Föreläsning 3
Innehåll

- Jämföra element
  - interfacen `Comparable` och och `Comparator`
- Implementera generiska klasser
- Exceptions
- Dokumentationekommentarer – javadoc
- Enhetstestning - jUnit

Diskutera
Sortera objekt

Metoden `sort` är en statisk metod i klassen `Arrays`. Den sorterar vektorn som skickas med som argument.

Vad tror du händer när följande rader exekveras? Förklara varför? Fundera på hur elementen jämförs.

```
Person[] persons = new Person[4];
persons[0] = new Person("Kili", 1);
persons[1] = new Person("Balin", 2);
persons[2] = new Person("Dori", 4);
persons[3] = new Person("Fili", 3);
Arrays.sort(persons);
for(Person p : persons) {
    System.out.println(p);
}
```


Vad händer inuti `sort`?

- Inuti metoden `sort` måste elementen jämföras (mindre än, större än).
- Anropet `Arrays.sort(persons)` kräver att klassen `Person` implementerar interfacet `Comparable` och har en metod `compareTo`.
- Inuti `sort` används `Comparable` som typ för de element som ska sorteras. Vid jämförelser anropas metoden `compareTo`.
- Det är alltså i `compareTo` man bestämmer hur elementen ska jämföras och vad de ska sorteras efter.
- Om klassen `Person` inte implementerar `Comparable` genereras `ClassCastException` vid exekveringen.

Interfacet Comparable

```
public interface Comparable<T> {
   /**
    * Compares this object with the specified object for order.
    * Returns a negative integer, zero, or a positive integer as
    * this object is less than, equal to, or greater than the
    * specified object.
    */
   public int compareTo(T x);
}
```

- Objekt av klasser som implementerar detta interface går att jämföra med varandra och kan t.ex. sorteras.

Implementering av interfacet Comparable

Exempel

```
public class Person implements Comparable<Person> {
   private String name;
   private int id;
   ...

   public int compareTo(Person obj) {
      return Integer.compare(id, obj.id);
   }
}
```


Diskutera

Implementering av interfacet Comparable

Hur ska metoden `compareTo` se ut ifall man istället vill sortera personerna med avseende på namn?

```
public class Person implements Comparable<Person> {
   private String name;
   private int id;
   ...

   public int compareTo(Person obj) {




   }
}
```


Jämföra likhet

Metoderna `compareTo` och `equals`

- Interfacet `Comparable` innehåller bara metoden `compareTo`.
- Men för klasser som implementerar interfacet `Comparable` finns det två sätt att jämföra avseende likhet:

```
Person p1 = ...;
Person p2 = ;...
if (p1.compareTo(p2) == 0) {...}
if (p1.equals(p2)) {...}
```

- Båda sätten att jämföra bör ge konsistenta resultat.
- Därför bör metoderna `equals` och `compareTo` jämföra samma attribut.

Interfacet Comparator

- Ibland passar inte/fungerar inte lösningen med `Comparable`. T.ex. om
  - man vill kunna jämföra objekt på olika sätt.
  - klassen redan implementerar `Comparable` och jämförelsen i `compareTo` görs på ett annat sätt än man vill.
- Interfacet `Comparator` ger oss möjlighet att jämföra objekt av en klass på flera olika sätt.

```
public interface Comparator<T> {
   /**
    * Compares its two arguments for order.
    * Returns a negative integer, zero, or a positive
    * integer as the first argument is less than,
    * equal to, or greater than the second.
    */
   int compare(T e1, T e2);
}
```

Implementera interfacet Comparator

- Antag att personerna i vektorn persons ska sorteras efter namn.
- Vi kan då skriva en klass som implementerar Comparator:

```
public class NameComparator implements Comparator<Person> {
     public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
     }
 }
```

Observera att:

- Metoden compare ligger i en egen klass (och inte i klassen Person). Man måste därför anropa metoder för att få tag på namnen.
- När vi jämför namnen använder vi metoden compareTo i klassen String.

Sortera med hjälp av komparatorobjekt

- För att sortera väljer vi en sort-metod med möjlighet att skicka med ett objekt av typen Comparator:

```
Arrays.sort(persons, new NameComparator());
```

- Inuti sort anropas metoden compare på NameComparator-objektet när två element ska jämföras.

Diskutera

Antag att vi vill sortera personerna i vektorn persons efter *avtagande* idnummer.

```
Arrays.sort(words, new ReversedIdComp());
```

Skriv klart metoden compare så att sorteringen fungerar:

```
public class ReversedIdComp implements Comparator<Person> {
     public int compare(Person p1, Person p2) {




     }
 }
```


Kortare kod med lambdauttryck

- Jämför de två tidigare exemplen med klasser som implementerar interfacet Comparator. Det som skiljer är framför allt hur jämförelsen går till, för övrigt är det bara att följa samma mönster.
- Det vi egentligen vill skicka med som argument till sort är den "kodsnutt" som ska exekveras vid jämförelser.
- Det kan vi göra med hjälp av ett lambdautryck:

```
Arrays.sort(persons,(p1,p2)-> p1.getName().compareTo(p2.getName())
```

Lambdauttryck – kort förklaring

```
Arrays.sort(persons,(p1,p2)-> p1.getName().compareTo(p2.getName()));
```

- I bakgrunden skapas ett objekt (med en metod `compare`) som skickas med som argument till `sort`. Men vi slipper skriva motsvarande klass.
- `p1` och `p2` motsvarar de två parametrarna i `compare`. Efter `->` finns det uttryck vars värde `compare` ska retunera.
- Kompilatorn gissar typen för `p1` och `p2` av sammanhanget. Att det handlar om metoden `compare` kan kompilatorn också lista ut eftersom det är den enda abstrakta metoden i interfacet `Comparator`.
- Lambdauttryck kommer att behandlas utförligare senare under kursen.

Diskutera

Skriv klart anropet av `sort` så att personerna i vektorn `persons` sorteras efter avtagande idnummer.

```
Arrays.sort(persons, (p1, p2) ->
```


Generiska klasser

- Man kan deklarera en eller flera **typparametrar** när man definierar en klass:

```
public class ArrayList<E> {...}
```

  - En typparameters namn brukar bestå av en versal (stor bokstav). `E` som i element, `T` som i type ...

- När man använder klassen ska man ange ett **typargument** (ett klassnamn):

```
ArrayList<Integer> myList = new ArrayList<Integer>();
```


Icke-generisk lista

Risk för fel

- I tidigare versioner av Java fanns inte generik. Istället användes typen `Objekt` som generell typ.
  - Klasen `Object` är superklass till alla andra klasser.
  - En variabel av typen `Object` kan referera till vilka slags objekt som helst.
- Problemet är att man kan blanda olika slags objekt hur som helst i listan.

```
ArrayList list = new ArrayList();
list.add("41");
list.add("42");
list.add(43);   // Ok, men förmodligen fel
```

Icke-generisk lista

`ArrayList`, skiss av implementering

```
public class ArrayList {
    private Object[] data;
    private int size;

    public ArrayList() {
        data = new Object[10];
    }

    public boolean add(Object e) { ... }

    public Object get(int i) {...}
}
```


Generisk lista

`ArrayList<E>`, skiss av implementering

```
public class ArrayList<E> {
    private E[] data;
    private int size;

    public ArrayList() {
        data = (E[]) new Object[10];
    }

    public boolean add(E e) { ... }

    public E get(int i) {...}
}
```

- `<E>` är deklarationen av typparametern `E`.
- När klassen används ersätts E av ett typargument (en riktig klass):

```
ArrayList<Integer> list = new ArrayList<Integer>();
```


Restriktioner för typparametrar

- Typparametrar kan *inte* användas för att skapa objekt:

```
public class SomeClass<E> {
    E e = new E();   // FEL
     ...
```

- En vektor som ska användas *internt* i den generiska klassen kan skapas så här:

```
data = (E[]) new Object[10];
```

(I en metod som ska returnera en vektor bör man istället använda metoden `Array.newInstance`.)

Fördel med generiska klasser

- Undviker fel
  - I en generisk lista med typparameter kan vi bara sätta in ett slags element:

```
ArrayList<String> list = new ArrayList<String>();
list.add("41");
list.add("42");
list.add(43);    // FEL, upptäcks vid kompileringen
```

- Slipper göra explicita typkonverteringar vid anrop av metoder som `get`.
  - Ett anrop av get på listan i exemplet returnerar ett objekt av typen `String`.

```
String s = list.get();
```

  - I den icke-generiska listan returneras ett element av typen `Object`.

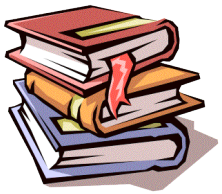
```
String s = (String) list.get();
```

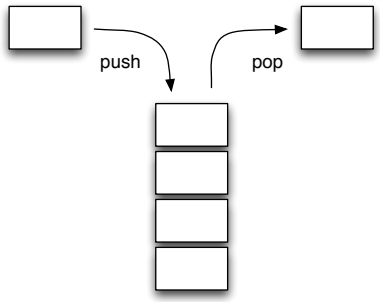
Implementera en generisk klass

Programexempel

Uppgift

Skriv en klass som implementerar den abstrakta datatypen stack.





Vad finns färdigt i Javas klassbibliotek?

I Javas klassbibliotek har man inte infört något interface för ADT:n Stack.

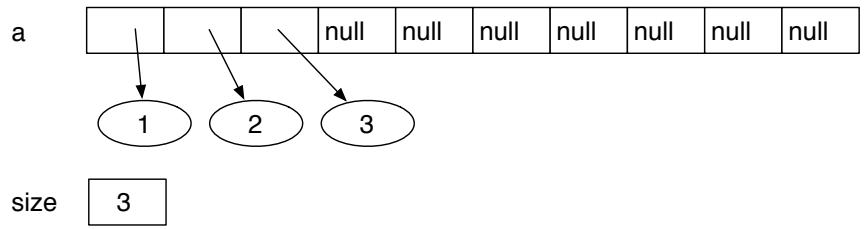
- Det finns en klass `Stack`, som ärver en annan klass `Vector`.
- Interfacet `Deque` i java.util innehåller bl. a. de operationer som ska finnas för en ADT Stack:
  - lägg ett element överst på stacken – `push(e)` eller `addFirst(e)`
  - tag bort och returnera översta elementet – `pop()` eller `removeFirst()`
  - returnera översta elementet – `peek()` eller `peekFirst()`
- Exempel på klasser som implementerar interfacet `Deque` är `ArrayDeque` och `LinkedList`.
- Dessa klasser innehåller många metoder och inte bara de vanliga stack-operationerna. – Lätt att råka anropa fel metod.

Om man behöver en stack-klass

- Använd en av Javas klasser, t.ex. `ArrayDeque` eller `LinkedList`.
  - Fördelar: Testad och färdig klass, bara att använda.
  - Nackdelar: "För många" metoder. Lätt att råka anropa fel metod,t. ex. `addLast` istället för `addFirst`.
- Skriv en egen stackklass.
  - Fördelar: Innehåller bara relevanta metoder. Designad för just ändamålet (kan vara effektivare, minnessnålare, specialiserad...).
  - Nackdelar: Mer jobb, mer kod att hantera (dokumentera, testa, felsöka, begripa ...).

Olika sätt att skriva en stack-klass

- Delegera till en av Javas klasser.
  - Ett attribut av typen `ArrayDequeu` eller eller `LinkedList` håller reda på elementen på stacken.
- Vektor:



- Länkad lista:

Operationer på en stack

Dessa (eller liknande) metoder förväntar man sig i en stack-klass

```
/** Lägger e överst på stacken. */
void push(E e);

/** Tar bort och returnerar översta elementet från stacken. */
E pop();

/** Returnerar översta elementet på stacken. */
E peek();

/** Undersöker om stacken är tom. */
boolean isEmpty();

/** Returnerar antal element i stacken. */
int size();
```


Implementering av stack - delegera till annan klass

- Implementera en egen klass, men **använd internt en annan klass**.

```
public class MultiStack<E> {
    private Deque<E> stack;

    public MultiStack() {
        stack = new ArrayDeque<E>();    // eller LinkedList<E>
    }

    public void push(E e) {
        stack.push(e);
    }

    public E pop() {
        return stack.pop();
    }

    ...
```


Diskutera

- Vad händer om man anropar pop eller peek när stacken är tom?
- Antag att vi lägger till en ny metod:

```
/** Tar bort de k översta elementen. */
public void multiPop(int k) {
   for (int i = 0; i < k; i++) {
      pop();
   }
}
```

Vad ska hända om `k` > antal element i stacken?

Fel i program ("buggar")

Olika slags fel

Kompileringsfel bryter mot språkets grammatik (syntax) eller semantik.
Exempel på syntaxfel: glömt ett {
Exempel på semantiskt fel: glömt deklarera en variabel innan den används.

Exekveringsfel (Runtime errors eller Exceptions) upptäcks vid exekvering.
Exempel: `ArrayIndexOutOfBoundException`, `NullPointerException`, ...

Logiska fel Programmet kan köras men ger fel resultat.

Exekveringsfel

Vad händer vid ett exekveringsfel?

- Ett objekt ("exception") skapas som beskriver typen av fel.
- Programmet avbryts.
- Ett felmeddelande skrivs ut där
  - typ av fel (exception-objektets typ)
  - stacktrace (sekvensen av metodanrop)

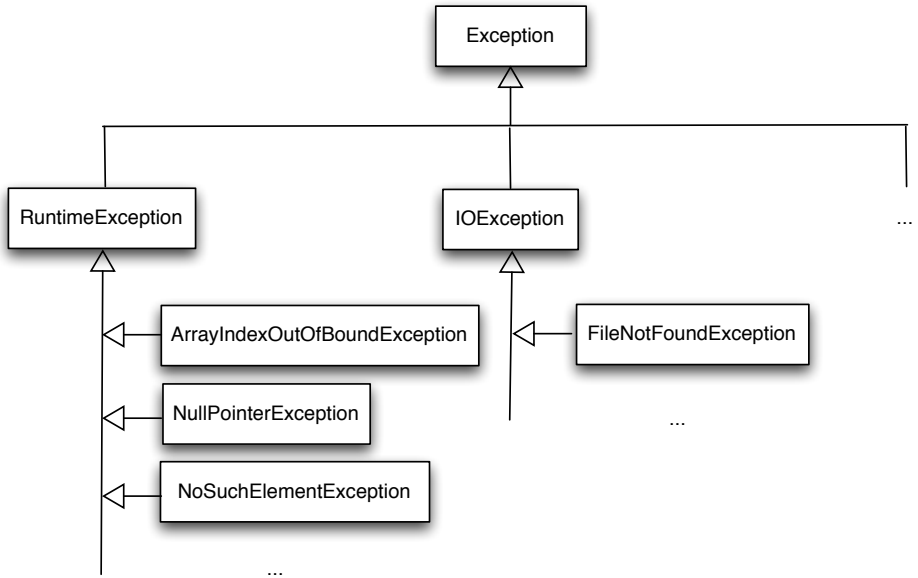
framgår.

Exception

- Betyder undantag
- Exception genereras ("throws") vid exekveringsfel.
- Man kan fånga ("catch") exception och då själv avgöra hur felsituationen ska hanteras.
- Man kan skriva kod som genererar exception inuti en metod.
  - Används om det uppstår en situation som gör det omöjligt för metoden att utföra sin uppgift (t.ex. något argument har ett konstigt värde).

Exception

Olika typer av fel beskrivs av olika subklasser till klassen `Exception`.



Exceptions

Unchecked och checked

Det finns två slag av Exceptions:

- Unchecked Exceptions
  - Subklass till `RuntimeException`.
  - Behöver inte fångas.
  - Exempel: `ArrayIndexOutOfBoundException`, `NullPointerException`
- Checked Exceptions
  - Måste fångas någonstans i anropskedjan, annars kompileringsfel.
  - Exempel: `FileNotFoundException`

Exceptions

Unchecked och checked, forts

- Unchecked Exceptions – används då felet beror på programmeraren
  - Ex: `NullPointerException` eller `ArrayIndexOutOfBoundsException`
- Checked Exceptions – används då felet inte beror på programmeraren
  - Ex: `FileNotFoundException` om man försöker öppna en fil som inte finns

Generera exception

Exempel:

```
throw new IllegalArgumentException("argument < 0");
```

Mönster:

```
throw new ExceptionClass();
throw new ExceptionClass(message);
```

Effekt:

- Ett nytt exception-objekt skapas.
- Exekveringen av metoden avbryts.
- Javasystemet letar efter fångande catch-block.

Generera exception

Exempel

```
void multiPop(int k) {
    if (k > stack.size()) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < k; i++) {
        pop();
    }
}
```


Egna exceptionklasser

- Om man vill kan man implementera en egen exceptionklass (behövs sällan, det finns färdiga exceptionklasser i Javas bibliotek för de flesta situationer).
- Om den ska vara checked:

```
public class SpellException extends Exception {
    ...
}
```

- Om den ska vara unchecked:

```
public class SomeSortOfException extends RuntimeException {
    ...
}
```

Fånga exception
Mönster

```
try {
   // kod som kan generera exception
} catch (ExceptionClass e) {
   // kod för att hantera exception
}

try {
   // kod som kan generera exception
} catch (ExceptionClass1 e1) {
   // kod för att hantera exception av typen ExceptionClass1
} catch (ExceptionClass2 e2) {
   // kod för att hantera exception av typen ExceptionClass2
} finally {
   // kod som utförs efter try-blocket eller efter catch-blocket
}
```


Fånga exceptions
Exempel

- När man anropar en metod som genererar ett exception fångar man normalt det i en try-catch-sats:

```
Scanner scan = null;
try {
   // försöker öppna en fil med namnet fileName
   scan = new Scanner(new File(fileName));
} catch (FileNotFoundException e) {
   System.err.println("Couldn't open file " + fileName);
   System.exit(1);
}
 ... använd scan ...
```

- Om exception inträffar, avbryts exekveringen av satserna i try-blocket och satserna i catch-blocket exekveras.

Fånga exceptions
Forts

- I satsen `catch(Exception e)` kan t.ex. följande metoder användas för att få mer information:
  - `e.printStackTrace();` som skriver ut information om raden där felet inträffat och den/de metodanrop som lett till denna rad.
  - `e.getMessage();` som returnerar en sträng med meddelande om felets art.
- Exempel:

```
Scanner scan = null;
try {
   scan = new Scanner(new File(fileName));
} catch (FileNotFoundException e) {
   e.printStackTrace();
   System.exit(1);
}
 ... använd scan ...
```


Ignorera checked exceptions

- Man kan ignorera en checked Exception och "kasta" det vidare till den anropande metoden.
- I så fall måste man ange det i metodrubriken i den metod där exception genereras:

```
public Scanner createScanner(String fileName) throws
                                    FileNotFoundException {
   // Här genereras exception om filen inte går att öppna
   Scanner scan = new Scanner(new File(fileName));
   return scan;
}
```

- I den metod som anropar `createScanner` måste man ta hand om detta exception och kan korrigera felet på valfritt sätt.

Hantera unchecked exceptions

- Metod som genererar unchecked exception behöver inte ange det i sin rubrik
  - Kan anges i kommentar
- Den som anropar en metod som kan generera en unchecked exception behöver inte (men kan) fånga den i en try-catch-sats
  - Leder till exekveringsfel om de inte fångas.

Dokumentationskommentarer

Exempel

```
/**
 * Removes and returns the element at the top of the stack.
 * @return the element at the top of the stack or null if this
   stack is empty
 */
public E pop() {
    ...
}

 /**
 * Removes the top k elements on the stack.
 * @param k the number of elements to remove
 * @throws IllegalArgumentException if k < 0 or k > number
    elements in the stack
 */
void multiPop(int k) {
    ...
}
```


Diskutera

- Vad betyder `@return`, `@param` och `@throws` i kommentarerna?
- Varför finns de med?

Dokumentationskommentar

En publik metod bör förses med en kommentar på formen `/** ... */`. Det är sedan enkelt att framställa javadoc-filer med information om klassen. Kommentaren ska innehålla:

- Minst en mening (avslutad med punkt) som beskriver vad metoden gör.
- Javadoc-taggarna `@param`, `@return`, `@throws`, ... för att beskriva parametrar, returvärde, om exception genereras, ...

```
/**
 * Removes and returns the element at the top of the stack.
 * @return the element at the top of the stack or null if this
   stack is empty
 */
public E pop() {
    stack.pop();
}
```

Pre- och postvillkor (eng. pre- och postconditions)

Kontrakt på metodnivå

I en metods dokumentationskommentar kan man ange pre- och postvillkor.

- Previllkor är villkor som måste vara uppfyllda för att en metod ska kunna utföra sin uppgift.
  - Ibland finns inga preconditions.
  - När de finns, är de ofta villkor som parametrarna ska uppfylla.
- Postvillkor beskriver hur exekvering av metoden förändrar tillståndet hos objektet.
  - Behöver bara anges om metoden förändrar tillståndet (oftast void-metoder).
  - För metoder som inte är void bör man i stället ge en kommentar om vad som returneras.

Pre- och postvillkor

Exempel

```
/**
 * Removes the top k elements on the stack.
 * pre: k <= the number of elements in the stack
 * post: k elements are removed from the stack
 * @param k the number of elements to remove
 * @throws IllegalArgumentException if  k < 0 or k > number
 * elements in the stack
 */
void multiPop(int k) {
    if (k > stack.size()) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < k; i++) {
        pop();
    }
}
```


Enhetstest

- Enhetstest (eng. unit test) innebär att man testar en mindre del av programmet, t ex en metod eller en klass.
- I kursen ska vi använda ett ramverk, jUnit, för detta.
- På kursens webbsida finns ett PM om jUnit samt länkar till dokumentation.

Testning med JUnit

- Antag att vi ska skriva tester för klassen `MultiStack`. Skriv då en testklass med följande utseende:

```
public class TestMultiStack {
    private MultiStack<Integer> intStack;

    @Before
    public void setUp() throws Exception {
        intStack = new MultiStack<Integer>();
    }

    @After
    public void tearDown() throws Exception {
        intStack = null;
    }
    ...
```

- Metoderna `setUp` och `tearDown` är annoterade med `@Before` respektive `@After`. De anropas av ramverket före, respektive efter, varje test i testklassen.

Testning med JUnit

Testmetod

- För varje test man vill utföra skriver man en metod som annoteras med `@Test`. Dessa metoder kommer att köras av ramverket. Exempel:

```
/**
 * Test if a newly created stack is empty.
 */
@Test
public final void testNewStackIsEmpty() {
   assertTrue(intStack.isEmpty());
   assertEquals(intStack.size(), 0);
}
```

- Testet kommer att ge grönt ljus om `intStack.isEmpty()` returnerar `true` och `intStack.size()` returnerar 0.

Test-first

Antag att du ska implementera en klass

- Skriv testerna för klassens metoder.
- Kör testet och notera att testerna inte går igenom.
- Implementera en metod/del av en metod i taget. Testa!
- Fortsätt så tills alla testerna går igenom.

Exempel på användning av stack

Om ett aritmetiskt heltalsuttryck är skrivet i omvänd polsk notation (postfix notation) kan vi använda en stack för att beräkna uttryckets värde.

- Operatorn placeras efter sina två operander: `10 12 -`
  - Ex: `3 10 12 - *` i postfix notation motsvarar uttrycket `3 * (10 - 12)` i vanlig (infix) notation.
- Fördel: alla uttryck kan skrivas utan parenteser och deras värde kan enkelt evalueras med hjälp av en stack.

Exempel på användning av stack

Algoritm

```
skapa en tom stack s                          // s = new ...
så länge uttrycket ej är slut
   läs in nästa element (tal eller operator)
   om tal
      lägg talet på stacken                   // push
   annars (operator)
      hämta de två översta talen t1 och t2    // 2 st. pop
      från stacken
      res = resultatet av operatorn använd på t2 och t1
      lägg res på stacken                     // push
```

Nu ska stacken bara innehålla ett värde – resultatet.

Exempel på användning av stack

Evaluering av uttrycket: 3 10 12 – *

```
Läst:  inget  3    10    12    -    *

                          12
                    10    10   -2
Stack: tom    3     3     3    3   -6
```

Om uttrycket är korrekt så ligger till sist resultatet som enda element på stacken.

Exempel på vad du ska kunna

- Implementera interfacen `Comparable` och `Comparator`
- Implementera generiska klasser
- Skriva programkod för att fånga exception
- Skriva metoder som genererar exception
- Skriva dokumentationskommentarer (javadoc)
- Formulera testfall för en klass och använda JUnit för att testa klasser

Datorlaboration 1

Abstrakta datatyper

- Räkna hur många gånger olika ord förekommer i en text. – Använd en map för att hålla reda på ord och deras frekvens.
- I programmet ska du använda interface och klasser från Java Collection Framework.
  - Tips! Se Javas dokumentation för nätet för de klasser du använder.
- Innehåll: abstrakta datatyperna lista, mängd, map. Använda generiska klasser.