NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Kontinuasjonseksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdag: Fredag 19. august

Eksamenstid (fra-til): 9.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: C

Kun ”Big Java”, av Cay S. Horstmann, er tillatt.

Annen informasjon:

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av Rune Sætre.

Målform/språk: Bokmål

Antall sider: 3

Antall sider vedlegg: 6

Kontrollert av:

_____

Dato Sign

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig. Hvis du i en del er bedt om å *implementere* klasser og metoder og du ikke klarer det (helt eller delvis), så kan du likevel *bruke* dem i senere deler.

En oversikt over klasser og metoder for alle oppgavene er gitt i vedlegg 3. Kommentarene inneholder krav til de ulike delene, som du må ta hensyn til når du løser oppgavene. I tillegg til metodene som er oppgitt, står du fritt til å definere ekstra metoder for å gjøre løsningen ryddigere. Nyttige standardklasser og -metoder finnes i vedlegg 4.

Del 1 – Teori (15%)

I vedlegg 1 finner du en datamodell som viser de tre klassene **Person**, **Course** og **Exam**.

a) Hva betyr de ulike diagramdelene, altså boksene, strekene/pilene og tegnene (ord, tall og *)?
b) Hvilke valg må en typisk ta når en skal skrive Java-kode for klassene basert slike diagrammer?
c) **Exam**-klassen bruker en **char**-verdi for å representere en eksamenskarakter. Hva er problematisk med dette med tanke på å sikre korrekt bruk av klassen? Skisser kort, med tekst og/eller kode, en alternativ teknikk som løser problemet.

Del 2 – Course- og Exam-klassene (30%)

Course-klassen tilsvarer et undervist emne i et bestemt semester. Kurskoden (**code**) oppgis ved opprettelsen av **Course**-objekter, mens studiepoeng (**credits**) og semesteret (**time**) kan settes senere.

a) Deklarer felt for **time**-egenskapen, og skriv kode for **getYear**, **getSemester**, **getTime** og **setTime** (se vedlegg 3). Med tanke på del 4, så kan det være lurt å lage en metode som sjekker formatet til **setTime** sitt **time**-argument, som kan brukes i andre klasser.
b) **Course**-objekter skal kunne sorteres kronologisk på semesteret de undervises. Forklar hvilke endringer som må gjøres på **Course**-klassen for å kunne bruke Java sin standard mekanisme for sortering, og skriv den nødvendige koden.

Exam-klassen tilsvarer en avlagt eksamen for et undervist emne. Både emnet (**course**) og karakteren (**grade**) oppgis ved opprettelse av **Exam**-objekter.

c) Deklarer felt for **course**- og **grade**-egenskapene, og skriv konstruktøren og **isPass**-metoden (se vedlegg 3).
d) **Exam**-objekter skal også kunne sorteres, men på to måter! Standard-sorteringen skal være kronologisk på semesteret kurset ble undervist, men **Exam**-objektene skal også kunne sorteres på karakter. Forklar hvordan begge sorteringene kan støttes og skriv nødvendig kode.

Del 3 – Person-klassen (20%)

Person-klassen er knyttet til **Course**- og **Exam**-klassene, og tanken er at **Course**-objekter legges til når en person melder seg opp og **Exam**-objekter legges til når eksamen er avlagt. En skal bare kunne ta eksamen i kurs en har meldt seg opp i!

a) Skriv kode for **name**-egenskapen, gitt at den bare skal kunne settes ved opprettelse av **Person**-objekter.
b) Deklarer felt for **courses**-egenskapen, og begrunn valg av type. Skriv kode for **addCourse** og **hasCourse** (se vedlegg 3).
c) Deklarer felt for **exams**-egenskapen, og begrunn valg av type. Skriv kode for **addExam**, **getLastExam** og **hasPassed** (se vedlegg 3).

d) Skriv kode for **countCredits**-metoden i **Person**-klassen. Hust at en ikke får studiepoeng for kurs uten å ha tatt og bestått eksamen, og at det er siste karakter som teller!

Del 4 – IO (10%)

Vedlegg 3 beskriver et tekstformat for informasjon om emner og eksamener.

a) Skriv metoden **Collection<Exam> readExams(Reader input)** i en tenkt **ExamReader**-klasse, som skal opprette **Course**- og **Exam**-objekter tilsvarende teksten lest fra **input**-argumentet, og returnere alle **Exam**-objektene. Det kreves ikke spesifikk håndtering av feil format ut over at metoden ikke skal utløse unntak.
b) Tegn et objektdiagram for objektene som opprettes ved innlesing av eksempelet i vedlegg 2.

Del 5 – ExamRequirement-klassen og IExamRequirement-grensesnittet (25%)

ExamRequirement-klassen (se vedlegg 3) representerer en sjekk for om et **Exam**-objekt, altså en avlagt eksamen, tilfredsstiller visse krav. F.eks. kan en lage et **ExamRequirement**-objekt som sjekker om en har fått minst C i TDT4100. Selve testen gjøres av **accepts**-metoden, som sjekker det angitte **Exam**-argumentet mot verdiene som ligger i **ExamRequirement**-objektet selv.

a) Skriv ferdig konstruktør nr. 2, og gjør den så kort og enkel som mulig.
b) **accepts**-metoden m/hjelpemetoden **acceptsCourse** er ferdigskrevet, men koden inneholder (minst) tre feil. Skriv korrekt kode.
c) Nederst i klassen er feltet **atLeastCInTdt4100** deklarert. Med koden **atLeastCInTdt4100.accepts(**...**)** skal en kunne sjekke om en har minst C i TDT4100. Skriv ferdig deklarasjonen, både modifikator(er) og initialiseringsuttrykket.
d) Du trenger å sjekke om en eksamen er for et masteremne på IDI, som har kode som begynner på "TDT42", f.eks. TDT4250. Forklar hvordan du kan bruke arv for å organisere koden og hvordan eksisterende kode evt. må endres.
e) Skriv kode for et *funksjonelt* grensesnitt **IExamRequirement**, som **ExamRequirement** (allerede implisitt) implementerer. Forklar hvorfor et slikt grensesnitt kan være nyttig, fremfor å bare ha **ExamRequirement**-klassen.
f) Skriv en *alternativ* deklarasjon av **atLeastCInTdt4100** som har **IExamRequirement** som type og utnytter Java 8 sin funksjonssyntaks i initialiseringsuttrykket.

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of computer and information science

Continuation examination paper for TDT4100 Object-oriented programming with Java

Academic contact during examination: Hallvard Trætteberg

Phone: 918 97 263

Examination date: 11. June

Examination time (from-to): 9:00-13:00

Permitted examination support material: C

Only ”Big Java”, by Cay S. Horstmann, is allowed.

Other information:

This examination paper is written by teacher Hallvard Trætteberg, with quality assurance by Rune Sætre.

Language: English

Number of pages: 3

Number of pages enclosed: 6

Checked by:

_____

Date Signature

If you feel necessary information is missing, state the assumptions you find it necessary to make. If you are not able to *implement* classes and method that a part asks for, you may still *use* these classes and methods later.

An overview of classes and methods for all the parts are provided in appendix 3. The comments contain requirements for the various programming tasks, that must be considered when you solve them. Feel free to define extra methods, in addition to those provided, to make your solution tidier. Useful standard classes and methods can be found in appendix 4.

Part 1 – Theory (15%)

In appendix 1 you'll find a data model that shows the three classes **Person**, **Course** og **Exam**.

a) What do the various diagram elements mean, i.e. the boxes, lines/arrows and text (words, numbers and *)?
b) What choices do you typically need to make when writing Java code for the classes in such diagrams?
c) The **Exam** class uses a **char** value to represent an exam grade. How is this problematic considering correct usage of the class? Sketch briefly, with text and/or code, an alternative technique that solves the problem.

Part 2 – The Course and Exam classes (30%)

The **Course** class corresponds to a taught topic in a specific semester. The course code (**code**) is provided when creating **Course** objects, while credits (**credits**) and semester (**time**) can be set later.

a) Declare fields for the **time** property, and write code for **getYear**, **getSemester**, **getTime** and **setTime** (see appendix 3). Considering part 4, it can be smart to write a method that checks the format of **setTime**'s **time** argument, that can be used in other classes.
b) It must be possible to sort **Course** objects chronologically on the semester they are taught. Explain what changes must be made to the Course class to be able to use Java's standard mechanism for sorting, and write the necessary code..

The **Exam** class corresponds to completed exams for a course. Both the course (**course**) and grade (**grade**) are provided when creating **Exam** objects.

c) Declare fields for the **course** and **grade** properties, and write the constructor and **isPass** method (see appendix 3).
d) It should also be possible to sort **Exam** objects, but in two different orders! The standard sorting order should be chronological on the semester the course was taught, but the **Exam** objects should also be sortable on the grade. Explain how both sorting orders can be supported, and write the necessary code.

Part 3 – The Person class (20%)

The **Person** class is related to the **Course** and **Exam** classes, and the idea is that **Course** objects are added when a person registers for a course and **Exam** objects are added when the exam is completed. You can only take exams for courses you are registered for!

a) Write code for the **name** property, given that it only can be set when creating **Person** objects.
b) Declare the field for the **courses** property, and explain your choice of type. Write code for **addCourse** and **hasCourse** (see appendix 3).

c) Declare the field for the **exams** property, and explain your choice of type. Write code for **addExam**, **getLastExam** and **hasPassed** (see appendix 3).
d) Write code for the **countCredits** method in the **Person** class. Remember you only get credits for courses you have taken and passed, and that the last exam is the one that counts for a topic!

Part 4 – IO (10%)

Appendix 2 specifies a text format for information about courses and exams.

a) Write the method **Collection<Exam> readExams(Reader input)** in a hypothetical **ExamReader** class, that should create **Course** and **Exam** objects corresponding to the text read from the **input** argument, and return all the **Exam** objects. No special handling of format errors is required except that the method should not throw exceptions.
b) Draw an object diagram for the objects that are created when reading the example in appendix 2.

Part 5 – The ExamRequirement class and IExamRequirement interface (25%)

The **ExamRequirement** class (see appendix 3) represents a check of whether an **Exam** object, i.e. a completed exam, satisfies certain requirements. E.g. you can create an **ExamRequirement** object that checks if you have at least a C in TDT4100. The check itself is performed by the **accepts** method, that checks the provided **Exam** argument against the values in the **ExamRequirement** object itself.

a) Complete the second constructor, making it as short and simple as possible.
b) The **accepts** method and helper method **acceptsCourse** are completed, but the code includes (at least) three bugs. Correct the code.
c) At the bottom of the class the **atLeastCInTdt4100** field is declared. Using the code **atLeastCInTdt4100.accepts(**...**)** you should be able to check that you have at least a C in TDT4100. Complete the declaration, both modifier(s) and initialising expression.
d) You need to check that an exam is for a master course at IDI, that has a code starting on ”TDT42”, like TDT4250. Explain how you can use inheritance to organise the code and how, if necessary, existing code must be changed.
e) Write code for a *functional* interface **IExamRequirement**, that **ExamRequirement** (already implicitly) implements. Explain why such an interface is useful, instead of just having the **ExamRequirement** class.
f) Write an *alternative* declaration for **atLeastCInTdt4100** that has **IExamRequirement** as its type and utilises Java 8’s function syntax in the initialising expression.

NTNU
Kunnskap for ei betre verd

Institutt for datateknikk og informasjonsvitskap

Kontinuasjonseksamensoppgåve i TDT4100 Objektorientert programmering

Fagleg kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdato: 11. juni

Eksamenstid (frå-til): 9.00-13.00

Hjelpemiddelkode/Tillatne hjelpemiddel: C

Berre ”Big Java”, av Cay S. Horstmann, er tillaten.

Annan informasjon:

Oppgåva er utarbeidd av faglærar Hallvard Trætteberg og kvalitetssikra av Rune Sætre.

Målform/språk: Nynorsk

Sidetal: 3

Sidetal vedlegg: 6

Kontrollert av:

_____

Dato Sign

Om du meiner at opplysningar manglar i ei oppgåveformulering, gjer kort greie for dei føresetnadene som du finn naudsynte. Om du i ein del er beden om å implementere klasser og metodar og du ikkje klarer det (heilt eller delvis), så kan du likevel nytte dei i seinare delar.

Eit oversyn over klasser og metodar for alle oppgåver er gjeve i vedlegg 3. Kommentarane inneheld krav til dei ulike delane, som du må ta omsyn til når du løyser oppgåva. I tillegg til metodane som er gjevne, står du fritt til å definere ekstra metodar for å gjere løysinga ryddigare. Nyttige standardklasser og -metodar finst i vedlegg 4.

Del 1 – Teori (15%)

I vedlegg 1 finn du ein datamodell som syner dei tre klassane **Person**, **Course** og **Exam**.

a) Kva tydar dei ulike diagramdelane, altså boksane, strekane/pilane og teikna (ord, tall og *)?
b) Kva for val må ein typisk ta når ein skal skrive Java-kode for klassane basert slike diagrammar?
c) **Exam**-klassen nyttar ein **char**-verdi for å representere ein eksamenskarakter. Kva er problematisk med dette med tanke på å sikre korrekt bruk av klassen? Skisser kort, med tekst og/eller kode, ein alternativ teknikk som løsar problemet.

Del 2 – Course- og Exam-klassene (30%)

Course-klassen svarar til eit undervist emne i eit bestemt semester. Kurskoden (**code**) vert gjeven når **Course**-objekt lagast, medan studiepoeng (**credits**) og semesteret (**time**) kan setjast seinare.

a) Deklarer felt for **time**-eigenskapen, og skriv kode for **getYear**, **getSemester**, **getTime** og **setTime** (sjå vedlegg 3). Med tanke på del 4, så kan det vere lurt å lage ein metode som sjekkar formatet til **setTime** sitt **time**-argument, som kan nyttast i andre klassar.
b) **Course**-objekt skal kunne sorterast kronologisk på semesteret dei vert undervist i. Forklar kva for endringar som må gjerast på **Course**-klassen for å kunne bruke Java sin standard mekanisme for sortering, og skriv den naudsynte koden.

Exam-klassen svarer til ein avlagd eksamen for eit undervist emne. Både emnet (**course**) og karakteren (**grade**) vert gjevne når **Exam**-objekt lagast.

c) Deklarer felt for **course**- og **grade**-eigenskapane, og skriv konstruktøren og **isPass**-metoden (sjå vedlegg 3).
d) **Exam**-objekter skal også kunne sorterast, men på to måtar! Standard-sorteringa skal vere kronologisk på semesteret kurset vert undervist, men **Exam**-objekta skal også kunne sorterast på karakter. Forklar korleis begge sorteringane kan stødast og skriv naudsynt kode.

Del 3 – Person-klassen (20%)

Person-klassen er knytt til **Course**- og **Exam**-klassane, og tanken er at **Course**-objekt leggjast til når ein person meldar seg opp og **Exam**-objekt leggjast til når eksamen er avlagt. Ein skal berre kunne ta eksamen i kurs ein har meldt seg opp i!

a) Skriv kode for **name**-eigenskapen, gjeven at den berre skal kunne setjast når **Person**-objekt lagast.
b) Deklarer felt for **courses**-eigenskapen, og grunngje val av type. Skriv kode for **addCourse** og **hasCourse** (sjå vedlegg 3).
c) Deklarer felt for **exams**-eigenskapen, og grunngje val av type. Skriv kode for **addExam**, **getLastExam** og **hasPassed** (sjå vedlegg 3).

d) Skriv kode for **countCredits**-metoden i **Person**-klassen. Hugs at ein ikkje får studiepoeng for kurs utan å ha teke og stått på eksamen, og at det er siste karakter som tel!

Del 4 – IO (10%)

Vedlegg 2 beskriv eit tekstformat for informasjon om emne og eksamenar.

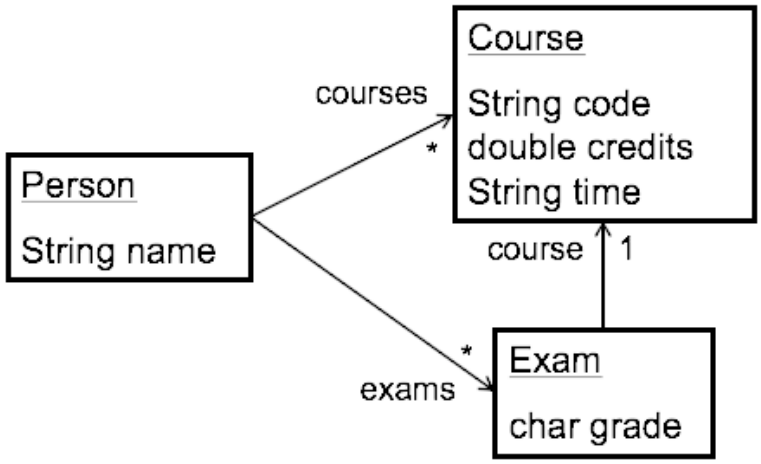
a) Skriv metoden **Collection<Exam> readExams(Reader input)** i ein tenkt **ExamReader**-klasse, som skal opprette **Course**- og **Exam**-objekt som svarar til teksten lest frå **input**-argumentet, og returnere alle **Exam**-objekta. Det krevjast ikkje spesifikk handtering av feil format ut over at metoden ikkje skal løyse ut unnatak.
b) Teikn eit objektdiagram for objekta som vert oppretta når eksempelet i vedlegg 2 lesast inn.

Del 5 – ExamRequirement-klassen og IExamRequirement-grensesnittet (25%)

ExamRequirement-klassen (sjå vedlegg 3) representerer ein sjekk for om eit **Exam**-objekt, altså ein avlagt eksamen, tilfredsstiller visse krav. T.d. kan ein lage eit **ExamRequirement**-objekt som sjekkar om ein har fått minst C i TDT4100. Sjølve testen gjerast av **accepts**-metoden, som sjekkar det gjevne **Exam**-argumentet mot verdiane som ligg i **ExamRequirement**-objektet sjølve.

a) Skriv ferdig konstruktør nr. 2, og gjer den så kort og enkel som mogleg.
b) **accepts**-metoden m/hjelpemetoden **acceptsCourse** er skriven ferdig, men koden inneheld (minst) tre feil. Skriv korrekt kode.
c) Nedst i klassen er feltet **atLeastCInTdt4100** deklarert. Med koden **atLeastCInTdt4100.accepts(**...**)** skal ein kunne sjekke om ein har minst C i TDT4100. Skriv ferdig deklarasjonen, både modifikator(ar) og initialiseringsuttrykket.
d) Du treng å sjekke om ein eksamen er for eit masteremne på IDI, som har kode som byrjar på ”TDT42”, t.d. TDT4250. Forklar korleis du kan bruke arv for å organisere koden og korleis eksisterande kode evt. må endrast.
e) Skriv kode for eit *funksjonelt* grensesnitt **IExamRequirement**, som **ExamRequirement** (allereie implisitt) implementerer. Forklar kvifor eit slikt grensesnitt kan vere nyttig, framfor å berre ha **ExamRequirement**-klassen.
f) Skriv en *alternativ* deklarasjon av **atLeastCInTdt4100** som har **IExamRequirement** som type og utnyttar Java 8 sin funksjonssyntaks i initialiseringsuttrykket.

Appendix 1: Class overview



Appendix 2: File format for exam results

The file format is line-based and has two kinds of lines:

1) *Semester lines* contain only one item, which is a valid time of the format used by **Course**'s **setTime** method. A semester line may be followed by zero, one or more *course lines*.
2) *Course lines* contain a course code followed by a credit value and at least one exam grade for that course. A single space is used as separator. All the courses are taught in the semester indicated in the previous *semester line*.

Example of the text format. The left column shows the text contents, and the right column explains the meaning:

`S2016`
`TDT4100 7.5 F C`
`S2017`
`TDT4100 7.5 A`
	Semester line for the Spring 2016 semester.
	Course line for TDT4100, with grades F and C.
	Semester line for the Spring 2017 semester.
	Course line for TDT4100, with grade A!

Appendix 3: Provided code (fragments)

```
public class Course {

      ... fields, constructors and methods for code and credits ...

      public int getYear() {  ... }

      public char getSemester() { ... }

      /**
       * Gets the time this Course is given, in the format <semester><year>
       * E.g. if the semester is 'S' and the year is 2016,
       * it should return S2016.
       */
      public String getTime() { ... }

      /**
       * Sets the time that this Course is taught. The format is the semester
       * (char) followed by the year. The year may be shortened to two digits;
       * if it is below 50 then 2000 should be added, otherwise 1900.
       * E.g. S16 means Spring 2016, while F86 means Fall 1986.
       * @param time The time in the format <semester><year>
       * @throws IllegalArgumentException if the format is incorrect
       */
      public void setTime(String time) {
            ...
      }
}

public class Exam {

      ... fields and methods for course and grade ...

      /**
       * Initialises an Exam, by setting the course and grade.
       * The grade can only be set to one of the characters 'A'-'F'.
       * @throws IllegalArgumentException if the grade is not legal
       */
      public Exam(...) { ... }

      /**
       * Tells whether this Exam has a result that is a passing grade.
       */
      public boolean isPass() { ... }
}
```

```java
public class Person {

    ... fields, constructors and methods for name ...

    /**
     * Adds a Course to this Person, if no Course is registered
     * for the same code, year and semester.
     * @param course
     * @return true if the course was added, false otherwise
     */
    public boolean addCourse(Course course) { ... }

    /**
     * Returns whether this Person has a Course with the given code.
     * @param code
     */
    public boolean hasCourse(String code) { ... }

    /**
     * Creates and adds an exam to this Person for the provided Course and
     * with the provided grade, but only if this Person has this Course and
     * no passing Exam is registered for that Course.
     * @param course
     * @param grade
     * @return the newly created and added Exam, or null
     */
    public Exam addExam(Course course, char grade) { ... }

    /**
     * Gets the exam that was registered last for the provided course code.
     * This is the exam that counts for that course!
     * @param course
     */
    public Exam getLastExam(String code) { ... }

    /**
     * Returns true of this Person has passed the Course for the provided code.
     * @param code
     */
    public boolean hasPassed(String code) { ... }

    /**
     * Counts the credits this Person has earned.
     */
    public double countCredits() { ... }
}
```

```java
/**
 * Represents a requirement concerning an Exam for a specific Course.
 * The Exam's result must not be before the provided year (sinceYear) and
 * the grade must not be worse than the provided grade (minGrade).
 */
public class ExamRequirement {

	public final String course;
	public final int sinceYear;
	public final char minGrade;

	public ExamRequirement(String course, int sinceYear, char minGrade) {
		this.course = course;
		this.sinceYear = sinceYear;
		this.minGrade = minGrade;
	}

	/**
	 * Initialises with the course and year with provided data and
	 * makes sure a passing grade is required.
	 * @param course
	 * @param sinceYear
	 */
	public ExamRequirement(String course, int sinceYear) { ... }

	/**
	 * Helper method for checking the course
	 * @param course
	 */
	private boolean acceptsCourse(Course course) {
		return this.course == course && sinceYear > course.getYear();
	}

	/**
	 * Returns true if the provided Exam is for the specified course,
	 * not before the specified year and not worse than the specified grade.
	 * @param exam
	 * @return
	 */
	public boolean accepts(Exam exam) {
		return acceptsCourse(exam.getCourse()) && exam.getGrade() >=
minGrade;
	}

	// declares atLeastCInTdt4100
	... ExamRequirement atLeastCInTdt4100 = ...;
}
```

Appendix 4: Useful standard classes and methods

public interface Collection<E> extends Iterable<E>

boolean	**add**(**E** e) Ensures that this collection contains the specified element.
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection.
void	**clear**() Removes all of the elements from this collection.
boolean	**contains**(**Object** o) Returns true if this collection contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this collection, if it is present.
boolean	**removeAll**(**Collection**<?> c) Removes all of this collection's elements that are also contained in the specified collection.
boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this collection that are contained in the specified collection.
int	**size**() Returns the number of elements in this collection.
Stream<**E**>	**stream**() Returns a sequential Stream with this collection as its source.

interface List<E> extends Collection<E>

void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list at the specified position.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if it does not contain the element.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if it does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list.
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element.
Void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.

interface Map<K,V>

void	**clear**() Removes all of the mappings from this map.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for t key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map.
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map.
V	**remove**(**Object** key) Removes the mapping for a key from this map if it is present.
int	**size**() Returns the number of key-value mappings in this map.

interface Stream<T>

boolean	**allMatch**(**Predicate**<? super **T**> predicate) Returns whether all elements of this stream match the provided predica
boolean	**anyMatch**(**Predicate**<? super **T**> predicate) Returns whether any elements of this stream match the provided predicate.
<R,A> R	**collect**(**Collector**<? super **T**,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
Stream<**T**>	**filter**(**Predicate**<? super **T**> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
void	**forEach**(**Consumer**<? super **T**> action) Performs an action for each element of this stream.
Stream<R>	**map**(**Function**<? super **T**,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.

T	**reduce**(**T** identity, **BinaryOperator**<**T**> accumulator) Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.
Stream<**T**>	**sorted**() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<**T**>	**sorted**(**Comparator**<? super **T**> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.

class String implements Comparable<String>

char	**charAt**(int index) Returns the char value at the specified index.
boolean	**contains**(**String** s) Returns true if and only if this string contains the specified string.
boolean	**endsWith**(**String** suffix) Tests if this string ends with the specified suffix.
static **String**	**format**(**String** format, **Object**... args) Returns a formatted string using the specified format string and arguments.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified characte starting the search at the specified index.
int	**indexOf**(**String** str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(**String** str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
boolean	**isEmpty**() Returns true if, and only if, **length()** is 0.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(**String** str) Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
boolean	**regionMatches**(int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar) Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replace**(**String** target, **String** replacement) Replaces each substring of this string that matches the literal target string with the specified literal replacement string.
String[]	**split**(**String** regex) Splits this string around matches of the given **regular expression**.
boolean	**startsWith**(**String** prefix) Tests if this string starts with the specified prefix.
String	**substring**(int beginIndex) Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a string that is a substring of this string.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a string whose value is this string, with any leading and trailing whitespace removed.

class Scanner

Scanner(**InputStream** source)	
Constructs a new Scanner that produces values scanned from the specified input stream.	
void	**close**() Closes this scanner.
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNextBoolean**() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
String	**next**() Finds and returns the next complete token from this scanner.
boolean	**nextBoolean**() Scans the next token of the input into a boolean value and returns that value.
double	**nextDouble**() Scans the next token of the input as a double.
int	**nextInt**() Scans the next token of the input as an int.
String	**nextLine**() Advances this scanner past the current line and returns the input that was skipped.