EDAA20
Programmering och databaser

Föreläsning 13 – Arv och uppgift om listor

2023-10-09, Niklas Fors

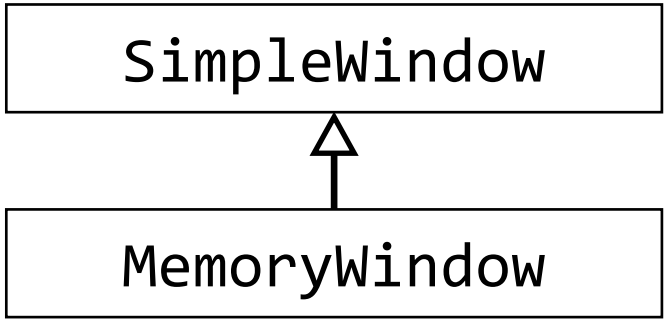
Arv

Objektorienterade språk stödjer ***arv*** för att
återanvända programkod *mellan* klasser.

En klass ***ärver*** från en annan klass och
får då alla attribut och metoder som den klassen har.

Exempel i labb 9:
`MemoryWindow` (***subklass***) ärver
från `SimpleWindow` (***superklass***)

Klassdiagram (UML)



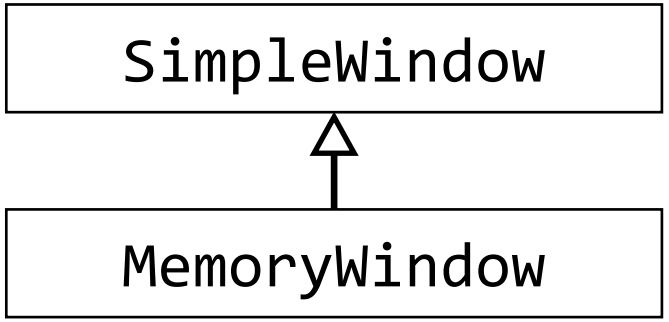
UML=Unified Modelling Language
Standard för att rita klassdiagram osv

Arv med `extends`

En klass ärver från en annan klass med `extends`:

```
public class MemoryWindow extends SimpleWindow {
   ...
}
```

Klassdiagram (UML)

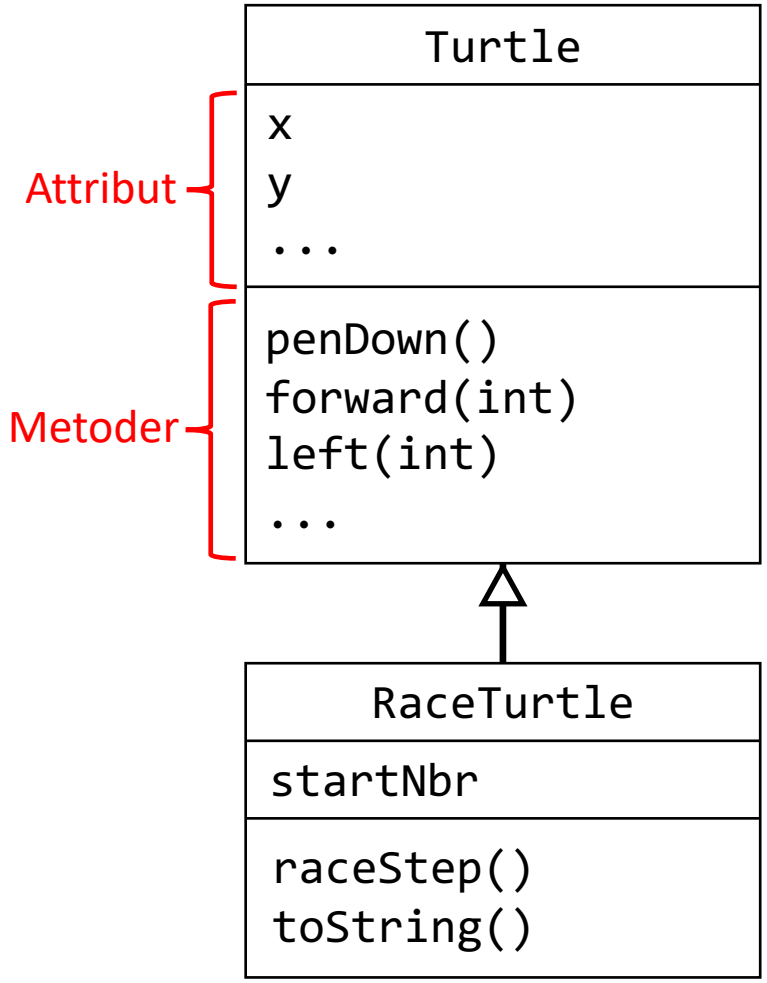


UML=Unified Modelling Language
Standard för att rita klassdiagram osv

Objekt skapas och metoder anropas som vanligt:

```
MemoryBoard board = ...
MemoryWindow window = new MemoryWindow(board);
window.drawBoard();           // metod deklarerad i MemoryWindow
window.waitForMouseClick();   // metod deklarerad i SimpleWindow
```

Klassdigram för `RaceTurtle` (Labb 11)



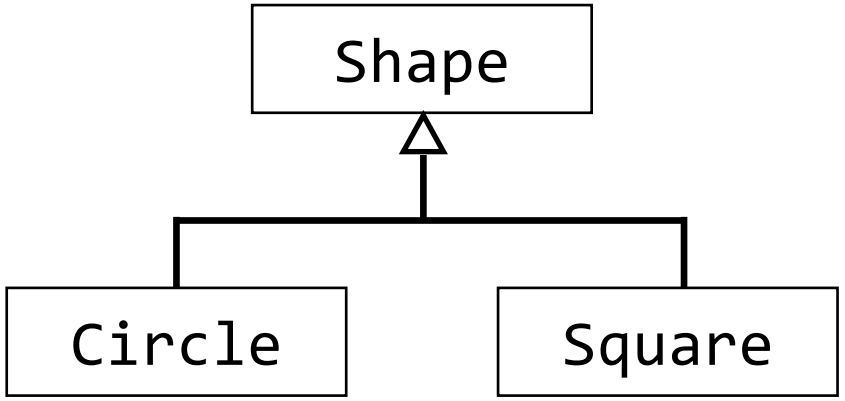
Klassen `Turtle` från Labb 6

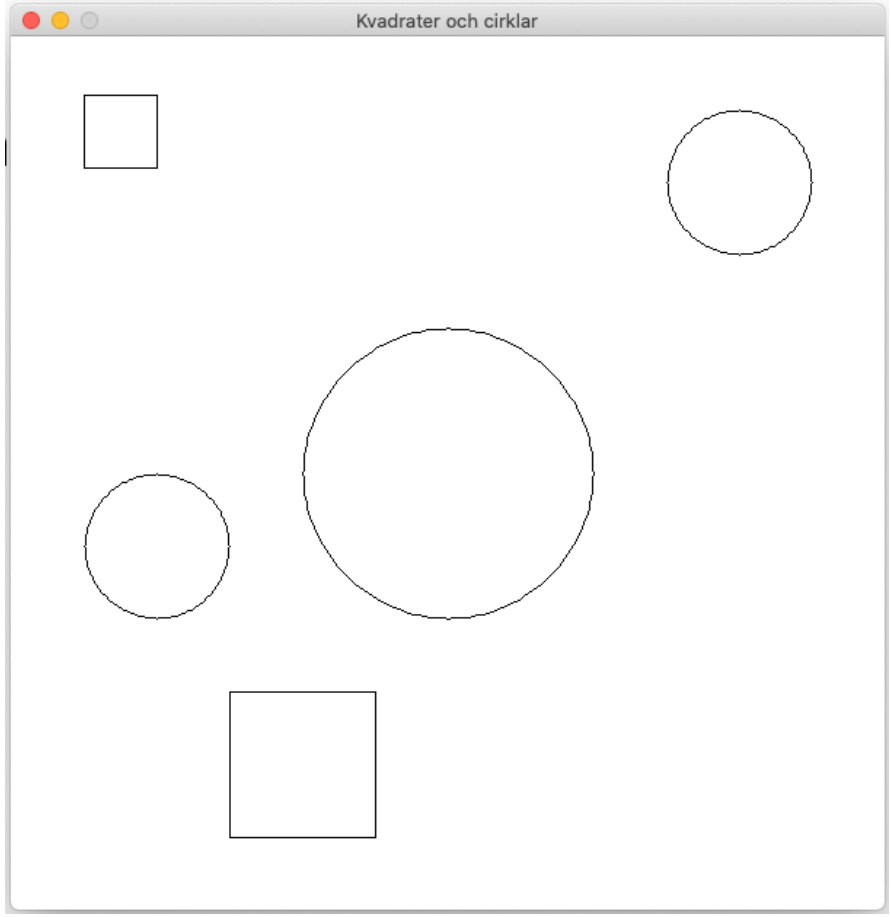
Ny subklass `RaceTurtle` som låter sköldpaddor tävla mot varandra

Metoden `raceStep` tar ett slumpmässigt steg

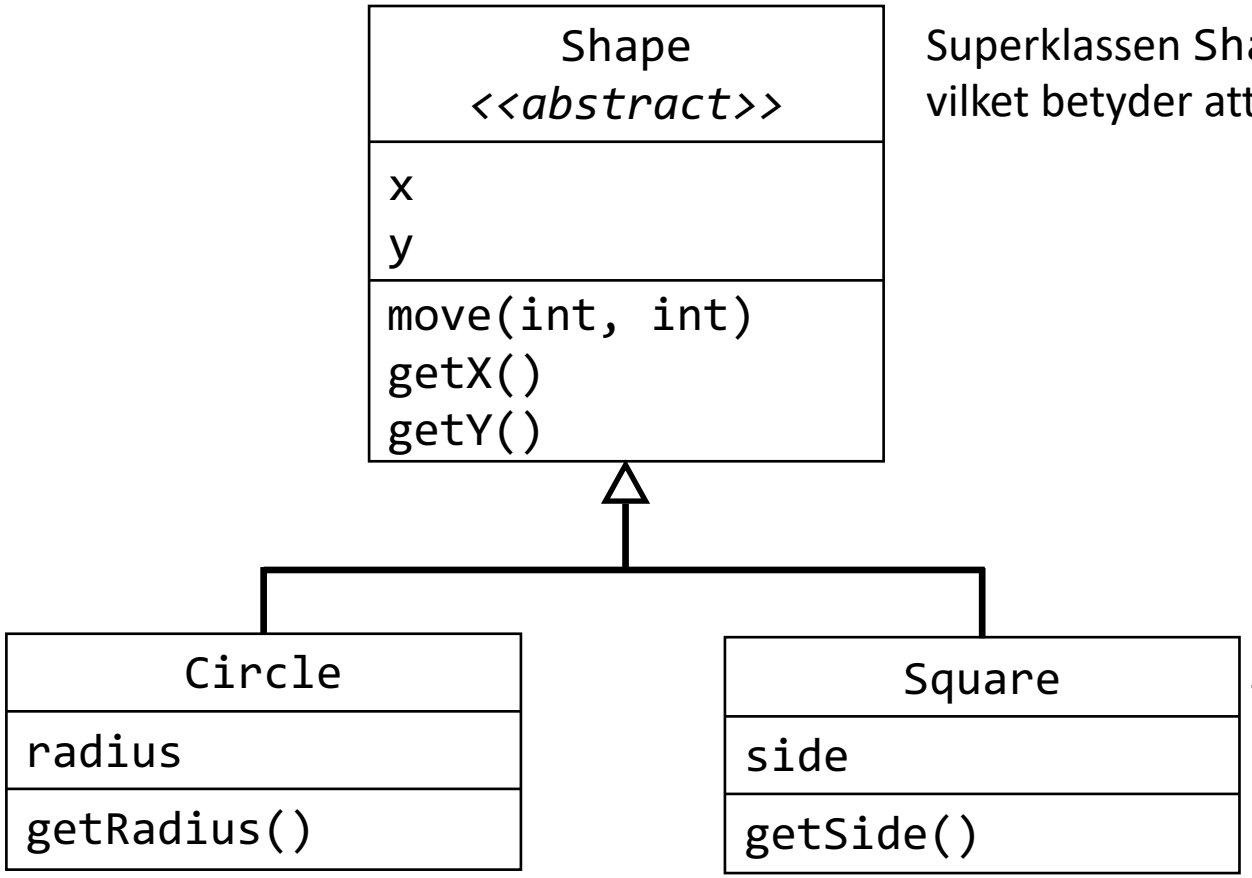
Geometriska figurer

Vi kommer att modellera olika slags geometriska figurer med en gemensam superklass Shape:





Klassdiagram



Superklassen Shape är abstrakt (**abstract**), vilket betyder att inga objekt av klassen kan skapas.

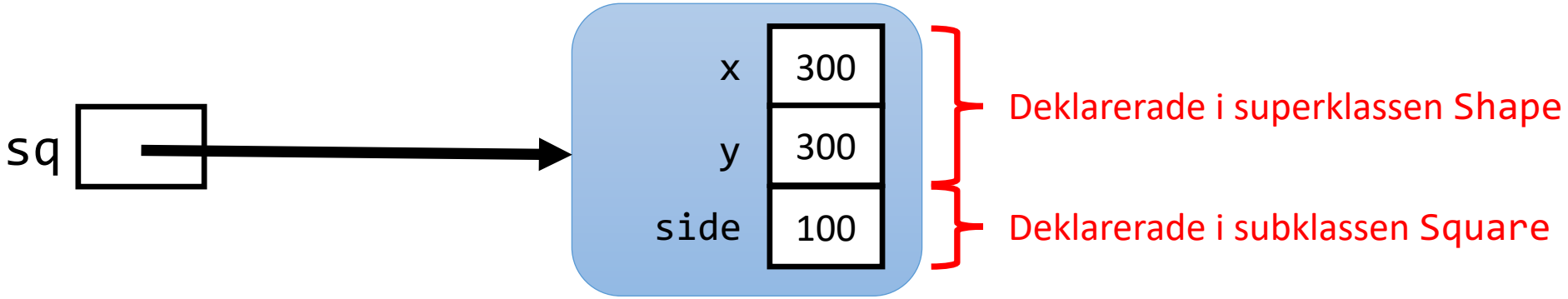
Subklassen Square har allt som Shape har, fast lite till

Specifikation av Square

```
/** Skapar en kvadrat med övre, vänstra hörnet i x,y och med
  * sidlängden side. */
Square(int x, int y, int side);

/** Tar reda på kvadratens sida. */
int getSide();

/** Flyttar figuren avståndet dx i x-led, dy i y-led. Deklarerad i Shape. */
void move(int dx, int dy);

/** Tar reda på x-koordinaten. Deklarerad i Shape. */
int getX();

/** Tar reda på y-koordinaten. Deklarerad i Shape. */
int getY();
```

Objekt får attribut från superklass

```
Square sq = new Square(300, 300, 100);
```



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

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

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

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

Klassen Shape är abstrakt (**abstract**), vilket betyder att inga objekt av klassen kan skapas. Därför ger följande kompileringsfel:

```java
new Shape(10, 20)
```

Synligheten **protected** betyder att attributen är åtkomliga i klassen och dess subklasser (Square osv).

Subklassen Square

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

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

  public int getSide() {
    return side;
  }
}
```

I Square behöver man ange värden för attributen x,y (deklarerade i Shape). Detta görs genom att anropa superklassens (Shape) konstruktor.

Superklassens konstruktorn måste anropas först i konstruktorn med: `super(argument)`

Subklassen `Circle`

```
public class Circle extends Shape {
  private int radius;

  public Circle(int x, int y, int radius) {
    super(x, y); // anropa superklassens konstruktor
    this.radius = radius;
  }

  public int getRadius() {
    return radius;
  }
}
```

En lista av figurer

```
public class FiguresProgram {
  public static void main(String args[]) {
    ArrayList<Shape> shapes = new ArrayList<Shape>();
    shapes.add(new Square(150, 450, 100));
    shapes.add(new Square(50, 40, 50));
    shapes.add(new Circle(300, 300, 100));
    shapes.add(new Circle(100, 350, 50));
    shapes.add(new Circle(500, 100, 50));

    for (Shape s: shapes) {
      s.move(10, 10);
    }
  }
}
```

En lista av olika sorters figurer som kan behandlas tillsammans!

Alla figurer flyttas, oavsett om de är cirklar eller kvadrater.

Abstrakta metoder och överskuggning

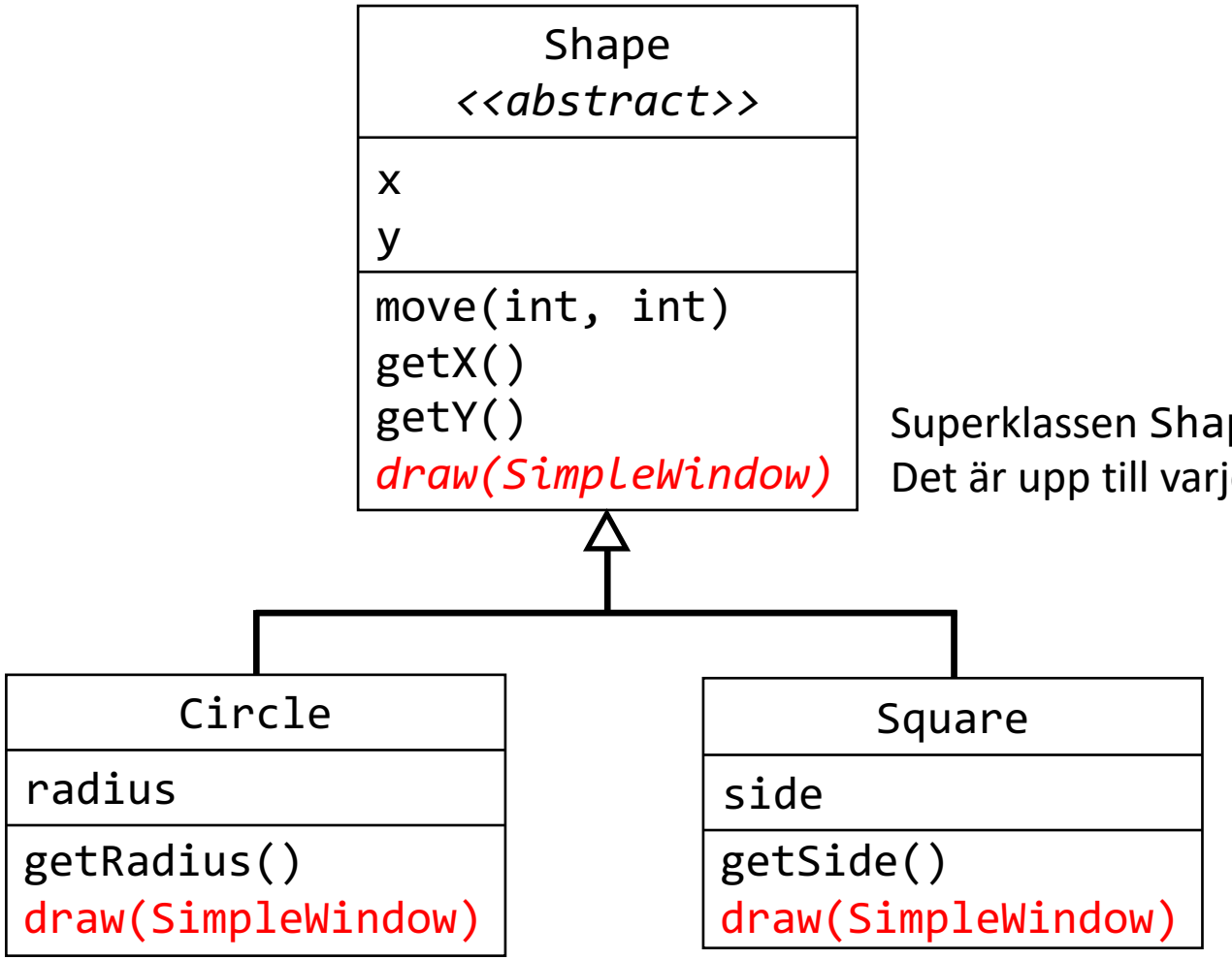
För att kunna rita alla figurer:

```
for (Shape s: shapes) {
    s.draw(w);
}
```

behöver klassen Shape metoden `draw`.

Dock vet inte klassen Shape hur en figur ska ritas, utan det vet bara subklasserna. Således måste klassen Shape ha en ***abstrakt metod*** `draw`, dvs, en metod utan programkod. Då är det upp till varje subklass att implementera (***överskugga***) metoden `draw`.

Klassdiagram



Superklassen Shape har den ***abstrakta*** metoden draw (utan kod). Det är upp till varje subklass att bestämma hur ritningen sker.

Subklassen Square implementerar (***överskuggar***) metoden draw som ritar en kvadrat.

Abstrakt metod

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

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

  ...

  public abstract void draw(SimpleWindow w);
}
```

Klassen Shape har nu en abstrakt metod draw utan programkod. Detta tvingar subklasserna att implementera metoden, annars blir det kompileringsfel.

En klass som har en abstrakt metod måste också vara abstrakt.

Implementera metoden i subklassen `Square`

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

  ...

  public void draw(SimpleWindow w) {
    w.moveTo(x, y) ;
    w.lineTo(x, y + side);
    w.lineTo(x + side, y + side);
    w.lineTo(x + side, y);
    w.lineTo(x, y);
  }
}
```

I `Square` kommer man åt attributen `x`, `y` (deklarerade i `Shape`) eftersom de har synligheten `protected`.

Dynamisk bindning

Nu kan man iterera över listan med figurer och rita dem:

```
for (Shape s: shapes) {
    s.draw(w);
}
```

Vilken metod anropas vid `s.draw(w)`? Det beror på vilken sorts objekt som referensvariabeln `s` refererar till!

Om `s` exempelvis refererar till en kvadrat då anropas `draw`-metoden i `Square`-klassen, osv. Detta kallas ***dynamisk bindning***, och innebär att vilken metod som anropas beror på vilken klass objektet är av.

Implementation av metoden `toString`:

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

  public String toString() {
    return "x=" + x + ", y=" + y;
  }
}
```

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

  public String toString() {
    String s = "Kvadrat side=" + side;
    s += ", " + super.toString();
    return s;
  }
}
```

Exempel på anrop av `toString`:

```
Square sq = new Square(150, 450, 100);
System.out.println(sq.toString());
```

Vilket skriver ut:

```
Kvadrat side=100, x=150, y=450
```

I en subklass kan man anropa metoder i superklassen med `super` framför anropet. Prefixet `super` måste användas om subklassen implementerar samma metod som superklassen (exemplet `toString` illustrerar detta).

Superklassens `toString`-metod anropas

När passar arv?

Arv passar bra vid en **"*är en*"-relation** mellan klasser

- Kvadrat *är en* geometrisk figur
- Cirkel *är en* geometrisk figur

Komposition (attribut) passar bra när det är en **"*har en*"-relation** mellan klasser. Exempel: en kurs *har* studenter.

Institutionen ger en kurs i *Objektorienterad modellering och design* där man lär sig om hur ett program delas in i klasser.

Referensvariabler och arv

En referensvariabel som deklareras med typen C
får referera till objekt av klassen C och subklasser till C.

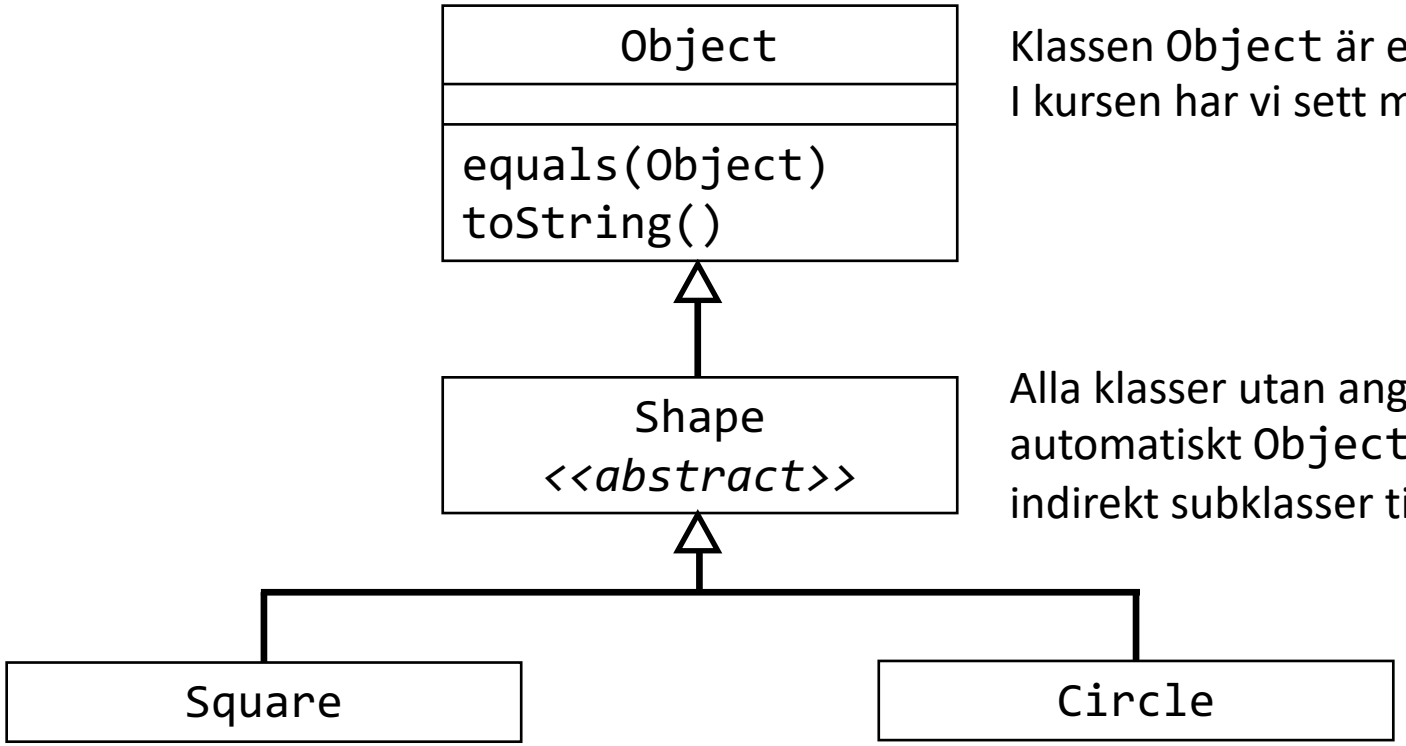
```
Shape shape;
Circle circle;

circle = new Circle(100, 100, 50);   // OK
shape  = new Circle(100, 100, 50);   // OK
circle = new Square(100, 100, 50);   // Kompileringsfel
shape  = new Square(100, 100, 50);   // OK
```

Referensvariabler och metodanrop

Med en referensvariabel som deklareras med typen C
kan man enbart anropa metoder som finns deklarerade i C.

```
Circle circle = new Circle(100, 100, 50);
Shape shape   = new Circle(100, 100, 50);

circle.move(10, 10);            // OK
shape.move(10, 10);             // OK
int side = circle.getRadius();  // OK
int side = shape.getRadius();   // Kompileringsfel. Klassen Shape
                                // har ej metoden getRadius()
```

Klassen `Object`



Klassen `Object` är en standardklass i Java med diverse metoder. I kursen har vi sett metoderna `equals` och `toString`

Alla klasser utan angiven superklass (exempelvis `Shape`) får automatiskt `Object` som superklass. De andra klasserna blir indirekt subklasser till `Object`.

Metoden `equals`

- Metoden `equals` används för att kontrollera om två objekt är likadana:

```
/** Indicates whether some other object is "equal to"
  * this one. */
public boolean equals(Object obj)
```

- Standardbeteendet är att referenserna jämförs (==)
- Man kan överskugga metoden för att jämföra innehållet och då skriva:

```
Point p1 = new Point(10, 30);
Point p2 = new Point(10, 30);
if (p1.equals(p2)) {
  ...
}
```

Metoden `toString`

- Metoden `toString` anropas automatiskt på ett objekt om en sträng behövs. Exempel:

```
Point p = new Point(10, 30);
System.out.println(p);   // toString anropas automatiskt på p
```

- Standardbeteendet är att klassnamnet och ett hexadecimalt tal skrivs ut.
- Man kan överskugga metoden, exempelvis:

```
public class Point {
  ...
  public String toString() {
    return "Punkt " + x + ", " + y;
  }
}
```

Uppgift om listor

Vi ska implementera klassen `Train` som beskriver ett tåg av vagnar, där man kan boka platser

En vagn representeras med klassen `RailwayCoach`

I Moodle under dagens föreläsning finns klasserna `RailwayCoach`, `Train`, `Booking` (huvudprogrammet). Lägg in filerna i Eclipse

Den ***färdigimplementerade*** klassen `RailwayCoach` beskriver en järnvägsvagn. Klassen har följande specifikation:

```
/** Skapar en vagn med nbrSeats platser.
    Från början är alla platser lediga. */
RailwayCoach(int nbrSeats);

/** Tar reda på antalet platser i vagnen. */
int getNbrSeats();

/** Tar reda på antal lediga platser i vagnen. */
int getNbrFreeSeats();

/** Bokar n platser i vagnen.
    Förutsätter att n är <= antalet lediga platser. */
void makeReservation(int n);
```

Vi ska *använda* klassen `RailwayCoach` för att **implementera klassen `Train`**, som beskriver ett tåg av vagnar enligt följande specifikation:

```
/** Skapar ett tåg utan vagnar. */
Train();

/** Lägger till vagnen c sist i tåget. */
void addCoach(RailwayCoach c);

/** Bokar n platser på tåget. Alla platserna ska bokas i
    samma vagn. Returnerar vagnens position i tåget om platsbokning
    har skett (om det fanns en vagn med n lediga platser), annars -1.
    Första vagnen har position 0, osv.*/
int makeReservations(int n);

/** Returnerar en sträng som representerar tåget.
    Strängen ska innehålla en rad för varje vagn på formen:
    antal lediga platser/antal platser */
String toString();
```

Exempel på användning

```
public class Booking {
  public static void main(String[] args) {
    Train t = new Train();
    for (int i = 0; i < 5; i++) {
      t.addCoach(new RailwayCoach(20));
    }
    t.makeReservations(10);
    t.makeReservations(7);
    t.makeReservations(8);
    t.makeReservations(12);
    t.makeReservations(5);
    System.out.println(t.toString());
  }
}
```

Vi ska implementera metoderna `makeReservations` och `toString` i klassen `Train`

Programmet ovan ska skriva ut:

```
3/20
0/20
15/20
20/20
20/20
```

(3 lediga platser av 20 i första vagnen, osv)

```java
public class Train {
  private ArrayList<RailwayCoach> coachList;

  /** Skapar ett tåg utan vagnar. */
  public Train() {
    coachList = new ArrayList<RailwayCoach>();
  }

  /** Lägger till vagnen c sist i tåget. */
  public void addCoach(RailwayCoach c) {
    coachList.add(c);
  }





}
```

Specifikation av RailwayCoach:

```java
RailwayCoach(int nbrSeats);
int getNbrSeats();
int getNbrFreeSeats();
void makeReservation(int n);
```

```java
public class Train {
  private ArrayList<RailwayCoach> coachList;

  /** Skapar ett tåg utan vagnar. */
  public Train() {
    coachList = new ArrayList<RailwayCoach>();
  }

  /** Lägger till vagnen c sist i tåget. */
  public void addCoach(RailwayCoach c) {
    coachList.add(c);
  }

  /** Bokar n platser på tåget. Alla platserna ska bokas i samma vagn.
      Returnerar vagnens position i tåget om platsbokning har skett
      (om det fanns en vagn med n lediga platser), annars -1. */
  public int makeReservations(int n) {
    // ÖVNING 1
    // Implementera metoden. (Tips: Använd vanlig for-sats med index)
  }
}
```

Specifikation av RailwayCoach:

```java
RailwayCoach(int nbrSeats);
int getNbrSeats();
int getNbrFreeSeats();
void makeReservation(int n);
```

```java
public class Train {
  private ArrayList<RailwayCoach> coachList;

  /** Skapar ett tåg utan vagnar. */
  public Train() {
    coachList = new ArrayList<RailwayCoach>();
  }

  /** Lägger till vagnen c sist i tåget. */
  public void addCoach(RailwayCoach c) {
    coachList.add(c);
  }

  /** Returnerar en sträng som representerar tåget.
      Strängen ska innehålla en rad för varje vagn på formen:
      antal lediga platser/antal platser */
  public String toString() {
    // ÖVNING 2
    // Implementera metoden. Använd gärna StringBuilder. Ny rad: "\n"
  }
}
```

Specifikation av RailwayCoach:

```java
RailwayCoach(int nbrSeats);
int getNbrSeats();
int getNbrFreeSeats();
void makeReservation(int n);
```

Lösningsförslag

```
public int makeReservations(int n) {
  for (int i = 0; i < coachList.size(); i++) {
    if (coachList.get(i).getNbrFreeSeats() >= n) {
      coachList.get(i).makeReservation(n);
      return i;
    }
  }
  return -1;
}

public String toString() {
  StringBuilder sb = new StringBuilder();
  for (RailwayCoach c: coachList) {
    sb.append(c.getNbrFreeSeats());
    sb.append("/");
    sb.append(c.getNbrSeats());
    sb.append("\n");
  }
  return sb.toString();
}
```

Checklista

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