Programmeringsteknik för BME1, F1, I1 och N1

PROGRAMMERINGSUPPGIFTER

EDAA50/EDAA55

`http://cs.lth.se/edaa5x`

LUNDS UNIVERSITET
Lunds Tekniska Högskola

2021/2022

Innehåll

1 Java-laborationer

Java-laborationerna ger exempel på tillämpningar av det material som behandlas under kursen och ger träning på att skriva program. I kursen ingår tio sådana laborationer.

Observera att samtliga Java-laborationer, liksom projektet, måste vara godkända innan du får lov att tentera! (Matlab-laborationerna, endast F1/I1 (EDAA55), är undantagna, och rapporteras separat i Ladok – se s. 61.)

Anmälan till laborationsgrupp sker via kursens hemsida (`cs.lth.se/edaa5x`). Laboration 1–5 är schemalagda till VT1 2022 och 6–10 till VT2 2022.

Laborationerna är individuella, det vill säga att de ska lösas med självständigt enskilt arbete. Det är tillåtet att diskutera laborationerna och dess lösningar med kurskamraterna, men var och en måste skriva sin egen lösning. Du måste arbeta med varje laboration under ”rätt” vecka (om du inte är sjuk, se nedan). Om du inte blir klar med laborationen, redovisa då den i första hand på nästa resurstid. Om du har alltför många laborationer efter dig kan du bli hänvisad till nästa kursomgång.

- Du behöver lösa laborationen **självständigt**, så du är färdig till den redovisningstid du bokat. Använd våra resurstider om du behöver hjälp.
- Innan du sätter igång med laborationen bör du
  - göra den övning som står som förberedelse till laborationen,
  - studera läroboken enligt läsanvisningarna,
  - läsa igenom *hela* laborationen noggrant,
  - lösa förberedelseuppgifterna. I dessa uppgifter ska du skriva delar av de program som ingår i laborationen. Kontakta en lärare om du får problem med uppgifterna.
- Om du är sjuk vid något laborationstillfälle så måste du anmäla detta till kursansvarig (patrik.persson@cs.lth.se) före laborationen. Annars riskerar du att bli hänvisad till nästa gång kursen går (nästa läsår), och du kan då inte slutföra kursen (och få poäng) förrän då.

  Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den vid ett senare tillfälle. Har du varit sjuk och behöver hjälp för att lösa laborationen så gå till något av de resurstillfällen som finns tillgängliga för det program du läser. Det kommer också att finnas ett uppsamlingstillfälle i slutet av kursen.

Laboration 1 – introduktion

Mål: Under denna laboration ska du lära dig vad ett datorprogram är och se exempel på enkla program. Du ska också lära dig att editera, kompilera och exekvera enkla Java-program med hjälp av programutvecklingsverktyget Eclipse. Du ska också prova att använda Eclipse debugger.

Förberedelseuppgifter

- Läs i läroboken: Downey & Mayfield, kapitel 1–2.
- Installera Eclipse, Java och kursens workspace på din dator (om du inte redan gjort det). Använd instruktionerna på kurssidan.
- Läs avsnittet Bakgrund.

Bakgrund

Ett datorprogram är ett antal rader text som beskriver lösningen till ett problem. Vi börjar med att titta på ett mycket enkelt problem: att från datorns tangentbord läsa in två tal och beräkna och skriva ut summan av talen. Lösningen kan se ut så här i Java:

```
1  import java.util.Scanner;
2
3  public class Calculator {
4      public static void main(String[] args) {
5          System.out.println("Skriv två tal");
6          Scanner scan = new Scanner(System.in);
7          double nbr1 = scan.nextDouble();
8          double nbr2 = scan.nextDouble();
9          double sum = nbr1 + nbr2;
10         System.out.println("Summan av talen är " + sum);
11     }
12 }
```

Talen som summeras läses från tangentbordet. Du behöver inte förstå detaljerna i programmet nu, men vi vill redan nu visa ett program som gör någonting intressantare än bara skriver ut ”Hello, world!”. Allt som behövs för att du ska förstå programmet kommer vi att gå igenom under de första veckorna av kursen. Förklaring av de viktigaste delarna av programmet:

- Rad 1: en inläsningsklass `Scanner` importeras från ett ”paket” med namnet `java.util`. Detta paket och många andra ingår i Javas standardbibliotek.
- Rad 3: `public class Calculator` talar om att programmet, klassen, heter `Calculator`.
- Rad 4: `public static void main(String[] args)` talar om att det som vi skriver är ett ”huvudprogram”. Varje Javaprogram måste innehålla en klass med en `main`-metod. Exekveringen börjar och slutar i `main`-metoden.
- Rad 5: texten `Skriv två tal` skrivs ut på skärmen (`println` betyder ”print line”). När man använder Eclipse så hamnar utskrifter i konsolfönstret.
- Rad 6: inläsningsobjektet `scan` skapas. Vi kommer senare att gå igenom ordentligt vad detta betyder — nu räcker det att du vet att man alltid måste skapa ett sådant objekt när man vill läsa från tangentbordet.
- Rad 7: en variabel `nbr1` som kan innehålla ett reellt tal (`double`-tal) deklareras. Det betyder att man skapar plats för variabeln i datorns minne. Man tilldelar också `nbr1` ett talvärde som man läser in från tangentbordet med `nextDouble()`.
- Rad 8: samma sak med variabeln `nbr2`.

- Rad 9: variabeln `sum` deklareras. Talen `nbr1` och `nbr2` adderas och summan lagras i `sum`.
- Rad 10: resultatet (innehållet i variabeln `sum`) skrivs ut.
- Rad 11: slut på `main`-metoden.
- Rad 12: slut på klassen.

Uppgifter

1. Starta Eclipse. (Om du inte redan installerat programmet utifrån våra instruktioner på kurssidan, gör det först.)

2. Du ska nu ladda in filen *Calculator.java* i en editor så att du kan ändra den. Man måste klicka en hel del för att öppna en fil:

   a) Öppna projektet *Lab01* genom att klicka på plustecknet (eller pilen) bredvid projektet.
   b) Öppna katalogen *src* genom att klicka på plustecknet.
   c) Öppna paketet *(default package)* genom att klicka på plustecknet.
   d) Öppna filen *Calculator.java* genom att dubbelklicka på filnamnet. Filen öppnas i en editorflik.

3. Kör programmet: markera *Calculator.java* i projektvyn, högerklicka och välj Run As > Java Application. I konsolfönstret skrivs texten `Skriv två tal`. Klicka i konsolfönstret, skriv två tal och tryck på RETURN. Observera: när man skriver reella tal ska man *vid inläsning* använda decimalkomma. När man skriver reella tal i programkod använder man decimalpunkt.

   Man kan köra det senaste programmet en gång till genom att klicka på Run-ikonen i verktygsraden.

4. Ändra `main`-metoden i klassen `Calculator` så att fyra rader skrivs ut: talens summa, skillnad, produkt och kvot. Exempel på utskrift när talen 24 och 10 har lästs in:

   ```
   Summan av talen är 34.0
   Skillnaden mellan talen är 14.0
   Produkten av talen är 240.0
   Kvoten mellan talen är 2.4
   ```

   Du ska alltså efter utskriften av summan lägga in rader där talens skillnad, produkt och kvot beräknas och skrivs ut. Subtraktion anger man med tecknet `-`, multiplikation med `*`, division med `/`.

   Under tiden du skriver så kommer du att märka att Eclipse hela tiden kontrollerar så att allt du skrivit är korrekt. Fel markeras med kryss till vänster om raden. Om du håller musmarkören över ett kryss så visas en förklaring av felet. Om man sparar en fil som innehåller fel så visas felmeddelandena också i Problem-fliken längst ner.

5. Spara filen. Filen kompileras automatiskt när den sparas.

   När *.java*-filen kompileras skapas en ny fil med samma namn som klassen men med tillägget *.class*. Denna fil innehåller programmet översatt till byte-kod och används när programmet exekveras.

   Man ser inte `.class`-filerna inuti Eclipse, men de lagras i arbetsområdet precis som `.java`-filerna. *.java*-filerna finns i en katalog *src* under respektive projektkatalog, *.class*-filerna finns i en katalog *bin*.

6. Kör programmet och kontrollera att utskriften är korrekt. Rätta programmet om den inte är det.

7. Du ska nu använda Eclipse debugger för att följa exekveringen av programmet. Normalt använder man debuggern för att hitta fel i program, men här är avsikten bara att du ska se hur programmet exekverar rad för rad och att du ska bekanta dig med debuggerkommandona.

   - Sätt en brytpunkt på den första raden i `main`-metoden, genom att dubbelklicka till vänster om radnumret vid den rad det gäller. Det ska dyka upp en liten blå bubbla där, som betyder just att det finns en brutpunkt.

     Kör programmet under debuggern (Debug As > Java Application).
   - Svara ”ja” på frågan om du vill byta till Debug-perspektivet.
   - Klicka på Step Over-ikonen några gånger. Notera att den rad i programmet som ska exekveras markeras i editorfönstret och att variabler som deklareras dyker upp i variabelvyn till höger.

     Variablernas aktuella värden visas i ett av Eclipse-fönstren – se till att du ser detta och kan identifiera hur variablerna får sina värden under programmets exekvering.
   - Sätt en brytpunkt på raden där du skriver kvoten mellan talen.
   - Klicka på Resume-ikonen för att köra programmet fram till brytpunkten.
   - Klicka på Resume igen för att köra programmet till slut.

   Byt tillbaka till Java-perspektivet genom att klicka på knappen Java längst till höger i verktygsfältet.

8. Följande program använder den färdiga klassen `SimpleWindow`:

```
import se.lth.cs.pt.window.SimpleWindow;

public class SimpleWindowExample {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(500, 500, "Drawing Window");
        w.moveTo(100, 100);
        w.lineTo(150, 100);
    }
}
```

   Specifikationen av klassen `SimpleWindow` finns online via länk från kurshemsidan. Förklaring av de viktigaste delarna av programmet:

   - På den första raden importeras den färdiga klassen SimpleWindow.
   - Raderna som börjar med `public` och de två sista raderna med `}` är till för att uppfylla Javas krav på hur ett program ska se ut.
   - På nästa rad deklareras en referensvariabel med namnet `w` och typen `SimpleWindow`. Därefter skapas ett `SimpleWindow`-objekt med storleken 500×500 pixlar och med titeln ”Drawing Window”. Referensvariabeln `w` tilldelas det nya fönsterobjektet.
   - Sedan flyttas fönstrets ”penna” till punkten 100, 100.
   - Slutligen ritas en linje.

   Programmet finns inte i arbetsområdet, utan du ska skriva det från början. Skapa en fil *SimpleWindowExample.java*:

   a) Markera projektet *Lab01* i projektvyn,
   b) Klicka på New Java Class-ikonen i verktygsraden.

c) Skriv namnet på klassen (`SimpleWindowExample`).
d) Klicka på Finish.

Dubbelklicka på *SimpleWindowExample.java* för att ladda in filen i editorn (om detta inte redan gjorts automatiskt). Som du ser har Eclipse redan fyllt i klassnamnet och de parenteser som alltid ska finnas. Komplettera klassen med `main`-metoden som visas ovan.

Spara filen, rätta eventuella fel och spara igen. Provkör programmet.

9. Ändra programmet genom att välja bland metoderna i klassen `SimpleWindow`. Till exempel kan du rita en kvadrat, ändra färg och linjebredd på pennan, rita fler linjer, skriva text, etc.

10. Öppna filen *LineDrawing.java*. I filen finns följande kod, komplettera programmet så att det fungerar. Raderna med kommentarer ska ersättas med riktig programkod.

```
public class LineDrawing {
   public static void main(String[] args) {
       SimpleWindow w = new SimpleWindow(500, 500, "LineDrawing");
       w.moveTo(0, 0);
       while (true) {
           // vänta tills användaren klickar på en musknapp
           // rita en linje till den punkt där användaren klickade
       }
   }
}
```

Ledtråd: `SimpleWindow` innehåller också metoder för att ta hand om musklick. Dessa metoder har följande beskrivning:

```
/** Väntar tills användaren har klickat på en musknapp */
void waitForMouseClick();

/** Tar reda på x-koordinaten för musens position
    vid senaste musklick */
int getClickedX();

/** Tar reda på y-koordinaten för musens position
    vid senaste musklick */
int getClickedY();
```

Fundera ut vad som händer i programmet. `while (true)` betyder att repetitionen ska fortsätta ”i oändlighet”. Man avbryter programmet genom att välja Quit i File-menyn i `SimpleWindow`-fönstret.

Spara filen, rätta eventuella fel och spara då igen. Provkör programmet.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Starta Eclipse, öppna önskat arbetsområde (workspace) och öppna önskat projekt.
- Skapa .java-filer och skriva in enkla Java-program. (Med enkla program menas här ett program på några rader, som till exempel `Calculator`.)
- Kunna använda metoden `System.out.println` och förstå vad som händer när den används.
- Förstå vad det innebär att läsa in tal från tangentbordet.
- Spara .java-filer (kompilering sker då automatiskt).

- Exekvera program.
- Använda debuggern:
  - Sätta och ta bort brytpunkt.
  - Starta debuggern.
  - Köra fram till en brytpunkt med Resume.
  - Exekvera stegvis med Step over (och senare i kursen med Step Into).
  - Byta mellan debug-perspektivet och Java-perspektivet.

Laboration 2 – använda färdigskrivna klasser, kvadrat

Mål: Du ska lära dig att läsa specifikationer av klasser och att utnyttja färdigskrivna klasser för att lösa enkla uppgifter. Du ska också lära dig lite mera om Eclipse.

Förberedelser

- Läs avsnittet Bakgrund.
- Tänk igenom följande frågor och se till att du kan svara på dem:
  1. Vad är en main-metod? Hur ser en klass med en main-metoden ut i stora drag?
  2. Vad är referensvariabler och vad har man dem till?
  3. Hur skapar man objekt?
  4. Hur anropar man en metod på ett objekt?
  5. Vad använder man parametrar till?

Bakgrund

En klass `Square` som beskriver kvadrater har nedanstående specifikation.

```
/** Skapar en kvadrat med övre, vänstra hörnet i x,y
    och med sidlängden side. */
Square(int x, int y, int side);

/** Ritar kvadraten i fönstret w. */
void draw(SimpleWindow w);

/** Raderar bilden av kvadraten i fönstret w. */
void erase(SimpleWindow w);

/** Flyttar kvadraten avståndet dx i x-led, dy i y-led. */
void move(int dx, int dy);

/** Tar reda på x-koordinaten för kvadratens läge. */
int getX();

/** Tar reda på y-koordinaten för kvadratens läge. */
int getY();

/** Tar reda på kvadratens area. */
int getArea();
```

Användning av klassen Square

I nedanstående program skapas först ett ritfönster. Därefter skapas en kvadrat som placeras mitt i fönstret och ritas upp.

```
1  import se.lth.cs.pt.window.SimpleWindow;
2  import se.lth.cs.pt.square.Square;
3
4  public class DrawSquare {
5      public static void main(String[] args) {
6          SimpleWindow w = new SimpleWindow(600, 600, "DrawSquare");
7          Square sq = new Square(250, 250, 100);
8          sq.draw(w);
```

```
9       }
10  }
```

- På rad 1 och rad 2 importeras de klasser som behövs, i detta fall: klassen `SimpleWindow` och klassen `Square`. De klasserna finns inte i Javas standardbibliotek utan har skrivits speciellt för kurserna i programmeringsteknik. Klasserna finns i ”paket” vars namn börjar med `se.lth.cs`; det betyder att de är utvecklade vid institutionen för datavetenskap vid LTH, som använder domännamnet `cs.lth.se`.
- På rad 6 skapas ett `SimpleWindow`-objekt. Refererensvariabeln `w` refererar till detta objekt.
- Därefter skapas ett `Square`-objekt som referensvariabeln `sq` refererar till.
- Slutligen ritas kvadraten `sq`.

Notera parametrarna som man använder när man skapar objekten. I `new`-uttrycket som skapar kvadratobjektet står det till exempel (250, 250, 100). Det betyder att kvadraten ska ha läget 250, 250 och sidlängden 100. Läget och sidlängden kan man ändra senare i programmet, med `sq.move(dx,dy)` och `sq.setSide(newSide)`.

Lägg också märke till att referensvariablerna `w` och `sq` har olika typer. En referensvariabels typ avgör vilka slags objekt variabeln får referera till. `w` får referera till `SimpleWindow`-objekt och `sq` får referera till `Square`-objekt.

Användning av klassen SimpleWindow

Bläddra fram källkoden till klassen `Square`. Du hittar klassen i Eclipse-projektet *cs_pt* under katalogen *src* och i paketet `se.lth.cs.pt.square`. Titta på källkoden och försök förstå vad som händer. Några av metoderna i `SimpleWindow` (metoderna för att flytta pennan och för att rita linjer) utnyttjas i metoden `draw` i klassen `Square`.

`SimpleWindow` innehåller också metoder, som inte används i `Square`, t.ex. för att ta hand om musklick. Dessa metoder har följande beskrivning:

```
/** Väntar tills användaren har klickat på en musknapp. */
void waitForMouseClick();

/** Tar reda på x-koordinaten för musens position vid senaste musklick. */
int getClickedX();

/** Tar reda på y-koordinaten för musens position vid senaste musklick. */
int getClickedY();
```

När exekveringen av ett program kommer fram till `waitForMouseClick` så ”stannar” programmet och fortsätter inte förrän man klickat med musen någonstans i programmets fönster. Ett program där man skapar ett fönster och skriver ut koordinaterna för varje punkt som användaren klickar på har följande utseende:

```
import se.lth.cs.pt.window.SimpleWindow;

public class PrintClicks {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "PrintClicks");
        while (true) {
            w.waitForMouseClick();
            w.moveTo(w.getClickedX(), w.getClickedY());
            w.writeText("x = " + w.getClickedX() + ", " + "y = " + w.getClickedY());
        }
    }
}
```

`while (true)` betyder att repetitionen ska fortsätta ”i oändlighet”. Man avbryter programmet genom att välja Quit i File-menyn i `SimpleWindow`-fönstret.

Också i nedanstående program ska användaren klicka på olika ställen i fönstret. Nu är det inte koordinaterna för punkten som användaren klickar på som skrivs ut, utan i stället avståndet mellan punkten och den förra punkten som användaren klickade på. Vi sparar hela tiden koordinaterna för den förra punkten (variablerna `oldX` och `oldY`).

Gå igenom ett exempel ”för hand” och övertyga dig om att du förstår hur programmet fungerar.

```
import se.lth.cs.pt.window.SimpleWindow;

public class PrintClickDists {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "PrintClickDists");
        int oldX = 0; // x-koordinaten för "förra punkten"
        int oldY = 0; // y-koordinaten
        while (true) {
            w.waitForMouseClick();
            int x = w.getClickedX();
            int y = w.getClickedY();
            w.moveTo(x, y);
            int xDist = x - oldX;
            int yDist = y - oldY;
            w.writeText("Avstånd: " + Math.sqrt(xDist * xDist + yDist * yDist));
            oldX = x;
            oldY = y;
        }
    }
}
```

Datorarbete

1. Logga in på datorn, starta Eclipse, öppna projektet *Lab02*. (Logga in, starta Eclipse och öppna ett projekt ska du alltid göra, så det skriver vi inte ut i fortsättningen.)

2. Öppna en webbläsare och gå till kursens hemsida, `cs.lth.se/edaa5x` . Under Dokumentation finns en länk till dokumentationen av de färdigskrivna klasser som används under laborationerna, alltså klasserna i paketet `se.lth.cs.pt`. Leta upp och titta igenom specifikationen av klassen `Square`. Det är samma specifikation som finns under Bakgrund, fast i något annorlunda form.

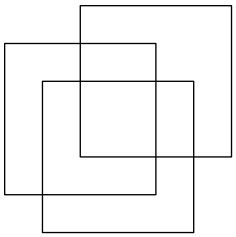
3. Klassen `DrawSquare` finns i filen *DrawSquare.java*. Öppna filen, kör programmet.

4. Kopiera filen *DrawSquare.java* till en ny fil med namnet *DrawThreeSquares.java*. Enklast är att göra så här:

   1. Markera filen i projektvyn, högerklicka, välj Copy.
   2. Högerklicka på *(default package)*, välj Paste, skriv det nya namnet på filen.

   Notera att Eclipse ändrar klassnamnet i den nya filen till `DrawThreeSquares`.

   Ändra sedan klassen så att kvadraten ritas tre gånger. Mellan uppritningarna ska kvadraten flyttas. Du ska fortfarande bara skapa ett kvadratobjekt i programmet. Resultatet ska bli en figur med ungefär följande utseende:

   Testa programmet, rätta eventuella fel.

5. Någonstans i ditt program skapas ett kvadratobjekt. Det görs i en sats som har ungefär följande utseende: `Square sq = new Square(300,300,200)`. Ta bort denna sats från programmet (eller kommentera bort den). Eclipse kommer att markera flera fel i programmet, eftersom `sq` inte är deklarerad och du senare i programmet utnyttjar den variabeln. Läs och tolka felmeddelandena.

   Lägg sedan in satsen `Square sq = null` i början av programmet. Eclipse kommer inte att hitta några fel, eftersom programmet nu följer de formella reglerna för Javaprogram. Exekvera sedan programmet och se vad som inträffar. Studera felmeddelandet så att du kan tolka det.

   Meddelanden om exekveringsfel skrivs i konsolfönstret. Det skrivs också ut en ”stack trace” för felet: var felet inträffade, vilken metod som anropade metoden där det blev fel, vilken metod som anropade den metoden, osv. Om man klickar på ett radnummer öppnas rätt fil i editorn med den raden markerad (under förutsättning att programtexten, ”källkoden”, för den filen är tillgänglig).

   Lägg sedan tillbaka den ursprungliga satsen för att skapa kvadratobjektet. Ändra argumentet `w` i det första anropet av `draw` till `null`. Kör programmet, studera det felmeddelande som du får.

   När man får exekveringsfel kan det vara svårt att hitta orsaken till felet. Här är en debugger till stor hjälp, i och med att man kan sätta brytpunkter och köra programmet stegvis.

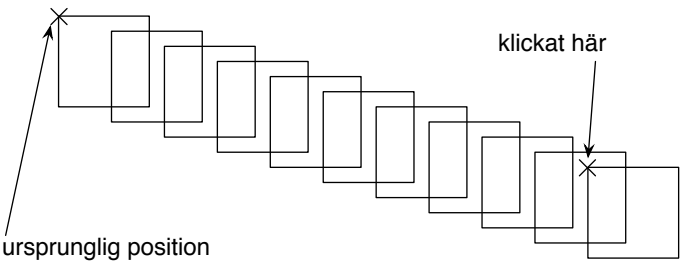
6. Kör programmen `PrintClicks` och `PrintClickDists`.

7. Skriv ett program, i en ny fil (men kopiera gärna din föregående eller något av exemplen, se nästa stycke), där ett kvadratobjekt skapas och ritas upp i ett ritfönster. När användaren klickar med musen i fönstret ska bilden av kvadraten raderas, kvadraten flyttas till markörens position och ritas upp på nytt. Titta på dokumentationen av SimpleWindow och Square som finns länkad från kurshemsidan. Vilka metoder som verkar användbara för uppgiften kan du hitta där?

   Välj ett lämpligt namn på klassen. För att skapa en fil för klassen kan du antingen skapa en tom klass (klicka på New Java Class-ikonen, skriv namnet på klassen, klicka på Finish) eller kopiera någon av de filer som du redan har.

Testa programmet.

8. Skapa en ny klass, ge den ett valfritt namn, och klistra därefter in ditt program från 7. Modifiera programmet i den nya filen så att varje flyttning går till så att kvadraten flyttas stegvis till den nya positionen. Efter varje steg ska kvadraten ritas upp (utan att den gamla bilden raderas). Exempel på kvadratbilder som ritas upp när flyttningen görs i 10 steg:



Kvadratens slutliga position behöver inte bli exakt den position som man klickat på. Om man till exempel ska flytta kvadraten 94 pixlar i 10 steg är det acceptabelt att ta 10 steg med längden 9.

Skriv in och testkör programmet.

9. Programmet från uppgift 8 ritar många bilder av samma kvadrat och alla bilderna kommer att synas i fönstret när programmet är slut. Om man istället raderar den ”gamla” bilden innan man ritar en ny bild så kommer man att få en ”rörlig”, ”animerad”, bild. För att man ska se vad som händer måste man då göra en paus mellan uppritning och radering — det gör man med `SimpleWindow`-metoden `delay`, som har en parameter som anger hur många millisekunder man ska vänta.

Kopiera koden från uppgift 8 till en ny fil `AnimatedSquare.java`. Lägg in fördröjning och radering så att kvadratbilden blir ”animerad”. (Använd *inte* `SimpleWindow`-metoden `clear`.)

Följande exempel visar en enkel animering, och kan tjäna som inspiration:

```
while (sq.getSide() > 0) {
    sq.draw(w);
    SimpleWindow.delay(10);
    sq.erase(w);
    sq.setSide(sq.getSide() - 10);
}
```

Kodsnutten ovan animerar en ny mindre kvadrat tills dess att kvadratens sida blivit mindre än eller lika med noll. Men det är alltså inte detta du ska göra: du ska istället animera en förflyttning av en kvadrat som är lika stor hela tiden.

Anmärkning: i ett ”riktigt” program som visar rörliga bilder åstadkommer man inte animeringen på det här sättet. Denna lösning har bristen att man inte kan göra något annat under tiden som animeringen pågår, till exempel kan programmet inte reagera på att man klickar med musen.

10. I denna uppgift ska du lära dig fler kommandon i Eclipse. Det finns ett otal kommandon, mer eller mindre avancerade, och många av dem använder man sällan. Två kommandon som man ofta använder:

- Source-menyn > Format. Korrigerar programlayouten i hela filen. Ser till exempel till att varje sats skrivs på en rad, att det är blanka runt om operatorer, och så vidare.

Man bör formatera sina program regelbundet, så att de blir läsbara. Observera att programmet ska vara korrekt formaterat när du visar det för labhandledaren för att få det godkänt. Notera också kortkommandot som står intill menyalternativet. Prova att använda kortkommandot också, och lär dig gärna det, då det är snabbare och mer bekvämt än att klicka.

- Refactor-menyn > Rename. Ändrar namn på den variabel eller metod som markerats (lokalt om det är en lokal variabel, i hela klassen om det är ett attribut, även i andra klasser om det är en publik metod).

11. Kontrollera med hjälp av checklistan nedan att du behärskar de olika momenten i laborationen. Diskutera med övningsledaren om någonting är oklart.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Tyda specifikationer av klasser.
- Skriva program som använder färdiga klasser:
  - Deklarera referensvariabler och skapa objekt.
  - Anropa metoder på objekt.

Laboration 3 – blockmullvad

Mål: Du ska lära dig mer om att strukturera kod genom att skriva och använda egna klasser. Du får öva på att skriva metoder och deklarera attribut. Du får även använda statiska variabler.

Förberedelseuppgifter

- Påbörja gärna datorarbetet nedan.

Bakgrund

Blockmullvad (*Talpa laterculus*) är ett fantasidjur i familjen mullvadsdjur. Den är känd för sitt karaktäristiska kvadratiska utseende. Den lever mest ensam i sina underjordiska gångar som till skillnad från mullvadens (*Talpa europaea*) har helt raka väggar.

Datorarbete

1. Du ska själv bygga upp ditt Java-program steg för steg i Eclipse men projektet finns förberett i det workspace som använts på tidigare laborationer.

   a) Skapa filen `Mole.java` genom att:

      a) Högerklicka på projektet *Lab03* i projektvyn,

      b) Expandera New i menyn och välj Class.

      c) Skriv namnet på klassen (`Mole`).

      d) Klicka på Finish.

   b) Öppna din nya klass (om det inte redan gjorts automatiskt). Lägg till en `main`-metod i klassen som skriver ut texten *Keep on digging!* med hjälp av `System.out.println`.

   c) Testa att köra ditt program. Om allt går bra ska texten du angivit skrivas ut i konsolfönstret i Eclipse.

   d) Nu har du skrivit ett Java-program som skriver ut en uppmaning till en mullvad att fortsätta gräva. Det programmet är inte så användbart, eftersom mullvadar inte kan inte läsa. Nästa steg är att skriva ett grafiskt program, snarare än ett textbaserat.

   Funktionen `println` som anropas i `main`-funktionen ingår i Javas standardbibliotek. Ett programbibliotek innehåller kod som kan användas av andra program, och för de flesta programspråk ingår ett standardbibliotek som alla program kan nyttja. Till grafiken i denna uppgift ska du, precis som i tidigare laborationer, använda den färdiga klassen `SimpleWindow`. Den finns i biblioteket *cs_pt* i ditt workspace.

   e) Låt `main`-metoden innehålla följande satser:

```
SimpleWindow w = new SimpleWindow(300, 500, "Digging");
w.moveTo(10, 10);
w.lineTo(10, 20);
w.lineTo(20, 20);
w.lineTo(20, 10);
w.lineTo(10, 10);
```

   Överst i filen måste du ha raden

```
import se.lth.cs.pt.window.SimpleWindow;
```

   för att klassen `SimpleWindow` ska bli tillgänglig.

Koden känner du nog igen från tidigare i kursen. Den första raden skapar ett nytt `SimpleWindow` som ritar upp ett fönster som är 300 bildpunkter brett och 500 bildpunkter högt med titeln *Digging*. `SimpleWindow` har en *penna* som kan flyttas runt och rita linjer. Anropet `w.moveTo(10, 10)` flyttar pennan för fönstret `w` till position $(10, 10)$ utan att rita något, och anropet `w.lineTo(10, 20)` ritar en linje därifrån till position $(10, 20)$.

f) Kör ditt program. Du ska nu få upp ett fönster med en liten kvadrat utritad i övre vänstra hörnet.

2. Hela ditt program är för tillfället samlat i en och samma metod, vilket fungerar bra för väldigt små program. Nu ska vi strukturera om programmet så det blir lättare att utöka senare.

a) Skapa en ny klass med namnet `Graphics` (se punkt a i föregående uppgift om du glömt hur man skapar en klass) och flytta dit deklarationen av fönstret `w` så att det blir ett attribut i klassen. Skapa en ny metod i klassen `Graphics` med namnet `square`, och flytta dit koden som ritar kvadraten.

Filen `Graphics.java` ska se ut såhär:

```
import se.lth.cs.pt.window.SimpleWindow;

public class Graphics {
    private SimpleWindow w = new SimpleWindow(300, 500, "Digging");

    public void square(){
        // Fyll i koden för att rita en kvadrat här.
        // Observera att w är definerat ovan.
    }
}
```

Nu har vi beskrivit vad klassen `Graphics` ska innehålla och hur den ska fungera. Men för att faktiskt använda den måste vi göra något mer. Vi måste anropa metoden `square` i `main`-metoden. Metoden `square` finns i klassen `Graphics`, och därför måste ett `Graphics`-objekt skapas först. Därefter använder vi punktnotation `square`-metoden genom att med punktnotation ange att det är en metod inuti `Graphics`-objektet som anropas.

Filen `Mole.java` ska se ut såhär:

```
public class Mole {

    public static void main(String[] args) {
        Graphics g = new Graphics();
        g.square();
    }

}
```

b) Kör programmet, om allt fungerar ska programmet göra samma sak som i föregående uppgift.

c) Kommentera bort raden `g.square();` genom att sätta `//` framför. Kör programmet igen och se vad som händer. Varför är det så?

d) Ta bort kommentartecknen vid `g.square();` igen och kommentera nu istället bort raden `Graphics g = new Graphics();`. Vad händer då och varför? Diskutera med

din labhandledare om det känns oklart. Ta bort kommentaren och kontrollera att programmet fungerar innan du går vidare.

3. Nu ska du skapa ett nytt koordinatsystem för `Graphics` som har *stora* bildpunkter. Vi kallar `Graphics` stora bildpunkter för *block* för att lättare skilja dem från `SimpleWindows` bildpunkter. Om blockstorleken är b, så ligger koordinaten (x, y) i `Graphics` på koordinaten (bx, by) i `SimpleWindow`.

   a) Lägg till följande deklarationer som attribut överst i klassen `Graphics`.

   ```
   private int width;
   private int blockSize;
   private int height;
   ```

   b) Nu vill vi att våra attribut ska få startvärden. Ett heltalsattribut är noll om inget annat anges. Om man vill att ett attribut alltid ska få samma startvärde så går det att tilldela värdet direkt vid deklarationen (alltså t.ex. `private int width = 30;`). Men i denna laborationen ska vi öva på att ge attributen dess startvärden via konstruktorn. På så vis kan den som skapar objektet själv välja vilka värden attributen ska få.

   En konstruktor har alltid samma namn som klassen. Skapa en konstruktor med heltalsparametrarna `w`, `h` samt `bs`. Låt värdet av `w` tilldelas till attributet `width`, värdet av `h` tilldelas till attributet `height` och värdet av `bs` tilldelas till attributet `blockSize`.

   c) Ändra bredden på ditt `SimpleWindow` till `width * blockSize` och ändra höjden till `height * blockSize`. Detta betyder att fönstret måste skapas i konstruktorn, men det måste ändå deklareras som attribut. Nu ska din klass alltså ha följande struktur:

   ```
   import se.lth.cs.pt.window.SimpleWindow;

   public class Graphics {
       private int width;
       private int blockSize;
       private int height;

       private SimpleWindow w;

       public Graphics(int w, int h, int bs){

           // Lägg till satser för att initiera width, blocksize och height

           this.w = new SimpleWindow(width * blockSize,
                                     height * blockSize,
                                     "Digging");
       }

       // Metoden square som du skrivit tidigare
   }
   ```

   d) Nu har du ändrat i `Graphics`. Bland annat har du ändrat hur objekt av klassen skapas. Detta kräver att du gör motsvarande ändring i klassen `Mole`. Nu måste du skicka in värden på varje parameter (`w`, `h` och `bS`) – dessa värden kallas argument. Ändra därför så att `main` nu ser ut så här i stället:

   ```
   public class Mole {
       public static void main(String[] args) {
           Graphics g = new Graphics(30,50,10);
           g.square();
       }
   }
   ```

Provkör och kontrollera att programmet fortfarande fungerar som innan.

e) Innan vi gjorde en egen konstruktor kunde vi ändå skapa ett nytt `Graphics`-objekt, men då utan att ange några värden. Varför? (Hitta ledtrådar genom att googla ”Java default constructor”).

f) Skapa en ny metod i `Graphics` med namnet `block` och två parametrar `x` och `y` av typen `int` och returtypen `void`. Metodens *kropp* ska se ut såhär:

```
int left = x * blockSize;
int right = left + blockSize - 1;
int top = y * blockSize;
int bottom = top + blockSize - 1;
for(int row = top; row <= bottom; row ++){
    w.moveTo(left, row);
    w.lineTo(right, row);
}
```

g) Metoden `block` ritar ett antal linjer. Hur många linjer ritas ut? I vilken ordning ritas linjerna?

h) Anropa funktionen `block` några gånger i `main`-metoden så att några olika block ritas upp i fönstret när programmet körs. Kör ditt program och kontrollera resultatet.

4. Det finns många sätt att beskriva färger. I naturligt språk har vi olika namn på färgerna, till exempel *vitt*, *rosa* och *magenta*. I datorn är det vanligt att beskriva färgerna som en blandning av *rött*, *grönt* och *blått* i det så kallade RGB-systemet. `SimpleWindow` använder typen `java.awt.Color` för att beskriva färger och `java.awt.Color` bygger på RGB. Det finns några fördefinierade färger i `java.awt.Color`, till exempel `java.awt.Color.BLACK` för svart och `java.awt.Color.GREEN` för grönt. Andra färger kan skapas genom att ange mängden rött, grönt och blått.

a) Skapa en ny klass med namnet `Colors` och lägg in följande definitioner:

```
public static final Color MOLE = new Color(51, 51, 0);
public static final Color SOIL = new Color(153, 102, 51);
public static final Color TUNNEL = new Color(204, 153, 102);
```

För att använda klassen `java.awt.Color` är det enklast att importera den genom att skriva `import java.awt.Color;` överst i filen.

Den tre parametrarna till `new Color(r, g, b)` anger hur mycket *rött*, *grönt* respektive *blått* som färgen ska innehålla, och mängderna ska vara i intervallet 0–255. Färgen $(153, 102, 51)$ innebär ganska mycket rött, lite mindre grönt och ännu mindre blått och det upplevs som brunt. Klassen `Colors` fungerar här som en färgpalett, men vi har inte ritat något med färg ännu. Kompilera och kör ditt program ändå, för att se så programmet fungerar likadant som sist.

b) Lägg till en parameter till metoden `block` i klassen `Graphics`. Skriv dit den sist i parameterlistan med namnet `color` och typen `java.awt.Color`. För att ändra färgen på blocket, till den färg som angivits som parameter, ska du byta linjefärg innan du ritar. Lägg till följande rad i början av metoden `block`:

```
w.setLineColor(color)
```

c) Ändra i `main` och lägg till en av färgerna från klassen `Colors` som tredje argument i dina anrop till `block`. Eftersom de färger du har definerat i ditt program

är publika (`public`) och statiska (`static`) i klassen Colors behöver vi denna gången inte skapa något objekt. Ett anrop till metoden `block` kan därför se ut så här: `g.block(10,2,Colors.MOLE);` Kompilera och kör ditt program och upplev världen i färg.

Ytterligare förklaring: Normalt sett har varje objekt sina egna värden på varje attribut (som i fallet med `width`, `height`, `blockSize` och `w` i klassen `Graphics`) men `static` betyder att det bara finns ett värde i hela programmet (det är alltså inte unikt per objekt). `public` betyder att det går att komma åt värdena även från andra klasser och final betyder att det aldrig kan ändras under programmets gång. Detta betyder att vi har definierat färgerna som tre globala konstanter. Konventionen i Java är att konstanter namnges med enbart versaler. Punktnotation används alltid för att komma åt saker i en annan klass eller ett annat objekt, och därför skriver vi `Colors.MOLE` för att komma åt färgen.

Tips: undvik att ange attribut som `public` eller `static` om du inte är riktigt säker på din sak. De flesta attribut framöver i kursen är privata och icke-statiska.

d) Eftersom attributen i klassen `Graphics` är privata kan vi inte komma åt dessa från klassen `Mole`. Men ofta finns det behov av att i efterhand komma åt attributens värden. I nästa uppgift kommer vi behöva komma åt bredden och höjden därför ska vi nu implementera get-metoder för dessa attribut.

En get-metod har ofta samma namn som attributet men med `get` framför. Då en metod består av flera ord låter man första ordet börja med liten bokstav (eftersom metoder alltid ska börja med liten bokstav) och alla efterföljande ord börja med stor bokstav (för att det ska bli lättare att läsa). Vidare brukar varje get-metod enbart returnera attributets värde, det betyder att get-metodernas returtyp ska vara samma som attributets typ. Det leder oss fram till att vi behöver följande i klassen `Graphics`:

```
public int getWidth() {
    return width;
}

public int getHeight() {
    return height;
}
```

Fråga din handledare om du behöver ytterligare förklaring.

5. Nu ska du skriva en funktion för att rita en rektangel och rektangeln ska ritas med hjälp av funktionen `block`. Sen ska du rita upp mullvadens underjordiska värld med hjälp av denna funktion.

a) Lägg till en metod i klassen `Graphics` med namnet `rectangle`. Metoden ska ta fem parametrar: `x`, `y`, `width` och `height` av typen `int`, samt `c` av typen `Color`.

Parametrarna `x` och `y` anger `Graphics`-koordinaten för rektangelns övre vänstra hörn och `width` och `height` anger bredden respektive höjden på rektangeln. Använd följande `for`-satser för att rita ut rektangeln.

```
for (int yy = y; yy < y + height; yy++){
    for(int xx = x; xx < x + width; xx++){
        block(xx, yy, c);
    }
}
```

b) I vilken ordning ritas blocken ut?

c) Skriv en metod i klassen `Mole` med namnet `drawWorld` som ritar ut mullvadens värld, det vill säga en massa jord där den kan gräva sina tunnlar. `drawWorld` ska inte ha några parametrar, returtypen ska vara `void`, och metoden ska anropa `rectangle` för att rita en rektangel med färgen `Colors.SOIL` som precis täcker fönstret. Anropa `drawWorld` i `main`-metoden och testa så att det fungerar genom att köra programmet.

Ledning: Eftersom funktionen `rectangle` finns i klassen `Graphics` så måste vi i klassen `Mole` ha tillgång till det `Graphics`-objekt som ska användas. Vi lägger det därför som attribut i klassen Mole. Eftersom attributet och metoden `drawWorld` inte är statiska måste vi då skapa ett objekt av typen `Mole` för att sen kunna använda metoderna. (Exekveringen startar ju i `main`, som är statisk, därför finns inget `Mole`-objekt från början, trots att vi är inuti klassen.) `Mole` måste alltså att se ut så här:

```
public class Mole {
    private Graphics g = new Graphics(30, 50, 10);

    public static void main(String[] args) {
        Mole m = new Mole();
        m.drawWorld();
    }

    public void drawWorld(){
        // Kod som ritar upp världen med hjälp av g.rectangle...
    }
}
```

Eftersom vi inte definerat någon konstruktor i klassen `Mole` är det återigen default-konstruktorn som anropas (vilket är tillräckligt för denna uppgift eftersom inga speciella värden måste tilldelas i konstruktorn).

6. I `SimpleWindow` finns en metod för att känna av tangenttryckningar. Du ska använda denna för att styra en liten blockmullvad.

a) Lägg till följande metod i klassen `Graphics`:

```
public char waitForKey() {
    return w.waitForKey();
}
```

I `Graphics`-klassen finns alltså nu en metod för att invänta en tangenttryckning. Den har vi implementerat genom att anropa metoden med samma namn i vårt `SimpleWindow`-objekt `w`.

Man kan säga att klassen `Graphics` *delegerar* uppgiften till `SimpleWindow`. Ser du varför det är en passande term?

7. Nu är det dags att få mullvaden att gräva på riktigt.

   a) Lägg till en funktion i klassen `Mole` med namnet `dig`, utan parametrar och med returtypen `void`. Funktionens kropp ska se ut såhär (fast utan `/* TODO */`):

   ```
   int x = g.getWidth() / 2;    // För att börja på mitten
   int y = g.getHeight() / 2;
   while (true) {
       g.block(x, y, Colors.MOLE);
       char key = g.waitForKey();

       if (key == 'w') { /* TODO */ }
       else if (key == 'a') { /* TODO */ }
       else if (key == 's') { /* TODO */ }
       else if (key == 'd') { /* TODO */ }
   }
   ```

   Byt ut i alla `/* TODO */` mot Java-satser så att `'w'` styr mullvaden ett steg uppåt, `'a'` ett steg åt vänster, `'s'` ett steg nedåt och `'d'` ett steg åt höger.

   b) Ändra `main` så att den anropar både `drawWorld` och `dig`. Kompilera och kör ditt program för att se om programmet reagerar på knapptryck på w, a, s och d.

   c) Om programmet fungerar kommer det bli många mullvadsfärgade block som tillsammans bildar en lång mask, och det är ju lite underligt. Lägg till ett anrop i `dig` som ritar ut en bit tunnel på position (x, y) efter anropet till `waitForKey` men innan `if`-satserna. Kompilera och kör ditt program för att gräva tunnlar med din blockmullvad.

Frivilliga extrauppgifter

8. Mullvaden kan för tillfället gräva sig utanför fönstret. Lägg till några `if`-satser i början av `while`-satsen som upptäcker om `x` eller `y` ligger utanför fönstrets kant och flyttar i så fall tillbaka mullvaden precis innanför kanten.

9. Mullvadar är inte så intresserade av livet ovanför jord, men det kan vara trevligt att se hur långt ner mullvaden grävt sig. Lägg till en himmelsfärg och en gräsfärg i objektet `Colors` och rita ut himmel och gräs i `drawWorld`. Justera också det du gjorde i föregående uppgift, så mullvaden håller sig under jord. (*Tips:* Den andra parametern till `Color` reglerar mängden grönt och den tredje parametern reglerar mängden blått.)

10. Ändra så att mullvaden kan springa uppe på gräset också, men se till så att ingen tunnel ritas ut där.

Checklista

I den här laborationen har du övat på att

- Bygga upp ett program steg för steg och testa ofta
- Strukturera vårt program med klasser och metoder
- Skriva egna klasser och metoder
- Skapa och anropa en egen konstruktor
- Anropa default-konstruktorn
- Definera och använda statiska konstanter
- Skriva get-metoder

Laboration 4 – eget spel

Mål: Du ska skriva ett eget program. Du ska träna på att deklarera och använda variabler, att göra anrop mellan klasser samt att använda alternativ och repetition.

Förberedelseuppgifter

- Läs igenom texten i avsnittet Bakgrund.
- Hitta på ett spel och bestäm hur det ska fungera.
- Skissa ditt program (gärna pseudokod) på papper.
- Skapa klasserna ditt program ska innehålla (enligt uppgift 2 under datorarbete nedan)
- Börja skriva kod så att du (minst) har ett minimalt program som går att köra när laborationen börjar. Men du får givetvis gärna gör mer, för att säkert hinna klart!

Bakgrund

På den här laborationen ska du skriva ett textbaserat spel. Du ska själv hitta på hur det ska fungera. Endast fantasin sätter gränser, det måste inte vara ett spel som redan finns. Vill du skriva något annat program, som inte är ett spel, så går det också bra. Stäm av med din lärare eller handledare om du är osäker. Känns det begränsat med textbaserat? Det går att göra mer än vad man tror. Googla gärna ”textspel” för inspiration.

Krav på programmet

- Utveckla ditt program stegvis. Börja med ett program som bara gör något litet. Då har du tidigt ett program som fungerar. Bygg sedan ut det efterhand.
- Ditt program ska bestå av minst två klasser. Det innebär minst en klass innehållande (minst) en main-metod och en annan klass med två eller fler metoder. Klassen med main-metoden ska skapa och använda ett objekt av den andra klassen.
- Ditt program måste innehålla minst ett alternativ (`if-sats`) och minst en repetition (`for-` eller `while`-sats).
- Din kod ska vara lätt att läsa och förstå (bra val av klass- och variabelnamn, vettig indentering etc).
- Det är en konst att begränsa sig. Se till att du inte tar dig vatten över huvudet. Kontrollera gärna din idé med labhandledaren.

Inspiration

Här finns några förslag att hämta inspiration ifrån. Du kan använda något av förslagen, modifiera något av förslagen eller hitta på något helt eget spel.

- Spela ”gissa talet” och ge ledtrådar om talet är för litet eller för stort.
- Hitta på något enkelt tärningsspel att spela med användaren.
- Träna användaren i multiplikationstabellen.
- Rita många kvadrater i ett fönster och låta användaren gissa antalet.
- Kolla reaktionstiden hos användaren genom att mäta tiden det tar att trycka Enter efter att man fått vänta en slumpmässig tid på att strängen ”NU!” skrivs ut. Om man trycker Enter innan startutskriften ges blir den uppmätta tiden 0 och på så sätt kan ditt program detektera att användaren har tryckt för tidigt. Mät reaktionstiden upprepade gånger och beräkna medelvärdet.
- Låt användaren svara på flervalsfrågor om din favoritfilm.
- Spela sten/sax/påse med användaren.
- Låt användaren på tid så snabbt som möjligt skriva olika ord baklänges.

- Ett textbaserat äventyrsspel där användaren får gå igenom en värld och göra val, svara på gåtor etc.

Några användbara metoder och hjälpklasser

Här följer några tips på hur du kan jämföra strängar, dra slumptal och hantera tid i Java. Vi kräver inte att du använder alla dessa finesser, men du har troligen nytta av något av dem.

Strängjämförelse: använd metoden equals för att jämföra om två strängar är lika. **Exempel:**

```
String svar = scan.next();
if (svar.equals("nej")) {
    // ... något händer ...
}
```

Ibland vill man inte att det ska spela någon roll ifall användaren har skrivit med små eller stora bokstäver. Då kan man istället använda `equalsIgnoreCase`:

```
String svar = scan.next();
if (svar.equalsIgnoreCase("nej")) {
    // ... något händer ...
}
```

Slumptal: använd ett `Random`-objekt för att ge variabler slumpmässiga värden:

`Random`

```
/** Skapar en slumptalsgenerator. */
Random();

/** Returnerar ett slumptal mellan 0 och bound-1. */
int nextInt(int bound);

/** Returnerar ett slumptal mellan 0.0 och 1.0. */
double nextDouble();
```

Exempel:

```
Random rand = new Random();
int nbr = rand.nextInt(10);  // nbr får slumpmässigt värde mellan 0 och 9
```

Mäta tid: använd den statiska metoden `currentTimeMillis` i klassen `System` för att mäta tid.

`System`

```
/** Returnerar aktuell tid i millisekunder, sedan någon ospecificerad
    startpunkt. Differensen mellan två värden anger hur lång tid som
    förflutit. */
static long currentTimeMillis();
```

Exempel:

```
long start = System.currentTimeMillis();
// ... här görs något som vi vill mäta tiden för ...
long end = System.currentTimeMillis();
long elapsed = end - start;
```

Fördröjning: för att låta programmet vänta en viss tid kan du använda klassen `Timer`, som finns i paketet `se.lth.cs.pt.timer`. Du importerar klassen på vanligt sätt, men notera att det

finns några andra Java-klasser med samma namn, så var säker på att du väljer rätt paket.[1]

`Timer`

```
/** Fördröj programmets exekvering med ett antal millisekunder. */
static void delay(long milliseconds);
```

Exempel:

```
Timer.delay(2000);  // Programmet väntar här i 2 sekunder
```

Datorarbete

1. Logga in på datorn, starta Eclipse, öppna projektet *Lab04*.

2. Skapa en fil med namnet *SmallGame.java* (eller något namn du väljer själv). För att skapa en fil ska du

   - markera projektet *Lab04*
   - klicka på New Java Class-ikonen eller högerklicka och välja New och därefter Class
   - skriva namn på klassen (t.ex. `SmallGame`)
   - klicka på Finish

   Skapa sedan ytterligare en klass, som ligger i en annan fil. I denna klass ska din main-metod ligga. Det går bra att döpa klassen till *Main.java* (eller något annat namn du själv väljer).

3. Skriv ditt program. Testa och se till att spelet fungerar som avsett. Tänk på att utveckla ditt spel i små steg, så att du kan provköra ofta.

4. Låt någon annan kursdeltagare läsa ditt program och köra spelet. Notera den återkoppling du får på programmet. På samma sätt ska du ge återkoppling på en annan kursdeltagares program.

   Exempel på frågor man kan ställa sig är:

   - Är spelet svårt eller lätt? Är det kul? Går det som användare att begripa vad man förväntas göra?
   - Är det lätt att förstå programkoden? Är det något som är särskilt klurigt? Lärde du dig något nytt genom att läsa programmet?
   - Hur ser uppdelningen i klasser ut? Vilka är de olika klassernas uppgifter?

5. Använd den återkoppling du fick för att förbättra ditt program.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Skapa .java-filer och skriva enkla program i Eclipse.
- Förstå hur klasser kan interagera.
- Förstå vad som händer när metoder anropas.
- Skriva programkod med alternativ (`if`-sats).
- Skriva programkod med repetition (`for`- eller `while`-sats).

[1] Din `import`-sats ska se ut så här: `import se.lth.cs.pt.timer.Timer;`

Laboration 5 – implementera klasser, Turtle

Mål: Du ska fortsätta att träna på att implementera och använda klasser. Du ska också få mer träning i att använda Eclipse debugger.

Förberedelseuppgifter

- Tänk igenom följande frågor och se till att du kan svara på dem:
  1. Vad är en specifikation respektive en implementering av en klass?
  2. Vad är ett attribut? Var deklareras attributen? När reserveras det plats för dem i datorns minne och hur länge finns de kvar?
  3. När exekveras satserna i konstruktorn? Vad brukar utföras i konstruktorn?
  4. Vilka likheter/skillnader finns det mellan attribut, lokala variabler och parametrar.
  5. Vad menas med public och private?
- Läs avsnittet Bakgrund.

Bakgrund

Turtle graphics är en teknik för att rita linjer. Linjerna ritas av en (tänkt) sköldpadda som går omkring i ett ritfönster. Sköldpaddan har en penna som antingen kan vara sänkt (då ritas en linje i sköldpaddans spår) eller lyft (då ritas ingen linje). Sköldpaddan kan bara gå rakt framåt, i den riktning som huvudet pekar, men när den står stilla kan den vända sig i en ny riktning.

En klass `Turtle` som beskriver en sköldpadda av detta slag har följande specifikation:

```
/** Skapar en sköldpadda som ritar i ritfönstret w. Från början
    befinner sig sköldpaddan i punkten x,y med pennan lyft och
    huvudet pekande rakt uppåt i fönstret (i negativ y-riktning). */
Turtle(SimpleWindow w, int x, int y);

/** Sänker pennan. */
void penDown();

/** Lyfter pennan. */
void penUp();

/** Går rakt framåt n pixlar i den riktning som huvudet pekar. */
void forward(int n);

/** Vrider beta grader åt vänster runt pennan. */
void left(int beta);

/** Går till punkten newX,newY utan att rita. Pennans läge (sänkt
    eller lyft) och huvudets riktning påverkas inte. */
void jumpTo(int newX, int newY);

/** Återställer huvudriktningen till den ursprungliga. */
void turnNorth();

/** Tar reda på x-koordinaten för sköldpaddans aktuella position. */
int getX();

/** Tar reda på y-koordinaten för sköldpaddans aktuella position. */
int getY();

/** Tar reda på sköldpaddans riktning, i grader från positiv x-led. */
int getDirection();
```

Observera att vi här bestämmer vilket fönster som sköldpaddan ska rita i när vi skapar ett sköldpaddsobjekt. Det kan väl sägas motsvara verkligheten: en sköldpadda ”befinner” sig ju alltid någonstans.

I följande program ritar en sköldpadda en kvadrat med sidorna parallella med axlarna:

```
import se.lth.cs.pt.window.SimpleWindow;

public class TurtleDrawSquare {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "TurtleDrawSquare");
        Turtle t = new Turtle(w, 300, 300);
        t.penDown();
        for (int i = 0; i < 4; i++) {
            t.forward(100);
            t.left(90);
        }
    }
}
```

I nedanstående variant av programmet har vi gjort två ändringar: 1) längden på sköldpaddans steg väljs slumpmässigt mellan 0 och 99 pixlar, 2) efter varje steg görs en paus på 100 millisekunder. Den första ändringen medför att figuren som ritas inte blir en kvadrat, den andra ändringen medför att man ser hur varje linje ritas.

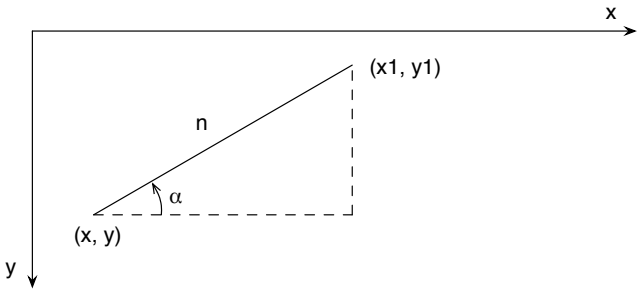
```
import se.lth.cs.pt.window.SimpleWindow;
import java.util.Random;

public class TurtleDrawRandomFigure {
    public static void main(String[] args) {
        Random rand = new Random();
        SimpleWindow w = new SimpleWindow(600, 600, "TurtleDrawRandomFigure");
        Turtle t = new Turtle(w, 300, 300);
        t.penDown();
        for (int i = 0; i < 4; i++) {
            t.forward(rand.nextInt(100));
            SimpleWindow.delay(100);
            t.left(90);
        }
    }
}
```

För att dra slumptal måste man skapa ett `Random`-objekt och att man får ett nytt slumpmässigt heltal i intervallet [0, n) med funktionen `nextInt(n)`. Skrivsättet [0, n) betyder att 0 ingår i intervallet, n ingår inte i intervallet. `rand.nextInt(100)` ger alltså ett slumpmässigt heltal mellan 0 och 99.

När klassen `Turtle` ska implementeras måste man bestämma vilka attribut som klassen ska ha. En sköldpadda måste hålla reda på:

- fönstret som den ska rita i: ett attribut `SimpleWindow w`,
- var i fönstret den befinner sig: en x-koordinat och en y-koordinat. För att minska inverkan av avrundningsfel i beräkningarna ska `x` och `y` vara av typ `double`,
- i vilken riktning huvudet pekar. Riktningen kommer alltid att ändras i hela grader. Man kan själv välja om attributet som anger riktningen ska vara i grader eller i radianer,
- om pennan är lyft eller sänkt. Ett sådant attribut bör ha typen `boolean`. booleanvariabler kan bara anta två värden: `true` eller `false`. Man testar om en booleanvariabel `isPenDown` har värdet `true` med `if (isPenDown) ...`.

När sköldpaddan ska gå rakt fram i den aktuella riktningen ska en linje ritas om pennan är sänkt. Den aktuella positionen ska också uppdateras. Antag att sköldpaddan i ett visst ögonblick är vriden vinkeln α i förhållande till positiv x-led. När metoden `forward(n)` utförs ska pennan flyttas från punkten (x, y) till en ny position, som vi kallar $(x1, y1)$:



Av figuren framgår att $(x1, y1)$ ska beräknas enligt:

$$\begin{aligned} x1 &= x + n\cos\alpha \\ y1 &= y - n\sin\alpha \end{aligned}$$

I Java utnyttjas standardfunktionerna `Math.cos(alpha)` och `Math.sin(alpha)` för att beräkna cosinus och sinus. Vinkeln `alpha` ska ges i radianer.

`x` och `y` är av typ `double`. När koordinaterna utnyttjas som parametrar till `SimpleWindow`-metoderna `moveTo` och `lineTo` och när de ska returneras som funktionsresultat i `getX` och `getY` måste de avrundas till heltalsvärden. Det kan till exempel se ut så här:

```
w.moveTo((int) Math.round(x), (int) Math.round(y));
```

Datorarbete

1. I filen *Turtle.java* i projektet *Lab05* finns ett ”skelett” (en klass utan attribut och med tomma metoder) till klassen `Turtle`.

   Implementera klassen `Turtle`: Skriv in attributen i klassen. Skriv gärna en kommentar till varje attribut som förklarar vad attributet betyder. Skriv också konstruktorn och se till att alla attribut får rätt startvärden. Implementera de övriga metoderna.

2. Klasserna `TurtleDrawSquare` och `TurtleDrawRandomFigure` finns i filerna *TurtleDrawSquare.java* och *TurtleDrawRandomFigure.java*. Kör programmen och kontrollera att din `Turtle`-implementation är korrekt.

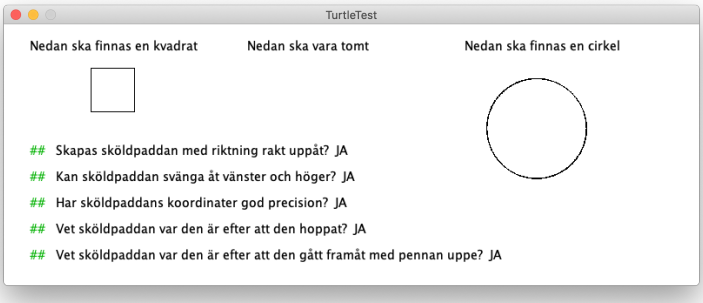
   Använd gärna debuggern för att hitta eventuella svårfunna fel. Här kan det vara bra att utnyttja kommandot Step Into. Det fungerar som Step Over med skillnaden att man följer exekveringen in i metoder som anropas.

3. Du ska nu träna mer på att använda debuggern i Eclipse. I fortsättningen förutsätter vi att du utnyttjar debuggern för att hitta fel i dina program.

   Välj ett av programmen från uppgift 2. Sätt en brytpunkt på en rad där en metod i Turtle anropas, t. ex. `t.penDown()` eller något liknande. Kör programmet i debuggern. Använd Step Into och följ hur man från `main`-metoden ”gör utflykter” in i metoderna i klassen `Turtle`.

   Lägg märke till vilka storheter (variabler, attribut, parametrar) som är tillgängliga när man ”är i” `main`-metoden resp. inuti någon metod i klassen `Turtle`. **Tips!** Om du klickar på trekanten framför this i variabelvyn visas `Turtle`-objektets attribut.

4. I projektet finns också programmet `TurtleTest`, som testar din `Turtle`-klass i olika avseenden. Ett antal misstag kan upptäckas på detta sätt. Resultatet ska se ut så här:



   Kör programmet `TurtleTest`. Om de ritade figurerna är felaktiga, eller något av testerna besvaras med ”NEJ”, gå tillbaka till din `Turtle`-klass och åtgärda felet.

5. Skriv ett program där en sköldpadda tar 1000 steg i ett fönster. Sköldpaddan ska börja sin vandring mitt i fönstret. I varje steg ska steglängden väljas slumpmässigt i intervallet [1, 10]. Efter varje steg ska sköldpaddan vridas ett slumpmässigt antal grader i intervallet [−180, 180].

6. Skriv ett program där *två* sköldpaddor vandrar över ritfönstret. Steglängden och vridningsvinkeln ska väljas slumpmässigt i samma intervall som i föregående uppgift. Vandringen ska avslutas när avståndet mellan de båda sköldpaddorna är mindre än 50 pixlar. Sköldpaddorna ska turas om att ta steg på följande sätt:

```
while ("avståndet mellan sköldpaddorna" >= 50) {
    "låt den ena sköldpaddan ta ett slumpmässigt steg och göra
     en slumpmässig vridning"
    "låt den andra sköldpaddan ta ett slumpmässigt steg och göra
     en slumpmässig vridning"
    SimpleWindow.delay(10);
}
```

   Låt den ena sköldpaddan börja sin vandring i punkten (250,250) och den andra i (350,350).

7. Betrakta klassen `Turtle` och de program du skrev i uppgift 5–6 ovan. Ge exempel på ett attribut, en parameter och en lokal variabel. Ange också var i klassen/programmet respektive storhet är tillgänglig och får användas.

Exempel på	*Namn*	*Får användas var*
Attribut	Nått et objekt har, private	
Parameter	Input til en metod	
Lokal variabel	En verdi som brukes inne i en metod	

Checklista

I den här laborationen har du övat på

- skillnaden mellan attribut, parametrar och lokala variabler,
- att använda konstruktorer för att sätta attributens startvärden,
- att använda klassen Math för beräkningar, och
- att skapa och använda en egen klass.

Laboration 6 – algoritmer, labyrint

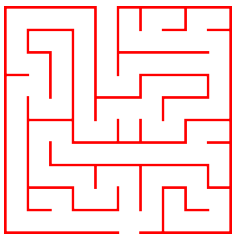
Mål: Du ska lösa ett problem där huvuduppgiften är att konstruera en algoritm för att hitta vägen genom en labyrint. Det ska du göra genom att skriva program som använder din `Turtle`-klass från laboration 5.

Förberedelseuppgifter

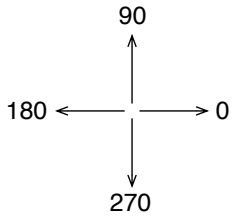
- Läs avsnittet Bakgrund.

Bakgrund

En labyrint består av ett rum med en ingång och en utgång. Alla väggar är parallella med x- eller y-axeln. Ingången finns alltid i den nedersta väggen, och utgången alltid i den översta (figur 1).



Figur 1: En labyrint.



Figur 2: Riktningar.

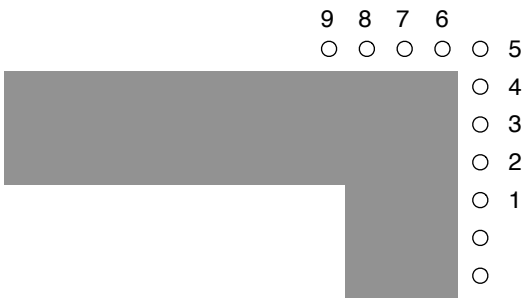
Ett sätt att hitta från ingången till utgången är att gå genom labyrinten och hela tiden hålla den vänstra handen i väggen. När man vandrar genom labyrinten är fyra riktningar möjliga. En riktning definieras som vinkeln mellan x-axeln och vandringsriktningen och mäts i grader (figur 2).

Labyrinten beskrivs av en färdigskriven klass `Maze`. Labyrintens utseende läses in från en fil när labyrintobjektet skapas. Klassen har följande specifikation:

```
/** Skapar en labyrint med nummer nbr. */
Maze(int nbr);

/** Ritar labyrinten i fönstret w. */
void draw(SimpleWindow w);

/** Tar reda på x-koordinaten för ingången. */
int getXEntry();

/** Tar reda på y-koordinaten för ingången. */
int getYEntry();

/** Undersöker om punkten x,y är vid utgången. */
boolean atExit(int x, int y);

/** Undersöker om man, när man befinner sig i punkten x,y och är på väg i
    riktningen direction, har en vägg direkt till vänster om sig. */
boolean wallAtLeft(int direction, int x, int y);

/** Som wallAtLeft, men undersöker om man har en vägg direkt framför sig. */
boolean wallInFront(int direction, int x, int y);
```

I metoderna `wallAtLeft` och `wallInFront` betraktas alla riktningar $R \pm n \cdot 360°, n = 1, 2, \ldots$ som ekvivalenta med R.

Väggarna ritas med lite tomrum mellan sköldpaddan och väggen, så att det blir lättare att skilja dem åt. Exempel där man vandrar runt ett hörn:



I punkterna 1–4 är riktningen 90. `wallAtLeft` ger i dessa punkter värdet `true`, `wallInFront` värdet `false`. I punkt 5 ger `wallAtLeft` värdet `false`, varför man svänger åt vänster (riktningen 90 ändras till 180). Man fortsätter sedan genom punkterna 6–9. I dessa punkter ger `wallAtLeft` värdet `true`, `wallInFront` värdet `false`.

I uppgiften ska du skriva en klass `MazeWalker` som låter en sköldpadda vandra i en labyrint. Låt klassen ha följande uppbyggnad:

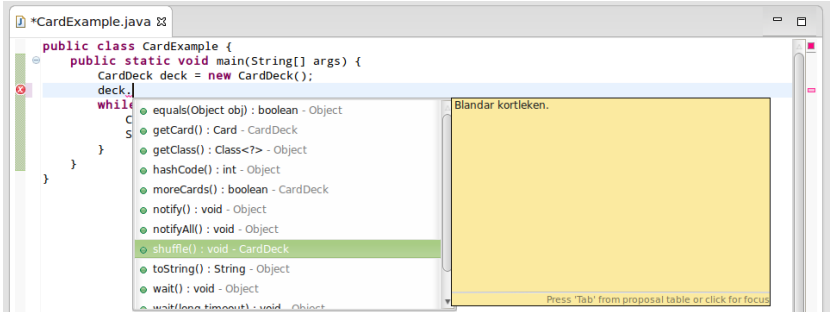
```
public class MazeWalker {
    private Turtle turtle;

    public MazeWalker(Turtle turtle) {
        // fyll i kod
    }

    /** Låter sköldpaddan vandra genom labyrinten maze, från
        ingången till utgången. */
    public void walk(Maze maze) {
        // fyll i kod
    }
}
```

Uppgifter

1. (valfritt) Om du vill kan du få lite mer hjälp av Eclipse när du programmerar. Ta fram fönstret med Eclipse-inställningarna:

   Window → *Preferences* (Windows och Linux, inklusive LTH:s Linuxdatorer)
   Eclipse → *Inställningar...* (Mac)

   Välj *Java* → *Editor* → *Content Assist* → *Advanced*. Klicka i *Java proposals* och *Template proposals*. Klicka *Apply*, därefter *OK*. Hädanefter kommer Eclipse att ge förslag beroende på sammanhanget. Följande bild visar hur det kan se ut när programmeraren skrivit ”`deck`”, tryckt punkt, och nu bläddrar bland förslagen:



2. Till denna uppgift finns inte något färdigt Eclipseprojekt. Du måste alltså skapa ett sådant:

   1. Klicka på *New*-ikonen i verktygsraden. Välj *Java Project*.
   2. Skriv namnet på projektet (t.ex. *MazeTurtle*).
   3. Klicka på *Finish*.
   4. Om Eclipse frågar om filen *module-info.java* ska skapas, så välj **"Don't Create"**. Någon sådan fil behövs inte.
      (Om du råkade trycka "Create" istället, och alltså skapade filen `module-info.java`, så ta bort den: högerklicka på filen i vänsterspalten och välj "Delete" i menyn. Annars kan du få mystiska kompileringsfel.)

3. Programmet som du skriver kommer att använda klasserna `SimpleWindow och Maze`. Dessa klasser finns i biblioteksfilen *cs_pt.jar*, som finns i projektet *cs_pt*. Programmet kommer också att utnyttja klassen `Turtle` från laboration 5.

   Du måste därför göra *cs_pt.jar* och projektet *Lab05* tillgängliga i ditt nya projekt.

   1. Högerklicka på projektet *MazeTurtle*, välj *Build Path* → *Configure Build Path*.
   2. Klicka på fliken *Libraries*. Om det finns en rad *Classpath*, markera då den.
   3. Klicka på *Add JARS...*, öppna projektet *cs_pt*, markera filen *cs_pt.jar* och klicka *OK*.
   4. Klicka nu på fliken *Projects* (istället för *Libraries*). Även här markerar du raden *Classpath* (om den finns).
   5. Klicka på *Add...*, markera *Lab05* och klicka *OK*.
   6. Klicka på *OK* för att lämna dialogen.

4. Skapa klassen `MazeWalker`. I metoden `walk` ska sköldpaddan börja sin vandring i punkten med koordinaterna `getXEntry()`, `getYEntry()` och gå uppåt i labyrinten. Alla steg ska ha längden 1.

   Lägg in en fördröjning efter varje steg så att man ser hur sköldpaddan vandrar. (Använd t.ex. metoden `delay` i `SimpleWindow`; ett exempel finns på s. 15 i detta kompendium.)

5. Skriv ett huvudprogram som kontrollerar att sköldpaddan kan vandra i labyrinterna. Numret på labyrinten ska läsas in från tangentbordet.

   Testa programmet. Det finns fem olika labyrinter, numrerade 1–5. Givetvis ska alla testas och fungera. För den sista labyrinten, som är ganska stor, kan du behöva justera (eller kanske helt ta bort) fördröjningen mellan varje steg.

Frivilliga extrauppgifter

6. Utforska källkoden för `Maze` som finns i projektet *cs_pt*. Kan du se hur du kan använda en egen karta[2] för labyrinten? (Tips: undersök `Maze`-klassens konstruktorer.)

7. Summera antalet steg sköldpaddan tar och antalet riktningsändringar som sköldpaddan gör och skriv ut dessa summor när sköldpaddan kommit i mål.

[2] Du kan exempelvis skapa en egen karta på `http://www.mazegenerator.se`. När du skapat en labyrint på sidan dyker det upp en nya ruta, med en rubrik i stil med "Labyrint med 20x20 fyrkantiga celler". I den rutan väljer du "PNG" i listan och klickar "Ladda ner". Om du använder en egen bild måste den ha helvit eller transparent bakgrund.

Laboration 7 – algoritmer, vektorer (parlaboration)

Denna laboration är en **parlaboration**. Du löser den i samarbete med en annan student (observera: inte större grupper än två och två). Laborationen består av två delar, och ni arbetar tillsammans med båda delarna. Ni ska båda kunna förklara lösningen och svara på handledarens frågor.

Förberedelseuppgifter

- Läs i läroboken, beroende på utgåva:
  Downey & Mayfield, första utgåvan (2016): kapitel 8 och 12 (ej 12.10) samt appendix C
  Downey & Mayfield, andra utgåvan (2019): kapitel 7 och 12 (ej 12.10) samt appendix D

- Läs igenom Bakgrund för del 1 (här nedan).
- Läs igenom Bakgrund för del 2 (s. 38).
- Betrakta följande satser:

```
Pair[] pairs = new Pair[7];         // 1
pairs[0] = new Pair(1, 2);
pairs[1] = new Pair(3, 4);          // 2
```

  Se till att ni båda kan förklara vad som händer i steg 1 respektive 2 ovan. Rita en figur (på papper) som visar vilka variabler och objekt som finns när man exekverat satserna till och med punkt 1 resp. 2 ovan.

Del 1: felsökning av kortläsarsystem

Mål: Här ska du öva på att självständigt hitta fel i program som du inte själv har skrivit.

Bakgrund

Fel i datorprogram kan yttra sig på två sätt: som kompileringsfel eller som exekveringsfel. Att finna och åtgärda sådana fel är en viktig del av programmeringsarbetet.

Kompileringsfel

Vid kompileringsfel markerar Eclipse den eller de felaktiga raderna med kryss till vänster i editorfönstret. Om man håller musmarkören över ett sådant kryss så visas en förklaring av felet. Man kan också se en lista med samtliga fel, genom att i menyn välja *Window* → *Show View* → *Problems*: då listas felen längst ner i Eclipse-fönstret, där konsollutskrifter annars visas.

Kompileringsfel innebär att programmet inte kan kompileras och därför inte heller exekveras. Kompileringsfelen måste alltså åtgärdas.

Förutom kompileringsfel kan kompilatorn också visa *varningar*: dessa betyder att programmet visserligen kan kompileras, men ändå innehåller något som verkar konstigt. Exempelvis varnar kompilatorn om man deklarerar en variabel utan att använda den. Det brukar löna sig att uppmärksamma varningar – de tyder ofta på att man gjort misstag.

Exekveringsfel

Exekveringsfel är fel som uppstår vid exekvering, det vill säga när man kör programmet. Om programmet under körning bryter mot Javas regler – exempelvis dividerar ett heltal med noll – så avbryts programmet och ett felmeddelande skrivs ut. Då skrivs också en anropskedja ut, där man kan se dels i vilken metod felet uppstod, dels varifrån den metoden i sin tur anropades. I

anropskedjan visas radnummer, och när man klickar på ett sådant radnummer öppnas rätt fil i Eclipse (under förutsättning att programtexten, källkoden, för filen är tillgänglig).

Till exekveringsfelen räknas även logiska fel. Ett program kan ge felaktigt resultat även om det följer Javas regler. Om man som programmerare exempelvis skrivit + istället för −, eller använder fel villkor i en if-sats, kommer programmet ju inte att fungera som förväntat.

Felsökning

Som framgått kan kompilatorn inte avgöra om programmet gör det man själv vill, och även om man tänkt igenom sitt program visar sig vissa fel först då programmet körs. För att felsöka i programmet behöver man alltså provköra det.

Felsökning är ett detektivarbete. Genom att pröva olika kombinationer av indata och studera programmets resultat kan man förstå mer om hur felet yttrar sig. Utifrån denna förståelse får man idéer om var i programmet felet kan finnas. När man sedan vill undersöka en viss del av programmet är debuggern till stor hjälp – särskilt möjligheten att sätta brytpunkter, studera variablernas värden och köra programmet stegvis.

I denna uppgift ska du få bekanta dig med sådant felsökningsarbete.

Ett (trasigt) system för passerkontroll

På ett företag används ett system för passerkontroll. När en anställd vill passera genom en dörr drar hen sitt kort i en kortläsare. Systemet slår då, med hjälp av kortets nummer, upp användaren i en tabell. Om personen ifråga finns med i tabellen skrivs namnet ut på en display och dörren låses upp. Annars förblir dörren låst.

Klassen `UserTable` används för att kontrollera vem ett visst passerkort tillhör. I kortläsaren finns en liten dator, som bland annat kör följande Java-kod:

```
UserTable t = new UserTable();
...
int cardNbr = sensor.getCardNbr();

User u = t.find(cardNbr);          // sök upp kortets användare

if (u == null) {
  display.showText("** ogiltigt kort **");
} else {
  display.showText("Välkommen, " + u.getName());
  door.unlock();
}
```

Den intressanta raden har markerats med en kommentar. Givet ett kortnummer (ett heltal) slås motsvarande användare upp. Variablerna `sensor`, `display` och `door` refererar till objekt som vi inte bryr oss om i denna uppgift. Kan du ändå förstå något om hur koden ovan fungerar?

Klassen `UserTable` fungerar dessvärre inte. Din uppgift är att finna och åtgärda felen, redovisa dem enligt protokollet nedan, och visa upp ett fungerande testprogram.

Uppgifter

1. I arbetsområdet finns ett projekt som heter *Lab07*. Öppna projektet och utforska dess två källkodsfiler.

   Det finns också en textfil, *users.txt*, som innehåller systemets alla användare och deras kortnummer. Öppna filen och se hur den är organiserad. I klassen `UserTable` läses användaruppgifterna in från *users.txt* och lagras i en vektor.

2. Vi ska nu söka fel i klassen `UserTable`. För att testa klassen är det opraktiskt att pröva en massa passerkort i kortläsaren. Det är bättre att skapa ett enkelt testprogram som skapar ett `UserTable`-objekt och anropar metoden `find` på samma sätt som kortläsaren gör. Du kan ju då hitta på och testa olika kortnummer själv.

   Skapa ett sådant testprogram (en klass med en `main`-metod) som

   - anropar metoden `find` för att slå upp användarna med kortnummer 24073 respektive 24074, samt
   - anropar metoden `findByName` för att slå upp användaren med namnet ”Jens Holmgren”.

   Vad är programmets resultat? Vad borde det vara? (Jämför med filen *users.txt*.)

   Anm. Då klassen `User` har en `toString`-metod kan man skriva ut ett `User`-objekt som följer. Detta fungerar även då referensen `u` råkar vara `null`.

   ```
   User u = ...;
   System.out.println(u);
   ```

3. Klassen `UserTable` innehåller en metod för att skriva ut tabellens innehåll. Utöka ditt testprogram så att denna metod anropas, och se vad som skrivs ut.

   Öppna konstruktorn för klassen `UserTable`. Här läses användaruppgifterna in från fil och lagras i en vektor. Antingen fungerar inte inläsningen, eller så hamnar de inlästa raderna på något vis fel i vektorn.

   Lägg till några rader i ditt testprogram, så att en ny (påhittad) användare med kortnummer 1234 läggs in, och tabellens innehåll därefter skrivs ut. Kör testprogrammet. Hur många användare finns nu i tabellen? Hur många hade du förväntat dig?

4. Hitta och åtgärda felet som du identifierat i föregående uppgift. För att få ytterligare ledtrådar kan du exempelvis ställa dig följande frågor:

   - I metoden `add` sätts ett `User`-objekt in på rätt plats i vektorn. Vilka platser hamnar objekten på, dvs vilket värde får variabeln `pos`? (Du kan exempelvis använda utskrifter eller brytpunkter.)
   - Metoden `getNbrUsers` returnerar fel resultat. (Pröva.) Metoden returnerar värdet av ett attribut. Hur ska attributet få rätt värde?

   När du åtgärdat felet ska *en* av sökningarna på kortnummer ovan fungera. Se vilket namn sökningen ger och jämför med motsvarande rad i *users.txt*. (Den andra sökningen på kortnummer returnerar fortfarande `null`.)

5. Kompilatorn varnar för en detalj i en av Java-filerna. Varningen tyder på att något är konstigt. Åtgärda felet. Du ska *inte* ta bort den gulmarkerade koden.

   När felet är åtgärdat ska testprogrammets sökning efter Jens Holmgren fungera.

   Anm. Varningen ”dead code” betyder att den markerade delen av koden aldrig kommer att köras, oavsett vad som händer i programmet. Felet ligger alltså inte i den markerade koden, utan troligen i den närmast omgivande koden.

6. Ännu återstår ett fel. Tydligen fungerar `find` ännu bara för vissa passerkort. Sökalgoritmen har anpassats från bokens exempel, och ett fel tycks ha smugit sig in.

- Det finns en metod i `UserTable` som kan användas för att testa metoden `find`. Låt ditt testprogram anropa metoden. Du kommer nu att upptäcka att det finns fler kortnummer som inte går att hitta med `find`.
- Åtgärda felet. I felsökningen har du kanske nytta av debuggern, eller av att införa utskrifter av variabelvärden. Det kan även löna sig att granska koden noggrant.
- När felet är löst: verifiera att alla användare (kortnummer) nu kan sökas upp.

Anm. Namnen i tabellen är genererade utifrån Sveriges 100 vanligaste efternamn samt manliga och kvinnliga förnamn.[3] De syftar inte på några verkliga personer.

Redovisning

För att bli godkänd på uppgiften ska du kunna visa upp ett program som fungerar enligt ovan. Därtill ska du vara beredd att berätta om ditt felsökningsarbete. Fyll därför i information om varje fel i tabellen nedan (kategorisera dem utifrån beskrivningen av olika slags fel ovan).

Skriv ner dina resultat (exempelvis i den följande tabellen) så att du minns dem till redovisningen.

Fel (vilken klass, vilken rad?)	*Beskriv: Vad var problemet?*	*Typ av fel*	*Hur upptäcktes felet?*

[3] Källa: SCB, `http://www.scb.se/hitta-statistik/statistik-efter-amne/befolkning/amnesovergripande-statistik/namnstatistik/`, 2015.

Laboration 7, del 2: simulering av patiens

Bakgrund

Patiensen 1–2–3 läggs på följande sätt: Man tar en kortlek, blandar den och lägger sedan ut korten ett efter ett. Samtidigt som man lägger korten räknar man 1–2–3–1–2–..., det vill säga när man lägger det första kortet säger man 1, när man lägger det andra kortet säger man 2, osv. Patiensen går ut om man lyckas lägga ut alla kort i leken utan att någon gång få upp ett ess när man säger 1, någon 2-a när man säger 2 eller någon 3-a när man säger 3.

Man kan med hjälp av sannolikhetslära bestämma exakt hur stor sannolikheten är att patiensen ska gå ut.[4] Men det är betydligt enklare att uppskatta sannolikheten genom att lägga patiensen många gånger och räkna antalet gånger som den går ut.

Allra snabbast går det om datorn lägger patiensen. Här bortser vi från en del detaljer kring spelkorten, och betraktar dem istället som par av heltal (a, b), där $0 \leq a < 4$ och $0 \leq b < 13$. Här representerar alltså a kortets färg (spader, hjärter, ruter, klöver) och b dess valör. För enkelhets skull låter vi dessa värden börja på 0, medan riktiga spelkort naturligtvis räknar från 1. Det spelar ingen större roll här, men gör programmeringen lite enklare.

Klassen `Pair` beskriver ett sådant par av heltal, som alltså kan användas för att representera ett spelkort:

`Pair`

```
/** Skapar ett par med a som första tal, och b som andra. */
Pair(int a, int b);

/** Hämtar parets första tal. */
int first();

/** Hämtar parets andra tal. */
int second();

/** Hämtar en sträng som beskriver paret, exempelvis "(5,3)". */
String toString();
```

Vidare behöver vi en klass som kan representera en blandad kortlek. Vi kan se kortleken som en mängd av talpar, där varje talpar förekommer exakt en gång. Klassen `PairSet` beskriver just en sådan mängd, som vi kan dra ett slumpmässigt valt `Pair`-objekt ur, på samma sätt som vi kan dra kort ur en kortlek:

`PairSet`

```
/** Skapar en mängd av alla talpar (a,b) sådana att
    0 <= a < rows  och  0 <= b < cols */
PairSet(int rows, int cols);

/** Undersöker om det finns fler par i mängden. */
boolean more();

/** Hämtar ett slumpmässigt valt talpar ur mängden. Mängden blir
    ett element mindre. Om mängden är tom returneras null. */
Pair pick();
```

[4] Se ”Fråga Lund om matematik”, `www.maths.lth.se/query/answers`, sök efter ”patiens” på sidan.

Följande program skriver ut färg (0..3) och valör (0..12) för alla kort som dras ur en blandad kortlek:

```
public class CardDeckExample {
    public static void main(String[] args) {
        PairSet cardDeck = new PairSet(4, 13);
        while (cardDeck.more()) {
            Pair p = cardDeck.pick();
            System.out.println(p);
        }
    }
}
```

Med hjälp av `System.out.println(p)` skriver man ut ett kort (talpar) på skärmen. Inuti `println` anropas metoden `toString` och den sträng som metoden returnerar kommer att skrivas ut. Det är alltså i `Pair`-klassens `toString`-metod man bestämmer hur talparet ska presenteras på skärmen.

Datorarbete

1. Öppna projektet *Lab07*. I projektet finns ett paket `pair`, och där finns bland annat den färdiga klassen `Pair`. Öppna filen *Pair.java* och se hur klassen är implementerad.

2. Det finns även ett paket `cardgame`, som innehåller några programexempel och ett testprogram. Ett program med programraderna från den tredje förberedelseuppgiften finns i filen *ArrayExample.java*. Sätt en brytpunkt i början av programmet och provkör programmet i debuggern. Framför `pairs` i variabelvyn visas ett plustecken eller en triangel – klicka där så ser du innehållet i vektorn. Stämmer resultatet med din skiss?

3. Implementera klassen `PairSet`, som ska hålla reda på en mängd av talpar, enligt specifikationen ovan. Det görs lämpligen genom att skapa de nödvändiga `Pair`-objekten och lagra dem i en vektor:

   ```
   public class PairSet {
       private int n;
       private Pair[] pairs;
       private static Random rand = new Random();

       public PairSet(int rows, int cols) {
           n = rows * cols;
           pairs = new Pair[n];

           ... // skapa paren, lägg in dem i vektorn
       }
       ...
   }
   ```

   Tips: När man tar bort ett element ur en mängd, där ordningen inte behöver bibehållas, så kan man ersätta det borttagna elementet med det sista, samt minska antalet element med 1. Jämför diskussionen i avsnitt 8.4 i boken, där det även diskuteras hur man kan göra när man vill bibehålla ordningen.

4. Ett testprogram för `PairSet` finns i projektet, i filen *TestCardDeck.java*.

   Kör programmet och kontrollera att det som skrivs ut och det som visas i fönstret ser ut att stämma.

5. Skapa ett huvudprogram (en klass med `main`-metod). I nästa uppgift kommer du att skriva ett program som simulerar patiensen ett stort antal gånger och uppskattar sannolikheten för att den går ut.

   Du kommer nog att behöva experimentera lite med hur många ”ett stort antal gånger” är. Det är därför praktiskt att införa en konstant som anger antalet upprepningar. Låt den exempelvis heta `NBR_ITERATIONS`, inledningsvis ha värdet 10000 och deklareras som konstant i din klass:

   ```
   private static final int NBR_ITERATIONS = 10000;
   ```

   Du kommer att använda detta värde på flera ställen i ditt program. Då använder du istället namnet `NBR_ITERATIONS`, och när du vill ändra antalet behöver du bara ändra på ett enda ställe i ditt program. Det står mer om detta i bokens avsnitt 3.5 (”Literals and constants”, Downey/Mayfield) respektive 5.4 (Holm).

6. Skriv ett huvudprogram som utnyttjar klasserna `Pair` och `PairSet` för att lägga patiensen ett stort antal gånger. Programmet ska skriva ut sannolikheten för att patiensen ska gå ut. Justera din konstant (enligt föregående uppgift) så att du får en bra precision i resultatet utan att exekveringstiden blir alltför lång.

   Kör programmet. Den korrekta sannolikheten är ungefär 0,008165.

   Tips: Det är en bra idé att lösa uppgiften i flera, mindre steg. En möjlighet är att börja med det enkla programmet på sidan 39, och bygga vidare därifrån. Kan du tänka dig någon liten förändring som (1) för dig närmare lösningen, och (2) går att köra?

Checklista

I den här laborationen har du övat på att

- använda vektorer för att lagra och hantera objekt,
- använda symboliska konstanter,
- använda attribut för att lagra information om objektets tillstånd, och
- konstruera algoritmer i Java.

Laboration 8 – matriser, Memory-spel

Mål: Du ska träna på att att använda matriser.

Förberedelseuppgifter

- Läs avsnittet Bakgrund.

Bakgrund

Memory är ett spel som går ut på att hitta matchande par på en spelplan med kort. I spelet används ett jämnt antal kort, där det finns exakt två kort med samma motiv på framsidan.

Spelet går till som följer. Alla korten blandas och placeras ut på spelplanen med baksidan uppåt. Spelaren vänder på två kort i taget. Om de båda korten visar olika bilder måste de vändas upp och ner igen. Om de visar samma bild får de ligga kvar med framsidan upp. Spelet fortsätter tills spelaren lyckats hitta alla par.

Datorarbete

1. För att beskriva en (tvåsidig) bild av ett memorykort används den färdiga klassen `MemoryCardImage`.

   `MemoryCardImage`

   ```
   /** Skapar en tvåsidig bild av ett memorykort.
       Bilden på framsidan finns i filen med namnet frontFileName och
       bilden på baksidan i filen med namnet backFileName. */
   MemoryCardImage(String frontFileName, String backFileName);

   /** Returnerar bilden på framsidan. */
   Image getFront();

   /** Returnerar bilden på baksidan. */
   Image getBack();
   ```

   Klassen `MemoryCardImage` finns i ditt projekt *Lab08*. Titta på koden, se vilka metoder som finns och ungefär hur de är skrivna. Kan du förstå koden på ett ungefär?

2. Klassen `MemoryBoard` beskriver ett bräde med 16 memorykort. Ett ”skelett” till klassen finns i projektet *Lab08*. Din uppgift är att implementera den enligt specifikationen nedan.

   `MemoryBoard`

   ```
   /** Skapar ett memorybräde med size * size kort.
       backFileName är filnamnet för filen med baksidesbilden.
       Vektorn frontFileNames innehåller filnamnen för frontbilderna. */
   MemoryBoard(int size, String backFileName, String[] frontFileNames);

   /** Tar reda på brädets storlek. */
   int getSize();

   /** Hämtar den tvåsidiga bilden av kortet på rad r, kolonn c.
   Raderna och kolonnerna numreras från 0 och uppåt. */
   MemoryCardImage getCard(int r, int c);

   /** Vänder kortet på rad r, kolonn c. */
   void turnCard(int r, int c);
   ```

```
/** Returnerar true om kortet r, c har framsidan upp. */
boolean frontUp(int r, int c);

/** Returnerar true om det är samma kort på rad r1, kolonn c1
    som på rad r2, kolonn c2. */
boolean same(int r1, int c1, int r2, int c2);

/** Returnerar true om alla kort har framsidan upp. */
boolean hasWon();
```

Att placera ut korten

När brädet skapas ska 8 st `MemoryCardImage`-objekt skapas. Varje sådant objekt ska placeras ut på två slumpmässiga platser på brädet. Om det redan finns ett objekt på en vald plats måste du välja en ny plats. Du kan alltså behöva dra slumptal flera gånger för att placera ut ett kort. En sådan algoritm kan informellt formuleras så här:

Upprepa följande för vart och ett av filnamnen i `frontFileNames`:

- Skapa ett `MemoryCardImage`-objekt med detta namn.
- Placera **två** referenser till detta objekt på brädet.
  Varje referens placeras ut så här:
  - Gissa (dra slumptal) värden på rad och kolumn för kortet.
    Kalla dessa värden r och c.
  - Så länge platsen (r, c) är upptagen på brädet, dra nya slumptal för r och c.
  - Därefter vet vi att platsen (r, c) är ledig.
    Det går därför bra att stoppa in en ny referens där.

Ledning

- Du måste alltså skilja på den bild av spelet som den som kör programmet har och hur det ser ut inuti klassen `MemoryBoard`. Spelaren ser 16 kort framför sig, men i själva verket representeras korten istället av de 8 `MemoryCardImage`-objekten.
- För att hålla reda på om ett kort har framsidan upp eller ej ska du använda en `boolean`-matris med samma dimensioner som brädet.

  Klassen `MemoryWindow` anropar metoden `frontUp` i `MemoryBoard` för att avgöra om ett givet kort ska ritas som upp- eller nedvänt. Det huvudprogram, som du kommer att skriva senare i laborationen, kommer också att använda den metoden för att avgöra om ett kort kan vändas upp eller ej.

  Medan du testar utplaceringen av kort i din `MemoryBoard`-klass (algoritmen ovan) kan det vara praktiskt att låta alla kort ritas uppvända från början. Då kan vi se att utplaceringen av korten fungerar som den ska. För sådan testning ska alltså metoden `frontUp` returnera `true` för alla platser på brädet.
- Vi vill gärna undvika alltför långa och stökiga metoder, och detta gäller särskilt konstruktorer. I kodskelettet finns därför en privat hjälpmetod `createCards`. Privata metoder syns inte i specifikationen, men genom att implementera metoden och använda den från konstruktorn blir konstruktorn kortare och koden mer lättläst.

  Det är inte nödvändigt att använda just denna privata metod: välj gärna en egen.

- När man utvecklar program implementerar man oftast inte en hel klass i taget, som det beskrivs här, utan det normala är att man vill testa mycket och ofta. De första testerna gör man helst innan man har hunnit skriva allt för mycket kod.

  För denna uppgift är det rekommenderat att implementera konstruktorn och metoderna `getSize` samt `getCard` och därefter gå vidare till `main`-metoden i `MemoryGame` (se uppgifterna 3 och 4) och där kontrollera att man kan skapa och rita upp ett bräde med alla kortens baksidor upp. När det fungerar implementerar man resten av `MemoryBoard` och för vaje metod som implementeras kan man göra ett enkelt test genom att anropa respektive metod från main-metoden.

3. För att visa korten och låta användaren klicka på ett kort finns den färdigskrivna klassen `MemoryWindow` (se specifikationen nedan). Klassen är en subklass till `SimpleWindow` vilket betyder att den har de attribut och metoder som finns i `MemoryWindow` plus allt som finns i `SimpleWindow`.

```
/** Skapar ett fönster som kan visa memorybrädet board. */
MemoryWindow(MemoryBoard board);

/** Ritar brädet med alla korten. */
void drawBoard();

/** Ritar kortet på rad r, kolonn c.
    Raderna och kolonnerna numreras från 0 och uppåt. */
void drawCard(int r, int c);

/** Tar reda på raden för senaste musklick. */
int getMouseRow();

/** Tar reda på kolonnen för senaste musklick. */
int getMouseCol();
```

   Vilka metoder som finns i `SimpleWindow` undersöker du enklast i ditt workspace. (I Eclipse-projektet *cs_pt* under katalogen *doc* kan man hitta filen `index.html`. Dubbelklicka på den för att få upp dokumentationen över de färdiga klasserna. Välj `SimpleWindow` i listan nere till vänster.)

   De metoder i `SimpleWindow` som du troligen behöver för att klara av uppgiften är följande:

```
/** Väntar ms millisekunder. */
static void delay(int ms);

/** Väntar tills användaren klickat på ett kort på brädet. */
void waitForMouseClick();
```

   Öppna `MemoryWindow` och läs igenom koden innan du går vidare.

4. Öppna klassen `MemoryGame` och skriv klart `main`-metoden som ska låta en person spela Memory. Korten ska visas i ett fönster och spelaren ska upprepade gånger välja två kort genom att klicka på dem. När alla kort visar framsidan är spelet slut och antal försök som krävdes ska skrivas ut.

```
public class MemoryGame {
    public static void main(String[] args) {
        String[] frontFileNames = { "can.jpg", "flopsy_mopsy_cottontail.jpg",
```

```
                "friends.jpg", "mother_ladybird.jpg", "mr_mcgregor.jpg",
                "mrs_rabbit.jpg", "mrs_tittlemouse.jpg", "radishes.jpg" };

            // Fyll i egen kod här
        }
    }
```

Vektorn `frontFileNames` innehåller de 8 filnamnen för framsidesbilderna. Bilden för kortens baksida finns i en fil med namnet *back.jpg*.

Ledning:

- Metoden `drawCard` i `MemoryWindow` ritar baksidan av det angivna kortet om det inte är vänt, och framsidan av kortet om det är noterat som vänt. Att vända ett kort är alltså i själva verket att ändra informationen om huruvida kortet är vänt eller ej, och därefter rita om kortet på nytt.
- Tänk på att spelaren måste hinna se på korten en stund innan de eventuellt vänds tillbaka.
- Om man klickar på ett redan uppvänt kort ska klicket ignoreras.

Frivilliga extrauppgifter

Extrauppgifterna har ingen inbördes ordning och beror inte av varandra.

5. Algoritmen för att placera ut korten (ledningen i uppgift 2 ovan) kan vara ineffektiv. Sannolikt behöver man dra nya slumptal flera gånger innan man hittar två lediga rutor. När det bara finns en ledig ruta kvar, exempelvis, är sannolikheten att man träffar rätt $\frac{1}{16}$.

   Konstruera om utplaceringen av kort så att inga nya försök behöver göras enligt ovan.

   Ledning. Man kan formulera en effektivare algoritm, om man ser utplaceringen av kort som en uppräkning av rutor i slumpmässig ordning. Varje rutas koordinatpar, t.ex. `(1,2)`, ska förekomma exakt en gång i sekvensen. Använd `PairSet` och `Pair` från föregående laboration. Gör dem tillgängliga i ditt projekt så här:

   1. Högerklicka på projektet *Lab08*, välj *Build Path* → *Configure Build Path*.
   2. Klicka på fliken *Projects*.
   3. Klicka på *Add...*, markera *Lab07* och klicka *OK*.
   4. Klicka på *OK* för att lämna dialogen.

   Nu kan `PairSet` och `Pair` användas även in denna laboration. Kom ihåg att de importeras från paketet `pair`.

6. Istället för att skriva ut antalet försök som krävdes till terminalfönstret kan det vara idé att visa resultatet i en ruta istället. Experimentera med Javas inbyggda klass `JOptionPane` för att göra detta. Testa t.ex:

   ```
   JOptionPane.showMessageDialog(null, "meddelande", "titel",
       JOptionPane.INFORMATION_MESSAGE);
   ```

7. Modifiera ditt spel så att användaren kan välja att köra en runda till, utan att stänga av mellan gångerna.

8. Inför ett rekord (”highscore”) som visas efter varje spelomgång och skrivs till fil.

9. För den intresserade: kör ditt Memory-program som en Android-applikation.
   Ledtrådar finns här: `http://cs.lth.se/androidmemory`

Laboration 9 – TurtleRace, del 1: ArrayList

Mål: Du ska träna på att använda listor för att hålla reda på objekt. Du ska även förstå grundläggande användning av arv.

Förberedelseuppgifter

- Läs i läroboken: Downey & Mayfield, kapitel 14.
- Läs avsnittet Bakgrund.

Bakgrund

Under den här laborationen ska du skriva ett program där du låter sköldpaddor tävla mot varandra i en kapplöpning.

Vi använder klassen `SimpleWindow` på ett annorlunda sätt än tidigare: vi har här specialiserat den genom subklassen `RaceWindow`. Ett `RaceWindow` fungerar som ett vanligt `SimpleWindow` (eftersom det ärver från `SimpleWindow`) men med tillägget att en kapplöpningsbana ritas upp då fönstret skapas.

Uppgiften går ut på att skriva ett program som simulerar en (slumpmässig) kapplöpning mellan sköldpaddor (`Turtle`-objekt). En sköldpadda tar sig fram genom simulerade tärningskast (dvs. slumpmässiga steg framåt mellan 1 och 6), med ett ”kast” i varje runda. Samtliga sköldpaddor ska lagras i en `ArrayList`.

En klass `RaceTurtle` ska representera en sköldpadda som kan simulera en kapplöpning. `RaceTurtle` ska ärva från klassen `Turtle` och därmed kan en `RaceTurtle` göra allt som en vanlig `Turtle` kan. Därtill ska en `RaceTurtle` innehålla en slumptalsgenerator och ett startnummer. RaceTurtle ska ha en metod `raceStep()` och en metod `toString()`. Metoden `raceStep()` ska beskriva hur sköldpaddan tar ett löpsteg.

`RaceTurtle` ska alltså ärva från Turtle och ha följande specifikation:

```
/**
 * Skapar en sköldpadda som ska springa i fönstret w och som har start-
 * nummer nbr. Sköldpaddan startar med pennan nere och nosen vänd åt höger.
 */
RaceTurtle(RaceWindow w, int nbr);

/**
 * Låter sköldpaddan gå framåt ett steg. Stegets längd ges av ett
 * slumptal (heltal) mellan 1 och 6.
 */
void raceStep();

/**
 * Returnerar en läsbar representation av denna RaceTurtle,
 * på formen "Nummer x" där x är sköldpaddans startnummer.
 */
String toString();
```

Datorarbete

1. Inkludera din `Turtle` från laboration 5 genom att:
   1. Högerklicka på projektet *Lab09*, välj *Build Path* → *Configure Build Path*.
   2. Klicka på fliken *Projects*.
   3. Klicka på *Add...*, markera *Lab05* och klicka *OK*.
   4. Klicka på *OK* för att lämna dialogen.

2. Implementera klassen `RaceTurtle` enligt beskrivning och specifikation ovan. När du implementerar konstruktorn för `RaceTurtle` måste du beräkna dess position. Det finns två hjälpmetoder i `RaceWindow`, som beroende på startnummer returnerar en lämplig x- respektive y-position. På så sätt får alla sköldpaddor en egen bana.

3. I klassen `RaceTurtleTest` finns en main-metod som skapar en sköldpadda och låter denna genomföra ett eget lopp utan motståndare. Klassen kan användas som ett första test av din `RaceTurtle`. Avkommentera och studera koden – det är meningen att du ska förstå vad den gör. Kör sedan programmet för att se om sköldpaddan går i mål som förväntat.

4. Gör en ny klass `TurtleRace` för att genomföra ett lopp enligt följande (det är lämpligt att inspireras av testprogrammet):
   - Skapa åtta sköldpaddor och lagra dessa i en `ArrayList`. Låt dem ha sin kapplöpning i ett och samma `RaceWindow`.
   - Efter loppet ska första, andra och tredje plats skrivas ut (enligt exemplet nedan). Det är ett krav att utskriften använder sig av `toString`-metoden.

     ```
     På plats 1: Nummer 5
     På plats 2: Nummer 6
     På plats 3: Nummer 1
     ```

   - **Tips:** När en sköldpadda går i mål kan du ta ut den ur den vanliga listan, och sätta in den i en separat lista. När den vanliga listan är tom har alla sköldpaddorna gått i mål, och då görs utskriften ovan.
   - Observera att en for-each-loop inte tillåter att vi ändrar en listas innehåll i loopen.
   - Små orättvisor i resultatberäkningen är acceptabla. Kanske upptäcker du att din lösning gynnar sköldpaddor med lägre startnummer över de med högre startnummer. Förklara för din handledare vari orättvisan består, och diskutera gärna hur man kan åtgärda den.
   - Det är därmed inte heller nödvändigt att hantera delade placeringar.

5. Provkör ditt sköldpaddslopp. Använd en fördröjning, t. ex. `RaceWindow.delay(10)`, så man hinner se hur loppet fortskrider. Det kan även vara trevligt att låta användaren starta loppet med ett musklick, dvs `w.waitForMouseClick()` när sköldpaddorna är uppställda på startlinjen. Kontrollera att rätt placeringar skrivs ut.

Checklista

I den här laborationen har du övat på att

- använda listor för att lagra objekt,
- ärva egenskaper från en superklass,

- implementera och använda `toString`-metoden, och
- använda statiska hjälpmetoder.

Laboration 10 – TurtleRace, del 2: mer arv

Mål: Du ska lära dig mer om att skriva och använda klasser som är strukturerade med hjälp av arv. Du får använda polymorfism – arv – för att beskriva olika slags sköldpaddor.

Förberedelseuppgifter

- Gör hela TurtleRace del 1. Det är viktigt att den fungerar innan du sätter igång.
- Läs i läroboken: Downey & Mayfield, kapitel 14.
- Läs avsnittet Bakgrund.

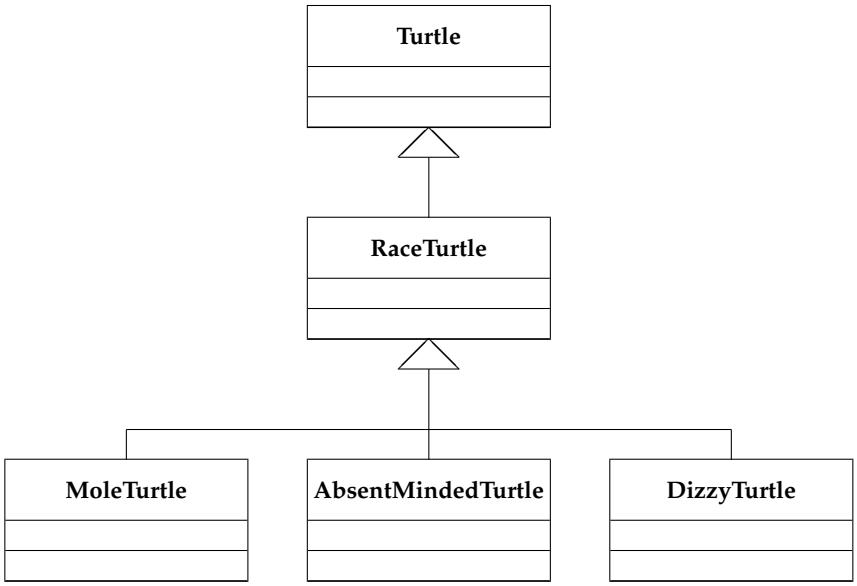
Bakgrund

Under den här laborationen ska du skriva ett program där du låter olika typer av sköldpaddor tävla mot varandra i en kapplöpning. Denna laboration bygger vidare på din laboration 9.

Uppgiften går ut på att skriva ett antal klasser som beskriver olika slags tävlande sköldpaddor, samt att modifiera huvudprogrammet för att genomföra en sådan kapplöpning. Klassen `RaceTurtle` (från laboration 9) ska representera det som samtliga tävlingssköldpaddor har gemensamt. Nu ska emellertid metoden `raceStep` överskuggas (override) av subklassernas mer specifika definition av metoden. Vi behöver tre subklasser, en för varje slags sköldpadda, och bilden nedan visar arvshierarkin.

Programmet ska innehålla tre slags tävlande sköldpaddor (subklasser till `RaceTurtle`):

- `MoleTurtle`, som beskriver en ”mullvadssköldpadda”, d.v.s. en tävlande sköldpadda som då och då går under jorden (genom att lyfta pennan slumpmässigt).
- `AbsentMindedTurtle`, som är en tankspridd sköldpadda. Graden av tankspriddhet anges i procent när sköldpaddan skapas, och sannolikheten att en tankspridd sköldpadda skall gå framåt bestäms av tankspriddhetsgraden. Exempel: en tankspriddhetsgrad på 34 procent ska medföra att sköldpaddan i 34 procent av fallen glömmer att ta sitt steg.
- `DizzyTurtle`, som beskriver en yr tävlingssköldpadda, som vinglar när den skall ta sig framåt. När dessa sköldpaddor skapas ska graden av yrsel (från 1 till 5) anges, och deras förmåga att hålla kursen skall bestämmas av yrselgraden. Du behöver inte göra någon avancerad simulering av yrseln, men en sköldpadda med högre yrselgrad bör vingla mer än en mindre yr sköldpadda.



Datorarbete

1. Kopiera filerna från förra laborationen genom att högerklicka på projektet *Lab09* och välja *Copy*. Högerklicka sedan igen och välj *Paste*. Ge projektet ett nytt namn, t.ex. *Lab10*. Öppna det nya projektet och fortsätt din utveckling där.

2. Implementera tävlingssköldpaddorna som beskrivs ovan. Skapa en ny klass för varje typ av sköldpadda och låt varje klass ärva från `RaceTurtle`. Varje sådan subklass ska ha sin egen definition av metoderna `raceStep()` och `toString()` – dessa ska alltså implementeras på nytt.

   Observera att `raceStep()` och `toString()` inte är abstrakta i `RaceTurtle`. De har alltså redan en implementation. Det gör att du själv måste komma ihåg att överskugga dessa i varje subklass, men de går samtidigt att utnyttja för att slippa duplicera kod. Exempelvis kan kan man från subklassernas `raceStep`-metoder anropa `super.raceStep()` för att genomföra själva steget, förutom den kod som är specifik för varje subklass (lyfta/sänka pennan, jämföra tankspriddhet, välja riktning).

3. Modifiera klassen `TurtleRace` för att genomföra ett lopp enligt följande:
   - Skapa åtta sköldpaddor av slumpmässig typ.
   - **Notera** att då alla ärver från en gemensam klass behöver listan inte ändras på något vis. Se till att du förstår varför en lista av `RaceTurtle` även kan innehålla objekt av klasserna `MoleTurtle`, `AbsentMindedTurtle` och `DizzyTurtle`. Fråga gärna handledaren.
   - Då `AbsentMindedTurtle` ska skapas ska tankspriddhetsgraden slumpas fram mellan 0 och 100.
   - Då `DizzyTurtle` ska skapas ska yrselgraden slumpas fram mellan 1 och 5.
   - Innan loppet börjar ska hela startuppställningen skrivas ut enligt exemplet nedan, ett krav är att detta sker genom respektive sköldpaddas `toString`-metod.

     ```
     Nummer 1 - MoleTurtle
     Nummer 2 - AbsentMindedTurtle (45% Frånvarande)
     Nummer 3 - AbsentMindedTurtle (43% Frånvarande)
     Nummer 4 - DizzyTurtle (Yrsel: 3)
     Nummer 5 - MoleTurtle
     Nummer 6 - MoleTurtle
     Nummer 7 - AbsentMindedTurtle (71% Frånvarande)
     Nummer 8 - DizzyTurtle (Yrsel: 5)
     ```

   - Efter loppet ska första, andra och tredje plats skrivas ut (enligt exemplet nedan).

     ```
     På plats 1: Nummer 5 - MoleTurtle
     På plats 2: Nummer 6 - MoleTurtle
     På plats 3: Nummer 4 - DizzyTurtle (Yrsel: 3)
     ```

Checklista

I den här laborationen har du övat på att

- använda superklasser för att definera generella typer, och
- använda arv för att specialisera beteendet för specifika objekt.

2 Projekt

- Projektet ska lösas i par (observera: inte större grupper än två och två) och ska redovisas på för ändamålet avsedd laborationstid.
- Var förberedd inför presentationen. Du ska vara beredd att svara på frågor kring ditt program och hur du har tänkt. Båda måste kunna svara på alla frågor. Dela därför inte upp arbetet utan samarbeta kring samma program.
- Det är tillåtet att diskutera uppgifterna och tolkningen av dessa med utomstående på ett allmänt plan men *inte* att få hjälp med de konkreta lösningarna.
- Det är inte tillåtet att kopiera annans lösningar helt eller delvis. Det är inte heller tillåtet att kopiera från litteratur eller internet.
- Väsentlig hjälp, av annat än lärare på kursen, för att genomföra en uppgift skall redovisas tydligt. Detsamma gäller om man använt någon annan form av hjälpmedel som läraren inte kan förutsättas känna till.
- Kontakta ansvarig lärare om du är osäker på någon av punkterna ovan.

Projektuppgift – bankapplikation

Uppgiften är att skriva ett program som håller reda på bankkonton och kunder i en bank. Programmet är helt textbaserat: en meny skrivs ut till konsollen och menyvalen görs via tangentbordet. Du ska skriva programmet helt själv, och det ska följa de objektorienterade principer du lärt dig i kursen.

Krav

Kraven för programmet beskrivs nedan. Sist i häftet återfinns också ett antal exempelkörningar, som illustrerar hur ditt program ska fungera.

För att ditt program ska bli **godkänt** ska det uppfylla följande krav:

- Programmet ska ha följande funktioner:
  1. Hitta konton för en viss kontoinnehavare
  2. Sök kontoinnehavare på (del av) namn
  3. Sätta in pengar
  4. Ta ut pengar
  5. Överföring mellan konton
  6. Skapa nytt konto
  7. Ta bort konto
  8. Skriv ut bankens alla konton
  9. Avsluta
- Programdesignen ska följa de specifikationer som finns under avsnittet Design nedan.
- Du ska använda en lista (t.ex. `ArrayList`) för att hålla reda på alla bankkonton.
- Inga utskrifter eller inläsningar (`System.out`, `Scanner` och liknande) får lov att finnas i klasserna `Customer`, `BankAccount` eller `Bank`. Allt som gäller användargränssnitt skall skötas från klassen `BankApplication`. I `BankApplication` får du lov att använda valfritt antal hjälpmetoder eller andra hjälpklasser för att sköta användargränssnittet. (Det är vanligt att man begränsar användargränssnittet till vissa klasser på detta sätt. På så sätt kan klasserna `Customer`, `BankAccount` och `Bank` återanvändas i ett annat program med ett annat användargränssnitt.)
- Hela projektet får bara innehålla ett `Scanner`-objekt för att läsa från tangentbordet (dvs. man får bara göra `new Scanner(System.in)` en gång).
- Klasserna `Customer`, `BankAccount` och `Bank` ska exakt följa specifikationerna som anges under avsnittet Design nedan. Det är tillåtet att lägga till privata hjälpmetoder. (Det är inte ovanligt att en klass, för att fungera ihop med andra klasser, måste se ut på ett visst sätt – följa en viss specifikation – trots att man implementerar funktionaliteten själv.)
- Alla metoder och attribut ska ha lämpliga åtkomsträttigheter (private/public).
- De indata som ges i exemplet nedan ska ge väsentligen samma utskrift som exemplen i bilagan. Du får gärna byta ordval, men ditt program ska i allt väsentligt fungera likadant.
- Listor ska skrivas ut genom att iterera över listans objekt och anropa respektive `toString`-metod. Det är inte tillräckligt att använda listans `toString`-metod direkt, eftersom utskriften då inte får rätt format.

- Koden ska vara snyggt formaterad med korrekt indrag och klamrar placerade så att koden blir lättläst. I möjligaste mån ska kodkonventioner gällande namn på variabler och klasser följas (framför allt: tydliga namn, liten begynnelsebokstav för variabler och metoder samt stor begynnelsebokstav för klasser).
- Du behöver inte hantera exceptions men övrig rimlig felhantering är önskvärd. Exempelvis ska programmet inte krascha om användaren matar in kontonummer eller menyval som inte finns. Det är inte nödvändigt att hantera indata av fel typ (exempelvis att användaren matar in bokstäver då programmet förväntar sig ett heltal).

Tips och övriga anvisningar

- I vissa lösningar kan Scannern tyckas bete sig konstigt vid inläsning av omväxlande tal och strängar. Detta kan exempelvis visa sig i att inlästa strängar blir tomma.

  Specifikt uppstår problemet när man anropar metoden `nextLine()` i `Scanner`. Metoden läser en sträng till och med nästa radmatning, dvs en sträng som avslutas med Return. Det passar ju bra om man ska läsa in namn, eftersom de innehåller mellanslag.

  Begrunda följande exempel, där användaren tänks mata in ett heltal och ett namn:

```
public class ScannerExample {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    System.out.print("ange ett heltal: ");
    int n = scan.nextInt();
    System.out.println("ange ett namn: ");
    String s = scan.nextLine();
    System.out.println("n = " + n);
    System.out.println("s = " + s);
  }
}
```

  När man kör programmet matar man in ett heltal och trycker på Return. Därefter hinner man emellertid aldrig mata in något namn, utan får ut följande:

```
ange ett heltal: 57
ange ett namn:
n = 57
s =
```

  Problemet är följande: metoden `nextLine()` läser tecken fram till Return. Användaren skrev Return efter 57 ovan, och Scannern ser detta Return som en tom rad. Därför returnerar `nextLine()` direkt med en tom sträng.

  En enkel lösning är att lägga in ett extra `nextLine()` efter inläsningen av heltalet ovan, för att på så vis ”förbruka” radmatningen efter heltalet. I uppgiften passar detta typiskt då menyvalet (1-9) precis lästs in. I exemplet ovan skulle det kunna se ut så här:

```
...
int n = scan.nextInt();
scan.nextLine();  // läs förbi Return
System.out.println("ange ett namn: ");
String s = scan.nextLine();
...
```

  Se även avsnitt 3.10 (The Scanner Bug) i *Think Java*.

- Tänk på val av attribut i klasserna. En variabel som bara används i en metod behöver inte vara åtkomlig i hela objektet och ska således ej vara attribut. Fråga en handledare om du är osäker.
- Observera att man inte ska kunna ta ut eller överföra pengar på ett sådant sätt att kontots saldo blir negativt. Av specifikationerna framgår att klassen `BankAccount` inte ska hantera övertrasseringar, men däremot ska `BankApplication` inte tillåta någon tar ut för mycket (vilket också framgår av exemplen i bilagan).
- Observera att `Customer` ska ha både ett idNr och ett kundnummer, dessa båda ska alltså inte sättas till samma värde. Kundnummer motsvarar ett löpnummer som är internt för banken och idNr kan tänkas motsvara verklighetens personnummer.
- Programmet ska testas noggrant. Tänk på att den som rättar kan mycket väl testa på helt andra fall än de som förekommer i exemplen. Till er hjälp finns testfall och rättningsprotokoll sist i denna uppgiftsbeskrivning. Läs igenom detta innan ni börjar programmera.

Frivillig utökning av uppgiften

Gör först klart inlämningsuppgiftens obligatoriska delar. Därefter kan du, om du vill, utöka ditt program enligt följande.

- Då man söker kunder utifrån en del av deras namn (menyval 2) kan samma kund förekomma flera gånger i resultatet. (Att det är samma kund ser man på att inte bara namn och personnummer är detsamma, utan även kundnumret.) Detta är naturligtvis inte helt önskvärt. Utöka gärna ditt program så att varje kund bara förekommer högst en gång i utskriften från menyval 2.
- En annan utökning, som även kan underlätta testningen av programmet, är att implementera läsning från och skrivning till fil och på så vis ge möjlighet att spara kund- och kontoinformationen mellan körningarna.

Design

Följande klass ska användas för att lagra grundläggande information om en kontoinnehavare:

`Customer`

```
/**
 * Skapar en kund (kontoinnehavare) med namnet 'name' och id-nummer 'idNr'.
 * Kunden tilldelas också ett unikt kundnummer.
 */
Customer(String name, long idNr);

/** Tar reda på kundens namn. */
String getName();

/** Tar reda på kundens personnummer. */
long getIdNr();

/** Tar reda på kundens kundnummer. */
int getCustomerNr();

/** Returnerar en strängbeskrivning av kunden. */
String toString();
```

Följande klass ska användas för att lagra information om ett bankkonto:

`BankAccount`

```
/**
 * Skapar ett nytt bankkonto åt en innehavare med namn 'holderName' och
 * id 'holderId'. Kontot tilldelas ett unikt kontonummer och innehåller
 * inledningsvis 0 kr.
 */
BankAccount(String holderName, long holderId);

/**
 * Skapar ett nytt bankkonto med innehavare 'holder'. Kontot tilldelas
 * ett unikt kontonummer och innehåller inledningsvis 0 kr.
 */
BankAccount(Customer holder);

/** Tar reda på kontots innehavare. */
Customer getHolder();

/** Tar reda på det kontonummer som identifierar detta konto. */
int getAccountNumber();

/** Tar reda på hur mycket pengar som finns på kontot. */
double getAmount();

/** Sätter in beloppet 'amount' på kontot. */
void deposit(double amount);

/**
 * Tar ut beloppet 'amount' från kontot. Om kontot saknar täckning
 * blir saldot negativt.
 */
void withdraw(double amount);

/** Returnerar en strängrepresentation av bankkontot. */
String toString();
```

Följande klass ska användas som register för att administrera bankens samtliga konton:

`Bank`

```
/** Skapar en ny bank utan konton. */
Bank();

/**
 * Öppna ett nytt konto i banken. Om det redan finns en kontoinnehavare
 * med de givna uppgifterna ska inte en ny Customer skapas, utan istället
 * den befintliga användas. Det nya kontonumret returneras.
 */
int addAccount(String holderName, long idNr);

/**
 * Returnerar den kontoinnehavaren som har det givna id-numret,
 * eller null om ingen sådan finns.
 */
Customer findHolder(long idNr);

/**
 * Tar bort konto med nummer 'number' från banken. Returnerar true om
 * kontot fanns (och kunde tas bort), annars false.
 */
boolean removeAccount(int number);
```

```
/**
 * Returnerar en lista innehållande samtliga bankkonton i banken.
 * Listan är sorterad på kontoinnehavarnas namn.
 */
ArrayList<BankAccount> getAllAccounts();

/**
 * Söker upp och returnerar bankkontot med kontonummer 'accountNumber'.
 * Returnerar null om inget sådant konto finns.
 */
BankAccount findByNumber(int accountNumber);

/**
 * Söker upp alla bankkonton som innehas av kunden med id-nummer 'idNr'.
 * Kontona returneras i en lista. Kunderna antas ha unika id-nummer.
 */
ArrayList<BankAccount> findAccountsForHolder(long idNr);

/**
 * Söker upp kunder utifrån en sökning på namn eller del av namn. Alla
 * personer vars namn innehåller strängen 'namePart' inkluderas i
 * resultatet, som returneras som en lista. Samma person kan förekomma
 * flera gånger i resultatet. Sökningen är "case insensitive", det vill
 * säga gör ingen skillnad på stora och små bokstäver.
 */
ArrayList<Customer> findByPartofName(String namePart);
```

Implementation

Implementationen görs lämpligen stegvis, genom att du implementerar några metoder i taget och därefter provkör resultatet. Därefter implementerar och provkör du fler metoder. Förslagsvis följer du dessa steg i ditt arbete:

1. Skapa projektet i Eclipse.
2. Implementera klassen `Customer`.
3. Implementera klassen `BankAccount`.
4. Skapa klassen `BankApplication`, som ska innehålla programmets `main`-metod.

   Det är lämpligt att införa ytterligare metoder i `BankApplication`: exempelvis en metod för att driva hela programmet (t. ex. `runApplication`) som i sin tur anropar andra metoder beroende på vad som ska göras. Ha gärna en separat metod för att skriva ut menyn.

   Implementera tillräckligt för att kunna köra igång programmet, skriva ut menyn och låta användaren välja i den.
5. Implementera menyval 6 och 8, som motsvarar metoderna `addAccount` och `getAllAccounts` i klassen `Bank`. Återigen kan du införa metoder i `BankApplication`, exempelvis för att läsa in data och anropa `Bank` med dessa.

   Observera din sortering i `getAllAccounts` inte ska använda någon av Javas standardklasser för sortering, som t. ex. `Collections.sort` eller `Arrays.sort`.

   Observera också återigen att det inte är tillåtet att använda `Scanner` och `System.out` i klassen `Bank`.
6. Testa menyval 6 och 8 noga.

Tips: under testningen kan det bli enformigt att mata in uppgifter om konton om och om igen. Du kan lägga in satser i ditt program som anropar `addAccount` ett antal gånger, så att ditt program startar med ett antal färdiga testkonton. Låt dessa satser skapa kontona i icke-alfabetisk ordning (med avseende på kontoinnehavarnas namn), så att du ser att sorteringen i `getAllAccounts` fungerar.

7. Implementera menyvalen 1 och 2. De motsvaras av metoderna `findAccountsForHolder` och `findByPartOfName` i klassen `Bank`.

8. Lägg till några konton (om du inte införde testkonton enligt punkt 6 ovan) och testa menyvalen 1 och 2.

9. Implementera resten av menyvalen en åt gången. Testa efter varje.

10. Testa ditt program som helhet. Om du lade in satser för att automatiskt skapa testkonton (under punkt 6 ovan), ta bort dem.

    Kör först igång programmet utan att skapa några konton, och kontrollera att menyns alternativ fungerar även utan konton. Skapa därefter ett antal konton och kontrollera att programmets delar fungerar som förväntat.

    Glöm inte testa med orimliga indata och kontrollera att programmet hanterar dessa på ett rimligt sätt. Gå tillbaka och åtgärda de fel du upptäcker.

Redovisning

När du gått igenom alla moment ovan, och testat noga, är det dags att redovisa. Förbered dig inför redovisningen. Om du hunnit glömma vad du gjort så repetera innan redovisningen – du förväntas kunna förklara ditt program!

Exempel på körning av programmet

Här visas en möjlig exempelkörning av programmet. Data som matas in av användaren visas i fetstil. Ditt program behöver inte efterlikna dessa utskrifter i exakta detaljer, men den övergripande strukturen ska vara densamma. De data som matas in av användaren har markerats i fetstil.

Ett antal konton skapas, och pengar sätts in på ett par av dem. Ett stort belopp överförs mellan två konton. Slutligen tas de flesta kontona bort, och endast ett finns kvar då programmet avslutas.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 6
namn: Doris Kruth
id: 4111194444
konto skapat: 1001
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 6
namn: Charles Ingvar Jönsson
id: 3705209999
```

```
konto skapat: 1002
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 6
namn: Jakob Morgan Rockefeller Wall-Enberg, Jr.
id: 2306207777
konto skapat: 1003
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 6
namn: Jakob Morgan Rockefeller Wall-Enberg, Jr.
id: 2306207777
konto skapat: 1004
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 2
namn: ROCK
Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103
Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 2
namn: or
Doris Kruth, id 4111194444, kundnr 101
Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103
Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 3
konto: 1003
belopp: 1000000.0
konto 1003 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 1000000.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
```

```
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 3
konto: 1001
belopp: 50.0
konto 1001 (Doris Kruth, id 4111194444, kundnr 101): 50.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 4
från konto: 1001
belopp: 900.0
uttaget misslyckades, endast 50.0 på kontot!
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 5
från konto: 1003
till konto: 1001
belopp: 500000.0
konto 1003 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 500000.0
konto 1001 (Doris Kruth, id 4111194444, kundnr 101): 500050.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 4
från konto: 1001
belopp: 900.0
konto 1001 (Doris Kruth, id 4111194444, kundnr 101): 499150.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 1
id: 2306207777
konto 1003 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 500000.0
konto 1004 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 0.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 8
konto 1002 (Charles Ingvar Jönsson, id 3705209999, kundnr 102): 0.0
```

```
konto 1001 (Doris Kruth, id 4111194444, kundnr 101): 499150.0
konto 1003 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 500000.0
konto 1004 (Jakob Morgan Rockefeller Wall-Enberg, Jr., id 2306207777, kundnr 103): 0.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 7
konto: 1002
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 7
konto: 1003
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 7
konto: 1004
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 7
konto: 87654
felaktigt kontonummer!
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 8
konto 1001 (Doris Kruth, id 4111194444, kundnr 101): 499150.0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1. Hitta konto utifrån innehavare
2. Sök kontoinnehavare utifrån (del av) namn
3. Sätt in
4. Ta ut
5. Överföring
6. Skapa konto
7. Ta bort konto
8. Skriv ut konton
9. Avsluta
val: 9
```

3 Matlab-laborationer – anvisningar

I dessa laborationer får du träna praktiskt på att använda Matlab för att lösa numeriska problem med dator.

Matlab-laborationerna ingår **bara för EDAA55 (F1/I1)**. Matlab-laborationerna är schemalagda till läsperiod VT1 2022. Uppgifterna finns i Moodle, under de veckor det gäller.

Laborationerna är obligatoriska

De tre Matlab-laborationerna måste vara godkända för att du ska kunna få poäng i Ladok (1,5hp) och slutföra kursen Programmeringsteknik. (De behöver dock inte vara godkända för att du ska få skriva tentamen.)

Jobba två och två, redovisa i Zoom

Laborationerna löses och redovisas i grupper om två studenter.

Ni genomför laborationen i förväg, och redovisar den vid ett särskilt Matlab-redovisningstillfälle i Zoom. Det finns ett sådant redovisningstillfälle i veckan, under tre veckor (ett för varje laboration). Ni kommer att få möjlighet att boka en tid för denna redovisning.

Vi erbjuder ingen möjlighet att redovisa Matlab-uppgifter på Java-laborationstillfällena.

Det ordnar sig: det finns hjälp att få

Om ni kör fast kan ni använda Matlab-resurstiden (se TimeEdit) för att få hjälp.

Om ni ändå inte hinner med laborationen under veckan så får ni i nödfall redovisa vid nästa Matlab-resurstid. Efter den sista (tredje) laborationen anordnas ett uppsamlingstillfälle, för dem som har någon enstaka laboration kvar. Annars finns alltid möjligheten att komplettera i samband med nästa Matlab-omgång. (Momentet ges två gånger per läsår, i november–december respektive februari–mars.)

Använd tiden väl

Inför varje redovisning behöver ni planera ert arbete tillsammans. Liksom Java-laborationerna kräver Matlab-laborationerna också förberedelser. (Matlab-momentet omfattar nominellt 28h självstudier.) Förberedelseuppgifter är angivna för respektive laboration på sidan ovan.