Forelæsning Uge 12

- **Abstrakte klasser og interfaces**
  - En abstrakt klasse er en klasse, som man ikke kan lave instanser af (men man kan lave instanser af dens subklasser)
  - En abstrakt klasse kan indeholde abstrakte metoder, hvor kun hovedet er angivet, mens implementationen (kroppen) mangler
  - I et interface er alle metoder abstrakte
- **Funktionelle interfaces**
  - Har kun én enkelt abstrakt metode
  - De steder, hvor man skal bruge et objekt, hvis type er et funktionelt interface, kan man i stedet bruge en lambda
- **Wildcards (jokere)**
  - Gør det muligt at beskrive komplicerede typebegrænsninger i forbindelse med generiske klasser
- **Afleveringsopgave: Computerspil 2**
  - Test af de klasser, som I har implementeret i den første delaflevering

Træning i mundtlig præsentation

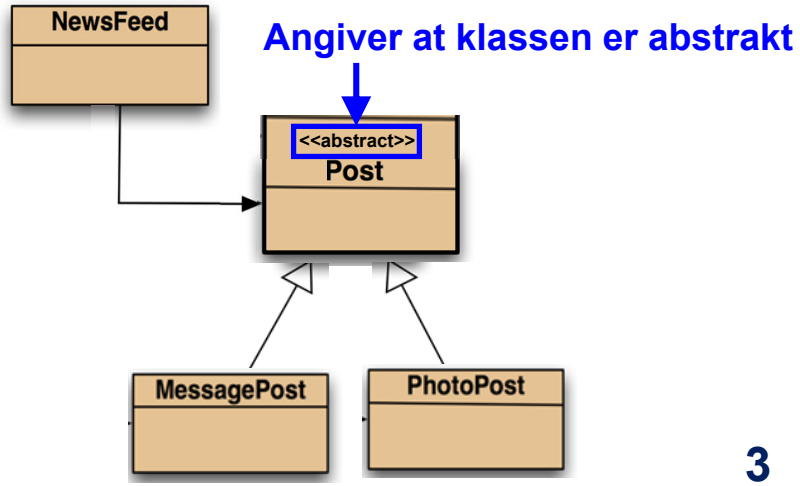
- **Deltag i træningen i mundtlig præsentation**
  - Den er uhyre vigtig for jeres succes ved mundtlig eksamen
  - Eneste gang under jeres studier, hvor I får systematisk træning heri
  - Træning gør mester – de timer I bruger på det, er godt givet ud
  - Se videoerne om den "perfekte" eksamenspræstation og hør jeres medstuderendes præsentationer – det lærer I også af
  - Hvis I er nervøse og usikre omkring eksamen og det at skulle lave præsentationer for andre, har I netop brug for træning
- **Det er vigtigt for it-folk at kunne præsentere tekniske problemstillinger for fagfæller og lægfolk**
  - Det er en essentiel del af vores faglige kompetencer, og I kommer alle til at gøre det i jeres daglige arbejde

Abstrakte klasser og interfaces

- **En abstrakt klasse er en klasse, som man ikke kan lave instanser af**
  - **new** operatoren kan ikke anvendes på klassen
- **I vores Newsfeed system vil det være oplagt at erklære Post til at være en abstrakt klasse**
  - Det signalerer, at vi ikke vil lave instanser af Post, men kun instanser af dens subklasser MessagePost og PhotoPost
  - Formålet med Post er udelukkende at samle de ting, der er fælles for MessagePost og PhotoPost, og på den måde undgå kodedublering og opnå større læsbarhed

Angiver at klassen er abstrakt
(rækkefølgen af public og
abstract er uden betydning)

```
public abstract class Post {
   ...
   ...
}
```

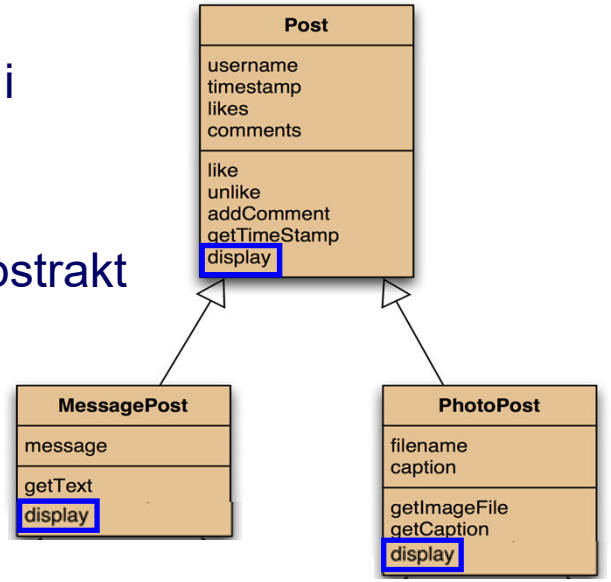


Abstrakte metoder

- **En abstrakt klasse kan indeholde abstrakte metoder**
  - En abstrakt metode er en metode, der ikke er implementeret i klassen
  - Vi angiver kun metodens **hoved**, mens kroppen udelades

- **I vores Newsfeed system har vi en display metode i hver af klasserne Post, MessagePost og PhotoPost**
  - Lad os for et øjeblik antage, at display metoden i Post klassen ikke har noget fornuftigt at lave, f.eks. fordi Post klassen ingen feltvariabler har
  - Så kan vi erklære display metoden til at være abstrakt

```
public abstract void display();
```



- **Betyder at implementationen af metoden overlades til subklasserne**
  - Hvis en subklasse ikke implementerer metoden, skal subklassen selv være abstrakt
  - Det sikrer, at alle konkrete subklasser har en implementation af metoden

Interfaces

- **Et interface ligner en abstrakt klasse, hvor alle metoder er abstrakte**
  - Derudover er der hverken konstruktører eller feltvariabler
- **De abstrakte metoder implementeres i de klasser, der implementerer interfacet**
- **Eksempel: Comparable**
  - For hver metode angives hovedet, mens kroppen udelades
  - Alle metoder er public og abstract (hvorfor dette ikke angives)
  - Comparable har kun én metode, compareTo

Angiver at Comparable er et interface

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

Angiver, at klassen inplementerer interfacet

```
public class Person implements Comparable<Person> {
  ...
  public int compareTo(Person p) {
    ...
  }
  ...
}
```

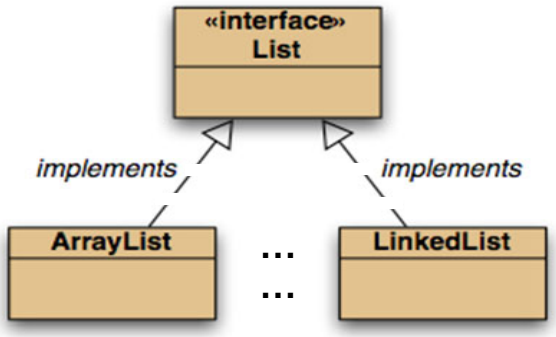
compareTo metoden implementeres i de klasser, der implementerer Comparable interfacet

- Interfaces kan nedarve fra hinanden på samme måde som klasser
- Det betyder, at vi taler om subinterfaces og superinterfaces

List interfacet

- **List interfacet beskriver en sekvens af elementer. Det har mange forskellige implementationer, bl.a.**
  - ArrayList, som opbevarer elementerne i et array
  - LinkedList, som opbevarer elementer i en dobbeltkædet liste, hvor hvert element har pegepinde til det foregående og det efterfølgende element

- **List interfacet gør det let at skifte mellem de forskelige implementationer**
  - Hvis man har erklæret alle sine variable til at være af typen **List** (i stedet for ArrayList eller LinkedList), er det eneste, der skal ændres, det sted hvor listen instantieres
  - Her skal man udskifte **new ArrayList<>()** med **new LinkedList<>()** (eller omvendt)



Set og Map interfacet

- **Set interfacet beskriver en mængde. Det har mange forskellige implementationer, bl.a.**
  - HashSet, som opbevarer mængden i en hashtabel, hvor nøglerne er elementernes hashkoder
  - LinkedHashSet, som opbevarer mængden i en hashtabel og derudover har en dobbeltkædet liste, der bestemmer den rækkefølge, som de besøges i, når man bruger en iterator (eller en for-each løkke)
  - TreeSet, som opbevarer mængden i en træstruktur, hvilket bevirker, at de fleste operationer kan udføres i logaritmisk tid
- **Map interfacet beskriver en afbildning, hvor man ud fra en nøgle kan slå en værdi op. Det har mange forskellige implementationer, bl.a.**
  - HashMap
  - LinkedHashMap
  - TreeMap

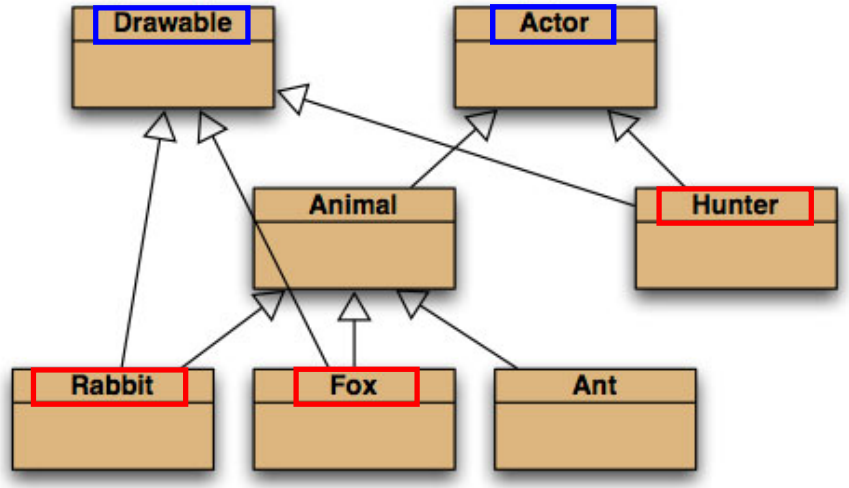
Hvad opnår vi?

- **Fælles for abstrakte klasser og interfaces**
  - De bestemmer en **type**
  - Man kan lave **polymorfe metoder**, som kan bruges på objekter fra
    - alle subklasser af en abstrakt klasse
    - alle de klasser, der implementerer et interface
  - F.eks. kan **reverse** og **shuffle** metoderne i Collections klassen bruges på alle objektsamlinger, der implementerer **List** interfacet, mens **sort** metoden endvidere kræver, at elementtypen i objektsamlingen implementerer **Comparable** interfacet
- **Nedarvning fra abstrakt eller konkret superklasse**
  - Vi arver både noget **implementation** og en **type**
  - Klassen bliver en subtype af superklassen, og dens objekter kan bruges alle de steder superklassens objekter kan bruges
- **Implementation af interface**
  - Vi arver **kun typen** (der er jo ingen implementation at arve)
  - Klassen bliver en subtype af interfacet, og dens objekter kan bruges alle de steder, hvor der kræves et objekt af interface typen

Abstrakt klasse eller interface?

- **Hvorfor har Java både abstrakte klasser og interfaces?**
  - En abstrakt klasse kan indeholde implementation (og feltvariabler) – det kan et interface ikke
  - En klasse kan implementere flere interfaces – men kun være (direkte) subklasse af én anden klasse

- **Multipel nedarvning**
  - De tre **røde** klasser nedarver fra **begge blå** klasser
  - Dette kan kun lade sig gøre, hvis **mindst en** af de **blå** er et interface



- **Hvor meget er implementeret?**
  - Almindelig konkret klasse: **alt** er implementeret (100%)
  - Interface: **ingen** implementation (0%)
  - Abstrakt klasse: **delvis** implementation (0-100%)

Interfaces i Java 8 og fremad

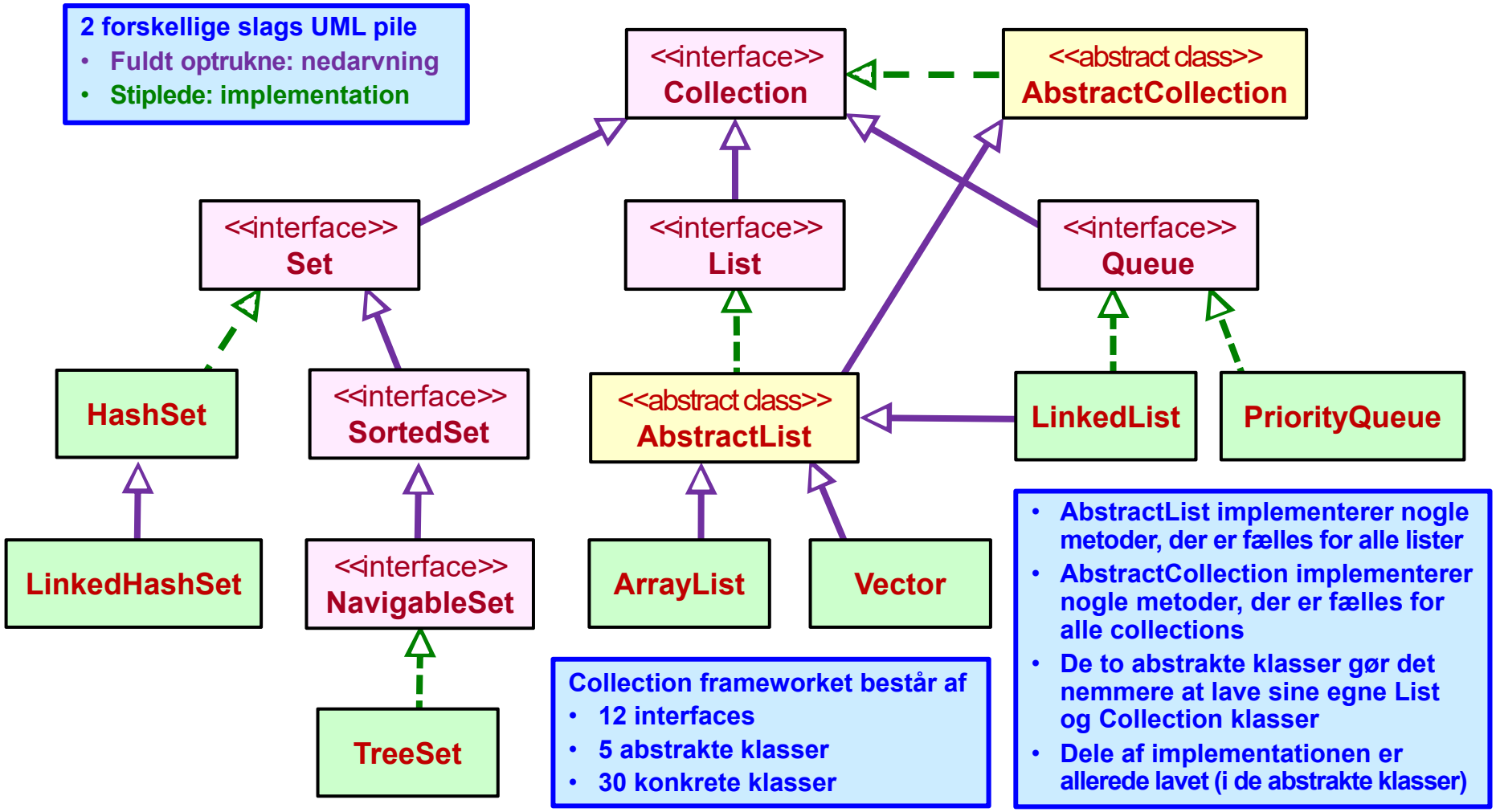
- **Virkeligheden er (desværre) ikke helt så pæn og simpel som beskrevet på de foregående slides**
  - Fra og med Java 8 kan et interface indeholde implementation i form af **klassemetoder** og såkaldte **default metoder**
- **Implementationen af disse metoder nedarves til de klasser, der implementerer interfacet**
  - Default metoder er primært indført for at kunne tilføje nye metoder til et eksisterende interface uden at genere de klasser, der allerede implementerer det
  - Da et interface ikke har feltvariabler og konstruktører, er det begrænset, hvad man kan gøre i en default metode (der er ingen tilstand at operere på)
- **Eksempel fra funktionel sortering**

Klassemetode i Comparator interfacet (tager en lambda som parameter og returnerer et Comparator objekt)

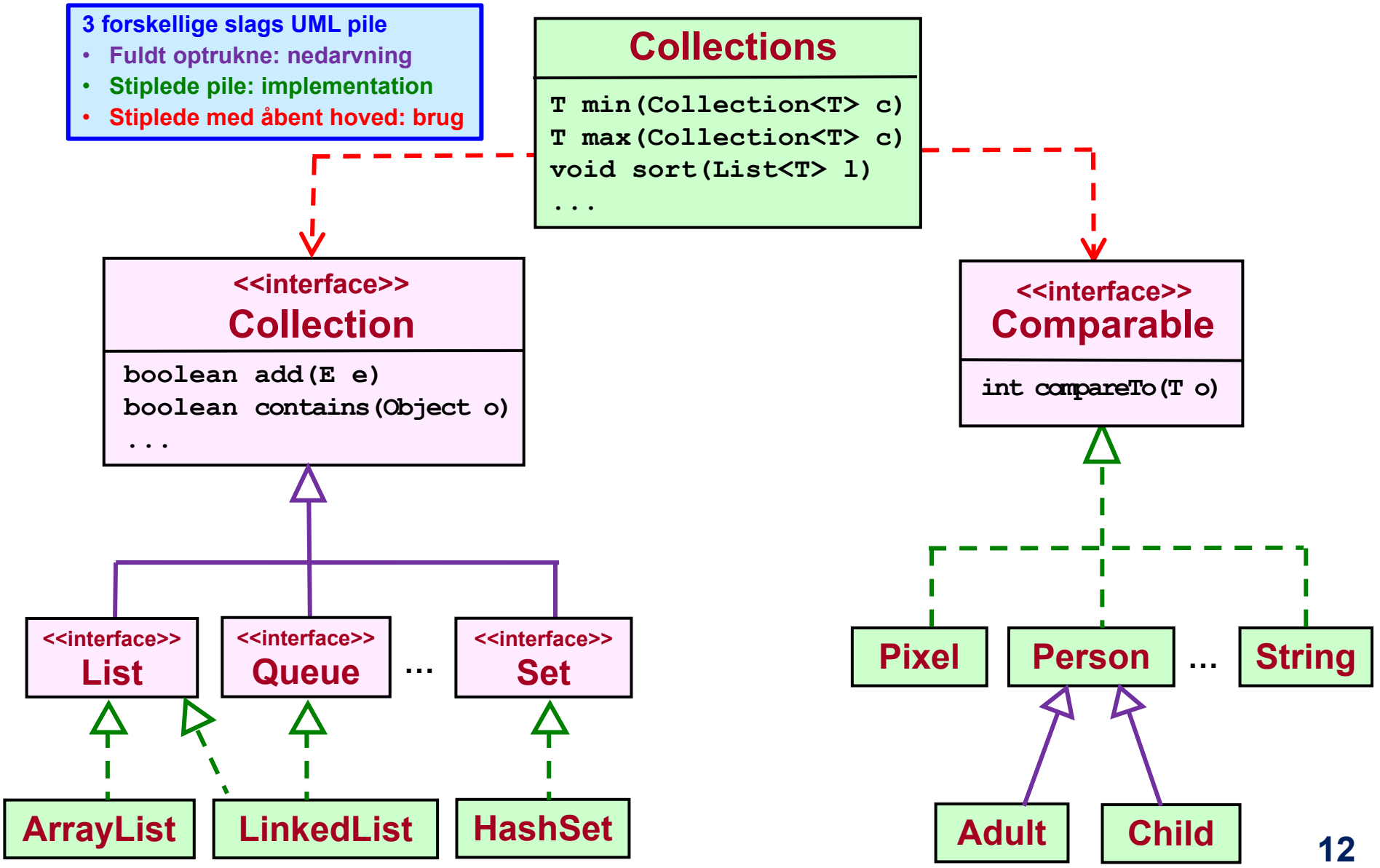
```
Collections.sort(persons, Comparator.comparing((Person p) -> p.getName())
                                              .thenComparing(p -> p.getAge()));
```

Default metode i Comparator interfacet (kaldes på det objekt, som comparing metoden returnerer, og returnerer et Comparator objekt)

Collection frameworket (udsnit)



Brug af Collection og Comparable



Funktionelle interfaces

- **Et funktionelt interface har én enkelt abstract metode**
  - De steder, hvor man skal bruge et objekt, hvis type er et funktionelt interface, kan man i stedet bruge en lambda

Comparable har også kun én abstract metode, men er ikke et funktionelt interface

- **Man kan erklære variabler, hvis type er et funktionelt interface**
  - Dermed bliver det muligt at **assigne lambda'er** til variabler, og dermed bruge dem forskellige steder i programmet, f.eks. som parameterværdier
- **java.util.function definerer en række funktionelle interfaces, bl.a.**
  - **Predicate** bruges til lambda'er, der returnerer en boolsk værdi
  - **BinaryOperator** bruges til lambda'er, hvor de to parametre og returværdien er af samme type (f.eks. String x String → String)
  - **UnaryOperator** bruges til lambda'er, hvor parameteren og returværdien er af samme type (f.eks. String → String)
  - **Function** bruges til lambda'er, hvor parameteren og returværdien er af forskellig type (f.eks. String → Integer)
  - **Consumer** bruges til lambda'er der ikke returnerer en værdi, men typisk har en sideeffekt

Metoder i Stream interfacet

- **I Stream interfacet er parametrene til filter, map og reduce metoderne funktionelle interfaces**
  - Det er derfor vi kan bruge lambda'er, når vi kalder de tre metoder
  - Parameteren til **filter** metoden er et **Predicate**
  - Parameteren til **map** metoden er en **Function**
  - Anden parameteren til **reduce** metoden er en **BinaryOperator**

```
public int getCount(String animal) {
  return sightings.stream()
                  .filter( s -> s.getAnimal().equals(animal) )
                  .map( s -> s.getCount() )
                  .reduce( 0 , (result, elem) -> result + elem );
}
```

Funktionel sortering

- **I funktionel sortering er parametrene til comparing og thenComparing metoderne et funktionelt interface af typen Function**
  - Det er derfor vi kan bruge lambda'er, når vi kalder de to metoder
  - Parametrene har **typebegrænsninger**, der sikrer, at de afbilder over i en type, som implementerer Comparable interfacet (eller har en supertype, der gør det)
  - Det betyder, at typen har en naturlig ordning
  - Det er denne naturlige ordning, der anvendes til sorteringen i de Comparator objekter, som de to metoder returnerer.

```
public void printPersons() {
  Collections.sort(persons, Comparator.comparing((Person p) ->p.getAge())
                                          .thenComparing (p -> p.getName()));
  persons.forEach(p -> System.out.println(p));
}
```

Billedredigering

- **Operationerne i billedredigeringsopgaven fra uge 4 kan implementeres ved hjælp af funktionelle interfaces**
  - I de simple billedoperationer (såsom brighten, darken, invert og noise) kan en pixel's nye gråtone beregnes ud fra den gamle ved hjælp af en **lambda**

Funktionelt interface (Integer → Integer)

```
public Image simpleFilter(UnaryOperator<Integer> modification) {
  for(int x = 0; x < image.getWidth(); x++) {
    for(int y = 0; y < image.getHeight(); y++) {
      int oldValue = image.getPixel(x,y).getValue();
      image.getPixel(x,y).setValue(modification.apply(oldValue));
    }
  }
  image.updateCanvas();
  return image;
}
```

Beregning af den nye værdi ved hjælp af apply metoden i det funktionelle interface UnaryOperator, dvs. den lambda, der bruges som parameterværdi i kaldet

```
public Image brighten(int amount) {
  return simpleFilter( v -> v + amount );
}
```

```
public Image invert() {
  return simpleFilter( v -> 255 - v );
}
```

Ved kald af simpleFilter er parameterværdien en lambda, der automatisk giver os et objekt af typen UnaryOperator

Billedredigering (fortsat)

- **I de operationer, der spejler, roterer og resizer, bliver gråtonen for en pixel kopieret fra en anden pixel**
  - Beregningen af denne pixel's koordinater kan beskrives ved hjælp af to lambda'er

Funktionel interface (Integer x Integer → Integer)

```
public Image complexFilter(int width, int height,
       BinaryOperator<Integer> xPos, BinaryOperator<Integer> yPos) {
  Image newImage = new Image(width, height, image.getTitle(), false);
  for(int x = 0; x < width; x++) {
    for(int y = 0; y < height; y++) {
      Pixel p = image.getPixel( xPos.apply(x,y), yPos.apply(x,y) );
      newImage.getPixel(x,y).setValue(p.getValue());
    }
  }
  image = newImage;
  image.updateCanvas();
  return image;
}
```

Positionen, hvor gråtoneværdien skal hentes

```
public Image mirror() {
  int w = image.getWidth();
  int h = image.getHeight();
  return complexFilter(w, h, (i,j) -> w-i-1, (i,j) -> j);
}
```

Den nye gråtoneværdi for (i,j) hentes i (w-i-1, j)

Pause

I næste forelæsning vil vi se, at funktionelle interfaces også er særdeles nyttige i forbindelse med grafiske brugergrænseflader

```
public Image rotate() {
  int w = image.getWidth();
  int h = image.getHeight();
  return complexFilter(h, w, (i,j) -> j, (i,j) -> h-i-1);
}
```

Den nye gråtoneværdi for (i,j) hentes i (j, h-i-1)

Wildcards og typebegrænsninger

- **Klasser med typeparametre (fx. ArrayList<E>) kaldes generiske klasser**
  - Når man, for metoderne i en generisk klasse, skal beskrive parametrenes type og returtypen, bruges der en speciel notation indeholdende **wildcards** (jokere) og **typebegrænsninger**
  - Det kan være nyttigt at kende og forstå de hyppigst forekomne wildcards og typebegrænsninger, f.eks. når man slår op i Javas API
  - Dem vil vi nu illustrere ved hjælp af nogle eksempler

Eksempler fra ArrayList<E> klassen

boolean add(E e)

Tilføjer elementet e til enden af listen

- Nem at forstå (ingen wildcards eller typebegrænsninger)

boolean addAll(Collection<? extends E> c)

Tilføjer alle elementerne fra c

- Parameteren skal implementere Collection interfacet (dvs. være en objektsamling)
- Elementtypen i objektsamlingen skal være en **subtype** af E (inklusiv E selv)
- Sikrer at det er lovligt at tilføje elementerne i c til vores arrayliste

boolean removeIf(Predicate<? super E> filter)

Fjerner alle de elementer der opfylder filter

- Parameteren skal være et prædikat (dvs. en boolsk lambda)
- Prædikatets parameter skal være en **supertype** af E (inklusiv E selv)
- Sikrer at prædikatet kan bruges på elementer af typen E

Eksempler fra Collections

void shuffle(List<?> list)

Permuterer listens elementer

- Parameteren skal være af en type, der implementere List interfacet
- Elementtypen i listen kan være vilkårlig
- Metoden kan bruges på alle lister, men f.eks. ikke på et Set eller Queue object (med mindre det også er en liste)

int frequency(Collection<?> c, Object o)

Returnerer antallet af forekomster af objektet o i objektsamlingen c

- Første parameter skal være af en type, der implementerer Collection interfacet
- Anden parameter er af vilkårlig type (alle klasser er subklasser af Object)
- Lovligt at spørge om frekvensen af objekter, der slet ikke kan forekomme i listen (de har trivielt frekvensen 0)

Eksempler fra Collections (fortsat)

<T> void copy(List<? super T> dest, List<? extends T> src)

Kopierer alle elementer fra source listen (src) til destination listen (dest)

- Begge parametre skal være af typer, der implementerer List interfacet
- Der skal eksistere en type T, således at elementtypen i dest er en supertype af T, og elementtypen i src er en subtype af T
- Dvs. at elementtypen i src er en subtype af elementtypen i dest (eller lig med denne)
- Sikrer at det er lovligt at indsætte elementerne fra src i dest

Foran returtypen listes de typer, der bruges til typebegrænsninger

- Det er dog ikke nødvendigt at liste de typer, der er type parametre for den generiske type, som vi er i færd med at definere
- F.eks. behøver vi ikke at liste E i
  - **boolean addAll(Collection<? extends E> c)**

  idet E allerede er introduceret som en type parameter i ArrayList<E>

Eksempler fra Collections (sort metoderne)

<T> void sort(List<T> list, Comparator<? super T> c)

Sorterer listen ved hjælp af en comparator

- Første parameter skal være en liste med en vilkårlig elementtype T
- Anden parameter skal være af en type, der implementere Comparator interfacet for T (eller for en supertype af T)
- Sikrer at T (eller en supertype af T) stiller en compare metode til rådighed for sorteringen

<T extends Comparable<? super T>> void sort(List<T> list)

Sorterer listen ved hjælp af den naturlige ordning

- Parameteren skal være en liste
- Elementtypen T skal implementerer Comparable interfacet (eller have en supertype, der gør det)
- Sikrer at T (eller en supertype af T) stiller en naturlig ordning til rådighed for sorteringen

Eksempel fra foxes and rabbits projektet

- **Vi vil lave en afbildning (Map), hvor nøglerne er dyrenes klasser og værdierne de farver, hvormed dyrene vises på spillepladen**
  - Klasserne repræsenteres ved hjælp af klassen Class<T>, der indeholder **ét objekt** for hver klasse i et kørende program
  - Rabbit repræsenteres af objektet **Rabbit.class** (af typen Class<Rabbit>)
  - Fox repræsenteres af objektet **Fox.class** (af typen Class<Fox>)

- **Nu kan vi definere vores afbildning på følgende måde**

Wildcardet ? angiver, at vi på dette sted kan bruge en vilkårlig klasse, f.eks. Rabbit og Fox

```
private Map<Class<?>, Color> colors;
......
colors.put(Rabbit.class,Color.ORANGE);
colors.put(Fox.class,Color.BLUE);
```

Nøgler (objekter i typen Class<?>)

Værdier (konstanter i Color klassen fra Javas API)

- **Yderligere info om wildcards og typebegrænsninger:**
  https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html Link

Opsummering af wildcards og typebegrænsninger

- **Wildcards og typebegrænsninger er komplekse**
  - Det er ikke noget vi vil høre jer I til eksamen
  - Men det kan være en stor fordel at kende lidt til dem, idet I så lettere kan forstå de typer, der angives i Javas API

Hjemmeopgave for de "barske"

- **Find nedenstående metoder i API'en og se, om I kan forstå deres wildcards og typebegrænsninger**
  - **filter**, **map**, **mapToInt** og **reduce** i Stream klassen
  - **comparing** og **thenComparing** i Comparator interfacet
- **Bemærk at der er flere metoder, der hedder det samme – men har forskellige parmetre (dvs. forskellig signatur)**
  - Start med at finde ud af, hvilken af metoderne det er, som I har brugt

Afleveringsopgave: Computerspil 2

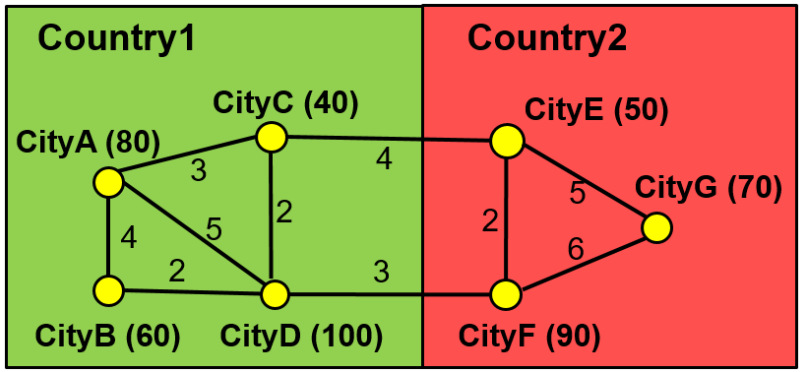
- **I den anden delaflevering skal I bruge de ting, som I har lært om regression tests, på de fire klasser, som I har implementeret i den første delaflevering**
  - Lav velvalgte **regression tests** for de fire klasser
    - **Positive tests** skal afprøve programmets normale opførsel – herunder om det fungerer korrekt omkring forskellige grænseværdier
    - **Negative tests** skal afprøve om programmet kan håndtere uventede situationer (f.eks. de specialtilfælde, der er beskrevet i opgaveformuleringen)
    - I behøver ikke lave testmetoder for
      - trivielle accessor og mutator metoder, der blot returnerer/ændrer en enkelt feltvariabel uden at gøre andet
      - compareTo, equals og hashCode metoderne
- **Herudover skal I rette de fejl og mangler, som instruktoren har påpeget i jeres første delaflevering**

Generelt om testmetoder

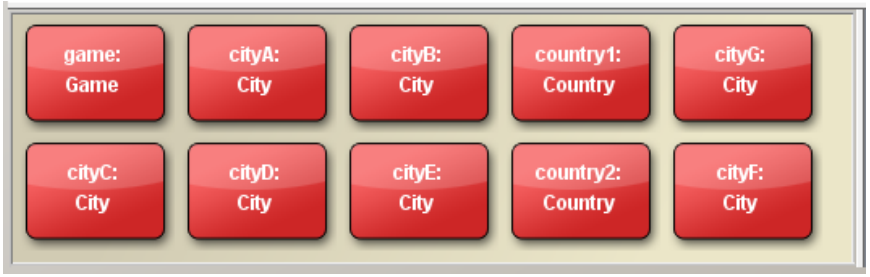
- **Der laves normalt én testmetode for hver (ikke triviel) konstruktør/metode**
  - En sådan testmetode kan teste mange forskellige ting (via forskellige assertions)
  - I nogle tilfælde kan det dog være hensigtsmæssigt at lave to testmetoder, for samme metode (f.eks. hvis man vil tjekke, hvordan metoden opfører sig i et grænsetilfælde, hvor den kaldes med en speciel værdi)
- **Navngivning af testmetoder**
  - Testmetoden for konstruktøren kaldes constructor
  - Testmetoden for en metode har samme navn, som den metode den tester
  - Testmetoden for toString kaldes dog testToString (idet Java ellers tror, at vi forsøger at overskrive toString metoden i Object klassen)
- **I behøver ikke at teste compareTo, equals og hashCode metoderne**
  - Det bør man normalt gøre, men da det kan være lidt besværligt og tidskrævende, har vi besluttet at lade jer slippe for det

Brug af Test Fixture

- **Når man skal teste metoderne i computerspillet har man behov for skabe nogle byer, lande og veje**
  - Til dette formål har vi lavet en **Test Fixture** (fast opsætning) som realiserer nedenstående netværk af lande, byer og veje



  - Test Fixturen kan kopieres til "Object benchen", som så får dette udseende



- 2 Country objekter
- 7 City objekter
- 1 Game objekt

  - Opgaveformuleringen forklarer, hvordan Test Fixture'n hentes og anvendes

Test of Road klassen

TestFixture for Road klassen

- Fra den generelle Test Fixture kan I kopiere import sætningerne, feltvariablerne og setUp metoden
- Tingene i de røde bokse skal I selv tilføje
- For overskuelighedens skyld fjernes de dele af den generelle Test Fixture, som I ikke bruger
- Her bruges kun det første land og de første fire byer (og ikke nogen af vejene, idet vi laver vores egne)

- Tagget @BeforeEach angiver, at setUp metoden udføres før hver testmetode (dette etablerer test fixturen)
- Analogt angiver tagget @AfterEach, at en metode udføres efter test fixturen – for at rydde op (men det får I ikke brug for i denne opgave)

Vi laver to Road objekter

- Den første vej går fra cityA til cityB og har længden 4
- Den anden vej går fra cityC til cityD og har længden 2

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
public class RoadTest {
  private Game game;
  private Country country1;
  private City cityA, cityB, cityC, cityD;
  private Road road1, road2;
  @BeforeEach
  public void setUp() {
    // Create game object
    game = new Game(0);

    // Create country
    country1 = new Country("Country 1");
    country1.setGame(game);

    // Create cities
    cityA = new City("City A", 80, country1);
    cityB = new City("City B", 60, country1);
    cityC = new City("City C", 40, country1);
    cityD = new City("City D", 100, country1);

    // Connect cities to countries
    country1.addCity(cityA);
    country1.addCity(cityB);
    country1.addCity(cityC);
    country1.addCity(cityD);

    // Create roads
    road1 = new Road(cityA, cityB, 4);
    road2 = new Road(cityC, cityD, 2);
  }
  ...
```

Test of Road klassen (fortsat)

Testmetode for konstruktøren

- Vi tjekker, at feltvariablerne initialiseres korrekt
- Brug det forventede som første parameter og metodekaldet som anden parameter
- Tjekker også (implicit) de tre simple accessormetoder

```
@Test
public void constructor() {
  // Første vej går fra CityA til CityB og har længde 4
  assertEquals(cityA, road1.getFrom());
  assertEquals(cityB, road1.getTo());
  assertEquals(4, road1.getLength());

  // Anden vej går fra CityC til CityD og har længde 2
  assertEquals(...., road2.getFrom());
  assertEquals(...., road2.getTo());
  assertEquals(.., road2.getLength());
}
```

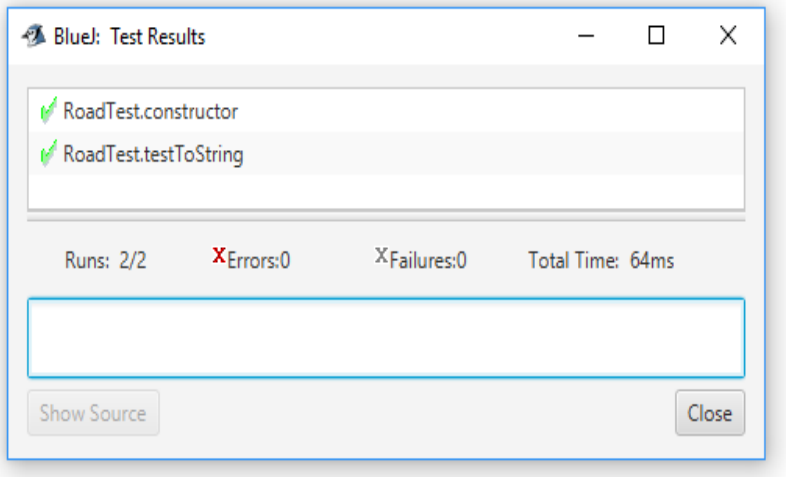
Testmetode for toString

- Vi tjekker at returværdien er korrekt

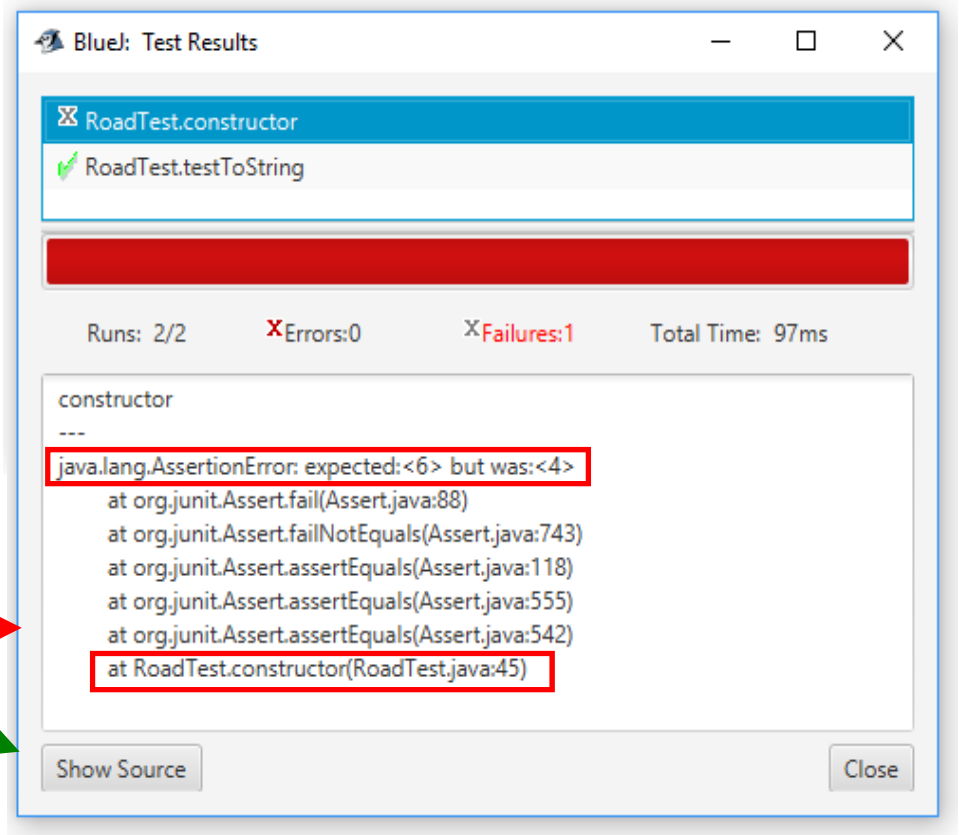
```
@Test
public void testToString() {
  assertEquals("City A (80) -> City B (60) : 4", road1.toString());
  assertEquals("..............................", road2.toString());
}
```

Test of Road klassen (fortsat)

Når man vælger Test All får man forhåbentlig nedenstående resultat



Hvis et eller flere af de grønne flueben erstattes af et sort kryds er der noget galt



Når man vælger den fejlramte metode kan man i det nederste felt se, hvad der er galt

- **I linje 45 i testklassen var der en assertion, der returnere 4 i stedet for 6**
- **Hvis man trykker på Show Source hopper man til den assertion, der fejlede**

```
40
41      @Test
42      public void constructor() {
43          assertEquals(cityA, road1.getFrom());
44          assertEquals(cityB, road1.getTo());
45          assertEquals(6, road1.getLength());
46
```

Bemærk at fejlen både kan ligge i den metode, der bliver testet, og i testmetoden

Test af Position klassen

TestFixture for Position klassen

- Som før kan I koppiere det meste fra den generelle Test Fixture
- Tingene i de to røde bokse skal I selv tilføje
- Fjern de ting i Test Fixturen, som I ikke bruger

Vi laver to Position objekter

- I det første står spilleren i cityA og er på vej mod cityB, som kan nås i 4 skridt
- I det andet står spilleren i cityC og er på vej mod cityD, som kan nås i 2 skridt

```
import ...

public class PositionTest {
  private Game game;
  private Country country1;
  private City cityA, cityB, cityC, cityD;
  private Position pos1, pos2;
  @BeforeEach
  public void setUp() {
    // Create game object
    game = new Game(0);

    // Create country
    country1 = new Country("Country 1");
    country1.setGame(game);

    // Create cities
    cityA = new City("City A", 80, country1);
    ...

    // Create positions
    pos1 = new Position(cityA, cityB, 4);
    pos2 = new Position(cityC, cityD, 2);
  }
  ...
```

Test af Position klassen (fortsat)

Testmetode for konstruktøren

- Vi tjekker, at de fire feltvariabler initialiseres korrekt
- Herved tjekker vi også implicit de fire accessor metoder

```
@Test
public void constructor() {
  assertEquals(...., pos1.getFrom());
  assertEquals(...., pos1.getTo());
  assertEquals(.., pos1.getDistance());
  assertEquals(.., pos1.getTotal());

  assertEquals(...., pos2.getFrom());
  assertEquals(...., pos2.getTo());
  assertEquals(.., pos2.getDistance());
  assertEquals(.., pos2.getTotal());
}
```

Testmetode for move metoden

- Vi kalder metoden nogle gange
- Efter hvert kald tester vi, at returværdien og distance er som forventet

```
@Test
public void move() {
  assertEquals(true, pos2.move());;
  assertEquals(1, pos2.getDistance());

  assertEquals(..., pos2.move());;
  assertEquals(.., pos2.getDistance());

  assertEquals(..., pos2.move());;
  assertEquals(.., pos2.getDistance());
}
```

Test af Position klassen (fortsat)

```
@Test
public void turnAround() {
  pos1.move();
  pos1.turnAround();
  assertEquals(...., pos1.getFrom());
  assertEquals(...., pos1.getTo());
  assertEquals(.., pos1.getDistance());

  pos1.turnAround();
  assertEquals(...., pos1.getFrom());
  assertEquals(...., pos1.getTo());
  assertEquals(.., pos1.getDistance());

  pos2.turnAround();
  assertEquals(...., ........);
  assertEquals(...., ........);
  assertEquals(.., ........);

  pos2.turnAround();
  assertEquals(....
  assertEquals(....
  assertEquals(..,
}
```

Testmetoderne for turnAround, hasArrived og toString

- Vi kalder metoderne nogle gange
- Efter hvert kald tester vi, om alt er som forventet

```
@Test
public void hasArrived() {
  assertEquals(...., pos2.hasArrived());

  pos2.move();
  assertEquals(...., pos2.hasArrived());

  pos2.move();
  assertEquals(...., pos2.hasArrived());

  pos2.move();
  assertEquals(...., pos2.hasArrived());
}
```

```
@Test
public void testToString() {
  assertEquals("City A (80) -> City B (60) : 4/4", pos1.toString());
  assertEquals("............................", pos2.toString());

  pos2.move();
  assertEquals("............................", pos2.toString());

  pos2.move();
  assertEquals("............................", pos2.toString());
}
```

Test af metoder med tilfældige værdier

- **Når man skal teste arrive metoden i City klassen kan det være nyttigt at resette den seed værdi som Random objektet bruger**
  - På den måde kan man finde ud af, hvad bonus metoden returnerer, når den kaldes inde fra arrive

```
@Test
public void arrive(){
   game.getRandom().setSeed(0);                    // Set seed
   int bonus = country1.bonus(80);                 // Remember bonus
   game.getRandom().setSeed(0);                    // Reset seed
   assertEquals(bonus, cityA.arrive());            // Same bonus
   assertEquals(....., cityA.getValue());
}
```

Test at returværdi og værdien af value er korrekt

Man kan også proppe det hele ind i en for løkke og dermed teste mange forskellige seed værdier

```
@Test
public void arrive(){
   for(int seed = 0; seed < 1000; seed++) {   // Try different seeds
      game.getRandom().setSeed(seed);                 // Set seed
      int bonus = country1.bonus(80);                 // Remember bonus
      game.getRandom().setSeed(seed);                 // Reset seed
      assertEquals(bonus, cityA.arrive());            // Same bonus
      assertEquals(....., cityA.getValue());
      cityA.reset();
   }
```

Som ovenfor, bortset fra, at seed nu sættes til seed

Husk at resette byen mellem de enkelte tests

Lav også en testmetode, der tester, hvad der sker, hvis byen har værdien 0

Eksempler på testmetoder for Country

```
@Test
public void reset() {
  cityA.arrive(); cityA.arrive(); cityA.arrive();
  cityE.arrive(); cityE.arrive(); cityE.arrive();
  int valueE = cityE.getValue();      // Remember value of cityE
  country1.reset();
  assertEquals(.., cityA.getValue());       // cityA is reset
  assertEquals(valueE, cityE.getValue());   // cityE is unchanged
}
```

Ved at kalde arrive 3 gange på hver by, er I rimeligt sikre på, at byernes værdi er ændret

```
@Test
public void bonus() {
  for(int seed = 0; seed < 100; seed++) {     // Try 100 different seeds
    game.getRandom().setSeed(seed);
    ...
    for(int i = 0; i < 100000; i++) {         // Call method 100.000 times
      int bonus = country1.bonus(80);
      Test at værdien ligger i det korrekte interval
      ...
    }
    Test at middelværdien er tæt på det forventede
    Test at alle de mulige værdier returneres
  }
}
```

Disse tre ting kan testes på samme måde, som I testede roll metoden i Die klassen (i Raflebæger 4)

Lav også en testmetode, der tester, hvad der sker, hvis bonus kaldes med 1 og 0

Negative tests og dokumentation

- **Husk de negative test, såsom**
  - **getCity("city")** hvor **"city"** ikke findes i landet
  - **getRoads(c)** hvor c ikke findes i landet
  - I skal teste alle de specialtilfælde, som er beskrevet i opgaveformuleringen
- **Dokumentationen for jeres testklasser kan holdes på et minimum**
  - Testklassens navn fortæller, hvilken klassen den tester
  - Testmetodens navn fortæller, hvilken metode den tester
  - Testmetoder har ingen parametre og returnerer intet, så @param og @return tags giver ikke mening
- **I kan derfor nøjes med at indsætte**
  - @author og @version tags
  - Forklarende // kommentarer i kompleks kode (som illustreret på mine slides)

Testserveren

- **Testserveren skal også anvendes for Computerspil 2**
  - Serveren tester at jeres regression tests er fornuftige
  - Regression tests kan imidlertid laves på mange forskellige måder, og det er ikke altid entydigt, hvad der bør testes
- **Som illustreret i videoen om Regression tests, afprøver testserveren derfor, at jeres regression tests**
  - **ikke** finder fejl i et korrekt projekt
  - finder de **fleste** fejl i nogle forkerte projekter
- **Brug testserveren med omtanke**
  - Som i Computerspil 1, kan I teste, hver enkelt opgave, så snart I er færdige med den
  - Når I får en fejlrapport, bør I rette alle de fejl, der rapporteres og kontrollere, at rettelserne er korrekte, **før** I atter forsøger at køre testserveren
- **Testserveren understøtter ikke Hamcrest og Jupiter**
  - Som nogle af jer måske kender fra andre kurser

Opsummering

- **Abstrakte klasser og interfaces**
  - En abstrakt klasse er en klasse, som man ikke kan lave instanser af
  - En abstrakt klasse kan indeholde abstrakte metoder, hvor kun hovedet er angivet, mens implementationen (kroppen) mangler
  - I et interface er alle metoder abstrakte (men fra og med Java 8 kan der være klassemetoder og default metoder)
- **Funktionelle interfaces**
  - Har kun én enkelt abstract metode
  - De steder, hvor man skal bruge et objekt, hvis type er et funktionelt interface, kan man i stedet bruge en lambda
- **Wildcards (jokere)**
  - Gør det muligt at beskrive komplicerede typebegrænsninger i forbindelse med generiske klasser
- **Afleveringsopgave: Computerspil 2**
  - Test af de klasser, som I har implementeret i den første delaflevering

Det var alt for nu….. … spørgsmål