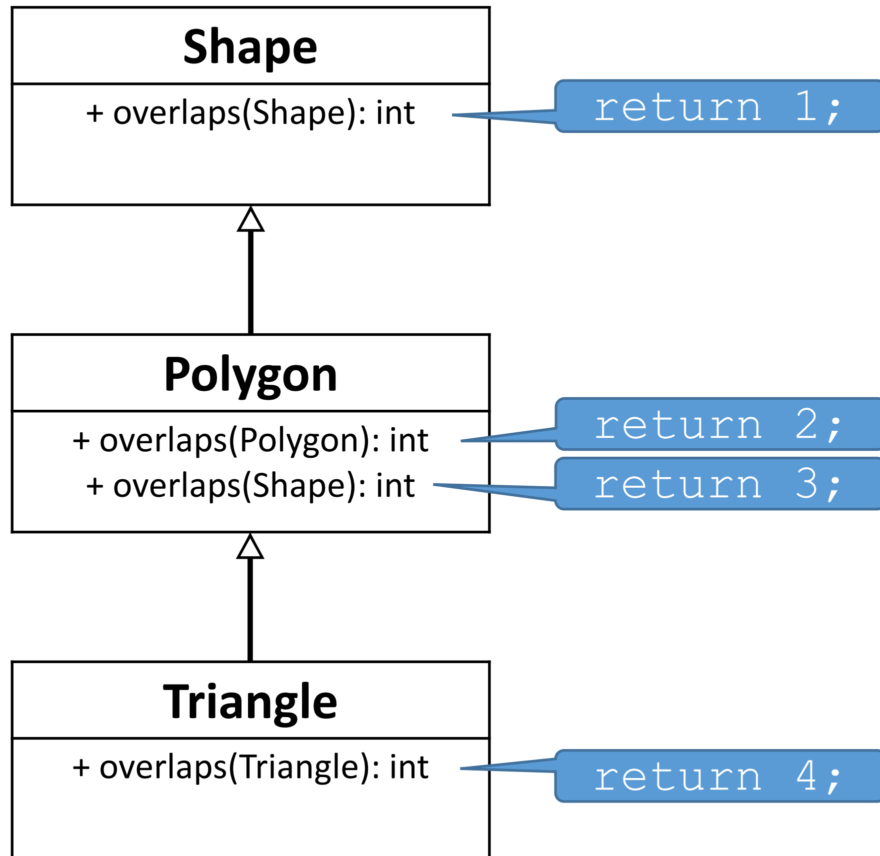
Static vs Dynamic binding
Polymorfism

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Diagnostiskt prov




```
Shape    shape    = new Shape();
Polygon  tripoly  = new Triangle();
Shape    trishape = new Triangle();
Triangle triangle = new Triangle();

shape.overlaps(shape);
shape.overlaps(triangle);

tripoly.overlaps(shape);
tripoly.overlaps(trishape);
tripoly.overlaps(tripoly);
tripoly.overlaps(triangle);

trishape.overlaps(shape);
trishape.overlaps(trishape);
trishape.overlaps(tripoly);

triangle.overlaps(shape);
triangle.overlaps(trishape);
triangle.overlaps(triangle);
```

Type checking

- Java är ett mestadels *starkt typat språk*. Kortfattat innebär detta att om kod kompilerar korrekt, så kommer den också att köra utan missöden.
  - Om en metod anropas via en variabel, och detta anrop bedöms giltigt av javas *type checker* (typkontrollen), då kommer objektet som referensen i variabeln pekar på när programmet körs garanterat ha den metoden definierad.
  - Mestadels – det finns undantag (som fångas vid run time).
  - (Det finns ingen formell definition av vad ”starkt typat” innebär.)

Static type vs Dynamic type

- En referens-variabel har en *declared type* – detta är dess *statiska* typ, och den kommer alltid att vara densamma (därav ordet statisk).

- En referens-variabel refererar till ett objekt i heapen. Det objektets typ är variabelns *dynamiska* typ. Denna kan förändras (därav ordet dynamisk), om variabeln ges ett nytt referensvärde som pekar på ett annat objekt med en annan typ.
  - Notera: *Objektets* typ förändras inte efter att det har skapats. *Referensen* förändras inte, den kommer alltid att peka på samma objekt. *Variabeln* (som namnet antyder) är det som kan ändras, genom att ges ett nytt referens-värde.

Compile time vs run time

- Typinformation används vid två olika tillfällen:
  - Kompilatorn utför *statisk typkontroll (static type checking*). Denna information används enbart vid kompileringen, för att garantera att allt kommer att gå rätt när programmet körs, och för att välja rätt metod(signatur). Informationen om statisk typ sparas inte efter kompileringen.
  - Under exekvering görs *dynamisk typkontroll (dynamic type checking),* och den dynamiska typen används för att bestämma vilka metoder som faktiskt körs.
    - Tack vare att statisk type checking redan gjorts vet vi att de metoder vi försöker anropa alltid finns hos objekten i fråga.

Overloading

- Overloading innebär att ett objekt (eller class) har flera metoder med samma namn, men olika signatur (dvs typer på dess parametrar).

```
String.subString(int beginIndex);
String.subString(int beginIndex, int endIndex);
```

- Vilken av signaturerna som används bestäms *statiskt*, vid kompilering.

- De två metoderna ovan är två helt olika metoder, vad Java anbelangar. De råkar bara ha samma namn. Vilken av dem som avses med ett specifikt anrop är vad som bestäms statiskt.

Overriding

- Overriding innebär att en subklass kan definiera en metod med samma namn och signatur som en metod i dess superklass.

```
Polygon.paint(Graphics g);
Square.paint(Graphics g);
```

- De två anropen ovan är till *samma* metod – som kan förekomma i flera olika implementationer. Vilken av *implementationerna* som används bestäms *dynamiskt*, vid exekvering.
  - Vi säger att metoden är *polymorf*.

Polymorfism

- Polymorfism är namnet på det fenomen som tillåter ett objekt (metod, funktion, …) att uppträda som om det hade flera olika typer *statiskt*.
  - ”Poly” = många, ”morf” = form

- Ett objekt skapat från class T kan uppträda som T, eller som vilken som helst av T’s supertyper – inklusive interface-typer som implementeras av T.

Polymorfism omvänt

- Ett objekt kan uppträda som flera olika statiska typer. Omvänt innebär det också att en variabel (eller metod-parameter) kan vid olika tidpunkter hålla referenser som pekar på objekt av olika faktisk typ. Variablerna har då olika *dynamisk* typ vid de olika tidpunkterna.

- En variabel med statisk (referens-)typ S kan hålla referenser som pekar på objekt av typ S (om sådana existerar), eller vilken (konkret) subtyp som helst till S.
  - Om S är en abstract class eller interface *måste* referensen peka på ett objekt av en subtyp.

- Mer om sub- och super-typer och deras användning senare.

Dynamisk binding

- Dynamic binding (även *dynamic dispatch*) innebär generellt att ett val mellan flera möjliga implementationer av en polymorf metod avgörs vid run time.

- Utan dynamic binding skulle inte polymorfism för objekt fungera!

```
for (Polygon p : polygons) {
  p.paint(g);
}
```

- Variabeln `p` har statisk typ `Polygon`. Om metodval gjordes baserat på statisk typ, då skulle koden ovan alltid anropa `paint`-metoden definierad i `Polygon`!

En värld utan dynamic binding…

- För att kunna ge de olika subklasserna olika beteende vid utritning skulle vi i princip behöva kolla den dynamiska typen på varje polygon:

```
for (Polygon p : polygons) {
  if (p instanceof Square) {
    ((Square) p).paint(g);
  } else if (p instanceof Square) {...}
}
```

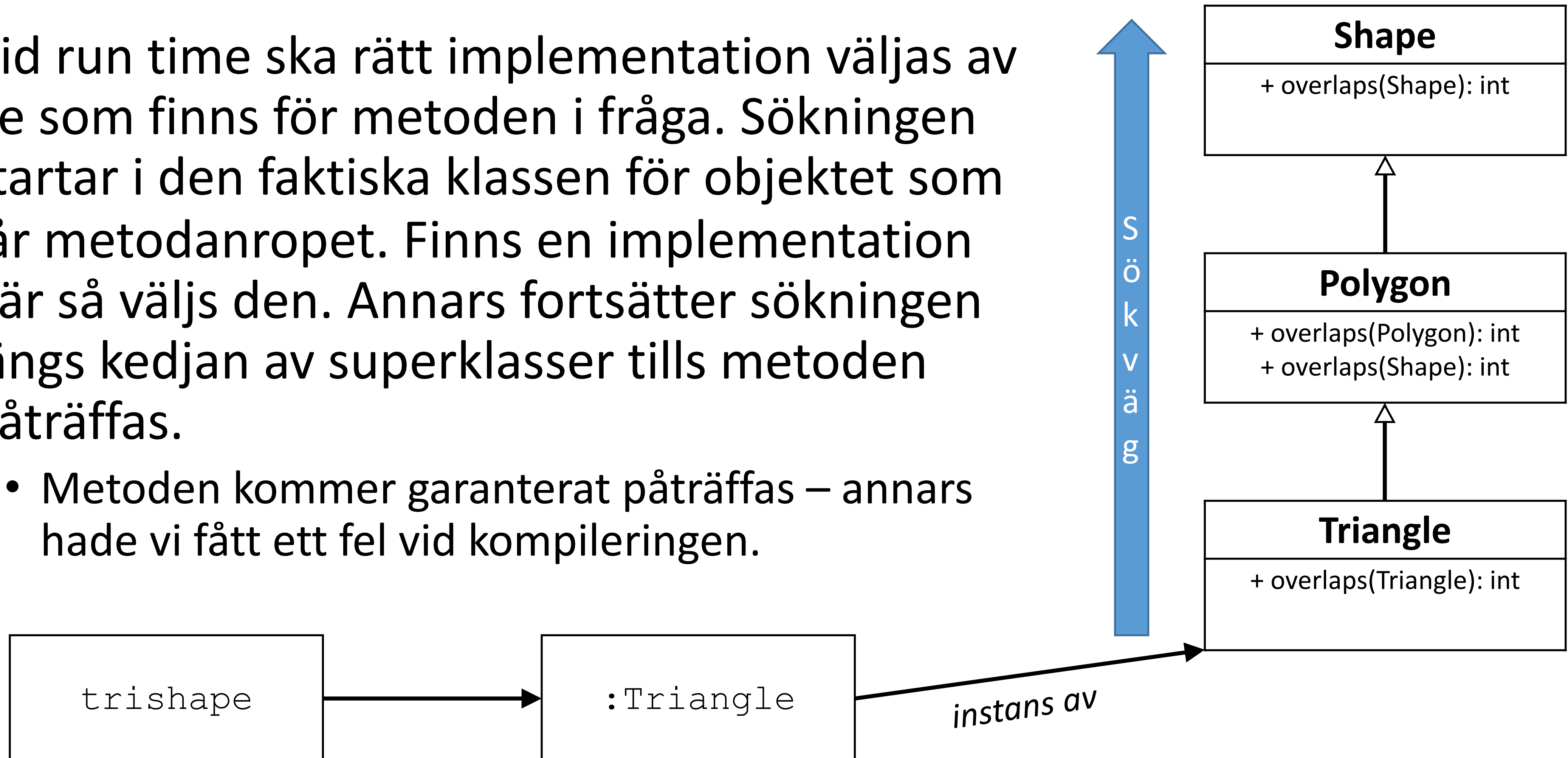
- Denna kod är definitivt *inte* extensible (eller vacker) – vi måste känna till alla olika sorters polygoner från början, vi kan inte lägga till fler i efterhand utan att ändra koden.

Resolving overloading

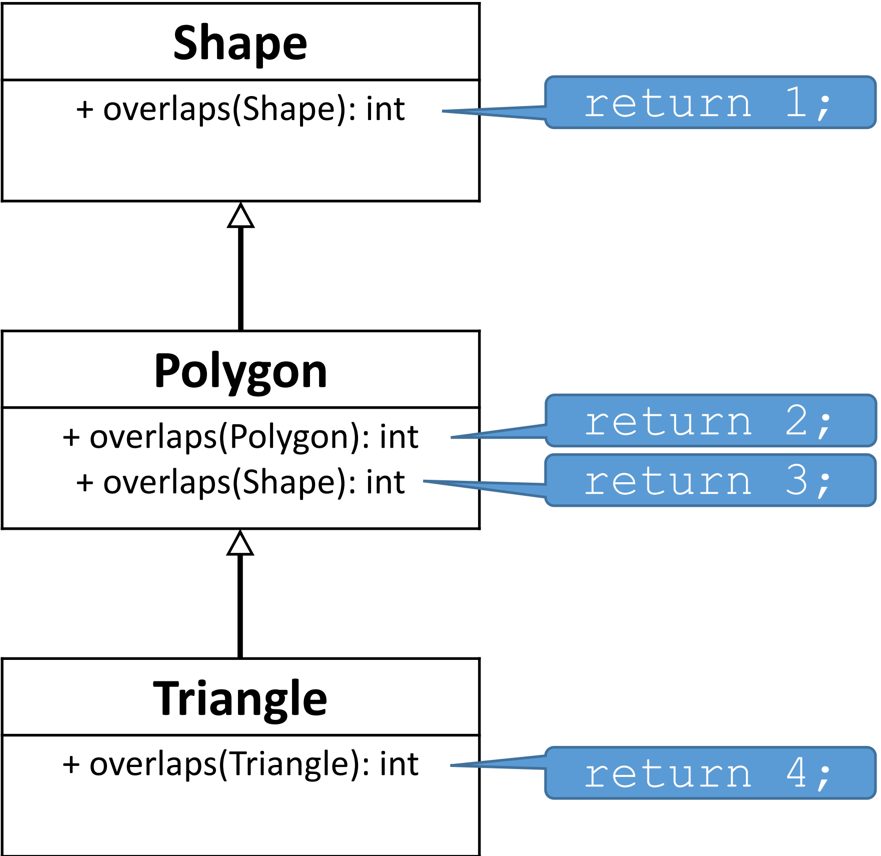
- Vid anrop till metoden tripoly.overlaps(triangle) händer följande statiskt:
    1. Kompilatorn tar reda på vilken *deklarerad* typ `tripoly` har.
    2. Kompilatorn tittar i den aktuella klassen (eller interfacet), och dess superklasser (superinterface) efter alla metoder med namn `overlaps`.
    3. Kompilatorn tittar på parameterlistorna för de olika alternativen, och väljer den som ”bäst” överensstämmer med den *deklarerade* typen för argumenten till anropet.
- I vårt fall gäller följande:
    - `tripoly` har deklarerad typ `Polygon`.
    - `Polygon` har två olika metoder med namn `overlaps`: `overlaps(Polygon)` och `overlaps(Shape)`.
    - Argumentet `triangle` har deklarerad typ `Triangle`. Den parameterlista som stämmer ”bäst” är `overlaps(Polygon)`, eftersom `Polygon` är en närmare superklass till `Triangle` än vad `Shape` är.
- Mer om overloading och vad ”bäst” innebär när vi går in djupare på subtyper.

Resolving overriding

- Vid run time ska rätt implementation väljas av de som finns för metoden i fråga. Sökningen startar i den faktiska klassen för objektet som får metodanropet. Finns en implementation där så väljs den. Annars fortsätter sökningen längs kedjan av superklasser tills metoden påträffas.
  - Metoden kommer garanterat påträffas – annars hade vi fått ett fel vid kompileringen.



Diagnostiskt prov - revisited




```
Shape    shape    = new Shape();
Polygon  tripoly  = new Triangle();
Shape    trishape = new Triangle();
Triangle triangle = new Triangle();

shape.overlaps(shape);                1
shape.overlaps(triangle);             1

tripoly.overlaps(shape);              3
tripoly.overlaps(trishape);           3
tripoly.overlaps(tripoly);            2
tripoly.overlaps(triangle);           2

trishape.overlaps(shape);             3
trishape.overlaps(trishape);          3
trishape.overlaps(tripoly);           3

triangle.overlaps(shape);             3
triangle.overlaps(trishape);          3
triangle.overlaps(triangle);          4
```

Quiz: ”Plug-n-play”

- Konceptet ”plug-n-play” innebär att en komponent kan direkt pluggas in i ett system, och systemet ska fungera utan att förändras eller konfigureras om. Hur kan vi bäst åstadkomma ”plug-n-play” för mjukvarukomponenter (e.g. objekt)?

- Svar: Med hjälp av polymorfism, genom att låta deklarerade statiska typer för alla variabler ha så *bred* (wide) typer som möjligt – dvs vara supertyper, inte subtyper.
  - ”Bred” typ = en typ som kan representera många olika sorters objekt

Exempel – Live code

- …

Dependency Inversion Principle

Depend on abstractions, not on concrete implementations.

- Genom att använda supertyper istället för subtyper kan vi *minska beroendet* av en specifik klass.
  - MYCKET mer om att minska beroenden senare i kursen.

Principer – hur hänger de ihop?

- Vårt slutgiltiga mål är extensibility, reusability, maintainability.

- The Open-Closed principle (OCP):
  - ”Software modules should be open for extension, but closed for modification.”
  - En konkret formulering av målet ovan, som ger oss ett sätt att tänka vid alla kodutveckling.

- The Dependency Inversion principle (DIP):
  - ”Depend on abstractions, not on concrete implementations.”
  - Ett av våra viktigaste verktyg för att uppnå OCP.