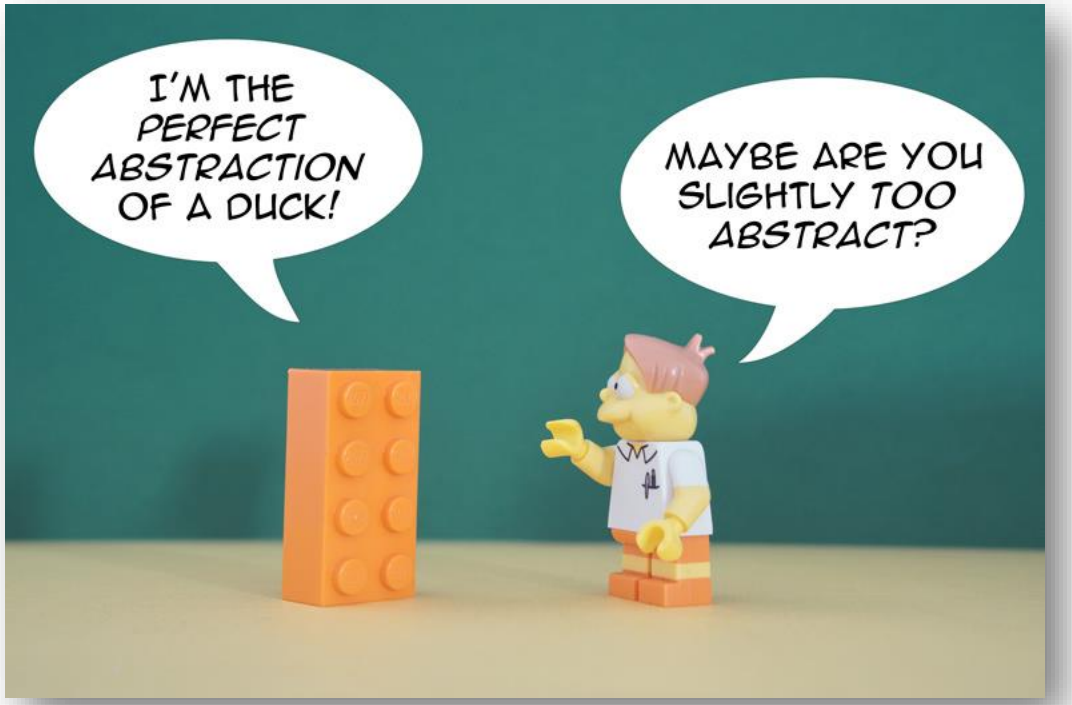
- DA343A -
Objektorienterad programutveckling, trådar och datakommunikation

Föreläsning 2

Föreläsning 2

- Abstraktion
- Generiska klasser (*generic*)
- Stack
- Kö (*queue*)



Abstraktion

Repetition

- När man skriver en **klass** i java anger man synligheten för instansvariabler och metoder.
- Abstraktion → Gömma vissa detaljer och enbart visa relevant information
- ***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.
- ***public*** anges för de ***metoder som ska kunna anropas från kod i andra klasser***.
- De metoder som är ***public***-deklarerade utgör ”användargränssnittet” till objektet.
  - Det är som regel endast dessa metoder som en användare behöver känna till.
- Abstraktion kan uppnås t.ex. genom att använda ***Interface*** som definierar ”regler” för klasserna

Abstraktion

Repetition

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

Abstraktion

Repetition

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
```

ADT – Abstact Data Type

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

- Inga objekt! (*pass-by-value*)

En abstrakt datatyp är en typ (eller klass) som lagrar någon form av data och har ett användargränssnitt (publika metoder) för att använda datan.

Hur datan lagras och klassen är implementerad behöver användaren inte nödvändigtvis känna till.

Man kan säga att en abstrakt datatyp är en datatyp vars interna representation är gömd för användaren.

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.

ADT – Abstact Data Type (String)

Method Summary

Methods

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(CharSequence cs) Compares this string to the specified CharSequence.
boolean	**contentEquals**(StringBuffer sb) Compares this string to the specified StringBuffer.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.

StringBuilderEx.java

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.

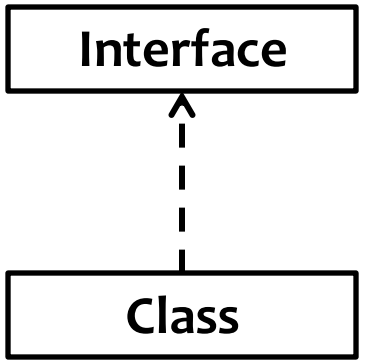
Exempel på publika metoder i StringBuilder:

```
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);
```

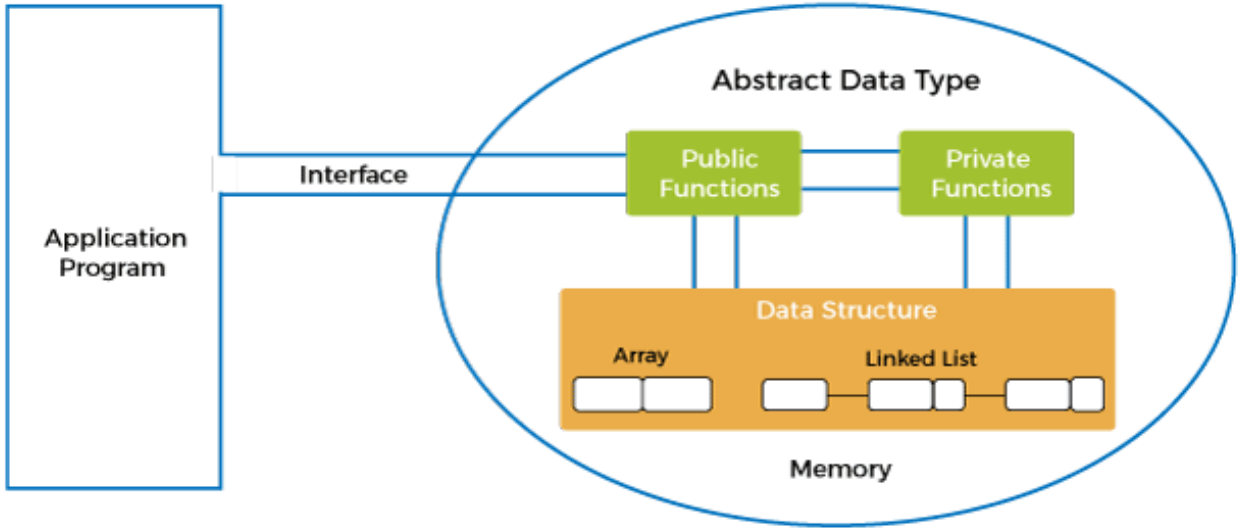
Exempel på användning

```
String name = "Gustav";
String street = "Föreningsgatan 11";
String town = "Malmö";

StringBuilder sb = new StringBuilder();

sb.append(name);
// method chaining
sb.append(",").append(street).append(",").append(town);
System.out.println(sb.toString());

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

ADT – Abstact Data Type

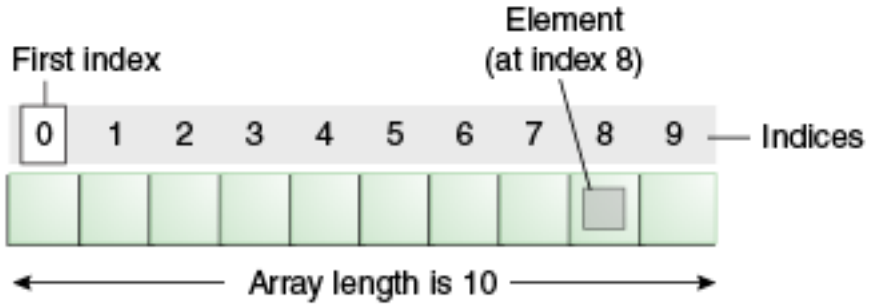


- Byggs i en klass (inkapsling).
- Definierar vilka operationer kan utföras men inte hur
- Abstrakt -> oberoende av implementationen
- Vanligt att man definierar funktionaliteten med hjälp av ett interface.
- Abstrakt syn på integer-array
  - Spara flera element av integer-typ
  - Läsa element på viss position (index)
  - Modifiera element på viss position
  - Sortera



ADT – Collection

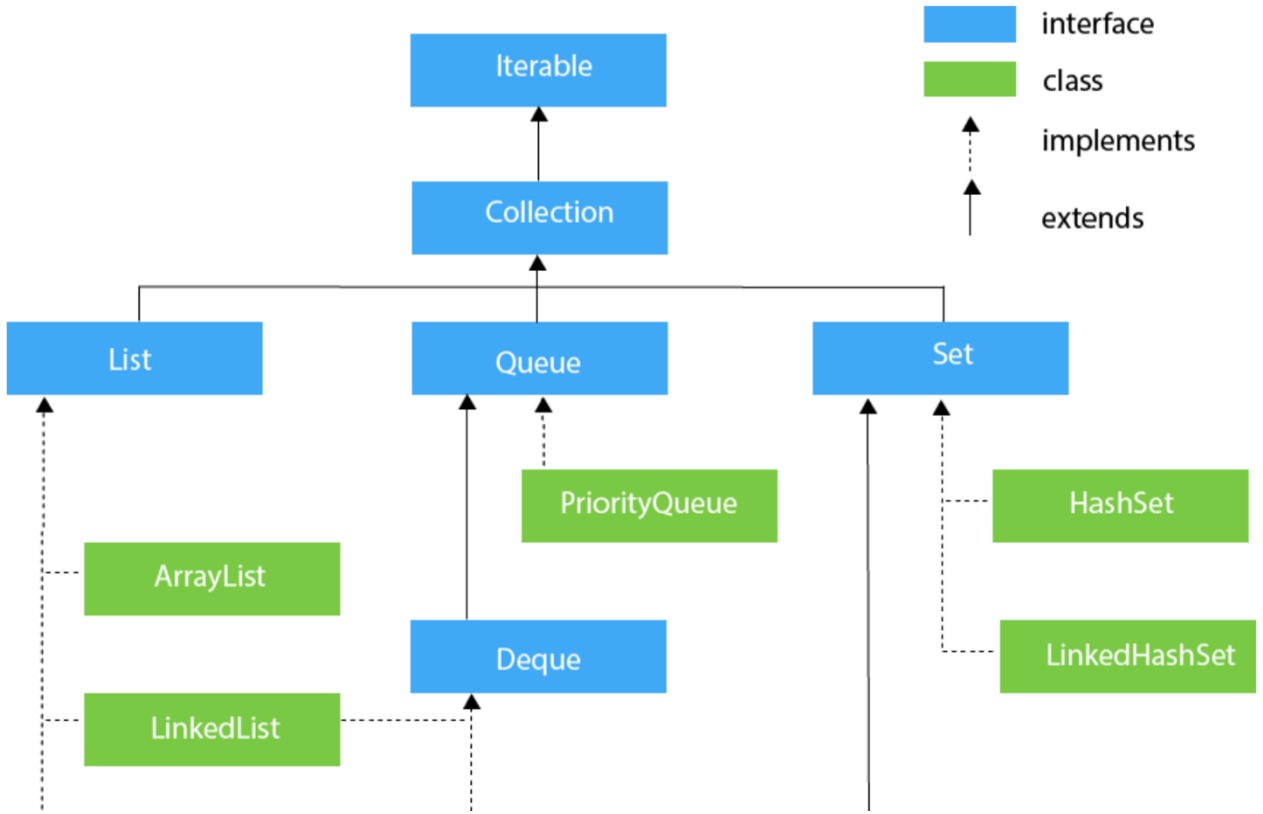
- Finns olika datatyper för att spara flera objekt
- T.ex *array*
  - Nackdelar: fast storlek, tar plats även om den är tom, objekt kan inte infogas ”i mitten”, …
  - Önskvärt med en mer dynamisk datastruktur




```
int[] intArray = new int[20];
intArray[0] = 10;
intArray[1] = 20;
System.out.println(intArray.length);
```

ADT – Collection

- En objektsamling (***collection***), lagrar *referenser till objekt* på ett organiserat sätt.
- Söka, sortera, infoga, ändra, ta bort objekt
- En Java Collection representerar en grupp av objekt
  - Snabb sökning av objekt
  - Objekten är ordnade enligt en viss princip
- Olika metoder
  - add(element), remove(element), size(), clear(), contains(element), …



ADT – Stack

- Objektsamling med speciella regler för insättning och borttagning av element

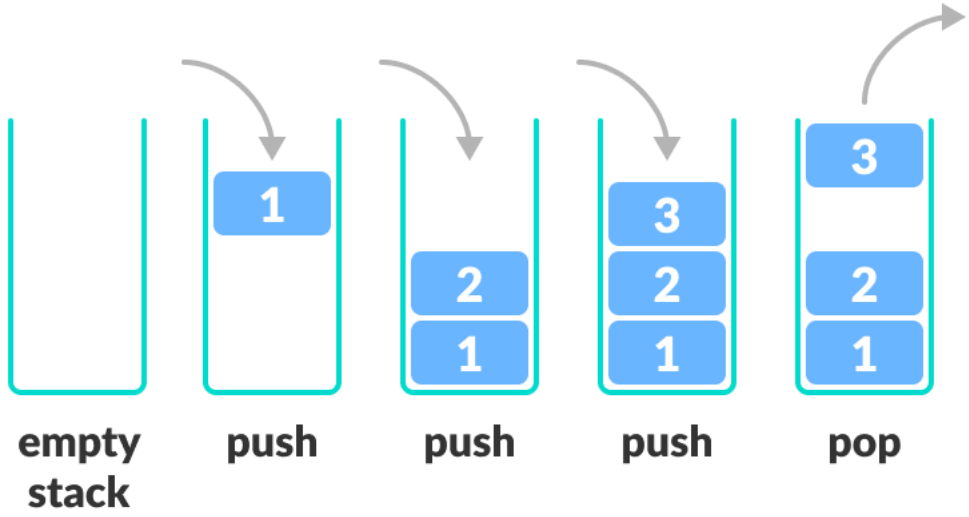
- **LIFO**-princip – ”*last-in, first-out*”
- Bara åtkomst till det senast lagrade elementet

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

- Används t.ex. inom sökning (*backtracking*)

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



MALMÖ UNIVERSITY

Stack1.java

Stack för Integer-objekt

Operationer på en stack som lagrar Integer-objekt

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();
}
```

IntegerStack.java

Implementering av Stack för Integer-objekt

Att avgöra:

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

Skillnad mellan int och Integer
int: primitiv datatyp
Integer: wrapper class

```
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){
            System.out.println("Stack overflow");
        } else {
            elements[ size ] = element;
            size++;}
    }

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

Stack för Object-referenser

Stack2.java

ObjectStack.java

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 f1.Stack2 {
    private Object[] elements;
    private int size=0;
```

Generics

- Generics medger typkontroll vid användning av objektsamlingar (Object-referenser).
- Det möjliggör att kompilatorn kan säkerställa rätt datatyp
- Generics används i klasser, interface och metoder.
- Vid generics använder man en *typ-variabel* (t.ex. *T*) 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 static < E > void printArray( E[] inputArray ) {
    for(E element : inputArray) {
        System.out.println(element);
    }
}

public static void main(String[] args) {
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    printArray(intArray);

    printArray(doubleArray);

    printArray(charArray);
}
```

Stack.java

Stack - interface för en generisk objekt-stack

```java
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 referens till det element som senast placerades i stacken
     */
    public T pop();

    /**
     * Returnerar det element som senast placerade i stacken. Elementet är kvar i stacken.
     * @return referens till 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 true om stacken är tom och false om det finns element i stacken
     */
    public boolean isEmpty();

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

ArrayStack.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 f1.Stack<T> {
    private T[] elements;
    private int size=0;

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

    public void push(T element) {
        if(size>=elements.length)
            throw new f1.StackOverflowException();
        elements[ size ] = element;
        size++;
    }
```

Klassens namn antyder hur implementeringen sker.
ArrayStack – elementen lagras i en array.

Method chaining

StackMC.java

ArrayStack2.java

- 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*** måste ändras så metoden returnerar en referens till ”sig själv” dvs. till aktuellt objekt (*this*).

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

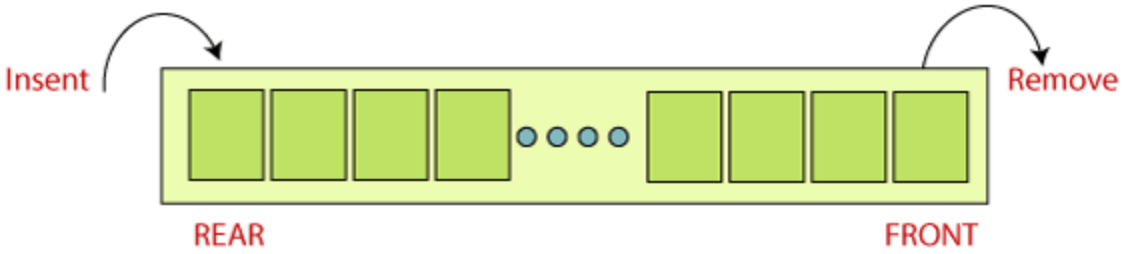
```
public class ArrayStack2<T> implements f1.StackMC<T> {
    private T[] elements;
    private int size=0;

    public ArrayStack2(int capacity) { elements = (T[])(new Object[capacity]); }

    public f1.StackMC<T> push(T element) {
        if(size>=elements.length)
            throw new f1.StackOverflowException();
        elements[ size ] = element;
        size++;
        return this;
    }
```

ADT – Queue

I en kö lagrars elementen enligt **FIFO**-principen – ”*First-in, first-out*”.

Principen innebär att man kommer åt elementen i samma ordning som de placerats i kön.

T.ex. telefonkö när man ringer till försäkringen



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