# Referenser, Metoder med Arrayer och Grafik

Vecka 2, Bildserie 2

# Innehåll

- switch-satsen
- Referenstyper och referensvariabler
- Metoder med arrayer
- Explicit typomvandling
- Lite om Java-plattformen
- Datorgrafik
- Java 2D

# Att Läsa i Boken

- 3.13
- 9.5.4
- 7.5-7.7
- 2.15

OBS! Att jag använder lite annorlunda stil

- Inledningsvis behövs aldrig ordet static eller public, bortse från dessa.

# switch-satsen

```
switch (k) {            // Compare k with ...
   case 10:            // ... 10. If equal ...
       out.println("k is exactly 10");  // .. do this.
       break;  // Must have "break" at all "case"
   case 20:        // ...20. And so on.
       out.println("k is exactly 20");
       break;
   case 30:
       out.println("k is exactly 30");
       break;
   default:   // If no match do this
       out.println("k is ? (no match)");
}
```

Switch-satsen

- Är också en selektion men den jämför bara på likhet
- Det som skall  jämförs skall vara av heltals eller String-typ (förenklat), skrivs inom parentesen.
- Varje case-gren jämförs i tur och ordning mot värdet i parentesen
    - Grenen skall innehålla ett konstant värde för jämförelse, variabler går inte.
    - Om likhet så körs satserna vid grenen (bara en case-gren körs)
    - Om ingen case-gren sann körs default-grenen
- OBS! Måste ha break efter varje case-gren, annar körs flera case-grenar.
- Break fungerar på samma sätt som för while men här hoppar vi ur switch-satsen.

Switch kan skrivas något kompaktare än if då man bara vill undersöka om något matchar ett antal alternativ…

- … det är därför den används
- Logisk går det lika bra med en if else if-satsen
- Används aldrig då flera värden skall ingå i jämförelsen (t.ex. if( a == 1 || b == 2))

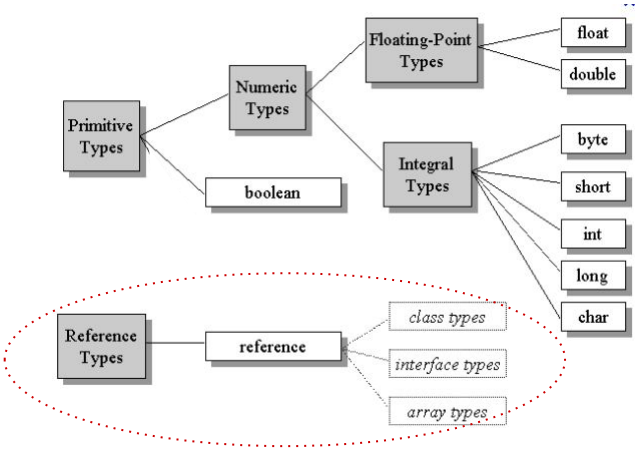
# Programmering

```
Enter command (1,2 or 3, 0 to quit) > 3
3 selected
Enter command (1,2 or 3, 0 to quit) > 1
1 selected
Enter command (1,2 or 3, 0 to quit) > 2
2 selected
Enter command (1,2 or 3, 0 to quit) > 5
?
Enter command (1,2 or 3, 0 to quit) > 1
1 selected
Enter command (1,2 or 3, 0 to quit) > 0
Bye bye
```

Hur skriver man programmet (en kommandorad)?

- TIPS: Använd while och switch

# Referenstyper

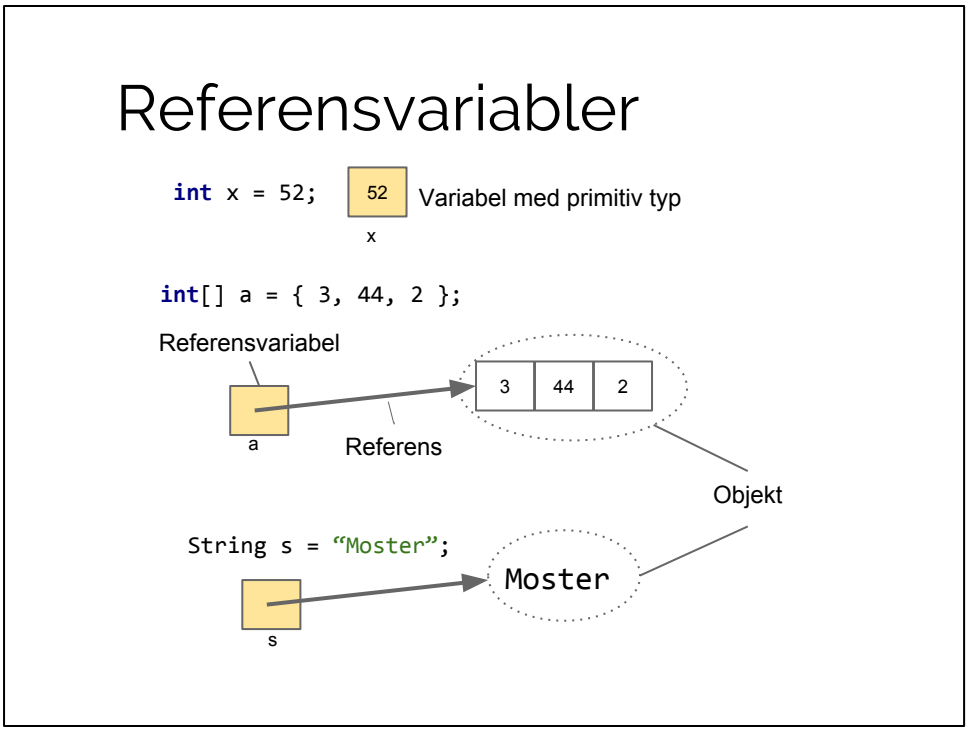



Har hittills bara talat om primitiva typer

- En annan familj av typer är referenstyper
- Mer om typer här.

**Referenstyper** kan vara arraytyper (t.ex. int[]), klassertyper (t.ex. String) och gränssnitt (interface)

- Man kan skapa nya referenstyper!
- Typsystemet är utbyggbart för array-, klass- och gränssnittstyper
  - Kallas också **egendefinierade typer** (vi definierar dem)
- Eftersom vi nu har ett antal nya typer kan vi deklarera variabler för dessa ...




Variabler av primitiv typ (**primitiva variabler**) innehåller värdet, värdet finns i variabeln (t.ex. värdet 52 en int)

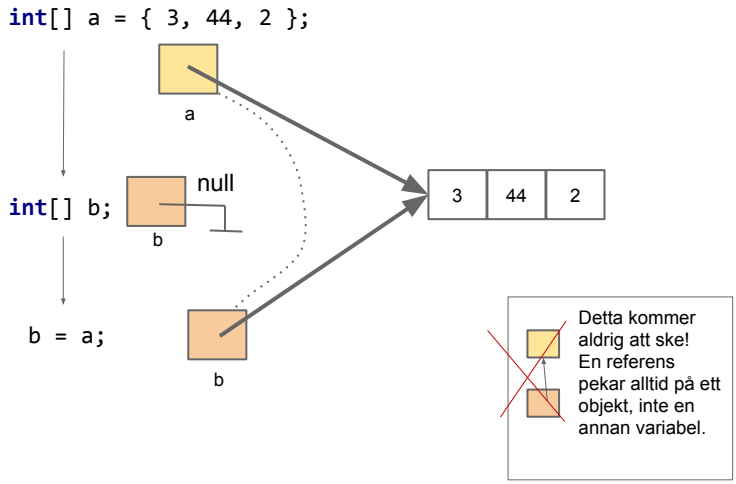
Variabler deklarerade med referenstyper innehåller inte värdet …

- … de innehåller en **referens** till ett namnlöst objekt som innehåller värdet/värdena
  - Vi säger också att referensen "pekare" på ett objekt.
  - Storleken för en referens är 64 bitar (om JVM:en är byggd för 64 bitar, har du en gammal dator kan det vara 32 bitar)
    - Vi har inga problem med olika minnesbehov för referenser, alla har samma.
- Variabler med referenstyper, kallas för **referensvariabler**
  - Enda sättet att komma åt det namnlösa objektet är via referensenvariabeln
  - Tappar vi referensen (värdet) i referensvariabeln är objektet oåtkomligt.

En mer teknisk förklaring är att en referens är en minnesadress.

- En referensvariabel innehåller en adress

# Referensvariabler och Tilldelning



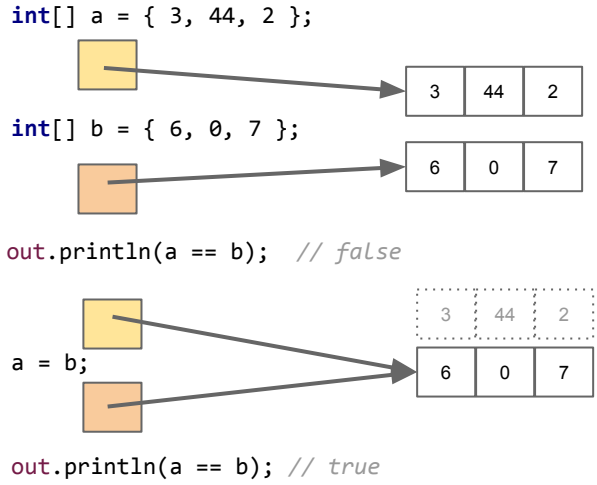

Tilldelning för referensvariabler fungerar som för primitiva typer

- Värde kopieras från vänster till höger MEN …
- .. "värdet" är en referens, det är referensen som kopieras!
- Effekten blir att två referenser pekar på samma objekt!

En referensvariabel som inte har initierats innehåller värdet null

- Värdet **null** (enda värdet i referenstypen Null) innebär att en referensvariabel inte refererar något
- Ett undantag uppstår om man försöker gör något med en null-referens, **NullPointerException, NPE**
  - Ett ständigt och stort problem är att hålla reda på om referenser är null.
  - Att tilldela en variabel null och skriva ut ett null-värde går bra (texten null visas).

# Referensvariabler och Likhet




Likhet för referensvariabler fungerar som för primitiva variabler

- Innehållet i variablerna jämförs MEN …
- … innehåller är nu referenser!
- Likhet innebär att referenserna refererar samma sak (pekar på samma objekt)
    - Innehåller samma adress!

Efter tilldelningen, a = b ovan, pekar referenserna på samma objekt.

- De ursprungliga objektet a pekade på saknar nu referens
    - Vi kan aldrig komma åt objektet mer
    - Objektet kommer automatiskt att **skräpsamlas (garbage collect),** tas bort ur minnet

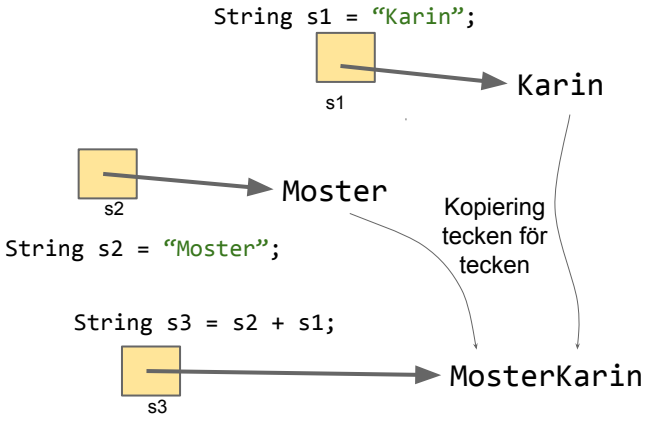
Att tilldelning och likhet för referenser får en speciell betydelse sammanfattas som **referenssematik**

- Semantiken (betydelsen) blir annorlunda pga av vi använder referenser
- För primitiva typer säger man **värdesematik**

OBS! String är en referenstyp, innebär att == inte "fungerar" för strängar!

- Normalt är de ju själva texten vi vill jämföra men == jämför referenserna.

# Strängar och +-operatorn




Konkatenering med +-operatorn innebär att ett nytt strängobjekt skapas och en referens till detta returneras.

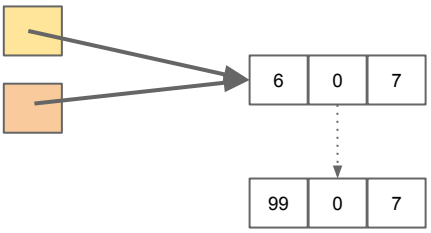
- Tecknen från operanderna kopieras till det nya objektet.
- +-operatorn kan vara ineffektiv t.ex. i en loop med många varv (kopierar mer och mer för varje varv)

Om flera strängvariabler pekar på exakt samma strängliteral kommer den att sparas på en egen plats i minnet som alla referenser (i sin tur) pekar på

- D.v.s. en strängliteral finns bara i en enda upplaga i programmet, mer senare ...

# Aliasproblem

```
a = b;
```




```
a[0] = 99;
```

```
if( b[0] == 99 ){ // b changed!!!
    // true
}
```

Att flera referenser kan peka på samma objekt innebär att det finns flera sätt att ändra objektets värde

- Kallas **alias**(problem) den ena referensen är ett alias för den andra
- Kan leda till svårlösta fel i program (men går inte att undvika i imperativa språk med referenser)

# Varför Referenser?

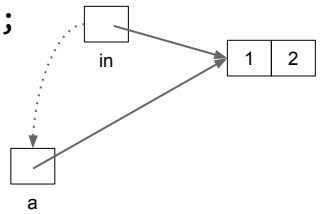

Referenser kan leda till många problem, varför finns de …?

Vi har sett att vid tilldelning och metodanrop/återhopp sker det en kopiering

- Datorer kan arbeta med många sorters data t.ex. bilder, bilder kan vara på flera MB
- Antag att bilder skulle sparas i variabler, då skulle flera MB kopieras vid varje tilldelning och metodanrop
- Att istället ha en referens till bilder är väldigt mycket effektivare, det är referensen som kopieras och skickas runt!
- I bilden låter vi två referenser peka på samma bild

# Metoder med Array-argument

```
void program() {
   int[] in = { 1,2 };
   zero(in);
   // in = { 0, 0 }
}

void zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
      a[i] = 0;
   }
}
```




Om en metod har array-parametrar sker precis samma sak som vanligt men …

- … den aktuella parametern som kopieras är en referens!
- Den formella kommer därför att peka på samma objekt som den aktuella!
- Metoden kan ändra på ett objekt utanför sig själv (referensen "in" kan vi däremot aldrig ändra)

Summa:

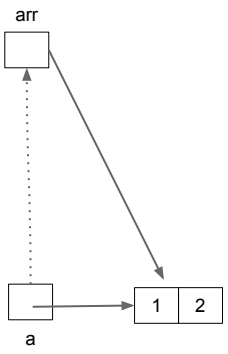
- Samma variabelnamn inom olika synlighetsområden syftar på på olika variabler
- Olika variabelnamn kan syfta på samma sak, om referenser är inblandade (alias igen!)

I koden ovan kommer array:en "in" att vara förändrad efter metodanropet

- Kallas **utparameter**
- Inget returvärde behövs om vi vill åstadkomma detta, metoden kan vara void

# Array som Returvärde 1

```
void program() {
   int[] arr = getArray();

}



int[] getArray(){
    int[] a = { 1, 2 };
    return a;
}
```




Om man returnerar en array sker samma sak som vid ett vanligt metodanrop

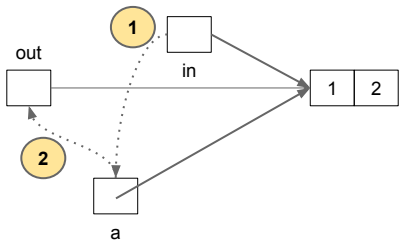
- Returvärdet kopieras till en tillfällig lagringplats
- I detta fall kopieras en referens

Vi kan skapa en array i en metod och skicka som returvärde.

- Den lokala variabeln a försvinner! …
- Men inte arrayen (om vi returnerar en referens som sparas i någon annan variabel).

# Array som Returvärde 2

```
void program() {
   int[] in = { 1,2 };
   int[] out = zero(in);

}

int[] zero( int[] a ){
   for(int i = 0; i < a.length ; i++){
        a[i] = 0;
    }
    return a;
}
```




Här returnerar vi samma referens som vi skickar in.

# Programmering

```
int[] arr = {2, 5, 0, -1, 5, 6};

out.println( findMin(arr) );    // 3
out.println( findIndex(arr, 5) );  // 1
```

Hur skriver man metoderna?

- TIPS: for, array och if

# Programmering

```
int[] arr = {2, 5, 0, -1, 5, 6};

incArr(arr);
incArr(arr);
String s = Arrays.toString(arr);
out.println(s);  // [4, 7, 2, 1, 7, 8]

s = Arrays.toString(incArr(incArr(arr)));
out.println(s);  //  [6, 9, 4, 3, 9, 10]
```

Hur skriver man metoden incArr?

- Metoden inkrementerar alla värden i en array.

# Överlagrade Metoder

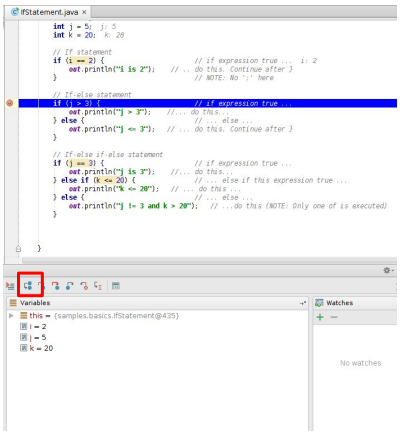
```
// Overloaded (from Math API)
double max( double d1, double d2){...}
float max( float d1, float d2){...}
int max( int d1, int d2){...}
long max( long d1, long d2){...}
```

Metoder (inte variabler) inom samma synlighetsområde får ha samma namn …

- … men då måste parameterlistorna skilja sig åt!
  - Returtypen spelar ingen roll, bara parametrarna räknas.
- Kallas att metoderna är **överlagrade** (overloaded)
  - Innebär att redan vid kompileringen väljs (utifrån bästa matchning) vilken metod som skall anropas vi körningen.
  - Vilken metoden som väljs beror alltså på parametrarna.
  - Jämför +-operatorn som beror på operanderna!
- Tanken med överlagrade metoder är att man skall slippa hitta på nya namn på metoder som gör "samma sak" men med olika antal eller parametertyper.
  - Metoder som gör olika saker skall inte ges samma namn

# Felsökning

Avlusare (debugger)




Utskrift av värde

```
out.print(someValue);
```

En stor del av programmeringsarbetet består av felsökning. Det finns flera alternativ

- Används utströmmen och skriv ut värden
  - Kan var en bra metod för upprepad  inspektion
    - Utskriften sker varje gång programmet körs
  - Nackdelar:
    - Blir det för många utskrifter kan man till slut inte hänga med
    - Utskrifterna måste tas bort
- Använd en avlusare (Debugger)
  - En avlusare är ett program som kan köra ett annat (ditt) program sats för sats
  - Finns inbyggd i IntelliJ

Avlusning (procedur)

- Klicka i vänstermarginalen för att få en brytpunkt (röd prick ovan).
  - Klicka igen om du vill ta bort..
- Högerklicka i kodfönstret och välj  Debug …
- Avlusaren startar och kör programmet fram till brytpunkten. Där stannar det.
- Därefter kan du köra sats för sats genom att klicka Step Over (röd fyrkant i bilden)

- Den blå raden skall hoppa en sats då du klickar
- Du kan hela tiden inspektera variabelvärden i det nedre fönstret
- Avsluta genom att klicka röd fyrkant ner till vänster (visas ej)
- Hakar något upp sig … börja om

# Explicit Typomvandling

## Primitiv typ

```
int width = 200;
double scale = width / 2 / PI;      //Built-in PI (double)
int x = width / 2 + (int) (scale * i);

int a = ...;
int b = ...;
double d = (double) a / b;      //Fix real division
```

## Referenstyp

```
Graphics g = ...
Graphics2D g2 = (Graphics2D) g;
```

Ibland måste vi uttryckligen säga åt typsystemet att vi vill behandla ett värde av en typ som en annan typ

- Kallas **explicit typomvandling (type casting, cast)**
- Parenteser kan behövas för att visa vad (vilket uttryck) som berörs

Omvandling primitiva typer

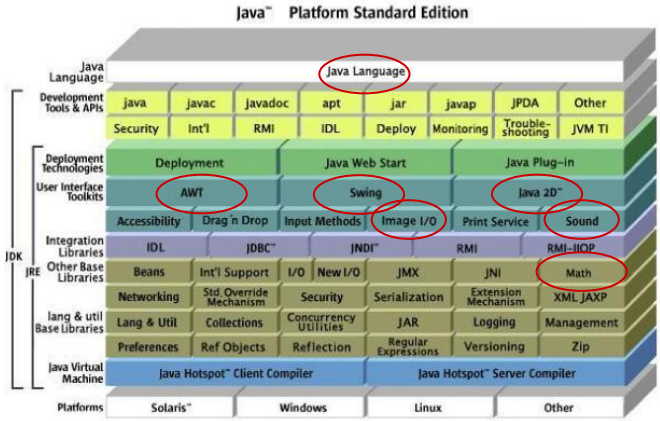
- Om vi vill omvandla ett double-värde till ett int-värde skriver vi (int) framför
  - Innebär att vi kapar alla decimaler (ingen avrundning)
  - Vi tappar information
  - Typsystemet satt ur spel vi får själva ansvara för följderna
  - Undvik om möjligt!
- Om vi vill omvandla en int till en double skriver vi (double) framför
  - Ingen information försvinner
- Vid denna typ av omvandling förändras minnet, en int och en double tar ju olika stor plats i minnet.

Omvandling av referenstyper

- Kan typomvandla en referenstyp till annan på samma sätt (dock finns väldigt många regler hur detta får gå till, kommer i fortsättningskursen)
- Vid denna typ av omvandling sker inget med minnet, alla referenser

- är 64bitar (eller 32 bitar)
    - Alltså samma adress ligger kvar i variabeln.
    - Omvandlingen är enbart till för typsystemet
        - Efter omvandlingen godkänner typsystemet bara operationerna för den nya typen
        - … gör vi inte omvandlingen i bilden kan vi inte använda metoderna i typen Graphics2D (bilden) bara de som finna i typen Graphics.

# Java-Plattformen




Java är en **plattform**

- Dels finns själva språket Java som är en ganska liten del
- Utöver detta finns en mängd **standarbiblioteket**
- Java-plattformen = språket + standardbiblioteken + andra (tredjeparts) bibliotek + utvecklings- och exekveringsmiljön
- Standarbiblioteket är organiserat i olika grupper utefter ändamål
    - Grupperna kallas för bibliotek eller API:n (**Application Programming Interface, programmeringsgränssnitt**)
- Vi har tidigare tittat på Math API:et
- Vi skall nu titta lite på ett API som heter Java2D. API:et gör det möjlig för våra program att på ett enkelt sätt rita 2D grafik.
    - Exakt hur Java2D fungerar kan inte fördjupa oss i
    - Vi får använda funktionell abstraktion och helt enkelt acceptera att om vi skriver vissa satser så utförs vissa saker.
    - För att använda Java2D API:et krävs två andra API:n: AWT (Abstract Windows Toolkit ) och Swing (varför går vi inte in på)
        - Vi måste i programmet ange
            - import javax.swing.*;
            - import java.awt.*;
    - I verkligheten används inte Java2D för spel (detta är en undervisningssituation)

# Datorgrafik


Datorgrafik

- Används för att rita bilder på en datorskärm
- Varje enskild bildpunkt (pixel) måste kunna adresseras (komma åt/ändra)
- Grafik innebär att (delar av) datorskärmen ritas om med en viss frekvens
- Man brukar skilja på 2D- och 3D-grafik, bilden visar en variant av 2D-grafik.

# Kollisioner

Finns väldigt mycket deklarerat (färdigt)  i Java bibliotek

- Kan leda till kollisioner mellan vår kod och bibliotekskod
    - Om vi skriver något (ev. av misstag) i vår kod som vi inte själva har deklarerat men som finns i något bibliotek vi använder så accepteras detta  (och används under körningen)
        - T.ex. finns HEIGHT deklarerat i Java2D …
        - … skriver vi detta i stället för height (som vi själva deklarerat) blir det fel!
- Som sagt: Ibland är IntelliJ överdrivet hjälpsam och lägger till saker vi inte vill ha (t.ex. vid import …)
    - Verkar programmet bete sig väldigt konstigt, kontakta handledare… kan handla om en kollision …
    - Använd Ctrl z !!!

# Mall för ett Grafikprogram

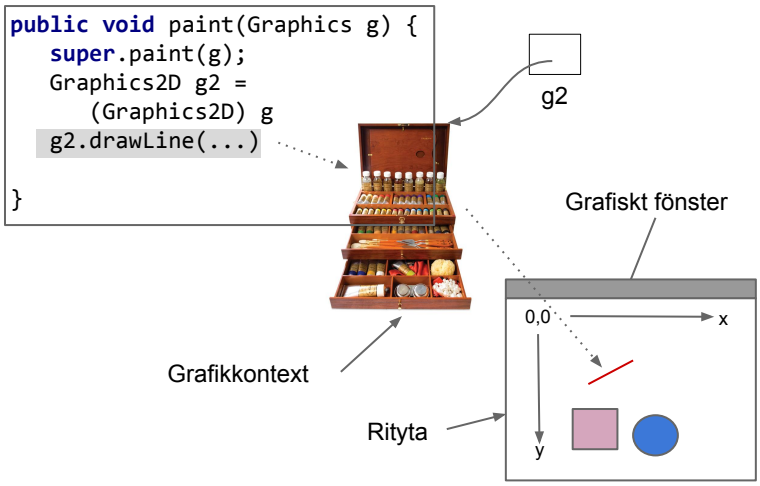
```
public class Graphics extends JPanel {
        void program() {
                initGraphics();
        }

        public void paint(java.awt.Graphics g) {
                super.paint(g);
                Graphics2D g2 = (Graphics2D) g
                // Do the drawing ….
        }

        void initGraphics() {
                setPreferredSize(new Dimension(400, 400));
                JFrame window = new JFrame();
                window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
                window.add(this);
                window.pack();
                window.setLocationRelativeTo(null);
                window.setVisible(true);
}
```

Analys av koden

- Vi måste lägga till extends JPanel, innebär att vårt program kan fungera som en rityta (som kan visas i ett fönster på skärmen)
- I program()
  - Anropas bara initieringen för grafiken (som finns i en egen metod, initGraphics())
- Metoden paint
  - Se nästa bild
- Metoden initGraphics (rad för rad)
  1. Sätter önskad storlek på ritytan
  2. Skapar ett fönster
  3. Säger att fönstret skall stängas och programmet avslutas då man klickar X i ramen
  4. Lägger till ritytan till fönstret
  5. Gör så att fönstret får samma storlek som ritytan
  6. Sättter fönstrets övre-vänstra hörn mitt på skärmen
  7. … och slutligen ritar fönstret på datorskärmen
     - Därefter sker ritningen i paint-metoden

Beskrivningen ovan är bara till för att stilla er nyfikenhet, inget att lära sig utantill (kommer inte på tentan)

# Paint-metoden




I Java2D APi.et finns två "grafikkontexter" (målarlådor) med typen Graphics respektive Graphics2D

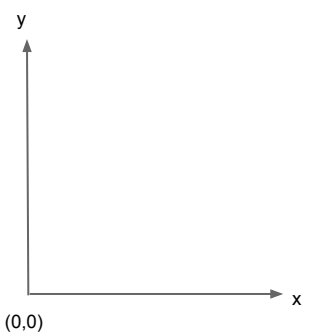
När ett grafiskt Java-program märker att något "behöver ritas om" t.ex. om storleken på ett fönster ändras, anropas automatiskt en dold metod "paint" (syns inte i koden)

- Metoden får dessutom automatiskt som parameter en referens till ett grafikkontext-objekt (av typen Graphics)
- Grafikkontexten är kopplad till en rityta (i ett fönster) och vet alltså var den skall rita
- Utritning kallas **rendering**

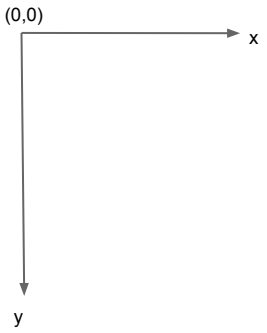
Om vi vill modifiera den förvalda renderingen kan vi lägga till en egen paint-method

- Då anropas vår metod istället för den dolda!
  - OBS! Måste ha public framför, mer senare ...
- Först i vår metod säger vi åt systemet att göra den förvalda renderingen (super.paint(g))
- Därefter gör vi en explicit typomvandling till den andra kontexttypen, Graphics2D
  - Vi använder därefter denna kontext för att rita genom att anropa **grafikprimitiver,** t.ex. g2.drawLine(...), mer strax ...

# Logiskt och Grafiskt universum




Logiskt 2D universum




Grafiskt (skärm) 2D universum

Våra program arbetar alltid i logiskt universum

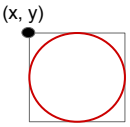
- I detta är koordinater angivna som "vanligt" med origo i nedre vänstra hörnet.

Utritning av objekt i vårt universum sker i ett grafiskt 2D universum (**skärmkoordinater**)

- (0,0) i övre vänstra hörnet
- Innebär att vi måste göra en omvandling från logiska koordinater till grafiska
  - Detta görs alltid "precis innan utritningen" (i paint-metoden eller ännu bättre, i en fristående metod som anropas från paint)
  - Dessutom måste vi ofta skala och translatera för att bilden skall hamna/se ut som vi vill

# Grafikprimitiver

```
Graphics2D g2 = ...

g2.drawLine(20, 20, 300, 300);
g2.drawOval(100, 100, 40, 70);
g2.drawString("Hej på dej", 150, 200);
g2.fillOval(300, 200, 100, 50);
g2.fillRect(100, 100, 0, 50);    // Hmm, problems!
g2.drawRect(300, 300, 100, 100 );
// Note, no drawPoint (use drawLine)
```




Allt vi behöver för att rita finns i Graphics2D objektet

- Genom att skriva g2**.** (punkt) och ett metodanrop kommer Java2D att se till att något ritas på ritytan
- Metoderna vi anropar kallas **grafikprimitiver** (finns liknande i många språk)
  - Kan bara rita linjer, rektanglar, cirklar o.s.v.
  - Alla primitiver tar heltalsparametrar
    - Arbetar vi med double får vi explicit typomvandla
  - För figurer utgår grafikprimitiverna från övre vänstra hörnet (x,y)
    - Vi får göra justeringar eftersom vi logiskt ofta utgår från något annat
    - T.ex mittpunkten i en cirkel (mittpunkten rekommenderas för allt, som har dylik)

# Färger

```
import static java.awt.Color.*; // Must have

g2.setColor(BLUE);      // Set color, will remain until ...
...
g2.setColor(RED);      // .. color change
...
g2.setColor(YELLOW);
...
g2.setColor(GREEN);
```

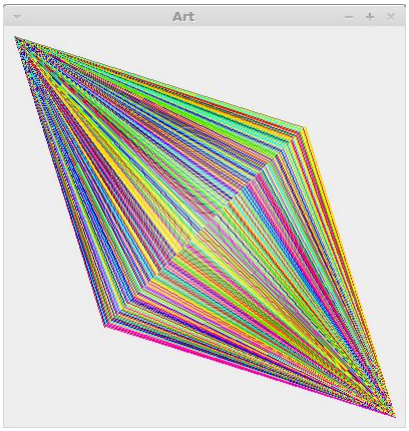
Det finns ett antal färdigdefinierade färger.

- Ange import raden så kan dessa användas

Ange färg för utritning

- setColor sätter en färg (inför nästa utritning).
  - Så länge vi inte ändrar färg kommer g2 att använda den senast angivna

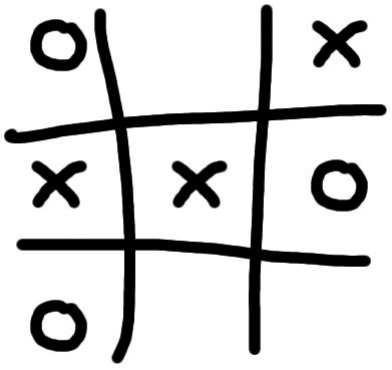
# Programmering




Hur skriver man ett program som ritar bilden?

- Använd Java 2D för att rita!

# Inför Övning 2



Vi gör ett Tre i rad-liknande spel (TicTactToe) uppbyggt med metoder.

- Placera ut X och O om någon får 3 i rad så vinner denna. Om ej …
- … flytta märkena horisontellt eller vertikalt tills 3 i rad …
- … eller avsluta (oavgjort).