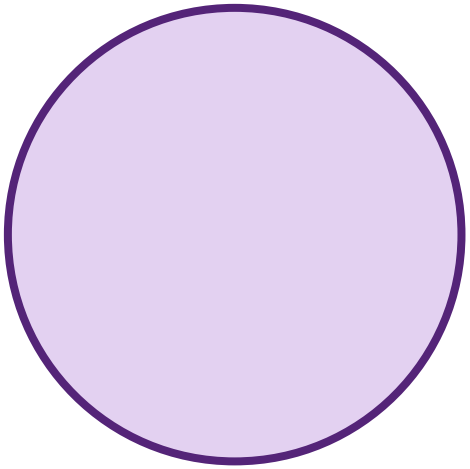
Introduktion till objektorienterad programmering i Java

OO steg 1:
Klasser

Klasser

jonkv@ida

- Bilar är komplicerade – vi tar *cirklar* som exempel (ritprogram?)



En klass i Java

jonkv@ida

- För att lagra info om **cirklar** som **objekt** i **Java**:
  - Skapa en **cirkelklass** som beskriver ”cirklar i allmänhet”

Varje Java-klass är i sin **egen fil**, med "samma" **namn**
(*Avancerat undantag*: nästlade klasser)

Circle.java

```
class Circle {
    Definiera fält                (se del 2)
    Definiera konstruktorer       (se del 4)
    Definiera metoder             (se del 3)
}
```

Medlemmar

…och vad ska klasser heta?

Klassnamn

jonkv@ida

- Klasser är **saker (substantiv)**
  - Namnges i **singular**
    - **class** Circle ➔ create a new Circle
    - **class** Tower ➔ every Tower has a Location
  - **Plural blir ”fel”**
    - **class** Locations? ➔ every Locations has...
  - **Verb blir ”fel”**
    - **class** MakeEnemies ➔ EnemyMaker

Klassnamn använder "**CamelCase**":
ArrayList, ProcessBuilder,
StackTraceElement

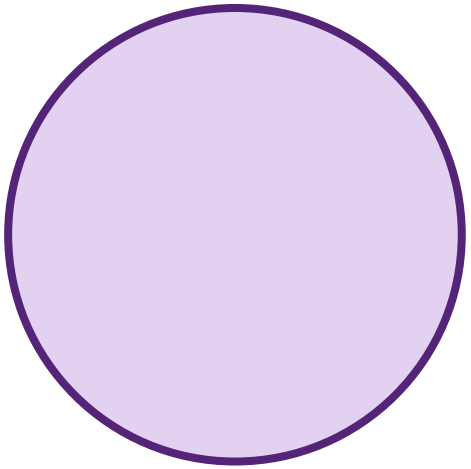
OO steg 2:
Egenskaper och fält

Egenskaper

jonkv@ida

- Vi har nämnt att **mojänger/objekt har egenskaper**
  - Vissa kan vara ***konstanta***, andra kan ***ändras*** efter att objektet skapas

Längd/höjd
Färg
Motorstyrka
Toppfart
Nuvarande fart
Kvarvarande bränsle



Koordinater (x,y)
Radie (radius)

Namngivning

- Egenskaper ska ha **beskrivande namn**, precis som variabler
  - En lista som innehåller alla *nätverksuppkopplingar*:
    - **c** // Contains all current connections
      ➔ **connections**, **currentConnections**
    - **(allaUtomFörstaBörjarMedStor,**
      ger skillnad mot klasser men man ser var ord börjar**)**

Bra namn ➔ färre kommentarer behövs!

Sträva efter självdokumenterande kod!

  - En egenskap som är sann om spelet är igång:
    - **if** (**startGame**)
      ➔ **if** (**gameRunning**)

  - **Fler exempel**
    - **x,y** – OK, koordinater
    - **temp**, **foo**, **bar** – vad är detta?
    - **index** – eller **selectedIndex**?
    - **left** – eller **livesLeft**?
    - **s**=52 – eller **deckSize**=52?
    -

Deklarera fält i en klass

jonkv@ida

- Egenskaper deklareras som **fält** (**medlemsvariabler**) i en **klass**

All kod för klassen
är *inuti* klassdeklarationen

Circle.java

```
class Circle {
    double x, y;
    double radius;
}
```

Ange *datatyp* för varje fält

En klass
talar om vilken sorts information
alla cirkelobjekt ska innehålla

Ännu har vi inte **skapat** någon cirkel,
bara en ”ritning” eller ”stämpel”!

medlemsvariabel = member variable
fält = field
klass = class

Lagra fält i ett objekt

jonkv@ida

- Varje **objekt** vi skapar får sitt **eget lagringsutrymme** för fälten
  - Alla cirklar **har** en radie;
    varje cirkel har **sin egen** radie

Datorns minne

Object header:	(data)
x	12.7
y	4.512
radius	0.0002
Object header:	(data)
x	23.9
y	3.222
radius	12.7

8-12-16 bytes
”extra objektinfo”,
inkl *typinfo* (”Circle”)

3*8=24 bytes för fälten

ÖVERKURS:
Project Valhalla ➔ Value Types,
objekt ”inline” utan headers

Python 3.9:
48+3*24 = 120 bytes
för motsvarande
objekt

Att skydda data

jonkv@ida

- Fält har **skyddsnivåer**:
  - **private** – bara **klassens egen** kod kan komma åt fältet
    - Klassen har full kontroll – oftast bra!
  - **public** – **alla** kan komma åt fältet
    - T.ex. svårt att ändra datatyp senare – annan kod kan sluta fungera
  - (Mer diskussion i ett annat avsnitt)

Princip:
Färre som kommer åt något → enklare att verifiera, ändra

Circle.java

```
public class Circle {  // Klasser är oftast public
    private double x, y;
    private double radius;
}
```

Punktnotation

- Använd **punkt** för att komma åt ett fält i ett objekt

Circle.java

```
public class Circle {
    private double x, y;
    private double radius;
    public static void inspect(Circle circ) {
        System.out.println("Hey! You gave me a circle!");
        System.out.println("Its radius is " + circ.radius);
    }
}
```

Fält och "final"

- Om inget annat sägs, kan fältvärden **ändras**

Person.java

```
public class Person {
    private String personnummer;
    public static void modify(Person person) {
        person.personnummer = “This is garbage!”
    }
}
```

- För att förhindra det: Deklarera fältet **final**

Person.java

```
public class Person {
    private final String personnummer;
    public static void modify(Person person) {
        person.personnummer = “This is garbage!”
    }
}
```

Kompileringsfel!

Alla fält är final → klassens objekt är ***immutable*** (≈ oföränderliga)

Mer om final när vi diskuterar pekare!

OO steg 3:
Beteende och metoder

Beteenden

jonkv@ida

Objekt i verkligheten
gör saker, har **beteenden**

Spontant, "aktivt":
Kamelen *väljer* att spotta

Inte associerat med just OO
Se multitrådning
eller autonoma agenter (AI)

Reaktivt, på begäran:
Tryck på tutan → bilen tutar

Uppnås via metoder (methods)
som kan anropas

Metoder

- Varje cirkel ska kunna beräkna sin **area**
  - Deklarera en **metod** som returnerar detta

Metoder placeras oftast *efter* fält

Circle.java

```
public class Circle {
   private double x, y;
   private double radius;
   public double getArea() {
      return Math.PI * radius * radius;
   }
   public void setRadius(double newR) {
      radius = newR;
   }
}
```

Ange alltid vad som returneras (datatyp eller void)

Parametrar har typer, namn

Metodkroppen

jonkv@ida

- I metodens **kropp (body)** anges vad metoden ska göra
  - Som i en "vanlig" funktion – med en skillnad:

Circle.java

```
public class Circle {
   private double x, y;
   private double radius;
   public double getArea() {
      return Math.PI * radius * radius;
   }
   public void setRadius(double newR) {
      radius = newR;
   }
}
```

Kan direkt komma åt objektets fält!

…Men **vilken radie**
i **vilket objekt**?

Vi kan skapa 1000 cirklar,
och var och en har en **egen** radie!

this

This 1: Detta objekt

jonkv@ida

- Om vi tar bort "förkortningar":

Circle.java

```
public class Circle {
   private double x, y, radius;
   public double getArea() {
      return Math.PI *                    ;
   }
   public static void main(String[] args) {
      Circle c1 = …;
      Circle c2 = …;
      double a = c1.getArea();
      double b = c2.getArea();
   }
}
```

"radius" var en *förkortning* för "this.radius"

Fältet "radius" i objektet "this"… men vad är "this"?

"this" är **objektet vars metod anropades**!
I första anropet: this är samma objekt som c1
I andra anropet: this är samma objekt som c2

(Python: "self")

This 2: Omskrivning

jonkv@ida

- **Nästan** som om kompilatorn **skrev om koden**
  - Adderade en *this-parameter* till varje metod

```
public class Circle {
   private double x, y, radius;
   public double getArea() {
      return Math.PI *
         this.radius * this.radius;
   }
   public static void main(String[] args) {
      Circle c1 = …;
      double a = c1.getArea();
   }
}
```

```
public class Circ
   private double x, y, radius;
   public double getArea(Circle this) {
      return Math.PI *
         this.radius * this.radius;
   }
   public static void main(String[] args) {
      Circle c1 = …;
      double a = getArea(c1);
   }
}
```

"Implicit parameter": **this**

(*Nästan*: Det finns viktiga skillnader, t.ex. i ärvning)

This 3: Utelämna this

jonkv@ida

- Kan ofta utelämna **this**
  - Givet en identifierare "radius" söker kompilatorn efter:
    1. Lokala variabler och parametrar "radius"
    2. Fält "radius"

"Det finns inget radius, så de måste mena this.radius…"

```
public class Circle {
   private double x, y, radius;
   public double getArea() {
      return Math.PI *
         this.radius * this.radius;
   }
   public static void main(String[] args) {
      Circle c1 = …;
      double a = c1.getArea();
   }
}
```



```
public class Circle {
   private double x, y, radius;
   public double getArea() {
      return Math.PI * radius * radius;

   }
   public static void main(String[] args) {
      Circle c1 = …;
      double a = c1.getArea();
   }
}
```

This 4: Gömda fält

jonkv@ida

- Men variabler och parametrar **gömmer** fält ("hiding")

Olika namn (radius, newR)

```
public class Circle {
   private double x, y, radius;
   public void setRadius2(double newR) {
      System.out.println(radius); // Finns variabel/parameter? Nej! Skriv ut fältet
      radius = newR;              -// Sätter fältets värde
   }
}
```

Samma namn (radius), parameter *gömmer* fält

```
public class Circle {
   private double x, y, radius;
   public void setRadius2(double radius) {
      System.out.println(radius);   // Finns variabel/parameter? Ja! Skriv ut den
      System.out.println(this.radius);   // Måste vara fältet
      radius = radius;              // Sätter argumentet till sitt eget värde...
      this.radius = radius;         // Sätter fältets värde
   }
}
```

Namngivning av metoder

Namngivning 1

- Namngivning: Ge **beskrivande namn** som normalt är **verb**
  - **allaUtomFörstaBörjarMedStor**
    - update(); // Moves all enemies
      ➔ moveEnemies();
    - doIt() ➔ **shuffleDeck**()
    - units() ➔ **setUnitType**()
    - moveList() missvisande om metoden *returnerar* en lista på förflyttningar
      ➔ **getMoveList**()
    - bandit() ”Kan du **bandita** lite?”
    - …
  - Vi **säger till** objektet att **göra** något!
    - **playerMovement()?** ”Hey you, playerMovement() for me!”
    - **movePlayer**()? ”Hey you, move player for me!” – better
    - **keyBindingsAdder()?** ➔ addKeyBindings()

Namngivning 2: Overloading

- Metoder kan **överlagras** (**overloading**)
  - **Olika metoder** i **samma klass** med **samma namn**

```
public class Printer {
    public void print(int val) { … }
    public void print(double val) { … }
    public void print(double val, int precision) { … }
}
```

Skiljs åt av antal argument + argumenttyperna

  - Användning:
    - När metoderna **gör i princip samma sak**
    - När det vore **onödigt krångligt** att hitta på egna namn (printInt, printDouble, …)

OO steg 4:
Att skapa objekt – via konstruktorer

Skapande

jonkv@ida

- Att **skapa** nya listor, tupler osv. i Python:

```
tomLista    = []
minLista    = ["a", "b", "c"]
minTupel    = (12, 34, 56)
minDict     = { "a": 1, "b": 2 }
```

Egen syntax för listor, …

Ange alla element som listan / tupeln / … ska innehålla

Hakparenteser [] ***skapar nya listobjekt*** **i Python!**

Skapa objekt 1

jonkv@ida

- Hur **skapar** man nya objekt i **Java**?

Generell "objektsyntax",
ange värdet på varje fält?

```
Circle myCircle =
Circle[[12.7, 4.512, 0.0002]];
```

Nej: Många klasser har fält
som ska **beräknas**
eller kan **lämnas tomma**
eller är privata för **internt bruk**

Måste kunna **välja** vilka fält vi sätter

Datorns minne

Object header:	(data)
x	12.7
y	4.512
radius	0.0002

Skapa objekt 2

jonkv@ida

- Hur **skapar** man nya objekt i **Java**?

Först skapa "nollställt" objekt,
med alla fält "nollställda",
sedan sätta önskade värden utifrån?

```
Circle myCircle;
myCircle.x = 10;
myCircle.y = 20;
...
```

Nej: Tillåter godtyckliga värden
Listor kan innehålla vad som helst
Men *cirklar* ska ha radie > 0
Klassen ska kunna garantera detta!

Fundamental princip:
"Klassen bestämmer över sina objekt"
(kan förhindra manipulation utifrån

Datorns minne

Object header:	(data)
x	0
y	0
radius	0

Konstruktorer 1

jonkv@ida

- Låt klassen ange **konstruktorer**, speciella **metoder** för att initialisera objekt

```
public class Circle {
  private double x, y, radius;

  public Circle(double x, double y, double radius) {
    …
    …
    …
  }
}
```

Ingen returtyp (inte void!)
Samma namn som klassen

→ **Detta är en konstruktor!**

konstruktor = constructor

Konstruktorer 2

- **Ibland** vill vi:
  - Kunna **sätta specifika värden** på alla fält

```
public class Circle {
  private double x, y, radi
  public Circle(double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
}
```

En parameter per fält

En tilldelning per fält;
"this" är det nya objektet

```
public static void main(String[] args) {
    Circle c1 = new Circle(10.0, 20.0, 14.2);
}
```

Vi anropar konstruktorn
med **new**
och anger lämpliga parametrar

Konstruktorer 3: Vad händer?

jonkv@ida

- 1: Minne allokeras (reserveras) för objektet

Object header:	(skräp)
x	(skräp)
y	(skräp)
radius	(skräp)

- 2: Alla fält får defaultvärden
  - Heltal: 0
  - Floating point: 0.0
  - Boolean: false
  - Objekt: null (diskuteras senare)

Object header:	Circle …
x	0.0
y	0.0
radius	0.0

- 3: Konstruktorn anropas

```
Circle(double x, double y, double radius) {
  this.x = x;
  this.y = y;
  this.radius = radius;
}
```

Fundamental princip:
"Klassen bestämmer över sina objekt" – *klassen* anger konstruktorkoden!

Konstruktorer 4: Kontroller

jonkv@ida

- Ibland vill vi:
  - **Kontrollera** att bara **tillåtna värden** anges

```
public class Circle {
  private double x, y, radius;

  public Circle(double x, double y, double radius) {
    if (radius <= 0.0) {
      throw …signalera fel…;
    }
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
}
```

Se till att radien > 0,
annars avbryts konstruktorn
➔ inget objekt skapas

Konstruktorer 5

jonkv@ida

- Ibland vill vi:
  - Ta in värden i **annat format** än det klassen använder

```
public class Circle {
  private double x, y, radi

  public Circle(Point center, double radius) {
    if (radius <= 0.0) throw ...;
    this.x = center.x;
    this.y = center.y;
    this.radius = radius;
  }
}
```

Anta en "punktklass" som lagrar x, y

Ta ut x och y från "centrumpunkten"

Konstruktorer 6

jonkv@ida

- Ibland vill vi:
  - **Beräkna värden på fält** från andra parametrar

```
public class Circle {
  private double x, y, radius;
  private double area;

  public Circle(double x, double y, double circumference) {
    if (circumference<= 0.0) throw ...;
    this.x = x;
    this.y = y;
    this.radius = circumference / (2*Math.PI);
    this.area = Math.PI * radius * radius;
  }
}
```

Beräkna radie från omkrets

Beräkna area från radie

Konstruktorer 7

jonkv@ida

- Ibland vill vi:
  - **Sätta värden själva** eftersom de är "interna" detaljer, inte något som anroparen ska bry sig om

```
public class Circle {
  private double x, y, radius;
  private int timesPainted;

  public Circle(double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.timesPainted = 0;
  }
}
```

Håll reda på antalet gånger vi har ritat cirkeln (statistik)

Från början är antalet alltid 0

Konstruktorer 8

- En konstruktor **kan** göra **vad som helst**

```
public class Circle {
  private double x, y, radius;
  private boolean debugMode;
  public Circle(double radius) {
    System.out.println("Hello, world!");
    System.exit(0);
  }
}
```

Inte vad man ska göra,
men man kan

Normalt: konstruera objektet
(sätt snabbt de fält som behövs),
lämna andra uppgifter
till de vanliga metoderna

Konstruktorer 9: Kopiering

jonkv@ida

- Ibland kan man vilja **kopiera objekt**
  - Konstruktor som tar ett objekt av samma typ

```
public class Circle {
  private double x, y, radius;
  public Circle(double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
  public Circle(Circle other) {
    this.x = other.x;
    this.y = other.y;
    this.radius = other.radius;
  }
}
```

Kopieringskonstruktor:
Tar ett objekt av samma typ

Konstruktorer 10: Overloading

jonkv@ida

- En klass kan ha **flera konstruktorer**
  - Alla heter samma som klassen ➔ **overloading**

```
public class Circle {
  private double x, y, radius;
  public Circle(double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
  public Circle(Point center, double diameter) {
    this.x = center.x;
    this.y = center.y;
    this.radius = diameter / 2;
  }
}
```

Konstruktorer 11: Minst en!

jonkv@ida

- Varje klass har **minst en konstruktor**
  - Har du inte skrivit någon?
    Då skapar Java en **tom konstruktor** utan argument!

```
public class Circle {
    private double x, y, radius;
}
```

```
public class Circle {
    private double x, y, radius;
    public Circle() {
        // Gör ingenting ➔
        // alla fält har defaultvärden (0, 0.0, false, …)
    }
}
```

Fält, metoder och konstruktorer i uppräkningsbara typer

Uppräkningsbara typer

jonkv@ida

- Uppräkningsbara typer **är** (en särskild sorts) klasser

```
public enum DayOfWeek
{
   MONDAY(true), TUESDAY(true), WEDNESDAY(true), THURSDAY(true),
   FRIDAY(true), SATURDAY(false), SUNDAY(false);

   private final boolean isWorkDay;

   private DayOfWeek(boolean isWorkDay) {
      this.isWorkDay = isWorkDay;
   }

   public boolean isWorkDay() {
      return isWorkDay;
   }
}
```

Kan ha egna egenskaper som sätts i konstruktorn

En konstant som MONDAY är ett *objekt* som *känner till* information om sig själv

Skillnad från vanlig klass: Kan inte skapa nya objekt utöver de 7 vi deklarerade