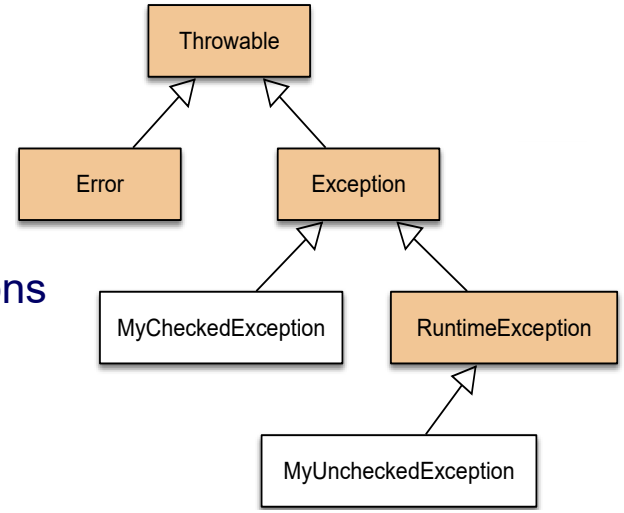
Forelæsning Uge 14

Tillykke
- **I mangler nu kun at aflevere Computerspil 4 og 5 (som de fleste synes er lettere end de foregående)**
- **Husk også de to mundtlige præsentationer**

- **Defensiv programmering**
  - Metoder og konstruktører bør tjekke de parameterværdier, som de kaldes med
  - Derved kan man ofte undgå ulovlige handlinger, såsom at dividere med nul eller tilgå et element, som ikke eksisterer
- **Exceptions**
  - Sprogkonstruktion til rapportering af fejl
  - En kaldt metode kan kaste en exception, som så efterfølgende gribes (håndteres) på det sted, hvor metoden blev kaldt
- **Assertions**
  - Sprogkonstruktion til beskrivelse af betingelser, som man forventer vil være opfyldt på bestemte steder i programmet
- **Fil-baseret input/output**
  - Hvordan læser og skriver man en fil?
  - Område, hvor der let kan ske fejl (forkert filnavn, disk full, no permission , netværksfejl, osv.)
  - Sådanne fejl håndteres elegant ved hjælp af exceptions
- **Afleveringsopgave: Computerspil 4**
  - Modifikation af den grafiske brugergrænseflade

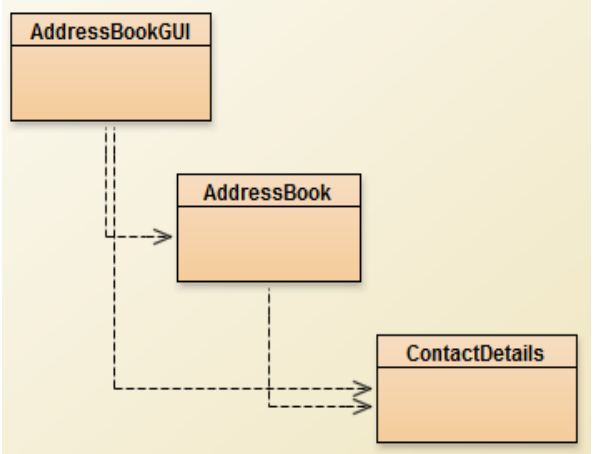


Defensiv programmering

- **I kapitlet om defensiv programmering og exceptions tager BlueJ bogen udgangspunkt i klient/server systemer**
  - Det er dog ikke kun for klient/server systemer, at defensiv programmering og exceptions er relevant
  - De kan bruges overalt, hvor metoder/konstruktører kalder hinanden
- **En server er karakteriseret ved, at den er reaktiv**
  - Serveren gør kun noget, når en klient anmoder om det (webserver, testserver, mailserver, osv.)
  - Serveren gør intet på egen hånd
- **To mulige strategier for programmering af servere (og andre systemer, hvor metoder/konstruktører kalder hinanden)**
  - Vi kan antage, at klienterne ved, hvad de gør, og kun foretager fornuftige og veldefinerede serverkald
  - Vi kan antage, at klienterne indeholder fejl eller med vilje forsøger at udføre illegale serverkald
  - Det kan vi forsvare os imod ved hjælp af defensiv programmering

AddressBook projektet

- **Lad os betragte et program, der implementerer en adressebog**
- **Systemet har tre klasser**
  - AddressBookGUI implementerer programmets grafiske brugergrænseflade
  - AddressBook implementerer programmets øvrige funktionalitet. Klassen indeholder metoder til at tilføje, ændre, fjerne og søge i kontaktinformation
  - ContactDetails indeholder kontaktinformationen, der består af navn, telefonnummer og adresse



- **AddressBook gemmer kontaktinformationerne i et Map, hvor de kan findes ved at bruge enten navnet eller telefonnummeret som nøgle**
- **Et AddressBook objekt er et typisk eksempel på en server**
  - Objektet gør intet på egen hånd
  - Det handler kun, når en klient (GUI-objektet) anmoder om det

removeDetails metoden i AddressBook

- **Parameterværdien bruges til at finde et sæt ContactDetails, hvis indgange derefter fjernes fra Map'en**

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

Find ContactDetails

Fjern de to indgange fra Map'en

Justér antal

- **Ovenstående giver os et problem**
  - Hvis den angivne nøgle ikke er i brug, vil metodekaldet **book.get(key)** returnere **null** – hvilket er OK
  - Men det betyder, at det efterfølgende metodekald **details.getName()** fejler med en **NullPointerException**
- **Det er ikke nødvendigvis forkert**
  - Hvis alle klienter "opfører sig korrekt", og kun kalder get metoden med nøgler, der er i brug, fungerer alt som det skal
  - Men det er en farlig fremgangsmåde – specielt hvis man ikke selv har kontrol over, hvordan klienterne programmeres

Kontrol af parameterværdier

- **Servere er særligt sårbare, når deres konstruktører og metoder modtager værdier via deres parametre**
  - Hvis man ved, at disse parameterværdier er fornuftige, er der ingen grund til at spilde tid på at teste dem
  - I mange tilfælde har man dog ingen eller kun ringe indflydelse på, hvordan klienter programmeres, og så bør alle parameterværdier tjekkes
  - Derved kan man ofte undgå ulovlige handlinger, såsom at dividere med 0, kalde en metode på en variabel, der har værdien null, eller tilgå et element som ikke eksisterer (i en arrayliste eller et array)

Samme metode som før

- **Vi har nu "pakket" de fire sætninger ind i en if sætning**
- **Hvis nøglen er i brug gøres det samme som før**
- **Ellers gør man ingen ting**

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

Rapportering af fejl

- **Hvad bør serveren gøre, hvis den finder en illegal parameter?**
- **Serveren kan undlade at udføre det foretagne request**
  - Det gjorde vi for removeDetails metoden – kan være forvirrende for brugeren
  - Serveren kan (sommetider) ændre parameterværdien til noget "fornuftigt"
- **Serveren kan rapportere fejlen til brugeren (dvs. et menneske)**
  - Serveren kan printe en fejlmeddelelse eller bringe en dialogboks op på skærmen
  - Begge dele har kun effekt, hvis der er en bruger til at se beskeden, og selv da vil de fleste brugere ikke forstå beskeden / vide hvad de skal gøre
  - Hvad vil I gøre, hvis en hæveautomat fortæller jer, at der er en Fejl 5614 eller kaster en NullPointerException?
- **Serveren kan rapportere fejlen til klienten (dvs. et objekt)**
  - Hvis metoden har returtypen void kan denne ændres til boolean, således at legale requests returnerer true og illegale returnerer false
  - Hvis metoden har en non-void returtype kan man returnere en speciel værdi, der ikke er i brug (f.eks. null, 0, eller en negativ værdi)
  - Serveren kan **kaste** en **exception**, som efterfølgende **gribes** af klienten (i den programkode, der kaldte den fejlende metode)

Forskellige strategier til at rapportere fejl

Brug af exception

- Det tjekkes om parameter-værdien er null
- Hvis det er tilfældet skabes et exception objekt, og dette kastes ved hjælp af det reserverede ord throw
- Exception objektets type beskriver fejlen
- Herudover kan der være en tekststreng med yderligere information om fejlen

Brug af returtype

- If sætningen er ændret til en if-else sætning
- Returværdien angiver om operationen lykkedes eller ej

Returtypen er ændret fra void til boolean

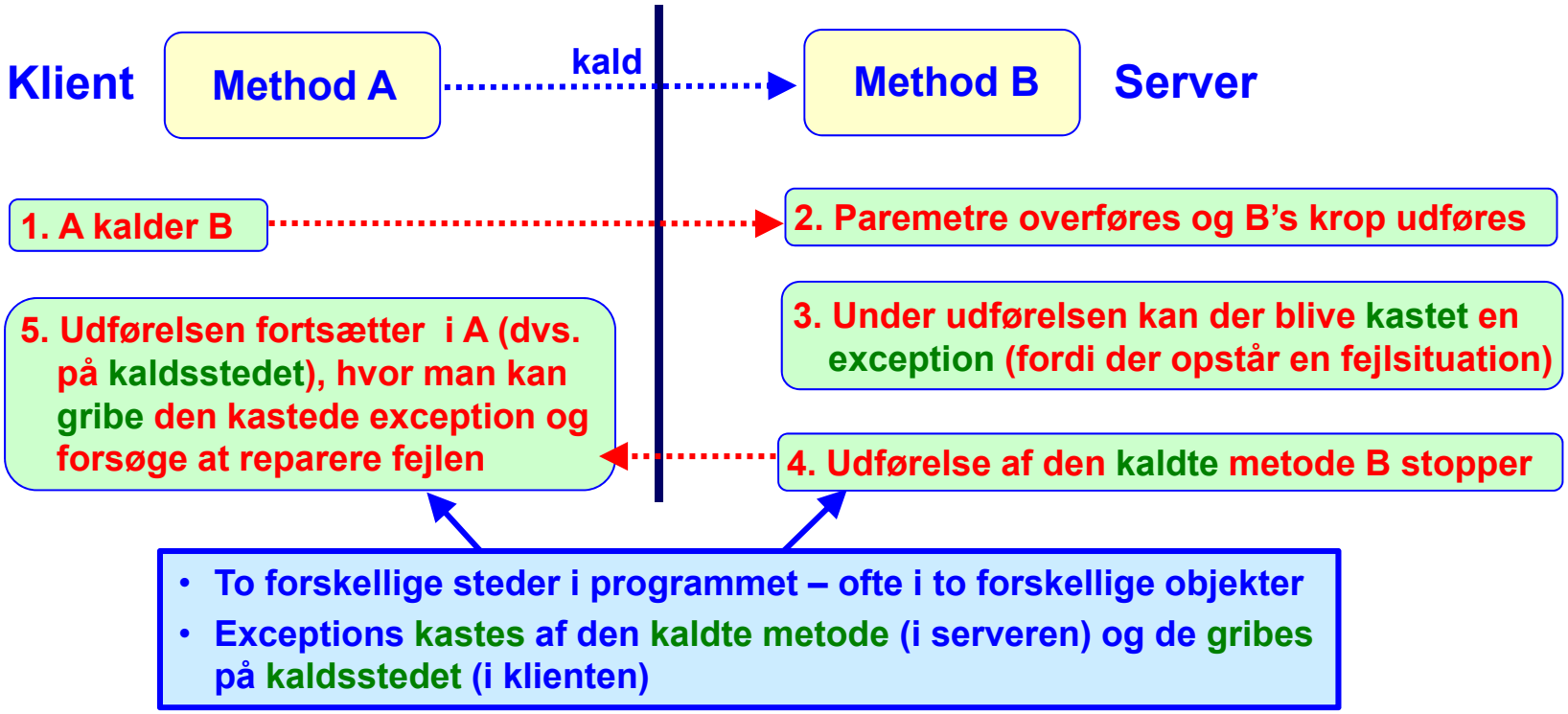
```
public boolean removeDetails(String key) {
  if(key == null){
    throw new IllegalArgumentException(
                  "Null key in removeDetails");
  }
  if( keyInUse(key) ) {
    ContactDetails details = book.get(key);
    book.remove(details.getName());
    book.remove(details.getPhone());
 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

Returtype kontra exception

- Klienter kan undlade at tjekke returværdien
- De har sværere ved at ignorere exceptions
- Derfor er exceptions i mange tilfælde bedst

Exceptions

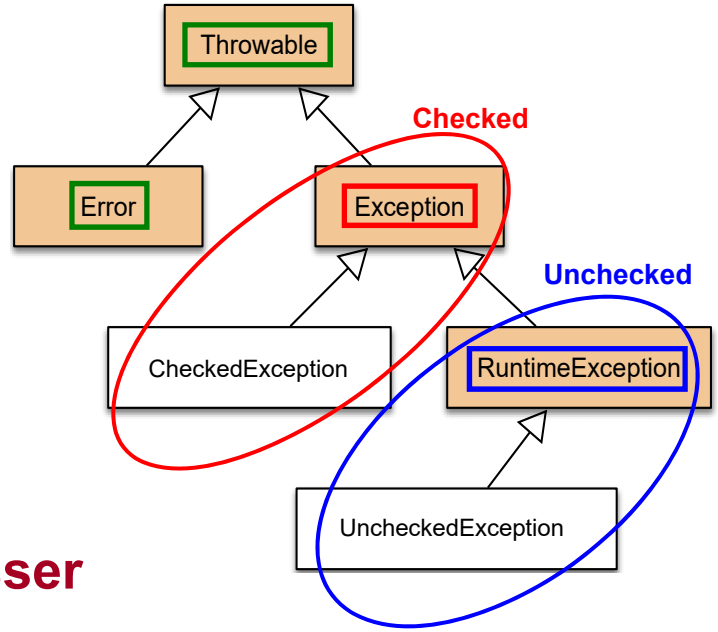
- **Exceptions er en sprogkonstruktion, som tillader klienter at forsøge at overleve/reparere fejl, som rapporteres fra servere**



- **Det er ikke kun i klient/server systemer, at exceptions er nyttige**
  - De kan bruges overalt, hvor metoder/konstruktører kalder hinanden
  - Exceptions kastes i den metode/konstruktør der bliver kaldt
  - Exceptions gribes af den metode/konstruktør, der foretager kaldet

Throwable

- **Alle exceptions er en subtype af Exception klassen**
  - **Error**'s er alvorlige runtime-fejl fejl, som et program ikke med fornuft kan forsøge at overleve, f.eks. AWTError, IOError, VirtualMachineError, AssertionError, osv
- **Der er to slags exceptions**
  - **Checked** exceptions tjekkes af oversætteren, som kontrollerer, at de **håndteres** på kaldsstedet
  - **Unchecked** exceptions (og Errors) er oversætteren ligeglad med



- **RuntimeException og dens subklasser er unchecked exceptions**
  - Alle andre exceptions er **checked**
- **Alle de exceptions, I har set indtil nu, har været unchecked**
  - Ellers ville oversætteren have krævet, at I håndterede dem

Unchecked versus checked exceptions

- **Unchecked exceptions bruges i situationer, hvor fejlen bør føre til, at programmet stopper**
  - Typisk fordi programmøren har lavet en logisk fejl, såsom at dividere med nul, kalde en metode på en variabel, der har værdien null, eller tilgå et element som ikke eksisterer (i en arrayliste eller et array)
  - Sådanne fejl kan undgås – hvis programmøren er kompetent og omhyggelig
- **Checked exceptions bruges i situationer, hvor**
  - fejlen **ikke** skyldes dårligt programmørarbejde, og
  - det giver mening, at den kaldende metode forsøger at reparere fejlen
- **Checked exceptions bruges bl.a. i forbindelse med input/output**
  - Hvis brugeren har angivet et filnavn, som ikke eksisterer, kan man lade brugeren vælge/indtaste et nyt
  - Hvis systemet ikke kan skrive en fil, fordi brugeren har manglende permissions eller disken er fuld, kan man lade brugeren angive et nyt sted at placere filen
- **Programmøren kan ikke undgå, at den slags fejl opstår**
  - Men hun kan forudse dem og specificere, hvordan de skal håndteres

Når en exception kastes

- **Metoden (der kaster en exception) stopper øjeblikkeligt**
  - Efter en throw sætningen udfører metoden ikke flere sætninger (statements)

```
public boolean removeDetails(String key) {
   if(key == null){
      throw new IllegalArgumentException(
                  "Null key in removeDetails");
   }
   if( keyInUse(key) ) {
      ...
      return true;
   }
   else {
      return false;
   }
}
```

Eksemplet fra før

Hvis throw sætningen udføres stopper udførelsen af removeDetails metoden

Kontrollen overføres til kaldsstedet, som kan gribe den kastede exception og forsøge at "reparere" fejlen

- **Hvis der kastes en exception returnere metoden ikke et resultat**
  - Det protesterer oversætteren ikke over
  - Den vil derimod protestere, hvis der lige efter throw sætningen indsættes en return sætning, idet denne aldrig vil kunne blive udført

Kontrol af parameterværdier

- **Unchecked exceptions bruges ofte til at standse programmet, hvis der anvendes en ulovlig parameterværdi**
  - Der gøres intet forsøg på at gribe den kastede exception, hvilket betyder at programmet standser
  - I stedet rettes den logiske fejl i programmet, således at der ikke fremover kastes en exception i denne situation

Det tjekkes om parameteren er null

Det tjekkes om parameteren er den tomme eller en blank tekststreng

Hvis alt er ok, bruges nøglen til at hente den ønskede kontaktinformation (hvis nøglen ikke er i brug returneres null)

```
public ContactDetails getDetails(String key) {
    if(key == null) {
      throw new IllegalArgumentException(
                    "Null key in getDetails");
    }
    if(key.trim().length() == 0) {
      throw new IllegalArgumentException(
                    "Empty key in getDetails");
    }
    return book.get(key);
}
```

- Parameteren til IllegalArgumentException er en tekststreng, der kopieres til den røde fejlmeddelelse i terminalvinduet
- Den kan også aflæses på kaldsstedet (når man griber den kastede exception)

Check af parameterværdier i konstruktør

- **Konstruktører kan også modtage illegale parameterværdier**
  - Her kan exceptions forhindre, at der skabes "sære" objekter

Det tjekkes om en eller flere parameter-værdier er null.

I så fald sættes de til den tomme streng

Feltvariablerne initialiseres

Hvis både name og phone er den tomme tekststreng, kastes en exception

```
public ContactDetails(String name, String phone, String address) {
  if(name == null) { name = ""; }
  if(phone == null) { phone = ""; }
  if(address == null) { address = ""; }
  this.name = name.trim();
  this.phone = phone.trim();
  this.address = address.trim();
  if(this.name.isEmpty() && this.phone.isEmpty() {
    throw new IllegalStateException(
      "Either name or phone must be non-empty");
  }
}
```

Det giver ikke mening at oprette objektet, idet man ikke har nogen fornuftig nøgle, hvormed det kan tilgås (i Map'en)

Checked exceptions

- **En metode, der kan kaste en checked exception, skal angive dette i sit hoved ved hjælp af det reserverede ord throws**
  - Tilladt at anvende throws for unchecked exceptions, men dette anbefales ikke

```
public void saveToFile(String filename) throws IOException {
} ...
```

- **Når man kalder en metode, der kan kaste en checked exception, skal man være parat til at gribe denne ved hjælp af en try-catch sætning**

try blokken indeholder de sætninger, der kan føre til, at der kastes en exception

catch blokken indeholder de sætninger, der skal udføres for at reparere situationen, dvs. gribe den kastede exception

```
try {
   filename = ... // Request filename from user
   addressbook.saveToFile(filename);
   successful = true;
}
catch(IOException e) {
   System.out.println("Unable to save to " + filename);
   successful = false;
}
```

Metodekald, der kan kaste exception

Den exception som kan kastes, og som skal gribes

- Hvis der kastes en exception, overføres kontrollen fra try blokken til catch blokken
- Catch blokken udføres kun, hvis der kastes en exception i try blokken

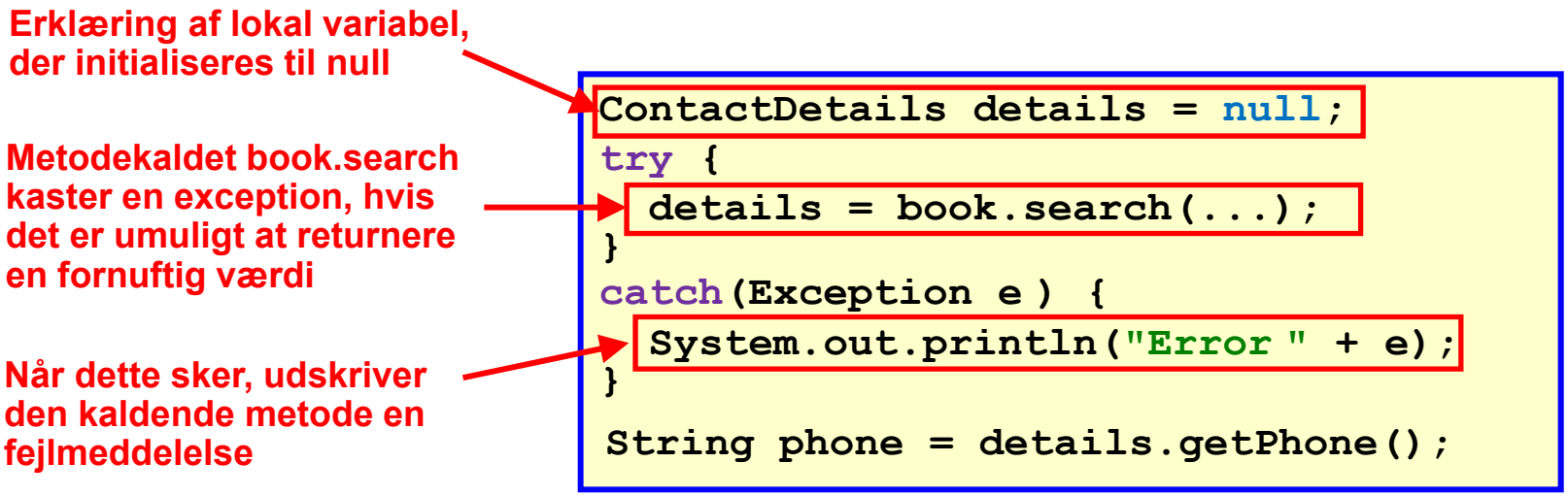
Try-catch sætning

- **Der kan være flere catch blokke**
  - I så fald søges catch blokkene igennem forfra (som i en switch sætning)
  - Den første catch blok, hvor exception typen matcher, udføres
  - Rækkefølgen af catch blokkene er vigtig
  - Hvis catch blokken med IOException flyttes op foran de to andre, vil disse aldrig kunne udføres (hvilket oversætteren vil påpege)
- **Via variablen e kan man få information om, hvad der gik galt**
  - Giver adgang til "detailed message", "stack trace", osv.
- **Der kan også være en finally blok**
  - Denne er placeret efter catch blokkene
  - Indeholder ting, der altid skal udføres, uanset om, der kastes en exception eller ej
  - F.eks. lukning af fil, der er åbnet i try blokken

```
try {
   ...
   object-reference.method(...);
   ...
}
catch (EOFException e) {
   // Handle end of file exception
   ...
}
catch (FileNotFoundException e) {
   // Handle file not found exception
   ...
}
catch (IOException e) {
   // Handle other IOExceptions
   ...
}
finally {
   // Any actions common to whether or
   // not an exception is raised
   ...
}
```

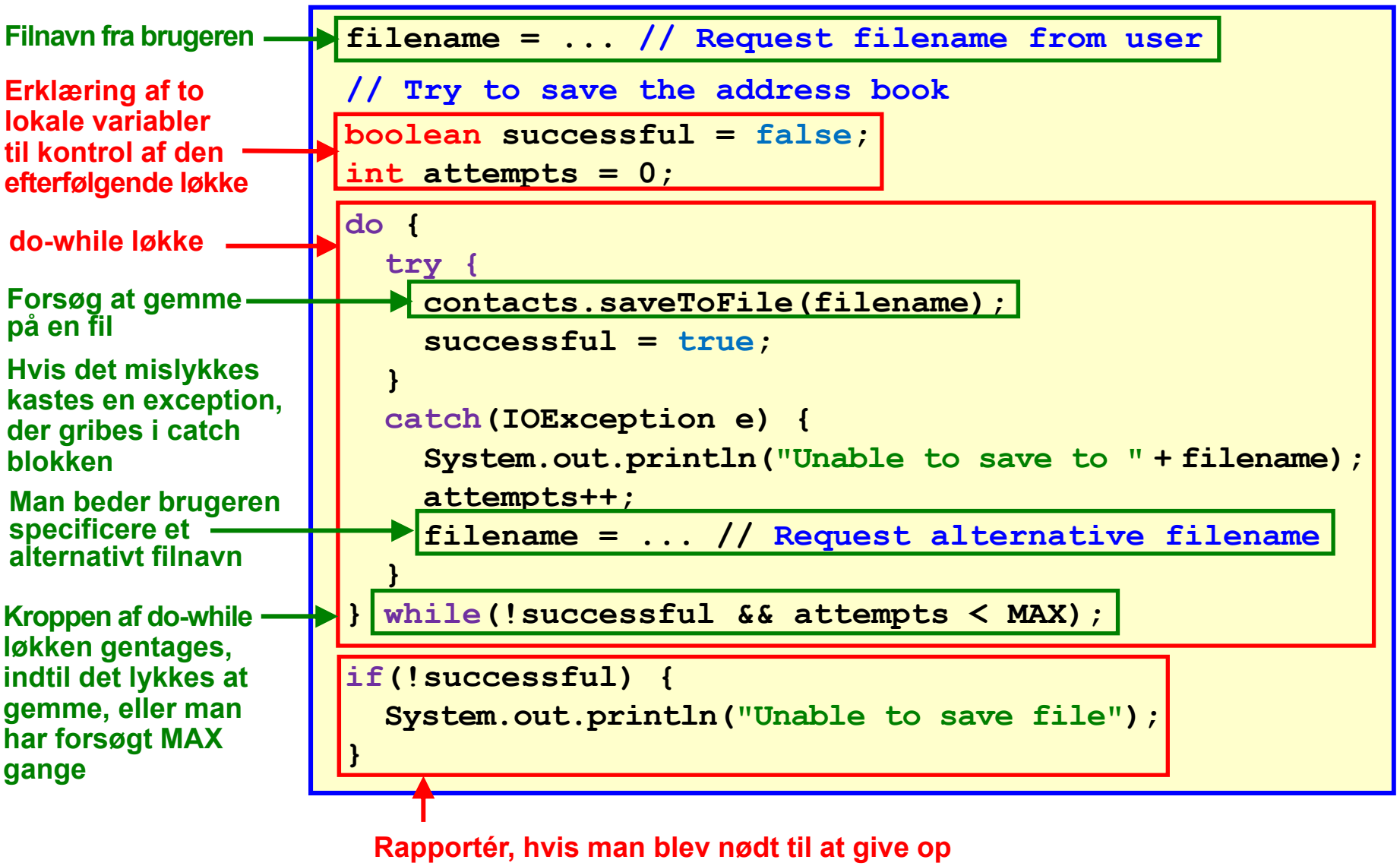
Eksempel på dårlig kode

- **Checked exceptions er kun nyttige, hvis programmøren (på kaldsstedet) forsøger at reparere situationen**
  - Nedenstående try sætning er **ikke** nyttig



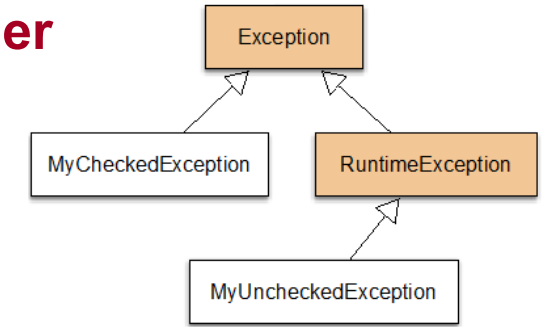
- **Herefter fortsætter man ufortrødent**
  - Men details er stadig **null**
  - Kaldet af **getPhone** giver derfor ikke mening, og vil kaste en NullPointerException

Eksempel på nyttig try sætning



Erklæring af nye exception klasser

- **Man kan definere sine egne exception klasser**
  - Dette gøres ved at lave en subklasse af en eksisterende exception klasse
  - Hvis klassen er en subklasse af RuntimeException, vil dens exceptions være **unchecked,** ellers vil de være **checked**



- **Der er intet nyt i erklæringen af exception klasser**

Checked exception

Feltvariabel

Konstruktør

Accessor metode

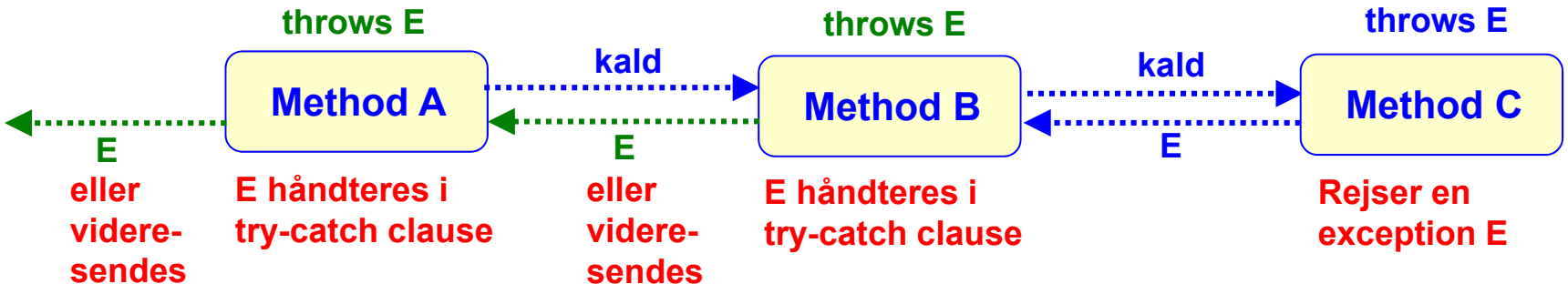
toString metode

```
public class NoMatchingDetailsException extends Exception {
   private String key;
   public NoMatchingDetailsException(String key) {
      super();
      this.key = key;
   }
   public String getKey() {
      return key;
   }
   public String toString() {
      return "No details matching: " + key + " were found";
   }
}
```

- Hvis den kastede exception gribes, kan den "dårlige" key, tilgås via getKey metoden
- Ellers kan den læses i den røde tekst i terminalvinduet, som udskrives via toString metoden

Videresendelse af exceptions (propagering)

- **På kaldsstedet kan man, i stedet for at gribe en exception, videresende den til omgivelserne (propagering)**
  - Det bruges, når en metode er ude af stand til selv at reparere situationen
  - Når en exception videresendes, skal metodens hoved have en **throws clause** med den pågældende exception type (eller en supertype heraf)
- **Videresendelse erstatter håndtering i try-catch clause**
  - Det giver kun mening at gøre én af delene



- **Hvis man når ud på yderste niveau af metodekald, uden at E er blevet håndteret i en try-catch clause, stopper programudførelsen (med rapportering af exception E)**

Checked exceptions

- **Når en metode kan kaste en checked exception, tjekker oversætteren, at alle kaldende metoder, enten indeholder en**
  - **try-catch sætning**, der ”beskytter" kaldet og specificerer, hvordan en kastet exception gribes

    eller

    - **Det giver kun mening at gøre en af delene**
  - **throws clause**, der videresender den kastede exception til omgivelserne
- **Det hedder en checked exception, fordi oversætteren checker, at den pågældende exception håndteres ved at blive grebet eller videresendt**
- **Det er også tilladt at gribe en unchecked exception**
  - Det kan f.eks. give mening, hvis brugeren indtaster noget forkert i en dialogboks, og der rejses en **IllegalFormatException**
  - Man kan så håndtere situationen ved at bede brugeren indtaste en ny værdi (som så forhåbentlig har et korrekt format)
  - Ovenstående er et eksempel på, at man også sommetider bruger **unchecked** exceptions i situationer, der er uden for programmørens kontrol

Pause

Assertions

- **Vi har ofte en forventning om at visse betingelser er opfyldt på bestemte steder i vores program**
- **Efter udførelsen af removeDetails metoden vil vi forvente, at**
  - nøglen, der blev anvendt som parameterværdi, ikke længere forekommer i adressebogen
  - størrelsen af adressebogen er konsistent, dvs. at værdien af feltvariablen numberOfEntries er lig med, antallet af kontaktinformationer

```
public void removeDetails(String key) {
  if(key == null){
    throw new IllegalArgumentException(
                "Null key in removeDetails");
  }
  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";
}
```

- **Dette kan tjekkes ved hjælp af to assertions indeholdende**
  - det reserverede ord assert
  - et boolsk udtryk
  - en (optional) tekststreng som beskriver, hvad der gik galt
- **Hvis det boolske udtryk evaluerer til false stopper programudførelsen med en AssertionError**

Metoderne keyInUse and consistentSize

- I eksemplet på foregående side bestod de to boolske udtryk af to metodekald
- Den første metode er simpel

Bruger containsKey metoden fra Map interfacet

```
private boolean keyInUse(String key) {
  return book.containsKey(key);
}
```

- Den anden metode er lidt mere kompleks

Lokal variabel, der initialiseres til at indeholde alle værdierne fra AddressBook

Værdierne kopieres over i en mænge (Set), hvorved dubletter elimineres

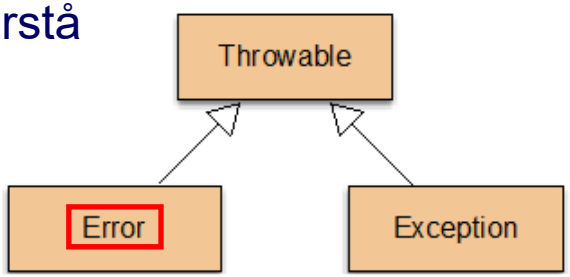
```
private boolean consistentSize() {
  Collection<ContactDetails> allEntries = book.values();
  Set<ContactDetails> uniqueEntries = new HashSet<>(allEntries);
  return numberOfEntries == uniqueEntries.size();
}
```

Tjek at antal elementer i mængden stemmer overens med værdien af feltvariablen numberOfEntries

Bemærk, at vi i de to erklæringer bruger Collection<…> og Set<…> i stedet for at angive den præcise implementering via f.eks. ArrayList<…> og HashSet<…>

Brug af assertions

- **En assertion sætning opfylder to formål**
  - Den **beskriver** en betingelse, som vi forventer er opfyldt på det sted, hvor assertion sætningen er indsat – en sådan betingelse kaldes en **invariant** og gør vores kode mere læselig og lettere at forstå
  - Under programudførelsen kan det **tjekkes**, at betingelsen virkelig er opfyldt, og hvis dette ikke er tilfældet kastes en **AssertionError** (subklasse af Error)



- **Assertions kan let slås til og fra – de vil typisk være slået**
  - **til** mens programmet udvikles og testes
  - **fra** når programmet anvendes af brugere (produktionsmode)
- **BlueJ's testfaciliteter bruger assertions**
  - Metoderne **assertEquals**, **assertTrue** og **assertFalse**, som I har brugt i forbindelse med jeres testmetoder, er implementeret ved hjælp af en **assert** sætning
  - De rejste **AssertionErrors** kan inspiceres i BlueJ's testvindue

Kursusevaluering

- **Vi forsøger løbende at forbedre kurset ved at lave små justeringer**
  - Til dette formål er den feedback, som I giver os via kursusevalueringen, særdeles nyttig
- **Jeg opfordrer derfor kraftigt til, at I bruger tid på at deltage i denne (og andre) kursusevalueringer**
  - Det er en nem måde at få indflydelse på
  - Det forbedrer forholdene for fremtidige studerende
  - Alle kursusevalueringer gennemgås af institutledelsen samt ledelsen af uddannelsesudvalget
- **For at forbedre svarprocenten har fakultetet besluttet, at der skal afsættes tid ved forelæsningerne til at besvare kursusevalueringen**
  - Derfor vil I nu få 10 minutter at gøre det
  - Kursusevalueringen kan besvares frem til onsdag den 27. november kl. 23.59 (se "Vigtig meddelelse" herom eller mail fra studieadministrationen)
  - Resultatet af kursusevalueringen vil blive gennemgået og diskuteret ved den afsluttende forelæsning mandag den 2. december

Fil-baseret input/output

- **Hvordan læser og skriver man en fil?**
  - Område, hvor der let kan ske fejl, som er helt uden for programmørens kontrol (forkert filnavn, disk fuld, manglende permission, netværksfejl, osv.)
  - Sådanne fejl håndteres ved hjælp af **checked** exceptions
- **Java's oprindelige support for input/output findes i pakken java.io**
  - Denne indeholder en lang række subklasser, som supporter input/output operationer
  - Herudover defineres **IOException**, som er en checked exception
  - IOException her mere end 30 subklasser heriblandt **FileNotFoundException** and **EOFException** (EOF ≈ end of file)
- **Senere versioner har tilføjet java.nio pakken (NIO ≈ New IO)**
  - Java.nio har en række tilhørende pakker såsom java.nio.file og java.nio.charset
  - Klasserne i nio pakkerne erstatter delvis klasserne i java.io pakken (på samme måde som Swing delvist erstatter AWT)

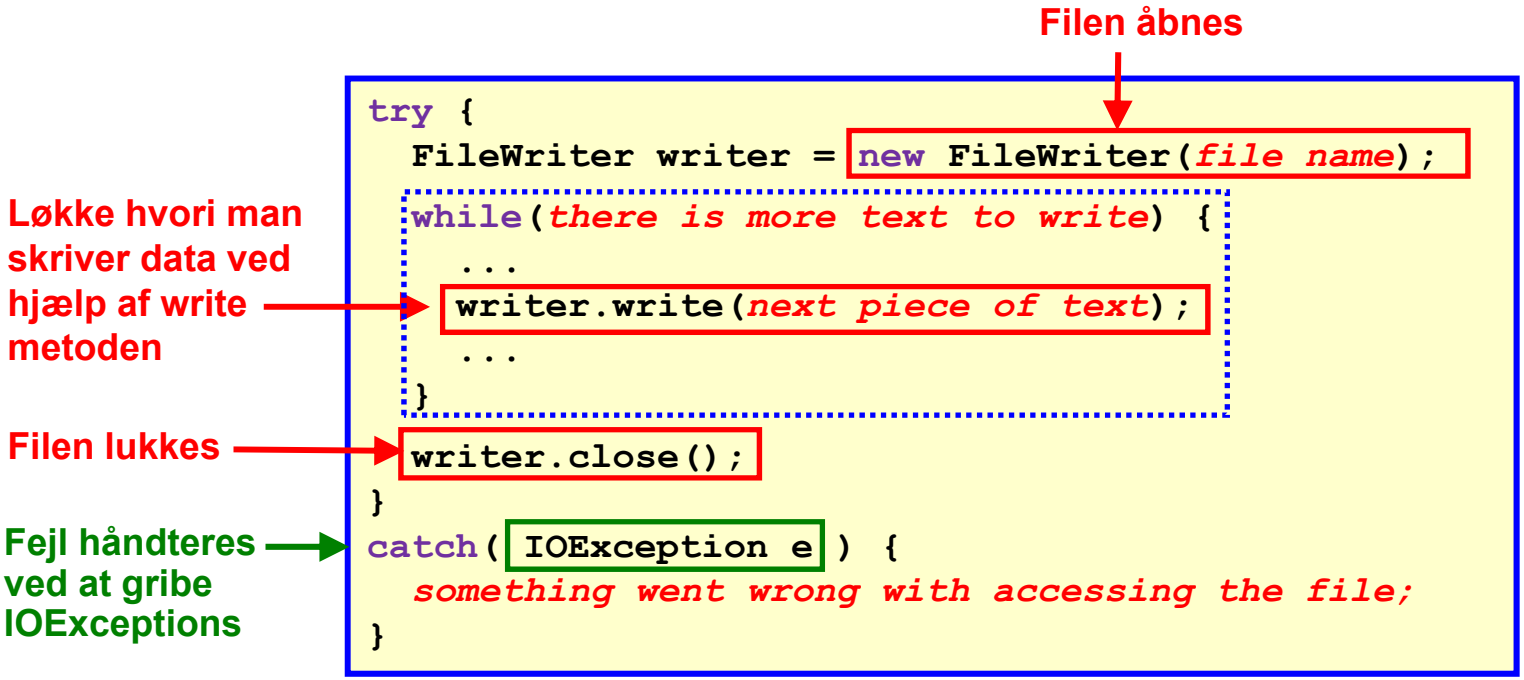
Files og streams

- **Input/output opdeles i**
- **Tekstfiler**
  - Indeholder tegn-baseret information i en form, som kan læses og forstås af mennesker – f.eks. html filer, Java programfiler og dokumentationsfiler
  - Svarer til **char** og **String** typerne
  - Håndteres ved hjælp af **readers** og **writers** såsom FileReader, BufferedReader og FileWriter
- **Streams**
  - Indeholder binær information såsom billeder eller eksekverbare programmer
  - Svarer til **byte** typen
  - Streams håndteres ved hjælp af **stream handlers**
  - Bemærk at disse streams **intet** har med de streams at gøre, som I kender fra funktionel programmering

File klassen og Path interfacet

- **En fil er ikke blot et navn og noget indhold**
  - Filer ligger i foldere (directories)
  - De indeholder information om størrelse, hvem der ejer dem, hvem der kan tilgå/ændre i dem, hvornår de blev skabt og sidst er ændret, om de er skjulte (hidden), osv.
- **java.io pakken indeholder File klassen, som har en lang række metoder til at understøtte ovenstående**
  - Et **File** objekt indeholder information om en fils egenskaber (men indeholder ikke filens indhold)
  - Ved hjælp af et **File** objekt kan man f.eks. undersøge, om en fil eksisterer, og på den måde undgå at fremprovokere en FileNotFoundException
- **Tilsvarende indeholder java.nio.file pakken Path interfacet og Files klassen (som er mere moderne)**

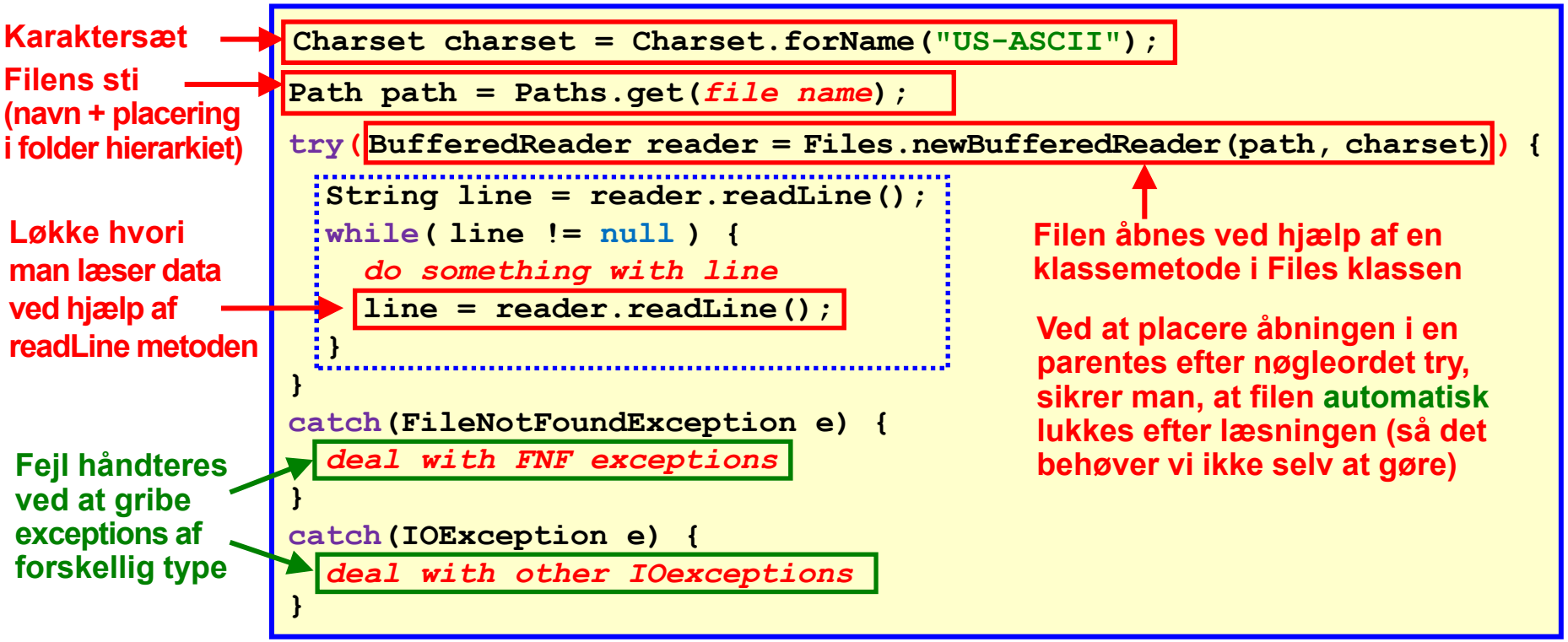
File output består af tre skridt



- **Alle tre skridt kan fejle – af forskellige grunde**
  - Mange af disse er fuldstændig udenfor programmørens kontrol (såsom en disk, der er fuld eller ødelagt eller et ugyldig filnavn opgivet af brugeren)
  - Fejl håndteres ved hjælp af IOException (og dens subklasser)
  - Læs detaljer i BlueJ bogen (de egner sig ikke til en forelæsning)

File input består af de samme tre skridt

- Nu beskrevet i lidt større detaljer med brug af faciliteter fra nio



- Alle tre skridt kan fejle – af forskellige grunde
  - Mange af disse er fuldstændig udenfor programmørens kontrol
  - Fejl håndteres ved hjælp af IOException (og dens subklasser)
  - Læs detaljer i BlueJ bogen (de egner sig ikke til en forelæsning)

Files klassen (java.nio) og Scanner klassen (java.util)

- **Files klassen indeholder klassemetoden lines**
  - Ved at bruge den kan man helt undgå at bruge klassen BufferedReader
  - Metoden tager en parameter af typen Path og returnerer en Stream<String> (som kendt fra afsnittet om funktionel programmering)

```
Stream<String> myStream = Files.lines(path);
```

  - Den skabte stream kan behandles via funktionel programmering eller konverteres til et array eller en arrayliste

- **Scanner klassen indeholder metoder til at opbryde teksten fra en fil i delkomponenter**
  - F.eks. returnerer nextInt et heltal ved at læse de næste tegn på filen
  - Ved at kalde hasNextInt før kaldet af nextInt, kan man undgå, at der kastes en exception, hvis de næste tegn ikke udgør et heltal
  - Der er analoge metoder for de andre typer, f.eks.:
    - nextLine, nextDouble, nextBoolean og nextByte
    - hasNextLine, hasNextDouble, hasNextBoolean og hasNextByte

System klassen

- **System klassen indeholder tre klassevariabler**
  - **in** af typen **InputStream** standard input stream
  - **out** af type **PrintStream** standard output stream
  - **err** af type **PrintStream** standard error output stream

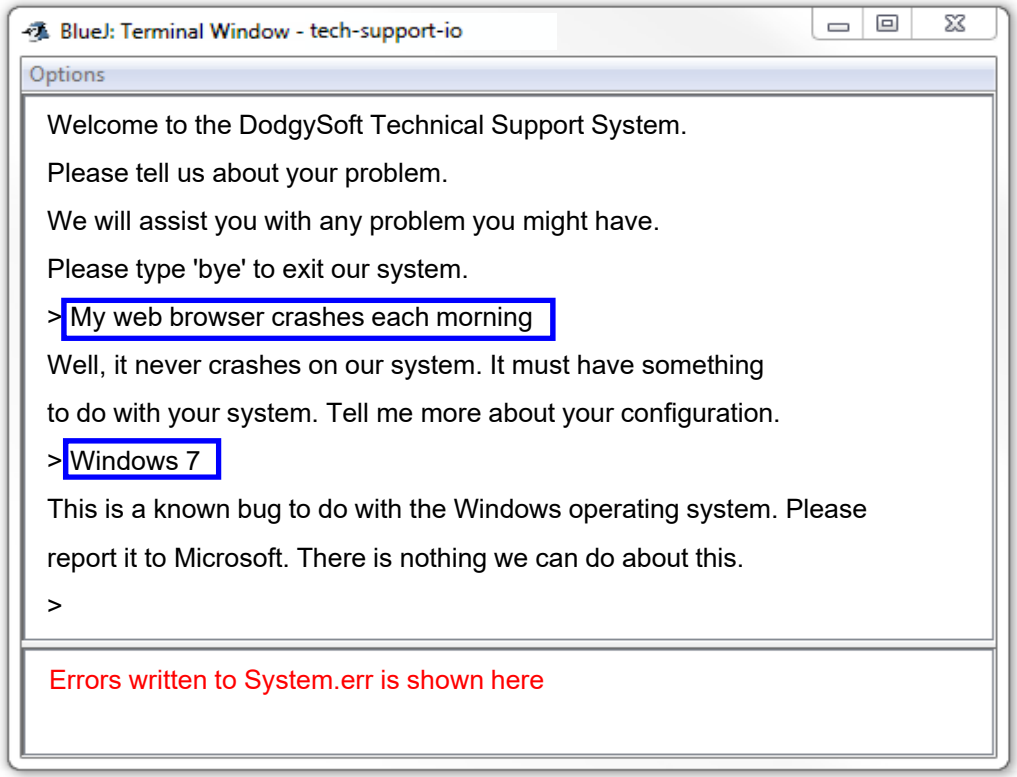
- **I BlueJ er alle tre knyttet til terminalvinduet**

Input via System.in →

Output via System.out →

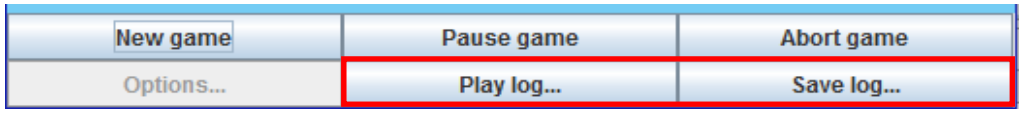
Errors via System.err →

I kan skrive i denne del af terminalvinduet via System.err.println("…")

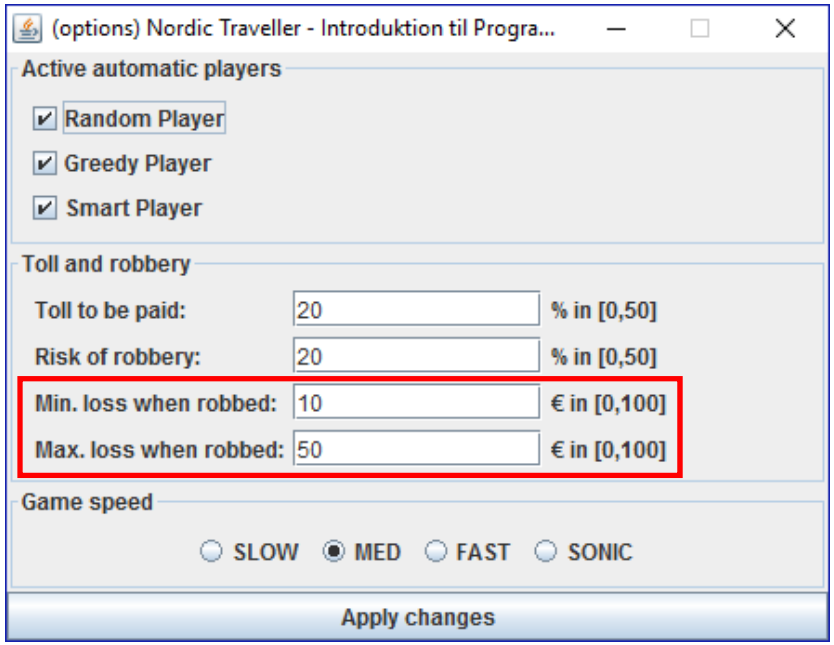


Afleveringsopgave: Computerspil 4

- I den fjerde delaflevering skal I bruge nogle af de ting, som I har lært om grafiske brugergrænseflader
- Først skal I tilføje to ekstra knapper til panelet nederst i vinduet



- Dernæst skal I udvide Options dialogboksen, med nogle ekstra valg



I begge tilfælde kan I se, hvordan koden for de eksisterende knapper/ valg ser ud, og derefter kopiere denne med oplagte småændringer

Computerspil 4 (fortsat)

- Til sidst skal I tilføje en menubar samt nogle tastaturgenveje

Log menuen indeholder to indgange Play log… og Save log…

Tastaturgenveje

Game menuen er hierarkisk, idet en af menuindgangene selv er en menu

- Kode for menuindgangen New game

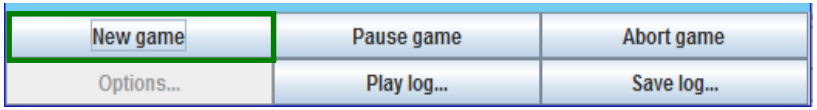
For Slow, Medium, Fast og Sonic skal man skrive CTRL i stedet for CTRL_SHIFT

Skab menuindgang og tilføj den til Game menuen

Tilføj tastaturgenvej via metode i JMenuItem

Håndtering af events

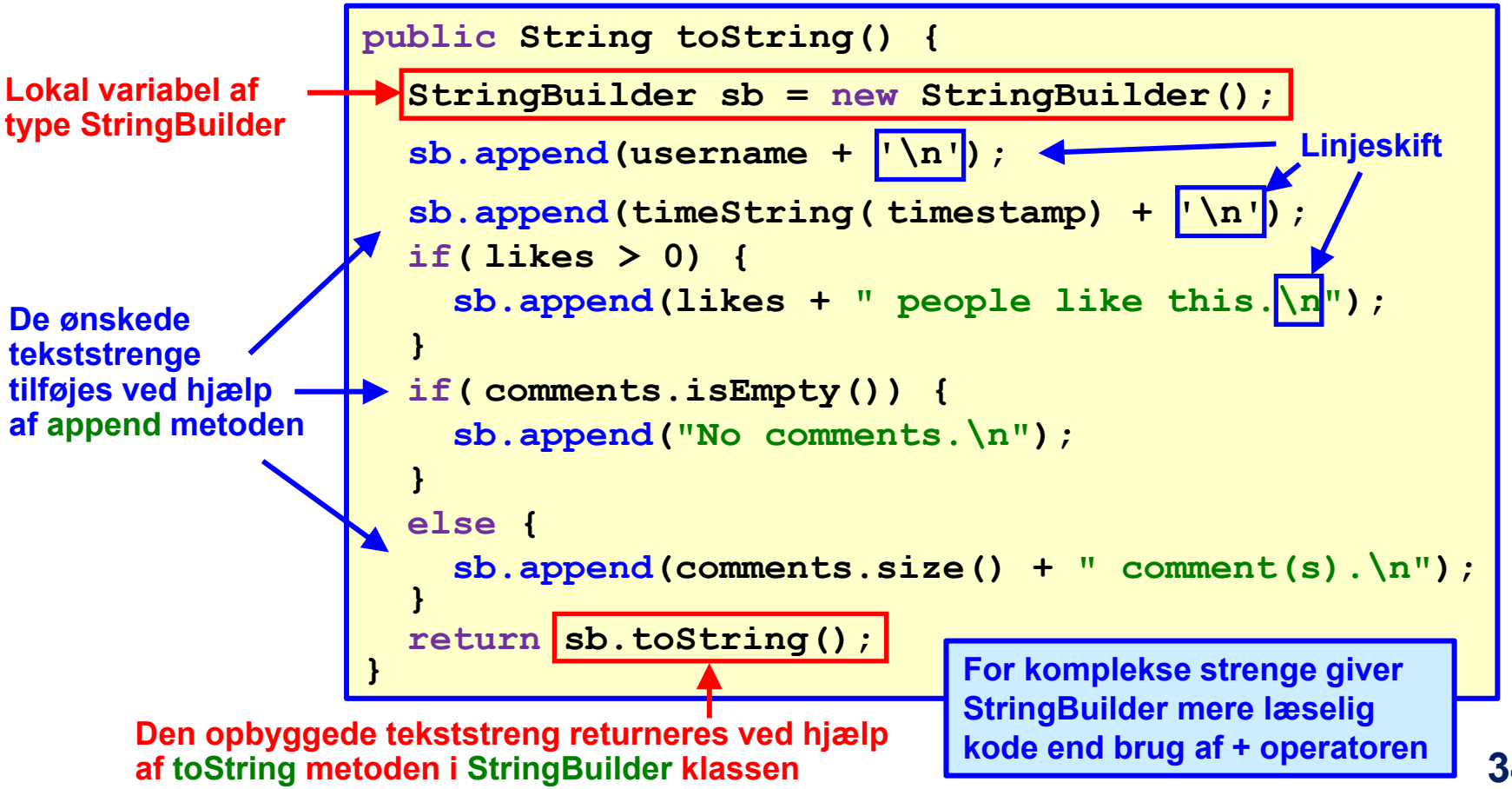
```
JMenuItem newGameItem = new JMenuItem("New game");
gameMenu.add(newGameItem);

newGameItem.setAccelerator(KeyStroke.getKeyStroke(
                                KeyEvent.VK_N, CTRL_SHIFT));

newGameItem.addActionListener( e -> newGameButton.doClick());
```



Når der modtages et event, skal der ske det samme, som når brugeren trykker på New game knappen (doClick er en metode i AbstractButton)

StringBuilder klassen (fra java.lang)

- **I opgave 3 skal fejl rapporteres til brugeren via en dialogboks**
  - Fejlene opsamles via **StringBuilder** klassen
  - Herunder illustreres brugen af StringBuilder ved at lave en toString metode for Post klassen i Network projektet fra kapitel 11




```
public String toString() {
  StringBuilder sb = new StringBuilder();
  sb.append(username + '\n');
  sb.append(timeString(timestamp) + '\n');
  if(likes > 0) {
    sb.append(likes + " people like this.\n");
  }
  if(comments.isEmpty()) {
    sb.append("No comments.\n");
  }
  else {
    sb.append(comments.size() + " comment(s).\n");
  }
  return sb.toString();
}
```

Opsummering

- **Defensiv programmering**
  - Metoder og konstruktører bør tjekke de modtagne parameterværdier, således at de kan undgå at udføre ulovlige handlinger
- **Exceptions**
  - Sprogkonstruktion til rapportering af fejl
  - En kaldt metode kan kaste en exception, som så efterfølgende kan gribes på det sted hvor metoden blev kaldt
- **Assertions**
  - Sprogkonstruktion til beskrivelse af betingelser, som man forventer vil være opfyldt på bestemte steder i programmet (invarianter)
  - Betingelserne kan testes under programudførelsen
- **Fil-baseret input/output**
  - Område, hvor der let kan ske fejl (forkert filnavn, disk full, no permission , osv.)
  - Sådanne fejl håndteres elegant ved hjælp af exceptions
- **Afleveringsopgave: Computerspil 4**
  - Modifikation af den grafiske brugergrænseflade

Husk at deltage i kursusevalueringen
Sidste dag er onsdag den 27. november

Husk træningen i mundtlig præsentation
- **Den er uhyre vigtig for jeres succes ved mundtlig eksamen**
- **Nu kan I med stor fordel se den sidste video, der handler om grafiske brugergrænseflader**

Det var alt for nu….. … spørgsmål