Föreläsning 20

Lars-Henrik Eriksson

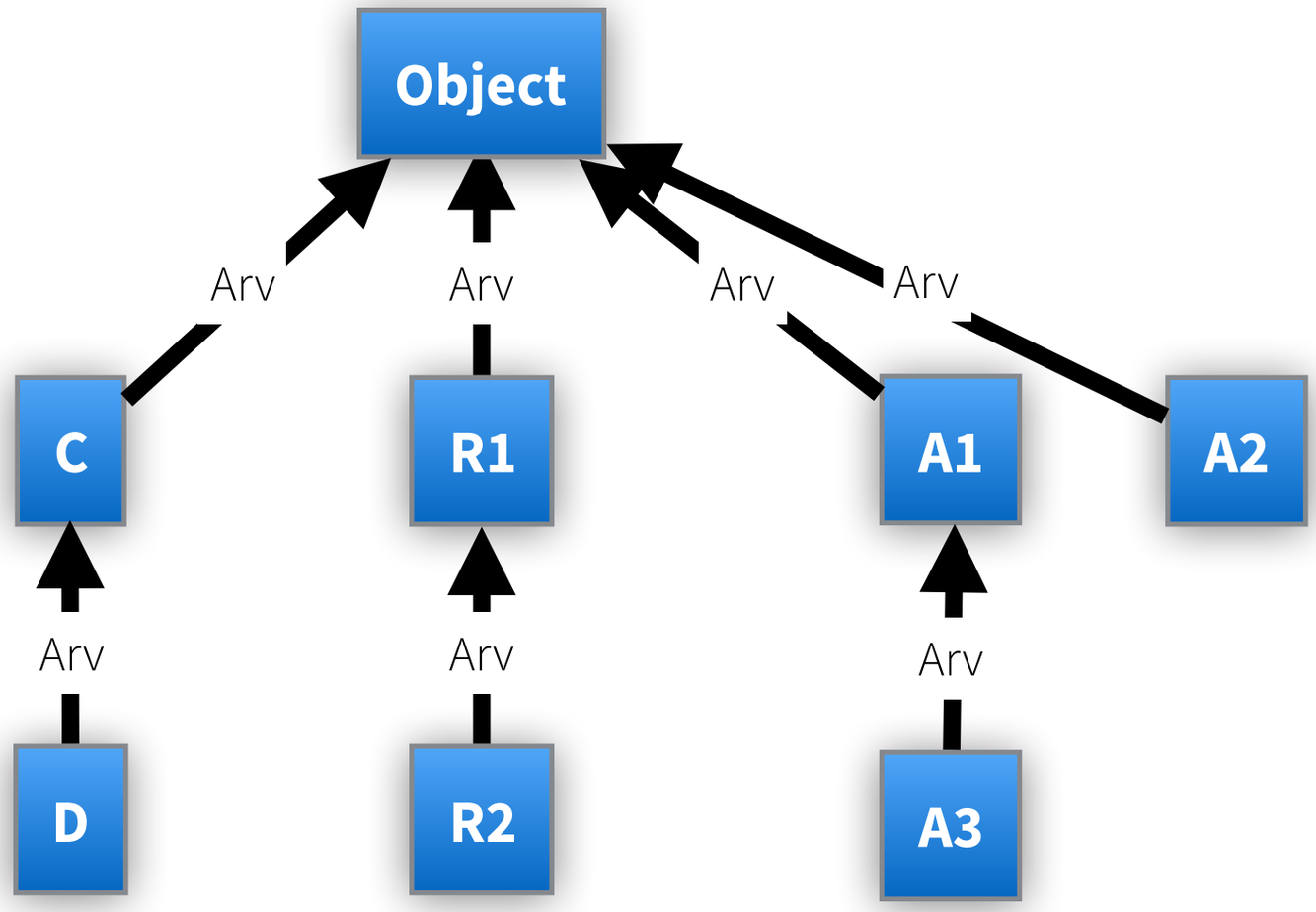
Föreläsningsbilderna är baserade på bilder gjorda av Tobias Wrigstad.

Subtyper, överlagring, overriding, Liskovs substitutionsprincip, undantagshantering

Inledningsfrågor

- Vad är en **typ**?
- Vad är en **subtyp**?
- Vad är relationen mellan **subklass** och **subtyp**?
- Hur kan man göra **överlagring** och **overriding** (i Java)?

Objekthierarki i exemplen



A_1 = Instrument

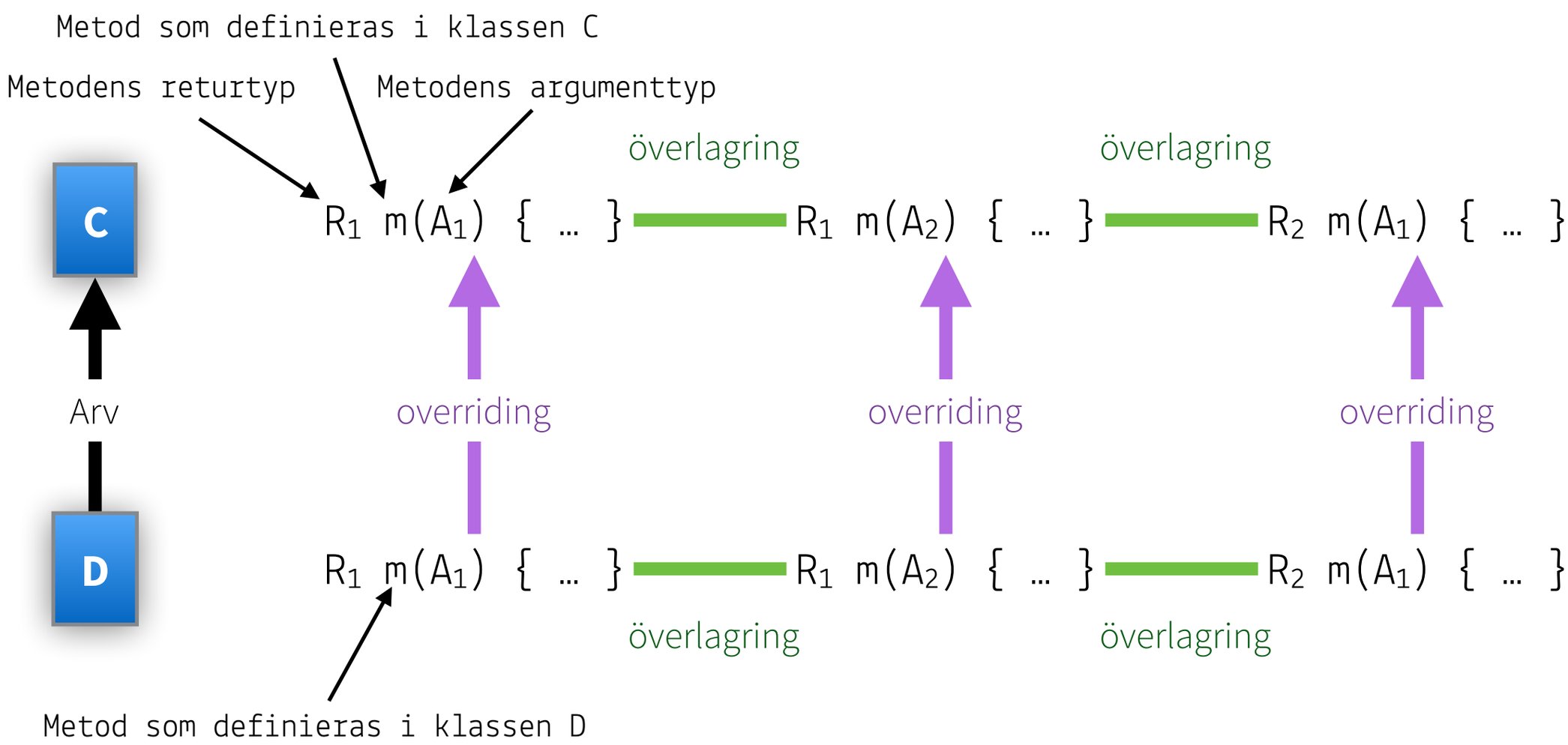
A_2 = Bostad

A_3 = Fiol

R_1 = Geometrisk form

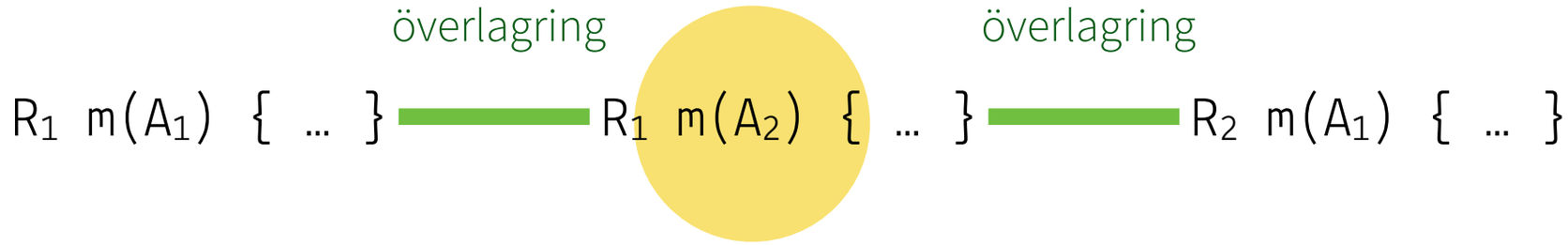
R_2 = Kvadrat

Överlagringar och overriding som vi skall studera



A_1 = Instrument R_1 = Geometrisk form

A_2 = Bostad R_2 = Kvadrat




```
C c = new C();
A2 a2 = new A2();
```

```
c.m(a2);
```

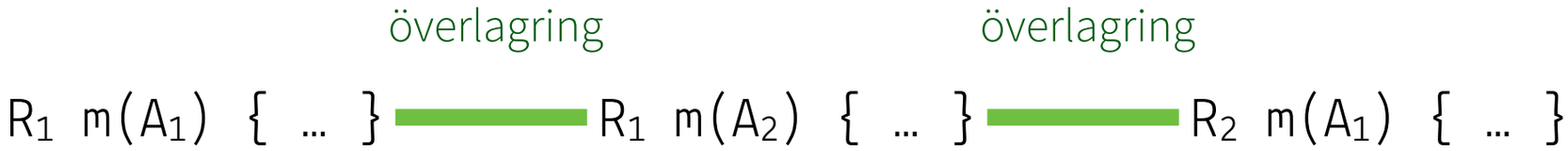
A_1 = Instrument

A_2 = Bostad

R_1 = Geometrisk form

R_2 = Kvadrat

C




```
C c = new C();
A1 a1 = new A1();
```

```
R2 r = c.m(a1);
```

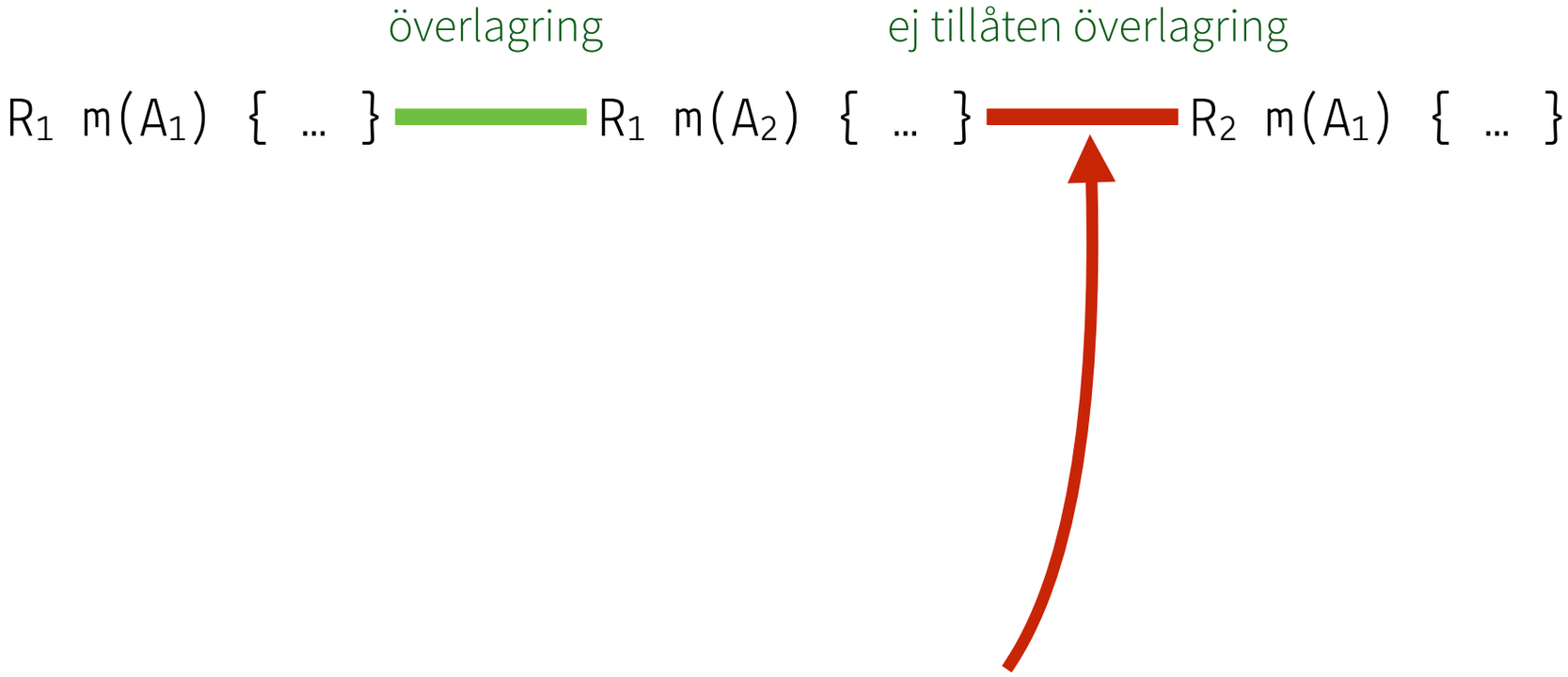
A1 = Instrument

A2 = Bostad

R1 = Geometrisk form

R2 = Kvadrat

C




```
C c = new C();
A1 a1 = new A1();
```

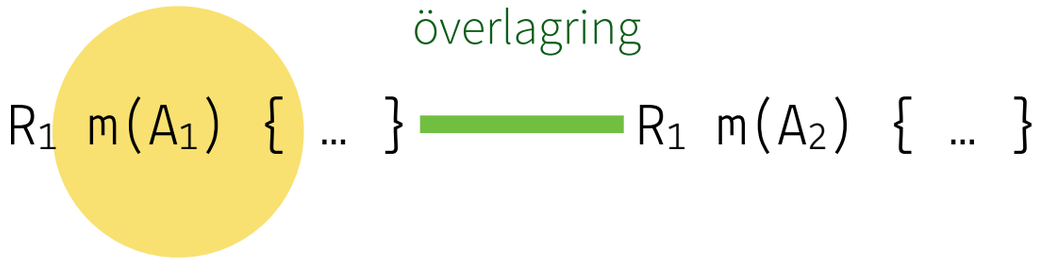
```
R2 r = c.m(a1);
```

A_1 = Instrument

A_2 = Bostad

R_1 = Geometrisk form

R_2 = Kvadrat




```
C c = new C();
A1 a1 = new A1();
R1 r1;
```

$$R_1\ r\ =\ c.m(a_1);$$

A_1 = Instrument

A_2 = Bostad

R_1 = Geometrisk form

R_2 = Kvadrat

överlagring

R_1 m(A_1) { … } —— R_1 m(A_2) { … }

```
C c = new C();
Object o = new A2();
```

```
c.m(o);
```

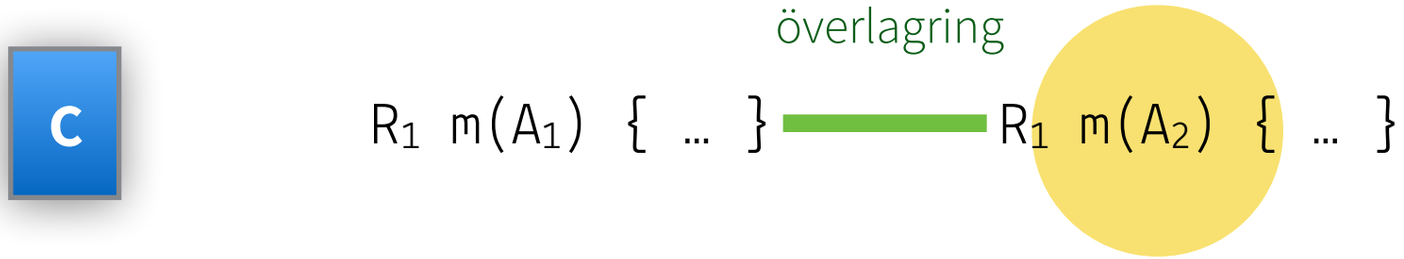
👎

A_1 = Instrument

A_2 = Bostad

R_1 = Geometrisk form

R_2 = Kvadrat




```
C c = new C();
Object o = new A2();
```

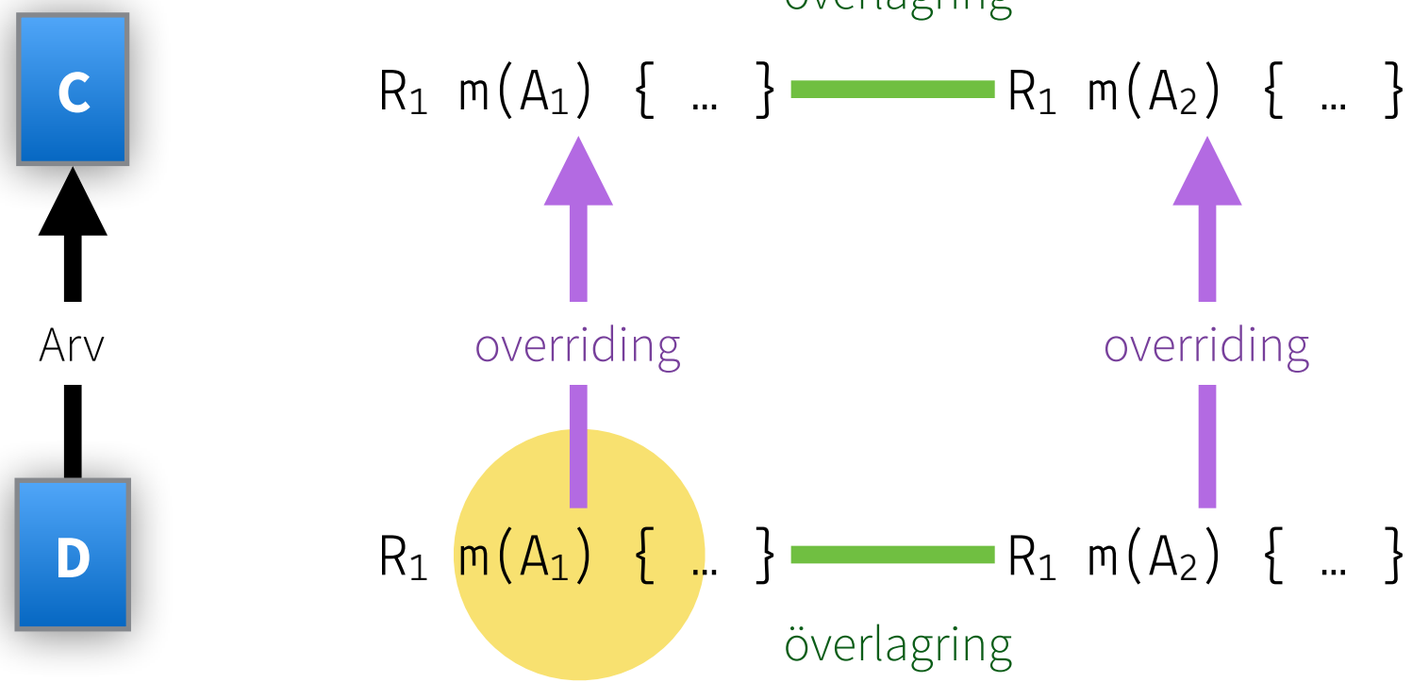
```
c.m((A2) o);
```

A_1 = Instrument

A_2 = Bostad

R_1 = Geometrisk form

R_2 = Kvadrat

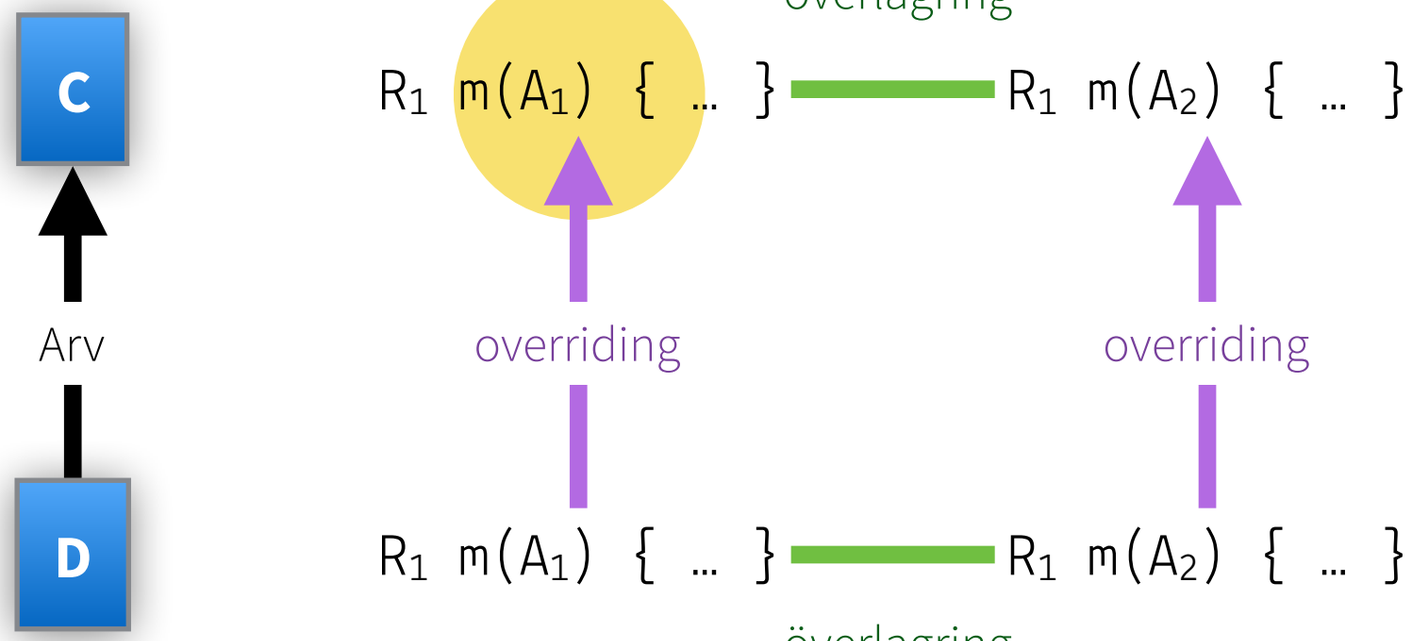



```
D d = new D();
A1 a1 = new A1();
```

```
d.m(a1);
```

A1 = Instrument R1 = Geometrisk form

A2 = Bostad R2 = Kvadrat

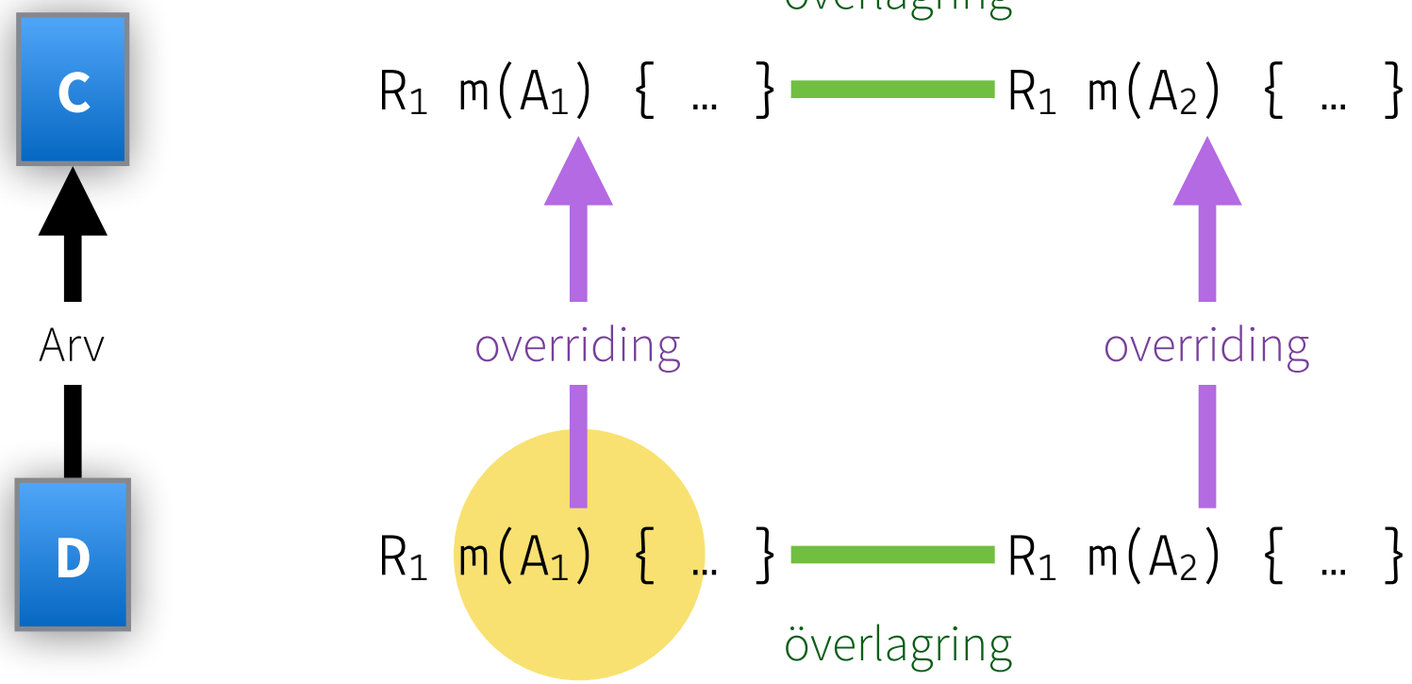



```
C c = new C();
A1 a1 = new A1();
```

```
c.m(a1);
```

A_1 = Instrument R_1 = Geometrisk form

A_2 = Bostad R_2 = Kvadrat




```
C cd = new D();
A1 a1 = new A1();
```

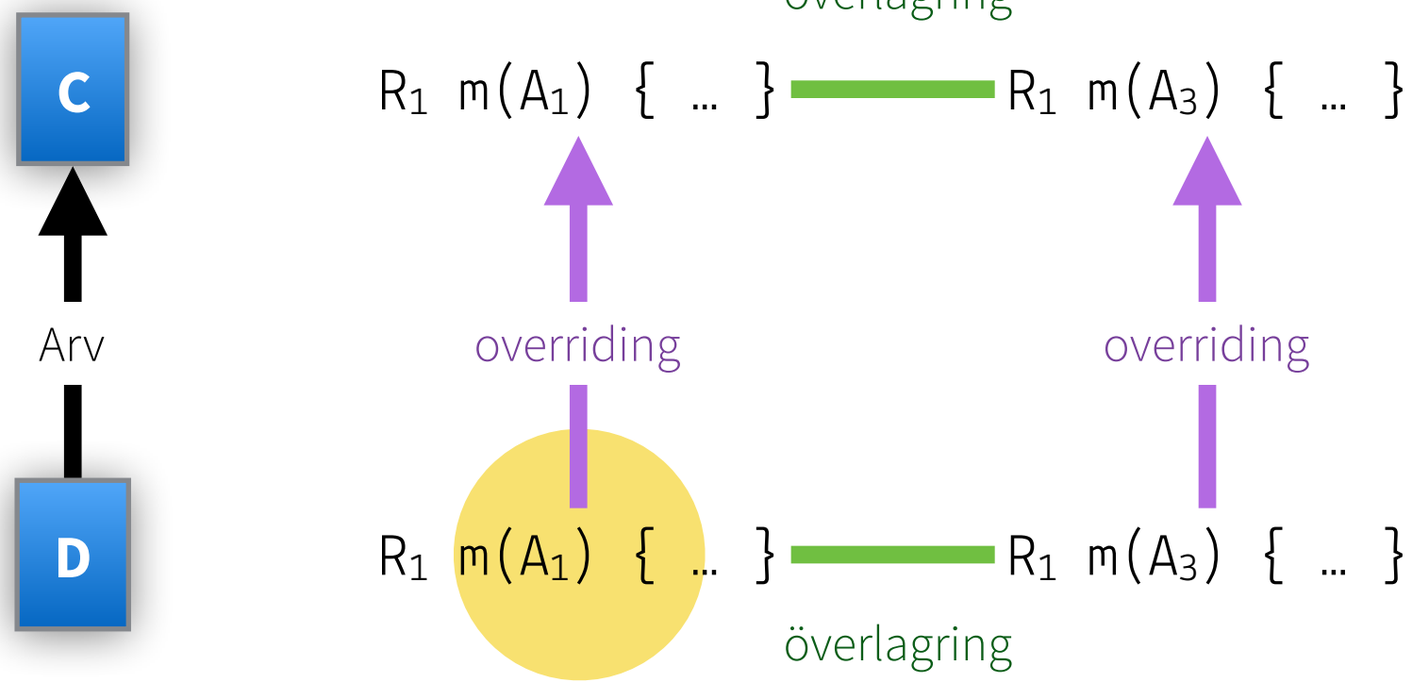
```
cd.m(a1);
```

A1 = Instrument

A2 = Bostad

R1 = Geometrisk form

R2 = Kvadrat




```
C cd = new D();
A1 a = new A3();
```

OBS!

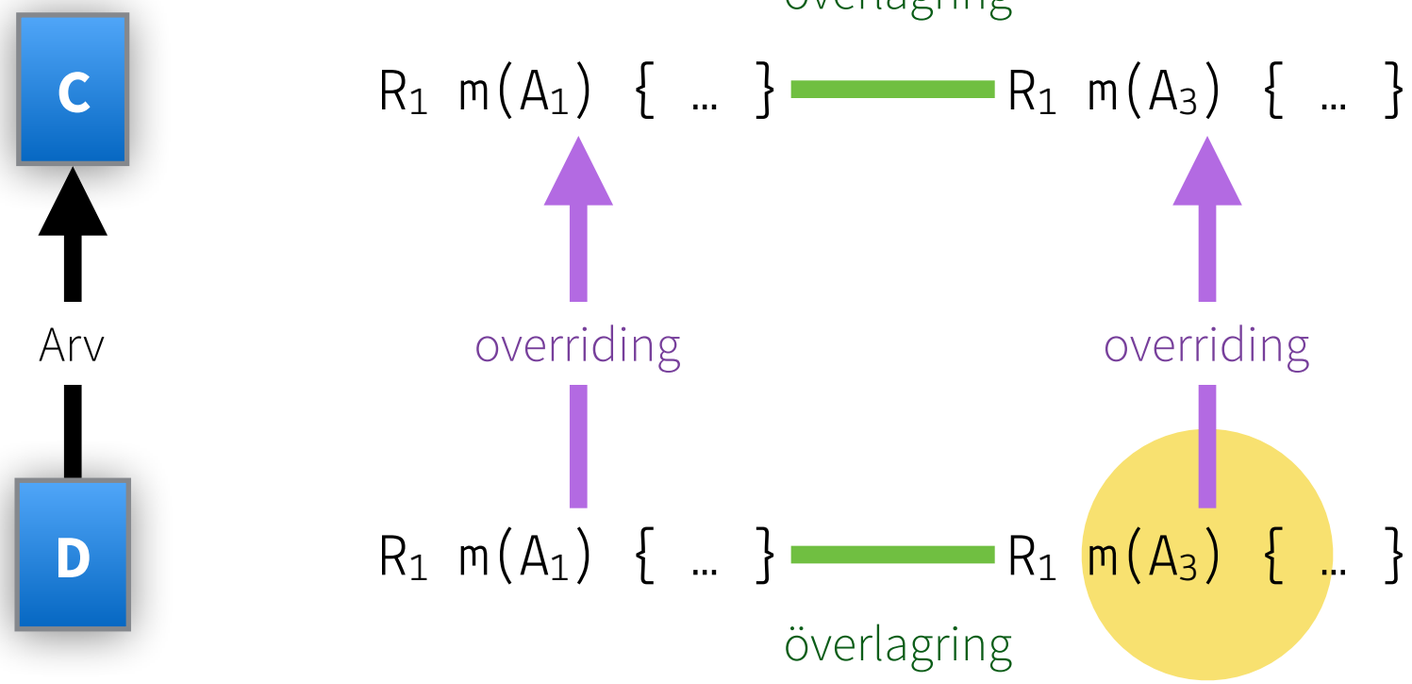
```
cd.m(a);
```

A1 = Instrument

A3 = Fiol

R1 = Geometrisk form

R2 = Kvadrat




```
C cd = new D();
A1 a = new A3();
```

OBS!

```
cd.m((A3)a);
```

A1 = Instrument

A3 = Fiol

R1 = Geometrisk form

R2 = Kvadrat

Sammanfattning

Överlagring (overloading)

Använd t.ex. antal argument och/eller argumentens typer för att välja funktion/metod/operation.
I Java tillåts överlagring av metoder baserat på antal argument och/eller argumenttyper.
Metodvalet görs *statiskt* vid kompileringen.

Overriding

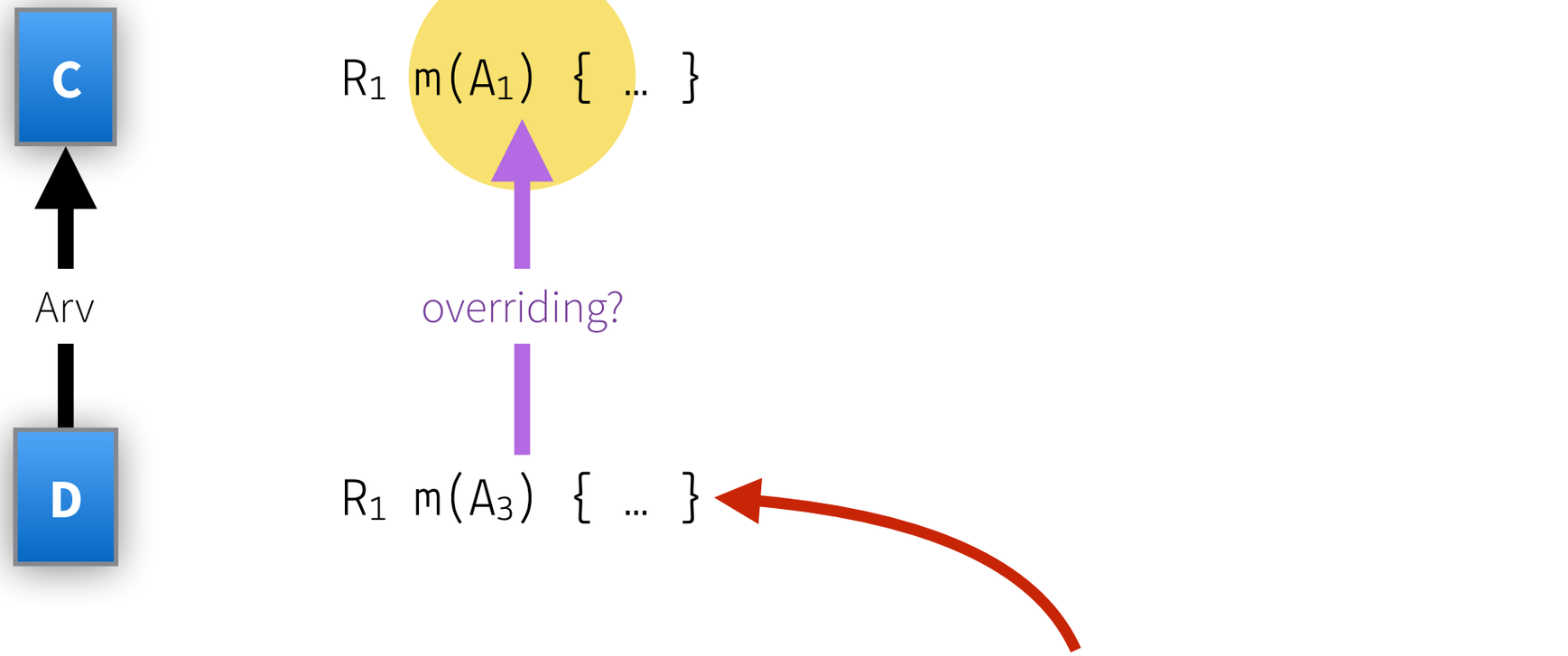
Den klass objektet tillhör avgör vilken metod ett anrop binder till.
Metodvalet görs *dynamiskt* under körningen.

Single dispatch

Dynamiskt används endast en typ för att avgöra vilken metod som körs. I Java är detta mottagarens (receiver) typ.

Multiple dispatch

Även argumenttyper m.m. kan användas för att välja metod. Förekommer men är inte vanligt (se t.ex. CLOS, Julia).




```
C cd = new D();
A1 a1 = new A1();
```

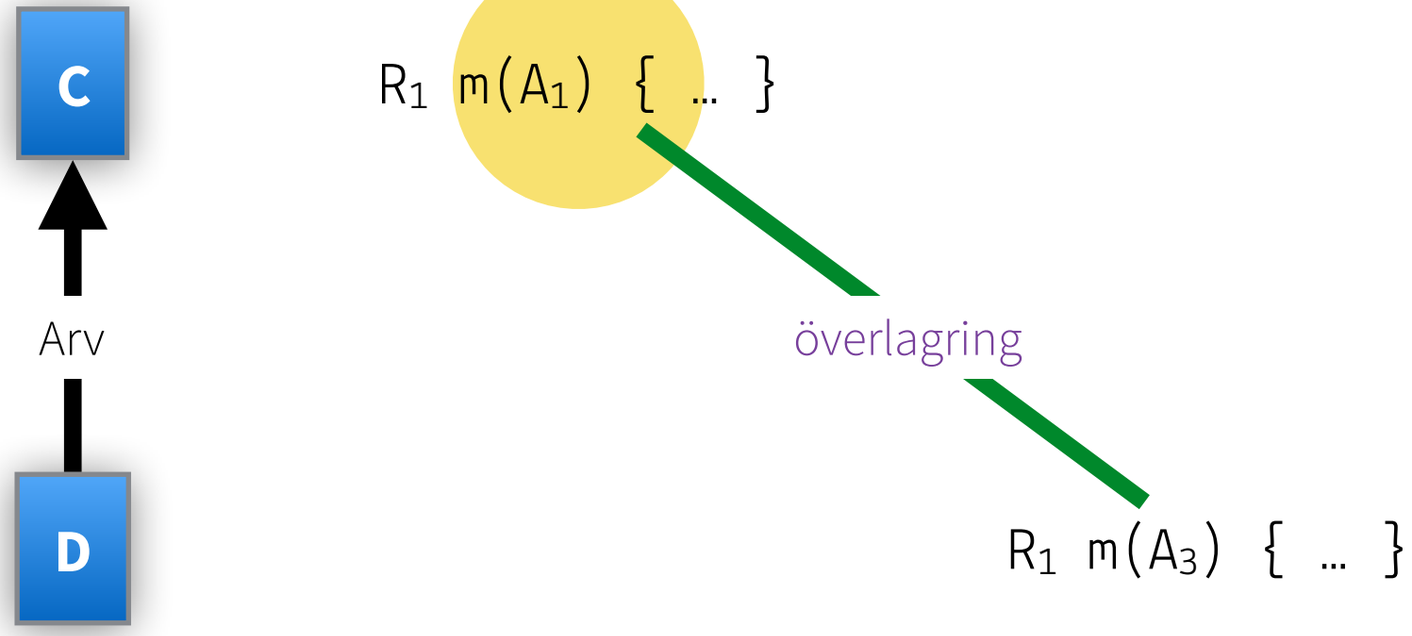
```
cd.m(a1);
```

A1 = Instrument

A3 = Fiol

R1 = Geometrisk form

R2 = Kvadrat




```
C cd = new D();
A1 a1 = new A1();
```

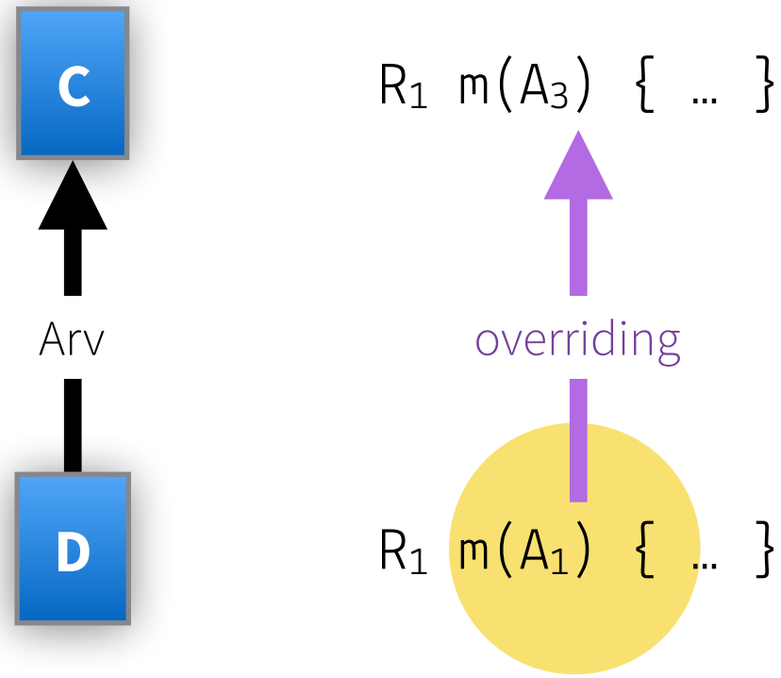
```
cd.m(a1);
```

A1 = Instrument

A3 = Fiol

R1 = Geometrisk form

R2 = Kvadrat



Weakening

```
C cd = new D();
A3 a3 = new A3();
```

```
cd.m(a3);
```

👍

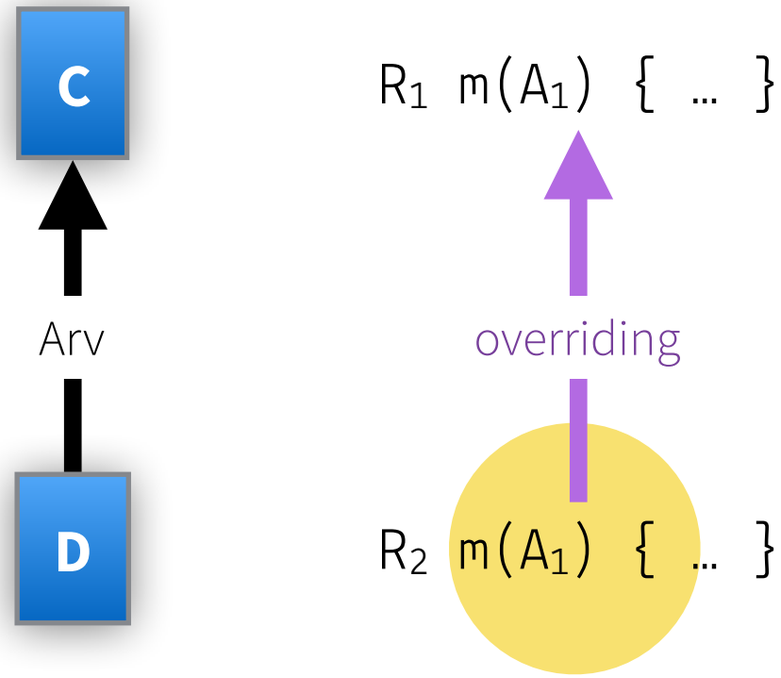
Men – fungerar inte i Java!

A_1 = Instrument

A_3 = Fiol

R_1 = Geometrisk form

R_2 = Kvadrat



Strengthening

```
C cd = new D();
A1 a1 = new A1();
```

```
R1 r = cd.m(a1);
```

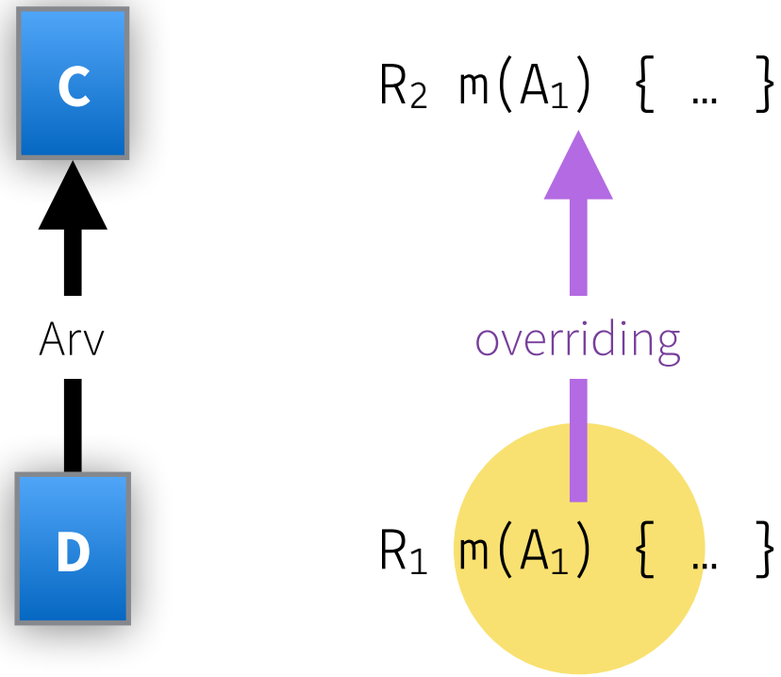
👍

A_1 = Instrument

A_3 = Fiol

R_1 = Geometrisk form

R_2 = Kvadrat




```
C cd = new D();
A1 a1 = new A1();
```

```
R2 r = cd.m(a1);
```

👎

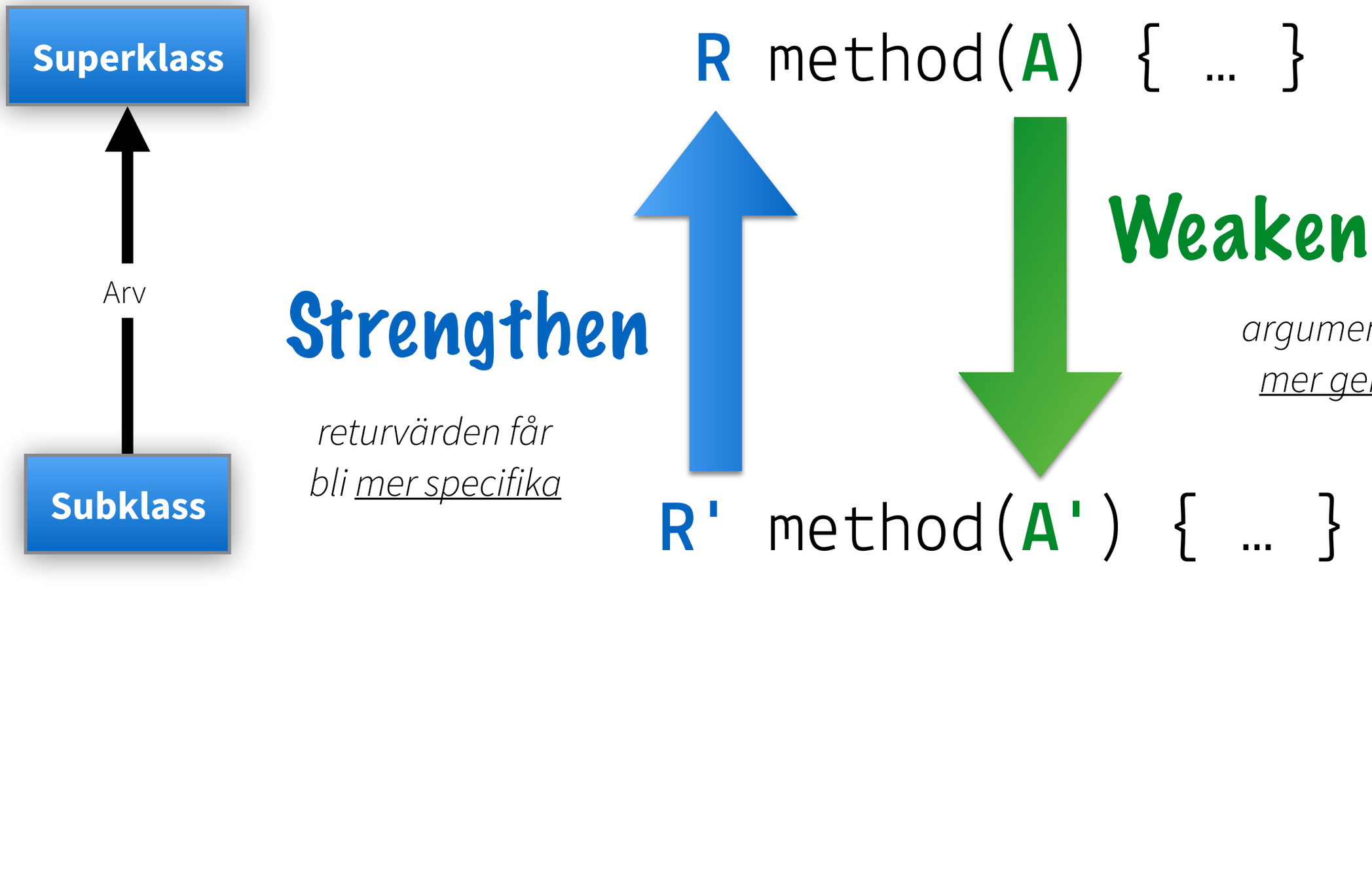
A1 = Instrument

A3 = Fiol

R1 = Geometrisk form

R2 = Kvadrat

Tillåtna typförändringar vid overriding



Sammanfattning

Kovarians (specifik → specifik)

En mer *specifik* metod (i en subklass) *kan* returnera mer *specifika returvärden* (men inte mer generella).

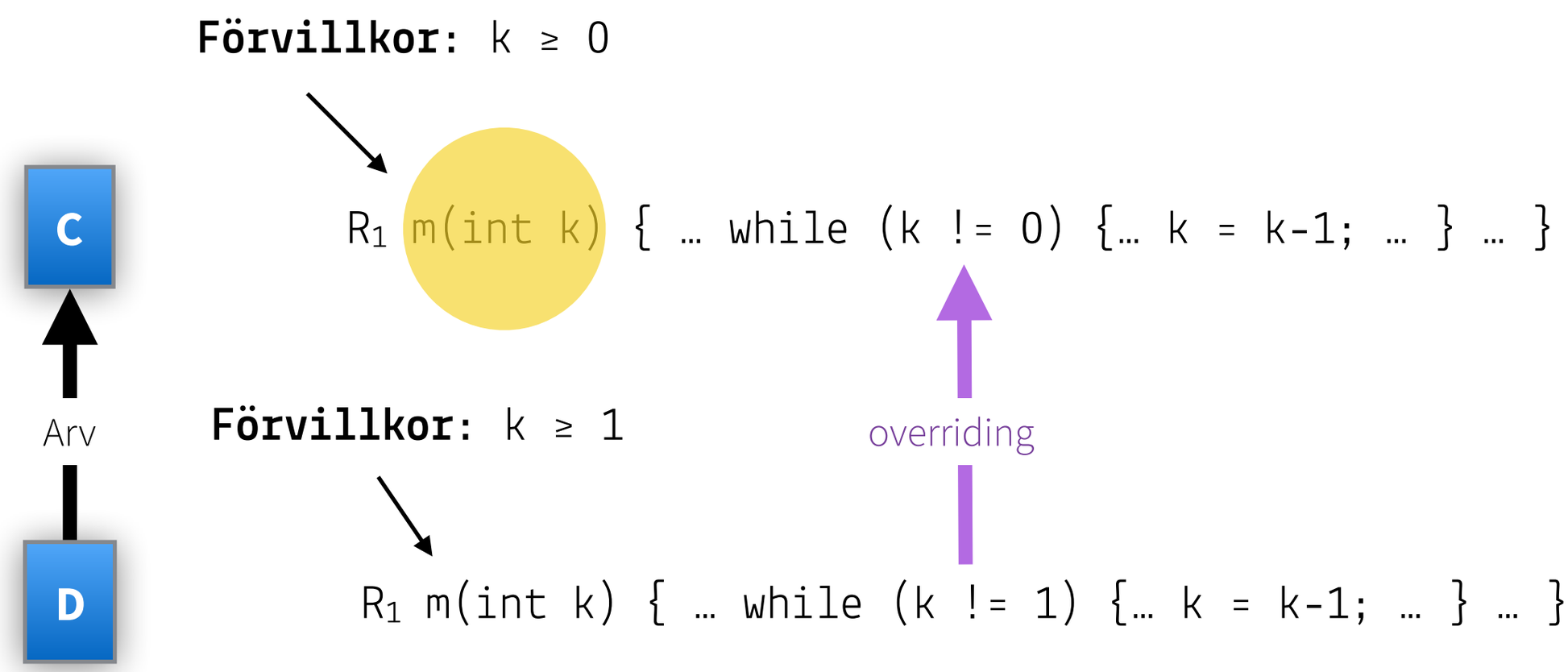
Kontravarians (specifik → generell)

En mer *specifik* metod (i en subklass) *kan* tillåta mer *generella argument* (men inte mer specifika).

Observera att olika objektorienterade språk kan implementera detta i olika utsträckning.

Java, t.ex., tillåter inte kontravariants i argumenten utan **invarians** – att argumenten har samma typer.

Med det gäller vad metoderna *gör* också!

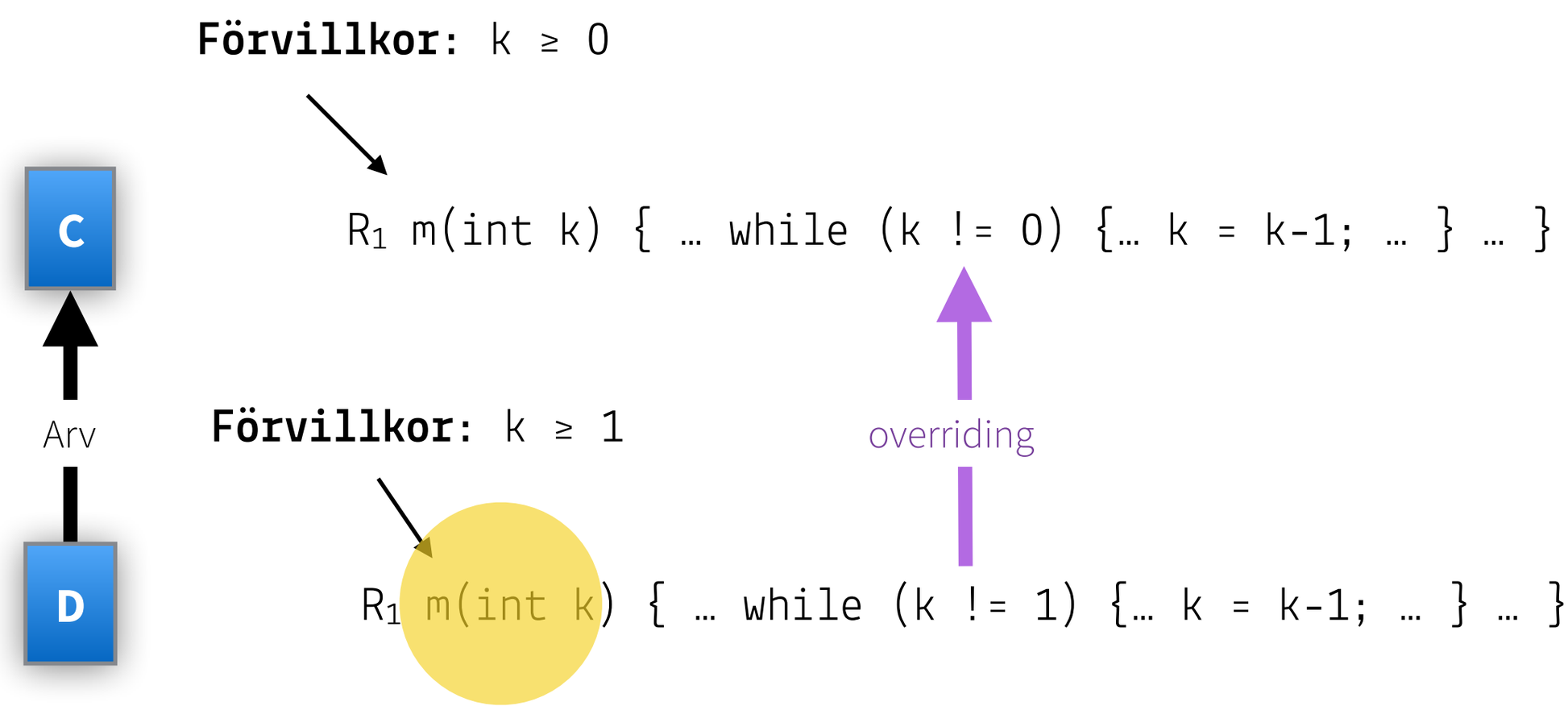



```
C c = new C();
```

```
c.m(0);
```

👍

Med det gäller vad metoderna *gör* också!

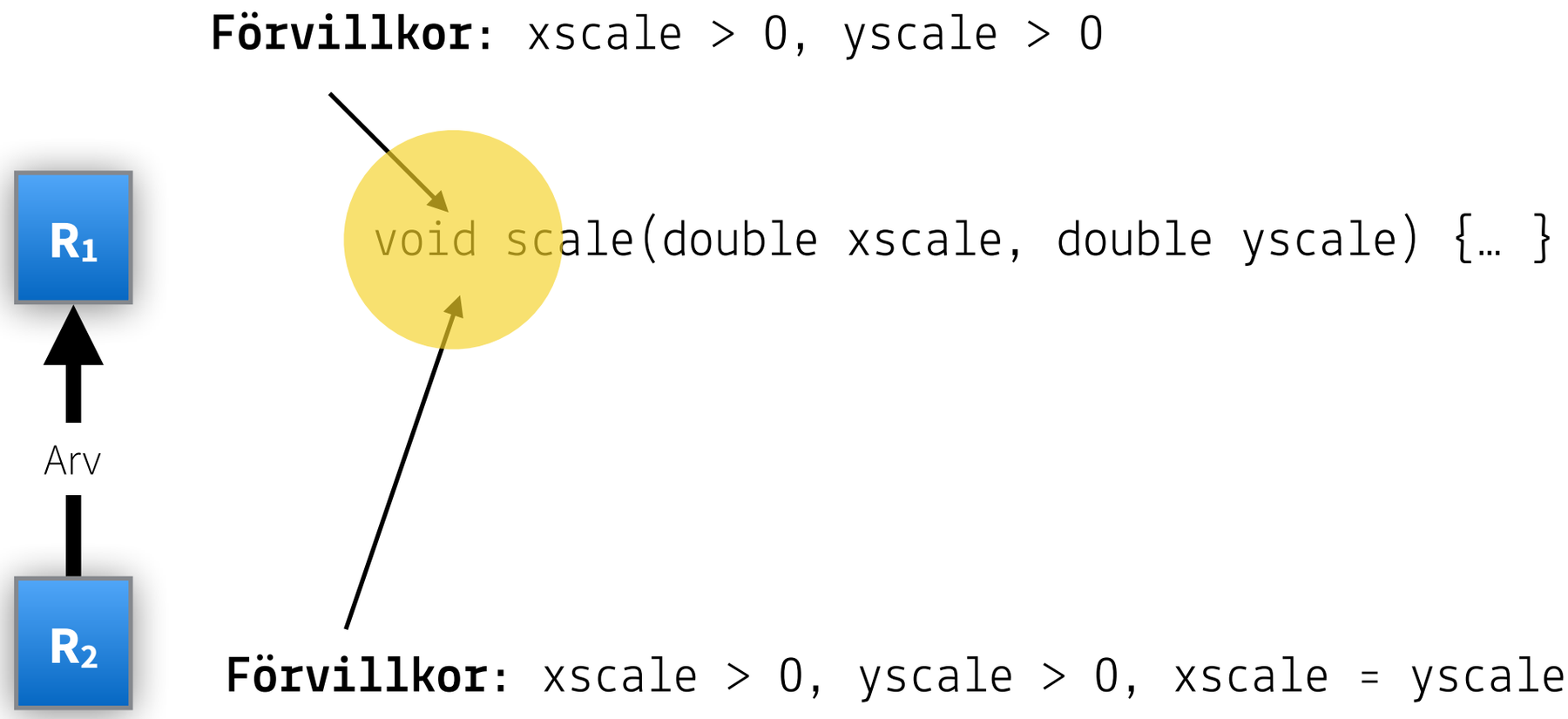



```
C cd = new D();
```

```
cd.m(0);
```

👎

Man behöver inte ens använda overriding…




```
R1 r = new R2();
```

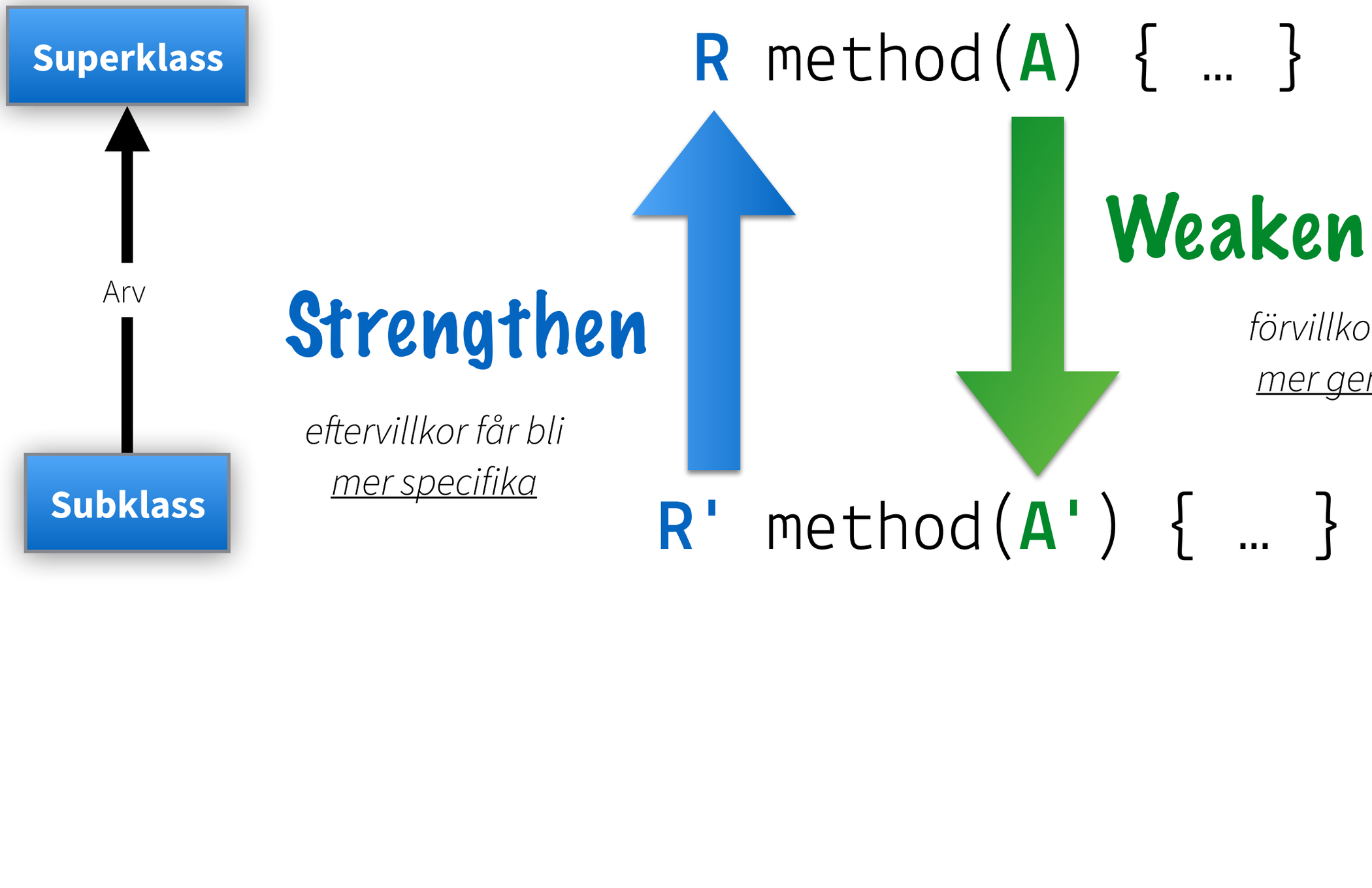
```
r.scale(2.0,3.0);
```

👎

R_1 = Geometrisk form

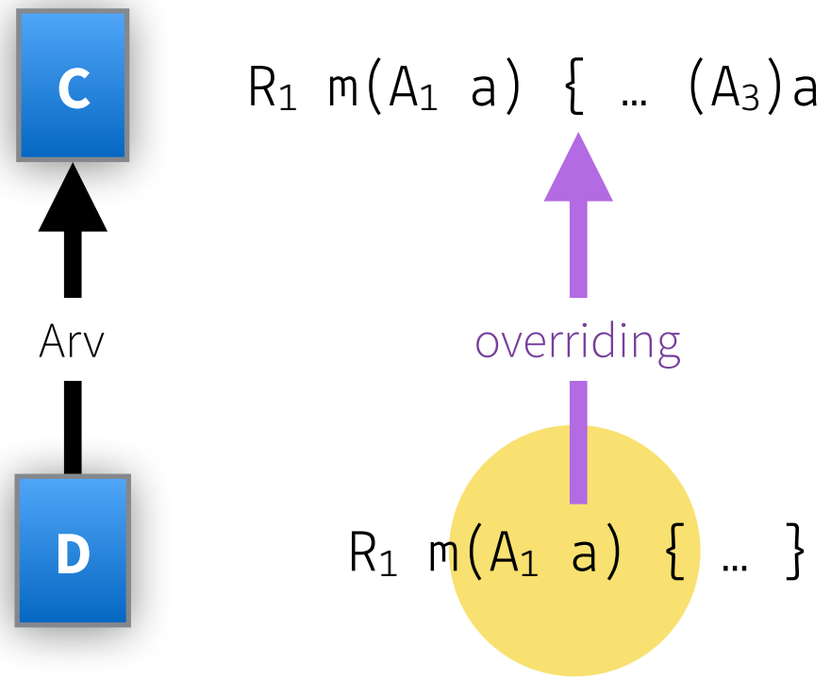
R_2 = Kvadrat

Tillåtna förändringar av för/eftervillkor



Simulerad weakening i Java

- Statisk typkontroll är egentligen bara specialfall av för/eftervillkor som går att kontrollera vid kompileringstillfället.
- en metoddeklaration R m(A a){…} har samma principiella innebörd som deklarationen Object m(Object a) {…} *plus* förvillkoret att a tillhör klassen A och eftervillkoret att returnvärdet tillhör klassen R.



Förvillkor: a tillhör klassen A_3

Simulerad Weakening

```
C cd = new D();
A3 a3 = new A3();
```

```
cd.m(a3);
```

👍

A_1 = Instrument

A_3 = Fiol

R_1 = Geometrisk form

R_2 = Kvadrat

Liskovs substitutionsprincip (LSP)

*Låt P vara ett program som bl.a. har objekt av typen T. Om S är en **subtyp** av T kan vi **ersätta** T-objekt med S-objekt i P utan att de eftersträvansvärde egenskaperna i programmet ändras. (Liskov 1987)*

Barbara Liskov
ACM Turing Award

- Kontravarians för argument.
- Kovarians för returtyper och undantag (exceptions).
- Beteendevillkor (behavioural constraints):
  Förvillkor kan inte förstärkas.
  Eftervillkor kan inte försvagas.
  Subtyper måste bevara supertypens samtliga invarianter.
  Subtyper får inte tillåta tillståndsförändringar som inte supertypen tillåter.
- **Om en subklass uppfyller villkoren för en subtyp enligt LSP underlättas programmeringen och risken för fel minskar.**
  I någon mening handlar LSP om inkapsling.

Tillbakablick

- Vad är en **typ**?
- Vad är en **subtyp**?
- Vad är relationen mellan **subklass** och **subtyp**?

  Subklassning i de flesta språk ger inte automatiskt subtypning (á la Liskov)

- Hur kan man göra **överlagring** och **overriding** (i Java)?

Undantagshantering

Undantagshantering

- I Java hanteras fel via undantag (exceptions). Undantag är objekt.
- Man kan själv ”kasta (throw) ett undantag (exception)”

```
throw new Exception()
```

- Flyttar kontrollflödet till **närmaste matchande omslutande** **`catch`**-block
- Exempel på hur undantag kan fångas:

```
try {
  Rectangle r = (Rectangle) someObject;
  int x = y / z;
} catch(ClassCastException e) {
   …
} catch(ArithmeticException e) {
   …
}
```


matchningsordning

Finally

```
void postMessage(User u, Server s, Message m) {
  try {
    Session session = s.logIn(u.id(), u.password());
    session.post(m);
  } catch (MalformedMessageException e) {
    u.notify(…);
  } finally {
    session.logout();
  }
}
```

- Körs alltid, oavsett utgång i **`try`**-blocket.
- Tillåter oss att lämna tillbaka resurser (”städa”) oavsett vad som händer.
- **`finally`** kan finnas utan **`catch`**. Om ett undantag kastats fortsätter undantagshanteringen efter **`finally`**-blocket.

```java
int foo(int a, int b) {
    return a / b;
}

int bar(int a) {
    return foo(a, 0);
}

int baz() {
    return bar(42);
}
```

Undantag!

```java
public static void main(String[] args) {
    try {
        baz();
    } catch (IllegalArgumentException e) {
        System.out.println("A");
    } catch (ArithmeticException e) {
        System.out.println("B");
    } finally {
        System.out.println("C");
    }
}
```

ArithmeticException
är inte en subklass till
IllegalArgumentException

B
C

```java
int foo(int a, int b) {
    return a / b;
}

int bar(int a) {
    return foo(a, 0);
}

int baz() {
    return bar(42);
}
```

Undantag!

```java
public static void main(String[] args) {
    try {
        baz();
    } catch (IllegalArgumentException e) {
        System.out.println("A");
    } catch (Exception e) {
        System.out.println("B");
    } finally {
        System.out.println("C");
    }
}
```

ArithmeticException

ArithmeticException är en subklass till Exception

B
C

```java
int foo(int a, int b) {
    return a / b;
}

int bar(int a) {
    return foo(a, 0);
}

int baz() {
    return bar(42);
}
```

Undantag!

```java
public static void main(String[] args) {
    try {
        baz();
    } catch (Exception e) {
        System.out.println("A");
    } catch (ArithmeticException e) {
        System.out.println("B");
    } finally {
        System.out.println("C");
    }
}
```

ArithmeticException is a subclass of Exception

A
C

```java
int foo(int a, int b) {
    return a + b;
}

int bar(int a) {
    return foo(a, 0);
}

int baz() {
    return bar(42);
}
```

```java
public static void main(String[] args) {
    try {
        baz();
    } catch (Exception e) {
        System.out.println("A");
    } catch (ArithmeticException e) {
        System.out.println("B");
    } finally {
        System.out.println("C");
    }
}
```

C

```java
int foo(int a, int b) {
    return a / b;
}

int bar(int a) {
    try { return foo(a, 0); } finally { System.out.println("Z"); }
}

int baz() {
    return bar(42);
}
```

Undantag!

```java
public static void main(String[] args) {
    try {
        baz();
    } catch (IllegalArgumentException e) {
        System.out.println("A");
    } catch (ArithmeticException e) {
        System.out.println("B");
    } finally {
        System.out.println("C");
    }
}
```

Z
B
C

"Checked" / "Unchecked"

- Kastande av undantag kan
  – Vara en del av programlogiken. (T.ex. när en fil som skall öppnas inte finns.)
  – Bero på programfel. (T.ex. användning av nullreferenser.)

- Java skiljer på
  – "**checked**" undantag som är en del av programmets funktion.
  – "**unchecked**" undantag som visar på programfel.

- Java kräver att man uttryckligen anger när en funktion/metod kan komma att kasta "checkade" undantag.

- Skillnaden mellan dem är i någon mening godtycklig.

Definiera egna undantag

```
/// Checked
class MalformedMessageException extends Exception { … }

/// Unchecked
class MalformedMessageException extends RuntimeException { … }
```

"Checked" / "Unchecked"

```
class MalformedMessageException extends Exception {}

void post(Message m) throws MalformedMessageException {
    … throw new MalformedMessageException(); …}
```

- Om `MalformedMessageException` ärver av `Exception` är den ”**checked**”

  Kräver att funktioner/metoder som anropar post också anger **throws** `MalformedMessageException`, *alternativt* har ett `catch`-block runt anropet av post.

```
class MalformedMessageException extends RuntimeException {}

void post(Message m) {
    … throw new MalformedMessageException(); …}
```

- Om `MalformedMessageException` ärver av `RuntimeException` är den ”**unchecked**” – behöver varken fångas eller explicit propageras.

- jmf. `NullPointerException`

Om MalformedMsg är ett "checkat" undantag

```
void post(Message m) throws MalformedMsgException {
    … throw new MalformedMessageException(); …}
```

propagera

```
void postMessage(User u, Server s, Message m) throws MalformedMsgException {
  Session session = s.logIn(u.id(), u.password());
  session.post(m);
}
```

eller

```
void postMessage(User u, Server s, Message m) {
  Session session = s.logIn(u.id(), u.password());
  try {
    session.post(m);
  } catch(MalformedMsgException e) {
    …
  }
}
```

hantera

Tänk på....

Kovarians för undantag!

Sammanfattning

Subklasser ger inte subtyper automatiskt — vi måste aktivt arbeta för detta

Liskovs substitutionsprincip guidar oss, och motiverar varför subtyper är A Good Thing™

Experimentera med detta tills du förstår varför och hur!

Java använder undantagshantering för felhantering

Finally används också för resurshantering orelaterat till undantag

Utdelad kod denna föreläsning: Undantagshantering.java — experimentera!