Læringsmål for forelesningen

- Objektorientering
  - Regler for oppførsel
- Java-programmering
  - Enhets-testning med Junit 5

- VS Code
  - Opprette JUnit-test og kjøre den

Pensum

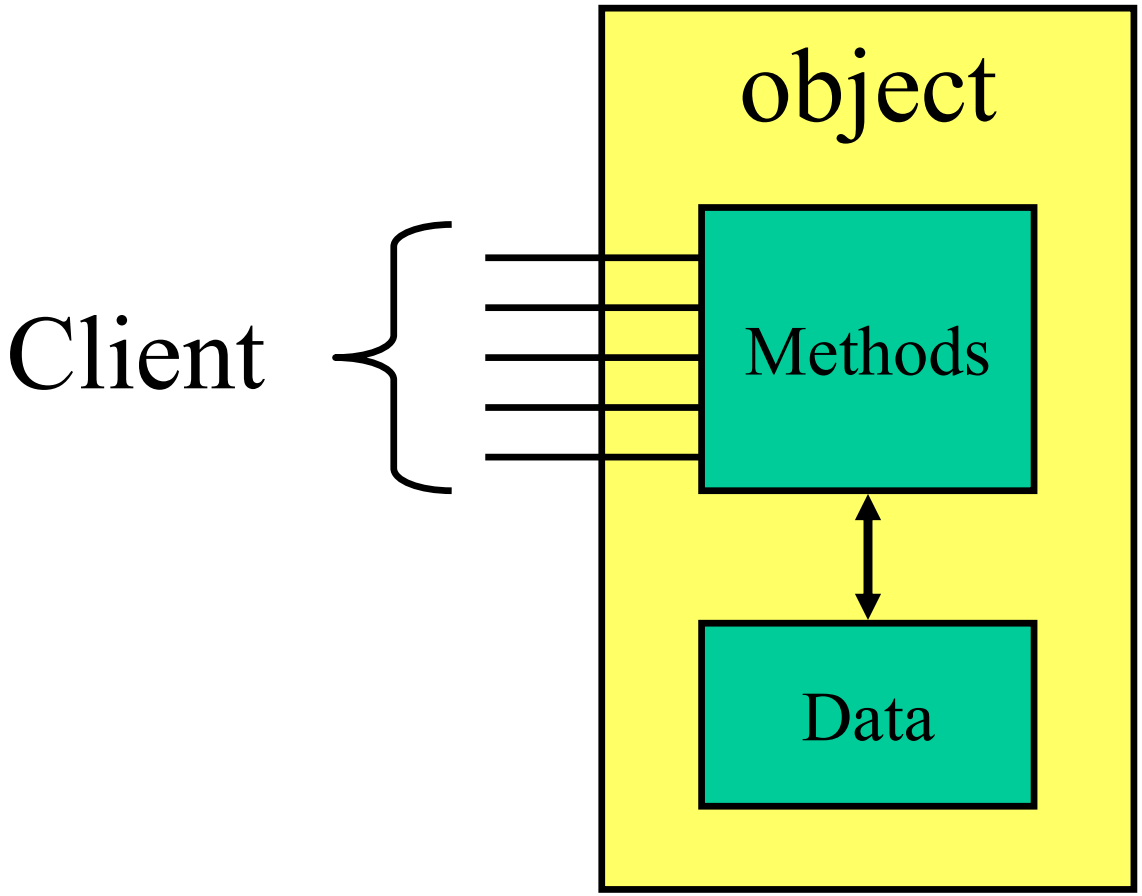
- Testing dekkes ikke godt av boka!
- Er en viktig del av programvare-utvikling og er del av pensum!
- Pensum for dette er:
  - Forelesingsnotater
  - Øvingene, inkludert testene som legges ved
- Enkel teori å lære, men krever øving!

Først litt repetisjon!

Innkapsling og metode-grensesnitt

Innkapsling av objekter, ved å definere et metode-*grensesnitt* som sikrer riktig bruk, er et sentral poeng innen objektorientering

Illustrasjon av innkapsling



Et grensesnitt er mer enn et sett metoder, metodene må også garantere en bestemt oppførsel!

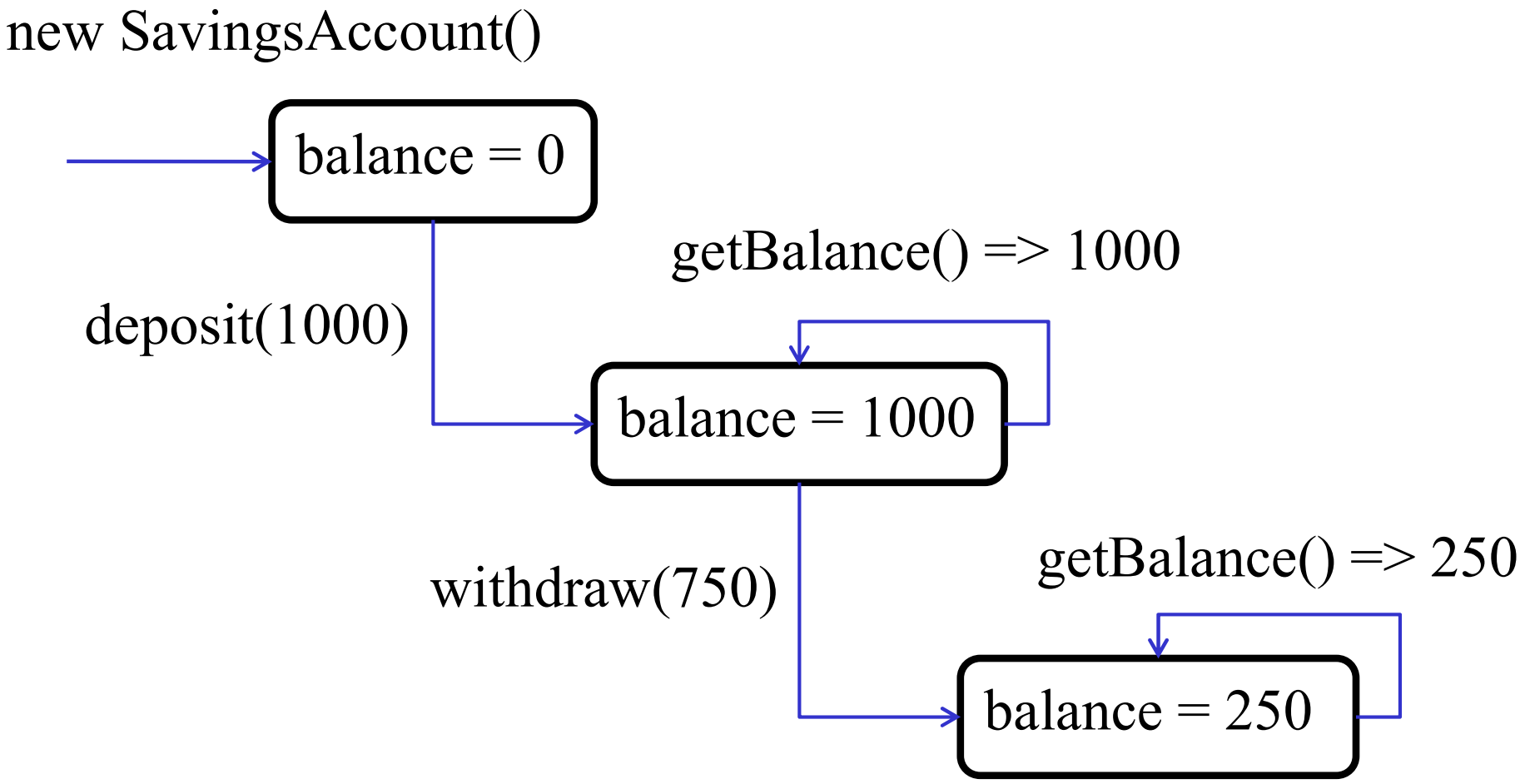
For å gjøre beskrivelsen av et grensesnitt komplett, må en også beskrive regler for hvordan metodene skal oppføre seg

Høres vanskelig og abstrakt ut?
La oss se på et enkelt eksempel

Regler for oppførsel

- Noen metoder leser tilstanden, mens andre endrer tilstanden.
- En måte å beskrive oppførsel på er å vise hvordan endringsmetodene skal påvirke resultatene fra lese-metodene.
- Eksempler:
  - etter `konto = new SavingsAccount()` `konto.deposit(1000)`
  - vil `konto.getBalance()` returnere `1000`
  - videre, etter `konto.withdraw(750)`
  - vil `konto.getBalance()` returnere `250`

Regler for oppførsel, som objekttilstandsdiagram



Regler for oppførsel

- Ved systematisk gjennomgang av metodene, kan en formulere mange slike regler
- Regler formulert på denne måten, kan kodes som et testprogram som
  - setter en tilstand,
  - utfører endringer, og
  - tester resultatet av lesinger og sier fra om feil

Regler for oppførsel

1. konto = new SavingsAccount()
   => konto.getBalance() == 0

2. konto.deposit(1000)
   => konto.getBalance() == 1000

3. konto.deposit(1000)
   => konto.getBalance == 2000

4. konto.withdraw(1500)
   => konto.getBalance == 500

5. konto.withdraw(500)
   => konto.getBalance() == 0

6. konto.withdraw(500)
   => konto.getBalance() == 0

Mål

- Vi ønsker objekter som oppfører seg som planlagt!
- Reglene definerer ønsket oppførsel
- Testing er nødvendig for å sjekke at objektene oppfører seg i henhold til reglene
- Systematisk testing må til for å avdekke feil i koden som vi lett kan overse
- Tester brukes for å avdekke feil som innføres ved vikere utvikling av koden.

Testing så langt i kurset

- Frem til nå har vi "testet" metoder vha.:
  - main-metode som skriver ut tilstanden til objektene etter at vi har gjort endringer
  - utskrift i metodene som viser endringer metodene gjør
  - kjøring i debug-modus
  - øvingene har ferdiglagde JUnit-tester
- Lite systematisk
  - De fleste har sikkert opplevd at det kan finnes feil
    - som som ikke blir oppdaget før etter en stund
    - som kun oppstår i spesielle situasjoner
    - Som blir innført ved senere endringer av koden.

Eksemplet kodet som testmetode

```
Account konto = new SavingsAccount();
if (konto.getBalance() != 0){
    System.err.println("Feil!");
}
konto.deposit(1000);
if (konto.getBalance() != 1000){
    System.err.println("Feil!");
}
konto.deposit(1000);
if (konto.getBalance() != 2000){
    System.err.println("Feil!");
}
konto.withdraw(1500);
if (konto.getBalance() != 500){
    System.err.println("Feil!");
}
konto.withdraw(500);
if (konto.getBalance() != 0){
    System.err.println("Feil!");
}
konto.withdraw(500);
if (konto.getBalance() != 0){
    System.err.println("Feil!");
}
```

Ikke veldig elegant, men uten andre testverktøy er dette en veldig nyttig og praktisk metode.

Hjelpemetoder for testing

- En hjelpemetode for å sjekke testresultat og si fra om feil

```
private static void test(boolean resultat) {
    if (! resultat) {
        System.err.println("Feil!!!");
    }
}
```

hjelpemetode

- Vi fletter testkode inn i koden som setter opp objektstrukturen vår

```
Account konto = new SavingsAccount();
test(konto.getBalance() == 0);
konto.deposit(1000);
test(konto.getBalance() == 1000);
konto.deposit(1000);
test(konto.getBalance() == 2000);
```

testkode

- Det er bedre å skille testkoden fra programkoden og gjøre mer systematisk testing av ulike samhandlingsmønstre.

Testing som del av programvare-utvikling

- Testing som en integrert del av koden er en lite egnet metode for systematisk testing
- Det er best å skille testkoden fra program-koden og gjøre mer systematisk testing
- JUnit-rammeverket er basert på denne teknikken og det skal dere lære å bruke.

JUnit-testing

- *Enhetstesting* (eng: unit testing) er en systematisk testing av funksjonalitet som vi kan skille ut. Typisk enkeltklasser eller metdoer, eller relaterte grupper av slike som utfører en gitt oppgave.
- *JUnit* er et rammeverk for enhetstesting
- En JUnit *TestCase* er en klasse med testkode som tester oppførselen for en eller flere andre klasser
- En TestCase består i hovedsak av et sett test-metoder, som hver typisk fokuserer på en eller et lite antall regler for oppførsel.

JUnit TestCase

- En TestCase-klasse er en helt vanlig Java-klasse, men må skrives etter spesielle regler for å kunne kjøres av JUnit-rammeverket
- Annotasjoner brukes for å angi metoder med spesifikke roller
  - **@Before** – rigger opp test-data, kjøres først
  - **@Test** – test-metoder, selve testen
  - **@After** – rigger ned test-data, kjøres etterpå
- test-metodene blir kjørt/kalt uavhengig av hverandre og vil typisk fokusere på én spesifikk regel for oppførsel

La oss prøve JUnit

Lager oss en enkel
Counter-klasse og en
CounterTest-klasse med testkode

Counter-klassen

- En klasse som teller opp til en max-verdi

- count-metoden øker telleren med 1
- getCount returnerer teller-verdien
- isMax sjekker om vi har nådd maximumsverdien

```
public class Counter {

    private int count;
    private int max;

    public Counter(int max){
        count = 0;
        this.max = max;
    }

    public int getCount(){
        return count;
    }

    public void count(){
        if (count < max){
            count ++;
        }
    }

    public boolean isMax(){
        return count >= max;
    }

}
```

Ønsket oppførsel

```
Counter c = new Counter(10);
for (int i = 0; i < 12; i ++){
    c.count();
    System.out.println(c.getCount());
}
```

```
1
2
3
4
5
6
7
8
9
10
10
10
```

```
Counter c2 = new Counter(10);
while (! c2.isMax()){
    c2.count();
    System.out.println(c2.getCount());
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Junit i VSCode

- Testene legges i et parallelt med kildekoden i mapper under src/test/java/…
- Vi kan gjøre dette ved å høyreklikke og velge Source action… -> Generate tests…
- Det kommer forslag på testklassenavn. Trykk enter. (En kan også huke av for hvilke metoder en vil lage tester for, men vi skal ikke disse direkte)
- VSCode oppretter test-klassen i test-mappen.

```
package uke14.counter;

import
org.junit.jupiter.api.Test;

public class CounterTest {

…


}
```

Testklassen

- @Test angir test, som kjøres uavhengig av hverandre
- Kode vi vil gjenta i hver test kan vi legge i en @BeforeEach-metode
- Testene kan kjøres ved å trykke på pil i margen, men bedre å bruke test-view'et i VSCode
- Innfør feil i Counter-klassen, og sjekk at minst en test feiler.
- Flere annotasjoner: https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations

```java
package uke14.counter;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class CounterTest {

    private Counter counter;

    @BeforeEach
    public void setUp() {
        counter = new Counter(5);
    }

    @Test
    @DisplayName("Rett tilstand etter konstruksjon")
    public void testInitialState() {
        assertEquals(0, counter.getCount());
    }
    @Test
    public void countIncreasesCounterIfNotMax() {
        assertFalse(counter.isMax());
        for (int i = 1; i <= 5; i++) {
            counter.count();
            assertEquals(i, counter.getCount());
        }
        assertTrue(counter.isMax());
        counter.count();
        assertEquals(5, counter.getCount());
    }
}
```

assert-metoder

- assertTrue(boolean)
  - rapporterer feil hvis boolean er false
- assertFalse(boolean)
  - rapporterer feil hvis boolean er true
- assertNull(Object)
  - rapporterer feil hvis referansen IKKE er null
- assertNotNull(Object)
  - rapporterer feil hvis referansen er null
- assertEquals()
  - for objekter, strenger og basistyper
  - rapporterer feil hvis de er forskjellig

- fail() (tilsvarer assertTrue(false))
  - gir feil direkte, brukes ifm. brutt kontrollflyt
  - https://www.baeldung.com/junit-fail for noen eksempler

- *Overlagring er benyttet for assert-metodene*
- *Med ekstra String-argument (først) for informativ melding*
- *Samme metodenavn uavhenging av datatypene for verdiene som sammenlignes*
- *Etc.*

jextest – språk for JUnit-tester

- Laget for å:
  - senke terskelen for å skrive tester
  - gjøre testene raskere å skrive (for fagstaben)
  - gjøre testene enklere å lese (for studentene)
- Bygger på objekttilstandsdiagrammer
  - har syntaks som ligner på tilstander og transisjoner
  - lett å oversette objekttilstandsdiagram til jextest-kode
- Koden auto-oversettes til JUnit-tester

https://www.ntnu.no/wiki/display/tdt4100/jextest

Konsistens mellom objekter

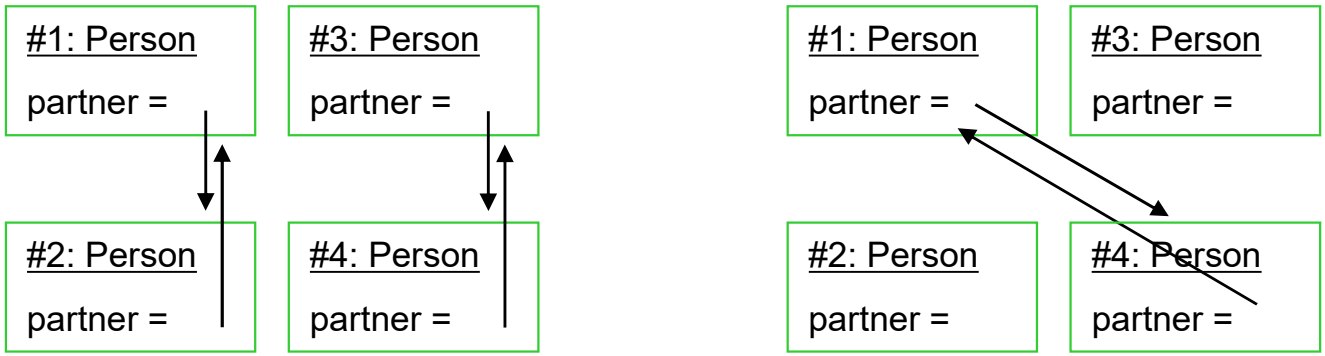
- I svært mange tilfeller finnes det avhengigheter mellom objekter, slik at en må sikre konsistens på tvers av objekter
- Eksempel: Person-klasse med partner-attributt (ekteskap/partnerskap)
  - en person kan ha 0 eller 1 partner
  - en person kan ikke ha seg selv som partner (!)
  - dersom x er partneren til y, må y være partneren til x
  - setPartner(Person partner) må kodes slik at partner-attributtet i mer enn ett Person-objekt må holdes konsistent
  - mye vanskeligere enn en skulle tro

Partner-eksempel

- Før #1.setPartner(#2) og etter:

- Deretter #1.setPartner(null):

- Før #1.setPartner(#4) og etter:

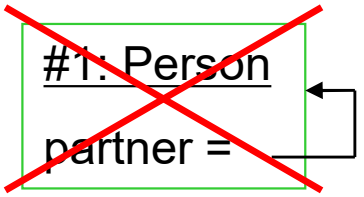


NTNU

Hvordan kodes dette i en JUnit-test?

```java
public class PersonTest {
    Person p1, p2, p3, p4;


    @BeforeEach
    void setUp() {
        p1 = new Person("Ola");
        p2 = new Person("Kari");
        p3 = new Person("Per");
        p4 = new Person("Pål");
    }

    @Test
    void testSetPartner() {
        p1.setPartner(p2);
        assertEquals(p2, p1.getPartner());
        assertEquals(p1, p2.getPartner());
    }

    @Test
    void testSetPartnerNull() {
        p1.setPartner(p2);
        p1.setPartner(null);
        assertNull(p1.getPartner());
        assertNull(p2.getPartner());
    }

    @Test
    void testSetPartnerWithExistingPartner() {
        p1.setPartner(p2);
        p3.setPartner(p4);
        p1.setPartner(p4);
        assertEquals(p4, p1.getPartner());
        assertEquals(p1, p4.getPartner());
        assertNull(p2.getPartner());
        assertNull(p3.getPartner());
    }

}
```

Partner-eksempel



- Dersom en Person settes til sin egen partner, skal det kastes en IllegalArgumentException

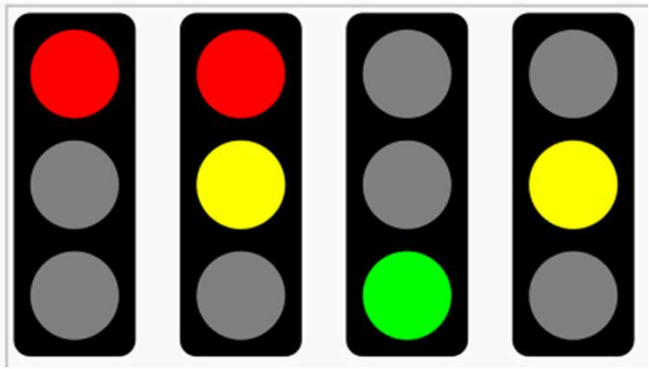
- Tre tilfeller:
  1. ingen exception: feil
  2. IllegalArgumentException: riktig!
  3. en annen type Exception

- Hvordan teste det?

Tre varianter

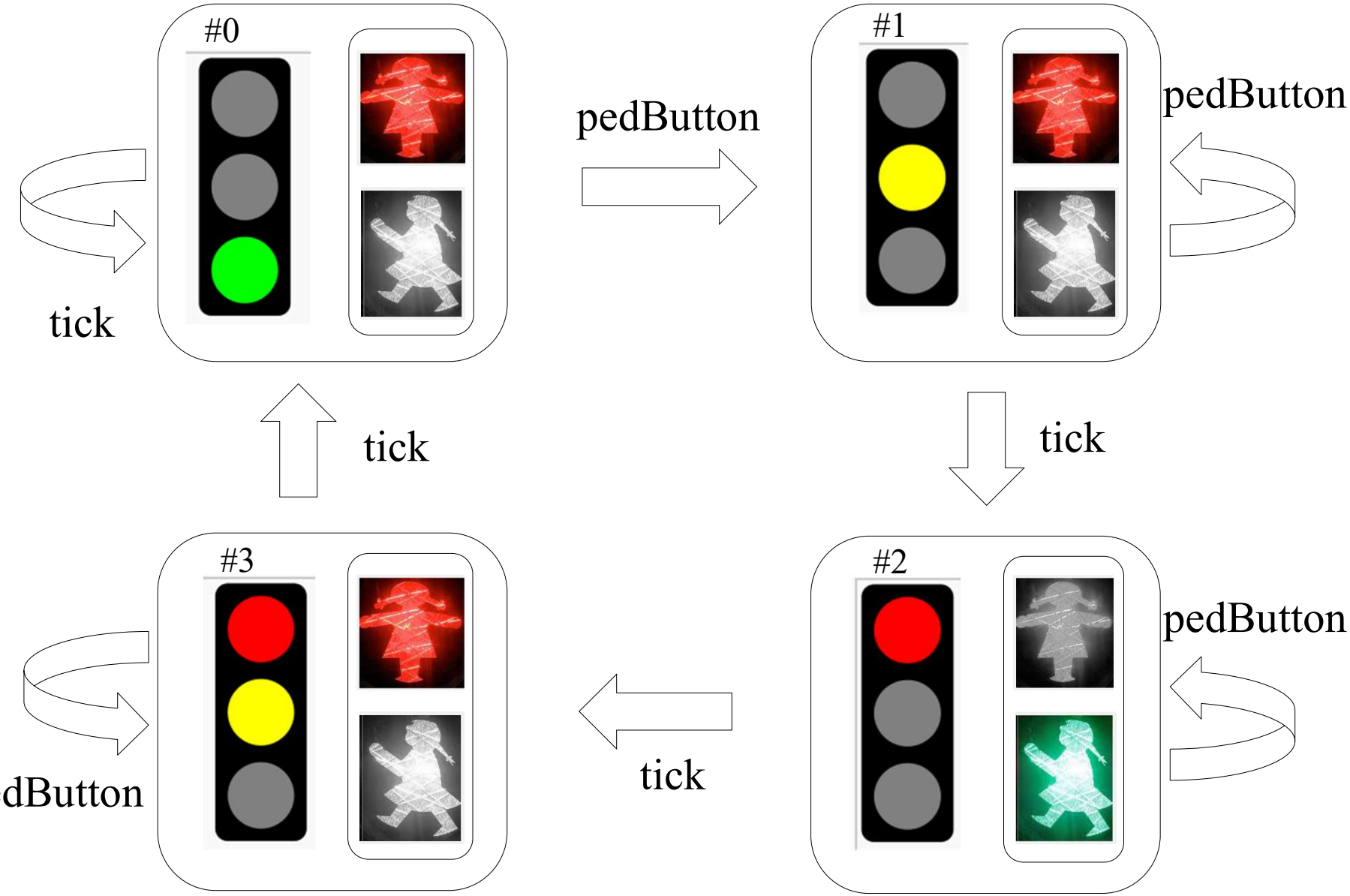
```
@Test
    void testSetPartnerWithSelf() {
        // Alternative 1 fra Junit 5
        assertThrows(IllegalArgumentException.class,
         () -> p1.setPartner(p1));

        // Alternative 2 pre JUnit 5
        try {
            p1.setPartner(p1);
            fail("Should have thrown an exception");
        } catch (IllegalArgumentException e) {
            // Expected
        } catch (Exception e) {
            fail("Should have thrown IllegalArgumentException");
        }
        // Alternative 3 også pre Junit 5
        try {
            p1.setPartner(p1);
            fail("Should have thrown an exception");
        } catch (Exception e) {
            assertTrue(e instanceof IllegalArgumentException);
        }


    }
```

Lyskryss



- To klasser
  - TrafficLight: holder rede på hvilke av et sett lys som er på
  - TrafficLightController: styrer trafikklys for biler og fotgjengere, etter gitte regler (for oppførsel)
    - tick() – går til neste tilstand
    - pedButton() - fotgjengerknapp

Ønsket oppførsel



Testing av oppførsel

- Definerer grensesnitt for TrafficLightController-logikk
- ITrafficLightController
  - isCarLight(boolean state, String… lights)
  - isPedLight(boolean state, String… lights)
  - tick() – neste (eller samme) tilstand
  - pedButton() – ønske om grønt forgjengerlys
- Og skriver test…

Læringsmål for forelesningen

- Objektorientering
  - Regler for oppførsel
- Java-programmering
  - JUnit-testing

- VSCode
  - Opprette JUnit-test og kjøre den