Objektorientētā programmēšana

3. laboratorijas darbs

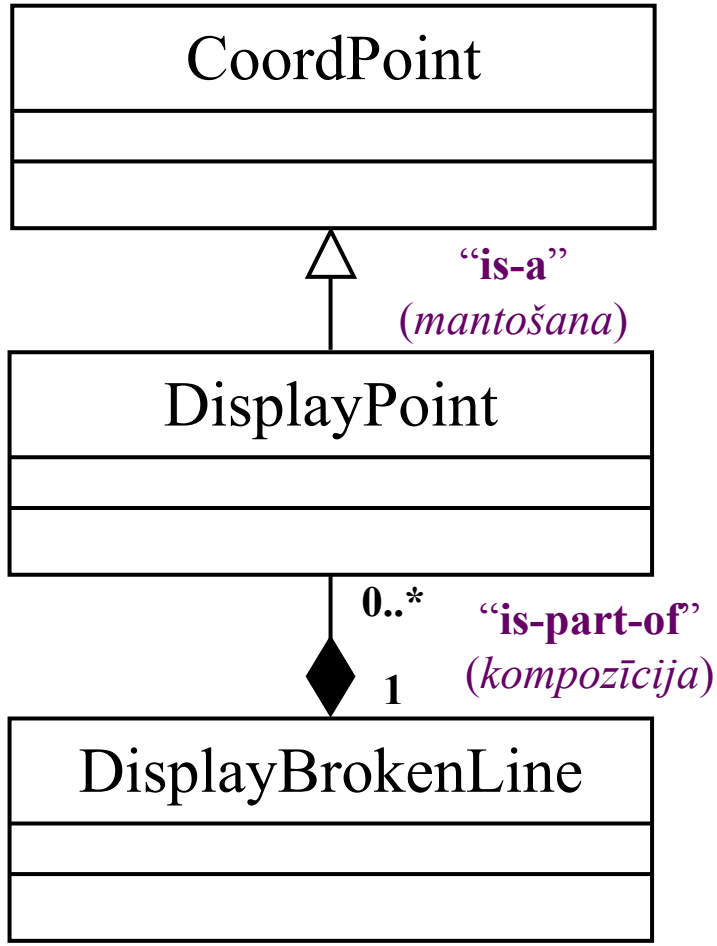
Kompozīcija. Statiskie locekļi. Izņēmumu apstrāde.

Dr. sc. ing. Pāvels Rusakovs
Mg. sc. ing. Vitālijs Zabiņako
Mg. sc. ing. Andrejs Jeršovs
Mg. sc. ing. Pāvels Semenčuks
Mg. sc. ing. Vladislavs Nazaruks

RTU 2012

Kompozīcija. Statiskie locekļi. Izņēmumu apstrāde

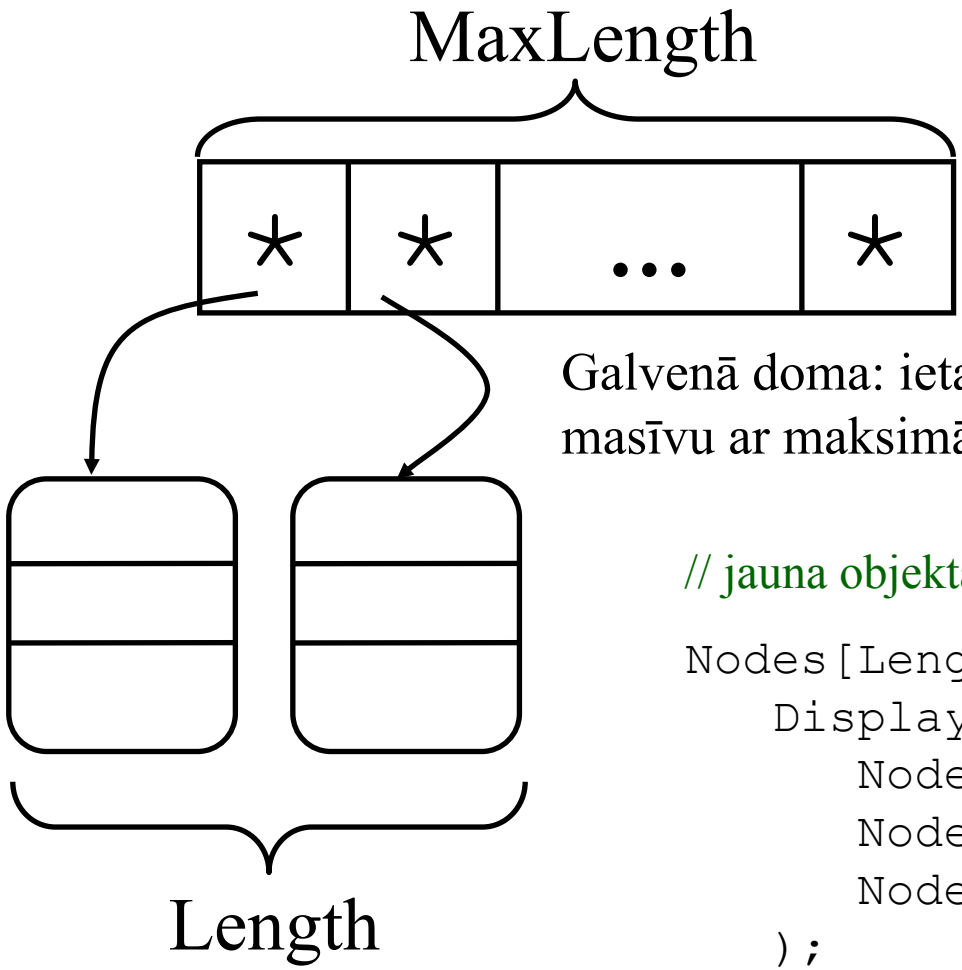
Lauzta līnija monitora ekrānā



Atribūtu sekcija klasē `DisplayBrokenLine`

```
class DisplayBrokenLine {
   private:
      // tipa "rādītājs" definēšana
      typedef DisplayPoint *DPPointer;

      // rādītājs uz rādītājiem
      DPPointer *Nodes;

      // maksimālais mezglu daudzums "pēc noklusēšanas" visās lauztās līnijās
      static const unsigned int DEF_MAX_LENGTH;

      // maksimālais mezglu daudzums konkrētā lauztā līnijā
      unsigned int MaxLength;

      // pašreizējais mezglu daudzums konkrētā lauztā līnijā
      unsigned int Length;

      // līnijas nogriežņu krāsa
      unsigned int LineColor;
  public:
      ...
```

Atmiņas izdalīšana displeja punktiem




```
// rādītāju masīva veidošana
        Nodes =
new DPPointer[MaxLength];
```

Galvenā doma: ietaupīt atmiņu un *sākumā* neveidot masīvu ar maksimālo *objektu* daudzumu.

```
// jauna objekta radīšana
Nodes[Length++] = new
    DisplayPoint(
        Node.GetX(),
        Node.GetY(),
        Node.GetColor()
    );
```

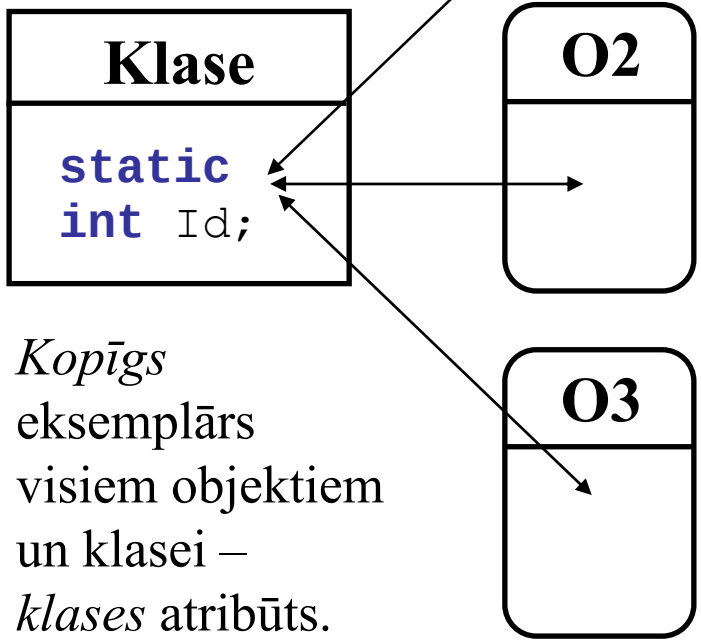
Statiskie klases locekļi: *atribūts* un *metode*

```
class DisplayBrokenLine {
   private:
      ...
      static const unsigned int DEF_MAX_LENGTH;
   public:
      static unsigned int GetDefaultMaxLength () {
         return DEF_MAX_LENGTH;
      }
      ...
};
// C++ valodā statiskā atribūta vērtību norāda aiz klases robežām
const unsigned int DisplayBrokenLine::DEF_MAX_LENGTH = 5;
```

```
// Līnijas radīšana ar parametriem: maksimālais mezglu daudzums un nogriežņu krāsa.
DisplayBrokenLine *Line = new DisplayBrokenLine(3, 1);
...
// statiskās metodes izsaukums no objekta
cout << Line->GetDefaultMaxLength() << ".";
// statiskās metodes izsaukums no klases
cout << DisplayBrokenLine::GetDefaultMaxLength () << ".";
```

Statiskais un *nestatiskais* atribūts `Id`

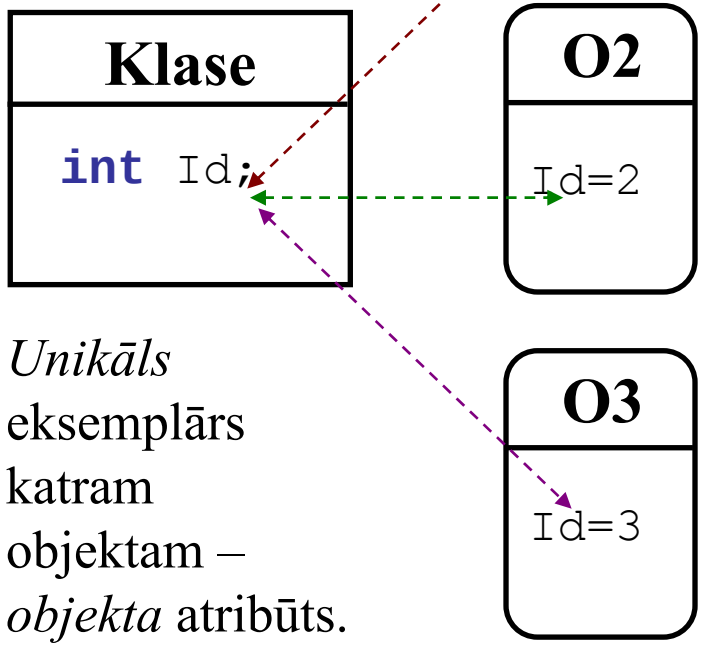
```
Klase::Id
   =
  O1.Id
   =
  O2.Id
   =
  O3.Id
```



Kopīgs eksemplārs visiem objektiem un klasei – *klases* atribūts.

~~`Klase::Id`~~

```
O1.Id
  !=
O2.Id
  !=
O3.Id
```



Unikāls eksemplārs katram objektam – *objekta* atribūts.

Klases `DisplayBrokenLine` konstruktors "pēc noklusēšanas"

```
DisplayBrokenLine() : MaxLength(DEF_MAX_LENGTH),
   Length(0), LineColor(0) {
      Nodes = new DPPointer[MaxLength];
}
```

Jauna mezgla pievienošana

```
void DisplayBrokenLine::AddNode(const DisplayPoint& Node) {
   if (Length == MaxLength)
      throw OverflowException();
   else
      Nodes[Length++] = new DisplayPoint(
         Node.GetX(), Node.GetY(), Node.GetColor()
      );
}
```

`DisplayPoint&` – *norāde* uz objektu (netiks izveidota *parametra kopija*)

`throw OverflowException()` – *izņēmuma* ierosināšana

Izņēmumu klase `OverflowException()`

```
class OverflowException {
   public:
      OverflowException() {
         cout << endl << "Exception created!" << endl;
      }
      OverflowException(OverflowException&) {
         cout << "Exception copied!" << endl;
      }
      ~OverflowException() {
         cout << "Exception finished!" << endl;
      }
};
```

C++ *rezervētie vārdi* izņēmumu apstrādei

`try` – *kontrolējamais* bloks

`catch` – izņēmuma *apstrādātājs*

`throw` – izņēmuma *ierosināšana*

Izņēmumu apstrāde programmā

```
try {
    Line->AddNode(D2);
    cout << "\nNew Node added successfully!" << endl;
}
    catch (OverflowException&) {  // masīva pārpilde
        cout << "Error: maximal length exceeded !" << endl;
    }
    catch (...) {    // visi pārējie izņēmumi
        cout << "Unknown Error !" << endl;
    }
```

Mezglu iznīcināšana destruktorā

```
DisplayBrokenLine::~DisplayBrokenLine() {
    for(unsigned int i=0; i<Length; i++)
        delete Nodes[i];
    delete [] Nodes;
}
```

Šoreiz destruktors ir *obligāts*.

Iegultās funkcijas ar *parametriem-objektiem*

```
class Info {
   private:
      string Key;
      …
   public:
      …
      void SetKey(string Key) { // netiks iegulta
         this->Key = Key;
      }
};
```

Rezultāts: brīdinājums `Functions taking class-by-value argument(s) are not expanded inline in function Info::SetKey(string)`

Iegultā funkcija: parametru nodošana *pēc norādes*

```
void SetKey(const string& Key) {
   this->Key = Key;
}
```

Dažas atšķirības starp *rādītājiem* un *norādēm*

1. Nevar deklarēt *tukšo* norādi.

```
int *Pointer;   // "tukšais" rādītājs
int &Reference; // kompilācijas kļūda
```

Pareiza norādes deklarēšana

```
int k=3;
int &Reference=k;
```

2. Parametru vērtību izmaiņa: *rādītāji* un *norādes*.

```
void ChangeSpeed(int *Speed){
   *Speed = ...;
}

int CurrSpeed = 10;
...
ChangeSpeed(&CurrSpeed);
```

```
void ChangeSpeed(int &Speed){
   Speed = ...;
}

int CurrSpeed = 10;
...
ChangeSpeed(CurrSpeed);
```