Föreläsning 1

- Kursinformation
- ADT, Abstraction
- Generics
- Stack
- Queue
- Method chaining

Kursinformation

Moment i kursen

Inlämningsuppgifter, 3 hp

- Laborationsserie
  9-10 laborationer ska genomföras. Obligatorisk närvaro
- Workshopserie
  3 workshops ska genomföras. Obligatorisk närvaro.
- Programmeringsuppgifter
  2 uppgifter ska lösas och redovisas individuellt
  Varje redovisning innebär även granskning av en studiekamrats lösning

Gruppuppgift, 1,5 hp

En grupp om 5-6 personer designar och implementerar ett system.

Tentamen, 3 hp, preliminärt

- Torsdagen den 28/3 - 2019, kl 14.15 – 19.15
- Lördagen den 4/5 - 2019, kl 09.15 – 14.15
- Torsdagen den 15/8 – 2019, kl 14.15 – 19.15

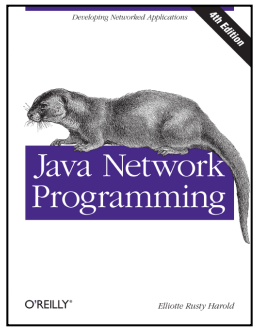
För godkänt resultat på kursen fordras:

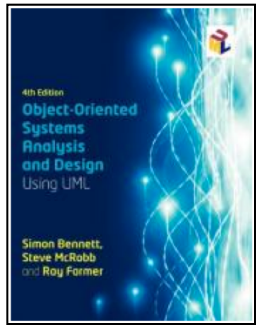
- godkända inlämningsuppgifter, laborationer (minst 8 godkända) och workshops (minst 2 godkända)
- godkänd gruppuppgift
- godkänd tentamen.

Betyget på kursen baseras på resultatet på tentamen.

Kursinformation

				Litteratur	Innehåll
V4	Må	DK1	RA	Collections1.pdf	Abstraction, ADT, Generics, Stack, Kö
	On	DK2	RA	Collections2.pdf	Collection, Collections, Map
	On	OOAD1	JH	OO 7.3, 7.5, 8.3-4	Klassdiagram fördjupning, objektdiagram
	To	OOAD2	JH	OO 7.3, 7.5, 8.3-4	Tillståndsdiagram, kommunikationsdiagram sekvensdiagram
V5	Må	DK3	RA	JNP 1, 2	Introduktion, Strömmar
	On	W1	JH		Klassdiagram, tillståndsdiagram
	To	DK4	RA	JNP s 53-70	Trådar
V6	Må	DK5	RA	JNP s 53-70	Trådar
	On	DK6	RA	JNP s 70-92	Trådar
	To	OOAD3	JH	9.1-3, 9.7	Sekvensdiagram
V7	Må	P1	RA		Frågestund + Handledning P1
	On	W2	JH		Sekvensdiagram
	To	P1	RA		Redovisning P1
V8	Må	DK7	RA	JNP 4, 5, 6, 7, 12	Nätverk 1
	On	DK8	RA	JNP 8	Nätverk 2
	To	DK9	RA	JNP 9	Nätverk 3
V9				Arbete med P2 och Gruppuppgift	
V10	Må	P2	RA		Frågestund + Handedning P2
	On	OOAD4	JH	8.5, 15	Designmönster
		W3	JH		Datorlab designmönster
	To	P2	RA		Redovisning P2
V11	Må	GU	RA		Handledning GU
	To	GU	RA		Handledning GU
V12	Må		RA,JH		Redovisning gruppuppgift
	On		RA,JH		Redovisning gruppuppgift
	To		RA,JH		Redovisning gruppuppgift
V13	Må		RA		Restredovisning
	To	Tentamen			
V18	Lö	Omtentamen 1			





Abstraktion

När man skriver en klass i java anger man synligheten för instansvariabler och metoder.

Synligheten ***private*** anges som regel för ***instansvariabler*** och för metoder som endast är avsedda att användas inom klassen. Instansvariablerna kan på så sätt endast användas av kod inom klassen.

Synligheten ***public*** anges för de ***metoder som ska kunna anropas från kod i andra klasser***.

De metoder som är ***public***-deklarerade utgör användargränsnittet (***interfacet***) till objektet. Och det är som regel endast dessa metoder som en användare behöver känna till.

Användaren är abstraherad från hur objektet är byggt (***implementerat***) men kan använda objektet.

Abstraktion - exempel

Det går utmärkt att använda ett JFrame-objekt och ett JLabel-objekt utan att känna till:
* instansvariabler som finns i respektive klass
* hur metoder i klasserna är implementerade


```
public class JFrameEx {
  private JLabel label = new JLabel();
  private JFrame frame = new JFrame();

  private void showFrame() {                                  // publika metoder i JFrame
    frame.setTitle("En instans av JFrame");                   // setTitle
    frame.setLocation(200, 100);                              // setLocation
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);     // setDefaultCloseOperation
    frame.add(label);                                         // add
    frame.pack();                                             // pack
    frame.setVisible(true);                                   // setVisible
  }

  public void action() {                                      // publika metoder i JLabel
    label.setText("DA343A");                                  // setText
    label.setHorizontalAlignment(JLabel.CENTER);              // setHorizontalAlignment
    label.setForeground(Color.BLUE);                          // setForeground
    label.setFont(new Font("SansSerif",Font.PLAIN,48));       // setFont
    label.setPreferredSize(new Dimension(222,58));            // setpreferredSize
    showFrame();
  }
}
```

Vilka metoder kan man anropa i klassen JFrameEx?

JFrameEx.java

ADT – Abstact Data Type

Primitiva datatyper i java är t.ex. ***int***, ***long*** och ***double***.

En abstrakt datatyp håller någon form av data och har ett användargränssnitt (publika metoder) för att använda datan. Hur datan lagras och klassen är implementerad behöver användaren ej nödvändigtvis känna till.

Ett exempel på en abstrakt datatyp är klassen ***String***.

- Klassens data består av en sekvens av tecken.
- Klassen har ett antal publika metoder att anropa, t.ex. charAt(pos), length() och substring(start,end). När du använder objekt av typen String så är det endast de publika metoderna du behöver känna till.

StringBuilder

StringBuilder kapslar också in sekvens av tecken. Skillnaden till klassen *String* är att det går att lägga till tecken och ta bort tecken ur sekvensen.

Exempel på publika metoder i StringBuilder:

```
public char charAt(int index);
public int indexOf(String s);
public int length();
public String substring(int start, int end);
public String toString();
public StringBuilder append(String s);
public StringBuilder delete(int start, int end);
public StringBuilder insert(int offset, String s);
```

Exempel på användning

```
String name = "Gustav";
String street = "Föreningsgatan 11";
String town = "Malmö";
StringBuilder sb = new StringBuilder();
sb.append(name);
sb.append(", ").append(street).append(", ").append(town); // method chaining

System.out.println(sb.toString());

sb.insert(6, " Hansson");
sb.delete(26, 30);
System.out.println(sb.toString());
```

StringBuilderEx.java

ADT – Abstact Data Type

Med hjälp av en klass bygger man den abstrakta datatypen i java. Det är vanligt att man definierar funktionaliteten i en abstrakt datatyp med hjälp av ett ***interface***. Sedan låter man klassen implementera interfacet.

En objektsamling (*collection*), lagrar *referenser till objekt* på ett organiserat sätt. Figuren nedan visar i mitten ett interface som definierar de operationer som man ska kunna göra på en klass som är en *collection*. Till höger ser man ett antal klasser vilka implementerar funktionaliteten i en *collection*. Till vänster är det en klass som använder sig av av en *collection*-klass.

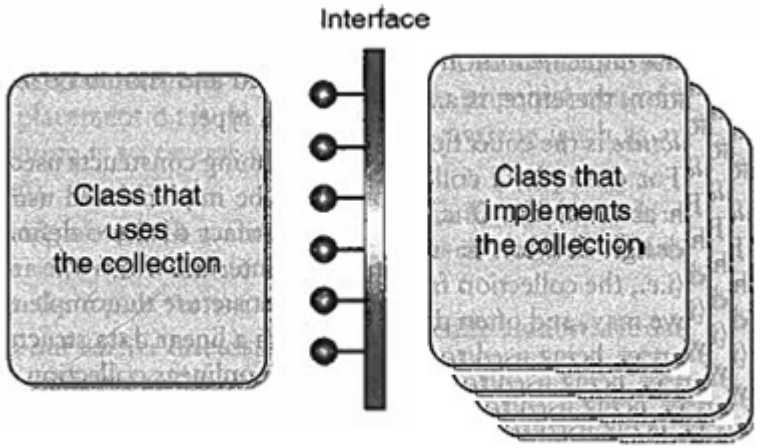
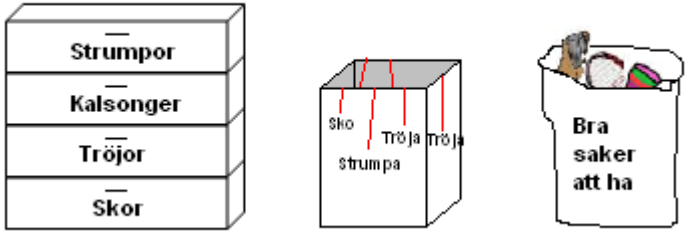


FIGURE 2 A well-defined interface masks the implementation of the collection

ADT – Abstact Data Type

Klasserna som implementerar interfacet lagrar elementen på olika sätt



vilket ger olika egenskaper, t.ex:

- Snabb sökning av objekt
- Objekten är ordnade enligt en viss princip

ADT - Stack

En **Stack** är en objektsamling med speciella regler för insättning och borttagning av element. I en stack lagrar och kommer man åt elementen enligt **LIFO**-principen – ”last-in, first-out”. Principen innebär att det endast är det senast lagrade elementet man har åtkomst till. Om man tar bort detta element så har man åtkomst till det näst senast lagrade elementet.

En vanlig bild av detta är en stapel av tallrikar. Den senast placerade tallriken är den man tar när man behöver en tallrik.

Operationer på en stack

push(elem)	lägger till ett element i stacken
pop() : elem	tar bort ett element från stacken och returnerar elementet
peek() : elem	returnerar elementet som är på tur att tas bort
isEmtpy() : boolean	returnerar true om stacken är tom och annars false
size() : int	returnerar antalet element på stacken

Stack för Integer-objekt

Operationer på en stack som lagrar Integer-referenser

push(Integer)	lägger till ett element i stacken
pop() : Integer	tar bort ett element från stacken och returnerar elementet
peek() : Integer	returnerar elementet som är på tur att tas bort
isEmtpy() : boolean	returnerar true om stacken är tom och annars false
size() : int	returnerar antalet element på stacken

Ett interface som definierar metoderna i en stack för Integer-objekt:

```
public interface Stack1 {
   public void push(Integer element);
   public Integer pop();
   public Integer peek();
   public boolean isEmpty();
   public int size();
}
```

Stack1.java

Stack för Integer-objekt

Att avgöra:

- Hur ska Integer-objekten lagras i klassen?
- Hur många element ska stacken rymma?

```
public class IntegerStack implements Stack1 {
   private Integer[] elements;
   private int size=0;

   public IntegerStack(int capacity) {
      elements = new Integer[capacity];
   }

   public void push(Integer element) {
      if(size>=elements.length)
         throw new StackOverflowException ();
      elements[ size ] = element;
      size++;
   }

   public Integer pop() {
      if(isEmpty()) {
         throw new EmptyStackException();
      }
      return elements[--size];
   }
   :
}
```

IntegerStack.java

Stack för Object-referenser

Operationer på en stack som lagrar Object-referenser

push(Object)	lägger till ett element i stacken
pop() : Object	tar bort ett element från stacken och returnerar elementet
peek() : Object	returnerar elementet som är på tur att tas bort
isEmtpy() : boolean	returnerar true om stacken är tom och annars false
size() : int	returnerar antalet element på stacken

Ett interface som definierar metoderna i en stack för Objekt-element:

```
public interface Stack2 {
   public void push(Object element);
   public Object pop();
   public Object peek();
   public boolean isEmpty();
   public int size();
}
```

En klass som implementerar Stack2:

```
public class ObjectStack  implements Stack2{
   private Object[] elements;
   private int size = 0;  // antal element i stacken
   :
}
```

Stack2.java

ObjectStack.java

Generics

Generics infördes i java 2004 (version 1.5).

Generics medger typkontroll vid användning av objektsamlingar (Object-referenser).

Generics används i klasser, interface och metoder.

Vid generics använder man en *typ-variabel* (T nedan) vilken kompilatorn ersätter med en explicit typ vid kompileringen. Man använder alltid stor bokstav för att ange en typvariabel. Vanliga är T, E, K, V…

```
public class GenericStack<T> implements Stack<T> {
   private T[] elements;
   private int size = 0;
   :
}
```

I kod:

```
GenericStack<Integer> stack1 = new GenericStack<Integer>(20);
GenericStack<String> stack2 = new GenericStack<String>(30);
:
stack1.push( 23 ); // autoboxing ok
stack2.push( ”HEJ” );
:
Integer nbr = stack1.pop(); // typkonvertering behövs ej
String str = stack2.pop();
```

Stack - interface för en generisk objekt-stack

```java
package f1;

public interface Stack<T> {
    /**
     * Placerar ett element i stacken.
     * @param element elementet att lägga på stacken
     */
    public void push(T element);

    /**
     * Returnerar det element som senast placerades i stacken. Elementet tas bort från stacken.
     * @return det element som senast placerades i stacken
     */
    public T pop();

    /**
     * Returnerar det element som senast placerade i stacken. Elementet är kvar i stacken.
     * @return det element som senast placerades i stacken
     */
    public T peek();

    /**
     * Returnerar true om stacken inte innehåller några element och false om det finns element i stacken.
     * @return
     */
    public boolean isEmpty();

    /**
     * Returnerar antalet element som finns i stacken.
     * @return antalet element som finns i stacken
     */
    public int size();
}
```

Stack.java

Generisk stack

Operationer på en stack som lagrar Generic-referenser

push(T)	lägger till ett element i stacken
pop() : T	tar bort ett element från stacken och returnerar elementet
peek() : T	returnerar elementet som är på tur att tas bort
isEmtpy() : boolean	returnerar true om stacken är tom och annars false
size() : int	returnerar antalet element på stacken

En klass som implementerar Stack

```
public class ArrayStack<T> implements Stack<T>{
   private T[] elements;
   private int size = 0;  // antal element i stacken

   private ArrayStack( capacity ) {
      elements = (T[])(new Object[ capacity ]);
   }

   public void push( T element ) { … }
   :
}
```

Klassens namn antyder hur implementeringen sker. **Array**Stack – elementen lagras i en array.

ArrayStack.java

Method chaining

En kedja av anrop i samma programsats kallas för *method chaining*. Det skulle t.ex. gå bra att skriva ArrayStack-klassen så att följande sats är korrekt:

```
ArrayStack<Integer> stack = new ArrayStack<Integer>(20);
stack.push(10).push(23).push(8).push(11);
```

Det krävs ändring i interfacet (StackMC). Metoden ***push*** ska nämligen returnera en referens till aktuellt ArrayStack-objekt (this).

```
public StackMC<T> push(T element);
```

Metoden ***push*** måste ändras så metoden returnerar en referens till ”sig själv” dvs. till aktuellt objekt.

```
public class ArrayStack2<T> implements StackMC<T>{
   :
   public ArrayStack<T> push( T element ) {
      if(size>=elements.length)
         throw new StackOverflowError();
      elements[ size++ ] = element;
      return this;
   }
   :
}
```

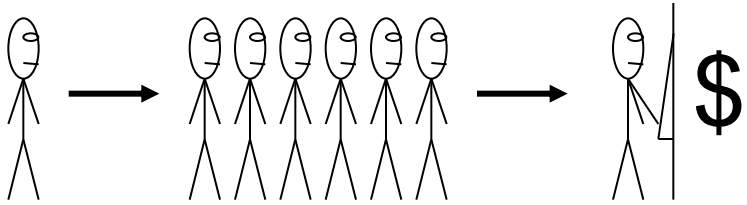
StackMC.java

ArrayStack2.java

Metoderna som returnerar element från stacken går ej att modifiera på ovanstående sätt.

ADT – Queue, Kö

En **Kö** är en datastruktur med speciella regler för insättning och borttagning av element. I en kö lagrar och kommer man åt elementen enligt **FIFO**-principen – ”First-in, first-out”. Principen innebär att man kommer åt elementen i samma ordning som de placerats i kön.
En vanlig bild av detta är en kö till uttagsautomaten. Den som är längst fram i kön står på tur att ta ut pengar medan nya köande ställer sig sist i kön.



Operationer på en kö

add(T)	lägger till ett objekt i kön
remove() : T	tar bort ett objekt från kön och returnerar referens till objektet
element() : T	returnerar en referens till objektet som är först i kön
isEmtpy() : boolean	returnerar true om kön är tom och annars false
size() : int	returnerar antalet objekt i kön

QueueEx.java