

Oversigt

- Indkapsling
- Enum
- Switch
- Exceptions og Exception handling



BRUGERINPUT



Scanner-klassen

- Brugerinput kan læses ind i et program på flere måder
 - Via en fil
 - Via en grafisk brugergrænseflade
 - **Via konsollen**
- I sidste tilfælde anvendes "Scanner" fra java.util-pakken.



Sådan bruges Scanner klassen

```
import java.util.Scanner
```

• • •

```
public class TextInput{
```

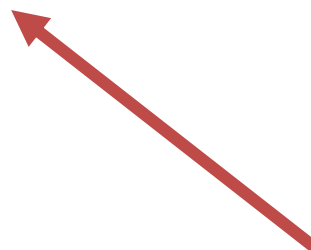
```
Scanner scanner = new Scanner(System.in);
```

```
String nextString = scanner.nextLine();
```

```
double nextDouble = scanner.nextDouble();
```

```
int nextInt = scanner.nextInt();
```

}



Udfordringer med Scanner klassen

- Eller historien om brugere der gør hvad det passer dem

- Fejlen: `java.util.InputMismatchException`
- Hvis brugerens indtastning ikke stemmer overens med metoden der er angivet
 - `nextInt()` \leftarrow input skal være et heltal
 - `nextBoolean()` \leftarrow input skal være true eller false (og, måske overraskende, ikke eks. 1 eller 0)
- Håndtering af dette kommer vi til i slutningen af lektionen



INDKAPSLING



Indkapsling

- Access modifiers bruges til at styre synligheden af attributter
- Instansattributter bør være `private`
 - Altså at det kun er objekter, der selv har direkte adgang til sine attributter
- Indkapsling vil sige, at et objekt skjuler sit state (sine data i instansvariable) og tilbyder/implementerer relevant adfærd i form af instansmetoder
- Et af de fire fundamentale principper bag OOP
 - De andre er abstraktion, arv og polymorfi



Accessor metoder

- Andre objekter bruger public accessor metoder til at **hente** værdien af private instansvariable
- Nomenklatur:
 - get + navnet på attributten
 - is + navnet på attributten (for boolean)
- Klassen Person har en attribut, der hedder firstName
 - Accessor metoden kaldes getFirstName()
- Klassen Person har en boolsk attribut, der hedder home
 - Accessor metoder kaldes isHome()



Mutator metoder

- Andre objekter bruger public mutator metoder til at ændre værdien af private attributter på et objekt
- Navngivningsstandard:
 - set + navnet på instansvariablen
- Klassen Person har en attribut, der hedder lastName
 - Mutator metoden kaldes setLastName(String newLastName)



Statements i accessor og mutator metoder

- Implementeringen er ikke begrænset til en linjes kode

```
public void setTitle( String newTitle ) {  
    if ( newTitle == null )    // Don't allow null strings as titles!  
        title = "(Untitled)"; // Use an appropriate default value instead.  
    else  
        title = newTitle;  
}
```



ENUMS



Enumerated types

- En speciel type klasse, der bruges til at begrænse udfaldsrummet for en given variabel

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

- Variable af typen Season kan enten være spring, summer, fall eller winter
- Hvorfor med store bogstaver? Hvornår bruger vi store bogstaver?



Enumerated types

- Enum værdier er konstanter – de kan ikke ændres.
 - Derfor bør værdierne skrives med stort
 - Guideline - ikke syntaks
 - Udfaldsrummet/de mulige værdier i enum-typer kaldes også enum constants
- Syntaks for enum:

```
enum <enum-type-name> { <list-of-enum-values> }
```



Enumerated types

- Enums er klasser
 - enum værdier er objekter
- Dvs. de har metoder og attributter
 - Eks. `ordinal()` – position i values-listen
 - Yderligere metoder kan defineres i Enum-klassen



Enumerated types

```
public enum CommandWord
{
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    private String commandString;

    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    public String toString()
    {
        return commandString;
    }
}
```



Enumerated types

```
private HashMap<String, CommandWord> validCommands;
```

```
...
```

```
validCommands = new HashMap<String, CommandWord>();  
    for(CommandWord command : CommandWord.values()) {  
        if (command != CommandWord.UNKNOWN) {  
            validCommands.put(command.toString(), command);  
        }  
    }  
}
```

```
...
```



Hvorfor bruge enums?

- Gør programmer mere læsbar da værdier har meningsfulde navne
- Forbygger fejl og flere tjek da de foreskriver et udfaldsrum for en given type
- Kan eksempelvis bruges i switch-statements – ser vi på om lidt



SWITCH STATEMENT



Switch

- Multiway branching statement
 - alternativ til if... else if... else konstruktionen
- Afhænger af en *expression* til at afgøre hvilken *case*, der kan udføres
- *Expression* kan være af typen int, short, byte, String eller enums



Switch

```
switch (⟨expression⟩) {
    case ⟨constant-1⟩:
        ⟨statements-1⟩
        break;
    case ⟨constant-2⟩:
        ⟨statements-2⟩
        break;
    .
    .    // (more cases)
    .
    case ⟨constant-N⟩:
        ⟨statements-N⟩
        break;
    default:    // optional default case
```

expression: den variabel der testes

case: keyword der bruges i switch konstruktioner

<constant-1> værdien af expression, i hvilket tilfælde statements eksekveres

Default: tilgås hvis ingen af de opstillede hvis expression ikke er lig nogle af de opstillede cases



Switch med enum

```
public class EnumTest {  
    private Day day;  
  
    public EnumTest(Day day)  
    { this.day = day; }  
  
    public void tellItLikeItIs() {  
        switch (day) {  
            case MONDAY:  
                System.out.println("Mondays are bad.");  
                break;  
  
            case FRIDAY:  
                System.out.println("Fridays are better.");  
                break;  
  
            case SATURDAY: case SUNDAY:  
                System.out.println("Weekends are best.");  
                break;  
  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```



Switch med enum

```
public class Starter {  
    public enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
                     FRIDAY, SATURDAY, SUNDAY }  
  
    public static void main(String[] args) {  
        EnumTest firstDay = new EnumTest(Day.MONDAY);  
        firstDay.tellItLikeItIs();  
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);  
        thirdDay.tellItLikeItIs();  
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);  
        fifthDay.tellItLikeItIs();  
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);  
        sixthDay.tellItLikeItIs();  
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);  
        seventhDay.tellItLikeItIs();  
    }  
}
```



EXCEPTIONS



Robusthed

- Et *robust* program er et, der kan “overleve” når der sker noget uventet
 - Mistet netforbindelse
 - Kan ikke læse fra fil (måske åbnet andet sted?)
 - Uventet brugerinput
 - ...
- Vi kunne manuelt programmere robusthed ved at teste for alt
 - Men det er omfattende!



Exceptions

- Vi kategoriserer *typer* af exceptions, og håndterer dem når de opstår
 - Måden hvorpå de skal håndteres afhænger af typen af exception
- Exception Handling
 - At håndtere *Exceptions*
- “Exception” og ikke “Error”
 - Error: Lad programmet dø!
 - Exception: Se om du kan rette op på det!
 - Som udgangspunkt




Exception Throwing og Catching

- Når der opstår noget uventet siger vi, at en exception *kastes*
 - *Exception Thrown*
- Exceptions bliver altid grebet (caught)
 - Af os selv i vores (eller andres) kode
 - Af Java Interpreter (den, der afvikler Java koden)
- Det sidste er smart
 - Programmet dør – ikke maskinen
 - Fejlen bliver indkapslet i Java miljøet.



Exception Throwing og Catching

- Det der “kastes” er et Object
 - Værdien af attributterne i dette objekt beskriver blandt andet
 - *Call stack*
 - *Beskrivelse*
 - Call stack
 - Viser os et spor hertil hvor fejlen er sket, fra hvor vi står.
 - Beskrivelse
 - Beskriver nærmere hvad der gik galt.
- 



Stack trace og beskrivelse

java.lang.UnsupportedOperationException: This method has not been implemented yet. ← Beskrivelse

at dk.sdu.mmmi.lmsimpletest.ExceptionExample.method2(ExceptionExample.java:34)
at dk.sdu.mmmi.lmsimpletest.ExceptionExample.method1(ExceptionExample.java:30) ← Stack trace
at dk.sdu.mmmi.lmsimpletest.ExceptionExample.main(ExceptionExample.java:21)

- Og hvordan får man så fat i et objekt der kastes?
 - Man griber det ->



EXCEPTION HANDLING



Exception Handling

- ...ved hjælp af et try...catch statement.

```
try {
    double determinant = M[0][0]*M[1][1] -
        M[0][1]*M[1][0];
    System.out.println("The determinant of M is " +
        determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("M is the wrong size to have a
        determinant.");
    e.printStackTrace();
}
```



Exception Handling

- Vi kan catche flere ting
 - Det giver mening, når der kan ske flere typer af fejl, der skal korrigeres på forskellig måde.

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " +
        determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
    System.out.println("M is the wrong size to have a
determinant.");
}
catch ( NullPointerException e ) {
    System.out.print("Programming error! M doesn't exist.");
}
```



Exception Handling

- Vi kan også sammenskrive

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException | NullPointerException e ) {
    System.out.println("Sorry, an error has occurred.");
    System.out.println("The error was: " + e);
}
```

- Eller bruge arv

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( RuntimeException err ) {
    System.out.println("Sorry, an error has occurred.");
    System.out.println("The error was: " + err);
}
```



Exception Handling

- Ved sammenskrivning og særligt i arv skal vi være opmærksom på
 - Selvom Exceptions nedarver fra samme type skal de ofte håndteres forskelligt.
 - Det vil sige: Forskellig catch logik, til at rette op på fejlen.
 - **Ingen catch all.**
 - Hvorfor?

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch (Exception err ) {
    System.out.println("Sorry, an error has occurred.");
    System.out.println("The error was: " + err);
}
```

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch (Throwable err ) {
    System.out.println("Sorry, an error has occurred.");
    System.out.println("The error was: " + err);
}
```

