Föreläsning 13-14

Innehåll

- Arv
- Repetition
- Om tentamen

Diskutera

- Här är början på klassen `MemoryWindow` som använts på en lab.
  - Vad kan menas med `extends SimpleWindow`?
  - Vad händer på raden `super(804, 804,''Memory'');`
    Ledning: Vilka parametrar har konstruktorn i SimpleWindow?

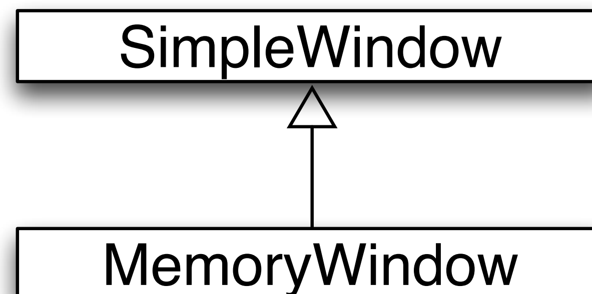
```
public class MemoryWindow extends SimpleWindow {
    private int imgSize;
    private MemoryBoard board;

    /** Skapar ett fönster som kan visa memorybrädet board. */
    public MemoryWindow(MemoryBoard board) {
        super(804, 804, "Memory");
        this.board = board;
        imgSize = 800 / board.getSize();
    }
    ...
```

Arv

- En klass kan ärva från en annan klass.
- Det innebär att klassen ärver attribut och metoder.
    - Det är i princip som om attributen och metoderna vore skrivna i klassen själv.
- Den klass som ärver kallas **subklass**, den klass den ärver ifrån kallas **superklass**.
- I exemplet från förra bilden ärver `MemoryWindow` från `SimpleWindow`.
    - Ett `MemoryWindow` kan samma saker som ett `SimpleWindow` och lite till.
    - Man kan säga att `MemoryWindow` utökar (eng. extends) `SimpleWindow`.
    - `MemoryWindow` är en specialisering av `SimpleWindow`.

Klassdiagram

- I ett klassdiagram kan man visa vilka klasser som används och hur de hänger ihop. UML (Unified Modelling Language) kan användas för detta.
- Klassdiagram med superklassen `SimpleWindow` och subklassen MemoryWindow:



Arv
extends

- Genom att skriva

  ```
  public class MemoryWindow extends SimpleWindow {
  ```

  blir `MemoryWindow` en subklass till `SimpleWindow`.

- Man skapar ett objekt av subklassen på vanligt sätt:

  ```
  ...
  MemoryBoard b = ...
  MemoryWindow mw = new MemoryWindow(b);
  ```

Arv

super

- Superklassen `SimpleWindow` har en konstruktor med satser för att tilldela värden till attribut för höjd, bredd och titel. Den måste också exekveras när man skapar ett `MemoryWindow`-objektet.
- Det gör man genom att anropa superklassens konstruktor *först* inuti subklassens konstruktor:

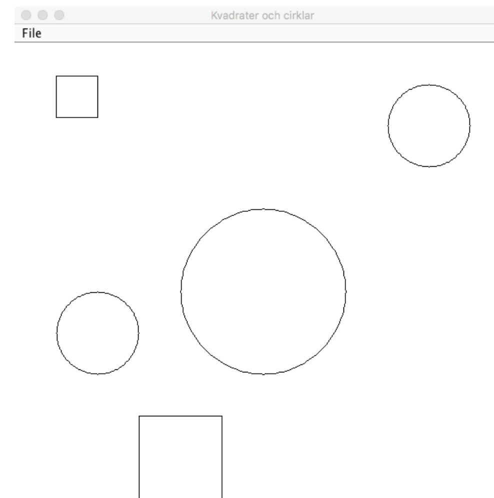
```
super(804, 804, "Memory");
```

Diskutera

I ett program ska geometriska former (kvadrater och cirklar) hanteras. Formerna ska kunna flyttas och ritas upp på skärmen.

- Formerna ska kunna ritas i ett `SimpleWindow`-fönster.
- Formerna ska kunna flyttas.
- Det ska vara lätt att lägga till nya former, t.ex. trianglar.

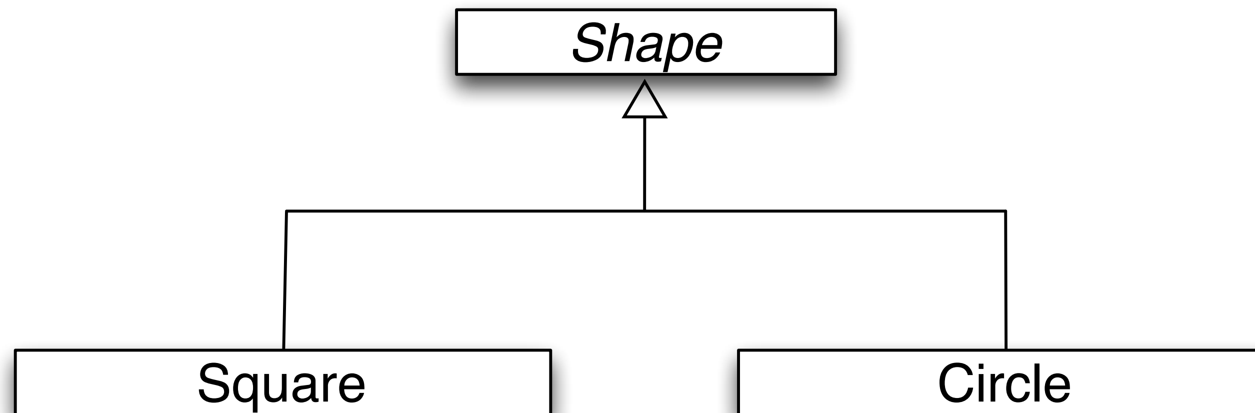
Vilka klasser behövs?



Exempel: Geometriska former

Klasser

- `Square` och `Circle` – med metoder för att rita.
- `Shape` – beskriver en geometrisk form med metoder för att ta reda på formens läge, flytta formen ....



Subklasserna Square och Circle

```
public class Square extends Shape {
    private int side;

    public Square(int x, int y, int side) {
        super(x, y);
        this.side = side;
    }
    ...
}

public class Circle extends Shape {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }
    ...
}
```

Superklassen Shape

```
public abstract class Shape {
    protected int x;
    protected int y;

    /** Skapar en geometrisk form med läget x,y. */
    protected Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Flyttar formen avståndet dx i x-led, dy i y-led. */
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    ...
}
```

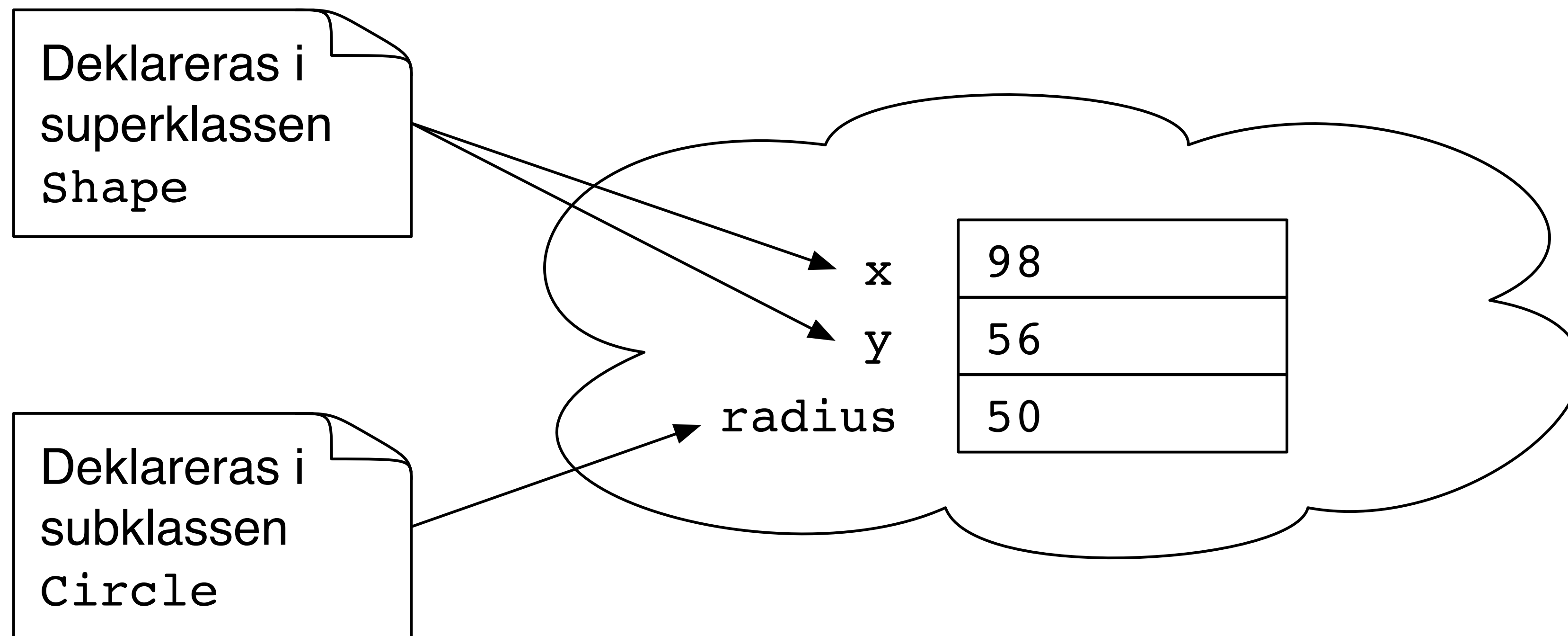
Abstrakta klasser

- I program som använder våra klasser för geometriska former kommer vi inte att skapa objekt av klassen `Shape`, utan bara av klasserna `Square` och `Circle`.
  - Vi kan rita kvadrater och cirklar, men ingenting som bara är en geometrisk form.
- Superklasser man *inte* ska skapa objekt av kallas **abstrakta klasser**.

```
public abstract class Shape {
    ...
}
```

Objekt innehåller både ärvda och egna attribut

Exempel: `Circle`-objekt



`public`, `private` eller `protected`

- I klassen måste man ange var ett attribut eller en metod ska vara åtkomlig:
  - `public` – åtkomlig utanför klassen
  - `private` – endast åtkomlig inuti i klassen
  - `protected` – åtkomlig i klassen och i alla dess subklassser
  - om man inte skriver någon av dessa modifierare får attributet/metoden ”paketskydd” och kan användas av andra klasser i samma paket.
- Ofta är attribut och konstruktorer i superklasser deklarerade protected.

När ska arv användas?

- Arv passar bra när man har en ”är en”-relation mellan klasser.
  - En kvadrat är en slags geometrisk form.
  - En cirkel är en slags geometrisk form.

Här ska arv *inte* användas

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

- Ska klassen `Point` vara superklass till klassen `Shape` så att klassen `Shape` kan ärva attributen `x`, `y` och metoden `move`?
- Nej! En geometrisk form är inte en slags punkt.

Sammansättning

- En geometrisk form kan däremot ha en punkt som anger formens läge:

```
public class Shape {
    private Point location // formens läge

    public Shapes(Point location) {
        this.location = location;
    }

    public void move(int dx, int dy) {
        location.move(dx, dy);
    }
}
```

Fördelar med arv

- Modellen beskriver det verkliga systemet bättre.
  - I problemet används både begreppet geometrisk form och kvadrater/cirklar.
- Gemensamma attribut och metoder behöver bara skrivas en gång i superklassen.
- Vi kan deklarera referensvariabler som får referera till objekt av alla de olika subklasserna. Exempel:

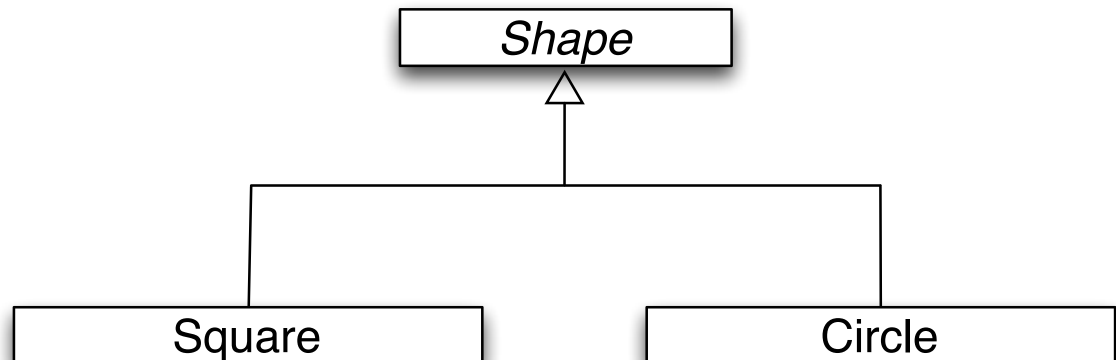
```
ArrayList<Shape> list = new ArrayList<Shape>();
list.add(new Circle(98, 56, 50));
list.add(new Square(205, 7, 100));
```

Referensvariabler och arv

En referensvariabel som deklarerats med typen `C` får referera till objekt av klassen `C` och dessutom till objekt av alla subklasser till `C`.

- Exempel: En variabel med typen `Shape` får referera till objekt av alla subklasser till `Shape`.

```
Shape aShape;
aShape = new Circle(98, 56, 50);
...
aShape = new Square(205, 7, 100);
```



Typregler för arv

Uppgift

En referensvariabel som deklarerats med typen `C` får referera till objekt av klassen `C` och dessutom till objekt av alla subklasser till `C`.

```
Shape s;
Circle c;
```

Vilka tilldelningssatser är korrekta?

1. `c = new Circle(100, 100, 50);`
2. `s = new Circle(100, 100, 50);`
3. `c = new Square(100, 100, 50);`
4. `s = new Square(100, 100, 50);`

Typregler för arv

Svar

```
Shape s;
Circle c;
```

Vilka tilldelningssatser är korrekta?

1. `c = new Circle(100, 100, 50);` Ok
2. `s = new Circle(100, 100, 50);` Ok
3. `c = new Square(100, 100, 50);` Fel
4. `s = new Square(100, 100, 50);` Ok

`s` får referera till `Square`- och Circle-objekt. `c` får bara referera till Circle-objekt.

Diskutera

Problem med metoden `draw`

- Ok:

```
Circle c = new Circle(100, 100, 50);
c.draw(w);
```

- Ok:

```
Shape s = new Circle(100, 100, 50);
s.move(10, 20);
```

- Borde fungera, men fungerar inte ännu:

```
Shape s = new Circle(100, 100, 50);
s.draw(w);
```

Varför fungerar inte `s.draw(w)`?

Abstrakt metod

- Eftersom `s` är deklarerad med typen `Shape` så söker kompilatorn i klassen `Shape` och uppåt i arvshierarkin efter `draw`.
  - Men det finns ingen metod `draw` i klassen `Shape`.
  - Istället är `draw` implementerad i subklasserna `Square` och `Circle`.
- För att vi ska kunna skriva `s.draw(w)` måste det finnas en metod `draw` i klassen `Shape`. Lösningen är att deklarera `draw` som en abstrakt metod:

```
public abstract class Shape {
    ...
    public abstract void draw(SimpleWindow w);
    ...
}
```

Abstrakt metod

- Nu ”finns” det en metod `draw` i klassen `Shape` och satsen `s.draw(w)` går att kompilera.
  - Eftersom `Shape` innehåller en abstrakt metod *måste* även `Shape` deklareras `abstract`.
- De riktiga `draw`-metoderna finns implementerade i subklasserna `Square` och `Circle`.
  - Alla subklasser till `Shape` *måste* implementera en `draw`-metod.

Deklarerad typ vs runtime-typ

```
Shape s = new Circle(100, 100, 50);
s.draw(w);
```

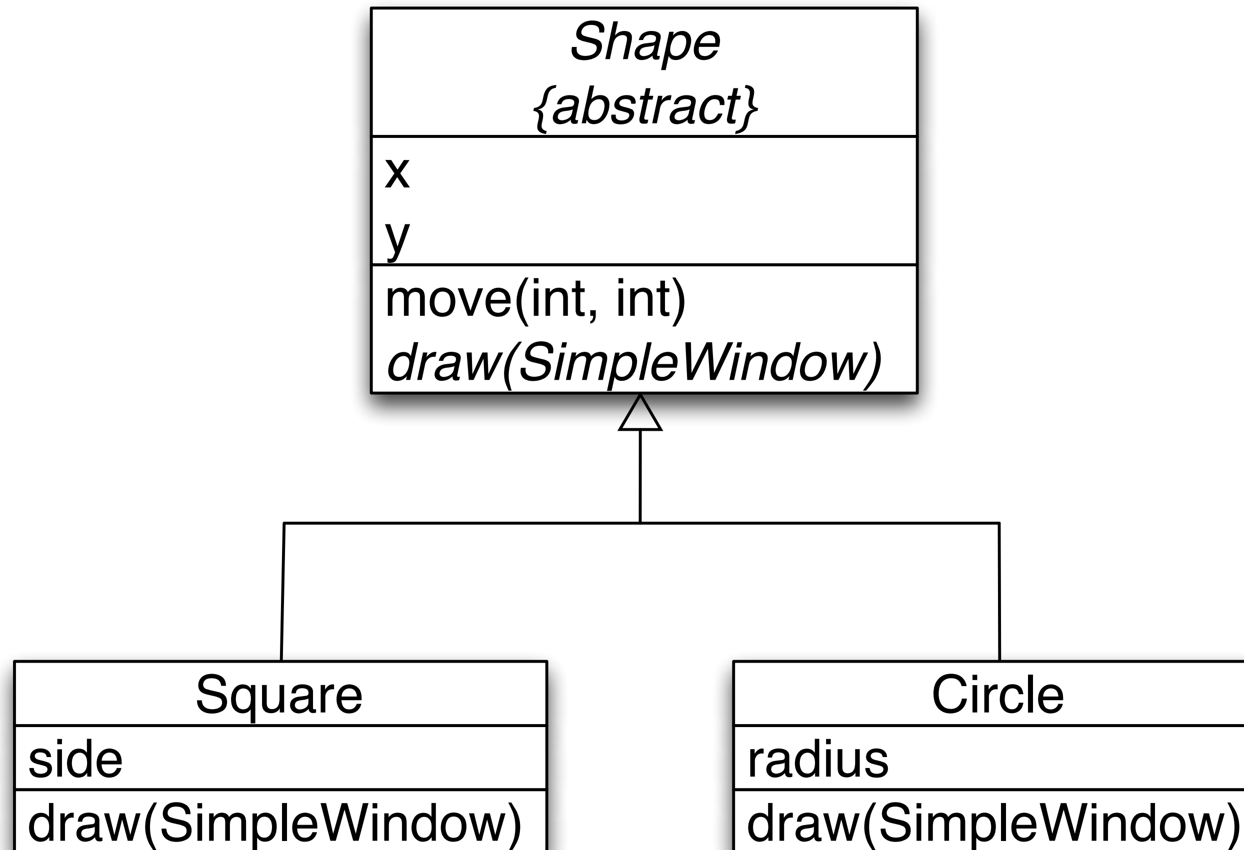
- Hur väljs rätt `draw`-metod vid exekveringen?
  - Ritas en kvadrat eller cirkel?
- Det beror på vad för slags objekt `s` refererar till. Dvs. vad som stod efter `new` när objektet skapades.
  - `new Square ...` eller `new Circle ...`
- Variabeln `s` har den deklarerade typen `Shape` men runtime-typen `Circle`.

Dynamisk bindning

- Java använder dynamisk bindning vid metodanrop. Det innebär att det är objektets typ som under exekveringen avgör vilken metod som ska anropas.
- När satsen `s.draw(w)` ska exekveras undersöker Java-systemet objektet `s` refererar till.
  - Om det är ett `Square`-objekt så anropas `draw` i `Square`, om det är ett `Circle`-objekt så anropas `draw` i `Circle`.

Exempel: geometriska former

Klassdiagram



Exempel: geometriska former

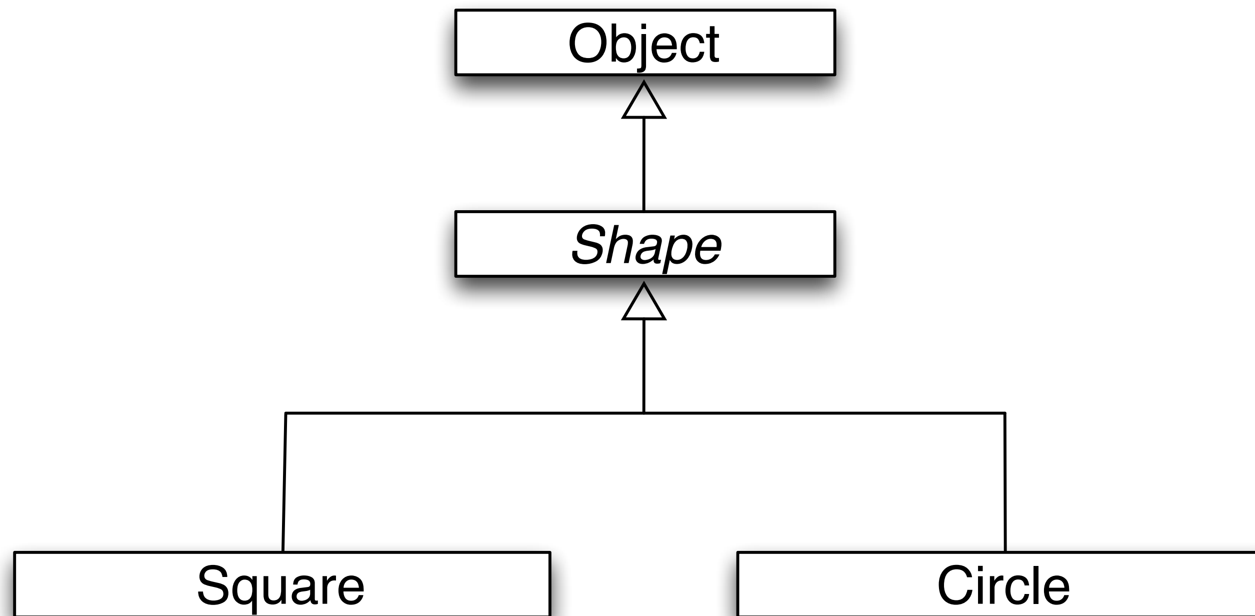
Programidé

- Skriv ett program som ritar några cirklar och kvadrater. Låt användaren klicka på en av formerna och flytta den formen till ett nytt läge.
- För att lösa problemet är det praktiskt att lägga till några metoder i klassen `Shape` som
  - undersöker om en punkt ligger ”nära” formen.
  - flyttar formen till ett nytt läge.
  - raderar bilden av formen i fönstret w.
- Programmet skulle t.ex. kunna användas till att göra en enkel skiss över en trädgård där cirklarna är träd och kvadraterna hus.
- Nya geografiska former kan lätt läggas till (blir nya subklasser).
- Data om de geografiska formerna kan läsas/skrivas på fil.

Klassen `Object`

- I Java finns en klass `Object` som är superklass till alla andra klasser.
- En referensvariabel av typen `Object` kan alltså referera till objekt av alla typer. Exempel:

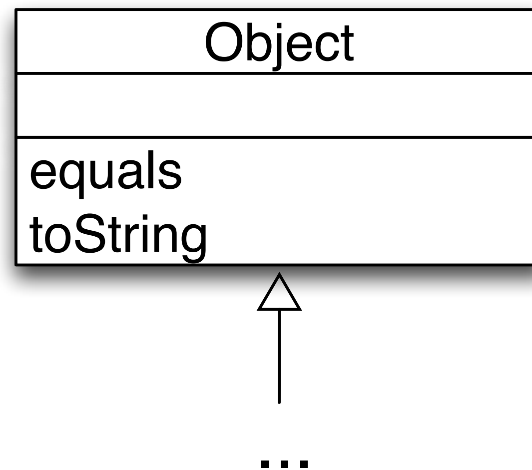
```
Object obj = new Square(100, 100, 50);
```



Klassen `Object`

metoder

- I klassen `Object` finns det några metoder (t.ex. `toString` och `equals`) som alla andra klasser ärver.



- Man kan definiera om (skugga) metoderna i sina egna klasser så att man får exakt det beteende man vill ha.

Metoden toString

- I klassen `Object` finns metoden `toString`:

```
/** Skapar en "läsbar representation" av objektet. */
String toString();
```

- När man skriver ut ett objekt, t.ex:

```
Square sq = new Square(100, 100, 50);
System.out.println(sq);
```

så är det metoden `toString` som används inuti `println`.

- Man kan definiera om (skugga) metoden `toString` så att man får exakt den utskrift man vill ha. Man kan t.ex. lägga till följande metod i klassen `Square`:

```
public String toString() {
    return "Sidlängd " + side + ", läge " + x + " " + y;
}
```

Metoden equals

- I klassen `Object` finns metoden `equals`:

```
/** Undersöker om detta objekt är lika med obj. */
boolean equals(Object obj);
```

- Metoden `equals` i `Object` fungerar likadant som ==, dvs. jämför referenser.

```
Square sq1 = new Square(100, 100, 50);
Square sq2 = new Square(100, 100, 50);
... sq1.equals(sq2)  // false om equals ej skuggad i Square
```

- Man kan själv definiera om (skugga) metoden `equals` så att den jämför innehållet i objekten istället. Detta är gjort i Javas standardklasser.

Checklista

Exempel på vad du ska kunna

- Förstå och rita mycket enkla klassdiagram i UML.
- Förklara begreppen: arv, superklass, subklass.
- Förklara begreppen: abstrakta klasser och metoder.
- Implementera superklasser och subklasser.