

# Mer om metoder och abstraktioner

Föreläsning 5

TDA540 - Objektorienterad Programmering



**CHALMERS**

# Sammanfattning

- Abstraktion, top-down design
- Gränsnitt av en metod
- Parameteröverföring
- För- och eftervilkor
- (kompilera projekt)

# Abstraktion = subtraktion

Det är svårt att tänka på flera nivåer samtidigt.

Abstraktion är ett verktyg som underlättar.

Abstraktion = subtraktion

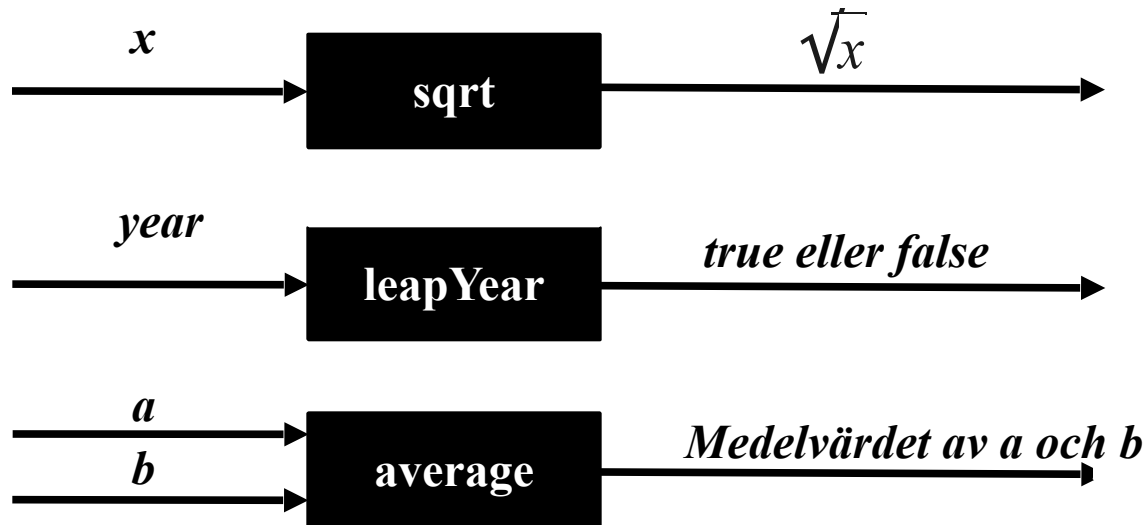
En abstraktion innebär att man bortser från vissa omständigheter och detaljer i det vi betraktar, för att bättre kunna uppmärksamma andra för tillfället mer väsentliga aspekter

Abstrahera = att införa ett begrepp

Ett begrepp fångar vad som är väsentligt och gemensamt

Abstraktioner hjälper oss att fokusera på de för tillfället viktiga egenskaperna.

# "Black box" - tänkande



Är ett sätt att uttrycka vissa steg i ett program på en *högre abstraktionsnivå*.

På en hög abstraktionsnivå är det vad som görs som är det intressanta, inte hur det görs.

Ger direkt stöd för *stegvis förfining*.

Varje abstraktion (black box) skall endast *göra en sak* och göra denna *bra*.

# "Black box" - tänkande

En black box har en insida och en utsida:

Gränssnittet (interfacet) är det som binder insidan och utsidan.

Gränssnittet har såväl syntax som semantik (specifikationen).

*"The interface of a black box should be fairly straightforward, well defined, and easy to understand."*

*"To use a black box, you shouldn't need to know anything about its implementation ; all you need to know is its interface."*

*"The implementor of a black box should not need to know anything about the larger systems in which the box will be used."*

En black box är ett sätt att uttrycka vissa steg i ett program på en högre abstraktionsnivå.

# Problemställning

I tärningsspelet Mexico deltar minst två spelare och spelet är ett elimineringsspel. När spelet börjar har alla spelare lika mycket pengar att spela med (= startavgift, dvs de pengar som spelaren riskerar att förlora).

En omgång går till på så sätt att varje spelare kastar två 6-sidiga tärningar. Spelaren som får lägst *poängvärde* förlorar omgången och måste lägga en i förväg fastställd insatsen i potten. Detta upprepas tills endast en av spelarna har pengar kvar. Denne är vinnaren och får hela potten.

För att beräkna *poängvärdet*, beräknas först den sammanlagda *tärningssumman*. Detta görs genom att först multiplicera värdet på tärningen med högst poäng med 10 och sedan addera värdet på tärningen med lägst poäng. Har tärningarna poängen 3 och 4 erhålls tärningssumman 43, har tärningarna poängen 5 och 5 erhålls tärningssumman 55, osv.

Tärningssumman ligger till grund för att beräkna poängvärdet. Tärningssummornas poängvärde från högsta till lägsta är enligt:

21, 66, 55, 44, 33, 22, 11, 65, ..., 61, 54, ..., 51, 43, 42, 41, 32, 31

Högsta poängvärdet är alltså tärningsvärdet 21 (som kallas "Mexico"), följt av "paren" (i numerisk ordning), följt av övriga tärningsvärden (i numerisk ordning).

# Uppgift

Du är med och utvecklar en datorversion av spelet.

Det behövs en metod som tar två tärningssummor och returnerar den tärningssumma som har lägsta poängvärdet. Din uppgift är att utveckla en sådan metod!

*Vad är det första som du måste göra?*

# Uppgift

Du är med och utvecklar en datorversion av spelet.

Det behövs en metod som tar två tärningssummor och returnerar den tärningssumma som har lägsta poängvärdet. Din uppgift är att utveckla en sådan metod!

*Vad är det första som du måste göra?*

*Bestämma (specificera) gränssnittet för metoden!!!*



# Bestämma gränssnitt

*Har du tillräckligt med information för att specificera gränssnittet?*

# Bestämma gränssnitt

*Har du tillräckligt med information för att specificera gränssnittet?*

*Nej, du måste komma överens med den som beställt (skall använda) metoden om metodens namn, om metoden skall vara en klassmetod eller en instansmetod, samt metodens synlighet.*

Låt oss säga att detta resulterar i att metoden skall ha namnet `lowestScoreValue`, samt vara en privat klassmetod.

# Bestämma gränssnitt

*Har du tillräckligt med information för att specificera gränssnittet?*

*Nej, du måste komma överens med den som beställt (skall använda) metoden om metodens namn, om metoden skall vara en klassmetod eller en instansmetod, samt metodens synlighet.*

Låt oss säga att detta resulterar i att metoden skall ha namnet `lowestScoreValue`, samt vara en privat klassmetod.

```
private static int lowestScoreValue(int score1, int score2)
```

Vidare måste vi dokumentera vad metoden gör och eventuella för- och eftervillkor!

# Bestämna gränssnitt

*Har du tillräckligt med information för att specificera gränssnittet?*

*Nej, du måste komma överens med den som beställt (skall använda) metoden om metodens namn, om metoden skall vara en klassmetod eller en instansmetod, samt metodens synlighet.*

Låt oss säga att detta resulterar i att metoden skall ha namnet `lowestScoreValue`, samt vara en privat klassmetod.

```
private static int lowestScoreValue(int score1, int score2)
```

Vidare måste vi dokumentera vad metoden gör och eventuella för- och eftervillkor!

```
//before: score1 and score2 are valid Mexico scores  
//returns the score with lowest score value  
private static int lowestScoreValue(int score1, int score2)
```

# Det färdiga gränssnittet

```
//before: score1 and score2 are valid Mexico scores  
//returns the score with lowest score value  
private static int lowestScoreValue(int score1, int score2)
```

Nu är alla förutsättningar givna för att:

- du skall kunna påbörja arbetet med att utveckla metoden.
- du eller andra utvecklare skall kunna skriva kod för att testa metoden
- andra utvecklare kan nyttja metoden som en abstraktion i sitt utvecklingsarbete.

# Testdriven programutveckling

Testdriven programutveckling innebär att man skriver testerna för koden innan koden skrivs. När koden som skrivits går igenom testerna gör koden det den skall göra. Koden kan dock behöva omstruktureras (refactoring) för att få koden snyggare och bättre (t.ex. mer lättläst).

- Ett test beskriver vad som skall göras medan implementationen beskriver hur det ska göras. Testerna blir därmed en del av specifikationen.
- Eftersom testet skrivs innan implementationen blir designen av gränssnittet gjort utifrån en brukarens perspektiv.
- Testerna blir verkligen skrivna, vilket inte alltid är fallet om man skriver dem efteråt.

Skriv en testmetod

```
private static void testLowestScoreValue()
```

För att testa metoden `lowestScoreValue`.

# Testdriven programutveckling

```
private static void testLowestScoreValue() {
    System.out.println(lowestScoreValue(21, 32) + " : should be 32");
    System.out.println(lowestScoreValue(66, 21) + " : should be 66");
    System.out.println(lowestScoreValue(43, 33) + " : should be 43");
    System.out.println(lowestScoreValue(33, 55) + " : should be 33");
    System.out.println(lowestScoreValue(65, 22) + " : should be 65");
    System.out.println(lowestScoreValue(53, 54) + " : should be 53");
} //testLowestScoreValue
```

```
private static boolean testLowestScoreValues() {
    return testLowestScoreValue(21, 32, 32) &&
        testLowestScoreValue(66, 21, 66) &&
        testLowestScoreValue(43, 33, 43) &&
        testLowestScoreValue(33, 55, 33) &&
        testLowestScoreValue(65, 22, 65) &&
        testLowestScoreValue(53, 54, 53);
}
```

```
private static boolean testLowestScoreValue(int x, int y, int exp) {
    int result = lowestScoreValue(x, y);
    System.out.println(result + " : should be " + exp);
    return result == exp;
}
```

# Implementation

```
//before: score1 and score2 are valid Mexico scores  
//returns the score with lowest score value  
private static int lowestScoreValue(int score1, int score2)
```

Börja med att ställa dej frågan:

*Har jag löst något liknande problem tidigare?*



# Implementation

```
//before: score1 and score2 are valid Mexico scores  
//returns the score with lowest score value  
private static int lowestScoreValue(int score1, int score2)
```

Börja med att ställa dej frågan:

*Har jag löst något liknande problem tidigare?*

```
public static int maxValue(int a, int  
b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}  
>//maxValue;
```



```
public static int minValue(int a, int b)  
{  
    if (a < b)  
        return a;  
    else  
        return b;  
}  
>//minValue;
```

# Implementation

Metoden `lowestScoreValue` skall bestämma det minsta av två numeriska värden. Dessa numeriska värden beräknas från `score1` respektive `score2`.

*Specificera en abstraktion som beräknar dessa numeriska värden:*

# Implementation

Metoden `lowestScoreValue` skall bestämma det minsta av två numeriska värden. Dessa numeriska värden beräknas från `score1` respektive `score2`.

*Specificera en abstraktion som beräknar dessa numeriska värden:*

```
//before: score is a valid Mexico score  
//returns the comparison value of score  
private static int getScoreValue(int score)
```

# Implementation

Metoden `lowestScoreValue` skall bestämma det minsta av två numeriska värden. Dessa numeriska värden beräknas från `score1` respektive `score2`.

*Specificera en abstraktion som beräknar dessa numeriska värden:*

```
//before: score is a valid Mexico score  
//returns the comparison value of score  
private static int getScoreValue(int score)
```

```
//before: score1 and score2 are valid Mexico scores  
//returns the score with lowest score value  
private static int lowestScoreValue(int score1, int score2) {  
    int value1 = getScoreValue(score1);  
    int value2 = getScoreValue(score2);  
    if ( value1 < value2)  
        return score1;  
    else  
        return score2;  
}  
// lowestScoreValue
```

# Implementation

```
//before: score is a valid Mexico score  
//returns the comparison value of score  
private static int getScoreValue(int score)
```

Problemet vi har är att för varje tärningssumma

21, 66, 55, 44, 33, 22, 11, 65, ..., 61, 54, ..., 51, 43, 42, 41, 32, 31

beräkna ett numeriskt värde (heltal) p.s.s. att dessa värden överensstämmer med ovan givna ordning på tärningssummorna.

*Någon idéer om hur detta skall gå till?*

# Implementation

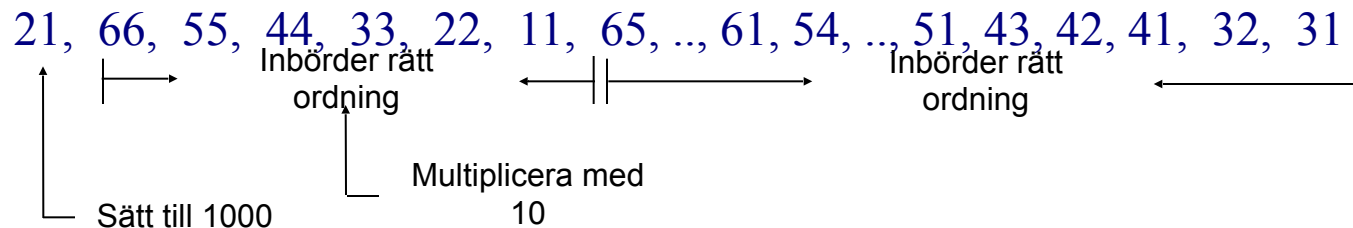
```
//before: score is a valid Mexico score  
//returns the comparison value of score  
private static int getScoreValue(int score)
```

Problemet vi har är att för varje tärningssumma

21, 66, 55, 44, 33, 22, 11, 65, ..., 61, 54, ..., 51, 43, 42, 41, 32, 31

beräkna ett numeriskt värde (heltal) p.s.s. att dessa värden överensstämmer med ovan givna ordning på tärningssummorna.

*Någon idéer om hur detta skall gå till?*

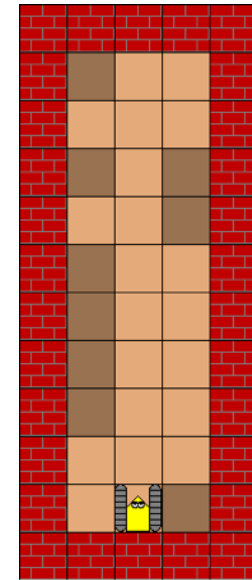
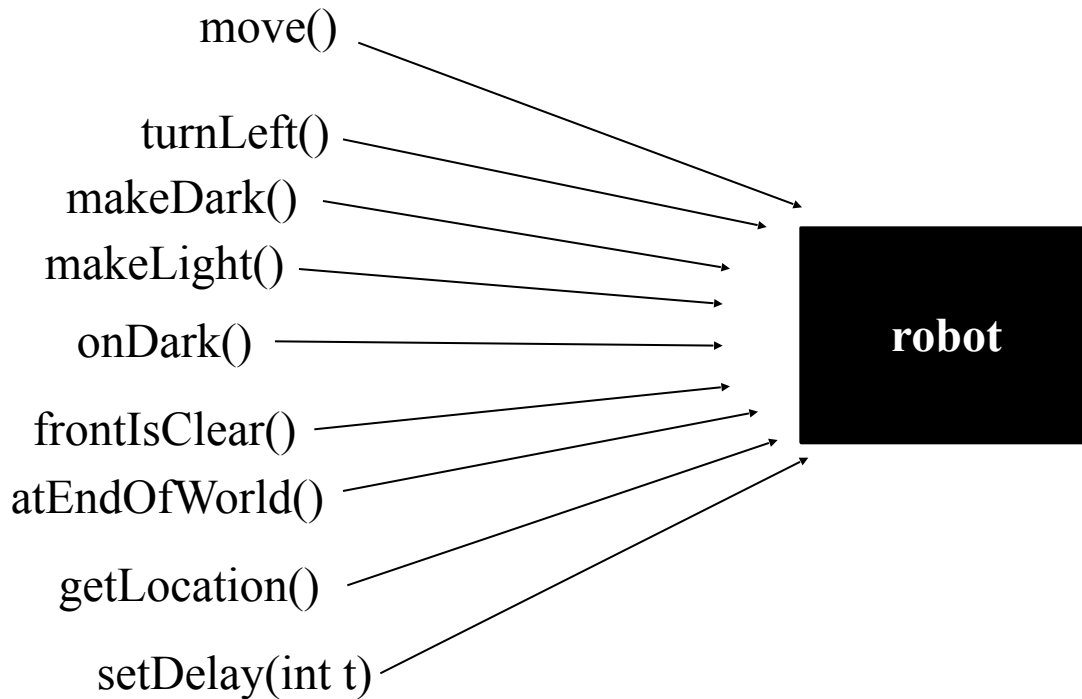


1000, 660, 550, 440, 330, 220, 110, 65, ..., 61, 54, ..., 51, 43, 42, 41, 32, 31

# Implementation

```
//before: score is a valid Mexico score
//returns the comparison value of score
public static int getScoreValue(int score) {
    int comparisonValue;
    if (score == 21)
        comparisonValue = 1000;
    else if (score / 10 == score % 10)
        comparisonValue = score * 10;
    else
        comparisonValue = score;
    return comparisonValue;
} // getScoreValue
```

# Behov av "större" svarta boxar än metoder



Ett objekt har beteenden:

- de operationer som kan göras på objektet
- implementeras mha metoder

Ett objekt har tillstånd ("värde"):

- robotens riktning
- robotens fysiska plats
- robotens hastighet



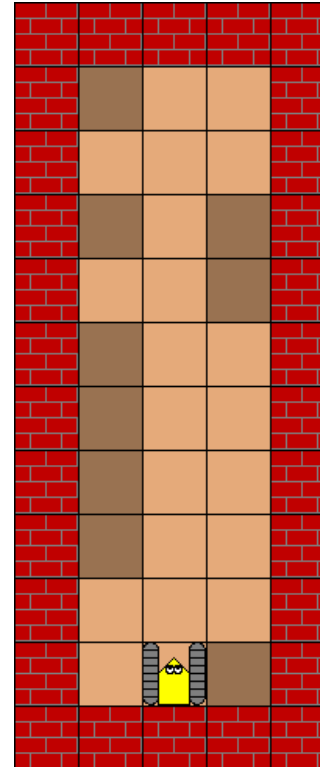
# "Black box" - tänkande

<b>void</b> move()	förflyttar sig ett steg framåt. Om roboten hamnar utanför världen eller i en mur fås ett exekveringsfel.
<b>void</b> turnLeft()	vrider sig 90° åt vänster.
<b>void</b> makeDark()	färgar rutan den står på till mörk. Om rutan redan är mörk fås ett exekveringsfel.
<b>void</b> makeLight()	färgar rutan den står på till ljus. Om rutan redan är ljus fås ett exekveringsfel.
<b>boolean</b> onDark()	returnerar <b>true</b> om roboten står på en mörk ruta, annars returneras <b>false</b> .
<b>boolean</b> frontIsClear()	returnerar <b>true</b> om det är möjligt för roboten att göra move() utan att ett exekveringsfel erhålls, annars returnera <b>false</b> .
<b>boolean</b> atEndOfWorld()	returnerar <b>true</b> om det är omöjligt för roboten att göra move() utan att hamna utanför världen, annars returneras <b>false</b> .
Location getLocation()	returnerar robotens position i världen som ett objekt av typen Location.
<b>void</b> setDelay( <b>int</b> t)	sätter hastigheten med vilken roboten rör sig.

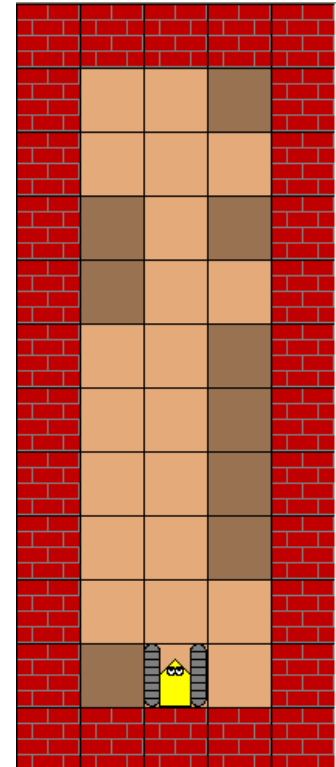
# Problem

I världen som skapas av klassen `Swapper` befinner sig roboten i ett scenario enligt den vänstra av de två figurerna bredvid. Roboten är placerad i början av en korridor och dess uppgift (som implementeras av metoden `swapAll`) är att byta plats på färgerna på cellerna som finns på ömse sidor om korridoren (se högra bilden bredvid). Efter slutförd uppgift skall roboten återvända till sin ursprungliga startposition.

Du skall bryta ner uppgiften i delproblem och utveckla kraftfullare abstraktioner (metoder) än de operationer som roboten tillhandahåller.



Före



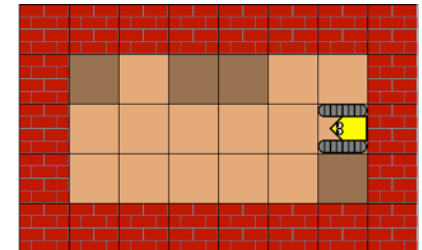
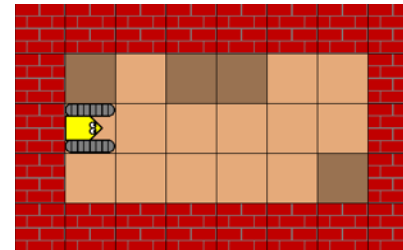
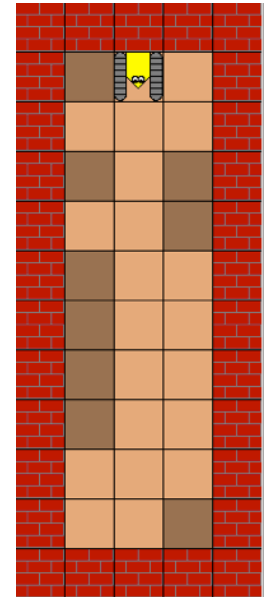
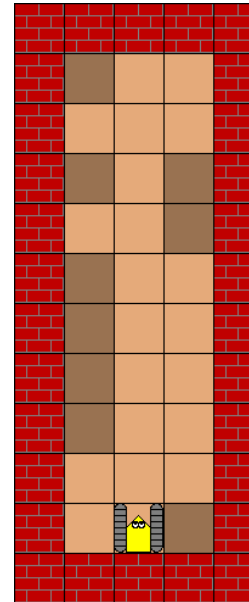
Efter

# Problem

Roboten skall kunna utföra sin arbetsuppgift i en godtycklig värld med samma *principiella uppbyggnad* som världen i figurerna ovan.

Var noggrann med att beskriva vad varje metod gör och vilka för- respektive eftervillkor som gäller (t.ex. vilken riktning roboten måste ha innan metoden anropas, och vilken riktning roboten får efter anropet). Välj också med omsorg lämpliga och beskrivande namn på metoderna.

När man löser ett problem är det sällan man finner den optimala lösningen direkt. När man väl har en lösning är det därför viktigt att reflektera över om det finns andra bättre lösningar. I det problem som ni just har blivit förelagda, kan antalet operationer som roboten behöver utföra vara ett mått på hur bra lösningen är. Ju färre operationer desto bättre.

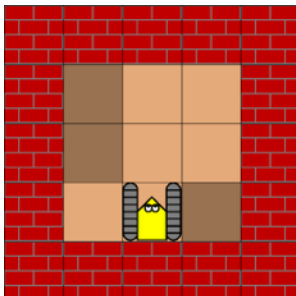
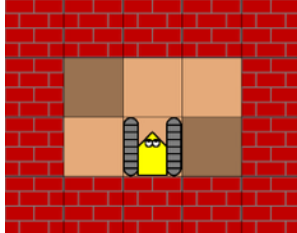
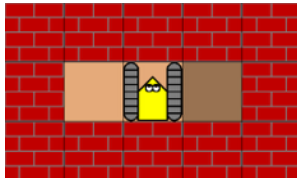


# Lös problemet

Handfallen? Råd vill? Kommer inte igång?

Förenkla problemet! Titta på en mindre instans av problemet!

Fokusera på vad som skall göras, inte på hur det skall göras! Finn abstraktioner!

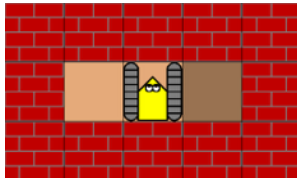


# Lös problemet

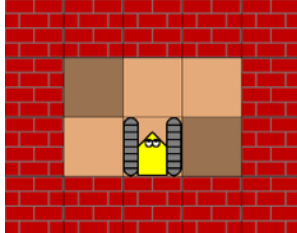
Handfallen? Rådsvill? Kommer inte igång?

Förenkla problemet! Titta på en mindre instans av problemet!

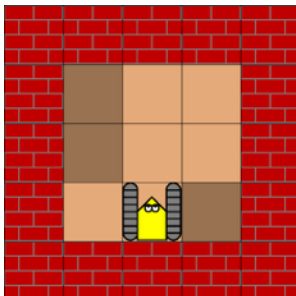
Fokusera på vad som skall göras, inte på hur det skall göras! Finn abstraktioner!



```
swapTwoCells();
```



```
swapTwoCells();  
robot.move();  
swapTwoCells();
```



```
swapTwoCells();  
robot.move();  
swapTwoCells();  
robot.move();  
swapTwoCells();
```

# Metoden swapAll

```
//swapping colours of two across cells  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void swapTwoCells()
```

```
//returns true if the robot is at the end of the corridor, otherwise false  
public boolean atEndOfCorridor()
```

```
//before: robot is at the end of the corridor, facing the wall  
//after: robot is at the beginning of the corridor, facing the corridor  
private void returnToStartPosition()
```

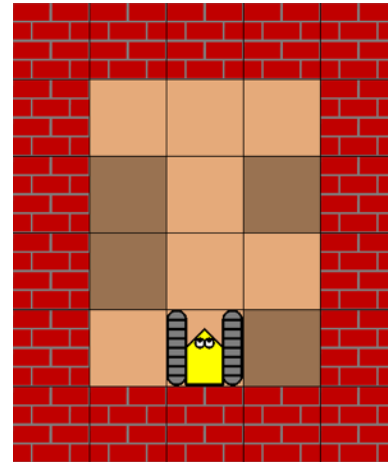
```
//swapping colors on all across cells in the corridor  
//before: the robot is located at the beginning of the corridor, facing the corridor  
//after: the robot has the same location and facing the same direction  
public void swapAll() {  
    boolean finished = false;  
    while (!finished) {  
        swapTwoCells();  
        if (!atEndOfCorridor())  
            robot.move();  
        else  
            finished = true;  
    }  
    returnToStartPosition();  
}  
} //swapAll
```

## Metoden atEndOfCorridor

```
//returns true if the robot is at the end of the corridor, otherwise false  
public boolean atEndOfCorridor() {  
    return !robot.frontIsClear();  
}  
//atEndOfCorridor
```

# Metoden swapTwoCells

*När behöver färgerna på cellerna bytas?*





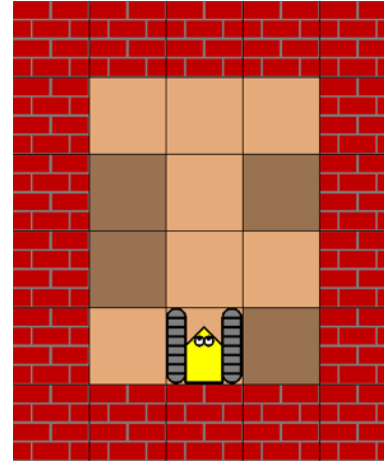
# Metoden `swapTwoCells`

*När behöver färgerna på cellerna bytas?*

*När cellerna har olika färg!*

*Kan uttryckas som*

*`leftCellIsDark() == !rightCellIsDark()`*



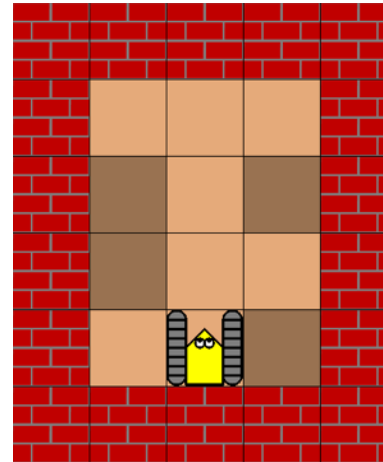
# Metoden `swapTwoCells`

*När behöver färgerna på cellerna bytas?*

*När cellerna har olika färg!*

*Kan uttryckas som*

*`leftCellIsDark() == !rightCellIsDark()`*



```
//return: true if the cell on left side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean leftCellIsDark()
```

```
//return: true if the cell on right side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean rightCellIsDark()
```

## Metoden swapTwoCells

```
//return: true if the cell on left side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean leftCellIsDark()
```

```
//return: true if the cell on right side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean rightCellIsDark()
```

```
//swapping colors of two across cells  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void swapTwoCells() {  
    if (leftCellIsDark() != rightCellIsDark())  
        changeColors();  
}  
} //swapTwoCells
```

```
//change color of the cell on left side and of the cell on right side  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void changeColors()
```

## Metoden leftCellsDark

```
//before: none  
//after: robot is facing the opposite direction  
private void turnAround()
```

```
//returns true if the cell on left side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean leftCellIsDark() {  
    robot.turnLeft();  
    robot.move();  
    boolean isDark = robot.onDark();  
    turnAround();  
    robot.move();  
    robot.turnLeft();  
    return isDark;  
} // leftCellIsDark
```

# Metoden turnAround

```
//before: none  
//after: robot is facing the opposite direction  
private void turnAround() {  
    robot.turnLeft();  
    robot.turnLeft();  
}  
} //turnAround
```

# Metoden rightCellsDark

```
//before: none  
//after: robot has turned 90 degree to right  
public void turnRight()
```

```
//returns true if the cell on right side of the robot is dark, otherwise false  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private boolean rightCellIsDark() {  
    turnRight();  
    robot.move();  
    boolean isDark = robot.onDark();  
    turnAround();  
    robot.move();  
    turnRight();  
    return isDark;  
} //rightCellIsDark
```

# Metoden turnRight

```
//before: none  
//after: robot has turned 90 degree to right  
public void turnRight() {  
    turnAround();  
    robot.turnLeft();  
}  
//turnRight
```

## Metoden changeColors

```
//change color of the cell on left side of the robot  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void changeColorOfLeftCell()
```

```
//change color of the cell on right side of the robot  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void changeColorOfRightCell()
```

```
//change colors of the cells on left side and on right side of the robot  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void changeColors() {  
    changeColorOfLeftCell();  
    changeColorOfRightCell();  
} //changeColors
```



## Metoden `changeColorOfLeftCell`

```
//switch color of the cell  
//before: none  
//after: if the cell the robot is at was dark it has become light,  
//       and if the cell was light it has become dark  
private void switchColor()
```

```
//change color of the cell on left side of the robot  
//before: robot is in the corridor facing the corridor  
//after: robot is in the corridor facing the corridor  
private void changeColorOfLeftCell() {  
    robot.turnLeft();  
    robot.move();  
    switchColor();  
    turnAround();  
    robot.move();  
    robot.turnLeft();  
}  
//changeColorOfLeftCell
```

## Metoden `changeColorOfRightCell`

```
//change color of the cell on right side of the robot
//before: robot is in the corridor facing the corridor
//after: robot is in the corridor facing the corridor
private void changeColorOfRightCell() {
    turnRight();
    robot.move();
    switchColor();
    turnAround();
    robot.move();
    turnRight();
} //changeColorOfRightCell
```

## Metoden **switchColor**

```
//switch color of the cell
//before: none
//after: if the cell the robot is at was dark it has become light,
//      and if the cell was light it has become dark
private void switchColor() {
    if (robot.onDark())
        robot.makeLight();
    else
        robot.makeDark();
} //switchColor
```

## Metoden `returnToStartPosition`

```
//before: robot is in the corridor, facing the cooridor  
//after: robot is at the end of the corridor, facing the wall  
private void goToEndOfCorridor()
```

```
//before: robot is at end of the corridor, facing the wall  
//after: robot is at beginning of the corridor, facing the corridor  
private void returnToStartPosition() {  
    turnAround();  
    goToEndOfCorridor();  
    turnAround();  
}  
} //returnToStartPosition
```

# Metoden returnToStartPosition

```
//before: robot is in the corridor, facing the cooridor  
//after: robot is at the end of the corridor, facing the wall  
private void goToEndOfCorridor() {  
    while (!atEndOfCorridor()) {  
        robot.move();  
    }  
} //returnToStartPosition
```