Inför projektet:
Några vanliga problem i granskningen

Vad är kodlukt?

Kodlukt

jonkv@ida

- **Kodlukt / code smell**:
  - Något man kan se "**på ytan**" på koden,
    som indikerar att det *kan finnas* ett **djupare problem**
    - Lukten är inte problemet, utan ett *tecken* på problem
  - Fixas ofta genom **refactoring**: Kod skrivs om, men gör exakt samma sak

Upprepning, redundans

Redundans?

jonkv@ida

- I vissa sammanhang kan redundans ha sina poänger…
  - *This parrot is no more. It has ceased to be.*
    It's expired and gone to meet its maker.
    This is a late parrot. It's a stiff. Bereft of life, it rests in peace.
    If you hadn't nailed it to the perch, it would be pushing up the daisies.
    It's rung down the curtain and joined the choir invisible.
    This is an ex-parrot.

Undvik redundans

- I programmering ska redundans och upprepning undvikas!
  - Gör det **svårare att förstå** koden
    - Mer att läsa
    - Måste se om upprepningen är exakt lika eller skiljer sig på något sätt
  - **Mer kod att underhålla**
  - **Risk att inte all kod uppdateras** på samma sätt

DRY-principen: Don't Repeat Yourself!
DIE: Duplication Is Evil

WET: Write Everything Twice
WET: We Enjoy Typing…

Onödigt många metoder

Onödigt många metoder

jonkv@ida

- Kod ska skrivas **effektivt** – exempel:

```
public void activateUp() {
    up = true;
}
public void deactivateUp() {
    up = false;
}
```

```
public void setUp(boolean up) {
   this.up = up;
}
```

```
public void moveLeft() {
    …;
}
public void moveRight() {
    …;
}
```

```
public void move(Direction dir) {
    …;
}
```

Onödig upprepning av kodmönster

```
public void checkKeyboard() {
    if (isPressed(Key.UP)) {
        player.directions.add(Direction.NORTH);
    } else {
        player.directions.remove(Direction.NORTH);
    }
    if (isPressed(Key.DOWN)) {
        player.directions.add(Direction.SOUTH);
    } else {
        player.directions.remove(Direction.SOUTH);
    }
    if (isPressed(Key.LEFT)) {
        player.directions.add(Direction.WEST);
    …
```

Extrahera en hjälpmetod!

```
public void checkKeyboard() {
    checkKey(Key.UP, Direction.NORTH);
    checkKey(Key.DOWN,Direction.SOUTH);
    checkKey(Key.LEFT, Direction.WEST);
    checkKey(Key.RIGHT, Direction.EAST);
}
```

```
private void checkKey(Key key, Direction dir) {
    if (isPressed(key)) {
        player.directions.add(dir);
    } else {
        player.directions.remove(dir);
} }
```

Upprepning av kodmönster (2)

jonkv@ida

- Extrahera **upprepade uttryck** till namngivna variabler

NameProcessor.java

```
static void printNames(final String fullName) {
        System.out.println(fullName.substring(fullName.indexOf(" ") + 1));
        System.out.println(fullName.substring(0, fullName.indexOf(" ")));
}
```

Multiple occurrences found

Replace this occurrence only

Replace all 2 occurrences

NameProcessor.java

```
static void printNames(final String fullName) {
    int firstNameEndIndex = fullName.indexOf(" ");
    System.out.println(fullName.substring(firstNameEndIndex + 1));
        System.out.println(fullName.substring(0, firstNameEndIndex));
}
```

Snabbare kod

Namnet ger en *förklaring* till deluttrycket

Lättare att se att delarna är *medvetet* identiska

IDEA kan hjälpa till: Refactor | Extract | Variable

Upprepning i konstruktorer

jonkv@ida

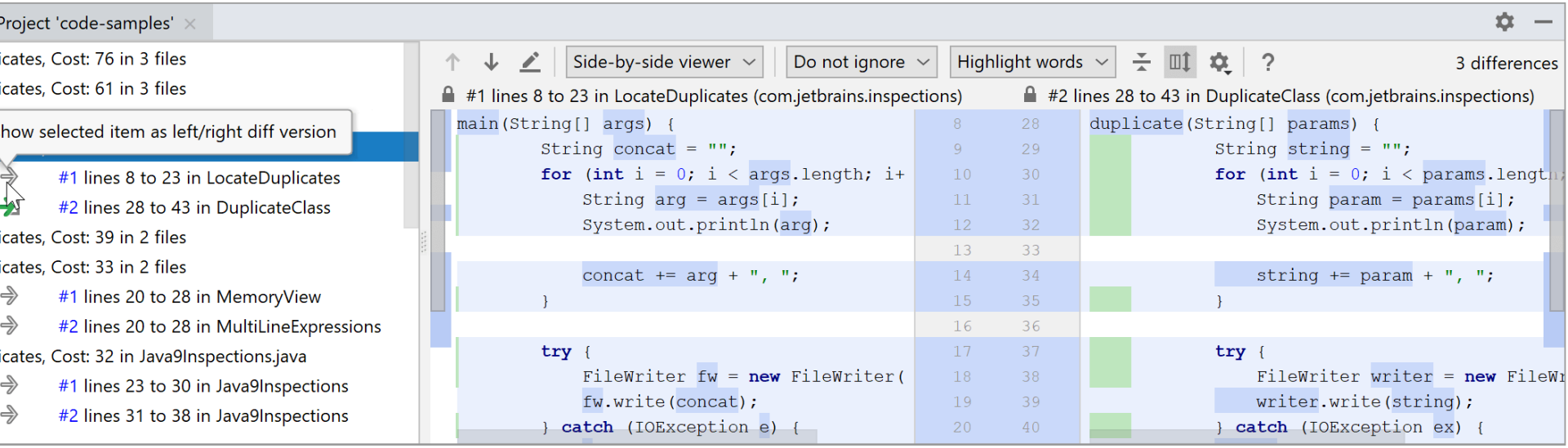
- Många konstruktorer ➔ upprepad kod?
  - En konstruktor kan anropa en annan – en "kedja"

```
class Circle {
  double x, y, r;
  Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
    // Testa att r >= 0
    // Beräkna arean
    // Mer initialiseringskod...
  }
  Circle(Circle other) {
    this(other.x, other.y, other.r);
  }
}
```

Specialsyntax:
this(args)
som *första* sats i en konstruktor
vidarekopplar till en *annan* konstruktor

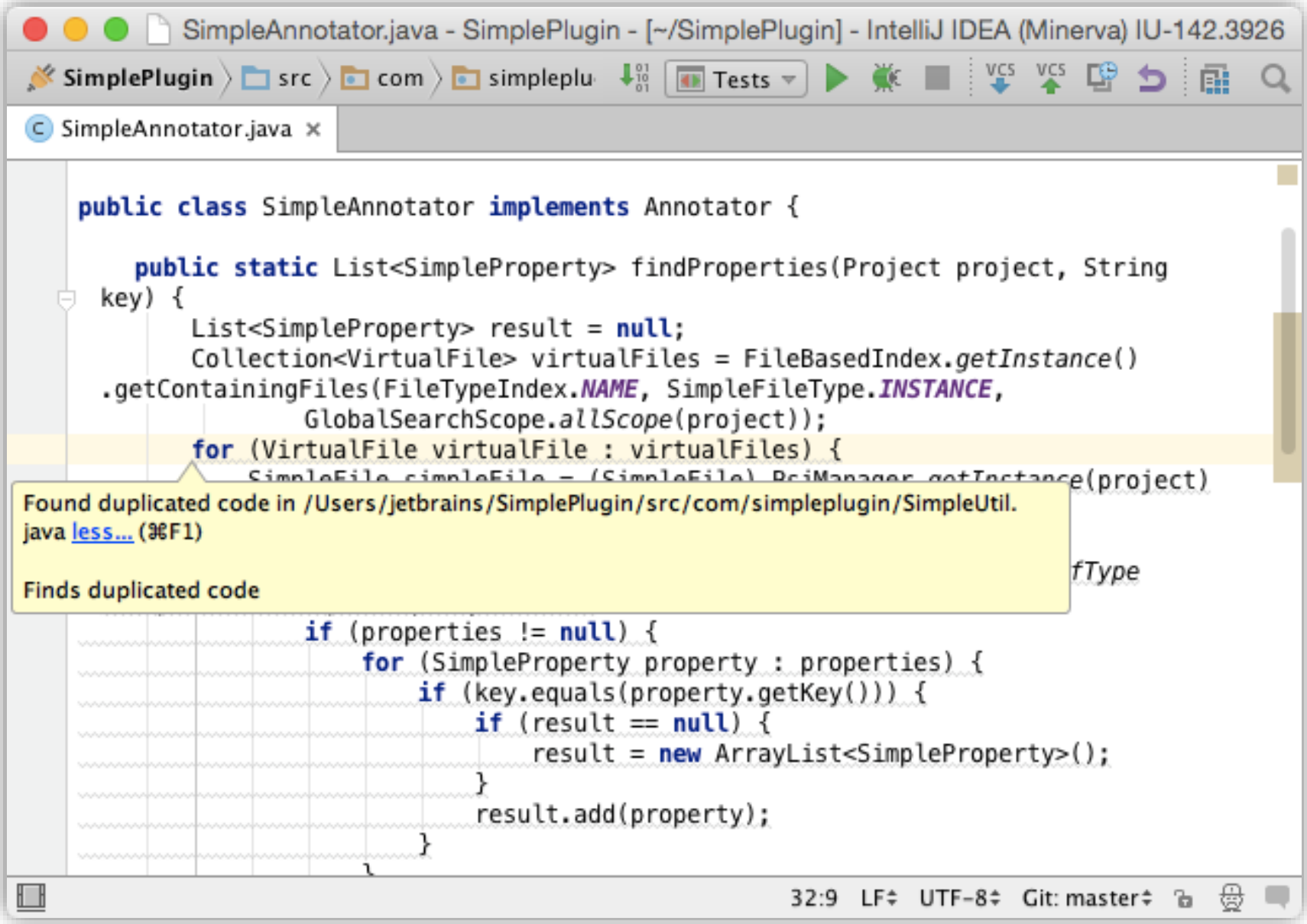
Att hitta upprepad kod

- IDEA kan hitta vissa former av **uppenbar repetition**
  - Analyze | Locate Duplicates



Ni gör resten – alltför redundant kod ger komplettering!

Kan även hitta duplicerad kod ”on the fly”



Avsaknad av dokumentation med Javadoc

Python: Docstrings

Java: **Javadoc**

Javadoc-kommentarer

- Kan dokumentera **klasser**, **fält**, **metoder**, **konstruktorer**, …
  - Kommentar omedelbart **ovanför** en deklaration

```
/**
 * Removes all mappings from this map (optional operation).
 *
 * @throws UnsupportedOperationException clear
 *   is not supported by this map.
 */
public void clear() {
   …
}
```

Startar med /**

Typ av info kan markeras…

Slutar med */

https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Javadoc (2)

- Korrekt format är viktigt: Javadoc används av **verktyg**!
  - IntelliJ IDEA: Tryck Ctrl-Q för att se Quick Documentation

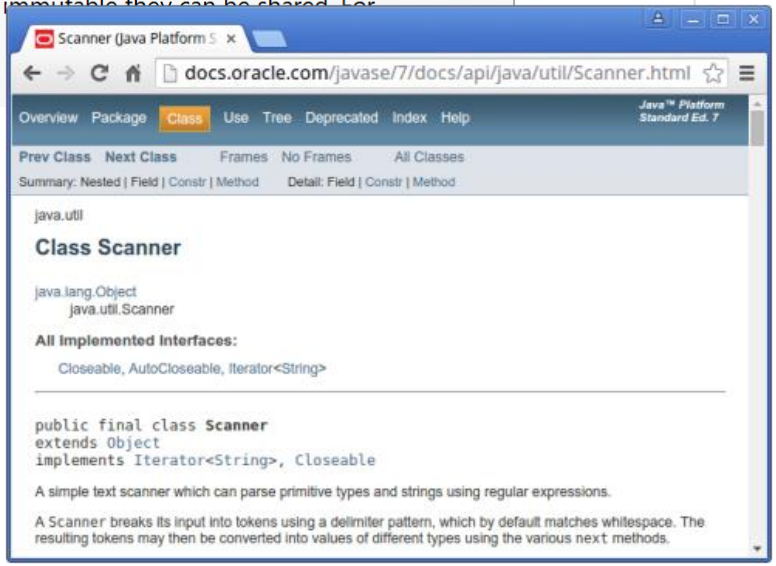
```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(
    }
}
```

```
java.lang
public final class String
extends Object
implements java.io.Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```



- Javadoc (kommandorad): Skapa websidor för API-dokumentation

Kan inte säga ”as above”, går inte att ersätta med annan kommentarstyp eller flytta!

Javadoc (3)

jonkv@ida

- Krav på javadoc i projektet:
  - Alla klasser
  - Alla publika fält
  - *Där det behövs* – mindre uppenbara delar av projektet

Kod som inte reflekterar / visar upp
de underliggande idéerna

Begrepp 1

- Commodore 64 BASIC V2:

”Anropa en funktion”:
GOSUB 1000
Tänker vi så?

```
10 PRINT CHR$(147)
20 SP = 20: ZE = 3: A$ = "Good Morning!": GOSUB 1000: GOSUB 2000
30 SP = 10: ZE = 3: A$ = "I'm the Commodore 64": GOSUB 1000: GOSUB 2000
40 SP = 12: ZE = 6: A$ = "And what is your name ?": GOSUB 1000
100 END
1000 REM cursor positioning and printing
1010 POKE 211,SP :POKE 214, ZE: SYS 58640 : PRINT A$
1020 RETURN
2000 REM delay-loop
2010 FOR X=0 TO 3000: NEXT X
2020 RETURN
```

POKE 211, SP ➔
set_column(SP)…

Knappast.
Vore bättre att kunna skriva
print_at(20, 3, ”Good Morning!”)

Vi vill ha begreppet print_at()!

Begrepp 2

jonkv@ida

- Java:
  - Det går utmärkt att skriva…

```
public class Game {
   public void moveEnemies() {
      …
      …
      …
      double newX = x + Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
      …
      …
}
```

Men det uttrycker inte vad vi tänkte!

Vad tänkte vi?

Om vi inte vet vad programmeraren tänkte,
hur vet vi då om det är korrekt implementerat?

Begrepp 3

- Java:
  - Det är lite bättre att skriva:

```
public class Game {
   public void moveEnemies() {
      …
      …
      …
      double distance = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
      double newX = x + distance;
      …
      …
}
```

Ge namn till deluttryck!

Nu ser vi lite mer av vad programmeraren tänkte!

Begrepp 4

jonkv@ida

- Java:
  - Det är ännu bättre att skriva:

Som att utöka språket med meningsfulla begrepp!

```
public class Game {
   public double distance(double x1, double y1, double x2, double y2) {
      return Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
   }
   public void moveEnemies() {
      …
      …
      double newX = x + distance(x1, y1, x2, y1);
      …
      …
}
```

Själva avståndsberäkningen är ett meningsfullt begrepp

Som en metod kan detta återanvändas och anropas igen

Uttrycket reflekterar direkt en tanke:
Nya värdet är x plus avståndet mellan…

Begrepp 5

jonkv@ida

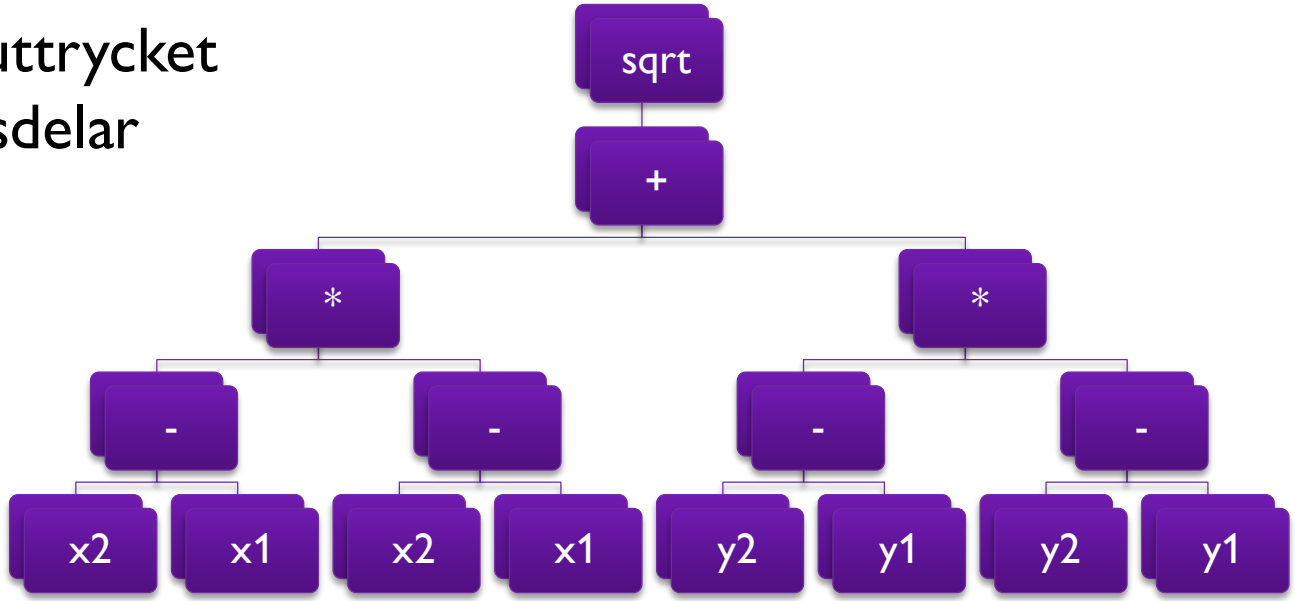
- Bättre att införa *punkter* som egna begrepp!
  - Vi vill ha avstånd mellan två punkter/saker, inte mellan 4 flyttal…

```
public class Game {
   public double distance(Point p1, Point p2) {
      return Math.sqrt((p2.x – p1.x)*(p2.x – p1.x) +
                       (p2.y – p1.y)*(p2.y – p1.y));
   }
   public void moveEnemies() {
      …
      …
      double newX = x + distance(enemy.getPos(), player.getPos());
      …
      …
}
```

Begrepp 6

jonkv@ida

- Det matematiska uttrycket har också beståndsdelar
  - Kan införa namn på olika delar




```
public class Game {
   public double distance(Point p1, Point p2) {
      double deltaX = p2.x – p1.x;
      double deltaY = p2.y – p1.y;
      return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
   }
}
```

Listor eller enskilda variabler?

Loopar eller enskilda satser?

Listor 1

jonkv@ida

- För att modellera en **fast uppsättning objekt**:
  - Kan vara frestande att använda **flera variabler**

```
public class Game {
   // Vi har två spelare, så vi använder två separata fält...
   private Player player1;
   private Player player2;
   // ...
   private Level level1;
   private Level level2;
   private Level level3;
}
```

Fungerar säkert...

Men det är omöjligt att prata om
”alla spelare”,
”alla nivåer”,
”nivåerna mellan 2 och 7”,
...

Listor 2

jonkv@ida

- Eller kod som denna:

```
public class Game {
   private Level level1;
   private Level level2;
   private Level level3;

   private void setLevel(int level) {
      if (level == 1)
         this.level = level1;
      else if (level == 2)
         this.level = level2;
      else if (level == 3)
         this.level = level3;
   }
}
```

Onödigt mycket kod

Måste uppdatera koden när man skapar nya nivåer, inte bara nivålistan

Lätt att göra fel någonstans

Listor 3

jonkv@ida

- Kan lätt leda till kod som denna:

```
public class Game {
   private Player player1;
   private Player player2;

   private void playerTakesPowerup(int playerID) {
      if (playerID == 1)
         player1.addPowerup();
      else if (playerID == 2)
         player2.addPowerup();
   }
}
```

Ser koden ut så här
går det alltid att göra bättre…

Listor 4

jonkv@ida

- För att vår **tanke** ”spelarna”
  ska kunna **materialiseras** i ett **objekt** ”spelarna”:

```
public class Game {
   private List<Player> players;

    private void tick() {
      for (Player player : players) { … }
   }

   public Player getPlayer(int index) {
      return players.get(index);
   }
}
```

Kan prata om ”spelarna”,
kan loopa över ”spelarna”

Om du tänker att något ska göras ”för alla”,
ska du väldigt sällan göra detta separat ”för A” och sedan ”för B”,
även om det bara finns A och B (2 objekt).
Låt tanken ”för alla” synas i koden!

Data är bättre än kod

Att undvika hårdkodning och bli mer datadriven

Data: EnumMap

jonkv@ida

Tillkommer ny SquareType måste vi ändra koden, kompilera om
Vill vi ändra en färg …

```
SquareType st = board.get(x,y);
switch (st) {
   case EMPTY:
      g2d.setColor(WHITE);
      break;
   case L:
      g2d.setColor(BLACK);
      break;
   case I:
      g2d.setColor(GREEN);
      break;
   case Z:
      g2d.setColor(BLUE);
      break;
   …
```

Kan skicka med olika färgkartor, läsa en Map från fil, …

Data är flexiblare!

```
EnumMap<SquareType, Color) colors =
   new EnumMap<>();

colors.put(SquareType.EMPTY, WHITE);
colors.put(SquareType.L, BLACK);
…

SquareType st = board.get(x,y);
g2d.setColor(colors.get(st));
```

Gäller i princip alla if/switch-kedjor där man bara vill hitta ett *värde*:
Sådant ska man slå upp i en mappning

```
if (name.equals(”Cow”)) icon = ”cow.jpg”;
```

Data: EnumMap

jonkv@ida

Tillkommer nya nivåer måste vi ändra denna kod…

Jobbig kod att läsa!

```
switch (level) {
   case 1:
      this.speed = 7
      this.bonus = 3
      break;
   case 2:
      this.speed = 10;
      this.bonus = 5;
      break;
  …
}
```

Kan lätt skapa nya nivåer, skicka runt dem som data, spara på fil!
Inte upprepad kod,
många rader per nivå

```
public class Level {
   private final int speed;
   private final int bonus;
   … constructor, getters, …
}

List<Level> levels = List.of(
   new Level(7, 3), new Level(10, 5)
);

this.speed = level.getSpeed();
this.bonus = level.getBonus();
```

Information ska inte representeras som procedurell kod!