


1918
TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

# IDK0051 Objektorienteeritud programmeerimine Javas

Martin Rebane (martin.rebane@ttu.ee)

# Mis võiks une pealt selge olla

- Programmikoodis (nt klassi või liidesega) defineerime **tüübi**
- Konkreetne tüüp „ärkab ellu”, kui loome objekti
- Üks objekt võib esindada mitut tüüpi (nt samaaegselt sõiduk, auto, sõiduauto)

# *Static factory method**

- *Static factory method* on alternatiiv konstruktorile – loote objekti staatilise meetodi sees ja tagastate selle

```
public class MyFactory {
      public static MyFactory getInstance() {
            return new MyFactory();
      }
}
```

Tagastatav tüüp

NB! Ei tagasta mitte klassi, vaid objekti!

# Static factory method – kasutuskohad

- Kontrollite väljastatud objektide arvu:
  - Võimaldab „kallite” objektide taaskasutust, nt andmebaasiühendus
- Kontrollite väljastatud objekti tüüpi:
  - Väljastate lubatud tüübi asemel mõne optimiseeritud alamtüübi

# Kokkulepped nimetuste osas

- `getInstanceBySomething()`
- `valueOf(args)`
- `of(args)`
- `getType(args)`
- `getInstance()`
- `newInstance()`
- `newType()`

Tundke ära, kui kohtate!

# Null ehk tühi viide

- *null* – praktilise käsitluse jaoks lihtsalt puuduv väärtus

```
Student s = null;
```

„Student-tüüpi objekti ei ole”

- Iga objektitüübi vaikeväärtus:
  - `Student s;`

Kuniks viide Student-tüüpi objektile puudub, asendab seda null-viide (tühi viide)

- Kuidas lahendame olukorra, kus meetod saaks äriloogika järgi mõnikord tagastada *null*'i?

# Optional – ei null'i tagastamisele!

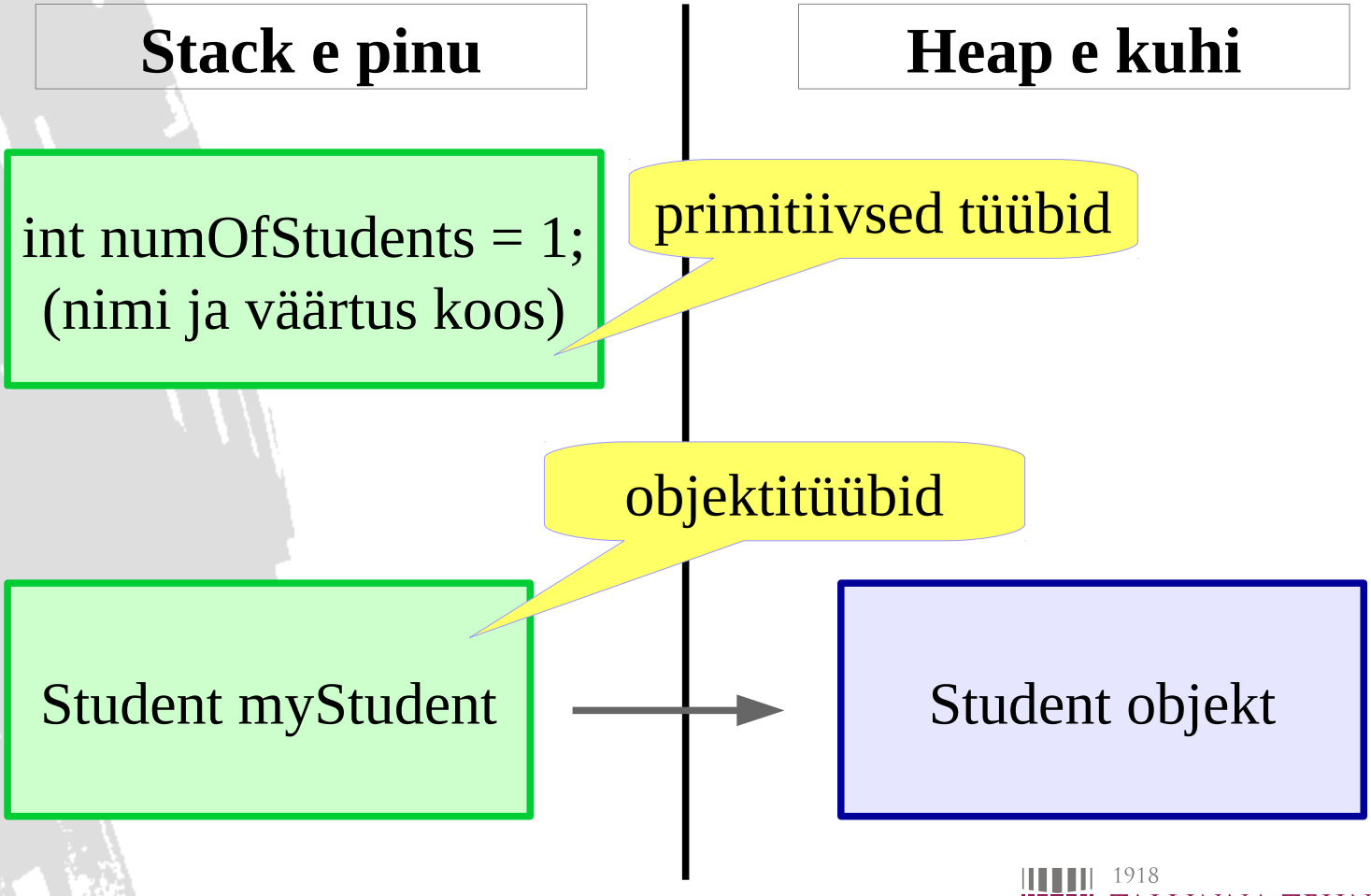
- Võimaldab selgelt väljendada, kui väärtuse puudumine on planeeritud stsenaarium

```
Optional<Product> product;
```

# Muutujate tüübid Javas

- Primitiivsed tüübid (*primitive types*): int, long, short, byte, float, double, boolean
  - Konkreeted väärtused – väärtus on seotud muutujaga
- Objektitüübid (*reference types*): kõik ülejäänud, sh massiivid (*arrays*)
  - Väärtus on muutujast lahutatav

# Muutujatüübid mälus



# Täna

- Varjamine ja kapseldamine
- Erindid ehk *exceptions*
- equals() ja hashCode()
- TDD
- Sõltuvuse sisestamine

# Varjamine

- Alamklass kirjutab üle ülemtüübi meetodi
- Alamklass ei kirjuta üle, vaid varjab ülemtüübi välja

# Varjamine

- Väljad
- Staatilised väljad ja meetodid
- Praktiline tulem: ülemklassist varjatud välja kutsudes kasutatakse ülemklassi väärtust, alamklassist alamklassi väärtust (erinevalt polümorfsetest meetoditest, kus kasutatakse alamklassi meetodit)

# Kapseldamine e encapsulation

- Iga moodul peab oma sisemist toimimist teiste eest peitma
- „Avalikkusele” tuleb näidata miinimumi
  - kõik väljad esmalt *private*, mida pole vaja avalikuks teha

# Kapseldamine

- Väljade kasutamine läbi getterite ja setterite

# Klassi kapseldamine

- Klassi nähtavus
  - Millised olid 2 varianti tavalisel klassil?
- public ja package-private (võtmesõnata)
  - package-private klass on realisatsiooni osa

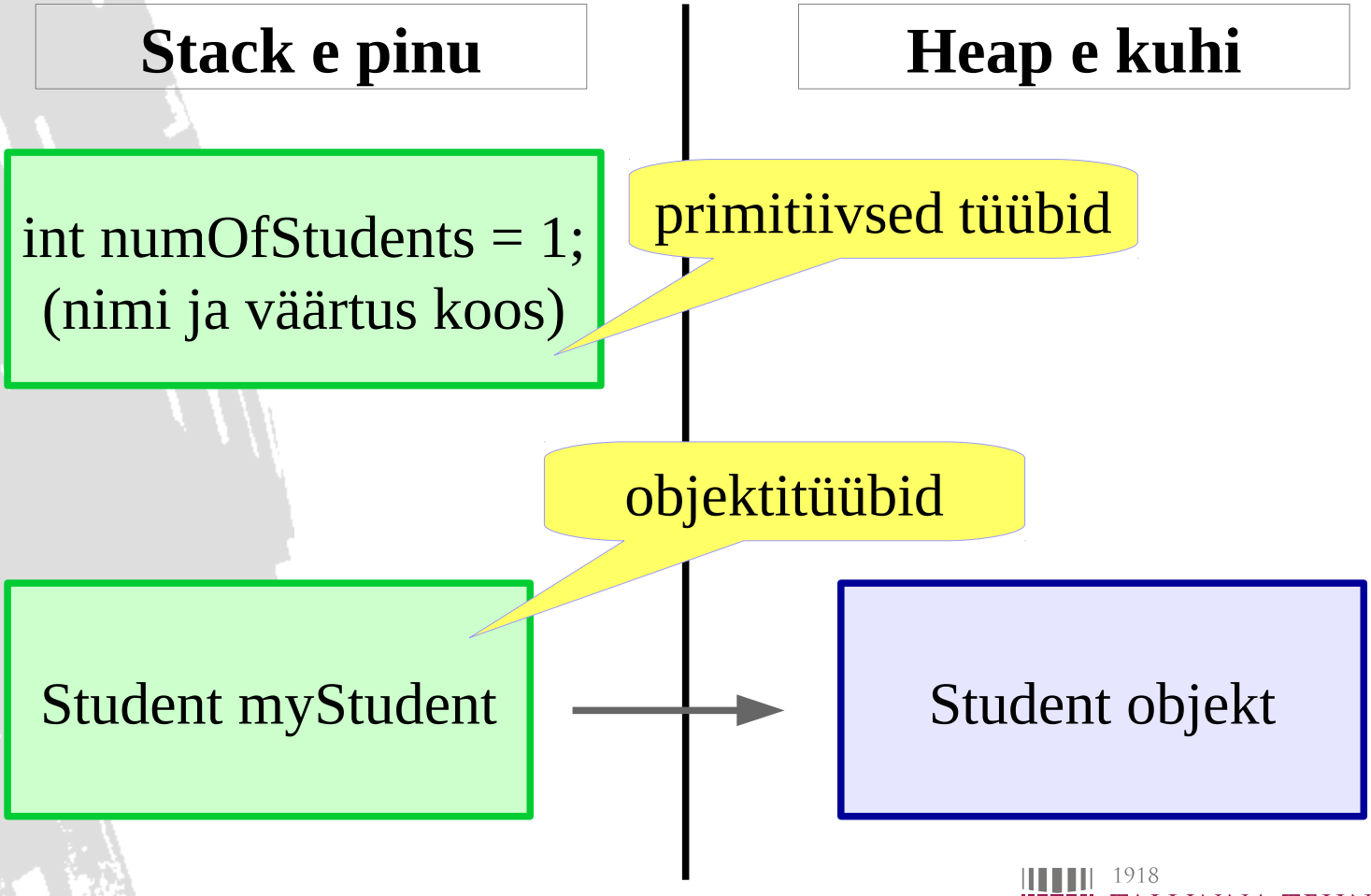
# Võrdlemine

# Mida saab kontrollida „==” abil ?

- Primitiivsete tüüpide korral sisulist võrdsust:

```
int a = 3;
int b = 3;
if (a == b) {
 // on võrdsed
}
```

# Muutujatüübid mälus



# Mida saab kontrollida „==” abil ?

- Objektitüüpide korral samasust:

```
Student a = new Student(„Mary”);
Student b = new Student(„Mary”);
if (a == b) {
 // ei ole samad
}
```

# equals() *vs* ==

- equals() on meetod, mille eesmärk on kontrollida objektide sisulist võrdsust
- Objektide korral kontrollib equals vaikimisi sama, mida ”==”: kas kaks viita viitavad samale objektile
- Objektide korral kasutage equals(), aga:

# equals()

- Objektitüübi jaoks tuleb equals() üle kirjutada, vaikeimplementatsioon Object klassis kontrollib, kas tegu on sama objektiga

- Stringi (jt Java tüüpide) jaoks on Java arendajad selle töö teinud. Enda tüüpide jaoks peate ise equals()-i üle kirjutama

# equals() omadused

- **Sümmeetria:** a.equals(b) saab olla tõene ainult siis kui ka b.equals(a)
- **Refleksiivsus:** a.equals(a)
- **Transitiivsus:** kui a.equals(b) ja b.equals(c), siis a.equals(c)

# hashCode()

- Meetod, mis tagastab objekti väljadele vastava räsiväärtuse (*hash*i), mis ei tohi muutuda kui objekti olek ei muutu

- Räsiväärtused jaotuvad ühtlaselt üle võimalike väärtuste hulga. Väga suure tõenäosusega on kahe erineva olekuga objekti räsid erinevad, kuid võivad siiski kattuda.

# hashCode

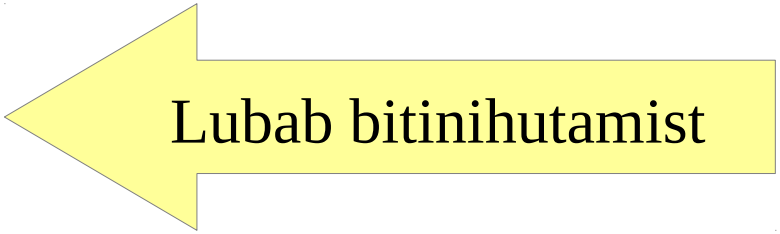
- equals() ja hashCode(): kui kirjutate üle equals(), tuleb kirjutada ka hashCode(). Miks?

- Vihje: HashMap
  - Lisame hashCode() järgi
  - Otsime hashCode() ja equals() järgi

# hashCode() loomine

- Initsialiseerige üheks: int hashCode = 1;
- Primitiivse numbrilise tüübi korral korrutage:

  31 * hashCode + value




- Objektitüüpi välja korral kasutage väljal oleva objekti hashCode()

# equals() ja hashCode ühilduvus

- Kui:
  a.equals(b)
- ..siis:
  a.hashCode() tagastatav räsi on identne b.hashCode() tagastavaga

- Vastupidi ei pea olema, st sama hashcode ei tähenda võrdsust (nt juhuslik kattuvus)

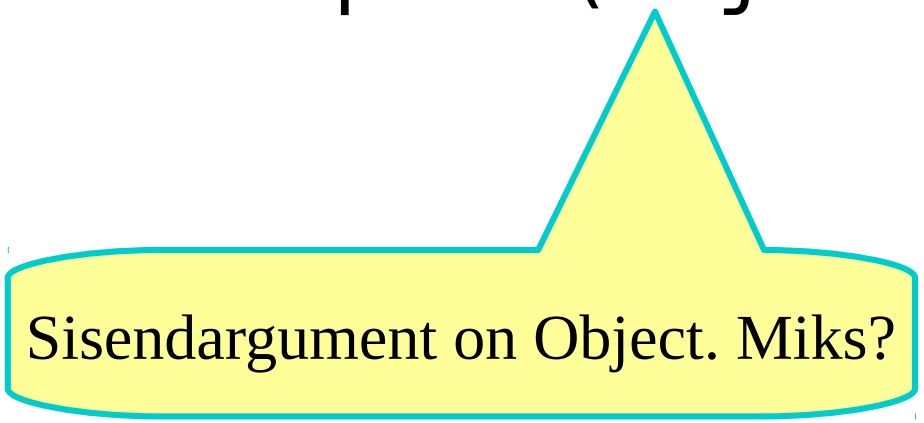
# equals() ja hashCode()

- equals() - objektide võrdlemiseks
- hashCode() - objektide paigutamiseks kiiresti otsitavatesse mappidesse

# equals()

- NB! equals() meetodi signatuur on:

```
public boolean equals(Object o)
```



# Objektide võrdlemine

- == kontrollib primitiivsete tüüpide korral sisulist võrdsust
- Objektitüüpide korral kontrollib, kas tegu on sama objektiga kuhjas (*heap*)
- Objektide sisuliseks võrdlemiseks tuleb kasutada equals() meetodit

# equals()

- Objektitüübi jaoks tuleb equals() üle kirjutada, vaikeimplementatsioon Object klassis kontrollib, kas tegu on sama objektiga

- Stringi (jt Java tüüpide) jaoks on Java arendajad selle töö teinud. Enda tüüpide jaoks peate ise equals()-i üle kirjutama

# hashCode()

- Meetod, mis tagastab objekti väljadele vastava räsiväärtuse (*hash*i), mis ei tohi muutuda kui objekti olek ei muutu

- Räsiväärtused jaotuvad ühtlaselt üle võimalike väärtuste hulga. Väga suure tõenäosusega on kahe erineva olekuga objekti räsid erinevad, kuid võivad siiski kattuda.

# Equals(), sümmeetria nõue ja alamklassi lõks

- Sümmeetria: IF a.equals(b) THEN b.equals(a)

- Kui alamklass defineerib lisavälju ja eraldi equals meetodi, siis võite rikkuda sümmeetria nõuet:

```
Student s = new Student();
Student s2 = new SomeStudent();

s.equals(s2); // true
s2.equals(s); // false
```

- Palun kirjutage equals() meetod alati üle nii, et sümmeetria nõue jääks kehtima :)

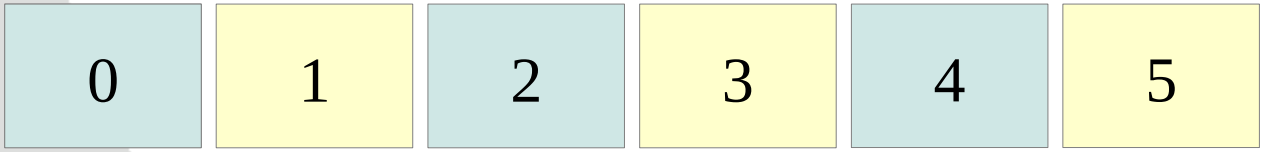
# HashMap<keyType, valueType>

- *Võti-väärtus (key – value)* paari hoidmine
- Kiire andmetagastus *võtme* järgi, Ω(1)
- *võti* on unikaalne – kahte sama võtmega kirjet hoida ei saa
- ei taga lisamise järjekorra säilimist

# HashMap

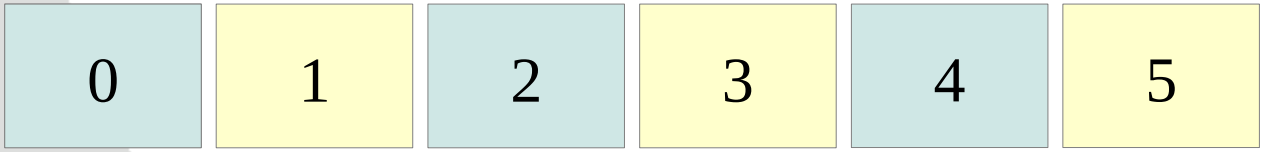
- Fail-fast – struktuuri ei tohi itereerimisel muuta
- ConcurrentModificationException
  - v.a remove() meetod
- Katsuge *key*'ks olevat objekti mitte muuta (hashCode muutub)!

# HashMap tööpõhimõte




- Lihtsustatult list, kus elementideks on linked listid
- Igat elementi kutsutakse lahtriks (bucket)
- Igas lahtris hoitakse samaväärse hashCodega objekte

# HashMap tööpõhimõte




- Objektipaari lisamisel otsustatakse hashCode alusel, kuhu lahtrisse (bucketisse) objektipaar lisada
- Tagastamisel otsitakse hashCode alusel õige lahter välja ja valitakse sealt seest õige paar equals() abil

# hashCode

- equals() ja hashCode(): kui kirjutate üle ühe, tuleb kirjutada ka teine. Miks?
- Jõudlus
  - Ebaefektiivne hashCode() lahendus vähendab räsidel põhinevate andmestruktuuride (nt HashMap) jõudlust
- Töökindlus
  - Ebastabiilne hashCode() arvutus teeb räsidel põhinevad andmestruktuurid katki

# hashCode() ja jõudlus

- Kui hashCode() on erinevatel objektidel tihti sama või samaväärne (*hash collision*), lisatakse nad HashMapis samasse lahtrisse (bucket).

- HashMapi kiirus põhineb aga sellel, et erinevad objektid paigutatakse hashCode alusel erinevatesse lahtritesse

# hashCode ja töökindlus

- Kui hashCode() on koos equals() meetodiga üle kirjutamata või töötab vigaselt, siis õnnestub objekti lisamine HashMapi, kuid get() operatsioon ei pruugi enam õnnestuda
- Põhjus: objekti otsitakse valest lahtrist (bucketist) kui hashCode lisamisel ja pärimisel on erinevad

# Võtme valik HashMap tarbeks

- Valige võti, mis ei muutuks sel ajal kui objekt HashMapis on – muidu mõjutab see hashCode väärtust ja te ei pruugi objekti enam üles leida

# hashCode() loomine

- Initsialiseerige: int hashCode = 1;
  - Eesmärk: algne väärtus erineks nullist
- int, char, byte, short korral korrutage:

  31 * hashCode + [value]
- Objektitüübi korral kasutage väärtuse asemel objekti hashCode()

# equals() ja hashCode ühilduvus

- Kui:
  a.equals(b)
- ..siis:
  a.hashCode() tagastatav räsi on identne b.hashCode() tagastatavaga

- Vastupidi ei pea olema (nt juhuslik kattuvus)

# HashMap – Java 8

- Täiendused jõudluses – teatud juhtudel on linked listi asemel kasutuses binary tree, mis on kiirem suure *hash collisioni* puhul

- HashMapi algne initsialiseerimine on kiirem (ei looda kohe 16 bucketiga listi nagu Java 7 ja enne)

# Veatöötlus

# Veatöötlus, erindid

- Oskame ette näha olukordi, kus programm ei pruugi käituda soovitult
- Võib-olla suudame programmi töö taastada
- Informeerime kasutajat veast viisakal moel

# try – catch – finally

```
try {
 new Student(89843984398);
} catch (Exception e) {
 System.out.print(e.getMessage());
}
```

- Lisaks võimalik *finally* osa – täidetakse alati kui programm ellu jääb

# Erind on object

- new Exception(„Viga”);
  - tavaline uue objekti loomine
  - konstruktorile anname String-tüüpi argumendi „Viga”
- catch (SomeException e)
  - püütakse kinni SomeException-tüüpi objekt, mida saab kasutaja nime „e” abil

# Erind on objekt

- Student s = new Student(„John”);
- Exception e = new Exception(„Veateade”);
- Ärge unustage, et erind on objekt!

# Erindite tüübid

- Checked exceptions – kontrollitud erindid
  - Enamasti teadlikult loodud erind, aga ka JVM poolt IOException ja FileNotFoundException
  - **Programmeerija peab neid töötlema.**
- Unchecked exceptions – kontrollimata
  - Enamasti JVM-i poolt loodud erind (nt NullPointerException), aga seda võib luua ka programmeerija (nt IllegalArgumentException)
  - **Te ei pea neid töötlema!**
- Errors - vead
  - Programmikoodist sõltumatud vead

# Erindite tüübid

- *checked exceptions* e kontrollitud erindeid ei saa ignoreerida
- *Unchecked exception* viitab mingile (loogika)veale programmis
- *Error* on programmiväline viga – seda me ise kunagi ei loo

# Kontrollimata erind

- Kood kompileerub, aga kuskil on (loogika)viga
- Näiteks meetod ootab objekti, aga saab hoopis *null;* või jagamine nulliga
- Võib jätta töötlemata, eriti kui tulevad kellegi teise loodud API kaudu

# Error ehk viga

- Programmiväline viga, nt faili lugemine katkeb kõvaketta vea tõttu
- Kasutajale kuvatakse töötlemata veateade
- Töötlemine ei ole enamasti mõtestatud

# Kontrollitud erind (checked ex.)

- Kontrollime alati kas erind tekitati
- Kui jah, siis töötleme seda
- Võimalusel taastame programmi normaalse töö ...
- ... ja/või vihjame kasutajale, mida teha

# Erindid on laiendatavad