Tentamen – extra del: lösning

Uppgifter: lösningar

Uppgift 1 (4 poäng)

```
// sort sorterar element i en vektor enligt deras storlek,
// från och med det minsta till och med det största
public static void sort (int[] element)
{
    int    forst = 0;
    int    sist = element.length - 1;
    int    minst;
    int    aktuell;

    while (forst < sist)
    {
        minst = forst;
        aktuell = forst + 1;
        while (aktuell <= sist)
        {
            if (element[aktuell] < element[minst])
                minst = aktuell;

            aktuell++;
         }

        int    p = element[forst];
        element[forst] = element[minst];
        element[minst] = p;

        forst++;
     }
}
```

Uppgift 2 (2 poäng + 1 poäng + 1 poäng)

a) (2 poäng)

Man skriver `Rectangle2D.Double`. Det betyder att klassen `Double` definieras som en nästlad klass inuti klassen `Rectangle2D`.

Referensen `rect` av typen `Rectangle2D` refererar till ett objekt av typen `Double`. Det betyder att klassen `Double` är en subklass till klassen `Rectangle2D`.

b) (1 poäng)

Vektorn `shapes` innehåller referenser av typen `Shape`. Dessa referenser kan referera till objekt av alla klasser som implementerar detta gränssnitt.

Ett objekt av klassen `Rectangle2D.Double` och ett oobjekt av klassen `Ellipse2D.Double` finns i vektorn `shapes`. Det betyder att dessa klasser implementerar gränssnittet `Shape`.

c) (1 poäng)

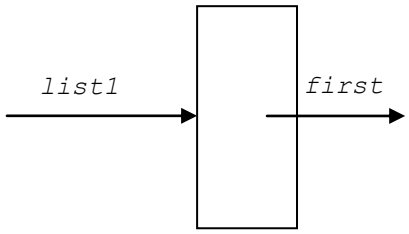
Metoden `draw` har en parameter av typen `Shape`. Det betyder att den kan rita alla figurer av typen `Shape` (objekt av alla klasser som implementerar gränssnittet `Shape`). Eftersom klasserna `Rectangle2D.Double` och `Ellipse2D.Double` implementerar gränssnittet `Shape`, kan metoden `draw` rita figurer av dessa typer:

```
Shape   r = new Rectangle2D.Double (10.0, 20.0, 60.0, 40.0);
Shape   e = new Ellipse2D.Double (70.0, 80.0, 40.0, 60.0);
g.draw (r);
g.draw (e);
```

Uppgift 3 (1 poäng + 3 poäng)

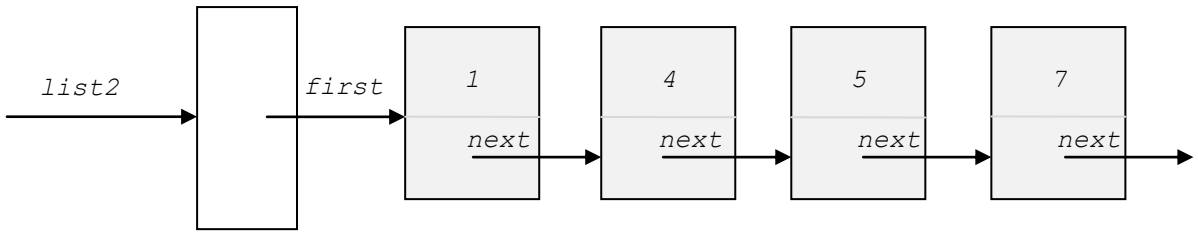
a) (1 poäng)

```
List    list1 = new List ();
```



b) (3 poäng)

```
List    list2 = new List ();
int[]    v = {1, 4, 5, 7};
for (int i = 0; i < v.length; i++)
    list2.add (v[i]);
```



Uppgift 4 (2 poäng + 2 poäng)

a) (2 poäng)

```
// countDigits tar emot en teckensekvens, och returnerar
// antalet siffror i den teckensekvensen.
public static int countDigits (CharSequence cs)
{
    int     count = 0;
    char    c = 0;
    int     len = cs.length ();
    for (int i = 0; i < len; i++)
    {
        c = cs.charAt (i);
        if (Character.isDigit (c))
            count++;
    }

    return count;
}
```

b) (2 poäng)

```
String    s = new String ("tio 10 femtio 50");
int    count1 = countDigits (s);
```

```
StringBuilder    sb = new StringBuilder ("E1D2C3B4A5");
int    count2 = countDigits (sb);
```

Uppgift 5 (1 poäng + 1 poäng + 2 poäng)

a) (1 poäng)

I bästa fall finns det sökta elementet på den första positionen i sekvensen. I så fall utförs enbart en elementjämförelse. Algoritmens tidskomplexitet är:

$b(n) = 1$

b) (1 poäng)

Det värsta fallet uppstår i två situationer: när det sökta elementet inte finns i sekvensen, och när det sökta elementet finns enbart på den sista positionen i sekvensen. I så fall jämförs elementet med alla element i sekvensen. Algoritmens tidskomplexitet är:

$w(n) = n$

c) (2 poäng)

Om det sökta elementet finns på den första positionen utförs *1* jämförelse, om det finns på den andra positionen utförs *2* jämförelser, och så vidare. Om elementet ligger på den sista positionen utförs *n* elementjämförelser. Sannolikheten för vart och ett av dessa fall är *1/n*. Det totala antalet jämförelser är:

$1/n + 2/n + 3/n + \ldots + n/n =$

$= (1 + 2 + 3 + \ldots + n)/n =$

$= (n(n + 1)/2)/n =$

$= (n + 1)/2$

Algoritmens tidskomplexitet i ett genomsnittligt fall är:

$a(n) = (n + 1)/2$