put(", Y: ");Put(CPoint.Y, 1);
       Put(".");
       New_Line;
    end Print;
end CP_Pack;
```

3. Galvenā programma (fails *Main.adb*).

```
with CP_Pack;
use CP_Pack;
procedure Main is
   CP : CoordPoint;
begin
   Init(CP, 1, 2);
   Print(CP);
end Main;
```

Komentāri:

1. Galvenajā programmā izmanto tikai pakotni *CP_Pack*.
2. Nav iespējams adresēt *CP.X* un *CP.Y* ar piešķires palīdzību.

Personisku tipu (**`private`**) eksemplārus var:

1. *Izveidot*.
2. *Nodot* apakšprogrammai kā parametrus.
3. *Salīdzināt*: "ir vienāds"/"nav vienāds" (**=** vai **/=**).
4. *Piešķirt* vienu otram.

Limitētu personisku tipu (**`limited private`**) eksemplārus var:

1. *Izveidot*.
2. *Nodot* kā parametrus.

Ierakstu mantošana (citējamie ieraksti)

Ierakstu tipu deklarēšana

```
type CoordPoint is tagged record
   X : Integer;
   Y : Integer;
end record;
type DisplayPoint is new CoordPoint with record
   Color : Integer;
end record;
```

Mainīgo deklarēšana

```
CP : CoordPoint := (1, 2);
DP : DisplayPoint := (3, 4, 5);
```

Mainīgajos var norādīt arī lauku vārdus:

```
CP : CoordPoint := (X => 1, Y => 2);
DP : DisplayPoint := (Y => 4, X => 3,
   Color => 5);
```

Darbs ar ierakstu laukiem notiek "parastajā" stilā:

```
Put(DP.X, 2);        -- 3
Put(DP.Y, 2);        -- 4
Put(DP.Color, 2);    -- 5
```

Ierakstu mantošanas dziļums nav ierobežots.

Pakotņu mantošana (*atvasinātās* pakotnes). Klase "displeja punkts".

1. Izmaiņas pakotnē "koordinātu punkts".

```
package CP_Pack is
   type CoordPoint is tagged private;
   procedure Init(CPoint : out CoordPoint'Class;
      ...);
   ...
private
   type CoordPoint is tagged record
      ...
   end record;
end CP_Pack;
```

2. Pakotnes "*displeja punkts*" interfeisa fragments.
Sākums.

```
package CP_Pack.DP_Pack is
   type DisplayPoint is new CP_Pack.CoordPoint
      with private;
   procedure Init(DPoint : out DisplayPoint;
      Px, Py, PColor : in Integer);
   function  GetColor(DPoint : in DisplayPoint)
      return Integer;
   procedure SetColor(DPoint : in out
      DisplayPoint; PColor : in Integer);
   procedure Print(DPoint : in DisplayPoint);
```

2. Pakotnes “displeja punkts” interfeisa fragments.
Beigas.

```
private
   type DisplayPoint is new CP_Pack.CoordPoint
      with record
      Color : Integer;
   end record;
end CP_Pack.DP_Pack;
```

Komentāri:

1. Atvasinātās pakotnes faila vārds: *cp_pack-dp_pack.ads*.
2. Pakotņu vārdi veido “ceļu” (*hierarhiju*).
3. Ir iespēja adresēt informāciju no *bāzes* pakotnes.

3. Pakotnes "*displeja punkts*" realizācija. Sākums.

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;
package body CP_Pack.DP_Pack is
   procedure Init(DPoint : out DisplayPoint;
      Px, Py, PColor : in Integer) is
   begin
      CP_Pack.Init(DPoint, Px, Py);
      DPoint.Color := PColor;
   end Init;

   function  GetColor(DPoint : in DisplayPoint)
      return Integer is
   begin
      return DPoint.Color;
   end GetColor;
```

3. Pakotnes "*displeja punkts*" realizācija. Beigas.

```
   procedure SetColor(DPoint : in out
      DisplayPoint; PColor : in Integer) is
   begin
      DPoint.Color := PColor;
   end SetColor;
   procedure Print(DPoint : in DisplayPoint) is
   begin
      Put("X: ");Put(DPoint.X, 1);
      Put(", Y: ");Put(DPoint.Y, 1);
      Put(", Color: ");
      Put(DPoint.Color, 1);Put(".");
      New_Line;
   end Print;
end CP_Pack.DP_Pack;
```

4. Galvenā programma.

```
with CP_Pack, CP_Pack.DP_Pack;
use  CP_Pack, CP_Pack.DP_Pack;
procedure Main is
   CP : CoordPoint;
   DP : DisplayPoint;
begin
   Init(CP, 1, 2);
   Init(DP, 3, 4, 5);
   Print(CP);    -- X: 1, Y: 2.
   Print(DP);    -- X: 3, Y: 4, Color: 5.
end Main;
```

Paralēlā programmēšana

1. Pavedienus (*threads*) realizē ar *uzdevumu* (**tasks**) palīdzību.
2. Uzdevumiem ir *interfeiss* (**task**) un *realizācija* (**task body**).
3. Uzdevumus nevar *kompilēt neatkarīgi* (tos izvieto procedūrās vai pakotnēs).
4. Ir iespējama uzdevumu *savstarpēja iedarbība*.
5. Eksistē *statiskie* un *dinamiskie* uzdevumi.

4. Elementārais uzdevums procedūrā.

```
procedure Main is
   task T;
   task body T is
   begin
      null;
   end T;
begin
   null;
end Main;
```

Komentāri:

1. *T* ir *statisks neatkarīgs* uzdevums.
2. *T automātiski* izpildās pateicoties savai deklarēšanai.

1. Paralēlā ziņojumu izvade. *Pirmais* uzdevums.

```
with Text_IO; use Text_IO;
procedure T1 is
   N : constant Integer := 5;
   task First;
   task Second;
   task body First is
      D : constant Duration := 0.5;
   begin
      for i in 1..N loop
         Put_Line("First.");
         delay D;
      end loop;
   end First;
```

2. Paralēlā ziņojumu izvade. *Otrais* uzdevums.

```
   task body Second is
      D : constant Duration := 0.3;
   begin
      for i in 1..N loop
         Put_Line("Second.");
         delay D;
      end loop;
   end Second;
begin
   Put_Line("Main program.");
end T1;
```

3. Paralēlā ziņojumu izvade. *Rezultāti*.

```
First.
Main program.
Second.
Second.
First.
Second.
Second.
First.
Second.
First.
First.
```

Uzdevumi var organizēt *savstarpējo iedarbību* ar citiem uzdevumiem, izmantojot dažādus mehānismus.

Viens no tiem ir *satikšanās* (`rendezvous`) mehānisms. Tas ir populārākais mehānisms.

Izsaucamā uzdevuma *interfeisā* jābūt *ieeja* (**`entry`**).

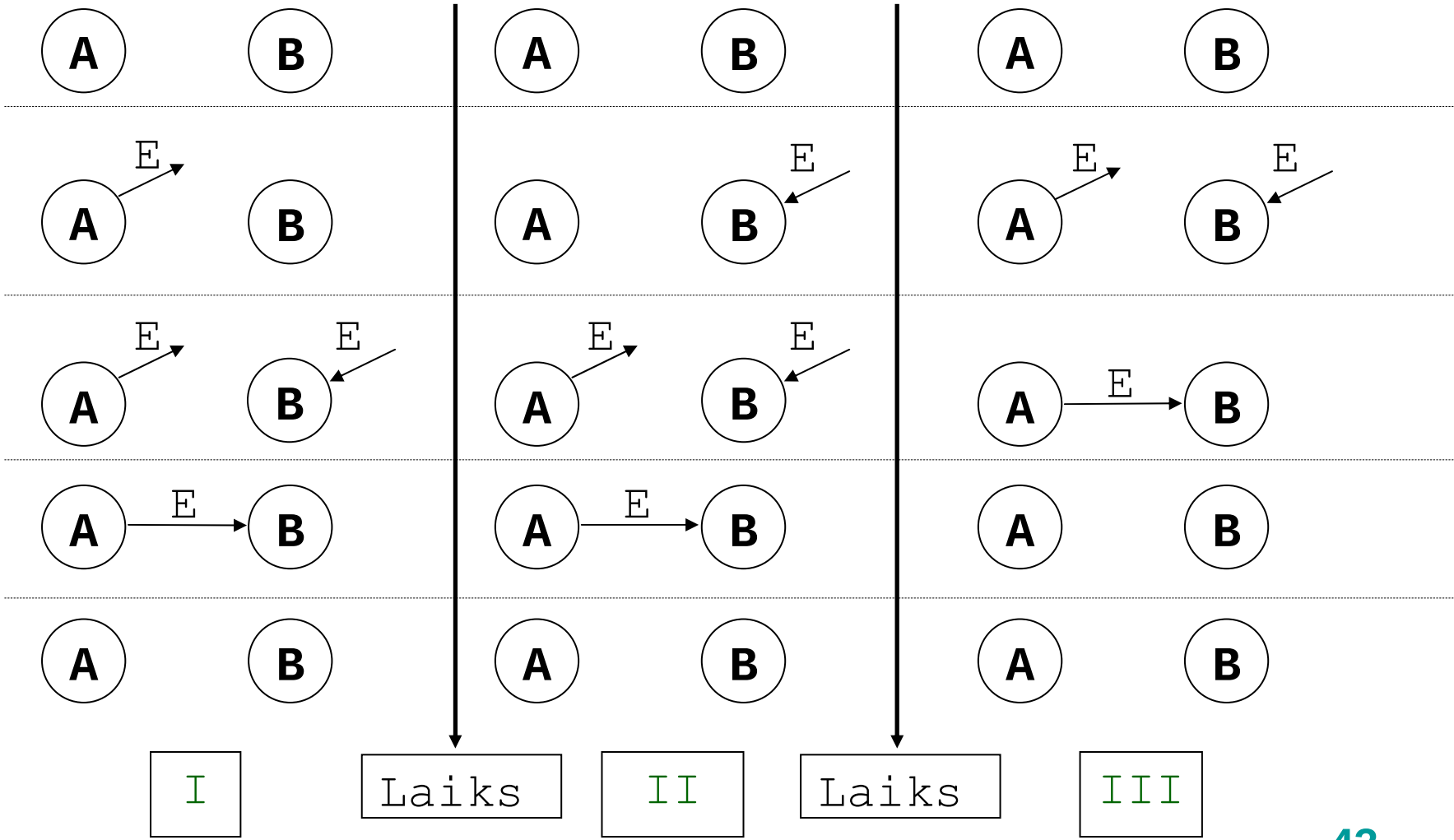
Izsaucamā uzdevuma *realizācijā* katrai ieejai atbilst *vismaz viens* pieņemšanas operators (**`accept`**).
Visbiežāk runa ir par vienu operatoru.

Satikšanu procesā abi uzdevumi *zaudē savu neatkarību*.

Nepieciešams *maksimāli saīsināt* pieņemšanas operatora kodu.

Savstarpējā iedarbība: vizualizācija.

Lai uzdevumam **B** ir ieeja E. Uzdevums **A** izsauc šo ieeju.



1. Uzdevuma *interfeisa* piemērs:

```
task Solver is
   entry GetParam(Param : in Integer);
end Solver;
```

---

2. Uzdevuma *realizācijas* piemērs:

```
task body Solver is
   Num : Integer;
begin
   ...
   accept GetParam (Param : in Integer) do
      Num := Param;
   end GetParam;
   Put ("The parameter is:");Put(Num, 2);
   ...
end Solver;
```

3. Uzdevuma ieejas *izsaukuma* piemērs:

```
Info : Integer;
...
Solver.GetParam(Info);
```

Sintaksiski, *ieejas izsaukums* ekvivalents procedūras izsaukumam.

Uzdevuma vārdu izmanto *kā prefiksu*.

Uzdevumu *var pārtraukt* izpildes laikā, lietojot **abort**.

abort T_1, T_2,...T_n;

Mūsu gadījumā:

```
abort Solver;
```

1. Satikšanās. Uzdevumu interfeisi un "*piegādātāja*" realizācija

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;
procedure T2 is
   N : constant Integer := 5;
   task First;
   task Second is
      entry Info(Num: in Integer);
   end Second;

   task body First is
   begin
      for i in 1..N loop
         Second.Info(i);
      end loop;
   end First;
```

2. Satikšanās. "*Saņēmēja*" realizācija.

```
task body Second is
   D : constant Duration := 0.3;
   X : Integer;
begin
   loop
      accept Info(Num : in Integer) do
         X := Num;
      end Info;
      Put("Message "); Put(X, 1); Put(".");
      New_Line;
      exit when X = 5;
      delay D;
   end loop;
end Second;
```

3. Satikšanās. Galvenā programma.

```
begin
   null;
end T2;
```

Rezultāti:

```
Message 1.
Message 2.
Message 3.
Message 4.
Message 5.
```

Uzdevuma *tips*: neatkarīga izpilde.

```
with Text_IO;
use  Text_IO;
procedure Test is
   task type T;
   task body T is
   begin
      Put("Hello !");
   end T;
   T1, T2 : T;
begin
   null;
end Test;
```

Rezultāts:

```
Hello !Hello !
```

1. Uzdevumu šablons. *Šablona* deklarācija un realizācija.

```
with Text_IO;
use  Text_IO;
procedure T3 is
   N : constant Integer := 3;
   task type BaseTask(C : Character;
      D1, D2 : Integer);
   task body BaseTask is
   begin
      for i in 1..N loop
         Put("Message " & C); Put(".");
         New_Line;
         delay Duration(D1)/Duration(D2);
      end loop;
   end BaseTask;
```

2. Uzdevumu šablons. *Uzdevumu* izveidošana.

```
   T1 : BaseTask('1', 1, 2);
   T2 : BaseTask('2', 2, 3);
begin
   null;
end T3;
```

Rezultāti:

```
Message 1.
Message 2.
Message 1.
Message 2.
Message 1.
Message 2.
```

Dinamiskie uzdevumi. Izmaiņas iepriekšējā piemērā:

```
procedure T3 is
   ...
   type DynTask is access BaseTask;
   D1, D2 : DynTask;
begin
   D1 := new BaseTask('1', 1, 2);
   D2 := new BaseTask('2', 2, 3);
end T3;
```

Komentāri:

1. Tips *DynTask* ir norāžu tips.
2. Uzdevumus izveido programmā ar ģeneratora **new** palīdzību.
3. Programmā izmanto bāzes tipu *BaseTask*.

Satikšanās: *viena* ieeja, *vairāki* pieņemšanas operatori.

```
task T is          -- interfeiss
   entry E;
end T;
task body T is  -- realizācija
begin
   loop
      accept E do
         Put_Line("E1");
         delay 0.2;
      end E;
      accept E do
         Put_Line("E2");
         delay 2.0;
      end E;
   end loop;
end T;
```

Galvenā programma:

```
loop
    T.E;
end loop;
```

Rezultāts:

```
E1
E2
E1
E2
E1
E2
```

Šis rezultāts *nav atkarīgs* no aiztures jebkurā pieņemšanas operatorā.

Abi pieņemšanas operatori izpildās *pēc kārtas*.

Pieņemšanas operatora lietošana selektīvajā operatorā: labāk *minimizēt* **accept** daudzumu.

Lai ir Būla mainīgais `Flag`. *Nav* ieteicams:

```
if Flag then
   accept Info;
   -- viena apstrāde
else
   accept Info;
   -- cita apstrāde
end if;
```

Ir ieteicams:

```
accept Info;
if Flag then
   -- viena apstrāde
else
   -- cita apstrāde
end if;
```

Savas ieejas izsaukums

Interfeiss:

```
task T is
   entry E;
end T;
```

Realizācija:

```
task body T is
begin
   T.E;
   accept E do
      Put_Line("E");
   end E;
end T;
```

Rezultāts: *strupceļš*.

Uzdevuma eksistēšanas pārbaude (atribūts **`T'Callable`**).

1. Uzdevumu *deklarācija*.

```
with Text_IO;
use Text_IO;

procedure Main is

   package Bool_IO is new Enumeration_Io(Boolean);
   use Bool_IO;

   task type T;
   task body T is
   begin
      delay 0.0001;
      Put_Line("Task finished.");
   end T;
   T1, T2 : T;
```

2. Uzdevuma eksistēšanas pārbaude. *Galvenā* programma.

```
begin
   Put(T1'Callable); New_Line;
   Put(T2'Callable); New_Line;
   while T1'Callable or T2'Callable loop
      Put("Waiting...");
   end loop;
   Put(T1'Callable); New_Line;
   Put(T2'Callable); New_Line;
end Main;
```

```
TRUE
TRUE
Waiting...Waiting...Waiting...Task finished.
Waiting...Waiting...Task finished.
FALSE
FALSE
```

To pašu rezultātu var iegūt, izmantojot pakotni `Ada.Task_Identification`.

```
with Ada.Task_Identification;
use  Ada.Task_Identification;
...
package Bool_IO is new Enumeration_Io(Boolean);
use Bool_IO;
...
begin
   ...
   Put(Is_Callable(T1'Identity)); New_Line;
   Put(Is_Callable(T2'Identity)); New_Line;
```

Uzdevums: pārbaudīt uzdevuma `T` pabeigšanu (atribūts **`T'Terminated`**.

Lai uzdevumā `T` ir aizture:

```
delay 1.0;
```

Programmas fragments:

```
Put(T'Terminated); -- FALSE
delay 2.0;
Put(T'Terminated); -- TRUE
```

Iepriekšējā piemērā var izmantot atribūtu `Terminated`:

```
while not(T1'Terminated) or not(T2'Terminated)
loop
```

Var arī izmantot funkciju `Is_Terminated()`:

```
Put(Is_Terminated(T1'Identity));
...
Put(Is_Terminated(T2'Identity));
```

Atlases operatoru veidi:

✓ Operatori *ar gaidīšanu* (*selective wait*).

✓ Operatori ar *ieejas nosacīto izsaukumu* (*conditional entry call*).

✓ Operatori ar *kavējuma ieejas izsaukumu* (*timed entry call*).

✓ Operators ar asinhronu vadības nodošanu (*asynchronous transfer of control*).

Operatora robežas:

```
select
   ...
end select;
```

1. Operators *selective wait*.

```
select
   <atlases alternatīva>
or
   <atlases alternatīva>
[ else
   <operatoru secība> ]
end select;
```

Atlases alternatīva:

```
[ when nosacījums => ]
   <atlases alternatīva ar gaidīšanu>
```

Atlases alternatīva ar gaidīšanu:

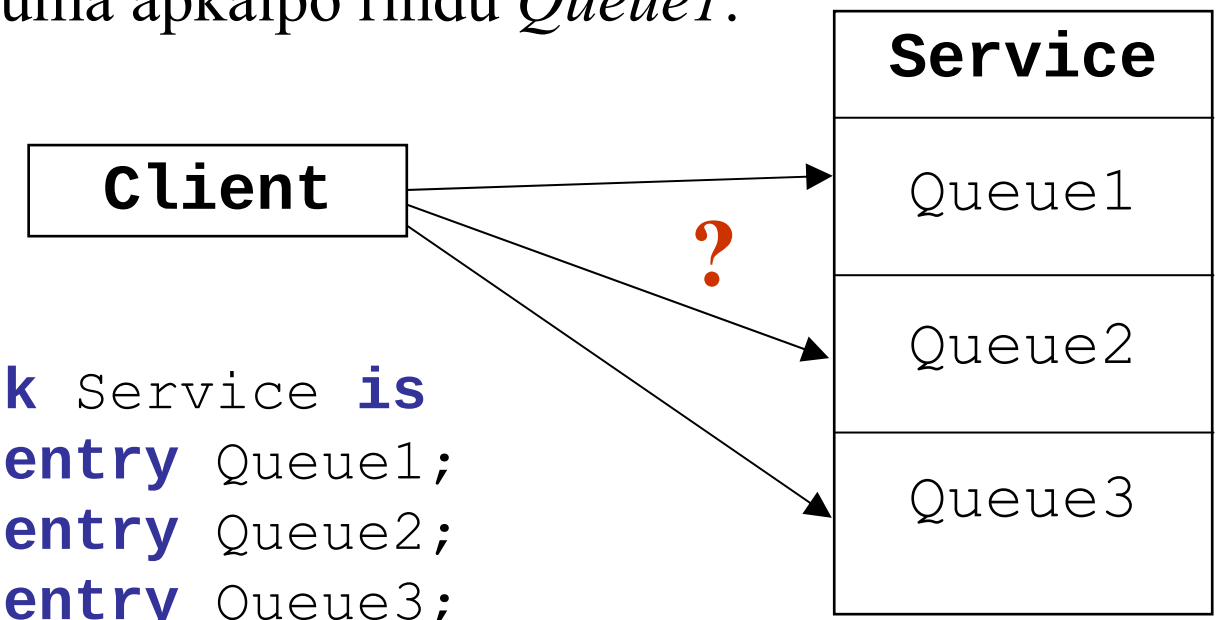
```
[ <pieņemšanas operators> [<operatoru secība>]
| <aiztures operators>    [<operatoru secība>]
| terminate
```

Parametra saņemšanas piemērs:

```
loop
   select
      accept Param (PNum : in Integer) do
         Num := PNum;
      end Param;
   or
      terminate;
   end select;
end loop;
```

Lai ir uzdevums *Service* ar ieejām *Queue**1***, *Queue**2***, *Queue**3***.

Ieejas apkalpo saskaņā ar to prioritātēm (1, 2, 3).
Sākumā apkalpo rindu *Queue1*.




```
task Service is
   entry Queue1;
   entry Queue2;
   entry Queue3;
end Service;

task Client;
```

Iespējama (bet ***ne****optimālā*) uzdevuma *Service* realizācija:

```
task body Service is
begin
   loop
      if Queue1'Count > 0 then
         accept Queue1;
         -- Put("Queue 1");
      elsif Queue2'Count > 0 then
         accept Queue2;
         -- Put("Queue 2");
      elsif Queue3'Count > 0 then
         accept Queue3;
         -- Put("Queue 3");
      end if;
   end loop;
end Service;
```

Problēma: uzdevums var tikt iznīcināts pēc veiksmīgas **if** pārbaudes, kad operators **accept** vēl *nav sasniegts*.

Uzdevums arī var *anulēt ieejas* izsaukumu līdzīgos apstākļos.

Labākais risinājums:

```
loop
   select
      accept Queue1;
         -- Put("Queue 1");
   or
      accept Queue2;
         -- Put("Queue 2");
   or
      accept Queue3;
         -- Put("Queue 3");
   end select;
end loop;
```

Lai ir uzdevums *Service* ar diviem pieņemšanas operatoriem **`accept`**, kas apkalpo vienu un to pašu ieeju (**`entry`**) *Section*.

Šo ieeju izsauc no uzdevuma *Client*.

Uzdevumu interfeisi:

```
task Client;
task Service is
   entry Section;
end Service;
```

Uzdevuma *Client* realizācija:

```
task body Client is
begin
   Service.Section;
end Client;
```

Uzdevuma *Client* realizācija:

```
task body Service is
   Flag : Integer := 1;
begin
   select
      when Flag=1 =>
         accept Section;
         Put("First section.");
   or
      when Flag=2 =>
         accept Section;
         Put("Second section.");
   else
      Put("Not accepted.");
   end select;
end Service;
```

1. Pašreizējais rezultāts: `First section`.

---

2. Lai mainīgajam `Flag` tika piešķirta vērtība 2.

```
Flag : Integer := 2;
```

Rezultāts: `Second section.`

---

3. Lai `Flag` ir 0 (vai 3, 4, …).

```
Flag : Integer := 0;
```

Rezultāts: `Not accepted.`

---

4. Lai uzdevumā *Client* pirms ieejas izsaukuma notiek kāda aizture:

```
delay 2.0;
Service.Section;
```

Rezultāts: `Not accepted.`

---

5. Lai uzdevumā *Service* nav zara **`else`**.

Viss ir atkarīgs no `Flag` vērtības. Var notikt izņēmums.

2. Operators *conditional entry call*.

```
select
   <ieejas izsaukums> [<operatoru secība>]
else
   <operatoru secība>
end select;
```

Mēģinājums *izdrukāt dokumentu*:

```
loop
   select
      Printer.Print(Id);
      exit;
   else
      -- citas lokālas darbības;
   end select;
end loop;
```

Izsaucošais uzdevums *negaida*, kad izsaucamais uzdevums pieņems ieejas izsaukumu, bet *turpina* darbu.

Citiem vārdiem sakot, ir iespēja *anulēt* nekorekto izsaukumu darba laikā.

Visoptimālākā mehānisma izmantošana - daudzkāršs cita uzdevuma izsaukums, lai beigu beigās iegūtu vajadzīgo informāciju (izpildītu vajadzīgo operāciju).

Operatora konstrukcija izskatās vienkāršāk, bez "`or`" daļas.

Uzdevums *Printer* var nekad neizpildīties, ja izsaucošam uzdevumam ar ciklu `loop` būs lielākā prioritāte.

Izsaucošais uzdevums vienkārši neatbrīvos apstrādājamo resursu.

3. Operators *timed entry call*.

```
select
   <pieņemšanas operators> [<operatoru secība>]
or
   <aiztures operators>    [<operatoru secība>]
end select;
```

Izsaucošais uzdevums gaidīs atbildi *noteiktu laiku*.
Nav vajadzīga *momentānā* reakcija.

Šo konstrukciju var izmantot iekārtās ar laika kontroli.

Lai vilciens atrodas ceļā. Vilciena brigādi regulāri kontrolē dispečeru dienests.

Kabīnē periodiski skan signāls. Mašīnista pienākums: reaģēt uz signālu *kādā laika periodā*.

Noteiktais rezervētais laika periods ir nepieciešams, jo mašīnists var būt *aizņemts ar citām lietām* – bremzēšanu, svarīgu sarunu.

Ja atbildēs *tomēr* nav, sistēmai jāpieņem kāds lēmums (automātiski paziņot par to ceļa distances priekšniekam, apstādināt vilcienu, vai kombinēt pirmo un otro darbību).

Vilciena mašīnista pārbaude:

```
loop
   select
      accept PressKey do
      end PressKey;
   or
      delay 3.0;
      raise DangerException;
   end select;
end loop;
```

4. Operators *asynchronous transfer of control*.

```
select
   <alternatīva pārslēgšanai>
then abort
   <pārtraucamā daļa>
end select;
```

Lai uzdevumam *Service* ir vienīga ieeja *Queue*. Pieņemšanas operators kopā ar kādu aizturi:

```
delay 3.0;
accept Queue;
```

Piezīme: operators *delay* imitē sistēmas noslodzi.

Ieeju *Queue* izsauc no uzdevuma *Client*.

Uzdevuma *Client* fragments:

```
select
   delay 2.0; -- 2.0 < 3.0
   Put("No service !");
then abort
   Service.Queue;
end select;
```

Rezultāts:

```
No Service !
```

Lai uzdevumā *Service* ir fragments:

```
delay 1.0; -- 2.0 > 1.0
accept Queue;
```

Rezultāts: notiek satikšanās.
Ziņojums `No Service !` netiks izvadīts.

Operatoru var izmantot *bez* ieejas izsaukuma:

```
select
   delay 2.0;
   Put("No result !");
then abort
   -- funkcijas izsaukuma imitēšana
   delay 3.0; --(*)
   Put("Done !");
end select;
```

Rezultāts:

```
No result ! -- 3.0 > 2.0
```

Lai rindiņā `(*)` norādīta aizture `1.0`. Rezultāts:

```
Done ! -- 1.0 < 2.0
```

Prioritātes

Katram uzdevumam var piešķirt prioritāti, izmantojot speciālo mehānismu "**pragma**".

```
-- aktuālā uzdevuma prioritāte ir 5
pragma Priority(5);
```

Piemērs:

```
task T is
   pragma Priority(8);
end T;
```

Minimālās un *maksimālās* prioritātes noteikšana:

```
with System;
...
MinP:Integer := System.Any_Priority'First; -- 0
MaxP:Integer := System.Any_Priority'Last;  -- 31
```

Mūsu gadījumā *nav* iespējamas programmas rindiņas:

```
pragma Priority(32); -- kļūda
pragma Priority(-1); -- kļūda
```

Rezultāts – kompilatora ziņojums:

```
„Constraint_Error” will be raised at run time
```

Prioritāte Ada'83 standartā: *tikai statiskā*.

Prioritāte Ada'95 standartā: statiskā un *dinamiskā* (pakotne `Ada.Dynamic_Priorities`).

```
with Ada.Dynamic_Priorities;
use Ada.Dynamic_Priorities;
...
task T;
task body T is
begin
   Set_Priority(1);
end T;
```

Prioritāšu *efektivitāte* – uzdevuma interfeiss:

```
N : constant Integer := 20;
task type T (Prior : Integer; C : Character) is
   pragma Priority(Prior);
end T;
```

Uzdevuma realizācija – simbolu izvade ciklā:

```
task body T is
begin
   for i in 1..N loop
      Put(C);
   end loop;
end T;
```

Uzdevuma eksemplāru radīšana:

```
T1: T(0, '1');
T2: T(10, '2');
```

Iespējamie rezultāti:

```
22222222222222222211111111111111111111
```

Lai uzdevumu prioritātes tika "*apmainītas*":

```
T1: T(10, '1');
T2: T(0, '2');
```

Iespējamie rezultāti:

```
11111111111111111122222222222222222222
```

Lai uzdevumu prioritātes *sakrīt*:

```
T1: T(10, '1');
T2: T(10, '2');
```

Iespējamie rezultāti:

```
22222222222222221111111111111111111122
```

Lai ciklā ir *nulles aizture* (`delay 0.0;`). Pēdējais rezultāts:

```
21212121212121212121212121212121212121
```

Uzdevumu *masīvs*. Uzdevumu *rinda* un atribūts **E'Count**.
Automašīnu masīvs un apkalpošanas sistēma.

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;

procedure Main is
   task type TAuto;

   task Service is
      entry Registration;
   end Service;

   task body TAuto is
   begin
      Service.Registration;
   end TAuto;

   Autos : array(1..5) of TAuto;
```

Automašīnu masīvs un apkalpošanas sistēma. Rindas *garums*.

```
   task body Service is
   begin
      loop
         accept Registration;
         Put("Total waiting: ");
         Put(Registration'Count, 2);
         New_Line;
         delay 1.0;
      end loop;
   end Service;
begin
   null;
end Main;
```

Automašīnu masīvs un apkalpošanas sistēma. Rezultāti.

```
Total waiting:  0
Total waiting:  3
Total waiting:  2
Total waiting:  1
Total waiting:  0
```

Lai uzdevumā *Service* pirms **`loop...end loop`** ir aizture 1 sekunde.

```
Total waiting:  4
Total waiting:  3
Total waiting:  2
Total waiting:  1
Total waiting:  0
```

Aizsargātie tipi

Aizsargāts objekts nodrošina *koordinēto piekļūšanu* datiem, izmantojot aizsargātās operācijas.

Tādi tipi ir *pasīvie* datu objekti, kuri nodrošina datu integritātes aizsardzību (piemēram, kad daudzie uzdevumi mēģina piekļūt datiem).

Aizsargātos tipus var saprast arī kā ļoti pastiprinātu semaforu vai monitoru formu.

Aizsargātos tipus visbiežāk organizē pārtraukumu aizliegšanai, bloķēšanas organizēšanai, un tā tālāk.

Ir *trīs* aizsargātie mehānismi.

1. Aizsargātās *funkcijas*.

Piekļūšana iekšējiem datiem režīmā "*tikai lasīšanai*".

Var tikt izsauktas no *daudziem uzdevumiem* vienlaicīgi.

---

2. Aizsargātās *procedūras*.

Piekļūšana iekšējiem datiem režīmā "*lasīšana/ierakstīšana*".

Var būt vienlaicīgi izsaukta tikai no *viena uzdevuma*.

Citiem vārdiem sakot, šis mehānisms nodrošina ekskluzīvu piekļūšanu datiem.

Aizsargātie mehānismi sastāv no:

1. Specifikācijas.
2. Ķermeņa.

Uzdevums: aizsargāt resursu no konkurējošiem pavedieniem.

Tikai viens pavediens var lasīt resursa vērtību (vai modificēt vērtību).

```
-- funkcijas un procedūras deklarācija
protected Resource is
   function Read return Integer;
   procedure Write(P: in Integer);
-- resursa vērtība
private
   Value : Integer := 0;
end Resource;
```

```
-- lasīšanas funkcijas realizācija
protected body Resource is
   function Read return Integer is
   begin
      return Value;
   end Read;
   -- ierakstīšanas procedūras realizācija
   procedure Write(P: in Integer) is
   begin
      Value := P;
   end Write;
end Resource;
```

Konkurējošo pavedienu fragments:

```
Resource.Write(2);
...
Put(Resource.Read);
```

Piezīme: funkcijā *Read* **ne**var eksistēt fragments:

```
Value := 2;
```

Rezultāts: ziņojums par kompilācijas kļūdu.

```
Protected function cannot modify protected object.
```

Aizsargātajā tipā var eksistēt *diskriminanta* daļa.

Piemēram, var inicializēt resursu:

```
protected type Resource(Start: Integer) is
...
private
   Value : Integer := Start;
end Resource;
```

Turpmākās operācijas:

```
R : Resource(0);
...
R.Write(i);
...
Put(R.Read);
```

3. Aizsargātās *ieejas*.

Šis mehānisms ir ļoti līdzīgs aizsargāto procedūru mehānismam.

Šajā gadījumā ir ieviesta papildu barjera - *Būla* izteiksme.

Šī barjera visbiežāk atkarīga no kāda iekšējā mainīgā, kurš iekapsulēts konkrētajā aizsargātajā tipā.

Aizsargātās ieejas. Tīkla printeris (interfeiss).

```
-- ieejas deklarācija
protected type NetPrinter is
   entry Print;
   procedure FreeNetPrinter;
private
   Printing : Boolean := False;
end NetPrinter;
```

Aizsargātās ieejas. Tīkla printeris (realizācija un mainīgais).

```
-- ieejas kontrole
protected body NetPrinter is
   entry Print when not Printing is
   begin
      Printing := True;
   end Print;
   procedure FreeNetPrinter is
   begin
      Printing := False;
   end FreeNetPrinter;
end NetPrinter;

HP_LaserJet : NetPrinter;
...
HP_LaserJet.Print;

HP_LaserJet.FreeNetPrinter;
```

Sinhronizācijas paņēmieni. *Signāli*

Signāli ir zemā līmeņa rīki paralēlo procesu sinhronizācijai.

Ja signāls tika nosūtīts *vienam no procesiem*, kurš gaida šo signālu, tad process turpinās savu darbu.

Ja neviens no procesiem negaida signālu, signāls tiks pazaudēts.

```
-- signāla interfeiss
task Signal is
   entry Wait;
   entry WakeUp;
end Signal;
```

Ja izsaucošais uzdevums izsauks ieeju *Wait*, tad pēc tam būs jāgaida uzdevuma ieejas *WakeUp* izsaukšana.

Ja neviens uzdevums neatrodas gaidīšanas stāvoklī, ieejas *WakeUp* izsaukuma pieņemšana nedos nekāda efekta.

```
-- signāla realizācija
task body Signal is
begin
   loop
      accept WakeUp;
      if (Wait'Count>0) then
         accept Wait;
      end if;
   end loop;
end Signal;
```

Labākais risinājums:

```
select
   accept Wait;
else
   null;
end select;
```

Signāli uz aizsargātā tipa pamata

```
-- deklarācija: ieeja un procedūra
protected type Signal is
   entry Wait;
   procedure WakeUp;
-- mainīgais - karodziņš
private
      Flag : Boolean  := False;
end Signal;

-- realizācija: ieeja
protected body Signal is
   entry Wait when Flag is
   begin
      Flag := False;
   end Wait;
```

```
    -- realizācija: atbrīvošanas procedūra
    procedure WakeUp is
    begin
       if Wait'Count>0 then
          Flag := True;
       end if;
    end;
end Signal;
```

Piezīme: procedūru *WakeUp* var noformēt kā ieeju.

```
entry WakeUp; -- deklarācijas fragments

-- realizācijas fragments
entry WakeUp when True is
```

Signāla piemēra *radīšana*.

```
S : Signal;
```

Uzdevuma *pamatprincips*: pabeigšana iespējama tikai pēc signāla saņemšanas.

```
task T;                 -- deklarācija
task body T is          -- realizācija
begin
    Put("T Start.");
    S.Wait;
    Put("T Finish.");
end T;
```

Galvenās programmas fragments:

```
delay 2.0;
S.WakeUp;
```

Rezultāts:

```
T Start.<gaidīšana>T Finish.
```

Bez signāla tiktu izvadīts *tikai pirmais teikums*.

Uzdevums *T* netiks pabeigts un paliks gaidīšanas stāvoklī.

Elementārais *semafors*:

```
-- deklarācija
task type Semaphore is
   entry P;
   entry V;
end Semaphore;
-- realizācija
task body Semaphore is
begin
   loop
      accept P;
      accept V;
   end loop;
end Semaphore;
Sem1, Sem2: Semaphore;
…
Sem1.P;...; Sem1.V;
```

Semafors uz *aizsargātā tipa* pamata:

```
-- deklarācija
protected type BinSem is
   entry P;
   entry V;
private
   Locked : Boolean := false;
end BinSem;

-- ieejas P realizācija
protected body BinSem is
   entry P when not Locked is
   begin
      Locked := true;
   end P;
```

```
    -- ieejas V realizācija
    entry V when True is
    begin
        Locked := false;
    end V;
end BinSem;

-- semafora radīšana
Sem : BinSem;


-- semafora lietošana
Sem.P;
...
Sem.V;
```

Semafors ar *n* uzdevumiem uz aizsargātā tipa pamata.

```
-- deklarācija
protected type Sem(Max : Integer) is
   entry P;
   entry V;
private
   Curr : Integer := Max;
end Sem;
```

```
-- realizācija
protected body Sem is
   entry P when (Curr > 0) is
   begin
      Curr := Curr - 1;
   end P;

   entry V when True is

   begin
      if  Curr < Max then
         Curr := Curr + 1;
      end if;

   end V;
end Sem;
```

Izņēmums *Tasking_Error*

Izņēmums var tikt ierosināts, ja satikšanās procesā izpildās operators **abort**.

Lai ir divi uzdevumi: izsaucošais *First* un izsaucamais *Second*.

Uzdevumu *interfeisi*:

```
task First;
task Second is
   entry Info;
end Second;
```

Uzdevuma *Second* pabeigšana notiks pieņemšanas operatorā *Info*.

Izņēmuma apstrāde notiks uzdevumā *First*.

Uzdevuma *First* realizācija

```
-- ziņojumu izvade un satikšanās
task body First is
begin
   Put("Start.");
   Second.Info;
   Put("Finish.");
-- izņēmuma apstrāde
exception
   when Tasking_Error =>
      Put("Tasking Error.");
end First;
```

Piezīme: var arī tvert visus iespējamus izņēmumus vienā blokā.

```
when others =>
   Put("Error.");
```

Uzdevuma *Second* realizācija

```
task body Second is
begin
   accept Info do
      abort Second;
   end Info;
end Second;
```

Rezultāts:

```
Start.Tasking Error.
```

Lai pirmajā uzdevumā *nav* izņēmuma apstrādes. Rezultāts:

```
Start.
```

Lai pieņemšanas operatorā ir **`null`**, nevis **`abort`**. Rezultāts:

```
Start.Finish.
```

Uzdevuma tips ir *limitētais* (**`limited private`**) tips.

Nav iespējams *piešķirt* uzdevumus vai *salīdzināt* tos.

Elementārais uzdevums: interfeiss un realizācija.

```
task type T;
task body T is
begin
   null;
end T;
```

Uzdevuma piemēri:

```
T1, T2: T;
```

Nav iespējams:

```
T1 := T2;
```

```
left hand of assignment must not be limited type
```

Uzdevuma *identifikators*

```
T1'Identity, T2'Identity
```

Atribūta tips: *Task_ID* no pakotnes *Ada.Task_Identification*.

Identifikatora izvade ekrānā:

```
with Text_IO, Ada.Task_Identification;
use Text_IO, Ada.Task_Identification;

...
Put_Line(Image(T1'Identity)); --t1_00284B90
Put_Line(Image(T2'Identity)); --t2_00287A58
```

Vēl viena iespēja pārtraukt uzdevuma izpildi:

```
Abort_Task(T1'Identity);
Abort_Task(T2'Identity);
```

Aktuālā uzdevuma noteikšana

Lai galvenajā programmā un uzdevumā ir rindiņa:

```
Put_Line(Image(Current_Task));
```

Current_Task atgriež aktuālā uzdevuma identifikatoru.

Rezultāti:

```
t1_00284B90
main_task_00283DF0
t2_00287A58
```

Izsaucošā uzdevuma noteikšana

```
accept E do
   Put_Line(Image(E'Caller));--main_task_00283DF0
end E;
```

Galvenās programmas fragments:

```
Server.E;
```

Pavediena aizture *līdz norādītajam laikam*

1. Pakotnes *Ada.Calendar* lietošana.

```
with Ada.Calendar;
use Ada.Calendar;
```

2. Aiztures operators.

```
delay until Time_Of(2009, 10, 26, 22.00*3600);
```

Funkcijas formāts: gads, mēnesis, diena, sekundes.

Aktuālais laiks: funkcija *Clock*.

Aiztures laiks: viena sekunde.

```
Tm : Time;
...
Tm := Clock;
delay until Time_Of(
   Year(Tm), Month(Tm), Day(Tm), Seconds(Tm)+1.0);
```

Paralēlā programmēšana Java valodā

Programmā *vienmēr* eksistē *vismaz viens* pavediens.

Pavediena "kontroles panelis" ("vadības pults") ir *Thread* klase.

```
class SimpleThread {
   public static void main(String args[]) {
      System.out.println("Thread info: " +
         Thread.currentThread());
      System.out.println("Total threads: " +
         Thread.activeCount());
   }
}
//Thread info: Thread[main,5,main]
//Total threads: 1
```

Pavediena aizture – bez izņēmuma apstrādes notiks *kompilācijas kļūda*:

```
try {
    Thread.sleep(1000); //milisekundes
}
    catch(InterruptedException e) {
    }

// var izmantot arī
// Thread.currentThread().sleep(1000);
```

Pavedienu izveidošanas iespējas:

- ✓ Mantošana no *Thread* klases
- ✓ *Runnable* interfeisa realizācija

Mantošana no *Thread* klases:

```
class <pavediens> extends Thread {
    ...
    public void run() {
        <pavediena darbības>
    }
}
```

Runnable interfeisa realizācija:

```
class <pavediens> implements Runnable {
    ...
    public void run() {
        <pavediena darbības>
    }
}
```

Šis variants ir *ievērojami populārāk*.

Paralēlā ziņojumu izvade: vadlīnijas.

1. Metodes *run()* iekšā ir izpildāmais pavediena kods.

2. Klases iekšā rāda pavedienu objektu *Thread*.

3. Ir divi populārie konstruktori:

```
Thread(Runnable <pavediena objekts>)
Thread(Runnable <pavediena objekts>,
    String <pavediena vārds>)
```

Piezīme: pirmais parametrs visbiežāk ir **`this`** (aktuālā klase).

Runa ir par vietu, kur sāksies pavediena izpilde.

4. Jaunais pavediens tiks palaists pēc metodes *start()* palaišanas.

5. Metodes *start()* izsauc metodi *run()*.

Paralēlā ziņojumu izvade: mantošana no *Thread*.

1. Klase ziņojuma izvadei.

```
class MyThread extends Thread {
    String Message;
    public MyThread(String S) {
        Message=S;
    }
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(Message);
            ...Thread.sleep(0);... //pseidokods
        }
    }
}
```

2. Galvenā programma. Pavedienu radīšana.

```
public class DemoThreads {
    public static void main(String args[]) {
        ...
        (new MyThread("FIRST Thread")).start();
        (new MyThread("SECOND Thread")).start();
        ...
    }
}
```

3. Rezultātu fragments.

```
FIRST Thread
SECOND Thread
FIRST Thread
SECOND Thread
...
```

Piezīme: šis piemērs **ne**strādās, ja klasē tiks realizēts *Runnable*.

Paralēlā ziņojumu izvade: *Runnable* realizācija.

1. Klase - pavedienu pamats. *Atribūti un konstruktors*.

```
class MyThread implements Runnable {
    final static int N = 4;
    Thread T;     //pavediens
    String Name; //pavediena vārds
    int Delay;    //aizture

    public MyThread(String Text, int Prior,
       int Delay) {
           Name = Text;
           this.Delay = Delay;
           T = new Thread(this);
           T.setPriority(Prior);
           T.start(); //izsaukt metodi run()
    }
```

2. Klase - pavedienu pamats. *Pavediena darbības*.

```
    public void run() {
        for (int i=0; i<N; i++) {
            System.out.println(Name + " Thread");
            try {
                T.sleep(Delay);
            }
                catch(InterruptedException e) {
                }
        }
    }
}
```

Komentāri:

1. Metode *run()* noteic pavediena *aktivitātes*.
2. Metode *start()* nodrošina pavediena *palaišanu*.

3. Pavedienu *radīšana*.

```
public class TwoThreads {
    public static void main(String args[]) {
        new MyThread("FIRST", 3, 0);
        new MyThread("SECOND", 3, 0);
    }
}
```

4. *Iespējamie* rezultāti.

```
FIRST Thread
SECOND Thread
FIRST Thread
SECOND Thread
FIRST Thread
SECOND Thread
FIRST Thread
SECOND Thread
```

5. Citi *prioritātes* un *aiztures* parametri.

```
new MyThread("FIRST", 6, 0);
new MyThread("SECOND", 3, 0);
```

Rezultāti:

```
FIRST Thread
FIRST Thread
FIRST Thread
FIRST Thread
SECOND Thread
SECOND Thread
SECOND Thread
SECOND Thread
```

Pavediena palaišana galvenajā programmā, *pēc radīšanas*.

1. Izmaiņas klasē *MyThread*: jauna metode *start()*.

```
class MyThread implements Runnable {
    ...
    public void start() {
        T.start();  // rindiņa no konstruktora
    }
}
```

2. Izmaiņas galvenajā programmā:

```
MyThread First, Second;
First = new MyThread("FIRST", 3, 0);
Second = new MyThread("SECOND", 3, 0);

...
First.start();
Second.start();
```

Pavediena palaišana galvenajā programmā, *pēc radīšanas*.
Otrais variants.

1. Izmaiņas klasē *MyThread*: jauna metode *getThread ()*.

```
class MyThread implements Runnable {
    ...
    public Thread getThread() {
        return T; //atgriezt pavedienu
    }
}
```

2. Izmaiņas galvenajā programmā:

```
MyThread First, Second;
First = new MyThread("FIRST", 3, 0);
Second = new MyThread("SECOND", 3, 0);
...
First.getThread().start();
Second.getThread().start();
```

Pavedienu prioritātes:

```
System.out.println(Thread.MIN_PRIORITY); //1
System.out.println(Thread.MAX_PRIORITY); //10
System.out.println(Thread.NORM_PRIORITY);//5
```

Prioritāšu diapazons: **[1, 10]**.

Nepareiza prioritāte: izņēmums
`IllegalArgumentException`.

Izmainīt aktuālā pavediena *vārdu*:

```
Thread.currentThread().setName("Main");
```

Izmainīt aktuālā pavediena *prioritāti*:

```
Thread.currentThread().setPriority(9);
```

Neaizsargātais resurss un *konkurējošie* pavedieni.

1. Klase ar *neaizsargāto* metodi:

```
class P {
    static public void printMessage() {
        final String S = "Hello, user !";
        for(int i=0; i<S.length(); i++){
            System.out.print(S.charAt(i));
            try {
                Thread.sleep(2);
            }
            catch(InterruptedException e) {
            }
        }
        System.out.println();
    }
}
```

2. Klase – pavedienu pamats:

```
class MyThread implements Runnable {
    final static int N = 2;
    Thread T;
    int Delay;
    public MyThread(int Prior) {
        T = new Thread(this);
        T.setPriority(Prior);
        T.start();
    }
    public void run() {
        for (int i=0; i < N; i++) {
            P.printMessage();
        }
    }
}
```

3. Pavedienu izveidošana un resursa izmantošana:

```
public class Threads {
    public static void main(String args[]) {
        new MyThread(5);
        new MyThread(8);
    }
}
```

---

4. *Iespējamie* (bet nevienīgie !) rezultāti:

```
HHelellol,o ,u suesre r!
H!e
Hlellol,o ,u suesre r!
!
```

Aizsargātais resurss: rezervētais vārds *synchronized*.

Klase ar *aizsargāto* metodi:

```
class P {
    synchronized static public void
        printMessage() {
        ...
    }
}
```

Vienīgi iespējamie rezultāti:

```
Hello, user !
Hello, user !
Hello, user !
Hello, user !
```

Alternatīvais risinājums: var sinhronizēt *koda bloku*.

1. Informācijas izvades metode *nav* statiskā.

```
class P {
    public void printMessage() {
      …
    }
}
```

2. Objekts informācijas izvadei tiks nodots pavedienam kā parametrs.

```
class MyThread implements Runnable {
    …
    P Pr;
    public MyThread(int Prior, P PrObj) {
        Pr = PrObj;
        …
    }
}
```

3. Izmaiņas metodē *run()*.

```
public void run() {
    synchronized (Pr) {
    for (int i=0; i < N; i++) {
            Pr.printMessage();
        }
    }
}
```

Sinhronizēt var *objektu*, nevis klasi.

4. Galvenās programmas fragments

```
...
P PrObj = new P();
new MyThread(5, PrObj);
new MyThread(8, PrObj);
```

Abi pavedieni strādā ar *vienu un to pašu* objektu.

Pavedienu *apvienošana*

Lai jaunie pavedieni izveidoti galvenajā pavedienā.
Tādus pavedienus sauc par *meitas pavedieniem*.

Tad lietderīgi:
a. Pabeigt visu *meitas pavedienu* darbu.
b. Pabeigt *galvenā pavediena* darbu.

1. Metode *join()* atļauj gaidīt visu meitas pavedienu pabeigšanu.

```
public final void join()
    throws InterruptedException
```

Metodi *join()* izmanto kopā ar izņēmuma *InterruptedException* apstrādātāju.

2. Metode *isAlive()* pārbauda pavediena eksistēšanu.

```
public final boolean isAlive()
```

Uzdevums:

1. Galvenajā pavedienā palaist divus meitas pavedienus.
2. Abi pavedieni izvada ekrānā savu vārdu.
3. Pabeigt galveno pavedienu tikai pēc visu meitas pavedienu pabeigšanas.

```
class MyThread implements Runnable {
    Thread T; // pavediena deklarācija

    /* pavediena vārda norādīšana */
    public MyThread(String Name) {
        T = new Thread(this);
        T.setName(Name);
    }
```

```
    /* pavediena saņemšana */
    public Thread getThread() {
        return T;
    }

    /* pavediena vārda drukāšana */
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(
                Thread.currentThread());
        }
    }

/* klases pabeigšana */
}
```

```
/* divu pavedienu radīšana */
public class Demo {
   public static void main(String args[]) {
      MyThread T1, T2;
      T1 = new MyThread("First");
      T2 = new MyThread("Second");

      /* divu pavedienu palaišana */
      T1.getThread().start();
      T2.getThread().start();

      /* divu pavedienu statusa izvade */
      System.out.println("First thread: " +
         T1.getThread().isAlive());
      System.out.println("Second thread: " +
         T2.getThread().isAlive());
```

```
        /* divu pavedienu apvienošana */
        try {
            T1.getThread().join();
            T2.getThread().join();
        }
        catch(InterruptedException e) {
        }

        /* divu pavedienu statusa izvade */
        System.out.println("First thread: " +
            T1.getThread().isAlive());
        System.out.println("Second thread: " +
            T2.getThread().isAlive());

        /* klases pabeigšana */
        System.out.println("Done.");
    }
}
```

Rezultāts:

```
First thread: true
Second thread: true
Thread[First,5,main]
Thread[First,5,main]
Thread[First,5,main]
Thread[First,5,main]
Thread[First,5,main]
Thread[Second,5,main]
Thread[Second,5,main]
Thread[Second,5,main]
Thread[Second,5,main]
Thread[Second,5,main]
First thread: false
Second thread: false
Done.
```

Uzdevuma "*ražotājs-patērētājs*" risināšana

1. Ir divi pavedieni: *Producer* un *Consumer*.

2. Abi pavedieni strādā ar vienu un to pašu rindu *Queue*.

3. Ražotājs izvieto rindā informāciju *n* reizes; patērētājs sinhronizēti izmanto šo informāciju *n* reizes.

4. Ieplānotie sinhronizētie rezultāti:

```
Put:1 Get:1
Put:2 Get:2
Put:3 Get:3
Put:4 Get:4
Put:5 Get:5
```

1. Klase *Queue*

```
/* Saņemamā vērtība */
class Queue {
    int x;

    /* Saņemšanas metode */
    synchronized int get() {
        System.out.println("Get:" + x);
        return x;
    }

    /* Izvietošanas metode */
    synchronized void put(int x) {
        this.x = x;
        System.out.print("Put:" + x + " ");
    }
}
```

2. Klase "Ražotājs" (*Producer)*

```
/* Rindas deklarēšana */
class Producer implements Runnable {
    Queue q;
/* Rindas sasaistīšana un pavediena palaišana */
    public Producer(Queue q) {
        this.q = q;
        new Thread(this).start();
    }
/* Secīga informācijas izvietošana rindā */
    public void run() {
        for(int i=1; i<=5; i++) {
            q.put(i);
        }
    }
}
```

3. Klase "Patērētājs" (*Consumer)*

```
/* Rindas deklarēšana */
class Consumer implements Runnable {
    Queue q;
/* Rindas sasaistīšana un pavediena palaišana */
    public Consumer(Queue q) {
        this.q = q;
        new Thread(this).start();
    }
    /* Secīga informācijas saņemšana no rindas */
    public void run() {
        for(int i=1;i<=5;i++) {
            q.get();
        }
    }
}
```

4. Galvenā programma

```
/* Rindas, ražotāja un patērētāja radīšana */
public class T {
    public static void main(String [] args) {
        Queue q = new Queue();
        new Producer(q);
        new Consumer(q);
    }
}
```

5. *Iespējamie* rezultāti

```
Put:1 Put:2 Put:3 Put:4 Put:5 Get:5
Get:5
Get:5
Get:5
Get:5
```

Bieži “parastajā” programmā ir kāds cikls, kur notiek nosacījuma(-u) pārbaude.

Ja kāds nosacījums ir patiess, izpildās kāda darbība.

Blakus efekts: procesora laiks tiks iztērēts uz papildu operācijām.

Java atļauj **ne**izmantot šo ciklu. Superklasē *Object* ir *trīs* speciālās metodes pavedienu savstarpējai iedarbībai.

Visas trīs metodes var izsaukt no *jebkuras* klases.

Metodēm nepieciešamas **`synchronized`** konteksts.

Mūsu gadījumā nelietderīgi tērēt procesora laiku uz datu gaidīšanu.

1. `final void wait() throws InterruptedException`

Izsaucošais pavediens būs spiests atdot monitoru un *apstādināt izpildi* uz kādu laiku.

---

2. `final void notify()`

Turpināt *apstādināta* pavediena izpildi.

Metode tiks izsaukta no *cita pavediena*.

---

3. `final void notifyAll()`

Turpināt *visu apstādinātu pavedienu* izpildi.

Piezīme: ieteicams izsaukt *wait()* cikla iekšā.

Cikls pārbauda pavediena gaidīšanas nosacījumu.

Ir varbūtība, ka pavedienu "pamodinās" ar viltotu signālu.

Klase "*Queue*": pareiza realizācija

```
/* Saņemamā vērtība un karodziņš */
class Queue {
    int x;
    boolean IsValue = false;
    /* Saņemšanas metode un ieeja ciklā */
    synchronized int get() {
        while (!IsValue) {
            /* Pavediena aizturēšana līdz
               ziņojumam no Producer */
            try {
               wait();
            }
               catch(InterruptedException e) {
               }
        }
```

```
        System.out.println("Get:" + x);
        IsValue = false;

        /* Pavediena darbības turpināšana */
        notify();
        return x;
    }

    /* Izvietošanas metode un ieeja ciklā */
    synchronized void put(int x) {
        while (IsValue) {
            try {
                wait();
            }
                catch(InterruptedException e) {
                }
        }
```

```
        this.x = x;
        IsValue = true;
        System.out.print("Put:" + x + " ");

        /* Pavediena darbības turpināšana */
        notify();
    }
}
```

Vienīgi pareizie rezultāti:

```
Put:1 Get:1
Put:2 Get:2
Put:3 Get:3
Put:4 Get:4
Put:5 Get:5
```

Pavediena apstādināšana līdz kāda nosacījuma izpildei

Lai pavediens cikliski izvada kādu informāciju.

```
class PrintThread implements Runnable {
    Thread T;

    boolean Waiting = false;

    public PrintThread() {
        T = new Thread(this);
        T.start();
    }
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.print('1');
            <aizture>
```

```
        synchronized(this) {
            while (Waiting) {
                try  {
                    wait();
                }
                  catch(InterruptedException e) {}
            }
        }
    }
}

/* apstādināšanas metode */
public void Wait() {
    Waiting = true;
}
```

```
    /* turpināšanas metode */
    synchronized public void Notify() {
        Waiting = false;
        notify();
    }
}
```

Galvenās programmas fragments:

```
PrintThread PT = new PrintThread();
PT.Wait();
<aizture>
PT.Notify();
```

Rezultāti:

```
1<aizture>111111111
```

Java 5 satur paralēlas *utilītprogrammas*.

Dažreiz runā par *paralēlo API*.

Attiecīgas pakotnes:

1. `java.util.concurrent`
2. `java.util.concurrent.atomic`
3. `java.util.concurrent.locks`

Pakotnes `java.util.concurrent` analīze

Klase `Semaphore`

Klasiskā semafora implementēšana.

Semafora darbību pamats: *skaitītājs*.

Skaitītājs > 0 → piekļuve atļauta.

Skaitītājs = 0 → piekļuve aizliegta.

Skaitītājs saistīts ar attiecīgajām *atļaujām*.

Lai kāds pavediens mēģina piekļūt resursam:

1. Ja (*skaitītāja vērtība > 0)*, pavediens dabūs atļauju. Skaitītāja vērtība tiks *samazināta par 1*.

2. Pretējā gadījumā pavediens tiks nobloķēts līdz atļaujas saņemšanai.

3. Kad pavediens atbrīvo resursu, viņš atbrīvo atļauju. Skaitītāja vērtība tiks *palielināta par 1*.

4. Ja kāds pavediens gaida atļauju, viņš to dabūs.

Klases `Semaphore` konstruktori

1. `Semaphore(int N)`

N – skaitītāja vērtība

1 – ar resursu var strādāt *tikai viens pavediens*

---

2. `Semaphore(int N, boolean Order)`

Order – gaidošo pavedienu apstrādes kārtība.

`true` – rindas kartība (FIFO); **`false`** – kārtība nav noteikta.

Klases `Semaphore` atļaujas

1. `acquire() throws InterruptedException`

2. `acquire(int N) throws InterruptedException`

Pirmajā gadījumā pieprasa *vienu atļauju* (visizplatītākā forma).

Klases `Semaphore` atļauju atbrīvošanas

1. `void release()`
2. `void release(int N)`

Pirmajā gadījumā atbrīvo *vienu atļauju* (visizplatītākā forma).

Aizsargātā resursa izmantošana:

1. Izsaukt `acquire()`.
2. Darbs ar resursu.
3. Izsaukt `release()`.

Lai ir statiskā funkcija, kas izvada ekrānā teksta ziņojumu pa simboliem.

Funkciju izsauc vairāki uzdevumi, kas drukā savu vārdu.

1. Pakotnes importēšana

```
import java.util.concurrent.*;
```

2. Funkcijas fragments (pamatā ir *augstākminētais* piemērs)

```
class P {
    public static void printMessage(String S) {
    …
        System.out.print(S.charAt(i));
    …
    }
}
```

3. Pavediena klase

Atribūti: *pavediena vārds* un *sinhronizējošais semafors*.

```
class MyThread implements Runnable {
    String Name;
    Semaphore S;
```

4. Pavediena klase.

Pavediena *konstruktors*: radīšana un palaišana.

```
public MyThread(String Name, Semaphore S) {
    this.Name = Name;
    this.S = S;
    new Thread(this, Name).start();
}
```

5. Pavediena klase.

Pavediena *darbības* metodē *run()*.

Atļaujas pieprasījums.

```
public void run() {
    try {
        S.acquire();
    }
        catch (InterruptedException e) {
        }
```

6. Pavediena klase.

Pavediena *darbības* metodē *run()*.

Pavediena vārda izvade un izvades funkcijas atbrīvošana.

```
        P.printMessage(Name);
        S.release();
    }
}
```

7. Galvenā programma.

```
public class T {
    public static void main(String [] args) {
        Semaphore Sem = new Semaphore(1);
        MyThread MT1 = new MyThread
            ("First Task", Sem),
          MT2 = new MyThread("Second Task", Sem);
    }
}
```

8. Programmas rezultāti.

```
First Task
Second Task
```

Piezīme: lai programmā *netika* izmantots semafors. Rezultāti:

```
FSiercsotn dT aTsaks
k
```

Pieņemsim, kā pavedienā ir *aizture*. Ir vienots bloks **`try`**:

```
try {
    S.acquire(1);
    Thread.sleep(100);
    P.printMessage(Name);
    S.release(1);
}
```

Iegūsim *tos pašus* rezultātus.

Aktuālā pavediena gaidīšanas laikā resurss *tik un tā aizsargāts*.

Semaforu lietošana uzdevumā "*ražotājs-patērētājs*"

Galvenās domas:

1. Atteikties no papildus "karodziņiem" – *Bula mainīgajiem*.

2. Atteikties no *sinhronizētajām* metodēm.

3. *Neko nemainīt* klasēs "Consumer" un "Producer".

```
import java.util.concurrent.*;

class Queue {
    int x;
    // sākumā atvērts tikai ražotāja semafors
    // patērētājs spiests gaidīt
    static Semaphore SemProd = new Semaphore(1);
    static Semaphore SemCons = new Semaphore(0);
```

```
void get() {
    /* Ir garantija, ka metode get()
        izpildās tikai pēc metodes put() */
    // 1. Aizvērt patērētāja semaforu.
    // 2. Izvadīt vērtību.
    // 3. Atvērt ražotāja semaforu.
    try {
        SemCons.acquire();
    }
    catch(InterruptedException e) {
    }
    System.out.println("Get:" + x);
    SemProd.release();
}
```

```
    void put(int x) {
        // 1. Aizvērt ražotāja semaforu.
        // 2. Nodot un izvadīt vērtību.
        // 3. Atvērt patērētāja semaforu.
        try {
            SemProd.acquire();
        }
            catch(InterruptedException e) {
            }
        this.x = x;
        System.out.print("Put:" + x + " ");
        SemCons.release();
    }
}
```

Reakcija uz *vairākiem* notikumiem

Piemērs: *kodolieroča* izmantošana.

Nepieciešami *divi* signāli: no prezidenta un aizsardzības ministra.

Tādiem nolūkiem var izmantot slēdzeni `CountDownLatch` no paralēlā API.

Slēdzenes *konstruktors*:

```
CountDownLatch(int N)
```

`N` ir *notikumu daudzums* slēdzenes noņemšanai.

Pēc katra notikuma attiecīgo skaitītāju *samazina par vienu*.

Slēdzeni noņem, ja skaitītāja vērtība ir *vienāda ar 0*.

a. *Gaidīšanas* parametru noteikšana.

1. `void await() throws InterruptedException`

Gaidīt, kamēr attiecīgā skaitītāja vērtība *nav vienāda ar 0*.

2. `void await(long Wait, TimeUnit TU) throws InterruptedException`

Gaidīt stingri *noteikto laiku*.

Piezīme: `TimeUnit` ir uzskaitījums (*enumeration*) no pakotnes `java.util.concurrent`.

```
…
CDL.await(10, TimeUnit.SECONDS)
…
```

---

b. *Signalizēšana* par notikumu.

`void countDown()`

Samazināt notikumu skaitītāju *par vienu*.

Testpiemēra idejas:

1. Izveidot slēdzeni *galvenajā programmā*.

2. Izveidot "prezidenta" un "aizsardzības ministra" pavedienus *galvenajā programmā*.

3. Nodot abiem pavedieniem *norādi* uz slēdzeni.

```
import java.util.concurrent.*;

class Start implements Runnable {
   CountDownLatch CDL_Start;

   public Start(CountDownLatch CDL_Start) {
      this.CDL_Start = CDL_Start;
      new Thread(this).start();
   }
```

```
    public void run() {
        System.out.println("OK.");
        CDL_Start.countDown();
    }
}
public class Demo {
    public static void main(String [] args) {
        CountDownLatch CDL = new
            CountDownLatch(2); // divi notikumi

        System.out.println("Waiting...");
        new Start(CDL); // prezidents
        new Start(CDL); // aizsardzības ministrs

        /* Nākamais uzdevums: apstādināt galvenās
           programmas izpildi.
           Nepieciešami divi notikumi */
```

```
        try {
            CDL.await();
        }
            catch(InterruptedException e) {
            }
        System.out.println("Start.");
    }
}
```

Rezultāti:

```
Waiting...
OK.
OK.
Start.
```

Piezīme: lai konstruktorā `CountDownLatch` norādīts tikai viens notikums.

Ziņojums "`Start`" var būt *trešais* pēc kārtas.

Papildu informācija par `TimeUnit`.

Mērvienības:

1. `TimeUnit.NANOSECONDS` – *nano*sekundes.
2. `TimeUnit.MILLISECONDS` – *mili*sekundes.
3. `TimeUnit.MICROSECONDS` – *mikro*sekundes.
4. `TimeUnit.SECONDS` – sekundes.
5. `TimeUnit.MINUTES` – minūtes (Java 6).
6. `TimeUnit.HOURS` – stundas (Java 6).
7. `TimeUnit.DAYS` – dienas (Java 6).

Lai iepriekšējā piemērā norāda arī reakcijas laiku:

```
...
CDL.await(5, TimeUnit.SECONDS);
```

Rezultāti pilnīgi sakritīs, nekādas gaidīšanas *nebūs*.

Bez pavedienu (-a) palaišanas viss notiks automātiski, *pēc piecām sekundēm* (`Waiting... Start`).

Pavediena sinhronizācija ar citu pavedienu (-iem)

Lai kāds pavediens spiests gaidīt citu pavedienu.

Klase `CyclicBarrier`

Lietošanas mērķis: noteikt sinhronizācijas objektu.

a. Divi konstruktori:

1. `CyclicBarrier(int N)`

Parametrs: gaidāmo pavedienu daudzums.

2. `CyclicBarrier(int N, Runnable Action)`

Papildus parametrs: pavediens, kurš tiks izpildīts pēc barjeras sasniegšanas.

b. Sinhronizācijas procesā lieto metodi `await()`.

```
1. int await() throws InterruptedException,
       BrokenBarrierException
```

Gaidīt, kad visi pavedieni sasniegs barjeru.

```
2. int await(long Wait, TimeUnit TU)
       throws InterruptedException,
       BrokenBarrierException, TimeoutException
```

Gaidīt stingri noteikto laiku.

Abos gadījumos tiks atgriezta vērtība.

Pirmais pavediens: gaidāmo pavedienu daudzums mīnuss 1.
...
Pēdējais pavediens: 0.

Lai uzdevums sastāv *no divām daļām*.

Uzdevums ir pilnīgi izpildīts, tikai ja pabeigtas *abas* daļas.

```
import java.util.concurrent.*;
class Task implements Runnable {
    CyclicBarrier CB;
    public Task(String Name, CyclicBarrier CB) {
        this.CB = CB;
        new Thread(this, Name).start();
    }
    public void run() {
        System.out.println(Thread.currentThread());
        try {
            CB.await();
        }
            catch (InterruptedException e) {}
            catch (BrokenBarrierException e) {}
```

```
    }
  }
```

```
  class Finish implements Runnable {
    public void run() {
      System.out.println("Finish.");
    }
  }
```

Galvenās programmas fragments:

```
    CyclicBarrier CB = new
      CyclicBarrier(2, new Finish());
    System.out.println("Start.");
    new Task("First part", CB);
    new Task("Second part", CB);
```

```
Start.
Thread[First part,5,main]
Thread[Second part,5,main]
Finish.
```

Koncepciju var izmantot *pavedienu grupu* sinhronizēšanai

Lai galvenajā programmā tika izveidoti *vairāki* pavedieni.

```
new Task("First part", CB);
new Task("Second part", CB);
new Task("Third part", CB);
new Task("Fourth part", CB);
```

Rezultāts:

```
Start.

Thread[First part,5,main]
Thread[Second part,5,main]
Finish.

Thread[Third part,5,main]
Thread[Fourth part,5,main]
Finish.
```

Apmaiņa ar informāciju

Noskaņojamā klase `Exchanger<T>` pavienkāršo komunikāciju starp diviem pavedieniem.

Informācijas apmaiņai lieto *patvaļīgo* buferi.

Klasei ir vienīga metode `exchange(...)`.

1. `T exchange(T Buffer)`
   `throws InterruptedException`
2. `T exchange(T Buffer, long Wait, TimeUnit TU)`
   `throws InterruptedException,`
   `TimeoutException`

Pēdējā gadījumā var norādīt gaidīšanas laiku.

Metode tiks izpildīta, tikai ja to izsaucēja *divi* pavedieni.

Lai ir divi pavedieni:

1. Pavediens *Strings*. Satur teksta rindiņu masīvu.
   Izvada ekrānā teksta rindiņu pēc norādītā indeksa.
2. Pavediens *Indexes*. Satur indeksu masīvu.
   Piegādā indeksus pavedienam *Strings*.

```
import java.util.concurrent.*;
class Strings implements Runnable {
   Exchanger<Integer> E;
   Integer BufIndex;
   String S[] = {"a", "b", "c", "d", "e", "f"};
   public Strings(Exchanger<Integer> E) {
      this.E = E;
      new Thread(this).start();
   }
```

```
    public void run() {
        for(int i=0; i<5; i++) {
            try {
                BufIndex = E.exchange(0);
            }
                catch (InterruptedException e) {
                }
            System.out.println(S[BufIndex]);
        }
    }
}
```

```
class Indexes implements Runnable {
    Exchanger<Integer> E;

    Integer BufInd;
    int Ind[] =  {4, 2, 1, 0, 3};
```

```
    public Indexes(Exchanger<Integer> E) {
        this.E = E;
        new Thread(this).start();
    }

    public void run() {
        for(int i=0; i<5; i++) {
            BufInd = Ind[i];
            try {
                E.exchange(BufInd);
            }
                catch (InterruptedException e) {
                }
        }
    }
}
```

Galvenā programma

```
public class Demo {
    public static void main(String [] args) {
        Exchanger<Integer> Ex = new
            Exchanger<Integer>();
        new Strings(Ex);
        new Indexes(Ex);
    }
}
```

Rezultāti:

```
e
c
b
a
d
```

Pavedienu pūla radīšana

Koncepcijas pamatā ir jēdziens "*izpildītājs*".

---

1. Interfeiss `Executor`.

```
void execute(Runnable Thread)
```

Izpildās norādītais pavediens.

2. Interfeiss `ExecutorService`.
Paplašina interfeisu `Executor`.

Pievienotas vairākas metodes pavedienu kontrolēšanai.

Viena no svarīgākām metodēm: visu vadāmo pavedienu apstādināšana.

```
void shutdown()
```

Ar abiem interfeisiem saistīta *papildu* klase.

Klase `Executors`.

1. `static ExecutorService newFixedThreadPool(int N)`

Izveidot pavedienu pūlu no *N* pavedieniem.

2. `static ExecutorService newScheduledThreadPool(int N)`

Izveidot pavedienu pūlu no *N* pavedieniem, kur ir iespējami papildus plānošanas pasākumi.

Ir arī divas klases pavedienu radīšanai

1. `ThreadPoolExecutor`

Pūls no *N* pavedieniem.

2. `ScheduledThreadPoolExecutor`

Plānošanas nodrošināšana.

Lai ir pavediens, kas cikliski izvadā ekrānā noteiktu simbolu.

```
import java.util.concurrent.*;

class CharThread implements Runnable {
    char C;
    public CharThread(char C) {
        this.C = C;
    }
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.print(C);
            <kāda aizture>
        }
        System.out.println();
    }
}
```

Galvenā programma (`main(...)` funkcijas fragments)

```
ExecutorService ES =
   Executors.newFixedThreadPool(3); // (*)
ES.execute(new CharThread('1'));
ES.execute(new CharThread('2'));
ES.execute(new CharThread('3'));
ES.shutdown(); // citādi programma nav pabeigta
```

```
123123123123123123123123123123 un trīs "\n"
```

Lai pūlā ir *divi* pavedieni: rindiņā `(*)` ir **2**, nevis 3.

```
12121212121212121212
3
333333333
```

Lai pūlā ir *viens* pavediens.

```
1111111111
2222222222
3333333333
```

Atomāras operācijas

Pakotne `java.util.concurrent.atomic`.

Dažos gadījumos var sinhronizēt lasīšanu/ierakstīšanu *bez* speciālo mehānismu lietošanas.

Ir divas klases: `AtomicInteger` un `AtomicLong`.

Objektu metodes:

1. `get()` // vērtības *iegūšana*
2. `set(<vērtība>)` // vērtības *piešķire*
3. `decrementAndGet()` // *samazināt* par vienu un *iegūt* vērtību
4. `getAndSet(<vērtība>)` // *iegūt* un *piešķirt* vērtību
5. `compareAndSet(<vērtība>, <vērtība>)`
   // *salīdzināt* un *izmainīt* vērtību

Lai visur ir sākotnējā piešķire:

```
AtomicInteger AI = new AtomicInteger(0);
```

```
System.out.println(AI.decrementAndGet()); // -1
```

```
System.out.println(AI.getAndSet(1)); // 0
System.out.println(AI.get());        // 1
```

```
System.out.println(AI.compareAndSet(2, 1));
// false
System.out.println(AI.get()); // 0

System.out.println(AI.compareAndSet(0, 1));
// true
System.out.println(AI.get()); // 1
```

Testpiemēra uzmetums

```
import java.util.concurrent.atomic.*;

class SharedInt {
    static AtomicInteger AI =
        new AtomicInteger(0);
}

class TestAtomic implements Runnable {
    ...
    public void run() {
        for(int i=Start; i<=Finish; i++) {
            System.out.println(
                SharedInt.AI.getAndSet(i));
        ...
        }
    }
}
```

Savstarpēja iedarbība ar Java *tipiem* un *klasēm*

```
int i = SharedInt.AI.get();
SharedInt.AI.set(i+1);
System.out.println(SharedInt.AI);
// bez "get()"
```

```
Integer i = SharedInt.AI.get();
```

Tālāk bez izmaiņām.

Ārēja bloķēšana

Programmētājs pats norāda bloķēšanas *sākumu* un *galu*.

Tas zināmā mērā atšķir situāciju no **`synchronized`** bloka.

Pamatā – interfeiss `Lock`.

Kopā ar interfeisu visbiežāk lieto klasi `ReentrantLock`.

Galvenās metodes:

```
1. public void lock()    //bloķēšana
2. public void unlock() //atbloķēšana
```

Lai klasē *P* ir statiskā funkcija *printMessage()*, kas izvada ekrānā kādu ziņojumu pa simboliem (tika apskatītā augstāk).

Ar funkciju vienlaicīgi strādā *tikai viens* pavediens.

```
import java.util.concurrent.locks.*;

class MyThread implements Runnable {
    private Thread T;
    Lock L;

    public MyThread(Lock L) {
        this.L = L;
        T = new Thread(this);
        T.start();
    }
    public void run() {
        L.lock();
        P.printMessage();
        L.unlock();
    }
}
```

Galvenā programma

```
Lock L = new ReentrantLock();
new MyThread(L);
new MyThread(L);
```

Vienīgi pareizie rezultāti:

```
Hello, user !
Hello, user !
```

Var *pārbaudīt* bloķēšanu. Piemēram, var saskaitīt atteikumus.

3. `public boolean tryLock()`

```
class MyThread implements Runnable {
   int Deny = 0; // vienā gadījumā paliks 0
   …
   public void run() {
      while (!L.tryLock())
         Deny++;
         …
```

Ir iespēja atdalīt *ierakstīšanas* un *lasīšanas* bloķēšanas.

Vairāki pavedieni var *lasīt* informāciju.

Tikai viens pavediens var *rakstīt* informāciju.

Pamatā – interfeiss `ReadWriteLock`.
Kopā ar interfeisu visbiežāk lieto klasi `ReentrantReadWriteLock`.

Lai iepriekšējā piemērā izpildītas izmaiņas

```
import java.util.concurrent.locks.*;

class MyThread implements Runnable {
   private Thread T;
   ReadWriteLock RWL;
   ...
```

```
    public MyThread(ReadWriteLock RWL) {
        this.RWL = RWL;
        ...
    }
    public void run() {
        RWL.writeLock().lock();
        P.printMessage();
        RWL.writeLock().unlock();
    }
}
...
    ReadWriteLock RWL = new
        ReentrantReadWriteLock();
    new MyThread(RWL);
    new MyThread(RWL);
```

```
Hello, user !
Hello, user !
```

Lai `writeLock()` vietā lieto `readLock()`.

```
    public void run() {
        RWL.readLock().lock();
        P.printMessage();
        RWL.readLock().unlock();
    }
}
```

Rezultātu *nevar prognozēt*

```
HeHellllloo,,  uusseerr  !!
```

Piezīme: nepieciešams ievērot `lock()`/`unlock()` *bilanci*.

Lai bloķēšanas nav, bet ir tikai *atbloķēšana*.

Rezultāts: *izņēmuma* ierosināšana programmas izpildes laikā.

Vērtību atgriešana galvenajām pavedienam

1. *Noskaņojamais* interfeiss `Callable<T>`.

Realizācijas gadījumā *veido pavedienu*, kas atgriež vērtību.

Interfeisā deklarēta tikai viena metode:

```
public T call()
```

Pavedienu palaiž ar interfeisa `ExecutorService` palīdzību:

```
<T> Future<T> submit (Callable<T>)
```

---

2. *Noskaņojamais* interfeiss `Future<T>`.

Saņem rezultātu, iegūto pateicoties `Callable`.

Bieži lietojamā metode:

```
public T get() throws
    InterruptedException, ExecutionException
```

Pavienkāršotais izpildes algoritms *dažādiem N* uzdevumiem

1. Izveidot *N* klases ar realizēto interfeisu `Callable<T>`.
   Prasāmie aprēķini notiek `T call()` metodē.
   Metode atgriež `T` informāciju, lietojot **`return`**.

2. Izveidot *N* `Future<T>` vērtības, lai perspektīvā saņemtu informāciju no objektiem ar realizēto `Callable<T>`.

3. Izveidot pavedienu pūlu no *N* pavedieniem ar `ExecutorService` un `Executors` palīdzību.

4. Palaist katru no *N* pavedieniem ar `submit()` palīdzību.
   Rezultātā iegūt `Future<T>` vērtību.

5. Iegūt *N* rezultātus no `Future<T>` vērtībām lietojot `get()`.

6. Apstādināt vadāmos pavedienus ar `shutdown()`.

Uzdevums: atrast *elementu summu* viendimensiju masīvā.

Pirmo un otro masīva puses apstrādā *neatkarīgie* pavedieni.

Katrs pavediens atgriež elementu summu *attiecīgajā pusē*.

```
import java.util.concurrent.*;
class FirstHalf implements Callable<Integer> {
    int [] V;
    public FirstHalf(int [] V) {
        this.V = V;
    }
    public Integer call() {
        int Sum=0;
        for(int i=0; i<V.length/2; i++) {
            Sum += V[i];
        }
        return Sum;
    }
}
```

```
class SecondHalf implements Callable<Integer> {
    int [] V;
    public SecondHalf(int [] V) {
        this.V = V;
    }
    public Integer call() {
        int Sum=0;
        for(int i=V.length/2; i<V.length; i++) {
            Sum += V[i];
        }
        return Sum;
    }
}
```

Galvenās programmas fragments:

```
int [] V = {1, 2, 3, 4, 5};
int SumFirst=0, SumSecond=0, Sum=0;
```

```
ExecutorService ES =
    Executors.newFixedThreadPool(2);

Future<Integer> First;
Future<Integer> Second;

First = ES.submit(new FirstHalf(V));
Second = ES.submit(new SecondHalf(V));

try {
    SumFirst = First.get();
    SumSecond = Second.get();
}
    catch (InterruptedException e) {}
    catch (ExecutionException e) {}

ES.shutdown();

Sum = SumFirst + SumSecond;
```

Piezīmes:

1. Visām summām *obligāti* jāpiešķir sākotnējas vērtības. Pavediena izpildes laikā var notikt izņēmums un rezultāts netiks aprēķināts.

2. Var norādīt *maksimāli atļauto* aprēķinu laiku.

```
try {
    SumFirst = First.get(1, TimeUnit.SECONDS);
}
    catch (InterruptedException e) {}
    catch (ExecutionException e) {}
    catch (TimeoutException e) {}
```

Ja šis laiks tiks pārsniegts, tiks ierosināts `TimeoutException`.

Rezultātā *SumFirst* paliks vienāda ar *0*.

Paralēlas kolekcijas: piemērs

Lai ir rinda ar maksimālo garumu 2

```
ArrayBlockingQueue<Integer> ABQ =
    new ArrayBlockingQueue<Integer>(2);

ABQ.add(1);
ABQ.add(2);

System.out.print(ABQ); //[1, 2]
```

```
ABQ.add(3);

Exception in thread "main"
java.lang.IllegalStateException: Queue full
```

```
System.out.print(ABQ.contains(2)); //true
```

Dēmoni

Dēmons ir jebkurš apkalpojošais pavediens.

Piemērs: *taimera* pavediens, kurš sūta signālus citiem pavedieniem.

Ja programmā palikuši tikai dēmoni, programma *beidz savu darbu*.

Lai pavediens-dēmons izvada ekrānā *veselus skaitļus*.

Galvenajā programmā ir *tikai aizture*.

Pēc galvenās programmas pabeigšanas tiks pabeigts arī dēmons.

```
class ThreadDaemon implements Runnable {
    Thread T;
    public ThreadDaemon() {
        T = new Thread(this);
        T.setDaemon(true);
        T.start();
    }
    public void run() {
        for(int i=1; ; i++) { // bezgalīgi
            System.out.print(i + " ");
            //aizture
        }
    }
}
```

Iespējamais rezultāts:

```
1 2 3 4 5 6
```