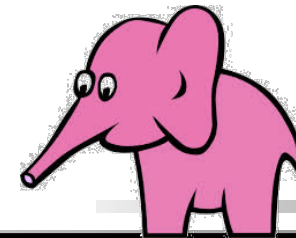


GRUNDLÆGGENDE PROGRAMMERING



Handling Errors (Fejlhåndtering)



Claus Brabrand

`(((brabrand@itu.dk)))`

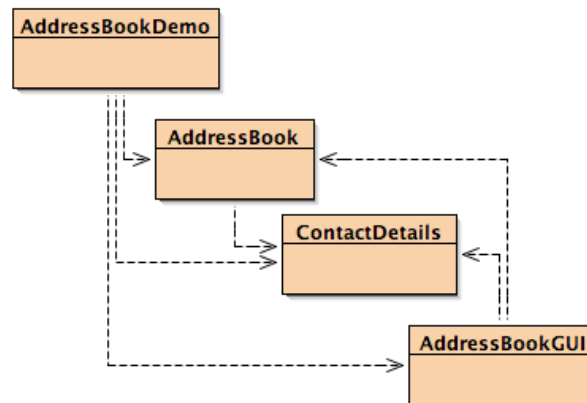
Associate Professor, Ph.D.

`(((Software and Systems)))`

 **IT University of Copenhagen**

A G E N D A

- Client-Server Architecture
- Exceptions (Undtagelser)
- Assertions
- I/O-Example (kort)
- Example: "Adressebog"



Client-Server Architecture

- **Client-Server Architecture:**

- En eller flere klienter anvender en server:

- **Server: Passiv (responsiv)**

- Vedligeholder en "intern tilstand"

- **Clients: Aktiv (proaktiv)**

- Beder server om at "gøre ting"
 - Ofte kan de bede server om at "ændre sin tilstand"
 - De kan bede server om data "baseret på dens tilstand"



Client-Server Architecture

■ Eksempler:

- Web-browser og Web-server
- Java-program og database
- Java-program og fil-system
- Menneske og kontantautomat
- ...



■ NB: Klient kan være:

- a) Homo Sapiens
- b) Maskine; *eller*
- c) Maskine-der-interagerer-med-en-Homo-Sapiens

Hvem er Server? Hvem er Klient?

- Det kan afhænge af synspunktet:



- Fx. kan en **web-server** *samtidigt* være **server** (for en **browser**) og **klient** (for en **database**) !

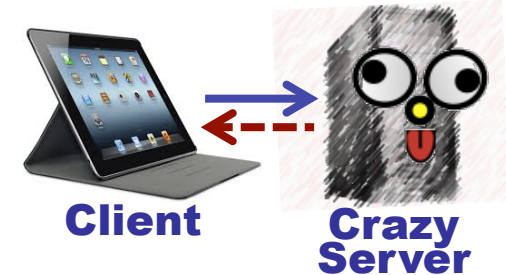
Fejlhåndtering & Forebyggelse

Robustness:

"Programmer der er afhængige af deres omgivelser skal kunne håndtere problemer opstået i deres omgivelser !"

Klient perspektiv:

- Server er ikke under clients kontrol, men utilregnelig
- Client skal være foreberedt på at serverkald kan give fejl
- Hvis server ikke kan gennemføre forespørgsel vil vi vide det



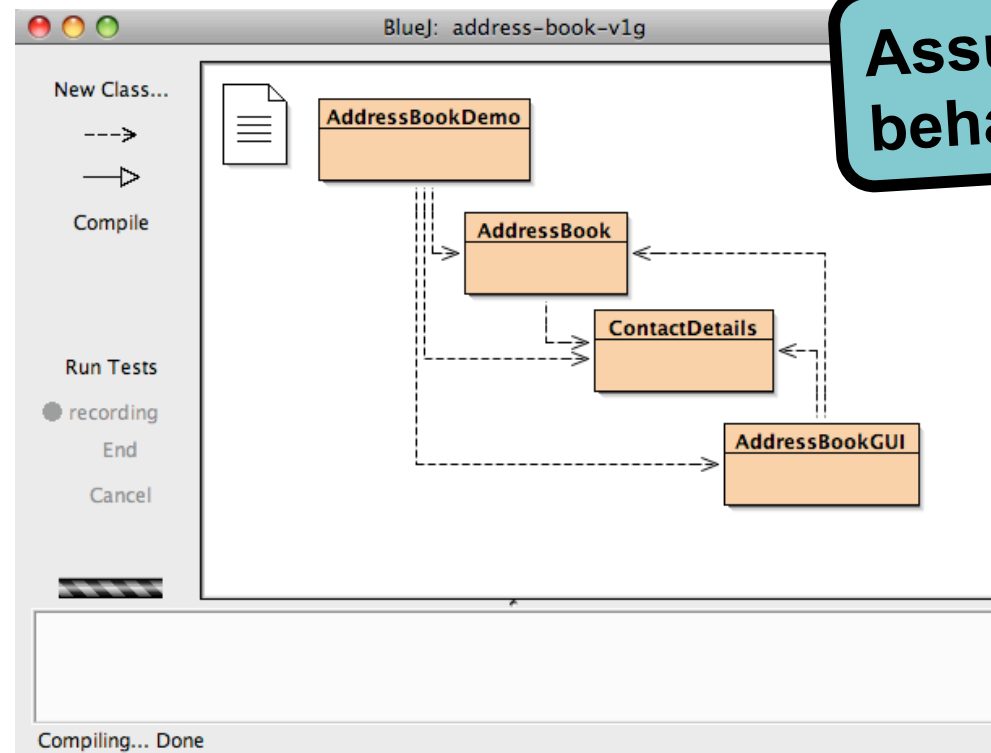
Server perspektiv:

- Client kan komme med meningsløse forespørgsler
- Server skal sikre sig mod at client kan bringe server i meningsløs tilstand [husk: SQL injection attacks (sidste uge)]
- Client skal kunne tvinges til at reagere på fejl



Example:

■ Address Book (V1G):



Assumes well-behaved clients

- The projects represent an application that ***stores personal-contact details*** (name, address, and phone number) for an arbitrary number of people.

Felter

```
public class AddressBook {  
    // Storage for an arbitrary number of details.  
    private TreeMap<String, ContactDetails> book;  
    private int numberOfEntries;  
    ... }
```


- Feltet **book** bruges til opslag i telefonbogen
- Hvert opslag gemmes både under navn og telefonnummer
- Samtidigt holder **numberOfEntries** styr på hvor mange opslag der er i adressebogen
- **Q:** Hvilke **implicitte antagelser** gør vi om en adressebogs tilstand?

Design af Server



- En adressebog er et eksempel på en **server**
- Ved design findes **to ekstreme synspunkter**:
 - **a) Clients** beder kun om **rimelige ting**
(og det er clientens ansvar hvis ting går galt)
 - **b) Serveren** skal operere i et **fjendtligt miljø**
(dvs håndtere urimelige clients)
- Nuværende implementation tager synspunkt **a)**
 - **Q**: Hvad sker der hvis man forsøger at **slette** et opslag som **ikke findes**?

Input Validering

```
public void removeDetails(String key) {
    ContactDetails details = book.get(key);
    book.remove(details.getName());  // CRASH: details is 'null'!
    book.remove(details.getPhone());
    numberOfEntries--;
}
```

```
java.lang.NullPointerException
    at AddressBook.removeDetails(AddressBook.java:121)
```

■ Defensive programming (with input validation):

```
public void removeDetails(String key) {
    if ( keyInUse(key) ) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

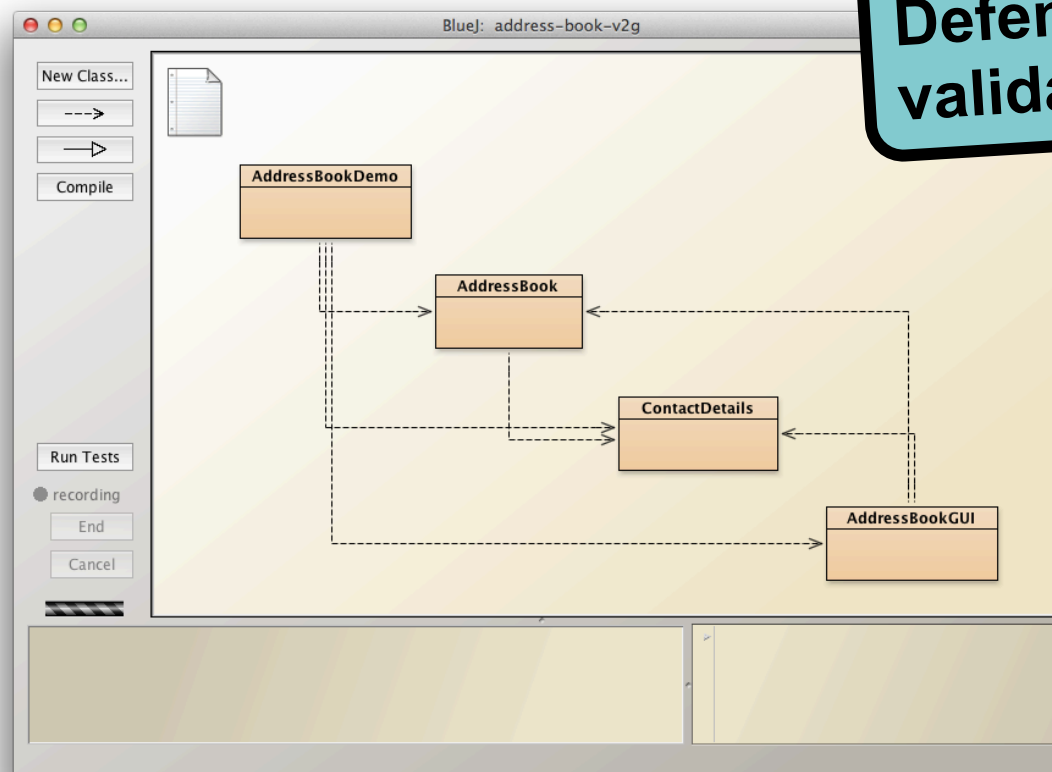
Input Validering (cont'd)

- Ofte er det nødvendigt at kommunikere til klienten *at dens forespørgsel fejlede*
- Dette kan gøres f.eks. ved at introducere returværdier:

```
public boolean removeDetails(String key) {  
    if (keyInUse(key)) {  
        ContactDetails details = book.get(key);  
        book.remove(details.getName());  
        book.remove(details.getPhone());  
        numberOfEntries--;  
        return true;  
    } else {  
        return false;  
    }  
}
```

Example

■ Address Book (V2G):



**Defensive input
validation checks**

Problemer med returværdier!

```
public boolean removeDetails(String key) {  
    if (keyInUse(key)) {  
        ContactDetails details = book.get(key);  
        book.remove(details.getName());  
        book.remove(details.getPhone());  
        numberOfEntries--;  
        return true;  
    } else {  
        return false;  
    }  
}
```

- Hvad hvis metoden allerede returnerer en værdi?
- Klienten kan "glemme" at inspicere returværdien...:

```
removeDetails(some_element); // i.e., ignoring return value !
```

- (og begynde at opføre sig forkert som følge deraf)
- + "real code" drowns in test-and-error-handling-code!

Eksempel fra sidste uge

■ Test (om går godt) + fejlhåndter

```
/* execute SQL query! */
public static void main(String[] args) {
    boolean ok = DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    if (!ok) {
        <<< handle register driver error >>>
    } else { // driver registered...:
        Connection connection = DriverManager.getConnection(DB_URL, USER, PASS);
        if (connection == null) {
            <<< handle connection error >>>
        } else { // we are connected...:
            Statement statement = connection.createStatement();
            if (statement == null) {
                <<< handle statement error >>>
            } else { // we have a statement...:
                String sql = "SELECT * FROM mailing_list";
                ResultSet rs = statement.executeQuery(sql);
                if (rs == null) {
                    <<< handle query error >>>
                } else { // we have a query response...:
                    <<< FINALLY PROCESS QUERY >>>
                }
            }
        }
    }
}
```

SQL Client

"Real code" drowns in:

- lots of tests; *and*
- error handling!

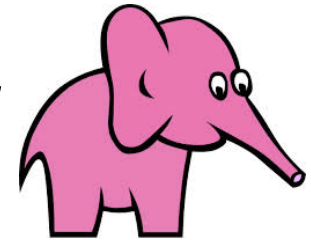
Exceptions (Undtagelser)



try / throw / catch / finally

Exceptions (Undtagelser)


- I stedet for at returnerer en værdi kan en metode ***"throw an exception"*** (på dansk: ***"rejse en undtagelse"***)



- I kender sandsynligvis allerede:
 - **NullPointerException**
 - **IndexOutOfBoundsException**
 - . . .

Exceptions (eksempel)

■ Før:

```
public void addDetails(ContactDetails details) {  
    book.put(details.getName(), details);  // details is 'null'!  
    book.put(details.getPhone(), details);  
    numberOfEntries++;  
}
```

■ Nu (med exceptions):

```
public void addDetails(ContactDetails details) {  
    if (details == null) {  
        throw new IllegalArgumentException("Null passed to addDetails");  
    }  
    book.put(details.getName(), details);  
    book.put(details.getPhone(), details);  
    numberOfEntries++;  
}
```

■ (Nu: stabil overfor urimelige/malicious clients)

Exceptions (fortsat)

- Først laver man et Exception objekt:

```
Exception e = new IllegalArgumentException("Null passed to ...");
```

(NB: 'IllegalArgumentException' er subklasse af 'Exception')

- Dernæst kastes exception: `throw e;`
- hvilket øjeblikkeligt afbryder method execution
- Der returneres *ikke* en værdi:
Resultatet af metodekaldet er en **exception**
- Ofte gøres disse to ting i én linie:

```
throw new IllegalArgumentException("Null passed to ...");
```

Håndtering vs Propagering (af Undtagelser)

- En client der kalder en metode (som rejser en undtagelse) har **to valgmuligheder**:
 - 1) den kan ***håndtere undtagelsen***; *xor*
 - 2) den kan ***propagere undtagelsen***

1) Håndtering af Undtagelser

- Kode i client:

```
try {  
    doSomething(); // may *throw* exception!  
} catch (IllegalArgumentException e) {  
    // exception handling/recovery code here  
}
```

- Hvis en undtagelse kastes inden for **try**-blokken afbrydes evalueringen øjeblikkeligt
- Evalueringen fortsættes da i **catch**-blokken
- Hvis **try**-blokken evalueres normalt skippes **catch**-blokken

2) Propagering af Undtagelser

- Hvis en metode *ikke* håndterer en undtagelse bliver denne undt. resultatet af metodekaldet:

```
class Waiter {  
    public Dish takeOrder(Order order) {  
        Dish dish = chef.cook(order); // May raise exc'!  
        return dish;  
    }  
}
```

- Dette kaldes *at propagere* en undtagelse
- Undtagelsen kan da håndteres i metoden-der-kalder-takeOrder !

Eksempel

```
class Customer {  
    public Dish placeOrder() {  
        try {  
            Dish dish = waiter.takeOrder(fish); // may throw exc!  
        } catch (RestaurantOnFireException e) {  
            run();  
        }  
        eat(dish);  
    }  
}
```

client
server

```
class Waiter {  
    public Dish takeOrder(MenuItem item) {  
        Dish dish = chef.cook(item); // may throw exception!  
        return dish;  
    }  
}
```

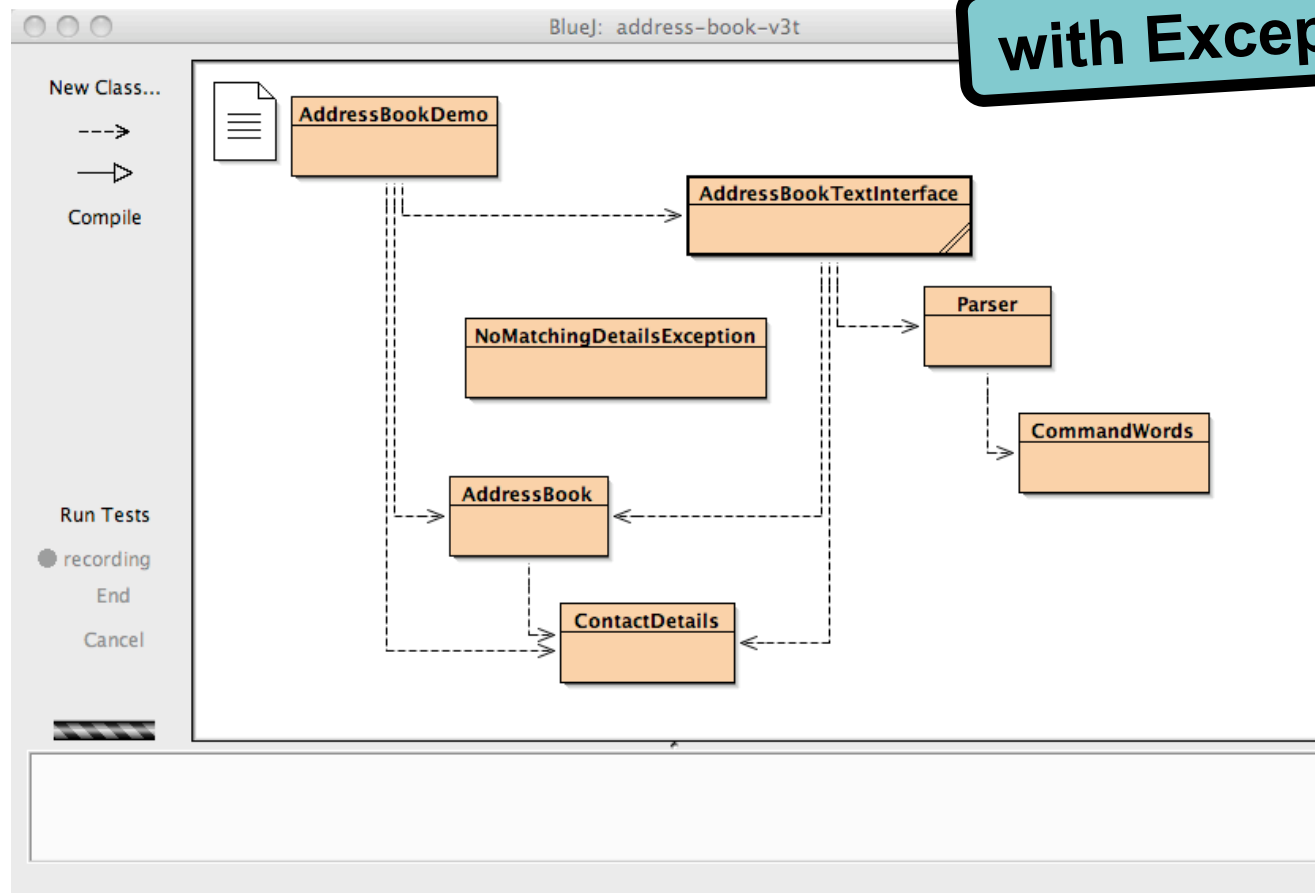
client
server

```
class Chef {  
    public Dish cook(MenuItem item) {  
        throw new RestaurantOnFireException("Fire!");  
    }  
}
```

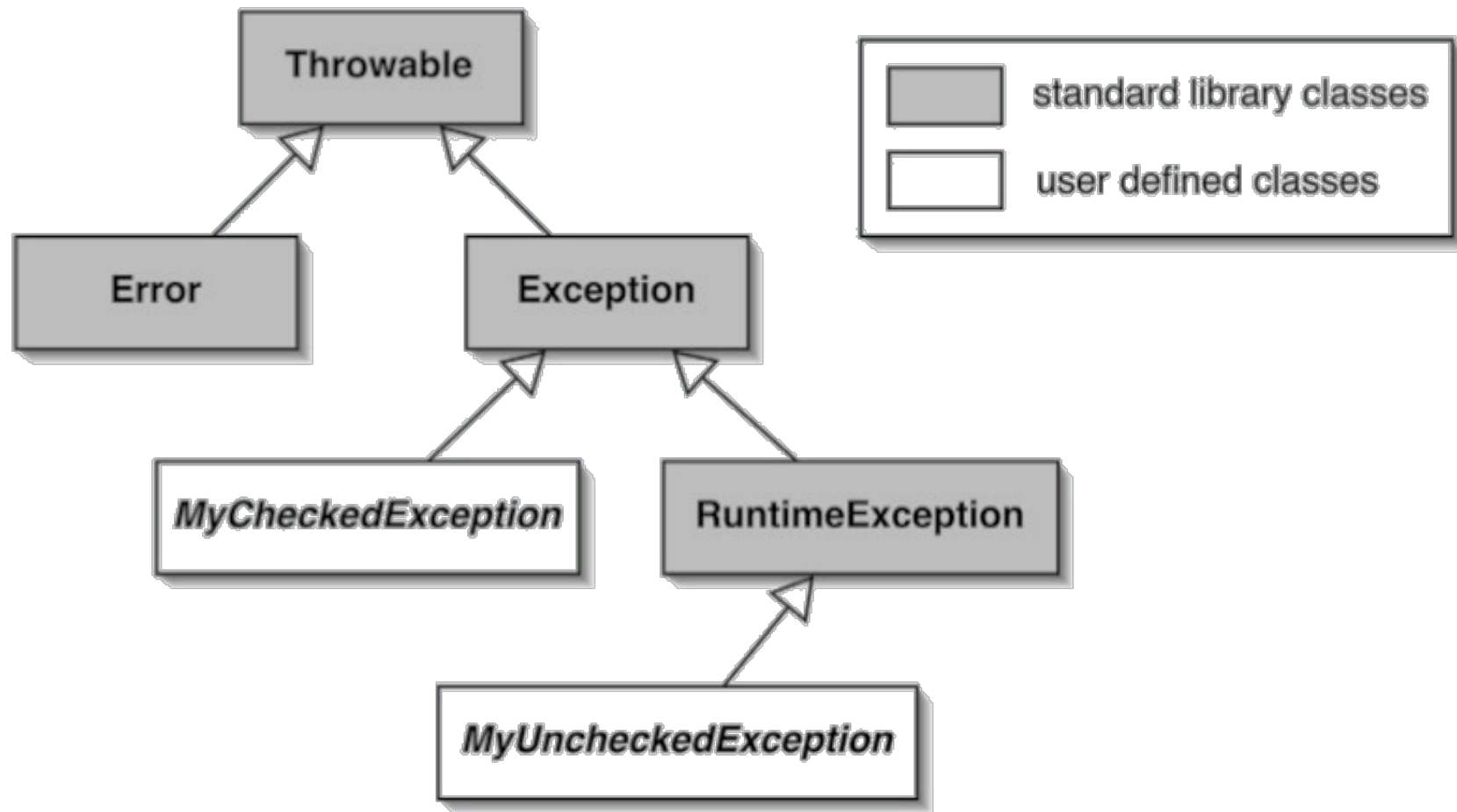
Example

■ Address Book (V3T):

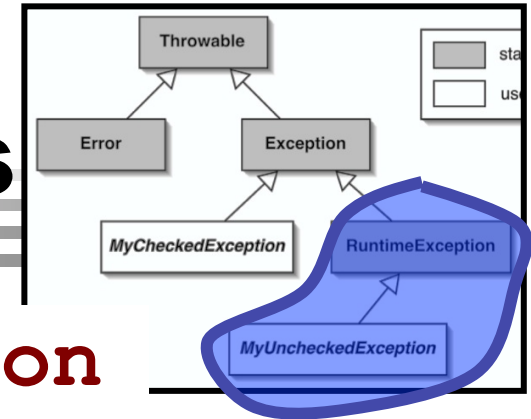
with Exceptions!

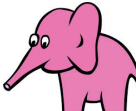


Exceptions (Class Hierarchy)

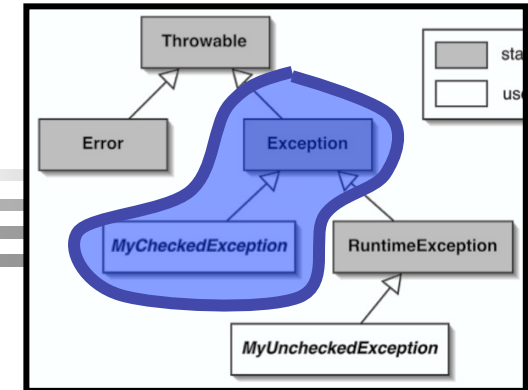


Unchecked Exceptions



- Subklasser af **RuntimeException**
- Intet krav om at *unchecked exceptions* skal håndteres (clients kan propagere dem)
- Eksempler:
 - **IllegalArgumentException**
 - **IndexOutOfBoundsException**
- Bruges især ved exceptions som:
 - **a)** er sjældnere ()
 - **b)** sandsynligvis ikke kan håndteres
 - **c)** når vi regner med at fejl \Rightarrow program termination!

Checked Exceptions



- Subklasser af **Exception** (men ikke subklasse af **RuntimeException**)
- For at kunne rejse en checked exception skal det være erklæret i signaturen til metoden at dette kan ske:

```
public void removeDetails(String key)
    throws NoMatchingDetailsException
```

- **NB:** Via denne erklæring sikrer compileren at en klient ikke kan ignorere en **"(compiler) checked exception"**
- Klienten skal da håndtere denne undtagelse (fx via en **try-catch**-blok)
- Bruges ved hyppige undtagelser, der bør kunne håndteres

Egne Exceptions



- Man kan selv erklære exceptions (klasser) ved at lave subklasser af **Exception** henholdsvis **RuntimeException**
- Hvilken af disse man vælger afgør om klassen bliver checked eller ej:
 - **Exception**: checked!
 - **RuntimeException**: unchecked!
- Nyttigt hvis man ønsker at exceptions skal "bære" program-specifik information med sig (fx til at hjælpe med fejl recovery/håndtering)

Multiple Exceptions

- En metode skal liste alle sine checked exceptions i signaturen:

```
public void removeDetails(String key)  
throws NoMatchingDetailsException, SomeOtherException
```

- **NB**: Resultatet af et kald af en metode er dog naturligvis altid blot én exception

try-catch

```
try {  
    ...  
} catch (SomeException e) {  
    ...  
} catch (AnotherException e) {  
    ...  
} finally {  
    ...  
}
```

- Hvis **try**-blok kaster en exception, så udfør da **første matchende catch**-blok
- Højst en **catch**-blok kan udføres
- **finally**-blok udføres **altid**, til sidst

Exercise

```
String attemptDiv(int a, int b) {
    try {
        divide(a, b); // may throw IllegalArgumentException !
        return "success";
    } catch (Exception e) {
        return "failure";
    } catch (IllegalArgumentException e) {
        return "bad arg";
    }
}

int divide(int x, int y) {
    if (y == 0) {
        throw new IllegalArgumentException("divide by zero");
    }
    return x / y;
}
```

- **Q:** Hvad er mulige værdier fra `attemptDiv`?

Exercises

```
int foo() {
    try {
        throw new Exception("What is the effect of this???");
    } catch(Exception e) {
        return 7;
    } finally {
        System.out.println("finally branch executed!");
    }
}
```

```
int bar() {
    try {
        throw new Exception("What is the result of this???");
    } catch(Exception e) {
        return 87;
    } finally {
        return 42;
    }
}
```

Exercise

```
int do() {  
    try {  
        cook(); // may throw RestaurantOnFireException!  
    } catch (RestaurantOnFireException e) {  
        extinguishFire(); // may throw PanicException!  
    } catch (PanicException e) {  
        System.out.println("run!");  
    }  
}
```

- **Q:** What is the effect of running this program?
(let's imagine that all exceptions are thrown)

Hvorfor *finally*? (motivation)

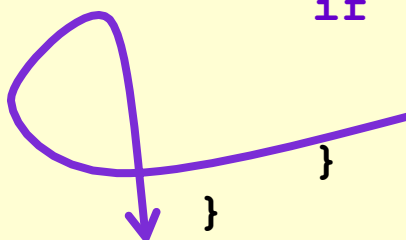
- *finally* kan bruges til at "*rydde op*":
 - close file
 - end connection
 - terminate processes
 - ...
- *finally*-blokken udføres *altid!:*
 - selv hvis der returneres en returværdi fra try blokken
 - selv hvis der rejses en undtagelse der ikke fanges

Abusing Exceptions!

- Man kan naturligvis "udnytte" exceptions; e.g.:

```
class FoundElement extends Exception {
    int index; // will hold index of element found
}

int findElementInList(int n, int[] list) {
    try {
        for (int index=0; index<list.length; index++) {
            if (list[index] == n) {
                // index of element found, so abort for-loop:
                throw new FoundElement(index); // giga hack!!!
            }
        }
    } catch (FoundElement e) { // hack: continue processing:
        int element = e.index;
        // continue normal processing
    }
}
```



Don't!

Possibly excluding "job indispensabilization" (producing deliberately obfuscated illegible code)!

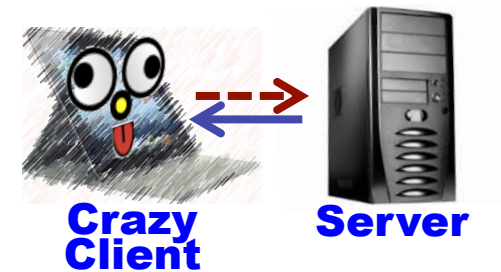
Assertions



assert

Assertions

- I bogens eksempel var der en masse *implicite antagelser* om tilstanden af et **AddressBook** objekt
 - (Alle andre tilstande er meningsløse for os)
- **Robust implementation af AddressBook** (vs crazy client):
 - Kald af metoder kan aldrig bringe et **AddressBook** objekt i en "meningsløs tilstand"



Assertions

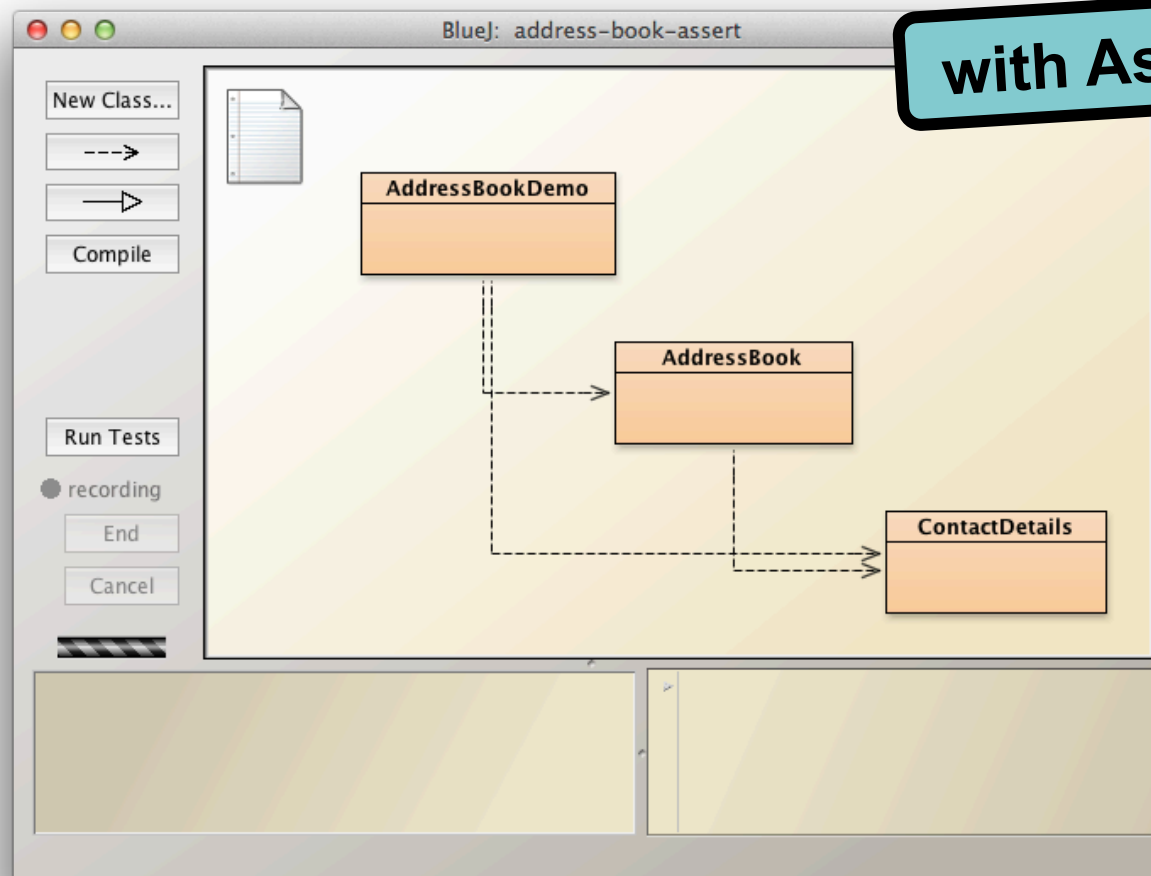
- Assertions er check for *implicitte antagelser* der indsættes *explicit i koden*:

```
assert size == x*y ;
```

- Hvis en *assertion fejler*, så afbrydes programkørslen med en fejlmeddelelse
- Disse checks køres i *udviklingsfasen* (men ikke i den endelige version)
- Hvis en *assertion fejler*, så ved vi at der er fejl i implementationen (samt hvor)

Example

■ Address Book Assert:



with Assertions!

Eksempel

■ Address Book with Assertions:

```
public void removeDetails(String key) {
    if (key == null){
        throw new IllegalArgumentException("RemoveDetails with null");
    }
    if (keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
    assert !keyInUse(key) ;
    assert consistentSize() :
        "Inconsistent book size in removeDetails" ;
}
```

Execution & Assertions

- Assertions checks kun, hvis *fortolkeren* (Java Runtime System) har fået at vide at den skal gøre det (`-ea: enable assertions`):

```
%> java -ea MyClass
```

- Det gør det let at fjerne assertions fra den endelige version:

```
%> java MyClass
```

- Assertion kode bør ikke indeholde egentlig programkode med "side-effekter" som i:

```
assert book.remove(key) != null ;
```


I/O (Input/Output)



I/O (Input/Output)



- I/O eller blot "IO" (input/output) er fx:
 - *åbne* en fil (NB: især dette er "*error-prone*")
 - *læse* fra en fil (eller **input** fra en Homo Sapiens)
 - *skrive* til en fil (eller **output** til en Homo Sapiens)
 - *lukke* en fil
- **NB:** I/O er særligt følsomt over for fejl !
(og man bør tage højde for problemer)
- I/O håndteres i pakken **java.io**:
 - **IOException** (checked) bruges til I/O-fejl

Eksempel

■ Opening files (common code structure):

```
boolean fileOpen = false;
while (!fileOpen) {
    try {
        << open the file >>
        fileOpen = true;
    } catch(IOException e) {
        << ask user for another filename >>
    }
}
// end of 'while' loop: file is now open
try {
    << read and write >>
} catch(IOException e) {
    << report error >>
} finally {
    << close file >>
}
```

Java + SQL ! (from last time)

■ Normal processing vs Exceptional processing:

```
try {
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    connection = DriverManager.getConnection(DB_URL, USER, PASS);
    statement = connection.createStatement();
    ResultSet rs = statement.executeQuery("...");
    while (rs.next()) {
        String name = rs.getString("name");
        String email = rs.getString("email");
        display(name, email);
    }
} catch (ClassNotFoundException e) {
    << Handle driver not found >>
} catch (SQLException e) {
    << Handle SQL error >>
} catch (Exception e) {
    << Handle general error >>
} finally {
    rs.close();
    connection.close();
}
```

normal processing

exceptional processing

always close

Thx!

A decorative graphic consisting of several horizontal lines of varying lengths and shades of gray, arranged in a stepped fashion on the right side of the slide.

Questions?