Objektorientering och UI-programmering

Yngve Sundblad
y@kth.se
Cristian Bogdan
cristi@kth.se

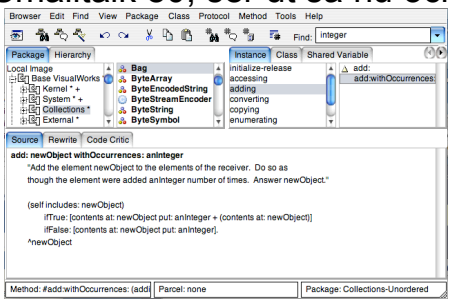
Objektorientering (OO)

- ”From the icy waters of Norway comes object oriented programming” *ur Eiffel PL av B Meyer*
- Första OO-begreppen i språket Simula (1967), skapat av Ole-Johan Dahl och Kristen Nygaard: klasser, subklasser, attribut (data och procedurer), objekt som instanser, referenser mm, syntaktiskt Algol60-utvidgning
- Simulas begreppsapparat lever vidare i Smalltalk, C++ (Simula med C), Objective-C (Smalltalk med C), Python, Ada 9X, Java, …

OO bra för WIMP-gränssnitt,

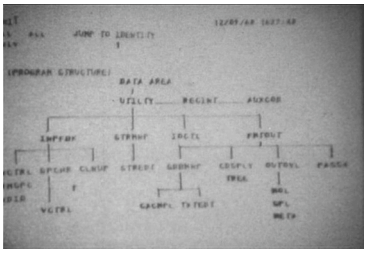
- byggda på Windows, Icons, Menus, Pointers.
- Detta insåg utvecklarna på Xerox Parc som 1972-80 skapade objektorienterade programmeringsmiljön Smalltalk för grafiska arbetsstationer (Alto 1972-): Allt är objekt, ”svagt” typat, browser där man kan se all kod i klasskategorier, återanvändning naturlig
- Trygve Renskaug (1978): MVC (Model-View-Controler)
- Visual Works (1985): Interface Builder
- Squeak (1995-): Vidareutvecklat Smalltalk
- Parallellt: Jean-Marie Hulots Interface Builder och programmeringsmiljö för NeXT machine (1985-90), finns nu till stora delar i Mac OSX-miljön.

Smalltalk 80, ser ut så nu också



Gränssnittsbyggare med metaforen filtyta (canvas), mera nästa gång

Lite WIMP-historik

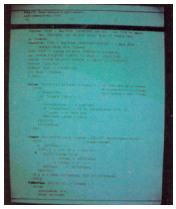


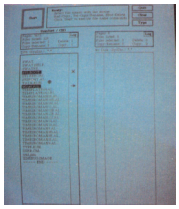
http://en.wikipedia.org/wiki/History_of_the_graphical_user_interface

Douglas Engelbart, 1968: The Augmentation Of Human Intellect Project.

Lite WIMP-historik

http://en.wikipedia.org/wiki/Xerox_Alto



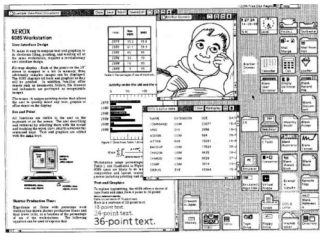


http://www.digibarn.com/collections/software/alto/index.html

Xerox Alto, 1972.

Lite WIMP-historik

http://www.thocp.net/
hardware/xerox_star.htm

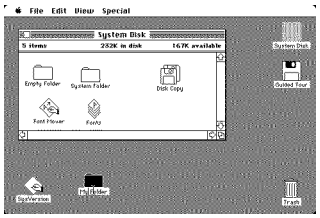


http://en.wikipedia.org/wiki/
History_of_the_graphical_user_interface

Xerox Star, 1981.

Lite WIMP-historik

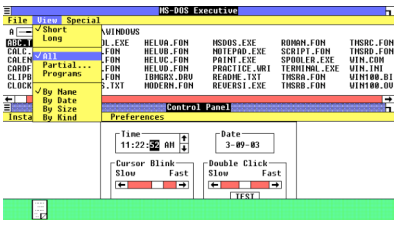
http://en.wikipedia.org/
wiki/Apple_Macintosh



http://en.wikipedia.org/wiki/
History_of_the_graphical_user_interface

Apple Macintosh, 1984:
Desktop-metafor (brett tillgänglig pga priset)

Lite WIMP-historik



http://www.guidebookgallery.org/screenshots/win101

Microsoft Windows 1.0, 1985:
Kooperativ multitasking.
Hoppa mellan DOS-program (ett aktivt åt gången).

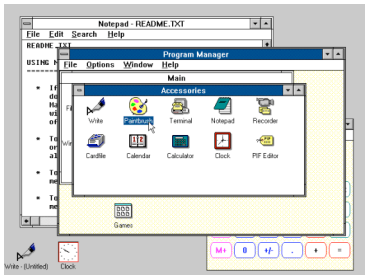
Lite WIMP-historik

Denna NeXT var Tim Berners-Lees och
världens först Web-server 1990

NeXT Step, programmeringsmiljö,
också för PC

NeXT-machine 1986-93:
Fantastisk programmeringsmiljö men dåliga prestanda

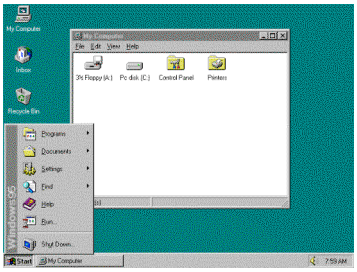
Lite WIMP-historik



http://www.guidebookgallery.org/screenshots/win30

Microsoft Windows 3.0, 1990:

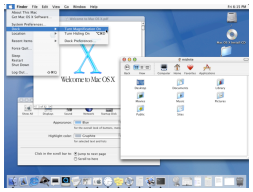
Lite WIMP-historik



http://en.wikipedia.org/wiki/History_of_Microsoft_Windows

Microsoft Windows 95, 1995:
"Gömde" MS-DOS (så att inte konkurrerande varianter kunde användas).

Lite WIMP-historik



Mac OSX 2001 (Cheetah) - 2007 (Leopard)
Med Mac och NeXT en Unix-utvecklingsmiljö mm

Lite WIMP-historik

http://archer.oregontel.com/brandon/vista/5219/3d.png

Microsoft Windows Vista, 2007:
Bygger på Windows XP. Första operativsystemet som kräver 3D-grafikkort.

Några andra WIMP-system

- **GEM Desktop** (PC, Atari, m.fl.), 1984.
- **X Window System** (Unix), mitten av 80-talet.
- **DESQview** (PC), 1985.
- **Workbench** (Commodore Amiga), 1985.
- **GEOS** (PC, Commodore 64/128), 1986.
- **OS/2** (PC), 1988.
- **MacOS 5-9**, 1988-1999.
- **BeOS** (BeBox, PC), 1991.

- Modern Linuxes: KDE, Gnome (X-based)

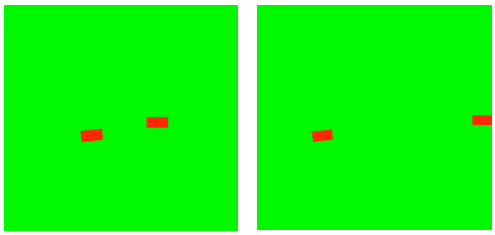
Fordonsspel

Vi ska göra ett enkelt spel med olika sorters fordon.

Vilka egenskaper har ett fordon?

- Position
- Orientering
- Hastighet
- Grafik / utseende

Första version av fordon (bilar)



Simulaversion av fordon (bilar)

```
BEGIN
  CLASS Point2D (x, y); REAL x,y; ;
  CLASS Vehicle (position, orientation, speed);
    REF (Point2D) position; REAL orientation, speed;
    BEGIN
      REF (Point2D) PROCEDURE setPosition(p); REF (Point2D) p;
        position :- p;
      PROCEDURE setOrientation(angle); REAL angle;
        orientaton := angle;
      PROCEDURE setSpeed(mps); REAL mps;
        speed := mps;
    END Vehicle;
```

Simulaversion av bilarna, forts

```
Vehicle CLASS Car;
  PROCEDURE draw;
  BEGIN Rita röd bil (rektangel) från punkten position ;
  END draw;
  PROCEDURE update(dt); REAL dt;
  BEGIN
    position.x := position.x+dt*speed*cos(orientation);
    position.y := position.y+dt*speed*sin(orientation);
  END update;
END Car;
REF (Car) myCar, myCar2;
REAL pi;
pi := 3.141593;
myCar :- NEW Car(NEW point2D(250,250),174*pi/180,20);
myCar2 :- NEW Car(NEW point2D(250,250),0,40);
WHILE TRUE DO BEGIN COMMENT Oändlig slinga ;
  myCar.update(0.01); myCar.draw(surf);
  myCar2.update(0.01); myCar2.draw(surf);
END slinga;
END program
```

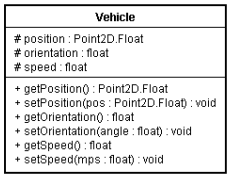
Fordonsspelet i Java (demas i Eclipse, källkod på kurswebben)

```
class Vehicle {
  protected Point2D.Float position;
  protected float orientation;
  protected float speed;

  public Point2D.Float getPosition();
  public void setPosition(Point2D.float p);
  public float getOrientation();
  public void setOrientation(float angle);
  public float getSpeed();
  public void setSpeed(float mps);
}
```

Dessutom importeras: `import java.awt.geom.*;import javax.swing.*;`
Och finns källkod till `public class CarGame`
`public class Car extends Vehicle`
`public class GameSurface extends JPanel`
`public class AnimationSystem implements Runnable`

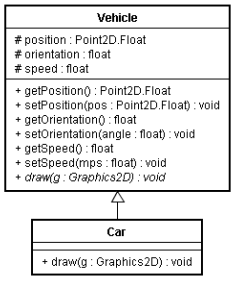
Fordon



- Klassnamn
- Data
- Metoder

- # = protected
- + = public
- - = private

Fordonsspel



- Varje typ av fordon vet hur det ska rita ut sig självt
- Så vi låter **Vehicle** bli en abstrakt klass med en metod **draw**, som alla fordon måste implementera
- The actual draw method of a Vehicle object is found at runtime (polymorphism)

Objektorientering i allmänhet

- Ett objekt är en instans av en klass, som beskriver "lådor", där man samlar
  - **Data**
  - **Metoder** som (oftast) manipulerar dessa data
- En princip som stöds av objektorientering är att **data blir gömt** för "utomstående" klasser
  - encapsulation
- Undantaget är subklasser till basklassen, dessa kan eventuellt få accessa data direkt
- Man kan allstå definiera en "hierarki" av klasser och subklasser.

Objektorientering i allmänhet

```
public class BaseClass {
  private int aNumber;
  protected int anotherNumber;
  public int add(int n) {
    aNumber += n;
    return aNumber;
  }
}

public class SubClass extends BaseClass {
  public void subtract(int n) {
    anotherNumber -= n;
  }
}
```

SubClass kan inte komma åt **aNumber** eftersom den är satt som **private**.

Objektorientering i allmänhet

```
BaseClass b = new BaseClass();
SubClass s = new SubClass();
int n;

n = b.add(10);
n = s.add(10);
s.subtract(3);
```

- Subklassen **ärver** alla basklassens metoder och data.
  - inheritance
- En **instans** av subklassen innehåller alltså alla data och alla metoder i basklassen plus eventuella data och metoder i subklassen.

Överlagring

- Det är helt OK att skriva olika versioner av samma metod.
- Det som inte får skilja mellan dem är typen på det som returneras.
- Detta brukar kallas **överlagring** (overloading).

```
public class ClassA {
  public void f(int n) {
    ...
  }
  public void f(float f) {
    ...
  }
  public void f(String s) {
    ...
  }
}

ClassA a = new ClassA();
a.f(10);
a.f(3.4f);
a.f("Hello");
```

Överlagring

- En subklass kan ändra beteendet hos en metod.
- Det är också en form av överlagring (overriding, see polymorphism).

```
public class ClassA {
  public int f() {
    return 10;
  }
}

public class ClassB extends ClassA {
  public int f() {
    return 15;
  }
}

ClassA a = new ClassA();
ClassB b = new ClassB();

int n = a.f();   // 10
int m = b.f();   // 15
```

Konstruktorer och destruktorer

- Klassens **konstruktor** anropas när en instans skapas.
- En default-konstruktor skapas alltid som gör ingenting.
- I t.ex. C++ finns också en **destruktor** som anropas innan instansen tas bort ur minnet.

```
public class MyClass {
  private int n;
  public MyClass(int number) {
    n = number;
  }
  public int getNumber() {
    return n;
  }
}

MyClass c = new MyClass(10);
int n = c.getNumber();
```

Gömda klassmedlemmar

- Man använder ordet **super** för att hänvisa till överordnade klassens uppsättning metoder och variabler.
- Det är det enda sättet att komma åt metoder eller variabler som **gömts** av subklassen.

```
public class Base {
  public boolean aVariable;
  public void aMethod() {
    aVariable = true;
  }
}

public class Sub extends Base {
  public boolean aVariable;
  public void aMethod() {
    aVariable = false;
    super.aMethod();
    aVariable = 10;
    super.aVariable = 15;
  }
}
```

Abstrakta klasser

```
interface MyInterface {
  void add(int n);
}

public class MyClass implements MyInterface {
  void add(int n) {
    ...
  }
}
```

Ibland vill man "tvinga" att en klass implementerar vissa metoder. Det bästa sättet att åstadkomma det är med en s.k. **abstrakt klass**,d.v.s. en klass som talar om vilka metoder som ska finnas i subklassen men som inte bryr sig om hur de är implementerade.

I Java kallas abstrakta klasser för **interfaces**.

Abstract classes and Polymorphism

- An instance of MyClass will "be" both a MyClass and a MyInterface
  - Some methods may require a MyClass parameter, some a MyInterface parameter, instances of MyClass can be passed to both
- In Java A class can inherit (extend) one class, and implement any number of interfaces
- Therefore a Java object can be of many types at the same time
  - The declared type (MyClass)
  - The inherited type (if any) and the types it inherits
  - The implemented abstract classes (interfaces)
  - The abstract classes implemented by the inherited type…

Object Orientation summary

- Encapsulation: Keep specific data and code only for object internal use, do not expose it to the outside world
- Inheritance: Derive a class from a conceptually more general one
- Polymorphism: An object is of multiple types, and can (re)define behavior declared by the respective types
- All improve code *reusability* and *reliability* but…

OO: No silver bullet

- Fred Brooks (1986)
- It is difficult to get a OO design (classes and their relations) right from the beginning
- *Refactoring* is standard practice by now, re-structure the code without changing overall behavior as we learn more
- In fact, any domain can be modeled in a large number of ways, all correct, depending on our priorities, concerns (see frameworks later)
- Class is maybe not the suitable abstraction
- Pattern: a re-occuring problem and an OO design that addresses it

Type casting

- Ibland måste man **omtolka** en variabel (type casting).
- Vissa omtolkningar är inte tillåtna - i så fall varnar kompilatorn.
- Men ibland kan den inte avgöra om man gjort rätt, så var försiktig.
- *Downcasting* from supertype to a subtype
- Top types: Object, always require downcasting
- A solution: generics

```
public class Base {
  ...
}
public class Sub1 extends Base {
  ...
}
public class Sub2 extends Base {
  ...
}

int a = 10;
float b = (float)a;
Base[] c = new Base[10];
c[0] = new Sub1();
c[1] = new Sub2();
if (c[0] instanceOf Sub1) {
  Sub1 s = (Sub1)c[0];
}
Sub2 t = (Sub2)c[0]; // VARNING!
```

Meddelanden

- I java-(och Smalltalk-)sammanhang brukar man säga att klasser/instanser kommunicerar via **meddelanden**.
- Man skickar ett meddelande genom att helt enkelt anropa en metod i en annan klass.

```
public class ClassA {
  public void f(int n) {
    ...
  }
}

public class ClassB {
  private ClassA a;
  public void setA(ClassA a) {
    this.a = a;
  }
  public void sendMsg(int n) {
    a.f(n);
  }
}

ClassB b = new ClassB();
ClassA a = new ClassA();
b.setA(a);
b.sendMsg(10);
```

Inre klasser

- Man kan skapa klasser inuti en klass.
- Dessa kan bara instansieras inuti den omslutande klassen.
- En inre klass har access till alla variabler och metoder i den omslutande klassen.
- **Anonymous** inner classes, redefine methods on the spot

```
public class MyClass {
  private int n;

  public class MyInnerClass {
    public int f() {
      n = 10;
    }
  }

  MyInnerClass c;

  public void g() {
    c.f();
  }
}

MyClass x= new MyClass(){
   public void g(){ … }
};
```

"Final"

- En klass som är deklarerad som **final** kan man inte göra subklasser till.
- En metod som är deklarerad som **final** kan inte överlagras.
- En variabel som är deklarerad som **final** kan aldrig ändra värde.

```
final class MyClass {
  ...
}

public class MyClass2 {
  final int f() {
    ...
  }
}

final int c = 10;
```

Instans- och klassvariabler

- Det finns **instansvariabler** och **klassvariabler**.
- Instansvariabler har en egen "kopia" i varje instans.
- Klassvariablerna finns det bara en av.
- Man använder ordet **static** för att ange att en variabel eller metod är en klassvariabel.

```
public class MyClass {
  public int n;
  public static int m;
  public int f() {
  }
  static public int g() {
  }
}

MyClass c = new MyClass();
int n = c.f();
int m = c.g();
int p = MyClass.g();
int q = MyClass.m;
```

Garbage collection

- I Java behöver man inte säga till när objekt ska frigöras.
- Istället finns en s.k. **garbage collector** som rensar minne för objekt som saknar referenser.
- Man kan "släppa" en referens explicit genom att sätta den till **null**.

```
public class MyClass {
  ...
}

MyClass c = new MyClass();
c = null;
```

Arrays (vektorer och matriser)

```
int[] num = new int[10];
for (int j = 0; j < 10; j++) {
  num[j] = j;
}

int[] num2 = { 0, 1, 2 };

System.out.println(num.size);  // Ger 10
System.out.println(num2.size); // Ger 3

int[][] twoDim = new int[2][3];
twoDim[0][0] = 0;

int[][] twoDimDyn = new int[2][];
twoDimDyn[0] = new int[10];
twoDimDyn[1] = new int[3];
```

Strängar

```
char c1 = 'a';

String s1 = new String("Hello");
char c2 = s1.charAt(0); // Ger 'H'

char[] c3 = { 'H', 'e', 'l', 'l', 'o' };
String s2 = new String(c3);
System.out.println(s2.length()); // Ger 5

String s3 = s1 + s2;
System.out.println(s3); // Ger "HelloHello"
```

Paket

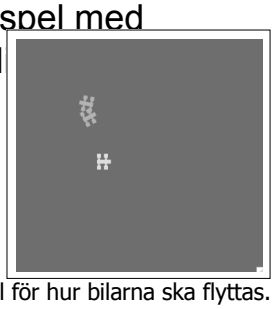
- I Java kan man låta klasser tillhöra ett s.k. **package**.
- Det gör att det är enklare att skilja på klasser som råkar ha samma namn.
- Ta för vana att lägga dina klasser i ett eget package!
- Package access is default: classes have access to each other's members unless they are declared private

```
package myPackage;

import java.swing.*;

public class Base {
  ...
}
```

Utökat bilspel med styrmöjl



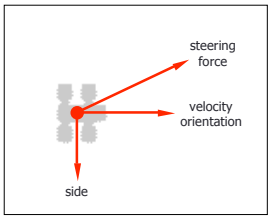
Vi utökar vårt spel.

Vi behöver en bättre modell för hur bilarna ska flyttas.
Vi vill kunna styra (åtminstone) en av bilarna.

Se vidare fullständiga programmet på kurswebplatsen.

Ny Vehicle-klass

Vehicle
position : Point2D.Float
orientation : Point2D.Float
side : Point2D.Float
velocity : Point2D.Float
steering : Point2D.Float
mass : float
maxSpeed : float
maxSteering : float
behaviours : ArrayList

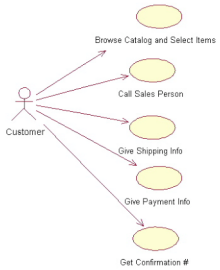


UML

- Både en mjukvarudesignprocess och en serie diagramtyper.
- Utvecklades av företaget Rational i mitten av 90-talet.
- Är idag standardiserat och används överallt i industrin.
- Ger dock mycket begränsad info om hur användargränssnittet ska se ut!

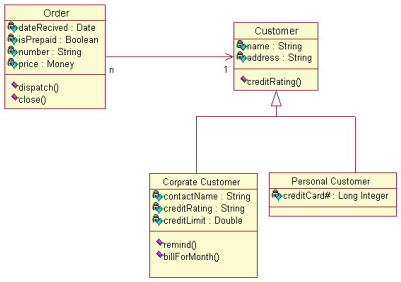
UML: Use case diagrams

- En abstraktion av en eller flera processer, d.v.s. ett **scenario**.
- Visar **aktörer** och **use cases**, d.v.s. vad aktörerna gör.
- One or more scenarios may be generated from a use case



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

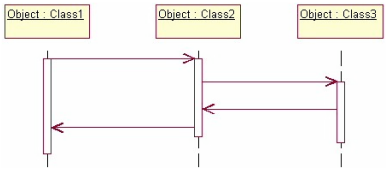
UML: Klassdiagram



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Interaktionsdiagram

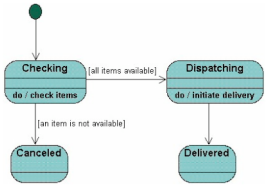
- Visar hur meddelanden skickas mellan klassinstanser i systemet.
- Ger en översikt över processer i systemet.



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Tillståndsdiagram

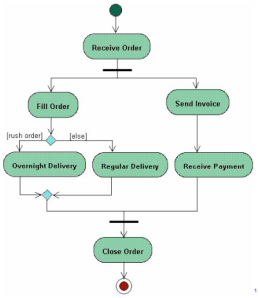
- Visar hur systemet går mellan olika tillstånd.



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Aktivitetsdiagram

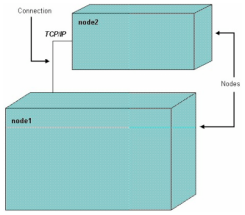
- Liknar tillståndsdiagrammet men fokuserar på aktivitet istället.
- En aktivitet behöver inte vara detsamma som ett tillstånd.
- Kan motsvara funktioner som användaren aktiverar.



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

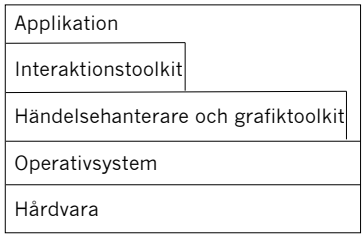
UML: Fysikdiagram

- Används för att visa relationen mellan hård- och mjukvara i systemet.
- Kan också användas för att visa hur systemet är distribuerat över olika maskiner, t.ex. i ett nätverk.



http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

Fönstersystem

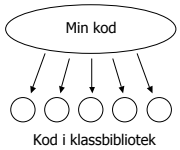


Frameworks

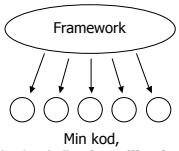
- Erfarenheten visar att det kan vara knepigt att hantera toolkits.
- Om allt är "tillåtet" är det lätt att applikationer blir inkonsekventa eller ser "fula" ut.
- Ett **framework** är en uppsättning klasser (abstrakta och vanliga) som "påtvingar" ett visst sätt att arbeta.
- Javas framework för fönsterhantering heter Swing. Swing vs IBM/Eclipse SWT
- Microsofts motsvarighet heter MFC (som nu är på väg att ersättas av .NET). MFC vs Borland OWL

Frameworks

Koda med klassbibliotek



Koda med framework



Inversion of control (see Spring framework)
Hollywood principle (we'll call you)
Template method design pattern

Frameworks

•Ett vanligt sätt att arbeta med frameworks är att man subklassar någon form av basklass med grundfunktionalitet.
•Sedan överlagrar man de metoder som man intresserar sig för.
•It is often suitable that the subclass is anonymous

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
  public void paint(Graphics g) {
    g.drawString("Hello", 10, 50);
  }

  public static void main(String[] args) {
    MyFrame f = new MyFrame();
  }
}
```