Viktiga programmeringsbegrepp: Kontrakt

Vad lovar {klassen, metoden, fältet}?

jonas.kvarnstrom@liu.se – 2017

Kontrakt

- **Kontrakt**: Överenskommelse som anger
  - Vad som ska **tillhandahållas**
  - Vad som förväntas i **utbyte**
  - Allmänna **regler** runt utbytet
  - …

- Inom **objektorienterad programmering**:
  - Vilka värden kan en metod **ta emot**?
  - Vad garanterar metoden att den **gör**, om den får sådana värden?
  - Vad **returnerar** den?
  - Vad garanterar en **klass** angående sitt tillstånd och beteende?
  - …

Kod kan innehålla (vissa) formella kontrakt

```
class Circle {
  private double x, y, r;
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
  public double getArea() {
     return Math.PI * this.r * this.r ;
   }
}
```

Krav på input: Parametrarna måste vara av typ double

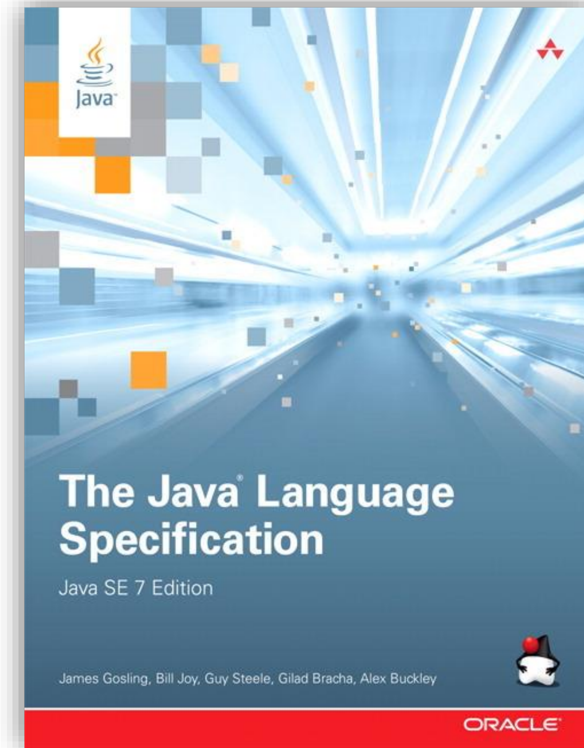
Löfte om resultattyp: Returnerar alltid double

Följer direkt från manifest typning
Så vanligt att man oftast inte ens ser det som kontrakt

Varför räcker inte koden?


```
class Circle {
  private double x, y, r;
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
  public double getArea() {
      return Math.PI * this.r * this.r ;
   }
}
```

+



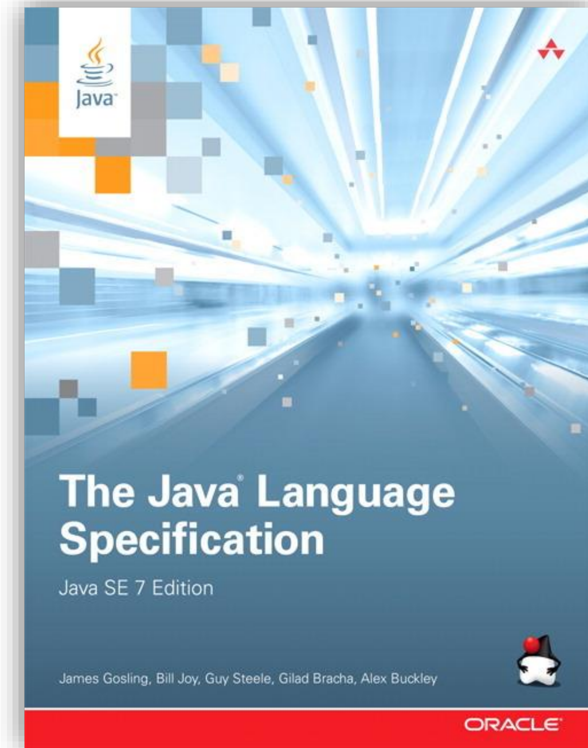
Definierar exakt vad programmet gör!

Varför räcker inte detta?

Varför räcker inte koden? (2)


```
class Circle {
  private double x, y, r;
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
  public double getArea() {
     return Math.PI * this.r * this.r ;
   }
}
```

+



Svårt att inse alla konsekvenser av tusentals rader kod

Kontraktet bör sammanfatta vad som garanteras, krävs

Varför räcker inte koden? (3)

- Vi är inte intresserade av vad koden gör!
  - Mycket viktig **konceptuell skillnad** mellan:

Vad vi faktiskt gör
(koden)

Implementations-detaljer

Vad vi lovar att göra, och att fortsätta göra i all evighet

(kontraktet)

Implementations-detaljer

Buggar

Buggar

Exemple 1: Sortering

jonkv@ida

- Exempel: **Sortera namn**
  - **Input**:
    - [A Z:son,
      E X:son,
      B Y:son,
      C Y:son,
      D Z:son]
  - Nu vill vi sortera:
    - Primärt – på *efternamn*
    - Sekundärt, vid samma efternamn – *på förnamn*
  - Vi vill alltså få:
    - [E X:son,
      B Y:son,
      C Y:son,
      A Z:son,
      D Z:son]

Samma efternamn

Samma efternamn

Exempel 1: Metod

- En metod:
  - Börja med en lista
    - [A Z:son, E X:son, B Y:son, C Y:son, D Z:son]
  - Sortera på förnamn *utan att tänka på efternamn*
    - [A Z:son, B Y:son, C Y:son, D Z:son, E X:son]
  - Sortera på efternamn *utan att tänka på förnamn*
    - **4 möjliga resultat, samtliga XYYZZ**:
      [E X:son, B Y:son, C Y:son, A Z:son, D Z:son]
      [E X:son, C Y:son, B Y:son, A Z:son, D Z:son]
      [E X:son, B Y:son, C Y:son, D Z:son, A Z:son]
      [E X:son, C Y:son, B Y:son, D Z:son, A Z:son]

Vi tänker bara på efternamn
Namn med samma förnamn
kan kastas om

- Med **stabil sortering**: Byt *bara* ordning på två element *om det krävs*
  [E X:son, B Y:son, C Y:son, A Z:son, D Z:son]

B Y:son fortfarande före C Y:son
A Z:son fortfarande före D Z:son
➔ Löser vårt problem!

Exempel 1: Vad kan vi se i koden?

- Anta sorteringsmetoden **saknar kontrakt**
  - Måste **läsa koden…**

```
public class List {
   private String[] strings;
   public void sort() {
      for (int i = 0; i < strings.length - 1; i++) {
         for (int j = 1; j < strings.length - i; j++) {
            if (strings[j - 1].compareTo(strings[j]) > 0) {
               String temp = strings[j - 1];
               strings[j - 1] = strings[j];
               strings[j] = temp;
            }
         }
      }
   }
}
```

Aha – den sorteringen är stabil! Då utnyttjar vi det!

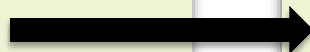
Exempel 1: Vad är tillåtet?

- Senare: Någon vill förbättra listklassen!
  - Sorteringen var **ineffektiv**

Om vi har detta…

```
public class List {
  …

  public void sort() {
    // Utför bubble sort
    // (råkar vara stabil)
    …
    …
  }
}
```



Får vi skriva om så här?

```
public class List {
  …

  public void sort() {
    // Utför heap sort
    // (snabbare, men inte stabil)
    //
    …
  }
}
```

Andra *kanske* förlitar sig på egenskaper koden "råkade" ha → dilemma!

Kontrakt 1

jonkv@ida

- Undvik genom kontrakt!
  - Vissa språk har formellt stöd – t.ex. Eiffel

```
set_hour (new_hour: INTEGER)
    -- Set `hour' to `new_hour'
    require
        valid_argument: 0 <= new_hour and new_hour <= 23
    do
        hour := new_hour
    ensure
        hour_set: hour = new_hour
    end
```

precondition

postcondition

  - Ofta får man istället använda **dokumentationen**
  - Idealet: **Bara** det som står i kontraktet gäller – titta aldrig på koden!

```
class List {
  /** Sorterar listan efter ordlängd. */
  void sort() { … }
}
```

Här står inget om stabilitet.

Då kan vi inte förutsätta det!

Kontrakt 2

- Ju tydligare, desto bättre…

```
class List {
  …

  /** Sorterar listan efter ordlängd.
      Stabilitet garanteras inte.
  */
  void sort() {
    …
  }
}
```

Beskriv gärna mer om
vad koden lovar och *inte* lovar

Exempel 2: Vad utlovas?

- Får jag ("Cirkel-användare") skicka in en negativ radie här?
  - **Ser** inget som skulle orsaka problem…
  - Men vem vet hur klassen kan **ändras i framtiden**?

```
class Circle {
  double x, y, r;

  Circle(double x, double y, double r) {

    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

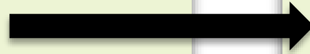
Exempel 2: Vad är tillåtet?

- Får jag ("Cirkel-utvecklare") ändra min kod så här?
  - Verkar rimligt att cirklar ska ha positiv radie
  - Men kod som förut fungerade kommer nu att krascha!

```
class Circle {
  double x, y, r;

  Circle(double x, double y, double r) {

    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```



```
class Circle {
  double x, y, r;

  Circle(double x, double y, double r) {
    if (r <= 0.0) {
      // Signalera fel…
      throw …;
    }
    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

Exempel 2: Kontrakt?

- Med kontrakttänkande:

```
class Circle {
  double x, y, r;
  /** Skapar en cirkel med angivna
      koordinater och radie.
   */
  Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

Metoden tar emot double.

Det står inget om begränsningar.

Alltså lovar den att klara godtycklig double.

Exempel 2: Kontrakt?

jonkv@ida

- Med kontrakttänkande:

```
class Circle {
  double x, y, r;
  /** Skapar en cirkel med angivna
      koordinater och radie. Radien
      måste vara strikt positiv.
   */
  Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }
}
```

Lovar att klara strikt positiva radier.

Kräver sådan input.

Skickar du in annat är beteendet **odefinierat** (kan krascha, …)

Grundregel för kontrakt:

Dokumentera alltid vad koden **kräver**
och vad den **lovar**

Relaterad grundregel:

Dokumentera inte vad koden **gör**
utan vad den **ska göra**
(och gärna **varför**)

Principer som underlättar:
Cohesion, Single Responsibility Principle

Introduktion

- **Kontrakt** anger **ansvarsområden**
  - Så hur **delar vi upp** ansvaret på flera klasser?
  - Underlättande principer:
    - Öka *Cohesion*
    - Minska *Coupling*
    - Single Responsibility Principle

Bra: Hög <u>cohesion</u> = sammanhållning

jonkv@ida

- Klassens funktionalitet ska **<u>vara meningsfull</u>** och **<u>hänga ihop</u>**
  - Bör gå att ge en **<u>kort</u>** **<u>sammanfattning</u>** av klassen
    - "En **<u>List</u>** är en sekvens av godtyckliga element"
    - "En **<u>LayoutEngine</u>** kan göra en specifik typ av layout för ett **<u>Diagram</u>**"
  - "En **<u>MiscWorker</u>** genererar slumptal, sparar filer och visar hjälptexter"?
    - ➔ Något är troligen fel…

NAMING IS KEY

Låg coupling = få kopplingar mellan klasser

- Låg **coupling** = färre kopplingar **mellan** klasser
  - Varje extra koppling ökar **beroendet** mellan klasserna
    - **Kan** ge problem
  - ➔ Funktionalitet som **hör ihop** bör vara **samlad**
- Som alltid måste man hitta en **balans**: Ansvarsområden ska vara
  - **Sammanhängande**
  - Lätta att **beskriva**
  - **Lagom** stora

Undvik överdriven uppdelning

- **Så: Överdriv inte uppdelningen**!
  - Exempel:
    - **ListStructure** lagrar data i ordnad listform
    - **ListPrinter** skriver ut listor
    - **ListSorter** sorterar listor
    - **ListRandomizer** slumpar ordningen i en lista
    - …
  - Klasserna blir **tätt kopplade**, beroende av varandra
  - Varje extra klass ger **ytterligare ett begrepp** att hålla reda på
  - ➔ För mycket!
    - Listfunktionerna borde vara **tillräckligt** sammanhängande för att vara **en** klass
    - **List** lagrar och manipulerar listor av element

Rätt indelning?

- Mönster: Skapa **gränssnitt** som ”håller reda på **alla konstanter**”
  - (Gränssnitt kan innehålla *konstanta* fält)
    - ```
      public interface Constants {
          public final double PI = 3.14159;
          public final int BUTTON_POS = 200;
          public final String playerName1 = "Main Player"
      }
      ```
  - Men: *Konstanterna hör ofta till någon annan* – placera dem där!
    - GameGUI hanterar gränssnittet [och vet/äger allt om det, inklusive positioner]
    - ```
      public class GameGUI {
          public final int BUTTON_POS = 200;
          …
      }
      ```

“It’s mine!
Mine mine mine
mine mine
mine!”

- ”Constant interface” ses ofta som ***antimönster***

Cohesion/Coupling hör ihop med:

Single Responsibility Principle

1. En klass bör ha **ett enda ansvarsområde**
   - "Göra **en** sak" – inte generera slumptal + spara filer
2. Ansvarsområdet bör hanteras **helt inom klassen**
   - Inte vara utspritt

Ökar cohesion

Minskar coupling

```
class GameComponent extends JComponent {
   public void paintComponent(...) {
      player.drawImage(...);
      if (player.getY()+50 < ball.getY()+32) { ... }
      ...
   }
}
```

50 råkar vara höjden på spelarfiguren.
Varför ska GUI-komponenten veta detta,
medan spelarobjektet innehåller bilden + koordinater?

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

SRP och sammanhållning för metoder

- **SRP** + önskan om **hög sammanhållning** gäller även metoder!
  - Varje metod ska ha ett **tydligt ansvar**, "göra **en** sak"

```
class MyGameComponent {
    void tick() { // Called 50 times per second
        ... A lot of code updating object positions ...
        ... A lot of code checking for collisions ...
        ... A lot of code updating scores ...
    }
}
```

3 separata uppgifter,
svårt att få översikt

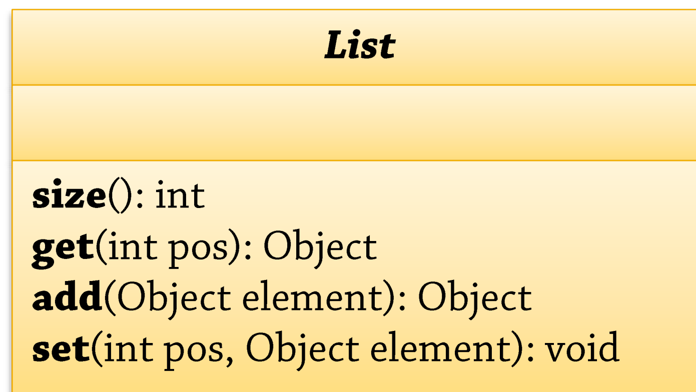
```
class MyGameComponent {
    void tick() { // Called 50 times per second
        updatePositions();
        checkCollisions();
        updateScores();
    }
    void updatePositions() { ... }
}
```

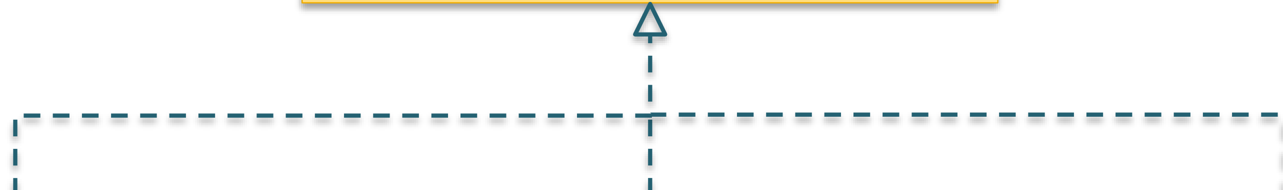
Lätt att se
vad som händer under tick()

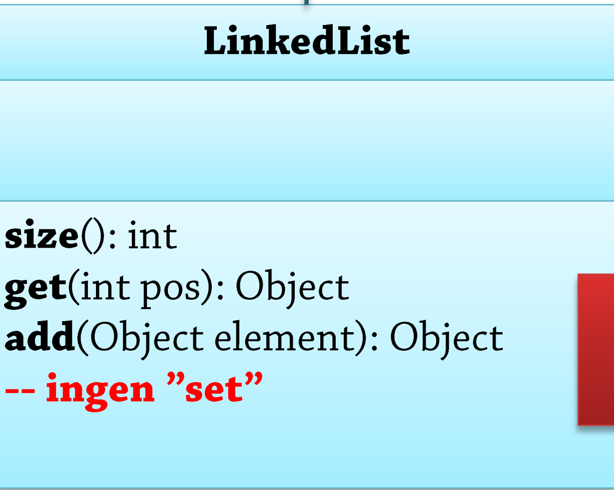
En metod som gör **en** sak

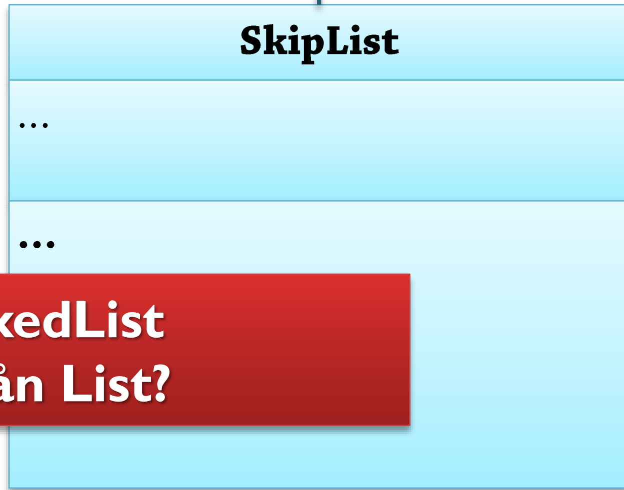
Hur ska en typhierarki se ut?
Kontrakt och Liskov Substitution Principle

Krav på hierarkier 1



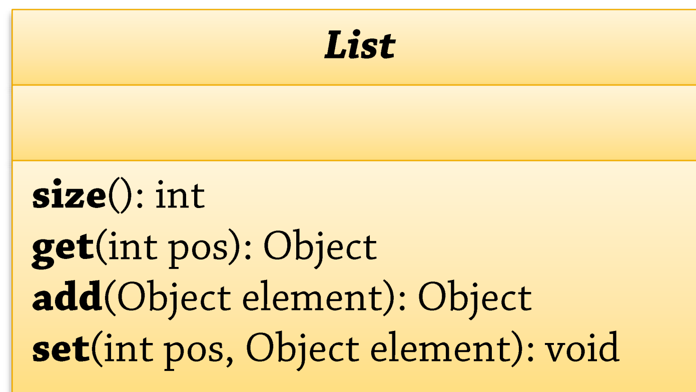


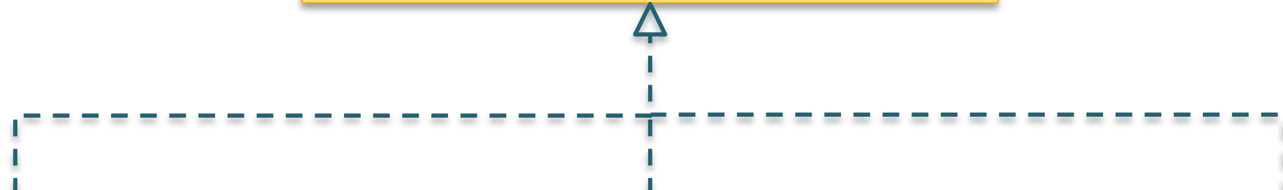


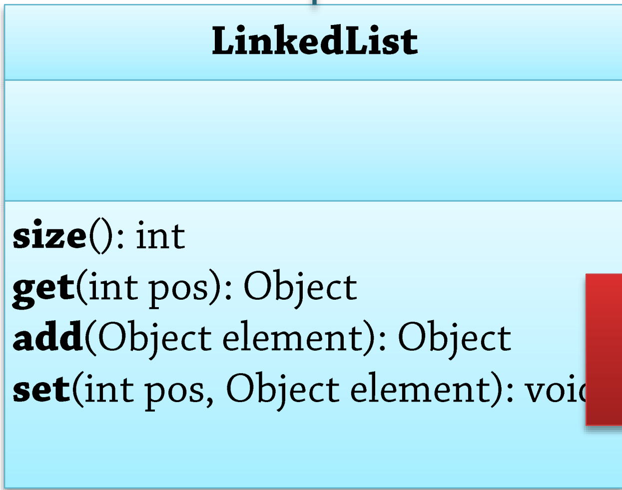


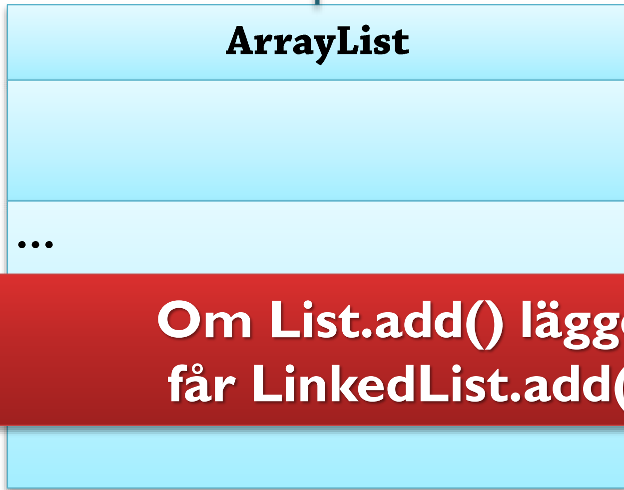
Får subtypen LinkedList sakna metoder från List?

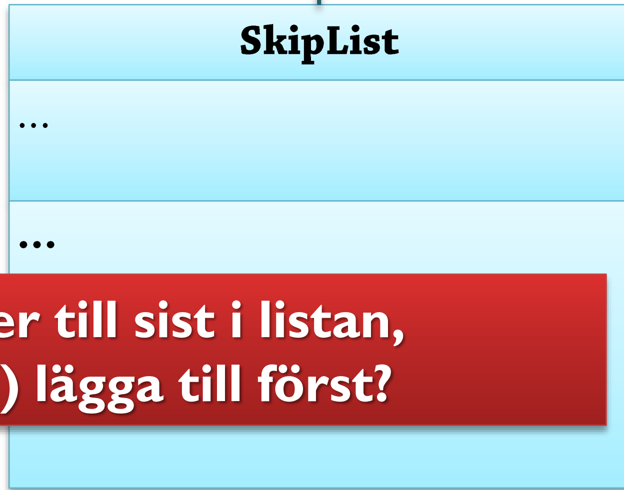
Krav på hierarkier 2











Om List.add() lägger till sist i listan, får LinkedList.add() lägga till först?

Krav på hierarkier 3

- Även **List** har ett **kontrakt**
  - En ArrayList **är** en List
    ➔ måste uppfylla samma kontrakt!
  - (Fåglar har fjädrar, ankor är fåglar
    ➔ ankor har fjädrar)

- Subtypens eget kontrakt får bara:
  - **Lova mer** – ge *starkare* garantier (ankor kan kvacka)
  - **Kräva mindre** – ha *svagare* krav

- Det är detta som garanterar:
  - Om vi accepterar en List,
    är vi nöjda med
    vilken subtyp som helst!

```
void quicksort(List someList) {
    // Vi programmerar
    // utifrån List:s kontrakt

    // Detta uppfylls oavsett om vi får en
    // ArrayList, SkipList, LinkedList, …
}
```

Krav på hierarkier 4: LSP

- En formell beskrivning: **Liskov Substitution Principle**
  - *Let q(x) be a property provable about objects x of type T.*
    Then q(y) should be provable for objects y of type S,
    where S is a subtype of T.
  - **Om** vi kan bevisa något om alla List-objekt,
    måste detta även gälla för alla LinkedList-objekt.

Barbara Liskov

Kontrakt, arv och kovarianta returtyper

Kovarianta returtyper!

- LSP säger:
  - Subtypens eget kontrakt får bara:
    - **Lova mer** – ge *starkare* garantier
    - **Kräva mindre** – ha *svagare* krav

En subklass kan stärka löftet om returtyp

```
public interface List
{
    List makeCopy();
    List convert();
}
```

```
public class ArrayList implements List
{
    ArrayList makeCopy() { … }
    SkipList convert() { … }
}
```

Varierar "åt samma håll"
(en specialisering av List specialiserar även returtyperna)
➔ *kovariant*

Kontravarianta parametertyper?

- LSP säger:
  - Subtypens eget kontrakt får bara:
    - **Lova mer** – ge *starkare* garantier
    - **Kräva mindre** – ha *svagare* krav

Kan en subklass ställa svagare krav på parametertyp?

```
public interface List extends Collection
{
    void addAll(List other);
}
```

```
public class ArrayList implements List
{
    void addAll(Collection other) { … }
}
```

Varierar "åt motsatt håll" → kontravariant

Inte tillåtet i Java:
Resulterar i overloading (två *varianter* av metoden)

Behåll dina hemligheter – minska ditt kontrakt

Information Hiding

Encapsulation

- **SortedList**: Garanterar att elementen är i **sorterad ordning**!

```
class SortedList {
   String[] elements;
   int      size;

   String   get(int pos) { return … }

   void     add(String el) {
      int pos = findNewPositionFor(el);
      insertAt(pos, el);
   }

   int      findNewPositionFor(String el) { … }
   void     insertAt(int pos, String el) { … }
}
```

Lagring för alla element

Hämtar element

Stoppar in element på rätt plats…

… med dessa hjälpmetoder

Att gömma information

jonkv@ida

- Ofta vill vi "gömma" delar av klasser och objekt – varför?

```
class SortedList {
    String[] elements;
    int      size;

    String   get(int pos) { return … }

    void     add(String el) {
        int pos = findNewPositionFor(el);
        insertAt(pos, el);
    }

    int      findNewPositionFor(String el) { … }
    void     insertAt(int pos, String el) { … }
}
```

Gömma: För att inte förvirra användarna

- Mindre synligt → mer överskådligt

```
class SortedList {
    String[] elements;
    int      size;

    String   get(int pos) { return … }

    void     add(String el) {
        int pos = findNewPositionFor(el);
        insertAt(pos, el);
    }

    int      findNewPositionFor(String el) { … }
    void     insertAt(int pos, String el) { … }
}
```

Användaren **behöver** bara detta!

Att det andra syns **förvirrar**, gör klassen oöverskådlig!

Gömma: Minska åtaganden (kontrakt)

- Det vi har **visat upp**, måste vi normalt **ha kvar**

```
class SortedList {
  String[] elements;
  int      size;

  String   get(int pos) { return … }

  void     add(String el) {
    int pos = findNewPositionFor(el);
    insertAt(pos, el);
  }

  int      findNewPositionFor(String el) { … }
  void     insertAt(int pos, String el) { … }
}
```

Visar vi detta, blir det en del av kontraktet

Behövs inte!

Göm det ➔
enbart get/add används ➔
vi kan **byta ut** resten
om vi vill ➔ frihet!

Gömma: Minska åtaganden (kontrakt)

- Det vi har **visat upp**, måste vi normalt **ha kvar**

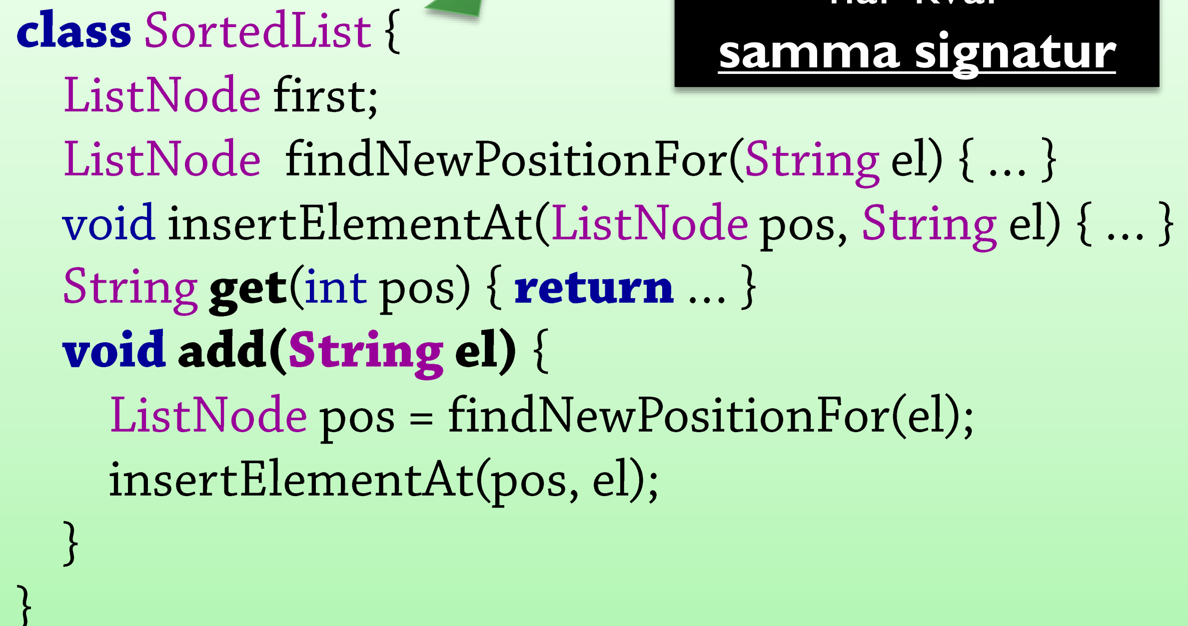
```
class SortedList {
  String[] elements;
  int      size;

  String   get(int pos) { return … }

  void     add(String el) {
     int pos = findNewPosit
     insertAt(pos, el);
  }

  int      findNewPositio
  void     insertAt(int pos
}
```

Alternativ implementation, där den "publika" delen har kvar **samma signatur**



Gömma: För att uppfylla vårt kontrakt


```
/**
 * This class maintains a sorted list of elements.
 * As long as the contents of an element does not change,
 * elements will always be in sorted order.
 */
class SortedList {
    String[] elements;
    int                  size;
    int    findNewPositionFor(String el) { … }
    void  insertAt(int pos, String el) { … }

    void  add(String el) {
        int pos = findNewPositionFor(el);
        insertAt(pos, el);
    }
}
```

Om användare kan anropa insertAt med fel position blir listan osorterad

Då **bryter** vi vårt **löfte**!

Hur gömmer vi information?

Att gömma data i Java

- Många OO-språk tillåter oss att **gömma data**
  - I Java kan klasser, gränssnitt, fält och metoder vara:
    - **public** – alla har tillgång
    - **protected** – tillgång i underklasser + andra i samma paket
    - [*inget*] – tillgång i samma paket (bör undvikas)
    - **private** – tillgång inom samma klass

```
public class SortedList {
    private String[]        elements;
    private int             size;
    private int findNewPositionFor(String el) { … }
    private void insertElementAt(int pos, String el) { … }
    public void add(String el) {
        int pos = findNewPositionFor(el);
        insertElementAt(pos, el);
    }
}
```

Vad hade private betytt utan OO?

Vi lovar mindre i vårt kontrakt!

Skapa ett **stabilt** gränssnitt för "allmänheten": add()
Göm all info om implementationsdetaljer

Två betydelser hos…

Inkapsling (Encapsulation)

Koppla ihop data och funktioner

- Fundamentalt i OO
- "Objekt: en **kapsel** som innehåller både data och funktioner"

Begränsa tillgång till medlemmar

- Genom accessmodifierare
- "Vissa medlemmar är instängda i en **ogenomskinlig kapsel**, och kan inte ses eller utnytttjas från utsidan"

Privata fält

jonkv@ida

- Vanlig princip: **Fält** är implementationsdetaljer, **ska vara privata**
  - **Om** det verkligen är rimligt att andra klasser ska komma åt fältvärden:
    - Skapa en public **accessor**-metod (**getter**) vid behov
      Skapa en public **mutator**-metod (**setter**) vid behov

Kan ändra intern representation:
Ändra bara getter/setter, som är i samma klass

```
private double diameter;
/** Set the radius to r (must not be negative). */
public void setRadius(final double radius) {
    if (r >= 0.0) diameter = 2 * radius;
    else throw new IllegalArgumentException("Negative radius: " + r);
}
```

Kan lägga till konsistenskontroll

Namngivning: Getter, Setter

- Namngivning:
  - ```
    private double radius;
    private boolean visible;

    /** Set the radius to r (must not be negative). */
    public void setRadius(final double r) {
        if (radius >= 0.0) radius = r;
        else throw new …;
    }

    public double getRadius() {
        return radius;
    }
    public boolean isVisible() {
        return visible;
    }
    ```

Setters:
void <u>set</u>Property(…)

Getters:
<u>get</u>Property()

Booleska getters:
<u>is</u>Property()

Mer info: http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html
    ```

Tillåtna publika fält

jonkv@ida

- Vissa anser att det finns **undantag**
  - ”Rena” datastrukturer
    (inte mycket beteende, osannolikt att det skulle ändras i framtiden)

```
public class Dimension {
    public int width;
    public int height;
}
```

Konstruktorer i abstrakta klasser


```
abstract class GenericList implements List {
    protected int length;
    protected GenericList() {
        this.length = 0;
    }
    public int size() { return length; }
    public int indexOf(Object obj) {
        for (int i = 0; i < length; i++)
            if (get(i).equals(obj)) return i;
        return –1;
    }
}
```

Konstruktorer i **abstrakta** klasser
kan vara *protected*:
Tillgängliga i subklasser