CHALMERS

GÖTEBORGS UNIVERSITET

Institutionen för data- och informationsteknik

TENTAMEN

KURSNAMN	**Objektorienterad programutveckling, fk 7.5p**
PROGRAM	**TKIEK-2, TKTFY-3, TKTEM-3** **2018/2019, lp 2**
KURSBETECKNING	**TDA550**
EXAMINATOR	**Wolfgang Ahrendt**
TID FÖR TENTAMEN	**Fredag 2020–01–17, 14:00**
HJÄLPMEDEL	**Java API-kompendium** **(delas ut av skrivningsvakten)**
ANSVARIG LÄRARE	**Pelle Evensen** **Tel. 0708–482879** **Besöker tentamen ca kl. 14:30 samt ca 17:00.**
DATUM FÖR ANSLAG	**Senast 2020–02–17.** **Datum för granskning meddelas på kursens hemsida.**
ÖVRIG INFORMATION	**Betygsgränser: 3 – 24p, 4 – 36p, 5 – 48p.** **(max 60p)**

TENTAMEN

Objektorienterad programvaruutveckling, fk

- Uppgifterna är inte ordnade efter svårighetsgrad.
- Börja varje hel uppgift på ett nytt blad. Skriv inte i tesen.
- Ordna bladen i uppgiftsordning.
- Skriv din tentamenskod **på varje blad** (så att vi inte slarvar bort dem).
- Skriv rent dina svar. Oläsliga svar **rättas ej!**
- Använd gärna färgade pennor om det ökar tydligheten men **rött är reserverat för dem som rättar – använd ej!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
  Det räcker med enbart relevanta kodavsnitt, övrig kod ersätts med ... (aldrig `import`, hjälpklasser såsom `Main`, `main`-metod som bara anropar relevant metod, etc. om det inte tydligt framgår i uppgiften att det förväntas).
- Oprecisa eller alltför generella (vaga) svar ger inga poäng. Konkretisera och/eller ge exempel. Det är aldrig någon risk att vara övertydlig!
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. **får inte ändras** om det inte uttryckligen är en del av uppgiften. Fråga i oklara fall!
- I en uppgift som består av flera delar får du använda dig av funktioner, klasser, etc. från tidigare deluppgifter, även om du inte löst dessa.
- Läs igenom tentamenstesen och förbered eventuella frågor.

Lycka till!

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 1 (11p)

Hammingavståndet mellan två sekvenser är antalet element i två strängar/listor/samlingar som är olika. Om *h(x, y)* ger Hammingavståndet mellan två listor *x* och *y* så skulle följande hålla:

```
h({0, 1, 2}, {0, 1, 3}) = 1 (skiljer sig åt i sista positionen)
h({0, 1, 2}, {1, 2, 3}) = 3 (skiljer sig åt i alla positioner)
h({2, 2, 2}, {1, 2, 3}) = 2 (skiljer sig åt i position 1 och 3)
h({0}, {0, 1}) => Otillåtet; listorna har olika längd.
```

Låt vår Hammingavståndsfunktion ha följande typsignatur:

```
public static <T> int hammingDistance(List<T> l1, List<T> l2)
```

(A) Implementera metoden `hammingDistance` med typsignatur som ovan. (2p)

(B) Formulera förvillkoren (formellt m.a.p. `l1` & `l2`) (1p)

(C) Formulera eftervillkor (informellt) (1p)

Implementera följande test (m.h.a. JUnit 5):

(D) Testa att metoden korrekt hanterar data som bryter mot förvillkoren. (1p)

(E) Testa symmetri, alltså att `hammingDistance(a, b) = hammingDistance(b, a)`. (2p)

(F) Testa att metodens utdata är rimliga för olika samlingar som uppfyller förvillkoren. (2p)

För full poäng på (E) och (F) skall minst ett av testfallen involvera samlingar vars data randomiseras fram under körning.

Notera att testen (D)–(F) inte får innehålla något som liknar en implementation av hammingDistance-metoden – det är ju den som ska testas!

Reflektion

(G) Överambitiöse Örjan tycker att typsignaturen nedan hade varit lämpligare då den verkar mer generell. Förklara kort varför Örjan har fel. (2p)

```
// Örjan's "superior" hammingDistance
public static <T> int hammingDistance(Collection<T> c1, Collection<T> c2)
```

Uppgift 2 (8p)

Studera klasserna och gränssnitten nedan.

(A) Rita ett komplett UML-klassdiagram för gränssnitten och klasserna nedan. (2p)

```
public interface IX {
  void doIt();
}
```

```
public interface IY {
  void doOther();
}
```

```
public class X implements IX {
  @Override
  public void doIt() {
   System.out.println("doIt X");
  }

  public void doAnother(final double i) {
    System.out.println("doAnother X " +
      i);
  }
}
```

```
public class Y extends X {
  @Override
  public void doIt() {
    System.out.println("doIt Y");
  }

@Override
  public void doAnother(final int d) {
    System.out.println("doAnother Y " +
      d);
  }
}
```

```
public class Z extends X implements IX {
  public void doYetAnother() {
     System.out.println("doYetanother Z");
  }
}
```

(B) Givet klasserna och gränssnitten i deluppgift (A), ange för fallen 1–6 nedan om koden är korrekt eller ej. Om den inte är korrekt, beskriv varför samt vilken typ av fel det är; kompileringsfel eller körningsfel? Om ett fall är korrekt, vad skrivs ut för just det fallet? (6p)

```
// 1
final Z zx = new X();
zx.doIt();
```

```
// 2
final IX ixy = new Y();
final IY iya = ixy;
ixy.doAnother();
```

```
// 3
final X xy1 = new Y();
xy1.doIt();
```

```
// 4
final IX ixz1 = new Z();
ixz1.doIt();
```

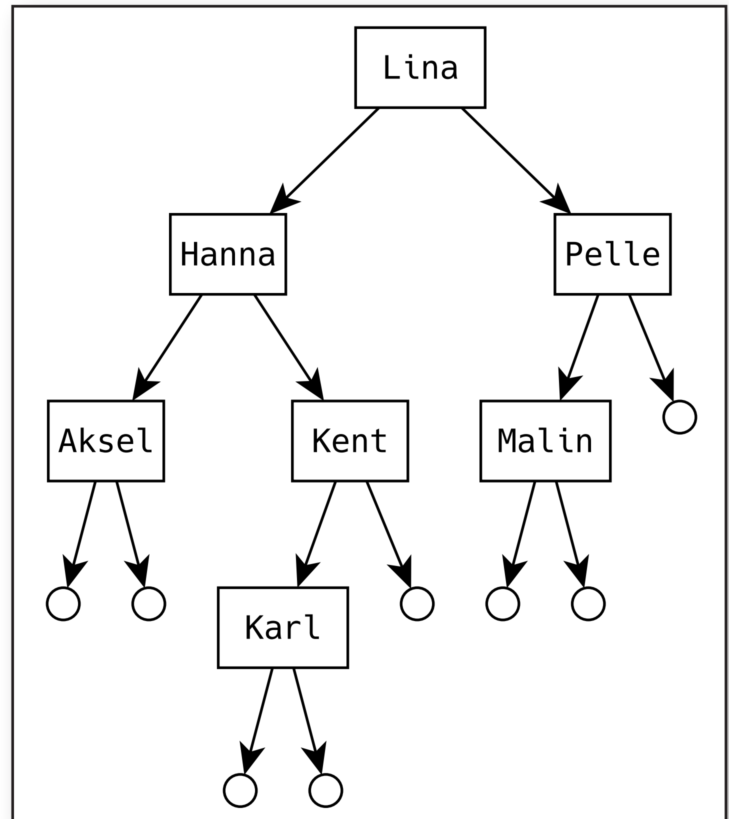
```
// 5
final IX ixz2 = new Z();
final X x = ixz2;
x.doIt();
```

```
// 6
final X xy2 = new Y();
xy2.doAnother(1);
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 3 (10p)

Ett binärt sökträd består av noder och löv. Varje nod pekar till vänster och höger antingen på en annan nod eller `null` (”löv”). Trädet till höger är ett exempel där ”Lina” utgör trädets rot-nod. Alla noder till vänster om ”Lina” är lexikografiskt mindre och alla till höger är lexikografiskt större. Noder är i diagrammet ritade som rektanglar och löv (`null`) som cirklar.



Trädet till höger kan fås genom att man till ett tomt träd lägger till namn i följande ordning (det finns flera ordningar som skulle gett samma träd):
Lina, Hanna, Pelle, Malin, Kent, Karl, Aksel

Ett interface för binära sökträd, som bara kan hantera strängar, kan se ut som följer:

```
public interface StringBinarySearchTree {
   boolean isEmpty();
   int size();

   // Returns true if s is in the tree.
   boolean contains(String s);

   // Returns true if s was not in the tree, false otherwise.
   boolean add(String s);
}
```

(A) Definiera ett komplett interface, `BinarySearchTree<E>`. Det **ska** innehålla samma metoder som interfacet `StringBinarySearchTree`. **Du får ej använda gränssnittet** `Comparator`. Förklara varför du valt/inte valt en viss inskränkning för typparametern `E`. För full poäng på uppgiften krävs att **typparametrar används korrekt**. (2p)
Beskriv samtliga metoder för det nya interfacet.
Följande metodsignaturer påverkas jämfört med `StringBinarySearchTree`:
1. Se om ett visst element finns i trädet (`contains`). (0.5p)
2. Lägg till nytt element (`add`). (0.5p)

(B) Skriv en klass, `BinarySearchTreeImpl<E>`, som implementerar `BinarySearchTree<E>`. Om du inte löst (A), implementera istället en klass `StringBinarySearchTreeImpl` som implementerar `StringBinarySearchTree` ovan. (7p)

Tips:

1. Du måste inte tillåta blandade typer i ett och samma träd.
2. En innerklass kan vara till god hjälp.
3. Du behöver bara implementera metoderna som ingår i interfacet (utelämna `toString`, `equals`, `hashCode`).

Uppgift 4 (10p)

Pelle är mycket nöjd med att ha skrivit ihop några implementationer som utför numerisk (reell) integrering samt partiell derivering.
Gränssnitten för funktions-, integrerings- och deriveringsklasserna ser ut som följer:

```
public interface Integrator {
  double integrate(RealFun f, double a, double b);
}

public interface Differentiator {
  double differentiate(RealFun f, double x);
}
```

```
public interface RealFun {
  double f(double x);
}
```

Vid ett första test skapar Pelle två klasser som ser ut som följer:

```
public QuadraticFunction implements
      RealFun {
  private final double a, b, c;

  public QuadraticFunction(double a,
      double b, double c) {
    this.a = a;    this.b = b;
    this.c = c;
  }

  @Override public double f(double x) {
    return this.a * x * x + this.b * x +
      this.c;
  }
}
```

```
public ExpFunction implements
      RealFun {
  private final double base, scale;

  public ExpFunction(double base,
      double scale) {
    this.base = base; this.scale = scale;
  }

  @Override public double f(double x) {
    return Math.pow(this.base,
      scale * x);
  }
}
```

Strax därefter inser han att det vore en klar finess att kunna beräkna exponentialfunktioner där exponenten är ett polynom. Ytterligare en klass...
Bråk av ett tredjegrads- och ett andragradspolynom...
Väldigt många klasser blev det. Och ändå räcker det inte; ytterligare en klass behövs för varje ny typ av funktion!
Hjälp Pelle genom att skapa klasser som låter användaren kombinera funktioner under körning.
Du behöver inte implementera `equals`, `hashCode` och `toString` för klasserna.

(A) Följande typer av funktioner ska kunna skapas:
1. Numeriskt konstanta, $f(x) = a$ (0.5p)
2. Linjära funktioner, $f(x) = ax$ (0.5p)
3. Produkt av två funktioner, $f(x)\ g(x)$ (1p)
4. Kvot av två funktioner. $f(x) / g(x)$ (1p)
5. Summa av två funktioner. $f(x) + g(x)$ (1p)
6. Potens-/exponentialfunktioner, $f(x)^{g(x)}$ (1p)
7. Kedjning, $f(g(x))$ (1p)

(B) Visa med ett kort exempel (programkod och möjligen ett diagram) hur följande funktioner kan skapas, givet de generella klasserna från (A)-uppgiften:
1. $2x^2 + 3^{-x}$ (1p)
2. $x^{2/x}$ (1p)

(C) Skapa en fabriksmetod (factory method) som tar tre parametrar, a, b & c och returnerar en funktion $f(x) = ax^2 + bx + c$ (2p)

Tips:

En smidig lösning innebär att composite-mönstret används.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 5 (6p)

Syftet med klassen `PrimeSet` nedan är att den ska kunna skapa en lista med alla primtal mellan `min` och `max`, m.h.a. ett antal trådar; `min`, `max` och antalet trådar ges som argument på kommandoraden. Antag att användare bara ger korrekta indata.
Tyvärr är det som händer när man kör programmet inte alls att man får ut en lista med alla primtal mellan `min` och `max`. Istället dyker det upp massa exceptions som ser ut ungefär så här:

```
Exception in thread ”Thread-0” java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds
for length 10
    ...
    at java.base/java.util.ArrayList.add(ArrayList.java:498)
    atPrimeSet$1.run(PrimeSet.java:47) // Raden markerad med <<---- 1 nedan.
    at java.base/java.lang.Thread.run(Thread.java:835)
```

samt

```
Exception in thread ”Thread-5” java.lang.IndexOutOfBoundsException: Index 0 out of bounds for
length -1
    ...
    at java.base/java.util.ArrayList.remove(ArrayList.java:535)
    at PrimeSet$2.run(PrimeSet.java:70) // Raden markerad med <<---- 2 nedan.
    at java.base/java.lang.Thread.run(Thread.java:835)
```

Efter ett antal såna här exceptions ställer sig programmet och skriver ut ”Waiting...” och verkar aldrig köra klart.

Klassen `PrimeSet`:

```
public class PrimeSet {
    private static final int MAX_SIMULTANEOUS_WORKUNITS = 1000;
    private final List<Long> workUnits, primes;
    private final long min, max;
    private long current;
    private final int threads;
    public PrimeSet(final int threads, final long min, final long max) {
        this.threads = threads;      this.max = max;        this.min = min;
        this.workUnits = new ArrayList<>();
        this.primes = new ArrayList<>();
    }
    public List<Long> getPrimes() {
        if (!isFinished()) { throw new IllegalStateException(”Not done.”); }
        Collections.sort(primes);
        return Collections.unmodifiableList(primes);
    }
    public static void main(final String[] args) throws InterruptedException {
        long min = Long.parseLong(args[0]);
        long max = Long.parseLong(args[1]);
        int threads = Integer.parseInt(args[2]);
        final PrimeSet s = new PrimeSet(threads, min, max);
        s.start();
        while (!s.isFinished()) {
            System.out.println(”Waiting...”);    Thread.sleep(1000);
        }
        for(long l : s.primes) {         System.out.println(l); }
    }
    public boolean isFinished() {    return current >= max && workUnits.isEmpty(); }
```

```
    private static boolean isPrime(final long n) {
        // A method without side effects, returns true if n is very probably (1 - 2^-100) prime.
        return BigInteger.valueOf(n).isProbablePrime(100);
    }

    public void start() {
        if(min == 2) { primes.add(2); }                 // 2 is the only even prime.
        this.current = min % 2 == 0 ? min + 1 : min;   // Ensure we start at an odd number.
        // Create and start a number generator thread.
        final Runnable numberGenerator = new Runnable() {
            @Override public void run() {
                while (current < max) {
                    // Make sure that no more than MAX_SIMULTANEOUS_WORKUNITS numbers are
                    // enqueued at any time.
                    if (workUnits.size() < MAX_SIMULTANEOUS_WORKUNITS) {
                        workUnits.add(current); // <<---- 1
                        current += 2;
                    } else {
                            // Sleep for a short while if the queue is ”full”.
                            try {   Thread.sleep(10); } catch (final InterruptedException e) {
                            System.out.println(”Interrupted.”);      return;
                        }
                    }
                }
            }
        };
        final Thread generatorThread = new Thread(numberGenerator);
        generatorThread.start();

        // Create and start the prime-testing threads.
        for (int i = 0; i < this.threads; i++) {
            final Runnable primeTester = new Runnable() {
                @Override public void run() {
                    while (!isFinished()) {
                        if (!workUnits.isEmpty()) {
                            final long candidate = workUnits.remove(0); <<----- 2
                            if (isPrime(candidate)) {
                                primes.add(candidate);
                            }
                        }
                    }
                }
            };
            final Thread primeTestThread = new Thread(primeTester);
            primeTestThread.start();
        }
    }
}
```

(A) Förklara vad det är som går fel. Varför kastar programmet exceptions och varför kör det aldrig klart? (4p)

(B) Åtgärda felet/felen. Du behöver inte skriva av hela den ursprungliga klassen men det ska tydligt framgå vilka förändringar du behöver göra. (2p)

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 6 (4p)

Henny Hacker har varit i farten igen. Nu har hon byggt ett mycket säkert system (tycker hon, alltså). Tanken är att det bara är klasser i paketet `se.henny.secure` som ska kunna skapa `VerySecureSystem`-objekt samt kunna lägga till och ta bort användare. Så är det tyvärr inte. Betrakta klasserna `VerySecureSystem`, `Clearance` samt `User` nedan.

```java
package se.henny.secure;
public final class VerySecureSystem {
  private final List<UserPrivileges> users =
    new ArrayList<>();
  public static class UserPrivileges {
    public final User user;
    public final Clearance clearance;
    public UserPrivileges(final User u,
        final Clearance c) {
      user = Objects.requireNonNull(u);
      clearance =
        Objects.requireNonNull(c);
    }
  }
  // Package private constructor;
  // only classes in the same package can
  // create instances of VerySecureSystem.
  VerySecureSystem() { }

  // Package private; only classes in the
  // same package can call addUser and
  // removeUser.
  void addUser(final User u,
    final Clearance c) {
    UserPrivileges up =
      new UserPrivileges(u, c)
    users.add(up);
  }

  void removeUser(final User u) {
    final Iterator<UserPrivileges> uIt
      = users.iterator();
    while (uIt.hasNext()) {
      final UserPrivileges up =
        uIt.next();
      if (up.user.equals(u)) {
        uIt.remove();
      }
    }
  }
  public List<UserPrivileges>
      getUserPrivileges() {
    return users;
  }
}
```

```java
package se.henny.secure;
public enum Clearance {
  Trusted, Untrusted;
}
```

```java
package se.henny.secure;
public final class User {
  public final String name;

  public User(final String name) {
    this.name =
      Objects.requireNonNull(name);
  }
  @Override public int hashCode() {
    return Objects.hash(name);
  }
  @Override public boolean equals(Object
      obj) {
    if (this == obj) {    return true; }
    if (obj == null || getClass() !=
        obj.getClass()) {
      return false;
    }
    User o = (User) obj;
    return this.name.equals(o.name);
  }
}
```

(A) Visa hur Ohederlige Otto kan lägga till och/eller ta bort användare med valfri clearance, givet att Otto får tillgång till ett objekt av `VerySecureSystem`-typ. Otto **kan inte lägga till eller förändra** klasser i paketet `se.henny.secure`. (2p)

(B) Föreslå en enkel förändring i Hennys kod som grusar Ottos ohederliga otygsplaner. (2p)

Uppgift 7 (4p)

En enkel färguppslagningstabell är implementerad m.h.a. klassen `Colour` nedan.
Koden är bristfällig och bryter mot Open-Closed-principen.

```
public class Colour {
  public enum Cols { RED, GREEN;  }
  private final Colour.Cols c;
  public Colour(final Colour.Cols c) {
      this.c = Objects.requireNonNull(c);
  }

  @Override public String toString() {
    switch (this.c) {
      case RED:  return ”Me red.”;
      case GREEN: return ”Me green.”;
    }
    return null; // Can’t happen.
  }

  public int toRGB() {
    switch (this.c) {
      case RED: return 0xFF0000;
      case GREEN: return 0x00FF00;
    }
    return 0; // Can’t happen.
  }
}
```

(A) Förklara varför/hur koden bryter mot OCP. (1p)

(B) Refaktorera så att OCP följs genom att utnyttja arv och dynamisk bindning. (3p)

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 8 (3p)

Betrakta interfacen `ReadableThing`, `ReadWritableThing` och klasserna `RWThingImplA` & `RWThingImplB`:

```
public interface ReadableThing<T> {
  /**
   * Reads a T from the thing.
   * @return The contents of the thing.
   */
  T read();
}
```

```
public interface ReadWritableThing<T> extends ReadableThing<T> {
    /**
     * Writes a T to the thing.
     * @param v The value to write.
     */
    void write(T v);
}
```

```
public class RWThingImplA<T> implements
        ReadWritableThing<T> {
    // ...

    @Override
    public T read() {
        // ...
    }

    @Override
    public void write(final T v) {
        // ...
    }
}
```

```
public class RWThingImplB<T> implements
        ReadWritableThing<T> {
    // ...

    @Override
    public T read() {
        // ...
    }

    @Override
    public void write(final T v) {
        // ...
    }
}
```

För att förenkla design kan det ibland vara fördelaktigt att t.ex. bara tillåta läsning men inte skrivning; jämför icke-muterbara klasser. Problemet här är att de klasser, som finns här, implementerar `ReadWritableThing`, inte bara `ReadableThing`.

Uppgift:

Skriv en kort klass som anpassar objekt av typen `ReadWritableThing<T>` till **bara** `ReadableThing<T>`. Den ska fungera för **både nuvarande och framtida** klasser som implementerar interfacet `ReadWritableThing<T>`. De befintliga klasserna och interfacen får **ej** ändras.

Uppgift 9 (4p)

Betrakta interfacen `Clock`, `ClockDisplay` och klassen `AtomClock` nedan.

```
public interface Clock {
  void start();
}
```

```
public interface ClockDisplay {
  void showTime(long time);
}
```

```
public class AtomClock implements Clock {
  private final ClockDisplay clockDisp;
  private final int granularity;
  private final int updateFreq;
  /**
   * @pre clock != null
   * @param clock The clock display
                 to update.
   * @param granularity The maximum clock
                       precision in ms.
   * @param updateFreq  The update frequ-
                       ency in ms.
   */
  public AtomClock(
    final ClockDisplay clock,
    final int granularity,
    final int updateFreq) {
    this.clockDisp = Objects.
      requireNonNull(clock);
    this.updateFreq = updateFreq;
    this.granularity = granularity;
  }

  @Override public void start() {
    while (true) {
      final long lastUpdated =
        System.currentTimeMillis();
      this.clockDisp.
        showTime(lastUpdated / 1000);
      try {
        while(System.currentTimeMillis() -
          lastUpdated < this.updateFreq) {
          Thread.sleep(this.granularity);
        }
      } catch (InterruptedException e) {
        // interrupt was called;
        // stop updates.
        return;
      }
    }
  }

  // Simple use-case, create a clock
  // to display time on BigBen.
  // BigBen is known to implement
  // ClockDisplay.
  public static void main(
      final String[] args) {
    final ClockDisplay d = getBigBen();
    final AtomClock c =
      new AtomClock(d, 100, 2000);
    c.start();
  }
}
```

Klassen `AtomClock` lider av (minst) en allvarlig brist: den kan bara knyta exakt en display till sig.

Uppgift:

Bygg om `AtomClock` med hjälp av ett för situationen passande designmönster så att den kan köra med godtyckligt många displayer (inklusive ingen display).
Displayer ska kunna lägga till och ta bort sig under klockans livscykel.
Även gränssnittet `ClockDisplay` kan behöva ändras eller ett nytt gränssnitt skapas.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Hjälpmetoder för testning

Desssa behöver inte skrivas av utan står dig fritt att anropa efter behov i en testklass.

```
public static List<Boolean> getRandomBooleanList(final int size,
          final SplittableRandom rng) {
   final List<Boolean> l = new LinkedList<>();
   for (int i = 0; i < size; i++) {
      Boolean b;
      switch (rng.nextInt(3)) {
         case 0:
            b = Boolean.FALSE;
            break;
         case 1:
            b = Boolean.TRUE;
            break;
         default:
            b = null;
      }
      l.add(b);
   }
   return l;
}
```

```
public static List<Boolean> getFilledBooleanList(final int size, final Boolean v) {
   final Boolean[] bArr = new Boolean[size];
   Arrays.fill(bArr, v);
   return List.of(bArr);
}
```

TENTAMEN – Lösningsförslag

Objektorienterad programvaruutveckling, fk

Uppgift 1

(A), (B) & (C):

```
public class HammingDistance {
    /**
     * @pre l1 != null && l2 != null && l1.size() == l2.size()
     * @post l1 and l2 unchanged.
     * @param <T> The type of the lists.
     * @param l1  The first list.
     * @param l2  The second list.
     * @return The positions in which l1 and l2 differ.
     */
    public static <T> int hammingDistance(final List<T> l1,
            final List<T> l2) {
        int distance = 0;
        // This test could be omitted iff we declare it to be a precondition.
        // If either l1 or l2 is null, a NullPointerException will be thrown,
        // also in accordance with the precondition.
        if (l1.size() != l2.size()) {
            throw new IllegalArgumentException("l1 and l2 differ in size, " +
                l1.size() + " != " + l2.size());
        }
        // Preconditions fulfilled.
        final Iterator<T> it1 = l1.iterator();
        final Iterator<T> it2 = l2.iterator();
        while (it1.hasNext()) {
            final T i1 = it1.next();
            final T i2 = it2.next();
            if (i1 != i2) {
                // We only need to check if i1 is null.
                // If i1 == null && i2 == null => i1 == i2 and we won't
                // end up in this case.
                if (i1 == null || !i1.equals(i2)) {
                    distance++;
                }
            }
        }
        return distance;
    }
}
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 1, fortsättning

```
// Test class.
import static exam.sol.q1.HammingDistance.hammingDistance;

class HammingDistanceTest {
   // A few useful lists. Similar lists could have been set up within the
   // @BeforeEach annotation as well.
   private List<String> l1 = List.of("1");
   private final List<String> l2 = List.of("1");
   private final List<String> l3 = List.of("0");

   // (D) "Testa att metoden korrekt hanterar data som bryter mot förvillkoren",
   // Jämför testfall från övning V6.5.B.7
   @Test  void testHammingDistanceInvalid() {
      // Vad man kommer fram till här är något som passar de förvillkor
      // man angivit. Om man använt assertions för förvillkoren i implementationen
      // så blir det andra exceptions.
      Assertions.assertThrows(NullPointerException.class,
         () -> hammingDistance(null, null));
      Assertions.assertThrows(NullPointerException.class,
         () -> hammingDistance(this.l1, null));
      Assertions.assertThrows(IllegalArgumentException.class,
         () -> hammingDistance(List.of(), this.l1));
   }

   // (E) Testa symmetri, alltså att hammingDistance(a, b) = hammingDistance(b, a).
   // Jämför hjälpmetoder från övning V6.5.B
   // Vi använder våra hjälpmetoder,
   static List<Boolean> getRandomBooleanList(final int size,
      final SplittableRandom rng) { ... }
   // och
    static List<Boolean> getFilledBooleanList(final int size, final Boolean v) { ... }

   @Test  void testHammingDistanceSymmetry() {
      final SplittableRandom rng = new SplittableRandom(1);
      // Test for a few differently sized inputs.
      for (int i = 0; i < 20; i++) {
         final int size = (i + 1) * 10;
         final List<Boolean> l1 = getRandomBooleanList(size, rng);
         final List<Boolean> l2 = getRandomBooleanList(size, rng);
         final int d1 = hammingDistance(l1, l2);
         final int d2 = hammingDistance(l2, l1);
         Assertions.assertEquals(d1, d2);
      }
   }
```

```
// (F)
// Något av följande test är att betrakta som enkla:
// Testa att H(a, a) = 0:
    @Test void testHammingIdentity() {
        final SplittableRandom rng = new SplittableRandom(1);
        for (int i = 0; i < 20; i++) {
            final int size = (i + 1) * 10;
            final List<Boolean> l1 = getRandomBooleanList(size, rng);
            final int d = hammingDistance(l1, l1);
            Assertions.assertEquals(d, 0);
        }
    }
```

```
// Testa att 0 <= H(a, b) <= a.size():
    @Test void testHammingRange() {
        final SplittableRandom rng = new SplittableRandom(1);
        for (int i = 0; i < 20; i++) {
            final int size = (i + 1) * 10;
            final List<Boolean> l1 = getRandomBooleanList(size, rng);
            final List<Boolean> l2 = getRandomBooleanList(size, rng);
            final int d = hammingDistance(l1, l2);
            Assertions.assertTrue(d >= 0);
            Assertions.assertTrue(d <= size);
        }
    }
```

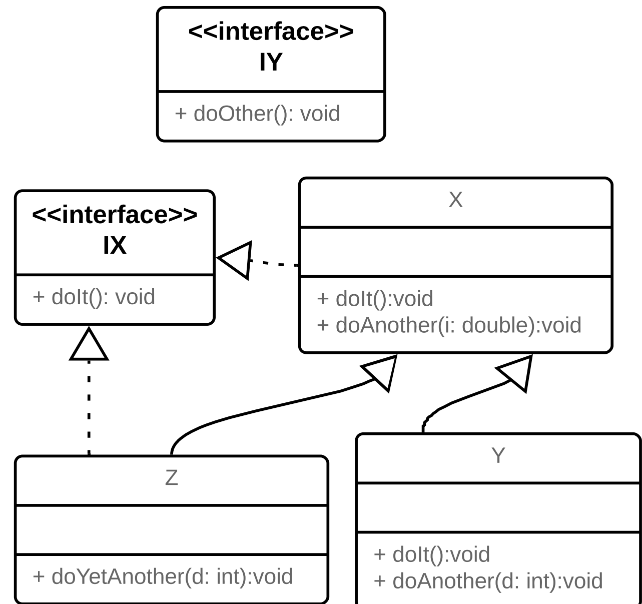
```
// Testa att H(a, b) == a.size() när a och b skiljer sig åt i alla positioner:
    @Test void testHammingDisjoint() {
        for (int i = 0; i < 20; i++) {
            final int size = (i + 1) * 10;
            final List<Boolean> l1 = getFilledBooleanList(size, Boolean.FALSE);
            final List<Boolean> l2 = getFilledBooleanList(size, Boolean.TRUE);
            Assertions.assertEquals(size, hammingDistance(l1, l2));
        }
    }
```

(G) `Collection` har inte nödvändigtvis någon särskild **ordning** i motsats till `List`. Då Hamming-avståndet definieras utifrån elementens **position** blir alltså en `Collection` för svag.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 2

(A) UML-diagram till höger:



(B)

```
// 1
final Z zx = new X(); // Kompileringsfel; en X är inte Z även om en Z är en X.
zx.doIt(); // Ger ingen utskrift då zx inte har något värde.

// 2
final IX ixy = new Y(); // Ok. En Y är en X är en IX.
final IY iya = ixy; // Kompileringsfel; en IX är inte en IY.
ixy.doAnother(); // Också kompileringsfel, doAnother() finns inte i IX.

// 3
final X xy1 = new Y(); // Ok. En Y är också en X.
xy1.doIt(); // Ger utskrift ”doIt Y”.

// 4
final IX ixz1 = new Z(); // Ok. En Z är en X (som också är en IX).
ixz1.doIt(); // Ger utskrift ”doIt X”

// 5
final IX ixz2 = new Z(); // Ok. En Z är en X (som också är en IX).
final X x = ixz2; // Kompileringsfel; en IX är inte en X (men en X är en IX).
x.doIt(); // Ger ingen utskrift då x inte har något värde.

// 6
final X xy2 = new Y(); // Ok. En Y är också en X.
xy2.doAnother(1);   // Ger utskrift ”doAnother X 1.0”. Notera att det är X:s metod
                    // som körs. Om Y.doAnother hade haft likadan parameterlista
                    // som X.doAnother så hade det blivit överskuggning istället.
```

Uppgift 3

(A) Interface:

```
public interface BinarySearchTree<
    E extends Comparable<? super E>> {
  boolean isEmpty();
  int size();
  boolean contains(E o);
  boolean add(E o);
}
```

En enklare inskränkning av E (som inte tillåter blandade typer i trädet) är `<E extends Comparable<E>>`

(B) En fungerande implementation kan se ut så här:

```
public class BinarySearchTreeImpl<
    E extends Comparable<? super E>>
  implements BinarySearchTree<E> {
  private Node<E> root;
  private int size;
  boolean value;

  // Static inner class; it depends on no
  // attributes of the enclosing class.
  private static class Node<
      E extends Comparable<? super E>>
      implements Comparable<E> {
    final E value;
    Node<E> leftChild, rightChild;
    // Creates a new node with both
    // children being null.
    public Node(final E value) {
      this.value = value;
      this.leftChild = this.rightChild = null;
    }
    @Override
    public int compareTo(final E item) {
      return this.value.compareTo(item);
    }
  }

  public BinarySearchTreeImpl() {
    root = null;    size = 0;
  }

  @Override
  public boolean contains(final E item) {
    if (isEmpty()) { return false; }
    return searchAndInsert(this.root,
      false, item);
  }
  @Override public boolean isEmpty() {
    return root == null;
  }
  @Override  public int size() { return size; }

  @Override
  public boolean add(final E item) {
    if (this.root == null) {
      this.root = new Node<>(item);
      this.size = 1;
      return true;
    }
    // Note that add and contains have reversed
    // results, add shall return false
    // if the item was in the tree.
    return !searchAndInsert(this.root,
      true, item);
  }

  private boolean searchAndInsert(
    final Node<E> curNode,
    final boolean insert, final E item) {
    final int c = curNode.compareTo(item);
    // Is the element already in the list?
    if (c == 0) {  return true;  }
    if (c < 0) {
      // If the left child is empty, the item
      // was not found.
      // If insert is true, add a new node as
      // the current node's left child.
      if (curNode.leftChild == null) {
        if (insert) {
          curNode.leftChild = new Node<>(item);
          this.size++;
        }
        return false;
      }
      // The left child was not empty:
      // continue looking, using the left
      // child as the new root node.
      return searchAndInsert(curNode.leftChild,
        insert, item);
    }
    // Handle the right child the same way
    // the left child was handled.
    if (curNode.rightChild == null) {
      if (insert) {
        curNode.rightChild = new Node<>(item);
        this.size++;
      }
      return false;
    }
    return searchAndInsert(curNode.rightChild,
    insert, item);
  }
}
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 4

(A) 1.

```
public class ConstFun implements RealFun {
  private final double a;
  public ConstFun(double a) { this.a = a; }

  @Override public double f(final double x) {
    return this.a;
  }
}
```

(A) 2.

```
public class LinFun implements RealFun {
  private final double a;
  public LinFun(double a) { this.a = a; }

  @Override  public double f(double x) {
    return this.a * x;
  }
}
```

(A) 3--7.

```
Låt oss kalla klasserna
3. ProdFun
4. RatFun
5. SumFun
6. PowFun
7. ChainFun
och ersätt FunName och f:
```

```
public class FunName implements RealFun {
  private final RealFun f, g;

  public FunName(final RealFun f,
    final RealFun g) {
    this.f = f; this.g = g;
  }

  @Override public double f(double x) {
// Implementationer för 3--7:
    3: return f.f(x) * g.f(x);
    4: return f.f(x) / g.f(x);
    5: return f.f(x) + g.f(x);
    6: return Math.pow(f.f(x), g.f(x);
    7: return f.f(g.f(x));
  }
```

Ett alternativ här är att använda en abstrakt klass för A.3–7 men vinsten blir inte så stor; varje konkret subklass måste ändå ha sin egen konstruktor.

(B) 1. $2x^2 + 3^{-x}$

```
RealFun xSquared =
  new PowFun(new LinFun(1),
    new ConstFun(2));
RealFun firstTerm =
  new ProdFun(new ConstFun(2),
    xSquared);
RealFun secondTerm =
  new PowFun(new ConstFun(3),
    new LinFun(-1));
RealFun b1Fun =
  new SumFun(firstTerm,
    secondTerm);
```

(B) 2. $x^{2/x}$

```
RealFun b2Fun =
  new PowFun(new LinFun(1),
   new RatFun(new ConstFun(2),
    new LinFun(1)));
```

(C)

```
public static RealFun getQuadraticPolynomial(
     double a, double b, double c) {
  RealFun quadraticTerm =
    new ProdFun(
      new ConstFun(a),
      new PowFun(
        new LinFun(1),
        new ConstFun(2)));
  RealFun linearTerm =
    new LinFun(b);
  RealFun constTerm =
    new ConstFun(c);
  return new SumFun(quadraticTerm,
    new SumFun(linearTerm, constTerm));
}
```

Eller ännu enklare (jag glömde skriva att ni bara fick använda er av klasserna från A-uppgiften i 4.C...):

```
public static RealFun getQuadraticPolynomial(
     double a, double b, double c) {
  // Här kan man dock låta bli att skapa
  // ett nytt objekt. Man skulle kunna
  // lagra undan funktioner och återanvända
  // dem de efterfrågas igen. Det går med en
  // fabriksmetod men inte med en konstruktor.
  return new QuadraticFunction(a, b, c);
}
```

Uppgift 5

(A) Programmet kastar exceptions då både producent- och konsumenttrådarna **samtidigt** försöker skriva till och läsa från samma samling, en `ArrayList`. Problemet med detta är att två trådar båda kan konstatera att kön inte är tom, den ena tråden tar det sista elementet och sen får den andra tråden problem; den försöker ta bort ett element ur en tom lista.
När producent-tråden väl misslyckats med att lägga till ett element så avbryts tråden p.g.a. att ett exception som inte fångas kastas. (`<<---- 1` i exemplet.)
När en konsumenttråd väl misslyckats (exception kastas vid `<<---- 2`) så avbryter den.
Till slut finns det inga körande konsumenttrådar kvar och programmet går in i en oändlig loop där det väntar på att `workUnit.size()` ska gå ner till 0. (Om man ska vara petig så är även variabeln `primes` problematisk.)

(B) Det går lätt att åtgärda problemet enligt följande:
Vi isolerar problemet till att bara omfatta simultana accesser (läs/skriv) till variabeln `workUnit`.
Genom att synkronisera över just den variabeln får exakt en tråd operera på den åt gången:

```
...
boolean queueIsFull = false;
// Man hade här kunna synkronisera över hela klassen genom att utelämna workUnits ur
// synchronized-direktivet. Det hade dock varit sämre ur prestandasynpunkt.
synchronized (workUnits) {
    queueIsFull = workUnits.size() >= MAX_SIMULTANEOUS_WORKUNITS;
    if (!queueIsFull) {
        // This is now safe; only one thread at a time can execute here due to the
        // operation being within a block synchronized on workUnits.
        workUnits.add(current); // <<---- 1
        current += 2;
    }
}
...
while (!isFinished()) {
    synchronized (workUnits) {
        // Do all queries and manipulations on workUnits within the synchronized block.
        if (!workUnits.isEmpty()) {
            final long candidate = workUnits.remove(0); // <<----- 2
            if (isPrime(candidate)) {
                primes.add(candidate);
            }
        }
    }
}
```

En nästan-lösning: att använda en trådsäker samling.

I klassen `java.util.Collections` finns metoden ”`static <T> List<T> synchronized List (List<T> list)`”. Givet en lista så returnerar den en synkroniserad lista. Detta hjälper dock inte fullt ut; det finns en risk att listan hinner ändra sig mellan det att anropen `workUnits.isEmpty()` och `workUnits.remove(0)` sker – en annan tråd kan ha hunnit plocka av elementet efter det att `!workUnits.isEmpty()` kontrollerades.

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 6

(A) Problemet uppstår eftersom Otto kan få ut en referens till Hennys lista. Exempel:

```
package com.1337haxxor.otto;

// Otto kan inte anropa addUser och removeUser i VerySecureSystem;
// de är package-private och således inte synliga från Ottos paket.
public class OttosSecurityViolator {private final VerySecureSystem s;
   // Otto somehow needs to get hold of a reference to a secure system.
   public OttosSecurityViolator(final VerySecureSystem s) {
      this.s = s;
   }
   public void addUser(final User u, final Clearance c) {
      final List<UserPrivileges> users = s.getUserPrivileges();
      users.add(new UserPrivileges(u, c)); // Oops! Not what Henny intended.
   }
   public void removeUser(final User u) {
      final Iterator<UserPrivileges> uIt = s.getUserPrivileges().iterator();
      while (uIt.hasNext()) {
         final UserPrivileges up = uIt.next();
         if (up.user.equals(u)) {
            uIt.remove(); // Oops! Not what Henny intended either.
         }
      }
   }
}
```

(B) Genom att bara lämna ut en kopia av listan så blir lösningen som Henny avsåg; alla kan läsa listan men bara klasser i Hennys säkra paket kan skriva till den. (Notera att `VerySecureSystem` är deklarerad som `final`, andra klasser kan alltså inte ärva från den heller.)

```
   public List<UserPrivileges> getUserPrivileges() {
      // The point here is not that the list is unmodifiable but that view
      // any caller gets most likely is a copy of the original list.
      // Any calls on the returned list is guaranteed to not mutate the
      // original. There are other ways to achieve the same goal,
      // but a defensive copy has to be made. (There are two other patterns,
      // Proxy and Facet, that also could be considered but we didn't cover them
      // in the the lecture notes or exercises.)
      return Collections.unmodifiableList(users); // Yay! Otto is thwarted!
   }
```

En icke-lösning:

Att förändra gränsytan för `VerySecureSystem`. Alla klasser ska kunna göra exakt det som `VerySecureSystem` tillåter från början, **förutom just att lägga till/ta bort/ändra rättigheter**.

Uppgift 7

(A) Klassen bryter mot Open-Closed-principen då den inte är öppen för utökning;
det enda sättet att lägga till fler färger är genom att **ändra** i klassen. Den är därmed inte closed for change om den ska vara open for extension.

(B) Ett sätt att lösa problemet är att antingen definiera ett gemensamt gränssnitt eller klass och låta varje färg få sin egen klass. Då alla färger har ganska mycket gemensamt verkar en klass enklast.
För full poäng krävs inte att man implementerar de specifika färgerna som singletons även om det är lämpligt för just det här fallet.

```
public class Colour {
   private final String message;
   private final int rgbValue;

   public Colour(final String message, final int rgbValue) {
      this.message = Objects.
         requireNonNull(message);
      this.rgbValue = rgbValue;
   }

   @Override public String toString() {
      return message;
   }

   public int toRGB() {
      return rgbValue;
   }
}
```

```
public class BlueColour extends Colour {
   // Implement as a singleton. Users would simply refer to
   // BlueColour.INSTANCE for the (only) blue instance.
   public static final BlueColour INSTANCE = new BlueColour();

   // Note the private constructor.
   private BlueColour() {
      super(”Me blue.”, 0x0000FF);
   }
}
```

```
Analogt för grön och röd.
```

CHALMERS
Institutionen för data- och informationsteknik
Pelle Evensen, evensen@chalmers.se

Uppgift 8

Att ärva från klassen hjälper inte; vi kan genom arv bara stärka/generalisera en specifikation, inte försvaga/specialisera. Det vi däremot kan göra är att presentera en vy av ett objekt som vi kapslar in:

```
public class ReadableAdapter<T> implements ReadableThing<T> {
    private final ReadableThing<T> t;

    public ReadableAdapter(final ReadableThing<T> t) {
        this.t = Objects.requireNonNull(t);
    }

    @Override
    public T read() {
        return t.read();
    }
}
```

Exempel:

```
final ReadWritableThing<String> thing = new RWThingImplA<>();

// Works as expected.
thing.write("Pelle");

final ReadableThing<String> nonWritableThing = new ReadableAdapter<>(thing);

// Works fine.
nonWritableThing.read();

// Compilation error. ReadableThing doesn't have a write-method. Good.
nonWritableThing.write();

// Runtime error. ReadableThingAdapter can not be cast to ReadWritableThing. Good.
((ReadWritableThing<String>) nonWritableThing).write("Pelle");

// A non-solution:
final ReadableThing<String> nonSolution = thing;
// nonSolution has the static type ReadableThing, so far so good...

// But this can be circumvented:
final ReadWritableThing<String> writableAfterAll = (ReadWritableThing<String>) nonSo-
lution;
// --- Not an error since the runtime type of nonSolution is a ReadWritableThing!
```

Uppgift 9

Vi bygger om klockan så att den blir observerbar genom att först skapa ett nytt gränssnitt för **observerbara** klockor, `ObservableClock`. Vi gör motsvarande förändring för att displayer ska kunna **observera** genom att implementera `PropertyChangeListener`.
Det finns andra sätt (och klasser/interface) men för full poäng så får det inte finnas några cirkulära beroenden mellan displayer och klockor. Idealiskt är att både klocka och display kommunicerar via gemensamma gränssnitt, inte implementationer (jfr. Dependency Inversion Principle).

```
public interface ObservableClock extends Clock {
 void addObserver(PropertyChangeListener l);
 void removeObserver(PropertyChangeListener l);
}
```

```
public interface ObserverDisplay extends PropertyChangeListener {
    // For now, doesn't provide any extras; a very simple observer.
}
```

```
public class ObservableAtomClock implements ObservableClock {
    private final PropertyChangeSupport obs;
    ...
    public ObservableAtomClock(final int granularity, final int updateFreq) {
        obs = new PropertyChangeSupport(this);
        ...
    }
    ...
    // replace clockDisp.showTime(lastUpdated / 1000); ->
    obs.firePropertyChange("newTime", 0, lastUpdated / 1000);
    ...
    @Override public void addObserver(final PropertyChangeListener l) {
        obs.addPropertyChangeListener(l);
    }
    @Override  public void removeObserver(final PropertyChangeListener l) {
        obs.removePropertyChangeListener(l);
    }
}
```

```
Exempel på användning (behövs inte ges som svar):
    public static void main(final String[] args) throws InterruptedException {
        final ObserverDisplay d1 = new ObserverDisplay() {
            @Override public void propertyChange(PropertyChangeEvent evt) {
                System.out.println("Display 1: " + evt.getNewValue());
            }};
        final ObserverDisplay d2 = new ObserverDisplay() {
            @Override public void propertyChange(PropertyChangeEvent evt) {
                System.out.println("Display 2: " + evt.getNewValue());
            }};
        final ObservableAtomClock c = new ObservableAtomClock(100, 2000);
        // Starting the clock in a separate thread for demonstration purposes.
        final Thread clockThread = new Thread(() -> c.start());
        clockThread.start();
        c.addObserver(d1);    Thread.sleep(5000);
        c.addObserver(d2);    Thread.sleep(5000);
        c.removeObserver(d1);    Thread.sleep(5000);
        clockThread.interrupt();
        System.out.println("Finished. Disintegrating atoms.");
    }
```