

Videregående programmering (02324)

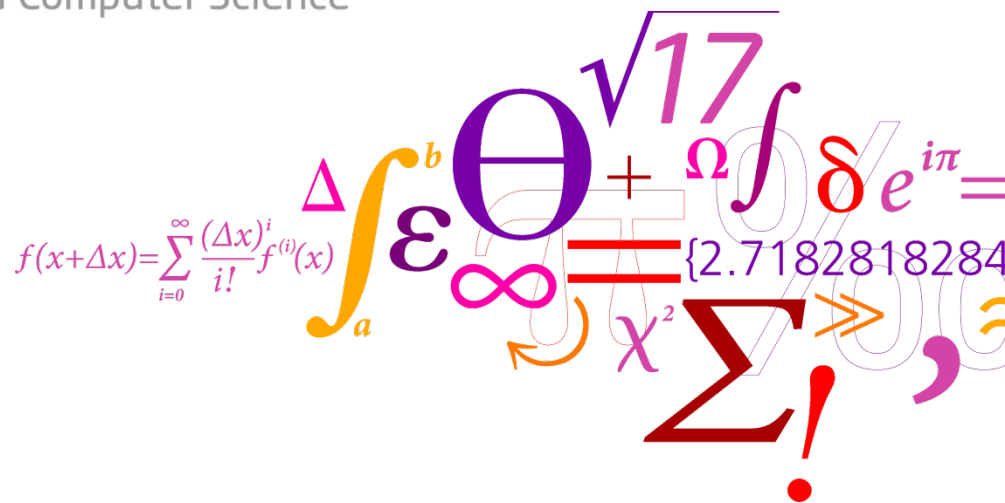
Projekt i software-udvikling (02362)

Forår 2021

Ekkart Kindler

DTU Compute

Department of Applied Mathematics and Computer Science

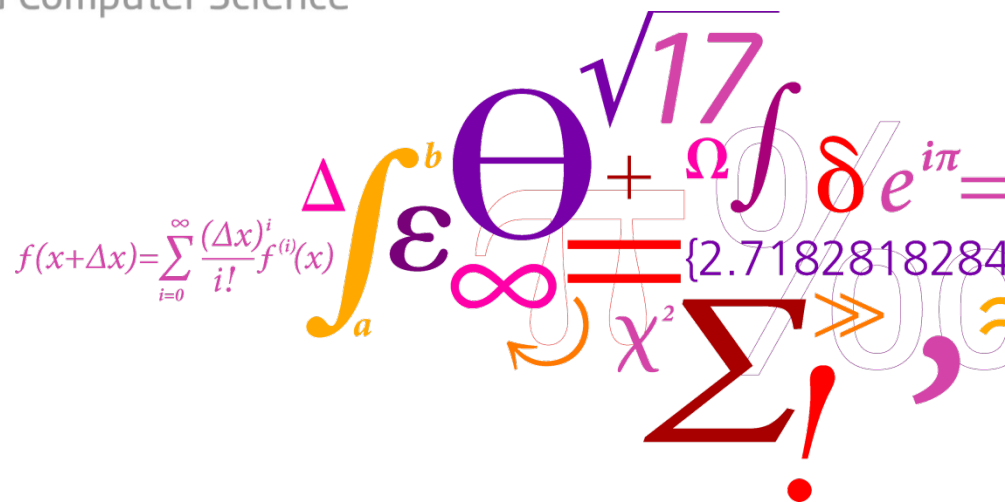


V. Rapportskrivning (Softwaredokumenter)

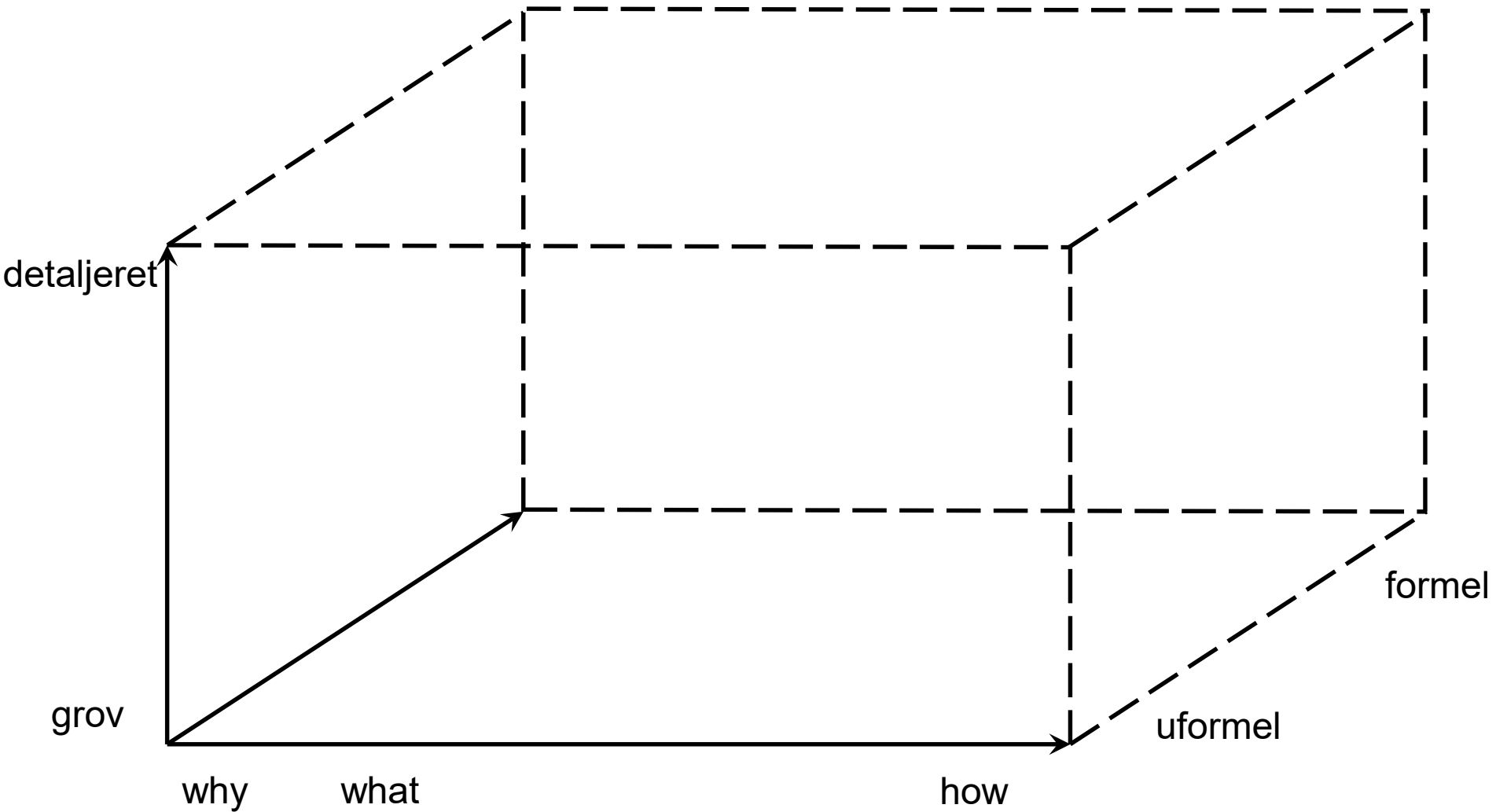
Spørgsmål
→ live-møde

DTU Compute

Department of Applied Mathematics and Computer Science



A collage of mathematical symbols and formulas in various colors. The symbols include: a large purple Θ , a blue $\sqrt{17}$, a red δ , a green $e^{i\pi}$, a yellow ∞ , a blue χ^2 , a red Σ , a green $!$, a blue $\int_a^b$, a red ε , a blue Δ , a green Ω , a red $+$, a blue $=$, a green $\{2.7182818284\}$, a blue $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$, a red $\int_a^b$, a blue ε , a green Δ , a blue Ω , a red $+$, a blue $=$, a green $\{2.7182818284\}$, a blue $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$, a red $\int_a^b$, a blue ε , a green Δ , a blue Ω , a red $+$, a blue $=$, a green $\{2.7182818284\}$, a blue $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$.



- I skriver rapporten ikke for jer selv eller Ekkart!
- I skriver rapporten til en som står udenfor projektet (især censor)!
- Der er brug for at introducere konteksten og hvad RoboRally og dens begreber og regler er!

- Spiralformskrivning ($\rightarrow$ tavle)
- Undgå indforståede afsnit
(fortæl alt – ingen implicitte antagelser)
- Diskuter en idé (et argument) ad gangen
(og lav pointen eksplicit)
- Brug enkle begreber og korte sætninger
- Et begreb = et ord (det samme ord hele vejen igennem)
- Brug entalsform (singularis) hvis muligt

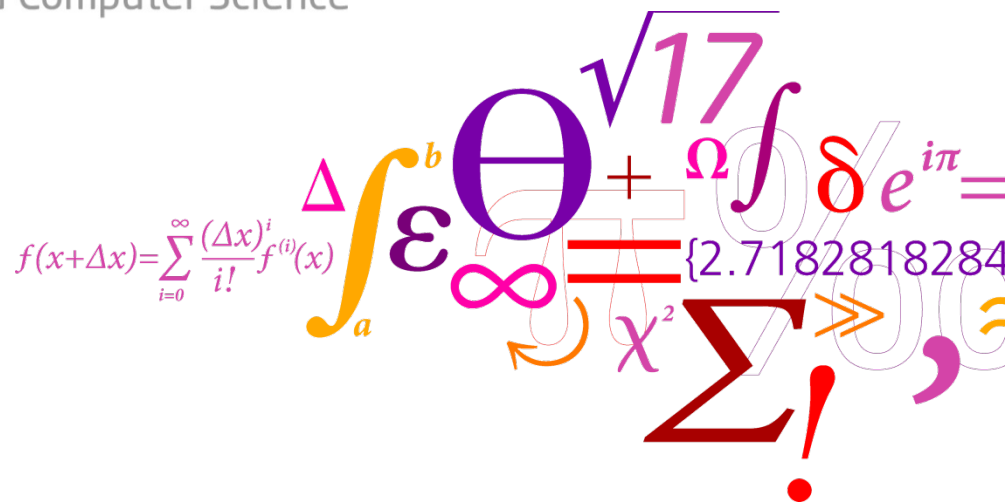
- @author tags i koden og rapporten
(helst studienummer s123456 ellers fuld navn)
- I koden:
 - på klasseniveau (for nye klasser)
 - på metode niveau (hvis eksisterende klasser ændres)
 - rækkefølgen af @author tags betyder noget
- I rapporten:
 - på afsnit eller underafsnitsniveau
 - nogle afsnit kan have flere autorer

IV. Java Praksis

Fortsættes fra tidligere

DTU Compute

Department of Applied Mathematics and Computer Science



A collage of colorful mathematical symbols and expressions. It includes a large purple Θ , a yellow integral $\int_a^b$, a pink infinity ∞ , a red summation $\sum$, a blue square root $\sqrt{17}$, a red delta δ , a blue $e^{i\pi}$, a yellow ϵ , a pink Ω , a blue χ^2 , a red $!$, a yellow $\gg$, a blue $\{2.7182818284\}$, and a pink $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$.

Derfor kigger vi nu kort på Javas indbyggede datastrukturer, især

- Collection (Set, List, Queue, Deque),
- Iterator,
- Comparable og
- Map

Siden Java 8 eksisterer der også streams med streams aggregate funktioner I Java, som er et meget elegant koncept (som man kender fra den funktionelle programmering). Det kommer vi desværre ikke ind på.

...

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class Board extends Subject {
```

...

```
private List<Player> players = new ArrayList<Player>();  
private Player current;
```

List er faktisk et interface (og kan derfor ikke instantieres)!

ArrayList er en klasse som implementerer interface **List**.

...

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class Board extends Subject {
```

...

```
private List<Player> players = new ArrayList<Player>();  
private Player current;
```

ArrayList og List er **generiske**, dvs. de har en typeparameter (fx. **Player**) som siger hvilken type alle listens elemeter skal have.

```
public void addPlayer(@NotNull Player player) {  
    if (player.board == this && !players.contains(player)) {  
        players.add(player);  
        notifyChange();  
    }  
}
```

```
public Player getPlayer(int i) {  
    if (i >= 0 && i < players.size()) {  
        return players.get(i);  
    } else {  
        return null;  
    }  
}
```

List (og dermed **ArrayList**) har en masse nyttige operationer:

- contains
- size
- add
- get
- remove
- ...

Se <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

```
for (int i = 0; i < players.size(); i++) {  
    System.out.println(players.get(i).getName());  
}
```

```
for (Player player: players) {  
    System.out.println(player.getName());  
}
```

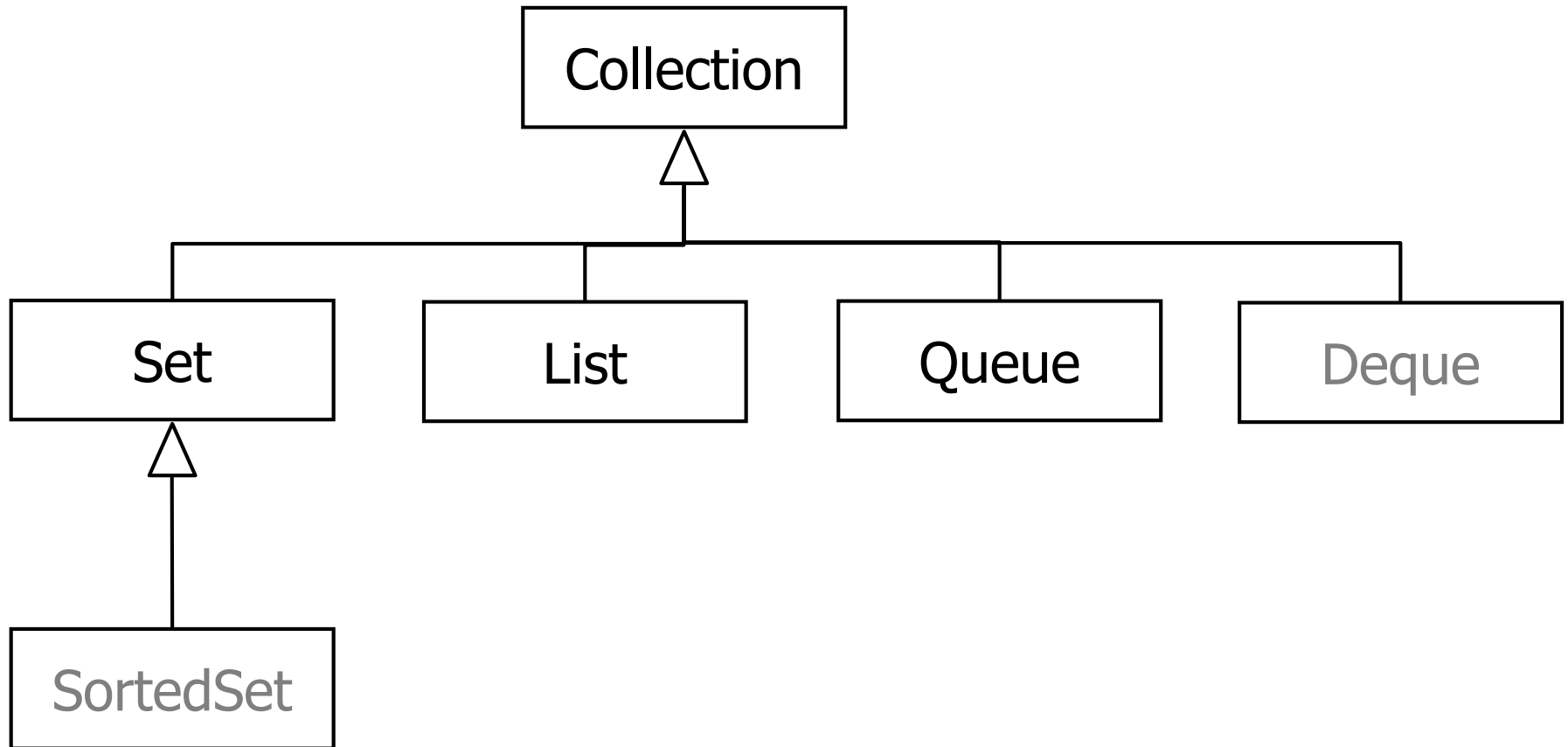
Denne form af løkken er mere elegant og sikkert. Men man må ikke direkte eller indirekte ændre listen imens løkken er i gang

→ `ConcurrentModificationException`

Nogle gange bruges der også lister (eller andre samlinger) uden dens type parameter! Fx

```
List players = new ArrayList();
```

- Der var ingen generiske datatyper inden Java 1.5, så at man måtte programmere på en sådan måde.
- Alle elementer er "bare" objekter (**Object**).
- Det er meget besværligt og tilbøjeligt til at lave fejl!
- En generisk datatype, som bliver brugt uden dens typeparameter, kaldes også rå type (**raw type**)
- **Raw types skal I ikke bruge i jeres software! Ellers vil jeg høre et rigtigt godt argument!**



er en samling af objekter af en bestemt type **E** med følgende (hoved-)metoder:

```
void clear();  
boolean add(E e);  
boolean remove(Object o);  
int size();  
boolean isEmpty();  
contains(Object o);
```

Bemærk at for nogle implementering kan det være, at nogle metoder ikke er implementeret, fordi de ikke giver mening. Så kan de kaste en **UnsupportedOperationException**

Og mange flere:

Se <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

```
boolean addAll(Collection<? extends E> c)
```

```
boolean removeAll(Collection<?> c)
```

```
Iterator<E> iterator()
```

...

Det kommer vi tilbage til, når vi diskuterer `Iterator`.

Collections har en metode, som producer en iterator; iteratoren tillader så at iterere over alle elementer af selve kollektionen.

Iterator er en interface med to hovedmetoder:

boolean hasNext() ;

Returner **true** hvis iteratoren har stadig elementer tilbage.

E next() ;

Returner det næste element og så skifter et element videre. Hvis den kaldes, når **hasNext()** returner **false**, kastes en exception.

Med en iterator kan man iterere over alle dens elementer fx:

```
Collection<E> c = ...
```

```
Iterator<E> i = c.iterator();
```

```
while (i.hasNext()) {
```

```
    E e = i.next();
```

```
    // do something with e, e.g.
```

```
    System.out.println("> " + e);
```

```
}
```

Faktisk er det endnu nemmere at iterere over alle elementer af en Collection.

```
Collection<E> collection = ...
```

```
for (E element : collection) {  
    // do something with e, e.g.  
    System.out.println("> " + element);  
}
```

Konkret eksempel

```
Collection<Integer> collection = ...
```

```
int sum = 0;
```

```
for (Integer value : collection) {
```

```
    sum = sum + value;
```

```
}
```

Hvis I itererer over alle elementer af en collection på denne måde, så undgår I mange muligheder for fejl.

... men man må ikke ændre samlingen under selve løkken!!

- er et struktur som repræsenterer en partiel funktion (afbildning) fra værdier af klassen **K** (for key/nøgle) til værdier af klassen **V** (for value/værdi)
- **map.get(k)** returnerer den aktuelle værdi af **map** for nøglen **k**
- kan nemt opdateres med operation **m.put(k,v)**
- Siden Java 8 har maps også en *convenience method*: **map.getOrDefault(k,d)**

Returner **null**, hvis **map** ingen værdi har for nøglen **k**

Returner **d** (default), hvis **map** ingen værdi har for nøglen **k**

- Vi bruger en map af type **Map<String, Integer>** til at beregne hvor ofte et navn forekommer i en liste af navne (Strings)

- Vi antager:

```
List<String> names = new ArrayList<String>();  
names.add("Emilie");  
names.add("Jens");  
names.add("Tim");  
names.add("Tom");  
names.add("Kurt");  
names.add("Jens");  
names.add("Anna");  
names.add("Jens");  
names.add("Emilie");  
names.add("Jens"); ...
```

Der er mere elegante måder at danne en konstant liste som denne her. Fx

```
List<String> names =  
    Arrays.asList("Emilie", "Jens", "Tim", ... );
```

Map<K,V>: Eksempel

```
Map<String,Integer> nameCount =  
    new HashMap<String,Integer>();
```

Her bruger vi (meget typisk) **HashMap** implementeringen af **Map**

```
for (String name: names) {  
    nameCount.put(  
        name,  
        nameCount.get(name)+1 );  
}
```

Itererer over alle navne; virker da liste er en collection.

```
for (String name: nameCount.keySet()) {  
    System.out.println(name + ": " + nameCount.get(name));  
}
```

Adder 1 på den nuværende værdi for navnet og gem det under samme navn

Virker bare ikke!!

```
Map<String,Integer> nameCount =  
    new HashMap<String,Integer>();
```

```
for (String name: names) {  
    nameCount.put(  
        name,  
        nameCount.getDefault(name,0)+1 );  
}
```

Adder 1 på den
nuværende værdi for
navnet og gem det
under samme navn;
hvis der ingen værdi
er, så vælg 0 som
default.

```
for (String name: nameCount.keySet()) {  
    System.out.println(name + ": " + nameCount.get(name));  
}
```

HashMap og nogle andre af Javas datastrukturer virker kun, hvis klassen **K** til nøgler (keys) er implementeret rigtigt: dvs.

- metoden **equals()** er implementert så at den er en ækvivalentsrelation (se næste slides)
- **hashCode()** metoden er kompatibel med identitet (se næste slides)

Man siger at der er kontrakter for disse metoder. Og nogle af Javas datatyper (som fx **HashMap<K, V>**) virker kun hvis de der kontrakter er overholdt.

For alle objekter `o1`, `o2` og `o3` skal altid gælde:

- *Refleksivitet*:
`o1.equals(o1)`
- *Symmetri*:
hvis `o1.equals(o2)` så gælder også
`o2.equals(o1)`
- *Transitivitet*:
hvis `o1.equals(o2)` og `o2.equals(o3)`
så gælder også `o1.equals(o3)`

Der er nogle flere krav, men dem diskuterer vi ikke her (se Javadoc for `Object`).

For alle objekter `o1` og `o2` skal altid gælde:

- *Kompatibilitet:*

hvis `o1.equals(o2)` så gælder også
`o1.hashCode() == o2.hashCode()`

Når man ændrer metoden `equals()` i sine egne klasser, så skal man også tilpasse `hashCode()` metoden – især når man bruger dem som nøgler i strukturer som baserer på hashing.

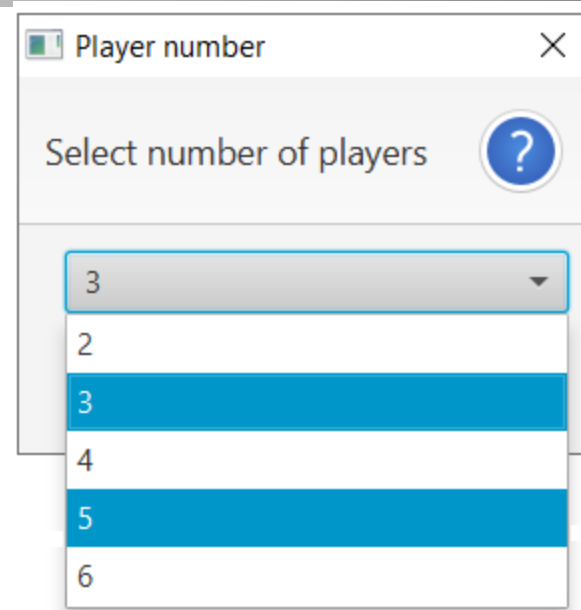
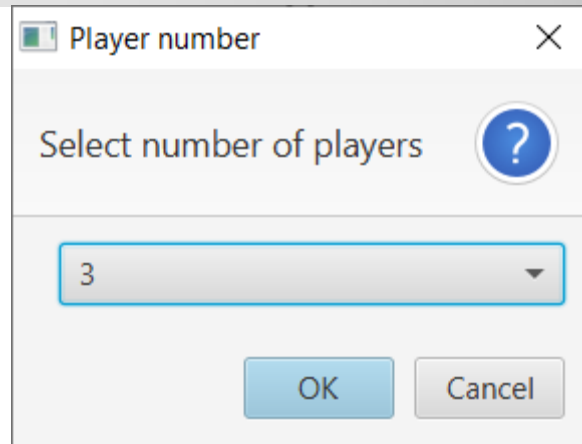
Hvis HashMaps eller HashSets forholder sig "lidt mærkeligt" (fx. hvis de "glemmer" elementer man havde tilføjet før) i jeres programmer, så er en forkert implementering af `equals()` og `hashCode()` en meget sandsynlig årsag!

- Indtil videre har vi kun anvendt nogle indbyggede datatyper fra Java!
- Hvordan man selv programmere nogle datatyper (eller udvidelser deraf) selv vender vi tilbage til

- Til at programmere GUIen bruger vi JavaFX
- Her er der nogle informationer om JavaFX programming:
- JavaFX Applikationen:
 - https://docs.oracle.com/javafx/2/get_started/hello_world.htm
 - https://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm
- JavaFX LayoutPanels:
 - https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm#
- JavaFX UI Controls:
 - https://docs.oracle.com/javafx/2/ui_controls/overview.htm#
 - https://docs.oracle.com/javafx/2/ui_controls/text-field.htm#
 - https://docs.oracle.com/javafx/2/ui_controls/button.htm#
- JavaFX Dialogs:
 - <https://examples.javacodegeeks.com/desktop-java/javafx/dialog-example/>

Fortsættes fra forelæsning 3

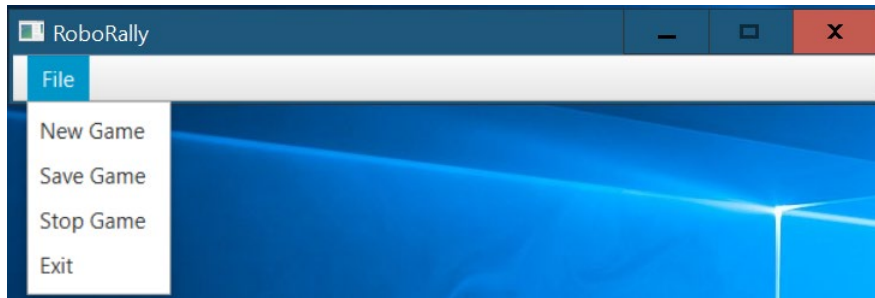
Eksempel: ChoiceDialog



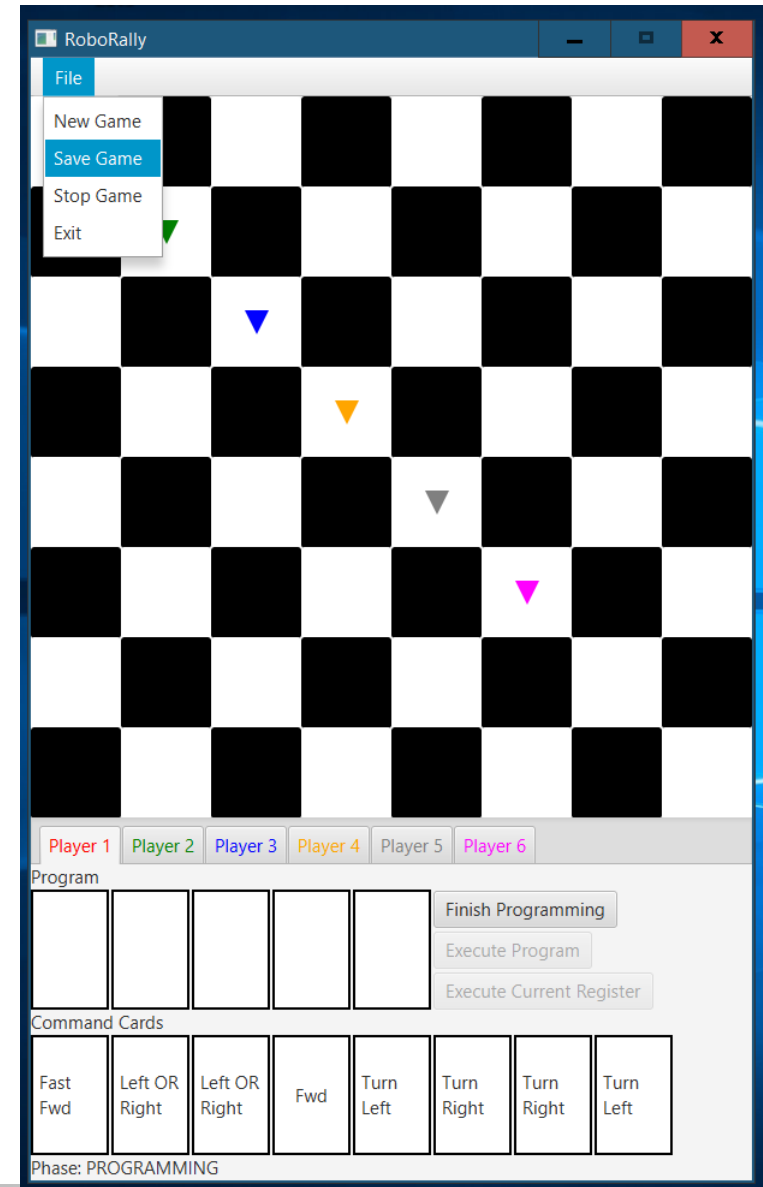
- Bemærk at dette er en modal dialog, dvs. brugeren kan først interagere med resten af GUI'en (eller forældre-dialog), når denne her dialog er afsluttet.

I **GameController** skal I helst ikke bruge (modale) dialoger! Se opgave V3, hvordan man kan undgå dette! Men under selve starten af spillet er det OK!

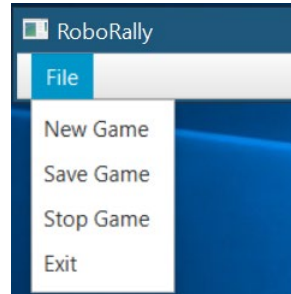
Menuer og menubjælker



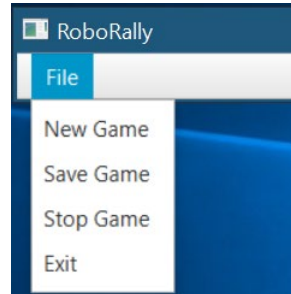
Kode følger på de næste slides.
Konceptuelt diskussion på tavlen
(noter følger på websiderne)



```
public class RoboRallyMenuBar extends MenuBar {  
  
    private ApplicationController appController;  
  
    public RoboRallyMenuBar(ApplicationController appController) {  
        this.appController = appController;  
  
        Menu controlMenu = new Menu("File");  
        this.getMenus().add(controlMenu);  
  
        MenuItem newGame = new MenuItem("New Game");  
        newGame.setOnAction( e -> {this.appController.newGame();} );  
        controlMenu.getItems().add(newGame);  
  
        MenuItem saveGame = new MenuItem("Save Game");  
        saveGame.setOnAction( e -> {this.appController.saveGame();} );  
        controlMenu.getItems().add(saveGame);  
  
        MenuItem stopGame = new MenuItem("Stop Game");  
        stopGame.setOnAction( e -> {this.appController.stopGame();} );  
        controlMenu.getItems().add(stopGame);  
  
        MenuItem exitApp = new MenuItem("Exit");  
        exitApp.setOnAction( e -> {this.appController.exit();} );  
        controlMenu.getItems().add(exitApp);  
  
    }  
}
```



```
public class RoboRallyMenuBar extends MenuBar {  
  
    private ApplicationController appController;  
  
    public RoboRallyMenuBar(ApplicationController appController)  
    {  
        this.appController = appController;  
  
        Menu controlMenu = new Menu("File");  
        this.getMenus().add(controlMenu);  
    }  
}
```



```
MenuItem newGame = new MenuItem("New Game");
newGame.setOnAction(
    e -> {this.appController.newGame();} );
controlMenu.getItems().add(newGame);

MenuItem saveGame = new MenuItem("Save Game");
saveGame.setOnAction(
    e -> {this.appController.saveGame();} );
controlMenu.getItems().add(saveGame);

// ...

}
```

Eksempel: Menubjælke

```
public class RoboRallyMenuBar extends MenuBar {
```

Menubjælke (MenuBar)
importeres fra JavaFX

```
    private ApplicationController appController;
```

```
    public RoboRallyMenuBar(ApplicationController appController) {  
        this.appController = appController;
```

Fil-menu

En controller for
hele appen.

```
        Menu controlMenu = new Menu("File");  
        this.getMenus().add(controlMenu);
```

Fil-menu tilføjes til bjælken

```
        MenuItem newGame = new MenuItem("New Game");  
        newGame.setOnAction( e -> {this.appController.newGame();} );  
        controlMenu.getItems().add(newGame);
```

Menupunkt "New Game"

```
        MenuItem saveGame = new MenuItem("Save Game");  
        saveGame.setOnAction( e -> {this.appController.saveGame();} );  
        controlMenu.getItems().add(saveGame);
```

```
        MenuItem stopGame = new MenuItem("Stop Game");  
        stopGame.setOnAction( e -> {this.appController.stopGame();} );  
        controlMenu.getItems().add(stopGame);
```

```
        MenuItem exitApp = new MenuItem("Exit");  
        exitApp.setOnAction( e -> {this.appController.exitApp();} );  
        controlMenu.getItems().add(exitApp);
```

Det her er såkaldte λ -expressions
(lambda), som hjælper at gøre
koden mere læsbart (korter). Ikke
i fokus her!

```

public class ApplicationController {

    final private List<Integer> PLAYER_NUMBER_OPTIONS = Arrays.asList(2, 3, 4, 5, 6);
    final private List<String> PLAYER_COLORS = Arrays.asList("red", "green", "blue", "orange", "grey", "magenta");

    private RoboRally roboRally;

    private GameController gameController;

    public ApplicationController(RoboRally roboRally) {
        this.roboRally = roboRally;
    }

    public void newGame() {
        ChoiceDialog<Integer> dialog = new ChoiceDialog<>(PLAYER_NUMBER_OPTIONS.get(0), PLAYER_NUMBER_OPTIONS);
        dialog.setTitle("Player number");
        dialog.setHeaderText("Select number of players");
        Optional<Integer> result = dialog.showAndWait();

        if (result.isPresent()) {
            if (gameController != null) {
                // give user the option to save the game
                if (!stopGame()) {
                    return;
                }
            }
            Board board = new Board(8, 8);
            gameController = new GameController(board);
            int no = result.get();
            for (int i = 0; i < no; i++) {
                Player player = new Player(board, PLAYER_COLORS.get(i), "Player " + (i + 1));
                board.addPlayer(player);
                player.setSpace(board.getSpace(i, i));
            }
            board.setCurrentPlayer(board.getPlayer(0));
            roboRally.createBoardView(gameController);
            gameController.initializeProgrammingPhase();
        }

        public void saveGame() {
    }
}

```

Nu kommer AppControllere!

```
public class ApplicationController {  
  
    // ...  
  
    private RoboRally roboRally;  
  
    private GameController gameController;  
  
    public ApplicationController(RoboRally roboRally) {  
        this.roboRally = roboRally;  
    }  
}
```

```
public void newGame() {  
    ChoiceDialog<Integer> dialog =  
        new ChoiceDialog<>(  
            PLAYER_NUMBER_OPTIONS.get(0),  
            PLAYER_NUMBER_OPTIONS);  
    dialog.setTitle("Player number");  
    dialog.setHeaderText("Select number");  
    Optional<Integer> result = dialog.showAndWait();  
  
    if (result.isPresent()) {  
        // ...  
        Board board = new Board(8, 8);  
        GameController = new GameController(board);  
    }  
}
```

```
int no = result.get();
for (int i = 0; i < no; i++) {
    Player player = new Player(
        board,
        PLAYER_COLORS.get(i),
        "Player " + (i + 1));
    board.addPlayer(player);
    player.setSpace(board.getSpace(i, i));
}
board.setCurrentPlayer(board.getPlayer(0));
roboRally.createBoardView(gameController);
gameController.initializeProgrammingPhase();
}
...
}
```

```
public class ApplicationController {
```

```
// ...
```

```
private RoboRally roboRally;
```

```
private GameController gameController;
```

```
public ApplicationController(RoboRally roboRally) {  
    this.roboRally = roboRally;  
}
```

AppControlleren
kender appen og
er kun ansvarlige
til denne ene
app.

```
public void newGame() {  
    ChoiceDialog<Integer> dialog =  
        new ChoiceDialog<>(  
            PLAYER_NUMBER_OPTIONS.get(0),  
            PLAYER_NUMBER_OPTIONS);  
    dialog.setTitle("Player number");  
    dialog.setHeaderText("Select number");  
    Optional<Integer> result = dialog.showAndWait();  
  
    if (result.isPresent()) {  
        // ...  
        Board board = new Board(8, 8);  
        GameController = new GameController(board);  
    }  
}
```

Brugerdialog

Kreer et nyt spil
(spilleplade) og en
GameController til den

```
int no = result.get();  
for (int i = 0; i < no; i++) {  
    Player player = new Player(  
        board,  
        PLAYER_COLORS.get(i),  
        "Player " + (i + 1));  
    board.addPlayer(player);  
    player.setSpace(board.getSpace(i, i));  
    board.setCurrentPlayer(board.getPlayer(0));  
    roboRally.createBoardView(gameController);  
    gameController.initializeProgrammingPhase();  
}
```

Kreer spillerne
og placer dem
på pladen!

Definer den
første spiller som
den aktuelle!

Register gameControlleren
med appen (og indirekte
kreer BoardView).

Sæt spillet på
programmeringsfasen!

Applikationen

```
public class RoboRally extends Application {
```

```
    private Stage stage;
```

```
    private BorderPane boardRoot;
```

```
    // ...
```

```
    public void start(Stage primaryStage) throws Exception {  
        stage = primaryStage;
```

```
        ApplicationController appController = new ApplicationController(this);
```

```
        // create the primary scene with the menu bar and a pane  
        // the board view (which initially is empty); it will be filled  
        // when the user creates a new game or loads a game
```

```
        RoboRallyMenuBar menuBar = new RoboRallyMenuBar(appController);
```

```
        boardRoot = new BorderPane();
```

```
        VBox vbox = new VBox(menuBar, boardRoot);
```

```
        vbox.setMinWidth(MIN_APP_WIDTH);
```

```
        Scene primaryScene = new Scene(vbox);
```

```
        stage.setScene(primaryScene);
```

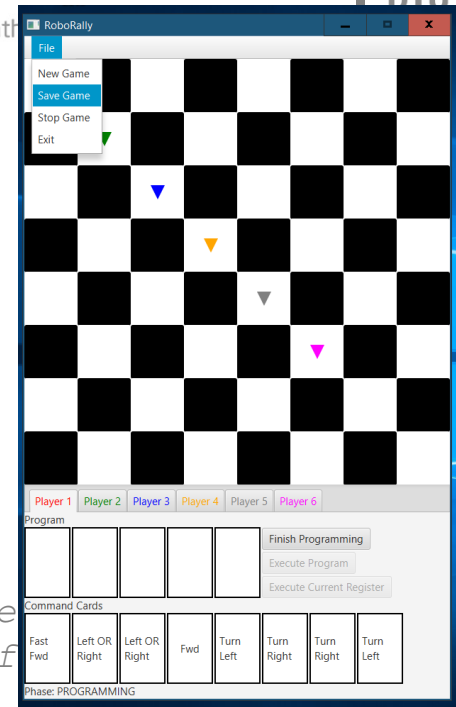
```
        stage.setTitle("RoboRally");
```

```
        stage.setResizable(false);
```

```
        stage.sizeToScene();
```

```
        stage.show();
```

```
    }
```



Applikationen kender top-level viewet (stage) og pane, hvor spillepladen skal placeres.

```
public void start(Stage primaryStage) throws Exception {  
    stage = primaryStage;
```

```
    ApplicationController appController = new ApplicationController(this);
```

```
    RoboRallyMenuBar menuBar = new  
        RoboRallyMenuBar(appController);
```

```
    boardRoot = new BorderPane();
```

```
    VBox vbox = new VBox(menuBar, boardRoot);
```

```
    Scene primaryScene = new Scene(vbox);
```

```
    stage.setScene(primaryScene);
```

```
    stage.setTitle("RoboRally");
```

```
    stage.setResizable(false);
```

```
    stage.sizeToScene();
```

```
    stage.show();
```

```
}
```

Applikationens
AppController bliver
kreeret og tilføjet til
menubælken som
også bliver kreeret.

```
public void start(Stage primaryStage) throws Exception {  
    stage = primaryStage;
```

```
    ApplicationController appController = new ApplicationController(this);
```

```
    RoboRallyMenuBar menuBar = new  
        RoboRallyMenuBar(appController);  
    boardRoot = new BorderPane();  
    VBox vbox = new VBox(menuBar, boardRoot);  
    Scene primaryScene = new Scene(vbox);
```

```
    stage.setScene(primaryScene);  
    stage.setTitle("RoboRally");  
    stage.setResizable(false);  
    stage.sizeToScene();  
    stage.show();
```

```
}
```

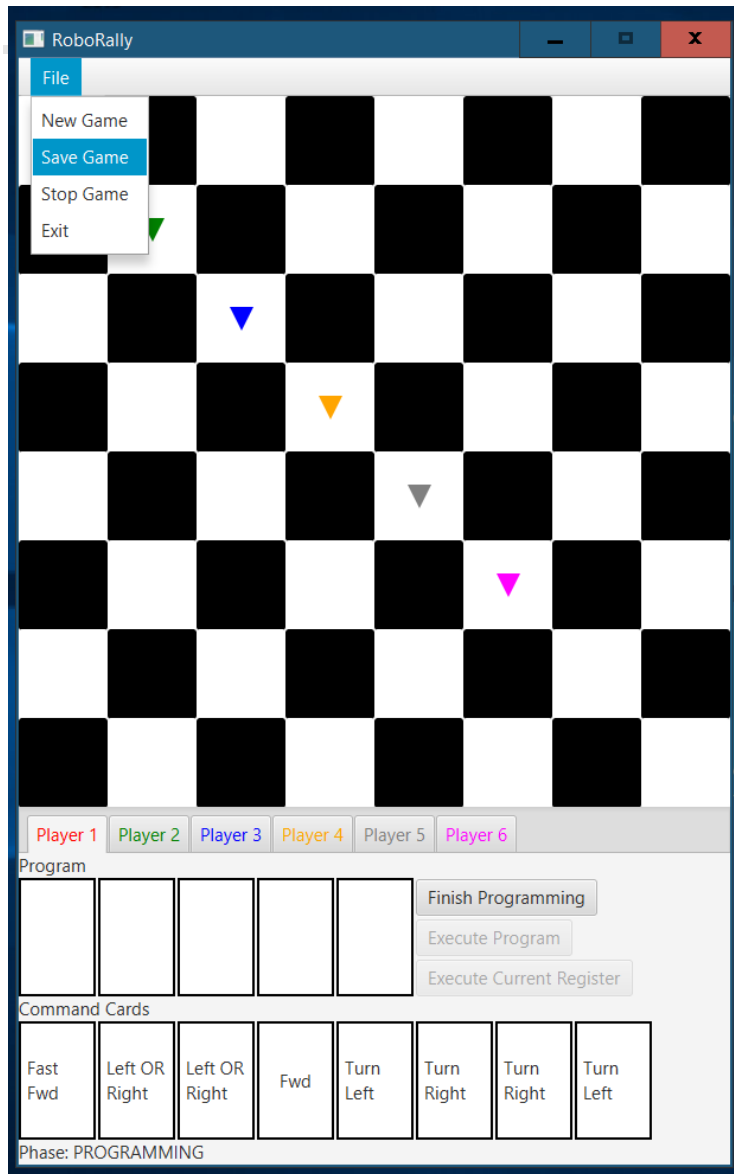
Pane hvor
spillepladen skal
placeres

Menubælket og plads
til spillepladen bliver
kombineret (i en
Vbox) og sat på en
scene!

Scenen kommer på
stage og lidt
konfigureret.

```
public void createBoardView(GameController gameController) {  
    // if present, remove old BoardView  
    boardRoot.getChildren().clear();  
  
    if (gameController != null) {  
        // create and add view for new board  
        BoardView boardView = new BoardView(gameController);  
        boardRoot.setCenter(boardView);  
    }  
  
    stage.sizeToScene();  
}
```

Sæt det nye
BoardView på den
forberedte plads.



Nogle gange kommer det frem midt i en metode, at operationen ikke kan gennemføres. Så er man nødt til at reagere og også den som har kaldt metoden er måske nødt til at reagere på det.

Såkalde undtagelser (exceptions) er en mulighed at programmere sådan noget.

- Hvis en robot skal flyttes, skubber den en robot som står foran den også i samme retning; og hvis der står en foran den igen, skal den også flyttes, osv.
- Det giver en kaskade af flytningerne
- Hvis så den sidste i kaskaden ikke kan flyttes (fx da den rammer en væg), så ryger hele kaskaden, og det oprindelige træk er ikke muligt

→ Se diskussion på tavlen!

```
class ImpossibleMoveException extends Exception {

    private Player player;
    private Space space;
    private Heading heading;

    public ImpossibleMoveException(Player player,
                                   Space space,
                                   Heading heading) {
        super("Move impossible");
        this.player = player;
        this.space = space;
        this.heading = heading;
    }

    // ...

}
```

```
public void moveForward(@NotNull Player player) {  
    if (player.board == board) {  
        Space space = player.getSpace();  
        Heading heading = player.getHeading();  
  
        Space target = board.getNeighbour(space, heading);  
        if (target != null) {  
            try {  
                moveToSpace(player, target, heading);  
            } catch (ImpossibleMoveException e) {  
                // we don't do anything here for now;  
                // we just catch the exception so that  
                // we do not pass it on to the caller  
                // (which would be very bad style).  
            }  
        }  
    }  
}
```

```
private void moveToSpace(  
    @NotNull Player player,  
    @NotNull Space space,  
    @NotNull Heading heading) throws ImpossibleMoveException {  
  
    Player other = space.getPlayer();  
    if (other != null) {  
        Space target = board.getNeighbour(space, heading);  
        if (target != null) {  
            // XXX Note that there might be additional problems  
            // with infinite recursion here!  
            moveToSpace(other, target, heading);  
        } else {  
            throw new ImpossibleMoveException(player, space, heading);  
        }  
    }  
    player.setSpace(space);  
}
```

Fortsættes

Opgave A3 (til 22/23. 3.)

Det er først om 3 uger!
Men husk aflevering A2
til næste uge!

Første **prototype** med **fokus på spillelogik***:

- Der skal være **flere funktioner og begreber** af RoboRally-spillet med **end i opgaverne V1-V3**! Fx:
 - Skubbe andre robotter foran sig (se slides 50-52)
 - Vægge eller forhindringer på spillepladen (implementering af metoden `getNeighbour()` se slide 51)
 - Aktionsfelter på spillepladen (fx. transportfelt)
 - Checkpoints
 - Flere slags programmeringskort
 - ...

*) AppControlleren og at gemme og lade spil (og spillepladen) kommer senere (V4a/4b)!

Opgave A3 (til 22/23. 3.)

Det er først om 3 uger!
Men husk aflevering A2
til næste uge!

Første **prototype** med **fokus på spillelogik***:

- Der skal være **flere funktioner og begreber** af RoboRally-spillet med **end i opgaverne V1-V3!** Fx:
 - ...
- Overvej hvilke ekstrafeatures giver mest mening (baseret på aflevering A2 og det hvad I har allerede med V1-3).
- Læg plan til jeres features opdel opgaverne i blandt jer!
- Aflever som komprimeret (zip-fil) af jeres IntelliJ-projekt via DTU Learn!