Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 1 – del 2

Datateknik GR(B), Java III, 7,5 högskolepoäng

Syfte:	Att kunna hantera trådar i Swing på ett korrekt sätt. Att förstå vad begreppet trådsäkert innebär. Att veta vad händelsetråden är. Att kunna skapa en SwingWorker för att utföra bakgrundsarbeten.
Att läsa:	Kursbok, Kapitel 13.4 och framåt The Java™ Tutorial, Concurrency in Swing http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

Java III

Lektion 1 – del 2
- Trådar i Swing
- Händelsetråden
- SwingWorker

Trådar i Swing

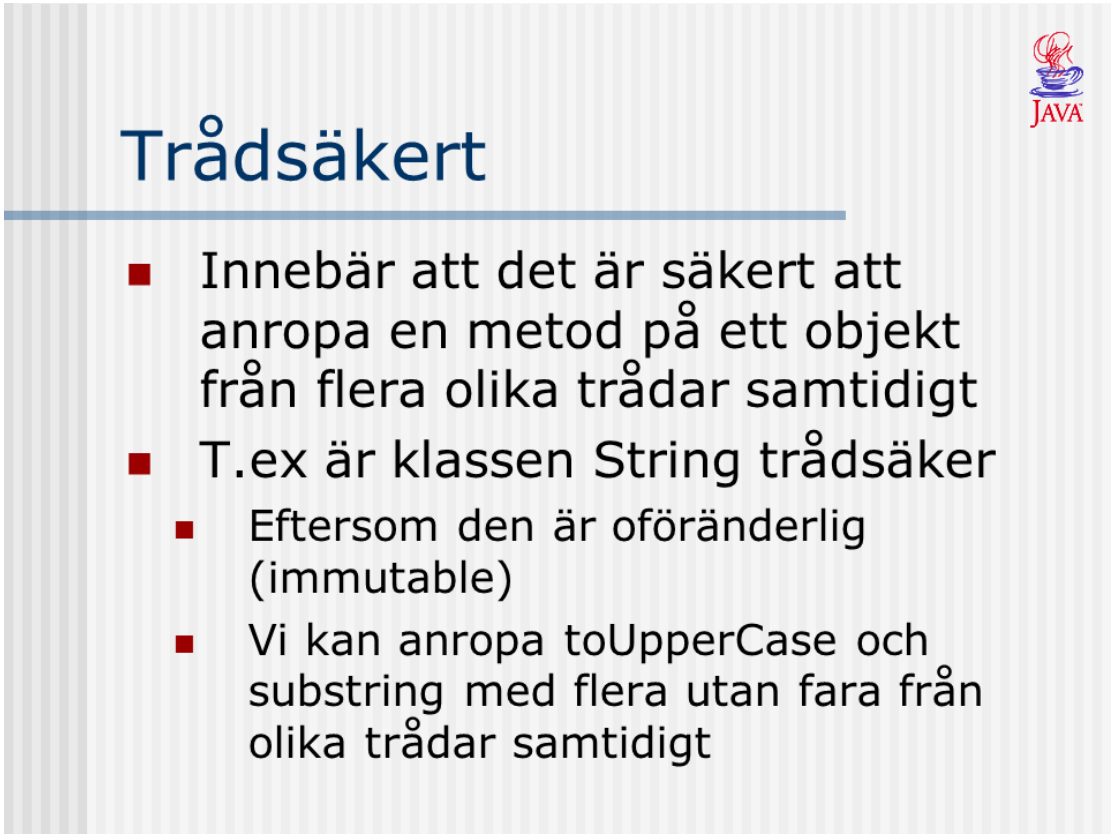
- Viktigt att känna till hur trådar ska användas i Swing för att:
  - Få GUI som reagerar snabbt på användarens interaktioner
  - Få GUI som aldrig ”låser sig”
  - Inte hantera långa operationer i fel tråd
  - Avgöra när vi behöver skapa nya trådar

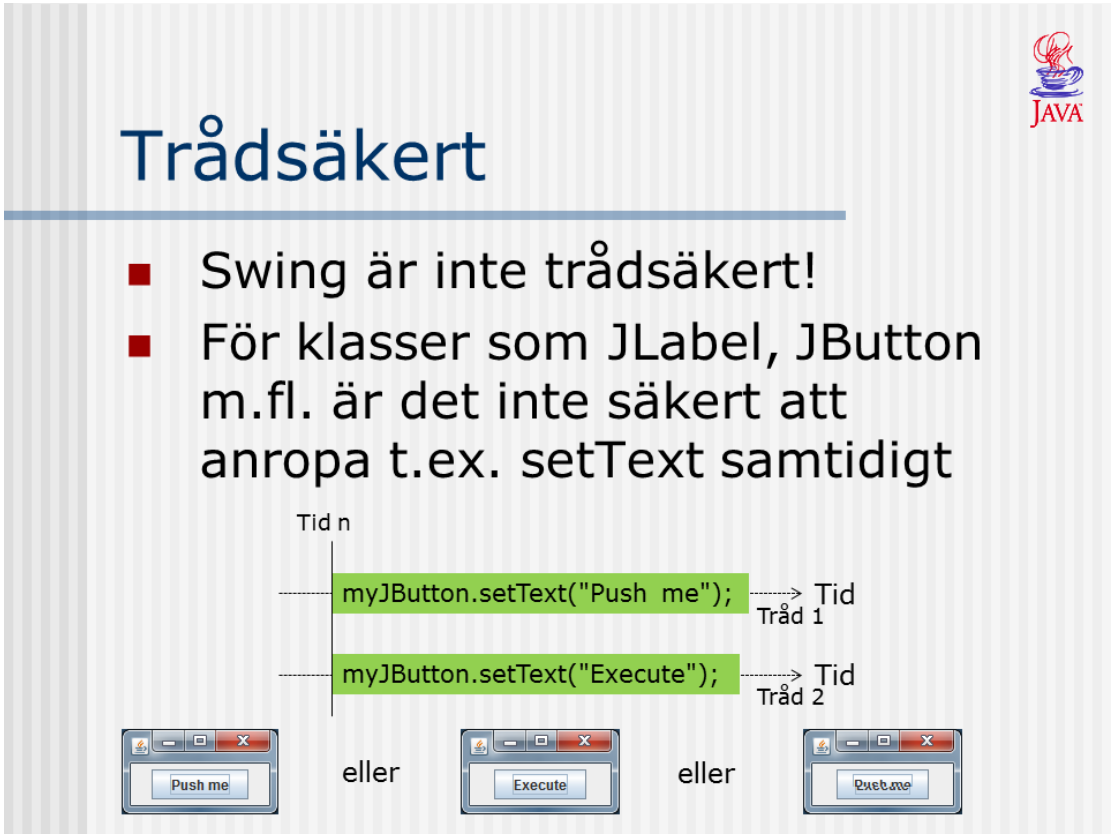
När vi skapar en applikation som använder Swing för dess grafiska användargränssnitt kommer vi normalt i kontakt med tre olika typer av trådar. Den första typen är huvudtråden (initial thread eller main-thread) som är den tråd som exekverar metoden `main` i en applikation eller metoderna `init` och `start` i en Applet. Denna tråd skapas automatiskt när ett program startas.

I en applikation som använder Swing har main-metoden normalt inte mycket att göra utan används endast för att skapa det grafiska användargränssnittet för att därefter avslutas. I en Applet har huvudtråden till uppgift att anropa metoderna `init` och `start`. Beroende på hur den virituella Javamaskinen är implementerad kan det vara två eller tre trådar inblandad i uppstarten av en applet.

Den andra typen av tråd kallas för händelsetråd (event dispatch thread). Den här tråden ansvarar för att exekvera all kod för händelsehantering samt för att uppdatera det grafiska användargränssnittet. Denna tråd skapas automatiskt så snart vi använder kod som till exempel skapar ett fönser (`JFrame`).

Den tredje typen av trådar kallas vanligtvis för bakgrundstrådar men kan även heta arbetstrådar (worker threads). Detta är trådar vi som programmerare själva måste skapa. De används för uppgifter som är tidskrävande som till exempel att ladda hem stora filer från nätverket.


Trådsäkert
Innebär att det är säkert att anropa en metod på ett objekt från flera olika trådar samtidigt
T.ex är klassen String trådsäker
Eftersom den är oföränderlig (immutable)
Vi kan anropa toUpperCase och substring med flera utan fara från olika trådar samtidigt
JAVA



Till skillnad från `String` så är klasser i Swing inte trådsäkra. Det är med andra ord inte säkert att till exempel anropa metoden `setText` på en `JButton` från flera olika trådar samtidigt. Resultatet av något sådant är oförutsägbart. Närmare bestämt kan problem med "thread interference" eller "memory consistency errors" uppstå. Detta är dock inget vi tittar närmare på i kursen. Den som är intresserad kan dock läsa mer om det på följande sidor.

https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html
https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html

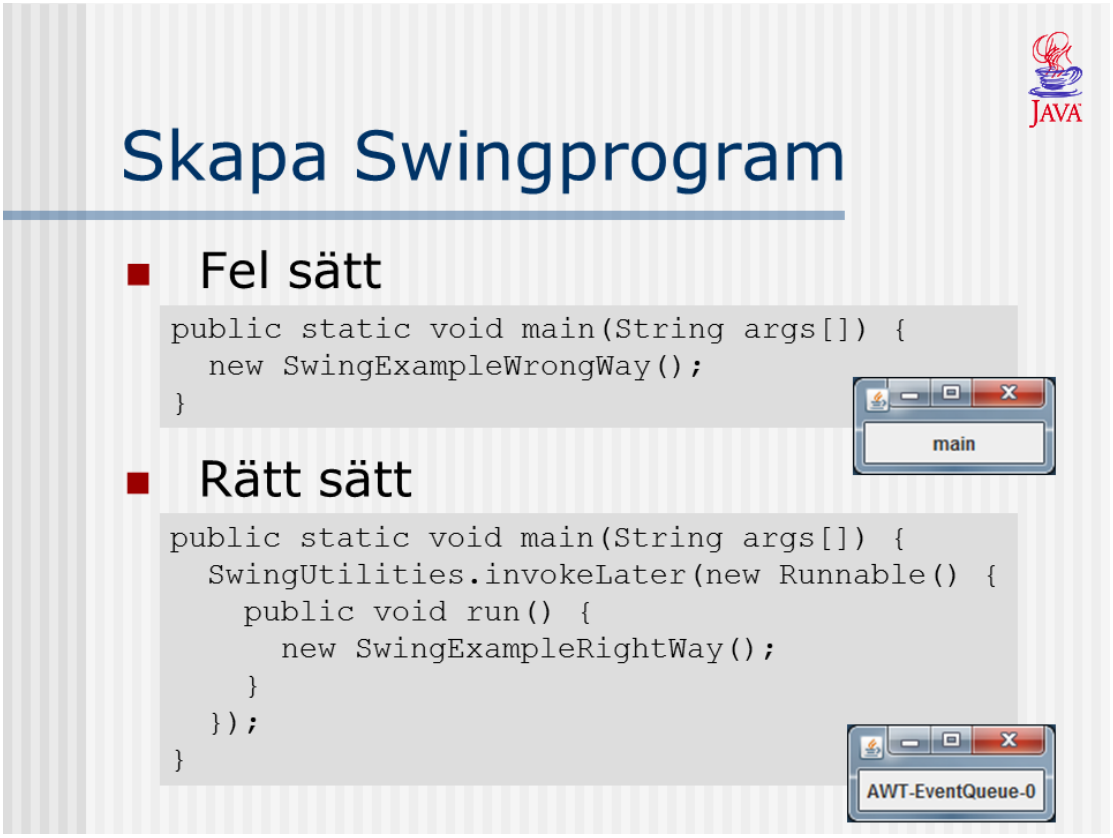
Trådsäkert

- De flesta metoder i Swing-komponenter är inte trådsäkra
- API specificerar vissa metoder som trådsäkra:
  - repaint
  - revalidate
  - Med flera
- Övriga måste exekveras av händelsetråden!

Händelsetråden

- Event dispatch thread
- All kod för att förändra Swing-komponenter och kod för händelsehantering ska utföras av händelsetråden!
- Exekverar t.ex. kod i lyssnar-metoder som actionPerformed
- Vi ska aldrig själva anropa en lyssnarmetod


Händelsetråden
Program som inte uppdaterar GUI i händelsetråden
Fungerar i stort sett alltid utan problem
Men kan lida av oförutsägbara problem som är svåra att felsöka och återskapa
Använd SwingUtilities
invokeAndWait
invokeLater



När vi skapar en applikation som använder Swing skriver vi en klass som ärver `JFrame`. I dess konstruktor har vi sen lagt all kod för att skapa komponenterna, placera ut dem i fönstret och sätta egenskaper på fönstret (som titel, storlek och placering). För att skapa och visa användargränssnittet har vi i main-metoden helt enkelt skapat ett nytt objekt av vår klass, som sist i konstruktorn gör vi fönstret synligt med anropet `setVisible(true)`.

Detta sätt att skapa och visa ett GUI är fel eftersom tråden som skapar användargränssnittet är samma tråd som exekverar main, det vill säga huvudtråden. Du kan se exempel på detta genom att titta på exempel 1, **SwingExempleWrongWay.java**. I klassen används följande kod för att ta reda på namnet på den tråd som exekverar koden:

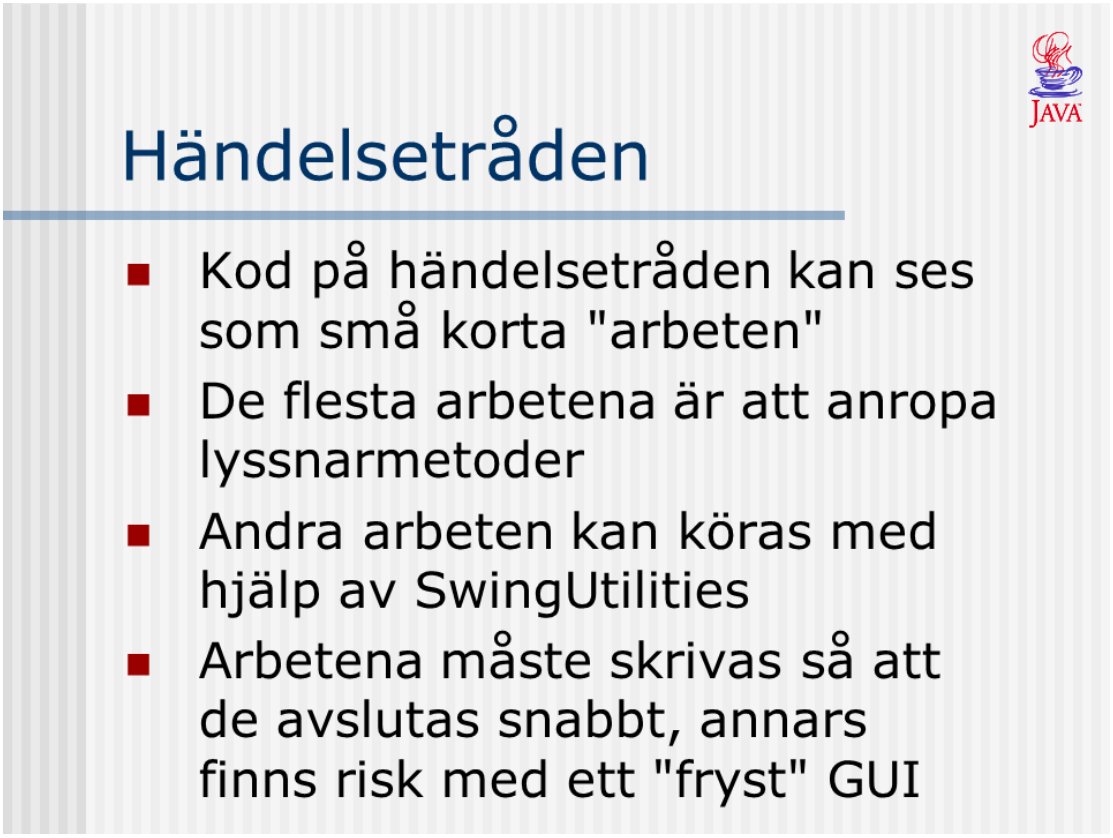
```
String name = Thread.currentThread().getName();
```

Namnet på tråden visas sen i en `JLabel` i fönstret. Som du kommer se står det "main" i fönstret när exemplet exekveras.

För att exekvera kod i händelsetråden finns klassen `SwingUtilities` och dess metoder `invokeAndWait` samt `invokeLater`. Både dessa tar som parameter ett `Runnable`-objekt. Metoderna kommer att se till så att detta `Runnable`-objekts `run`-metod körs av händelsetråden. Skillnaden mellan de båda metoderna är att `invokeLater` returnerar genast och exekveringen fortsätter direkt i den anropande tråden. När metoden `invokeAndWait` används blockerar den istället den anropande tråden så att den måste vänta till dess att all kod i `Runnable`-objektet har exekverats klart.

Det korrekta sättet att skapa en applikation som använder Swing är därför att använda `SwingUtilities` och någon av metoderna `invokeAndWait` eller `invokeLater`. Enklast är att skapa klassen som representerar `Runnable`-objektet som en anonym klass. I dess run-metod skapar vi ett nytt objekt av vår klass och koden som skapar användargränssnittet kommer nu att köras av händelsetråden.

Ta en titt på exempel 2, **SwingExempleRightWay.java** och du kommer att se att utskriften i fönstret blir AWT-EventQueue-0 som är det namn som används internt för händelsetråden.

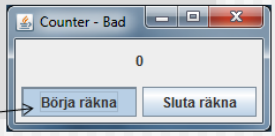

Händelsetråden
Kod på händelsetråden kan ses som små korta "arbeten"
De flesta arbetena är att anropa lyssnarmetoder
Andra arbeten kan köras med hjälp av SwingUtilities
Arbetena måste skrivas så att de avslutas snabbt, annars finns risk med ett "fryst" GUI

Exempel - Dåligt

- Räkna upp ett heltal i ett GUI

```
public class CounterBad extends JFrame implements ActionListener {
  private boolean stop = false;
  private int counter = 0;
  ...
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == startButton) {
      stop = false;
      for (int i = 0; i < 2000000; i++) {
        if (stop) { break; }
        counter++;
        // Uppdatera GUI (sker i händelsetråden så helt ok)
        counterLabel.setText(Integer.toString(counter));
      }
    }
    else if (e.getSource() == stopButton) {
      stop = true;
    }
  }
}
```

Knappens tillstånd är "fryst"



Som ett första exempel på händelsetråden ska vi titta på en applikation som i ett grafiskt användargränssnitt räknar upp och visar ett heltal i en `JLabel`. Detta är ett dåligt försök där uppräkningen sker i händelsetråden vilket resulterar i att knappens tillstånd kommer att frysa så länge som uppräkningen pågår (knappen ser ut att vara intryckt och sluta-knappen går inte heller att trycka på).

Som instansvariabler i klassen har vi, förutom variabler för komponenterna, en `boolean` som används som flagga för att avgöra när uppräkningen ska sluta. Vi har även ett heltal som används för att räkna upp värdet och visa det i fönstret. I klassens konstruktor skapar vi användargränssnittet och i main-metoden använder vi `SwingUtilities` och dess metod `invokeLater` för att skapa gränssnittet på händelsetråden.

I metoden `actionPerformed`, som kommer att anropas av händelsetråden när vi trycker på någon av knapparna, har vi en `for`-loop som snurrar i 2 miljoner varv. Vi börjar med att kontrollera om variabeln `stop` är lika med `true`, vilket den blir när vi klickar på stoppknappen, och avbryter i så fall loopen. Därefter räknar vi upp värdet på heltalet och visar det i vår `JLabel` genom att anropa dess metod `setText`. Eftersom denna lyssnarmetod anropats av händelsetråden är det riskfritt att uppdatera GUI här.

Ta en titt på exempel 3, **CounterBad.java** och testkör det. När du trycker på knappen Börja räkna kommer hela gränssnittet att frysa och inget händer så länge som uppräkningen pågår. Knappen ser ut att vara nertryckt, vår `JLabel` uppdateras inte, det går inte att klicka på knappen Sluta räkna och inte heller går det att ändra fönstrets storlek. Allt detta på grund av att händelsetråden, som är tänkt att användas för att uppdatera GUI, är upptagen med vårt långa arbete att räkna upp ett heltal 2 miljoner gånger (du får själv justera värdet i källkoden så uppgiften tar rimlig lång tid att utföra).

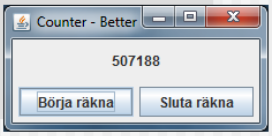
Exempel - Bättre

- Gör uppräkningen i en ny tråd

```
if (e.getSource() == startButton) {
  stop = false;
  Thread t = new Thread() {
    public void run() {
      for (int i = 0; i < 2000000; i++) {
        if (stop) { break; }
        counter++;

        // Vi är inte längre i händelsetråden, måste använda
        // SwingUtilities och invokeLater för att uppdatera GUI
        SwingUtilities.invokeLater(new Runnable() {
          public void run() {
            counterLabel.setText(Integer.toString(counter));
          }
        });
    } }
  };
  t.start();
}
```

GUI uppdateras hyffsat under uppräkningen



I det här exemplet ser vi ett bättre sätt att lösa det på. I stället för att göra uppräkningen direkt i `actionPerformed`, vilket innebär i händelsetråden, skapar vi en ny tråd som får sköta uppräkningen. Det enda händelsetråden nu gör är att skapa en tråd och starta den. Det är ett arbete som går väldigt snabbt att utföra och därefter blir händelsetråden ledig för att utföra andra arbeten (som att uppdatera GUI).

I den nya tråden kan vi inte direkt uppdatera texten på vår `JLabel` eftersom det skulle bryta mot regeln om att all GUI ska uppdateras av händelsetråden. Vi måste därför använda oss av `SwingUtilities` och dess metod `invokeLater`. Värt att veta är att alla schemalagda körningar via `invokeLater` alltid körs i den ordning de hamna på kön. Dvs ett senare anrop till `invokeLater` kan aldrig startas innan samtliga tidigare kö-lagda arbeten har bearbetats.

Ta nu en titt på exempel 4, **CounterBetter.java** och testkör det. Som du kommer att märka set knappen inte längre nertryckt ut och GUI reagerar på övriga yttre händelser (som att ändra storlek och trycka på sluta-knappen). Dock sker uppdateringen av vår `JLabel` väldigt sporadiskt och sker inte alls med flyt (jämn och fin takt). Även om att uppdatera texten i en `JLabel`, genom att anropa `setText`, är ett väldigt kort arbete kommer våra 2 miljoner anrop leda till att händelsetråden får svårt att hinna med.

Vi skulle behöva en lösning som uppdaterade vårt GUI med jämn takt även om vi gör många, många uppdateringar. Det är här klassen `SwingWorker` kommer in som ett bra exempel.

SwingWorker

- Kan starta ett långt arbete i en bakgrundstråd och returnera
  - Delresultat till händelsetråden
  - Slutresultatet till händelsetråden
- Hjälper till att hantera interaktion mellan händelsetråd och bakgrundstrådar
- Är en abstrakt klass som måste ärvas (tips: inre klass)

SwingWorker

```
public abstract class SwingWorker<T,V>
  implements RunnableFuture
```

- Är en generisk klass med 2 generiska parametrar
- T är den typ resultatet, som arbetet resulterar i, ska ha
- V är den typ delresultatet, som används för uppdateringar av GUI, ska ha

SwingWorker

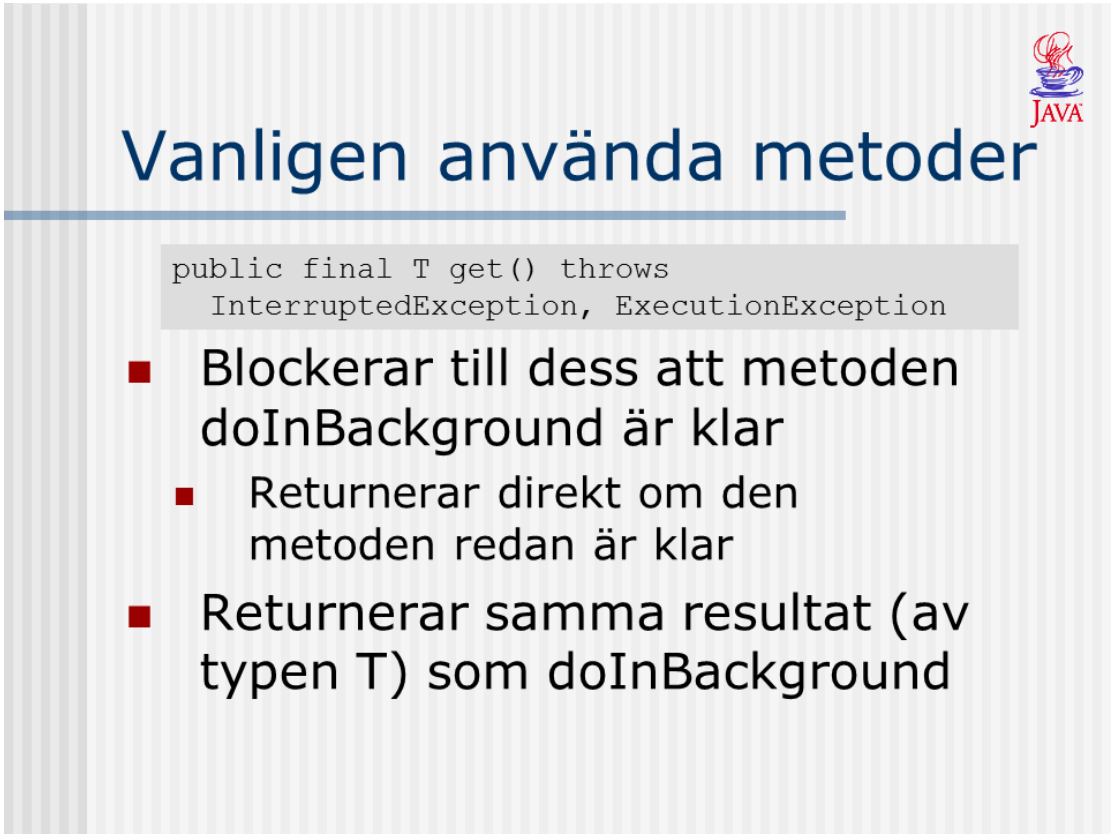
```
public abstract class SwingWorker<T,V>
  implements RunnableFuture
```

- Gränssnittet RunnableFuture är en kombination av:
  - Runnable
    - Metoden run
  - Future
    - Metoderna get, cancel, isDone och isCanselled

Vanligen använda metoder

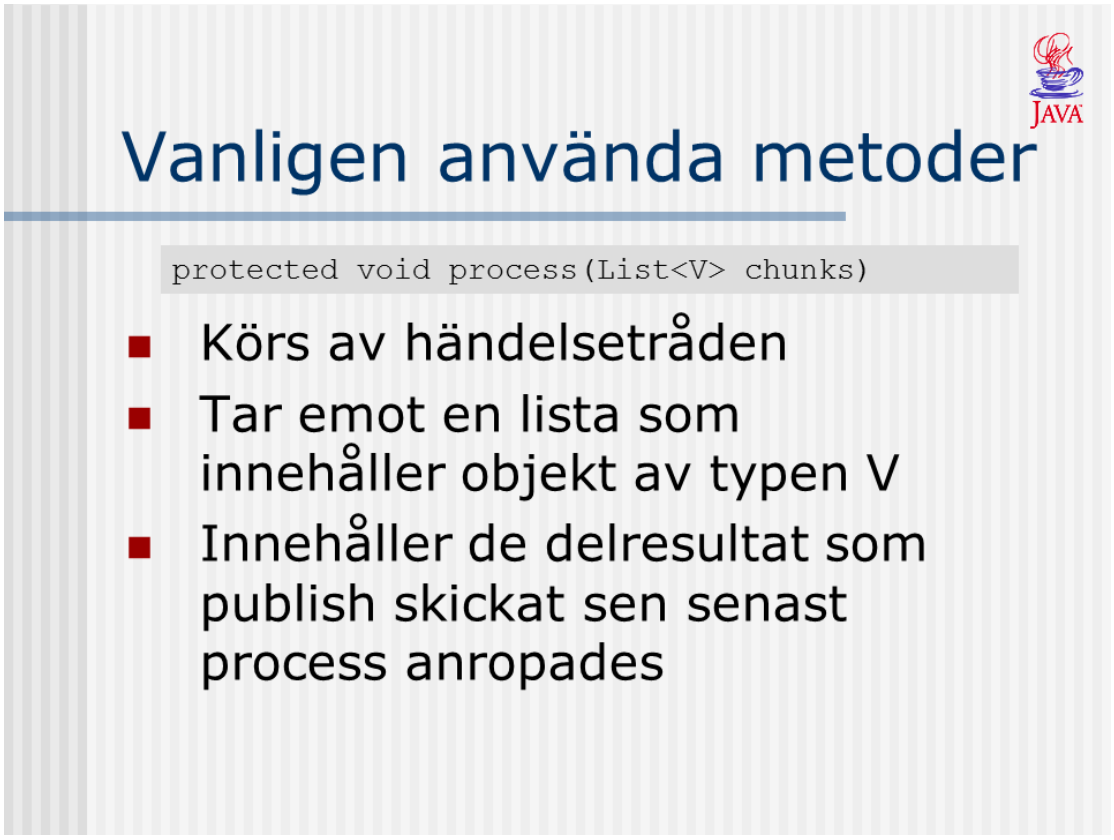
```
protected abstract T doInBackground()
```

- Den metod som utför arbetet
- Sker i en separat tråd
- Returnerar ett resultat av typen T när arbetet är klart
- Kan endast anropas en gång (måste skapa ett nytt objekt om arbetet ska utföras igen)


Vanligen använda metoder
public final T get() throws
InterruptedException, ExecutionException
Blockerar till dess att metoden doInBackground är klar
Returnerar direkt om den metoden redan är klar
Returnerar samma resultat (av typen T) som doInBackground

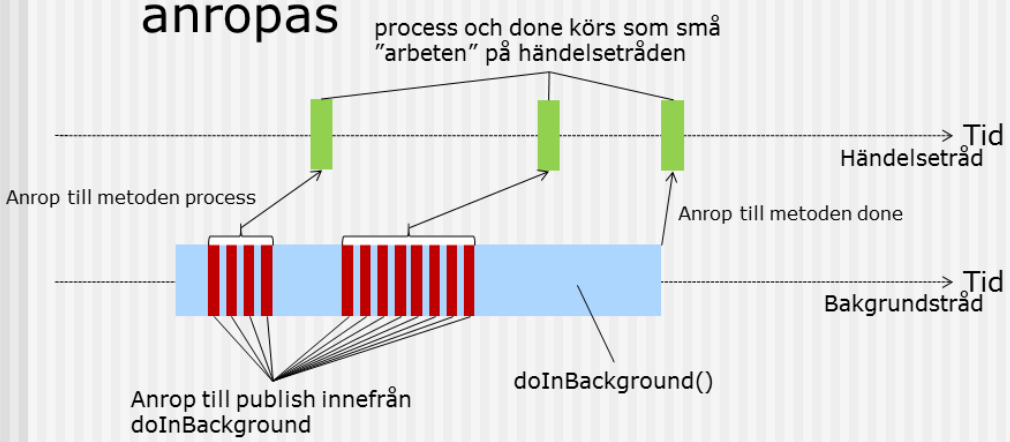

Vanligen använda metoder
protected void done()
Anropas automatiskt när doInBackground är klar
Exekveras av händelsetråden
Ok att uppdatera grafiska komponenter från denna metod
Inte ett måste att överskuggas
Vanligt att anropa get härifrån för att få slutresultatet av arbetet


Vanligen använda metoder
protected final void publish(V... chunks)
Anropas från doInBackground för att skicka delresultat
Skickar data av typen V (som är en vararg). Ex:
publish("1");
publish("2", "3");
publish("4", "5", "6");
Resulterar i ett litet "arbete" som utförs på händelsetråden


Vanligen använda metoder
protected void process(List<V> chunks)
Körs av händelsetråden
Tar emot en lista som innehåller objekt av typen V
Innehåller de delresultat som publish skickat sen senast process anropades

publish --> progress

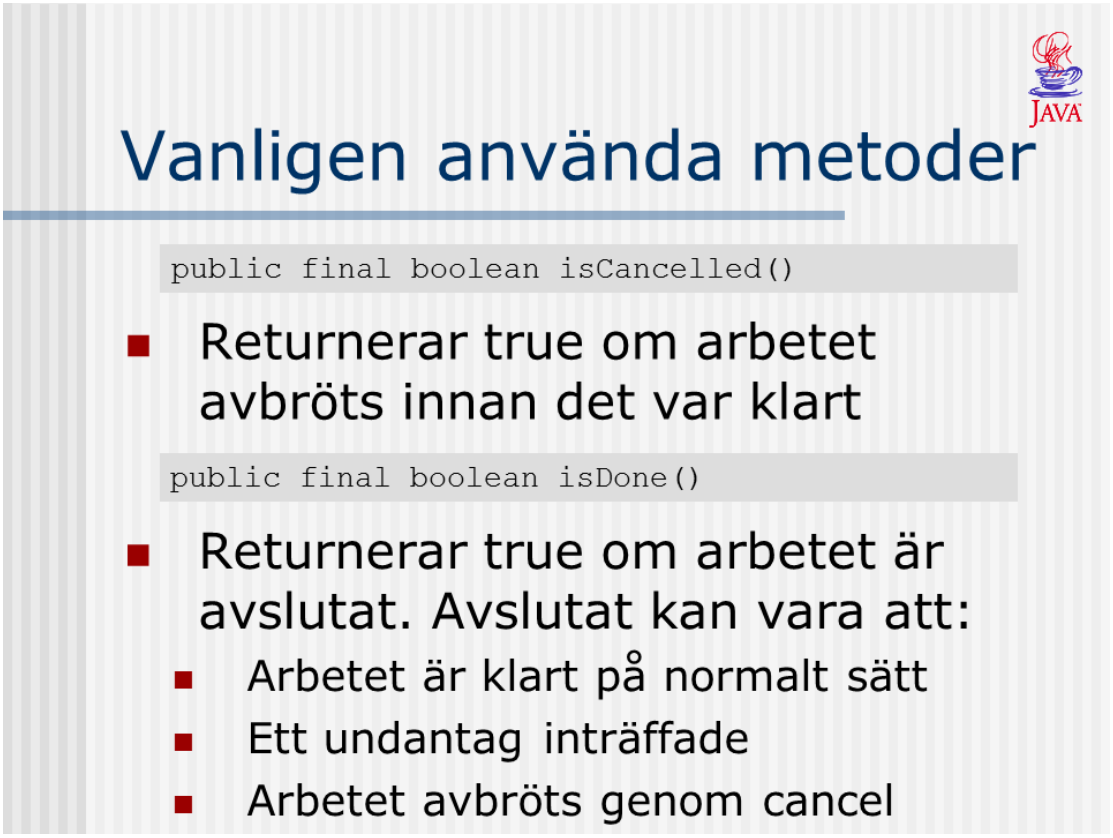
- Om publish anropas ofta kan data ackumuleras innan process anropas



Vanligen använda metoder

```
public final boolean cancel(
  boolean mayInterruptIfRunning)
```

- Används för att försöka avbryta ett arbete i förtid
  - true – avbryt tråden som utför arbetet
  - false – låt pågående arbete göra klart
- Returnerar true om arbetet avbröts, annars false


Vanligen använda metoder
public final boolean isCancelled()
Returnerar true om arbetet avbröts innan det var klart
public final boolean isDone()
Returnerar true om arbetet är avslutat. Avslutat kan vara att:
Arbetet är klart på normalt sätt
Ett undantag inträffade
Arbetet avbröts genom cancel


Vanligen använda metoder
public final void execute()
Startar exekveringen av doInBackGround
Schemalägger denna SwingWorker att exekveras i en bakgrundstråd
Finns begränsat antal bakgrundstrådar
Om alla bakgrundstrådar är upptagna med andra SwingWorker kommer denna att placeras i en kö
Anropa denna metod endast en gång per objekt!

Exempel 1

```
public class TestSwingWorker extends SwingWorker<Integer, String> {
  protected Integer doInBackground() {
    publish("Start");
    publish("Halvvägs");
    publish("Slutfört");
    return 1;
  }

  protected void process(java.util.List<String> chunks) {
    for (String result : chunks) {
      System.out.println("Status: " + result);
    }
  }

  protected void done() {
    Integer finalResult = get();
    System.out.println("Slutresultat: " + finalResult);
  }
}
```

Vi börjar med ett första väldigt enkelt exempel på en `SwingWorker`. Trots sin enkelhet demonstrerar exemplet ändå väl hur strukturen på en `SwingWorker` ser ut. Vi börjar med att skriva en klass `TestSwingWorker` som ärver från klassen `SwingWorker`. Den första generiska parametern som klassen har är en `Integer` och denna anger vilket slutresultat arbetet har. Dvs vilket returvärde metoden `doInBackground` har. Den andra generiska parametern är en `String` och denna anger vilken typ delresultaten ska ha. Dvs vilken typ listan, som används som parameter till metoden process, ska ha.

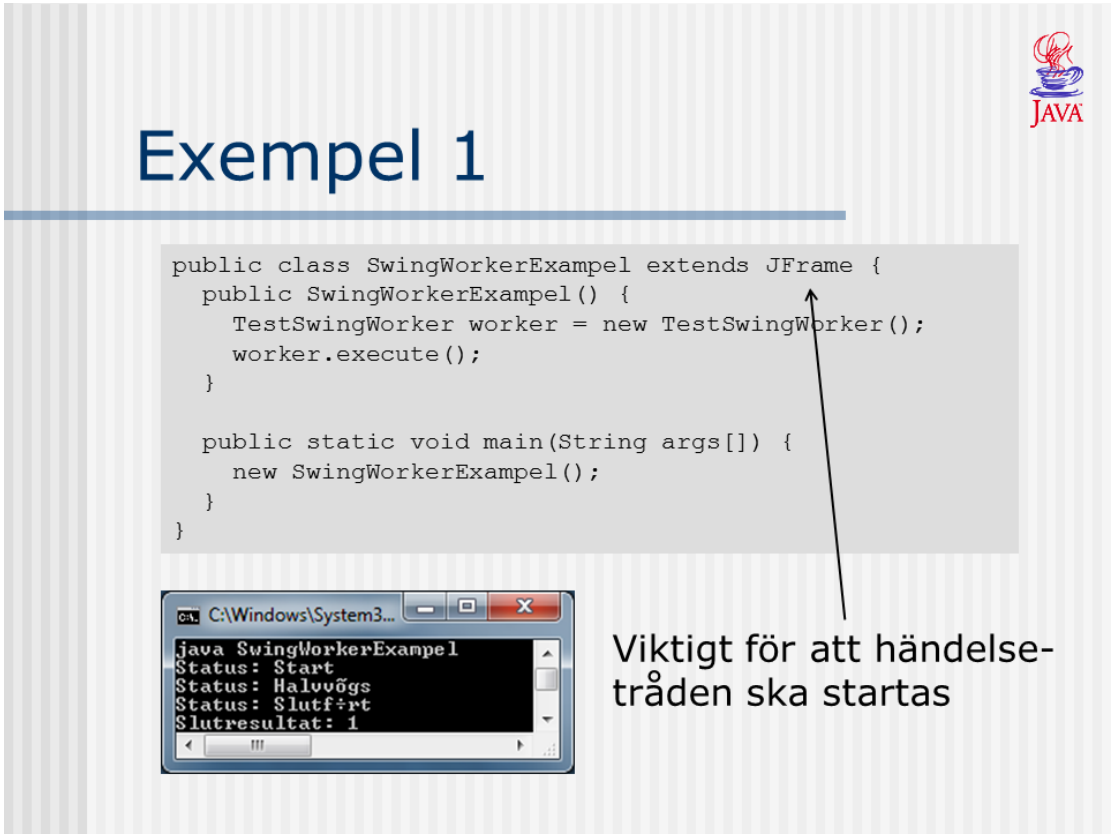
Därefter överskuggar vi metoden `doInBackground`. Det är i denna metod själva arbetet ska utföras. Arbetet sker som bekant i en ny separat tråd (om alla bakgrundstrådar är upptagna med andra `SwingWorker` kommer denna att placeras i en kö till dess att en tråd blir ledig). Eftersom arbetet utförs i en tråd som inte är händelsetråden, får vi inte från denna metod anropa metoder på grafiska komponenter. Behöver vi uppdatera status på hur arbetet fortlöper kan vi istället använda oss av metoden `publish`. Den datatyp vi använder som argument till `publish` anges av den andra generiska parametern som klassen har. Dvs i det här fallet en `String`. Det enda vi gör i det här exemplet är att publicera delresultaten Start, Hälvvägs och Slutfört innan vi returnerar att bakgrundsarbetet är klart. I exemplet returnerar vi 1 som ett godtyckligt heltal.

När vi anropar metoden `publish` kommer det att schemaläggas som en liten uppgift på händelsetråden. Uppgiften är väldigt kort, nämligen enbart att anropa metoden process. Fördelen är som sagt att det sker på händelsetråden och från process är det därför säkert att uppdatera grafiska komponenter. Som parameter till den överskuggade metoden har vi en lista som innehåller objekt av typen `String` (den

andra generiska parametern till klassen). Att det är en lista beror på att det av effektivitetsskäl kan vara så att flera i rad snabba anrop till `publish` kan ackumuleras till ett enda anrop till `process`. I vårt exempel loopar vi igenom denna lista och skriver ut delresultatet på skärmen med `System.out`. I stället för att publicera alla delresultat är det även vanligt att endast publicera det senaste resultatet i listan:

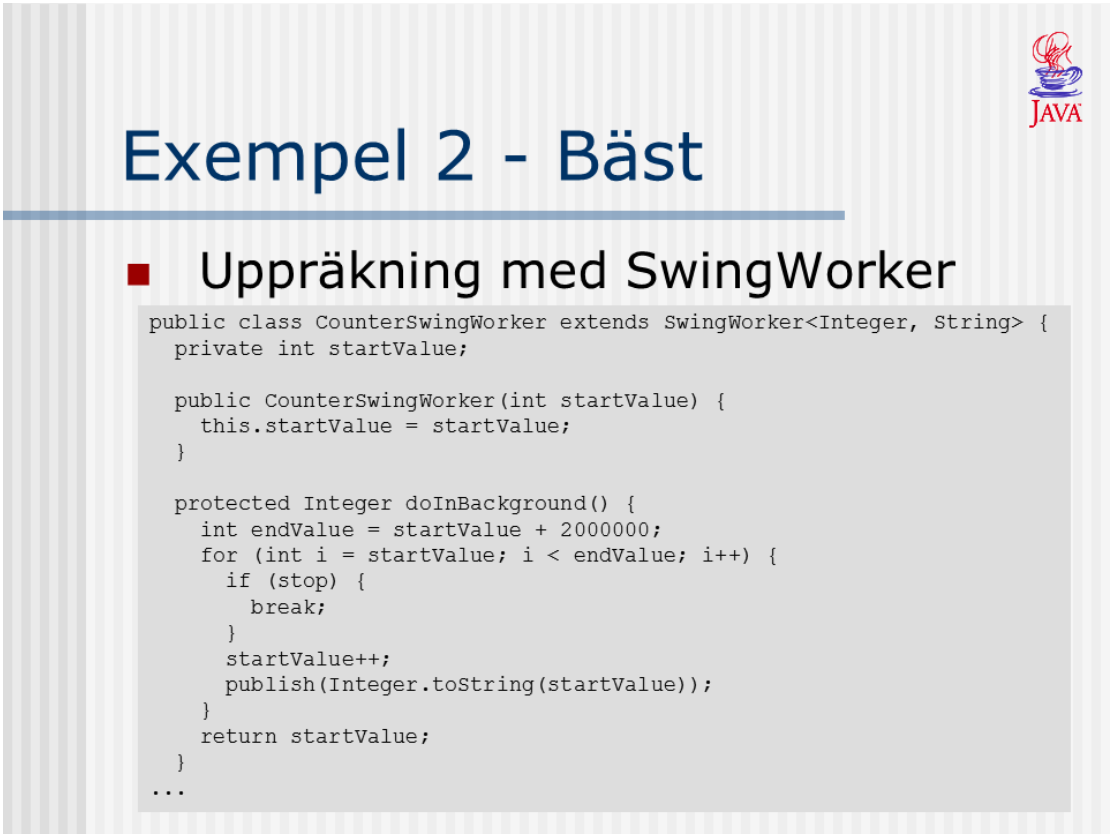
```
String latestResult = chunks.get(chunks.size() - 1);
```

Den sista metoden vi överskuggar är `done`. Det är denna metod som anropas när vi returnerar från metoden `doInBackground`. Precis som metoden process är även detta en metod som exekveras av händelsetråden. Härifrån kan vi med andra ord utan problem uppdatera vårt grafiska användargränssnitt. För att få resultatet som `doInBackground` returnerade anropar vi metoden `get`. Vi lagrar det returnerade värdet i en variabel av samma datatyp som returvärdet `doInBackground` har.



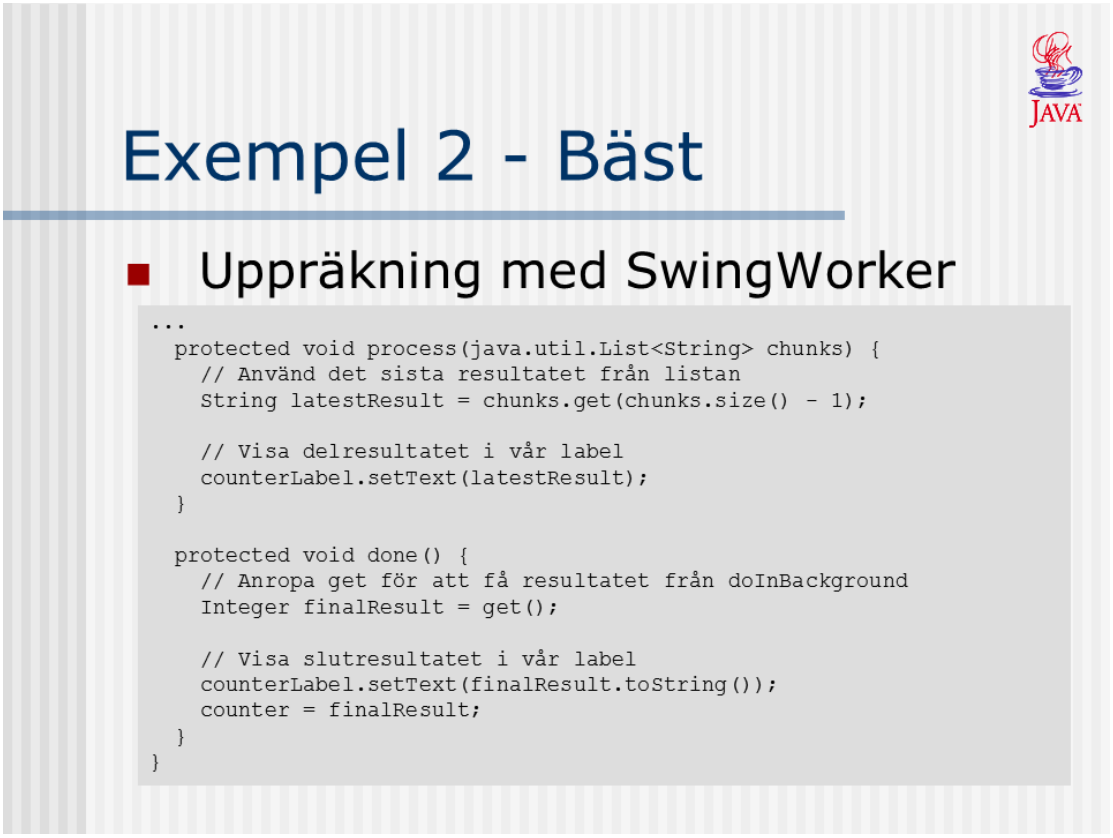
För att testa vår `SwingWorker` skriver vi en klass som ärver `JFrame`. Att den ärver `JFrame` är viktigt för annars startas inte händelsetråden. Om inte händelsetråden är startad kommer en `SwingWorker` inte att fungera (`doInBackground` kommer aldrig att köras). I `main`-metoden skapar vi ett nytt objekt av vår klass. I konstruktorn skapar vi en instans av vår `SwingWorker` och därefter anropar vi metoden `execute`. Detta anrop leder i sin tur till att metoden `doInBackground` exekveras (schemalägger denna `SwingWorker` att exekveras i en av de tillgängliga bakgrundstrådarna).

Ta en titt på exempel 5, **SwingWorkerExampel.java** som följer med lektionen.



I det sista exemplet ska vi fortsätta med vår uppräknare. I stället för att starta en ny tråd i `actionPerformed`, som sköter uppräkningen, ska vi i det här exemplet använda oss av en `SwingWorker`. Klassen kallar vi `CounterSwingWorker` och den första generiska parametern är av typen `Integer` (vad metoden `doInBackground` ska returnera) och den andra är av typen `String` (vilken datatyp som ska användas som argument till `publish`). Klassen har en instansvariabel `startValue` som används för att lagra från vilket startvärde vi ska börja räkna upp. I konstruktorn tar vi emot och lagrar detta startvärde.

I metoden `doInBackground` börjar vi med att räkna ut vilket slutvärde vi ska räkna upp till. I en `for`-loop räknar vi därefter upp värdet på `startValue` med 1 ända till dess att vi når slutvärdet. I varje varv kontrollerar vi om vår `boolean stop` är satt till `true`, i så fall avbryter vi `for`-loopen. I varje varv publicerar vi även ett delresultat, det uppräknade värdet, genom att anropa `publish`. Sist i metoden, när hela arbetet är slutfört eller om `stop` är satt till `true`, returnerar vi det aktuella värdet i instansvariabeln `startValue`.



När vi anropat `publish` kommer händelsetråden i sin tur att anropa `process` och där är vi endast intresserad av det sista värdet i listan. Vi använder detta värde för att uppdatera vår `JLabel` som visar räknarens värde.

I metoden `done`, som anropas när metoden `doInBackground` returnerar, gör vi ett anrop till get för att få värdet som `doInBackground` returnerade. Vi använder även detta värde för att uppdatera vår `JLabel` som visar räknarens värde. Vi passar även på att tilldela slutresultatet till vår instansvariabel `counter`.

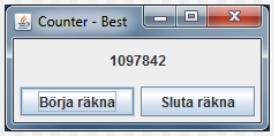
Exempel 2 - Bäst

- Uppräkning med SwingWorker

```
public void actionPerformed(ActionEvent e) {
  if (e.getSource() == startButton) {
    stop = false;

    // Skapa och starta vårt bakgrundsjobb. Skicka med nuvarande värde
    // på vår räknare till konstruktorn.
    CounterSwingWorker worker = new CounterSwingWorker(counter);
    worker.execute();
  }
  else if (e.getSource() == stopButton) {
    stop = true;
  }
}
```

GUI uppdateras jämt och fint under uppräkningen



När vi trycker på knappen Börja räkna skapar vi i `actionPerformed` en ny instans av `CounterSwingWorker`. Som argument till konstruktorn skickar vi med värdet på instansvariabeln `counter` (detta blir uppräknarens startvärde). Därefter anropar vi metoden `execute` på vår `SwingWorker` för att starta exekveringen av bakgrundsarbetet. Ett tryck på knappen Sluta räkna leder till att vi i `actionPerformed` sätter instansvariabeln `stop` till `true`. Som du minns från föregående bild leder detta till att `for`-loopen i `doInBackground` avbryts (dvs exekveringen bakgrundsarbetet avbryts).

Ta en titt på exempel 6, **CounterBest.java** och testkör det. Som du kommer att se uppdateras värdet i vår `JLabel` i en jämn och fin takt. Medan uppräkningen sker är vårt GUI i övrigt responsivt precis som vanligt (vi kan ändra storlek med mera utan några ”hack” eller ”frysningar”). Vi kan även använda knappen Sluta räkna för att avbryta uppräkningen.