Undantag

och lite IO

Input/ output

- Java I/O baseras på ***streams*** (strömmar)
- En ström är en ***sekvens av bytes*** som flödar från en källa till ett mål
- Fördefinierade ***standardströmmar***:

ström	syfte	enhet	datatyp
`System.in`	läsa input	tangentbord	`InputStream`
`System.out`	skriva output	bildskärm	`PrintStream`
`System.err`	skriva fel	bildskärm	`PrintStream`

- Standard strömmarna kan läggas om med `System.setIn/Out/Err` metoderna

`InputStream`

- Inputströmmar är mera komplicerade
- `InputStream` är en abstrakt klass
  - Mer om abstrakta klasser senare under kursen
- `InputStream` har väldigt begränsad mängd metoder för att hantera inläsning
- `System.in` måste därför ofta ”kapslas in” i en mer specifik ström med de egenskaper man vill ha
  - Läsa in strängar
  - Läsa in heltal
  - Läsa in …
- Vi kommer titta på några olika klasser vi kan använda till detta tex `Scanner`

Scanner

- Osmidigt att läsa in en hel rad som sträng! Bättre med ett”ord” i taget.
- Vad är ett ”ord”?
- Tecknen måste grupperas till s.k. Tokens
- Man kopplar en scanner till en ström

```
Scanner myScan = new Scanner(System.in);
```

- Finns metoder för att läsa in heltal, flyttal, ord, rader etc. Se metoder i API-beskrivningen
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html
- För att `Scanner` ska kunna användas så måste klassen importeras då den till skillnad från tex `String` inte ligger i paketet java.lang.
  - Lägg till raden `import java.util.Scanner;` ovanför deklarationen av klassen i filen för att få det att funka.

Fel och felhantering

- Ett program kan fungera korrekt i normal- fallet men vara känsligt för störningar
  - Felaktiga indata (från användaren/andra program)
  - Yttre omständigheter
- Bra program är förberedda på att fel kan uppstå

Hur upptäcker man fel?

- *Partiella funktioner*: Funktioner som inte är definierade för vissa inputvärden
- Fel uppstår ofta då metoder får oväntade indata.
  - Kontrollera indata i metoden innan något görs
  - Kontrollera parametrarna innan metoden anropas
    - Vad passar bäst?

Felhantering

- Vissa fel kan inte förebyggas med hjälp av if-satser
  - En fil kan t ex raderas under skrivningen/läsningen, fast man har kollat i förväg att allt är OK
- Möjliga åtgärder:
  - Avbryt eller avsluta programmet
  - Ignorera felet och fortsätt
  - Skriv ut ett meddelande och fortsätt
  - Ignorera anropet som ledde till felet
  - Tolka anropet på ett meningsfullt sätt
  - Kräver åtgärd från användaren interaktivt
  - Producera ett felaktigt resultat
  - Returnera en felkod (går ej alltid)
  - ...

Hur bör man hantera fel?

- Det finns många alternativ
- Det finns ingen åtgärd som är bäst i alla lägen
- Beror på anroparens mål

➡ Låt anroparen bestämma

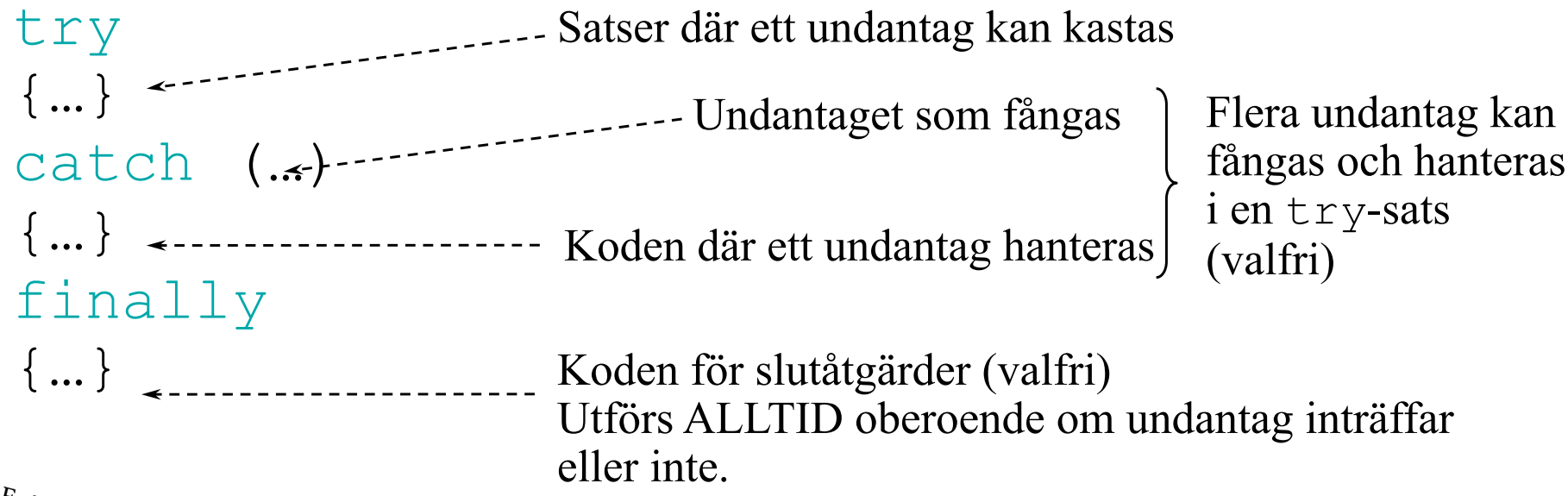
➡ Throw/ raise exception

➡ "Kasta"/ "flagga" ett undantag

➡ Den anropade metoden överlåter åt anroparen att genomföra lämplig åtgärd

Undantag

- En ***exception*** är ett objekt som signalerar ett undantag
- `throw`-satsen kastar ett undantag
  … som kan fångas m h a en `try`-sats



Hantera undantag

- Då ett undantag kastas så bryts den normala exekveringsordningen och man går upp i anropskedjan tills man hittar en matchande `catch`.
  - Felet kan alltså hanteras på den nivå i anropskedjan där man har bäst möjlighet att lösa problemet
    - Och vi behöver ej göra något för varje steg i denna.
  - Efter att ett fel hanterats i en catch-sats så fortsätter exekveringen som normalt efter catch-satsen som hanterade problemet.
  - Fångas inte felet av vår kod så avbryts programmet (ett runtime fel uppstår.

Fånga undantag

```
int a;
try {
   a = b / c;
}
catch(ArithmeticException e) {
   a = 9999;
}
```

Kasta Undantag

- Exceptions kastas med hjälp av `throw`-satsen
- Throw-satsen finns i normala fall i en if-sats som kollar ifall undantaget skall kastas.
- Exempel:

```
throw new IllegalArgumentException();
```

Två kategorier av undantag

- ***Checked exceptions***
  - Fel som vanligtvis inte beror på fel i logiken av programmet
  - Fel som kan uppträda under "normala" förhållanden
  - Måste fångas (dvs kod som kan kasta en checked exception måste skrivas i en `try`-sats) eller kastas vidare med metoden, t ex

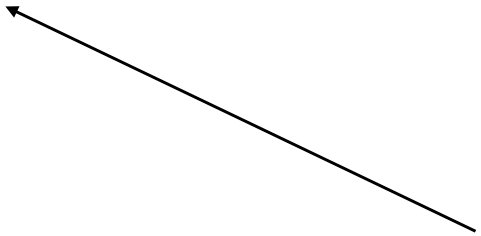
  ➡ `IOException`
- ***Unchecked exceptions***
  - Fel som brukar beror på fel i logiken av programmet
  - Fel som kan uppträda under "normala" förhållanden
  - Får ignoreras (men det är god programmeringsstil att fånga eller deklarera det), t ex

  ➡ `ArrayIndexOutOfBoundsException`

Mer om undantag - deklaration

- Exempel:

```
public void readFile (String fileName)
    throws IOException {
    …
}
```



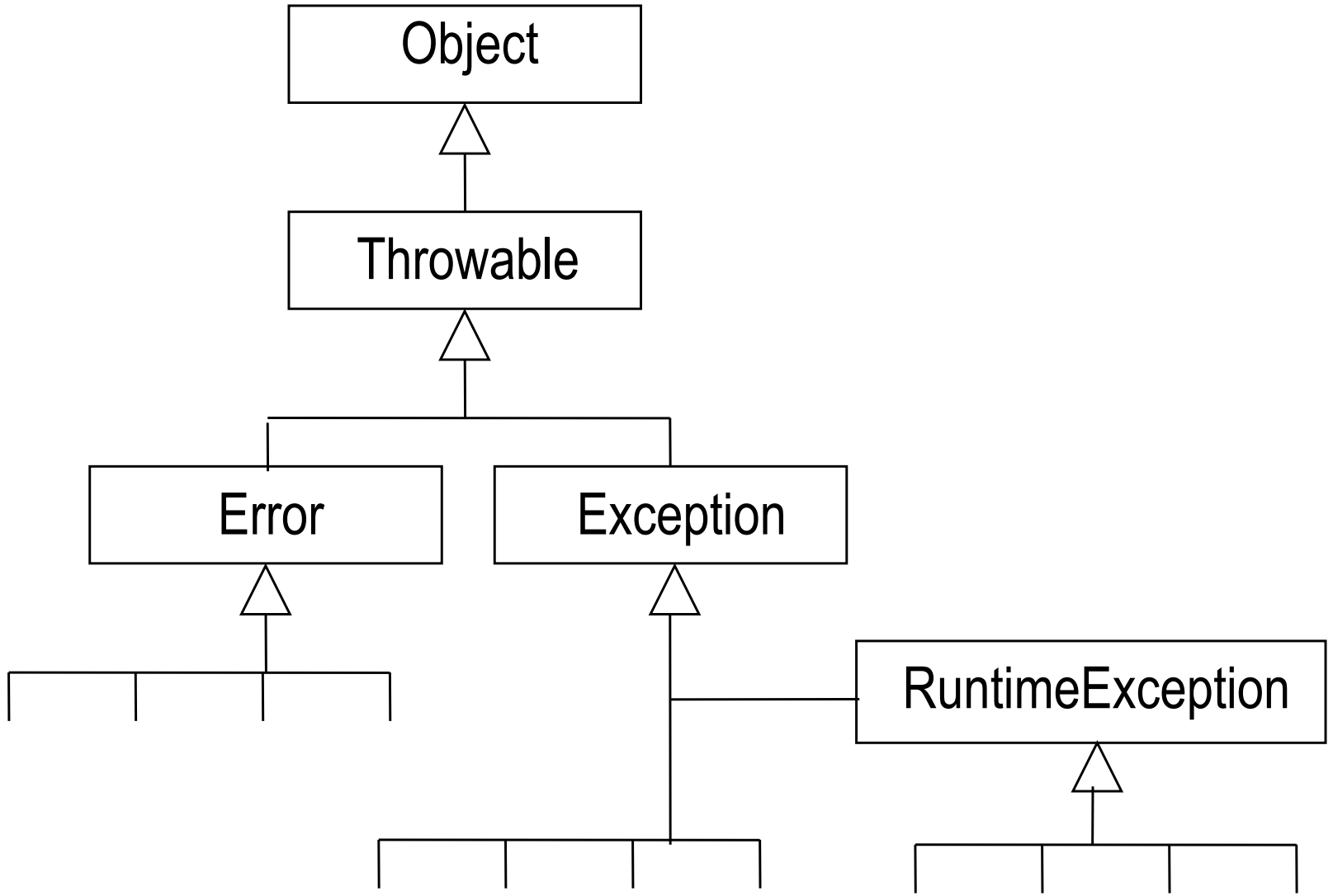
Anger att metoden kastar vidare det kontrollerade undantaget IOException

- Krävs för kontrollerade undantag. För Okontrollerade är det valfritt att göra

Kontrollerade undantag

- Ett undantag är antingen kontrollerat(*checked)* eller okontrollerat (*unchecked)*
- Ett kontrollerat undantag kan endast kastas i ett try-block eller i en metod som explicit säger att den kastar detta undantag.
- Kompilatorn kommer att klaga om ett kontrollerat undantag inte hanteras korrekt.
- Ett okontrollerat undantag kräver ingen särskild hantering, men kan hanteras på samma sätt.
- Okontrollerade undantag är underklasser till `Error` eller `RunTimeException`. Övriga är kontrollerade.

Två kategorier av undantag



Egna undantag

- Man kan definiera egna undantag genom att ärva från någon av de befintliga undantagsklasserna
  - Ofta så behöver man inte lägga till så mycket extra kod då typen är det viktiga för att kunna signalera att en ny typ av fel har inträffat och kunna hantera detta typ av fel separat från andra typer av fel

Några vanliga undantagsklasser

- `IllegalArgumentException`
  - En metod har tagit emot en parameter som ej är korrekt.
- `IOException`
  - Något har gått fel vid läsning/Skrivning av data
- `ArrayOutOfBoundsException`
  - Man har indexerat sig utanför en array
- `NullPointerException`
  - Vi försöker göra något via en referens-variabel som ej är tilldelad något objekt utan har värdet `null`.