Föreläsning 2

Täcker material från lektion 1, 2, 3 och 4:

- Datatyper
- Aritmetik
- Tecken och strängar
- Klasser, Objekt
- Metoder
- Villkor, villkorssatser och iterationer
- `main`-metoden
- Exempel

Datatyper

- Primitiva typer:
  - *heltalstyper*: `byte`, `short`, `int`, `long`
  - *flyttalstyper*: `float`, `double`
  - *övriga*: `char`, `boolean` (Anm: `char` kan räknas som heltalstyp)
- Objekttyper: `String`, `World`, `Turtle`

För de primitiva typerna gäller att

- de är inbyggda i språket (går inte att göra egna),
- de har väldefinierade värdemängder,
- de har ett antal *operatorer* som verkar på dem: (`+`, `<`, `&&`, ...) och
- det går att typkonvertera (implicit eller explicit) mellan dem (ej `boolean`).

Alla uttryck (utom anrop av `void`-metoder) har en typ!

Klasser och objekttyper

- Klassen `String`: representerar *textsträngar*. Inbyggd i själva språket
- Klasserna `World` och `Turtle`: Definieras i programvara som ni hämtade första lektionen.
- Ni kommer att definiera många egna objekttyper (klasser) under kursens gång.

quiz

Quiz

Datatypen `char`

Primitiv datatyp som representerar exakt ett tecken (bokstav, siffra, skiljetecken, . . .)

Representeras som 16-bits heltal med så kallad *Unicode*. Kan alltså representera MÅNGA olika tecken.

Teckenkonstanter omges med apostrofer ('). Exempel: `'a'`, `'1'`. och `'?'`

Typkonverteras autmatiskt till `int` i uttryck.

Datatypen char

Prova följande uttryck i interaktionsrutan:

```
'A'

'A' + 1

(char)('A' + 1)

(char)((int)(Math.random()*26) + 'a')
```

Klassen Math

Klassen `Math` används för att samla ihop matematiska funktioner och konstanter – man gör aldrig `new Math()`.

Funktionerna anropas via klassnamnet. Exempel:

```
Math.exp(3.5)
Math.pow(sin(2.5),2)
Math.PI + Math.sinh(x/2)
```

Se Javadoumentationen

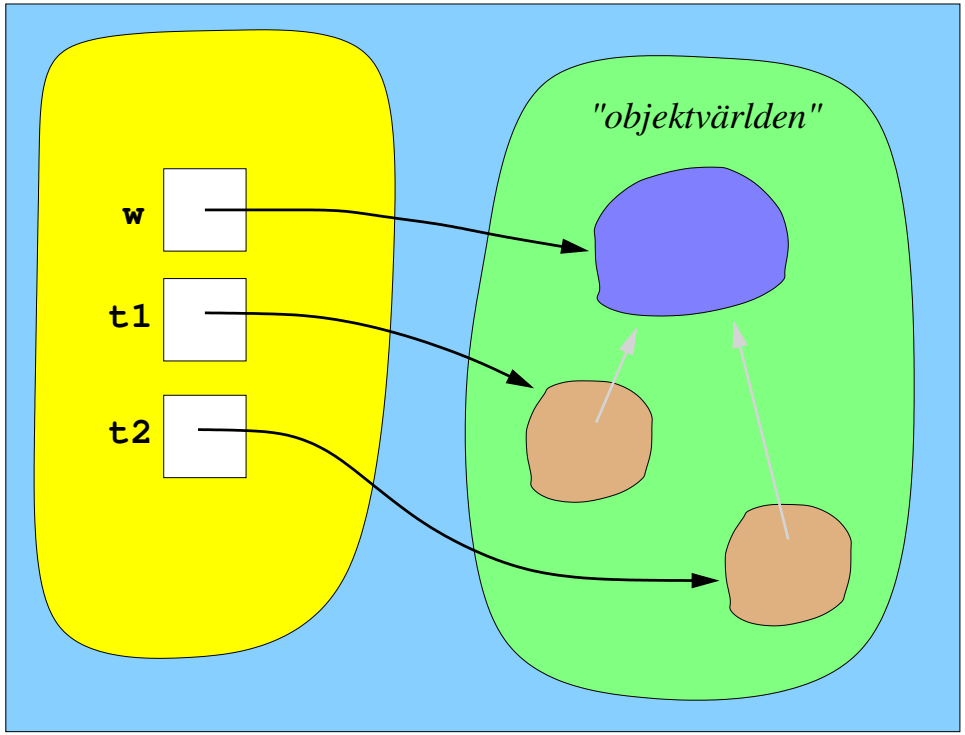
Klasserna World och Turtle

Klasserna `World` och `Turtle` gör man vanligen *objekt* av.

Objekt skapas med operatorn **new**.

Exempel:

```
World w = new World(400,400);
Turtle t1 = new Turtle(w);
Turtle t2 = new Turtle(w);
```



Variablerna `w`, `t1` och `t2` håller reda på de skapade objekten (''referenser", ''pekare'').

Metoder

Vi anropar metoder för att få reda på egenskaper hos objekten.

Exempel:

```
int dx = t1.getXPos() - t2.getXPos();
```

Vi anropar också metoder för att beordra objekten att utföra saker.

Exempel:

```
t1.enablePath();
```

Observera skillnaden mot när vi anropar metoder (funktioner) i klassen `Math`!

Parametrar

Ibland vill man skicka information till metoden. För detta används *parametrar*.

Exempel:

```
t1.turn(45);
t1.move(100);
t1.turnTo(0, 0);
t2.setSize(2.0);
t2.setColor(255, 0, 0);
```

Av anropen framgår dels vilket objekt som skall påverkas och dels eventuell information som skickas till metoden — de *aktuella parametrarna*.

Som aktuell parameter kan ett godtyckligt uttryck *av rätt typ* användas. Ex:

```
t1.move((int)(Math.random()*100));
```

Summering av metoder i klassen `Turtle`

Hittills har vi sett:

`move(int n)`	Gå `n` steg framåt
`turn(int n)`	Ändra riktning `n` grader
`turnTo(int x, int y)`	Sätt rikningen mot punkten (`x`,`y`)
`setSize(double size)`	Sätt storlek
`setColor(int red, int green, int blue)`	Sätt färg
`disablePath()`	Sluta rita spår
`enablePath()`	Börja rita spår

Skriva egna metoder

Exempel:

```
/**
 * Draws a triangle with specified side length
 */
 public void drawTriangle(int side) {
     this.move(side);
     this.turn(120);
     this.move(side);
     this.turn(120);
     this.move(side);
 }
```

DrJava

Metoder har

- en *synlighet* — i detta fall **public**
- en *returtyp* — i detta fall **void** som anger att inget värde returneras
- 0 eller flera *formella parametrar* — i detta fall 1 av typ **int**

Metoder

- Metoder måste *alltid* placeras i en klass
- I DrJava har man den klass som man vill redigera *definitionsrutan*
- En metod kan innehålla *lokala* variabler. Existerar bara i metoden och "glöms bort" när man lämnar metoden.
- Man måste alltid *kompilera* koden innan den kan användas

Strängar samt utmatning i terminalfönster

Generellt sätt att få se resultat från ett program. Exempel:

```
String namn="Tom";
System.out.print("Välkommen ");
System.out.print(namn);
System.out.print('!');
System.out.println();
```

Kortare:

```
String namn="Tom";
System.out.println("Välkommen " + namn + '!');
```

Observera: `println` får en sträng som bildats genom *konkatenering*

Algoritmer

En *algoritm* är en följd av väldefinierade instruktioner för hur en uppgift skall lösas.

Vilka är de väldefinierade instruktionerna? Beror på!

I Java är det *uttryck* (konstanter, variabler, metodanrop kombinerade med operatorer.

Ett uttryck blir en *sats* när det följs av ett semikolon (undantag: interaktionrutan i DrJava)

Exempel: metoden `drawTriangle` ovan.

Algoritmformuleringar

Det finns följande generella sätt att formulera algoritmer med hjälp av de tillgängliga instruktionerna:

- *sekvens* — utför en följd av instruktioner i tur och ordning (ex: satserna i `drawTriangle)`
- *selektion* — välj mellan olika alternativa sekvenser
- *iteration* — upprepa en sekvens ett antal gånger
- *abstraktion* — sätt ihop en algoritm enl ovanstående punkter och ge hopsättningen ett namn (ex: `drawTriangle`)

Selektion: `if`-satsen

Utför satser om ett visst *villkor* är sant

if (*villkor*) {
    satser
}

Exempel: Beräkna max av två tal

```
max = x;
if (y > x) {
   max = y;
}
```

Selektion: `if`-satsen

Variant: Utför *olika* satser beroende på om ett visst *villkor* är sant eller ej.

if (*villkor*) {
    satser
} **else** {
    satser
}

Exempel:

```
if (y > x) {
   max = y;
} else {
   max = x;
}
```

Selektion: `if`-satsen

Båda exemplen ovan förutsätter att variabeln `max` liksom `x` och `y` är deklarerade "i förväg".

Varför är det för fel på nedanstående två exempel?

```
if (y > x) {
   double max = y;
} else {
   double max = x;
}
```

```
if (y > x) {
   double max = y;
} else {
   max = x;
}
```

if-satsen: Exempel

Satserna i grenarna är godtyckliga satser — t ex **if**-satser

Exempel: Beräkna det minsta värden `x`, `y` och `z`

```
if (x < y) {
    if (x < z) {
        min = z;
    } else {
        min = x;
    }
} else {
    if (y < z) {
        min = y;
    } else {
        min = z;
    }
}
```

Fast enklare:

```
min = x;
if (y<min) {
    min = y;
}
if (z<min) {
    min = z;
}
```

eller `min = Math.min(x, Math.min(y, z));`

Iterationer

I java finns det tre konstruktioner för upprepning av kod: **while**, **for** och **do**.

Här tar vi upp **while**-satsen

while (*villkor*) {
    satser
}

Exempel: Beräkna x upphöjt till 10

```
int i = 1;
double p = 1;
while (i <= 10) {
      p = p*x;
      i++;
}
```

(eller `Math.pow(x,10)`)

Logiska uttryck

Villkoren i `if`- och `while`-satserna ska skriva som *logiska* eller `boolska` uttryck.

Sådana byggs upp av:

- de *logiska konstanterna* `true` och `false`,
- *relationsuttryck* som skrivs med relationsoperatorerna:
  `<`, `<=`, `==`, `>=`, `>` och `!=` samt
- de *logiska operatorerna*:
  `&&` för *and*,
  `||` för *or* och
  `!` för *not*.

Exempel: `x<y && z==2 || a!=b`

Se vidare minilektionen om logiska uttryck!

main-metoden

Antag att jag vill skapa en värld med några paddor och låta de gå omkring på något sätt dvs lite `new`, några `move`, `turn`, ...

Var skall jag göra detta?

Två möjligheter:

- i interaktionsrutan eller
- i en metod

Interaktionsrutan passar bra för att skriva enskilda uttryck men besvärligare när det blir många uttryck och satser.

Dessutom: finns bara i DrJava.

Alltså: i en metod!

main-metoden

Var skall metoden placeras?

Måste vara i en klass!

Klassen `Turtle`?

Objekt ur klassen `Turtle` representerar *en* padda. Onaturligt med en "vanlig" objektmetod skapar båda världar och paddor.

Alternativen är då att antingen

- göra en *klassmetod* eller
- skriva en *ny klass* t ex kallad `TestTurtles`

main-metoden i klassen Turtle

En *klassmetod* hör till klassen som helhet, inte till enskilda objekt.

Anges som **static** i metodhuvudet. Exempel:

```
public static void main() {
    World w = new World(500,500);
    Turtle t1 = new Turtle(w);
    Turtle t2 = new Turtle(w);
    int step = 30;
    int i = 1;
    while (i<20) {
        t1.move(step);
        t1.turn(-15);
        t2.move(step);
        t2.turn(15);
        i++;
    }
}
```

Körs med kommandot
`Turtle.main()`
i interaktionsrutan

main-metod i klassen Turtle

Hur kör man ett program om man inte har en interaktionsruta?

Det finns en definierad *startmetod* i Java. Den heter `main` som i föregående exempel men den har en array-parameter. Metodhuvudet ser typiskt ut så här:

```
public static void main(String[] args)
```

Man kan fortfarande köra den från interaktionsrutan men det vanliga sättet är att använda *Run*-knappen.

Observera att metoden måste deklareras *exakt* som ovan för (parametern `args` får heta något annat) för att det skall fungera med *Run*-knappen.

main-metod i annan klass

```
public class TestTurtles {

   public static void main(String[] args)  {
      World w = new World(500,500);
      Turtle t1 = new Turtle(w);
      Turtle t2 = new Turtle(w);
      int step = 30;
      int i = 1;
      while (i < 20) {
        t1.move(step);
        t1.turn(-15);
        t2.move(step);
        t2.turn(15);
        i++;
      }
   }
}
```

Klassen innehåller en enda metod — `main`-metoden
Det är den aktuella klassens `main`-metod som körs vi *Run*

DrJava

Tidsfördröjning med Thread.sleep

```
public static void main(String[] args)
   throws InterruptedException
 {
     World w = new World(500,500);
     Turtle t1 = new Turtle(w);
     Turtle t2 = new Turtle(w);
     int step = 30;
     int napTime = 500;
     int i = 1;
     while (i < 20) {
       t1.move(step);
       t1.turn(-15);
       t2.move(step);
       t2.turn(15);
       i++;
       Thread.sleep(napTime);
     }
 }
```

Klasser

En *klass*-deklaration beskriver vilka egenskaper en viss typ av objekt har och vad man kan göra med dessa objekt.

Exempel: Ett objekt ur klassen `Turtle` har ett antal *egenskaper* (färg, storlek, position, ...)

Det finns också ett antal *metoder* som berättar saker om objektet eller ändrar objektet på något sätt.

Java-programmering består av att *designa* och *implementera* klasser.

Nästa föreläsning börjar vi diskutera hur man skriver egna klasser. Idag ska vi titta på några metodexempel.

Exempel på metoder

En metod som tar två heltalsparametrar och som returnerar -1 om den första är minst, 0 om de är lika och 1 om den första är störst

```
public int compare(int x,
                   int y)
{
   int result;
   if (x < y) {
      result = -1;
   } else if (x == y) {
      result = 0;
   } else {
      result = 1;
   }
   return result;
}
```

```
public int compare(int x,
                   int y)
{
   if (x < y) {
      return -1;
   } else if (x == y) {
      return 0;
   } else {
      return 1;
   }
}
```

När en `return`-sats utförs lämnas metoden.

Var ska compare-metoden placeras?

Metoden har ju inget med Turtlar eller någon annan typ av objekt att göra så det är bäst att göra en egen klass t ex `MyMath`.

Den bör också deklareras som `static` eftersom vi inte vill behöva skapa något objekt för att använda den. Anropas med `MyMath.compare(...)`

```
public class MyMath {

      public static int compare(int x, int y) {
         if (x < y) {
            return -1;
         } else if (x == y) {
            return 0;
         }
         return 1;
      }

      // och andra metoder
```

Exempel: Andragradsekvation

Skriv en metod `double maxRoot(double a, double b)` som beräknar och returnerar den största av rötterna till ekvationen $x^2 + px + q = 0$.

Metoden ska ge en felutskrift om komplexa rötter.

Fråga: Varför inte båda rötterna?

Svar: Vi har inget sätt att returnera två värden (än).

Metoden deklareras `static` och placeras i den ovan nämnda klassen `MyMath`.

Andragradsekvation forts

```
public static double maxRoot(double p, double q) {
    double d = (p*p - 4*q)/4;
    if (d < 0) {
        System.out.println("Complex roots");
        return Double.NaN;     // "Not a Number"
    } else {
        d = Math.sqrt(d);
        double x1 = -p/2 + d;
        double x2 = -p/2 - d;
        if (x1 > x2) {
            return x1;
        } else {
            return x2;
        }
    }
}
```

Exempel: Primtal

Uppgift: Skriv en metod `isPrime(int n)` som returnerar `true` om `n` är ett primtal, annars `false`.

En enkel algoritm: Om talet är jämnt delbart med något tal mindre än $\sqrt{n}$ så är det inte ett primtal.

```
public static boolean isPrime(int n) {
   int i = 2;
   int lim = (int) Math.sqrt(n);
   while (i < lim) {
      if (n%i == 0) {
         return false;
      }
      i++;
   }
   return true;
}
```

Övning

Skriv en metod `isPrimeTwin(int n)` som returnerar `true` om n är det första talet i en *primtalstvilling* dvs om n och $n + 2$ båda är primtal, annars `false`.

```
public static boolean isPrimeTwin(int n) {

    if  (isPrime(n) == true) {
         if (isPrime(n+2) == true) {
            return true;
         } else {
            return false;
         }
     } else {
        return false;
     }
}
```

Vad tycker ni om denna lösning?

isPrimeTwin lite enklare

```
public static boolean isPrimeTwin(int n) {
     if (isPrime(n) == true && isPrime(n+2) == true) {
            return true;
         } else {
            return false;
         }
     }
}
```

`if`-uttrycket kan skrivas enklare:

```
     if (isPrime(n) && isPrime(n+2) ) {
```

isPrimeTwin ännu enklare

```
public static boolean isPrimeTwin(int n) {
    return isPrime(n) && isPrime(n+2);
}
```

Enkelt och lättförståeligt!

Lägg ner lite möda på att uttrycka algoritmerna klart och koncist!

Övning

Skriv kod som skriver ut det största primtalet mindre än 1000.

```
int n = 999;
while (isPrime(n) == false) {
   n--;
}
System.out.println("Largest prime less than 1000 is " + n);
```

Alternativ med

```
...
while (!isPrime(n)) {
   ...
```

Övning

Skriv en metod `int sumPrimes(int n)` som beräknar och returnerar summan av alla primtal som är $\leq n$

```
public static int sumPrimes(int n) {
  int sum = 0;
  int i= 2;
  while (i <= n ) {
    if (isPrime(i)) {
      sum += i;
    }
    i++;
  }
  return sum;
}
```

-][]