Objektorienterad programmering

Föreläsning 5: mer om metoder och abstraktioner

Dr. Alex Gerdes | Dr. Carlo A. Furia

Hösttermin 2016

Chalmers University of Technology

Några meddelande

- Man får inte ändra klassen `Robot`, `Location`, osv i lab 2
- Handledarna rättar efter deadlinen

Sammanfattning föreläsning 4

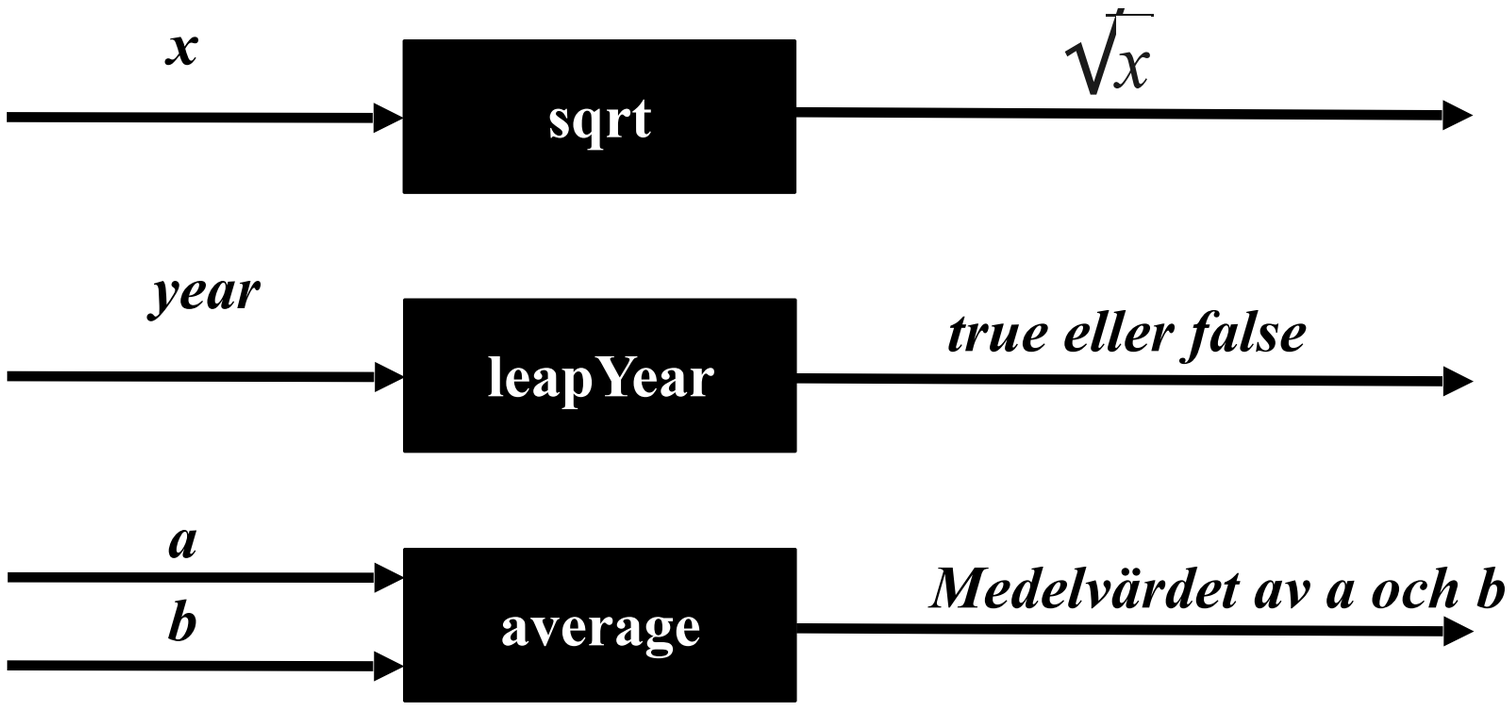
- **Abstraktion**, top-down design
- Gränssnitt av en metod
- Parameteröverföring
- För- och eftervillkor
- Kopiera/klistra -> larm!

Abstraktion

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
- 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

Elimineringsspelet Mexico

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).

Uppgiften

- 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?*
- 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
// after:  returns the score with lowest score value
private static int lowestScoreValue(int score1, int score2)
```

Det färdiga gränssnittet

- 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

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

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.e. 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

CHALMERS

```
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");
}
```

Testdriven programutveckling

CHALMERS

```
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);
  boolean passed = result == exp;

  if (passed)
    System.out.println("Ok!");
  else
    System.out.println("Failed: " + result + " should be " + exp);

  return passed;
}
```

Implementation

- Börja med att ställa dig frågan:

Har jag löst något liknande problem tidigare?

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

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

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

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
// after:  returns the comparison value of score
private static int scoreValue(int score)
```

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

  if (scoreValue(score1) < scoreValue(score2))
    low = score1;
  else
    low = score2;

  return low;
}
```

Implementation

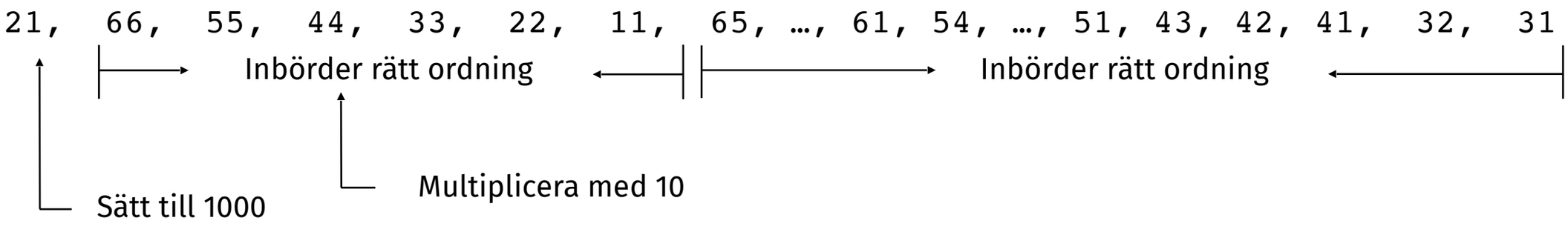
- Problemet vi har är att för varje tärningssumma

  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?*

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



Implementation

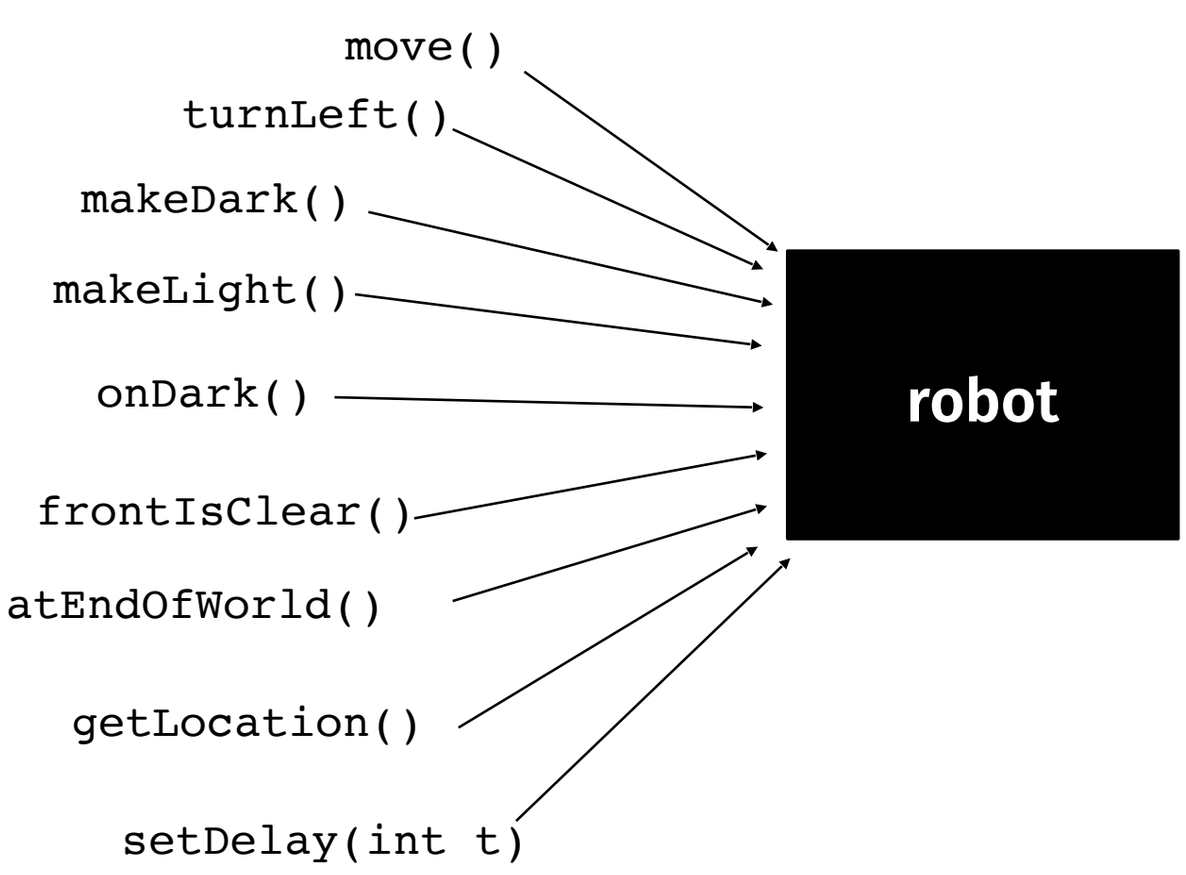
```
// before: score is a valid Mexico score
// after:  returns the comparison value of score
public static int getScoreValue(int score) {
  int comparisonValue;

  if (score == 21)
    comparisonValue = 1000;
  else if (score / 10 == score % 10)
    comparisonValue = score * 10;
  else
    comparisonValue = score;

  return comparisonValue;
}
```

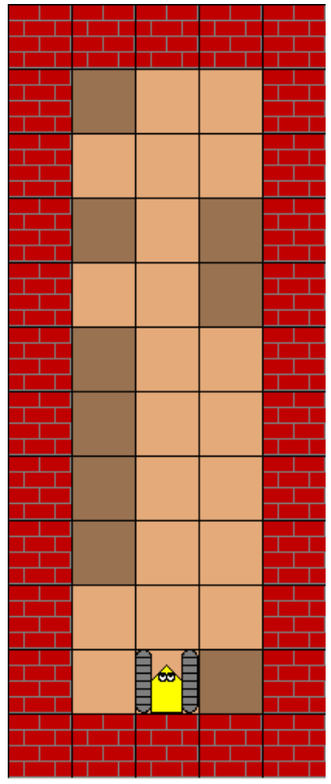
Abstraktioner på en annan nivå

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

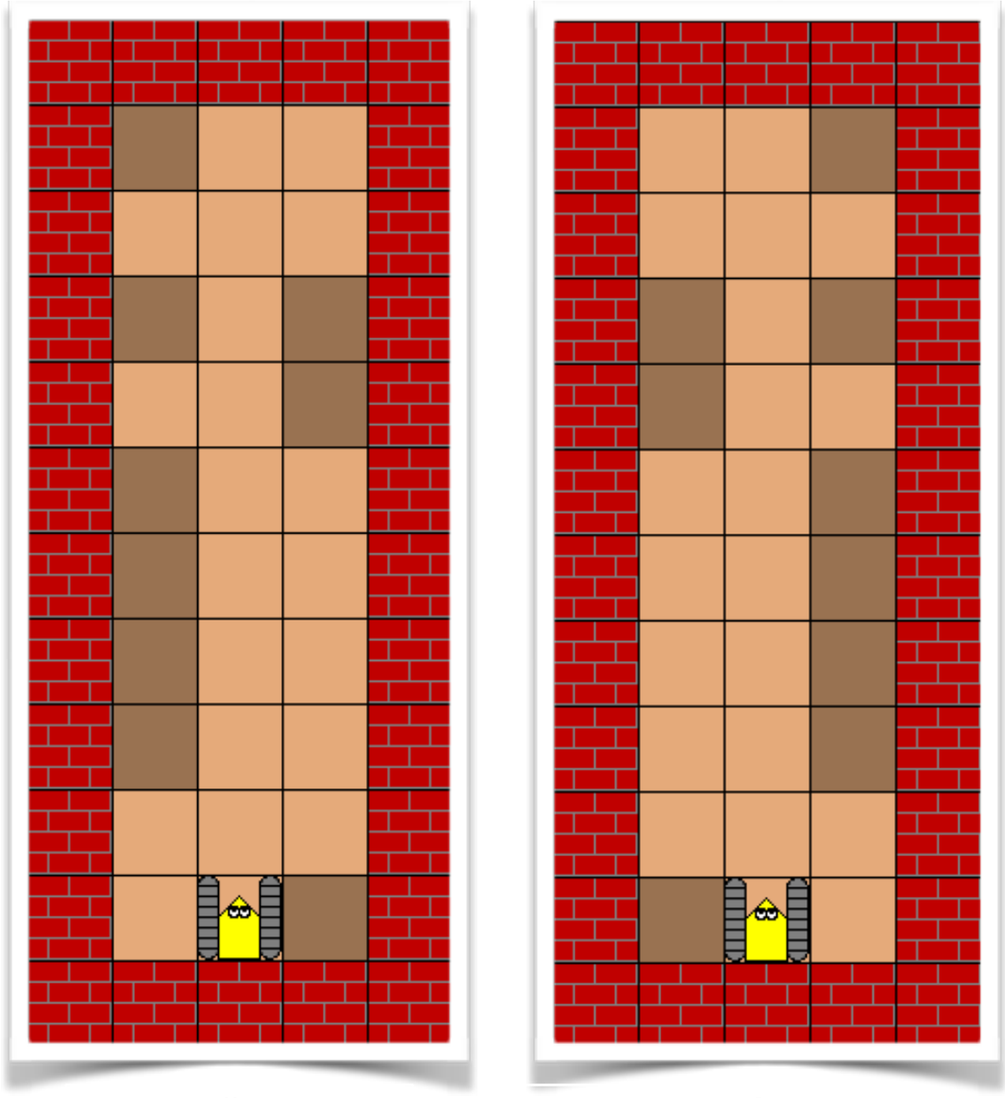
Metod	Användning
`void move()`	förflyttar sig ett steg framåt, om roboten hamnar utanför världen eller i en mur fås ett exekveringsfel
`boolean frontIsClear()`	returnerar `true` om det är möjligt för roboten att göra `move()` utan att ett exekveringsfel erhålls, annars returnera `false`
`void turnLeft()`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

Swap

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 sig ursprungliga startposition.

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



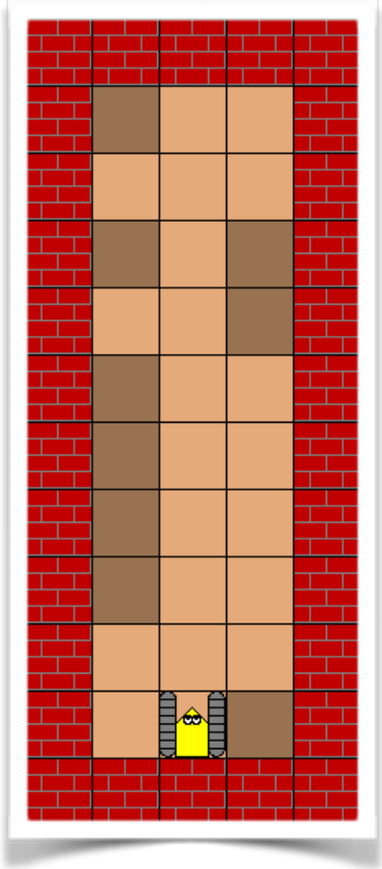
Före Efter

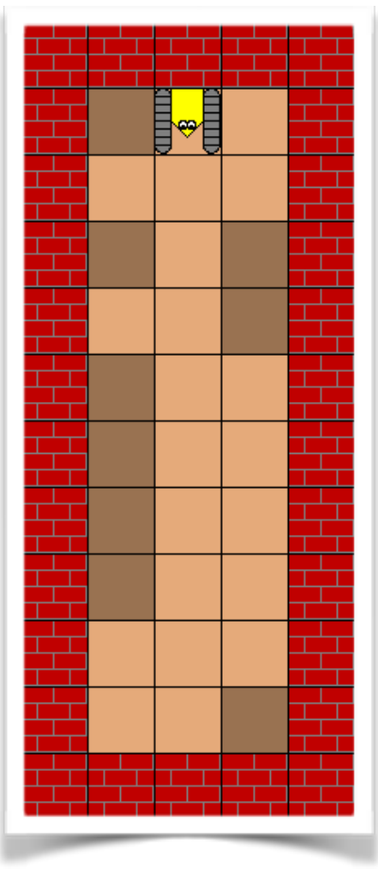
Problem

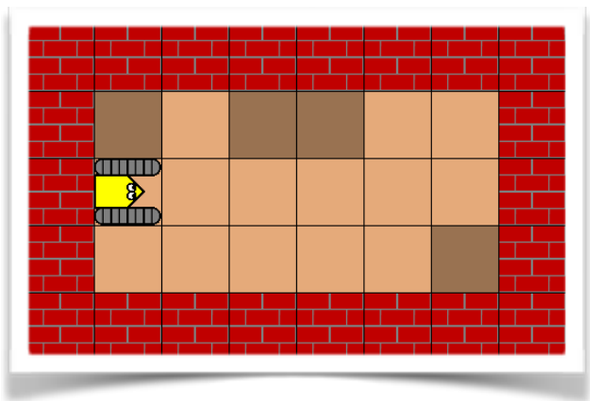
- Roboten skall kunna utföra sin arbetsuppgift i en godtycklig värld med samma principiella uppbyggnad som världen i figurerna bredvid.

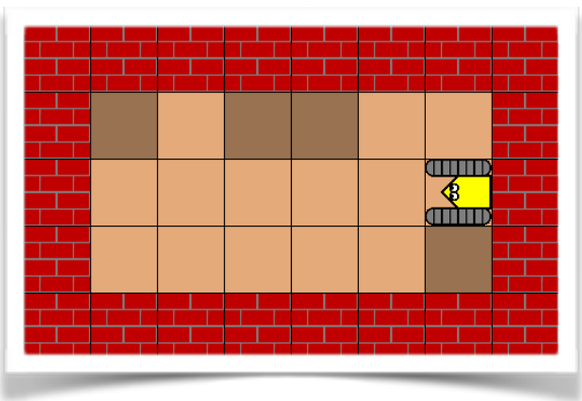
- Var noggrann med att beskriva vad varje metod gör och vilka för- respektive eftervillkor som gäller (t.ex. vilken riktning roboten måste ha innan metoden anropas, och vilken riktning roboten får efter anropet). Välj också med omsorg lämpliga och beskrivande namn på metoderna.

- 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.

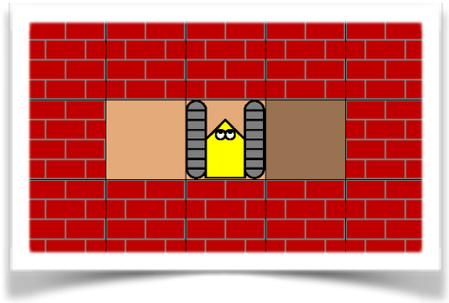






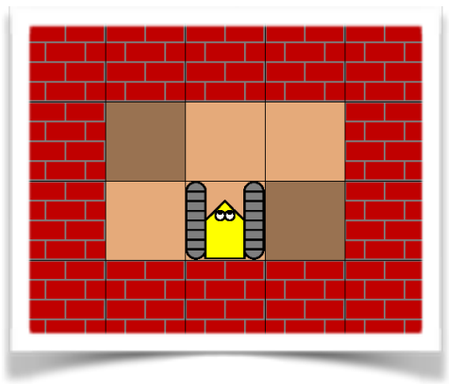


Lös problemet



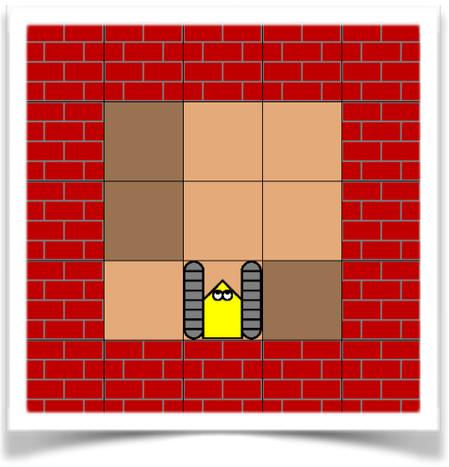
```
swapTwoCells();
```

- Handfallen? Rådvill? Kommer inte igång?



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

- Förenkla problemet! Titta på en *mindre* instans av problemet!



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

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

Metoden `swapAll`

```
// 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();
}
```

```
// 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 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()
```

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

Metoden `atEndOfCorridor`

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

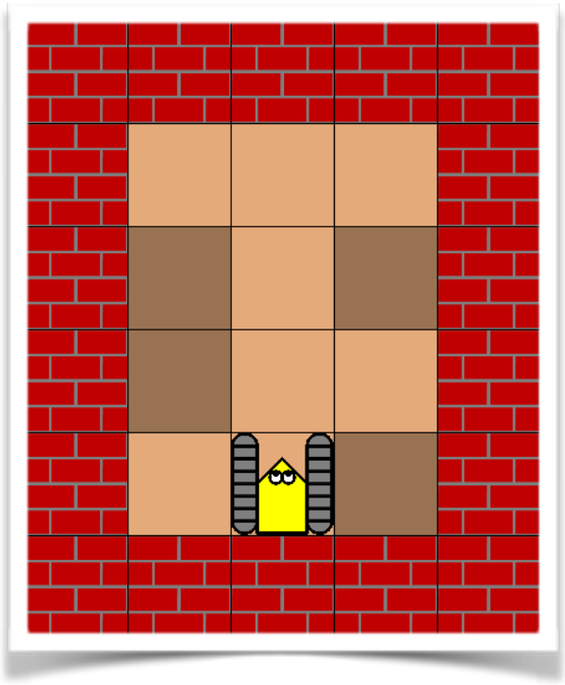
Metoden `swapTwoCells`

- *När behöver färgerna på cellerna bytas?*
- När cellerna har olika färg!
- Kan uttryckas som

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

- Alternativt

```
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`

```
// 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();
}
```

```
// 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 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()
```

```
// 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 `leftCellIsDark`

```java
// 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;
}
```

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

Metoden `turnAround`

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

Metoden `rightCellIsDark`

```
// 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;
}
```

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

Metoden `turnRight`

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

Metoden `swapTwoCells`

```
// 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();
}
```

```
// 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()
```

Metoden `changeColorOfLeftCell`

```
// 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();
}
```

```
// 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()
```

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();
}
```

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();
}
```

Metoden `returnToStartPosition`

```
// 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();
}
```

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

Metoden `goToEndOfCorridor`

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

Demo: `Swapper`