NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdag: Tirsdag 16. mai

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 Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Bokmål

Antall sider: 3

Antall sider vedlegg: 6 (4+2)

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 1. 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 2.

Temaet for oppgaven er et spisested (**Diner**) og problemstillingen er plassering (**Seating**) av grupper (**Group**) av gjester ved bordene (**Table**).

Del 1 – Group, Table- og Seating-klassene (15%)

Group-, **Table**- og **Seating**-klassene (vedlegg 1) er såkalte verdi-klasser, med data som skal oppgis ved opprettelsen av objektene og siden ikke skal kunne endres. **Group** skal inneholde data om antall gjestar i gruppa, **Table** skal inneholde data om antall sitteplasser (capacity) og **Seating** skal holde rede på bordet en gitt gruppe sitter på.

a) Skriv ferdig **Group** og **Seating**, inkludert metoder nødvendig for innkapsling.

b) En skal ikke kunne ha **Seating**-objekter for bord som ikke har mange nok sitteplasser til hele gruppa som er plassert der. Skriv koden som trengs for å sikre at denne regelen overholdes.

c) Anta at **Group** hadde en metode for å endre antall gjester. Forklar med tekst og/eller kode hvilke endringer du måtte gjort for å sikre at regelen i b) overholdes.

d) I tillegg til antall sitteplasser, skal et bord ha et bordnummer. Dette skal være et unikt løpenummer som ikke oppgis, men settes automatisk av kode i **Table**-klassen selv når **Table**-objekter opprettes. Det aller første bordet som lages skal få 1 som nummer, det andre skal få 2 osv. Implementer konstruktøren og annen nødvendig kode, inkludert **getNum**-metoden!

Del 2 – Diner-klassen (40%)

Diner-klassen (vedlegg 1) holder rede på bord (tables) og bordplasseringer (seatings), altså hvilke grupper som sitter ved hvilke bord.

a) Skriv nødvendige felt-deklarasjoner og konstruktør(er), gitt at spisestedet har mer enn ett bord. Skriv også metodene for å legge til og fjerne bord.

b) Skriv metodene **isOccupied** og **getCapacity**.

c) Bord kan settes sammen, typisk for å få plass til store grupper med gjester. Tilsvarende kan bord deles opp, for å unngå at en liten gruppe tar opp et stort bord. Skriv metodene **mergeTables** og **splitTable**. I denne omgang skal det ikke registreres hvilke bord som faktisk settes sammen, de forsvinner bare, og må opprettes på nytt ved oppdeling.

d) Tegn et objekttilstandsdiagram som illustrerer virkemåten til **mergeTables**.

e) Når gjester skal plasseres må en finne det minste, ledige bordet med nok kapasitet. Skriv metodene **hasCapacity** og **findAvailableTables**. Skriv også annen nødvendig kode for å sikre at returverdien til **findAvailableTables** er sortert.

f) En ny bordplassering registreres i et **Seating**-objekt. Skriv metodene **createSeating**, **addSeating** og **removeSeating**.

Del 3 – Table-, SimpleTable- og CompositeTable-klassene (15%)

Et problem ved sammensetting og oppdeling av bord er at bordnummeringen blir gal, når logisk sett samme bord opprettes på nytt og får et annet nummer. En måte å håndtere det på er å ha to bord-typer, enkeltbord (**SimpleTable**) og sammensatt bord (**CompositeTable**), hvor sistnevnte holder rede på hvilke bord som er satt sammen. Dette krever en ny versjon av **mergeTable**-metoden, som altså må opprette et **CompositeTable** som inneholder de to bordene som settes sammen, og **splitTable**-metoden må skrives om så det deler opp et **CompositeTable** i de *samme* to bordene som ble satt sammen. **splitTable**-metoden trenger da ikke lenger de to **capacity**-argumentene fordi de to bordene jo vet sin kapasitet.

a) Forklar med tekst og/eller kode hvordan du vil bruke arv og/eller grensesnitt, slik at **Table** fortsatt kan brukes som generell bord-type og **SimpleTable** og **CompositeTable** kan håndtere hver sine spesialtilfeller. Forklar også virkemåten til **SimpleTable** og **CompositeTable**.

b) Skriv nye versjoner av **Diner** sine **mergeTable**- og **splitTable**-metoder. Merk at den nye **splitTable**-metoden kun skal ta inn en **CompositeTable** som argument.

Del 4 – GuestManager-klassen (20%)

Gjester på en **Diner** tas i mot av en tilhørende **GuestManager** (se vedlegg 1), som prøver å plassere dem. Dersom det ikke går, så må de vente på at et bord med nok kapasitet blir ledig. **GuestManager** vil altså ha behov for å følge med på hvordan kapasiteten til **Diner**-objektet endres. Dette gjøres ved å gjøre **Diner** sin kapasitet-egenskapen, som returneres av et kall til **getCapacity(false)**, *observerbar*.

a) Hva innebærer observerbarhet? Forklar kort med tekst og/eller kode hvordan en gjør en (egenskap i en) klasse observerbar.

b) Forklar med tekst og/eller kode hvordan du vil endre **Diner** slik at **GuestManager** kan lytte til endringer i capacity-egenskapen (ved å implementere **CapacityListener**-grensesnittet).

c) Forklar med tekst og/eller kode hvordan du vil skrive **GuestManager**-klassen. Vi forventer ikke automatisk sammenslåing eller oppdeling av bord, men de som kom først skal helst få bord først.

Del 5 – Diverse (10%)

a) Er **CapacityListener** et *funksjonelt* grensesnitt? Begrunn svaret!

b) Skriv (om) en av **isOccupied** eller **getCapacity** i **Diner** slik at den bruker **Stream**-teknikken og Java 8 sin funksjonssyntaks (hvis du ikke har gjort det fra før, da!).

c) Forklar med tekst og/eller kode hvordan du vil teste **isOccupied**-metoden til **Diner** i en separat **DinerTest**-klasse, og nevn hvilke metoder i Diner du vil bruke og evt. hvordan **Diner** må endres for at **isOccupied** skal være enkelt testbar.

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of computer and information science

Examination paper for TDT4100
Object-oriented programming with Java

Academic contact during examination: Hallvard Trætteberg

Phone: 918 97 263

Examination date: Tuesday 16. May

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 Ragnhild Kobro Runde (Ifi, UiO).

Language: English

Number of pages: 3

Number of pages enclosed: 6 (4+2)

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 1. 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 2.

The topic is a diner (**Diner**) and the problem is seating (**Seating**) groups (**Group**) of guests at the tables (**Table**).

Part 1 – The Group, Table and Seating classes (15%)

The **Group**, **Table** and **Seating** classes (appendix 1) are so-called value classes, with data that must be provided when objects are created and cannot be changed later. **Group** must contain data about the number of guests in the group, **Table** must contain data about the number of seats (capacity) and **Seating** must keep track of the table a group is seated at.

a) Finish the **Group** and **Seating** classes, including necessary encapsulation methods.

b) It should not be possible to have **Seating** objects for tables without enough seats for the whole group seated there. Write the code needed for enforcing this rule.

c) Assume **Group** had a method for changing the number of guests. Explain with text and/or code what changes you would need to enforce the rule in b).

d) In addition to the number of seats, a table must have a table number. This number must be a unique counter that is not provided, but is automatically set by code in the **Table** class itself when **Table** objects are created. The very first table that is created should have number 1, the second have number 2, and so forth. Implement the constructor and other necessary code, including the **getNum** method!

Part 2 – The Diner class (40%)

The **Diner** class (appendix 1) keeps track of tables and seatings, i.e. which groups are seated at which tables.

a) Write the necessary field declarations and constructor(s), given that the diner has more than one table. Also write the methods for adding and removing tables.

b) Write the **isOccupied** and **getCapacity** methods.

c) Tables can be merged, typically to make room for large groups of guests. Tables can correspondingly be split, to avoid that a small group occupies a large table. Write the **mergeTables** and **splitTable** methods. At this point, you don't need to represent which tables are actually merged, they just disappear, and must be re-created when split.

d) Draw an object state diagram that illustrates the behavior of **mergeTables**.

e) When guests are seated you must find the smallest, available table with enough capacity. Write the **hasCapacity** and **findAvailableTables** methods. Also write other code necessary for ensuring return value of **findAvailableTables** is sorted.

f) A new seating of guests is registered in a **Seating** object. Write the **createSeating**, **addSeating** and **removeSeating** methods.

Part 3 – The Table, SimpleTable and CompositeTable classes (15%)

A problem when merging and splitting tables is that the table numbering becomes wrong, when the logically same table is re-created and is assigned a different number. One way of handling this is to have two table types, simple tables (**SimpleTable**) and composite tables (**CompositeTable**), where the latter keeps track of the tables that are merged. This requires a new version of the **mergeTable** method that must create a **CompositeTable** containing the two tables that are merged, and the **splitTable** method must be re-written so it splits the a **CompositeTable** into the *same* two tables that were merged. The **splitTable** method does not need the two **capacity** arguments any more because the tables know their capacity.

a) Explain with text and/or code how who will use inheritance and/or interfaces, so **Table** still can be used as a general table type and **SimpleTable** and **CompositeTable** can handle respective special cases. Also explain the behavior of **SimpleTable** and **CompositeTable**.

b) Write new versions of **Diner**'s **mergeTable** and **splitTable** methods. Note that the new **splitTable** method only takes a **CompositeTable** argument.

Part 4 – The GuestManager class (20%)

Guests arriving at a **Diner** are received by a corresponding **GuestManager** (see appendix 1), that tries to seat them. If it fails, the guests must wait for a table with enough seats to become available. Hence, **GuestManager** needs to track how the capacity of the **Diner** object changes. This is done by making **Diner**'s capacity property, as returned by a call to **getCapacity(false)**, *observable*.

a) What does observability entail? Briefly explain with text and/or code how to make a (property of a) class observable.

b) Explain with text and/or code how you would modify **Diner** so **GuestManager** can listen to changes to the capacity property (by implementing the **CapacityListener** interface).

c) Explain with text and/or code how you would write the **GuestManager** class. We don't expect automatically merging and splitting of tables, but those arriving first should preferably be seated first.

Part 5 – Misc. (10%)

a) Is **CapacityListener** a *functional* interface? Explain your answer!

b) (Re)write one of **isOccupied** or **getCapacity** in **Diner** so it uses the **Stream** technique and the function syntax of Java 8 (if you haven't already, that is!).

c) Explain with text and/or code how you would test the **isOccupied** method in **Diner** in a separate **DinerTest** class, and mention which methods in Diner you would use and how **Diner** if necessary must be modified for **isOccupied** to be easily testable.

NTNU
Kunnskap for ei betre verd

Institutt for datateknikk og informasjonsvitskap

Eksamensoppgåve i TDT4100
Objektorientert programmering

Fagleg kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdato: 16. mai

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 Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Nynorsk

Sidetal: 3

Sidetal vedlegg: 6 (4+2)

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 1. 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 2.

Temaet for oppgåva er ein spisestad (**Diner**) og problemstillinga er plassering (**Seating**) av grupper (**Group**) av gjestar ved bordane (**Table**).

Del 1 – Group, Table- og Seating-klassene (15%)

Group-, **Table**- og **Seating**-klassene (vedlegg 1) er såkalla verdi-klasser, med data som skal gis inn ved oppretting av objekta og sidan ikkje skal kunne endrast. **Group** skal innehalde data om talet på gjester i gruppa, **Table** skal innehalde data om talet på sitteplassar (capacity) og **Seating** skal halde reidde på bordet ein gitt gruppe sit på.

a) Skriv ferdig **Group** og **Seating**, inkludert metodar naudsynte for innkapsling.

b) Ein skal ikkje kunne ha **Seating**-objekt for bord som ikkje har mange nok sitteplassar til heile gruppa som er plassert der. Skriv koden som trengst for å sikre at denne regelen overhaldast.

c) Anta at **Group** hadde ein metode for å endre talet på gjestar. Forklar med tekst og/eller kode kva for endringar du måtte gjort for å sikre at regelen i b) overhaldast.

d) I tillegg til talet på sitteplassar, skal eit bord ha eit bordnummer. Dette skal vere eit unikt løpenummer som ikkje verte oppgitt, men setjast automatisk av kode i **Table**-klassen sjølv når **Table**-objekt opprettast. Det aller første bordet som lagast skal få 1 som nummer, det andre skal få 2 osb. Implementer konstruktøren og annan naudsynt kode, inkludert **getNum**-metoden!

Del 2 – Diner-klassen (40%)

Diner-klassen (vedlegg 1) held reidde på bord (tables) og bordplasseringar (seatings), altså kva for grupper som sit ved kva for bord.

g) Skriv naudsynte felt-deklarasjonar og konstruktør(er), gitt at spisestaden har meir enn eitt bord. Skriv også metodane for å leggje til og fjerne bord.

h) Skriv metodane **isOccupied** og **getCapacity**.

i) Bord kan setjast saman, typisk for å få plass til store grupper med gjestar. Tilsvarande kan bord delast opp, for å unngå at ein liten gruppe tek opp eit stort bord. Skriv metodane **mergeTables** og **splitTable**. I denne omgang skal det ikkje registrerast kva for bord som faktisk setjast saman, dei forsvinn berre, og må opprettast på ny ved oppdeling.

j) Teikn eit objekttilstandsdiagram som illustrerer verkemåten til **mergeTables**.

k) Når gjester skal plasserast må ein finne det minste, ledige bordet med nok kapasitet. Skriv metodane **hasCapacity** og **findAvailableTables**. Skriv og annan naudsynt kode for å sikre at returverdien til **findAvailableTables** er sortert.

l) Ein ny bordplassering registrerast i eit **Seating**-objekt. Skriv metodane **createSeating**, **addSeating** og **removeSeating**.

Del 3 – Table-, SimpleTable- og CompositeTable-klassene (15%)

Eit problem ved samansetting og oppdeling av bord er at bordnummeringa vert gal, når logisk sett same bord opprettast på nytt og får eit anna nummer. Ein måte å handtere det på er å ha to bord-typar, enkeltbord (**SimpleTable**) og samansett bord (**CompositeTable**), kor sistnemnte held reidde på kva for bord som er sett saman. Dette krev ein ny versjon av **mergeTable**-metoden, som altså må opprette eit **CompositeTable** som inneheld dei to borda som settast saman, og **splitTable**-metoden må skrivast om så det deler opp eit **CompositeTable** i dei *same* to borda som vart satt saman. **splitTable**-metoden treng da ikkje lenger dei to **capacity**-argumenta fordi dei to borda jo veit si kapasitet.

c) Forklar med tekst og/eller kode korleis du vil nytta arv og/eller grensesnitt, slik at **Table** fortsatt kan nyttast som generell bord-type og **SimpleTable** og **CompositeTable** kan handtere kvar sine spesialtilfelle. Forklar også verkemåten til **SimpleTable** og **CompositeTable**.

d) Skriv nye versjonar av **Diner** sine **mergeTable**- og **splitTable**-metodar. Merk at den nye **splitTable**-metoden berre skal ta inn ein **CompositeTable** som argument.

Del 4 – GuestManager-klassen (20%)

Gjester på ein **Diner** tas i mot av en tilhøyrande **GuestManager** (sjå vedlegg 1), som freistar å plassere dei. Dersom det ikkje går, så må dei vente på at eit bord med nok kapasitet vert ledig. **GuestManager** vil altså ha behov for å fylgje med på korleis kapasiteten til **Diner**-objektet endrast. Dette gjerast ved å gjere **Diner** sin kapasitet-eigenskap, som returnerast av eit kall til **getCapacity(false)**, *observerbar*.

d) Kva inneber observerbarheit? Forklar kort med tekst og/eller kode korleis ein gjer ein (eigenskap i en) klasse observerbar.

e) Forklar med tekst og/eller kode korleis du vil endre **Diner** slik at **GuestManager** kan lytte til endringar i capacity-eigenskapen (ved å implementere **CapacityListener**-grensesnittet).

f) Forklar med tekst og/eller kode korleis du vil skrive **GuestManager**-klassen. Vi forventar ikkje automatisk samanslåing eller oppdeling av bord, men dei som kom først skal helst få bord først.

Del 5 – Diverse (10%)

d) Er **CapacityListener** eit *funksjonelt* grensesnitt? Grunngje svaret!

e) Skriv (om) ein av **isOccupied** eller **getCapacity** i **Diner** slik at den nyttar **Stream**-teknikken og Java 8 sin funksjonssyntaks (om du ikkje har gjort det frå før, da!).

f) Forklar med tekst og/eller kode korleis du vil teste **isOccupied**-metoden til **Diner** i en separat **DinerTest**-klasse, og nemn kva for metodar i Diner du vil nytte og evt. korleis **Diner** må endrast for at **isOccupied** skal vere enkelt testbar.

Appendix 1: Provided code (fragments)

```
// part 1

/**
 * A group (of people) dining together, and should be seated at the same table.
 * We currently only need to handle the size.
 */
public class Group {

	/**
	* Initializes this Group with the provided guest count
	*/
	public Group(int guestCount) {
		...
	}
}

/**
 * A table with a certain maximum capacity.
 */
public class Table {

	/**
	* Initializes this Table with the provided capacity.
	* The table is also assigned a unique number.
	* @param capacity
	*/
	public Table(int capacity) {
		...
	}

	/**
	* @return the table number
	*/
	public int getNum() {
		...
	}
}

/**
 * Represents the fact that a Group is seated at and occupies a Table
 */
public class Seating {

	/**
	* Initializes this Seating ...
	*/
	public Seating(...) {
		...
	}
}
```

```
// part 2

/**
 * A place where groups of guests can buy a meal
 */
public class Diner {

	/**
	* Tells whether a Table is occupied.
	* @param table the Table to check
	* @return true if anyone is sitting at the provided Table
	*/
	public boolean isOccupied(Table table) {
		...
	}

	/**
	* Computes the guest capacity,
	* either the remaining (includeOccupied == false) or total (includeOccupied == true).
	* @param includeOccupied controls whether to include tables that are occupied.
	* @return the guest capacity
	*/
	public int getCapacity(boolean includeOccupied) {
		...
	}

	/**
	* Adds a table to this Diner
	* @param table
	*/
	public void addTable(Table table) {
		...
	}

	/**
	* Removes a Table from this Diner.
	* If the table is occupied an IllegalArgumentException exception should be thrown.
	* @param table
	* @throws IllegalArgumentException
	*/
	public void removeTable(Table table) {
		...
	}

	/**
	* Merges two tables, i.e. replaces two tables with one table.
	* lostCapacity is the difference between the old capacity and the new.
	* This number is typically positive, since seats are lost when moving two tables
	* close to each other.
	* @param table1
	* @param table2
	* @param lostCapacity
	* @throws IllegalArgumentException if any of the tables are occupied
	*/
	public void mergeTables(Table table1, Table table2, int lostCapacity) {
		...
	}

	/**
	* Splits a table into two, i.e. replaces one tables with two tables.
	* The two capacities are the capacities of the two new tables.
	* @param table
	* @param capacity1
	* @param capacity2
```

```
        * @throws IllegalArgumentException if the table is occupied
        */
        public void splitTable(Table table, int capacity1, int capacity2) {
                ...
        }

        /**
        * Tells whether a table has the provided capacity,
        * i.e. if that number of new guests can be seated there.
        * Note that a table cannot be shared among different groups.
        * @param table
        * @param capacity
        * @return true of capacity number of guests can be seated here, false otherwise.
        */
        public boolean hasCapacity(Table table, int capacity) {
                ...
        }

        /**
        * Returns the tables that has the provided capacity.
        * The tables should be sorted with the one with the least capacity (but enough) first.
        * @param capacity
        * @return the tables that has the provided capacity
        */
        public Collection<Table> findAvailableTables(int capacity) {
                ...
        }

        /**
        * Finds a suitable, existing table for the provided group, and creates
        * (but doesn't add) a corresponding Seating.
        * The chosen table should be the one with the least capacity.
        * @param group the group to be seated
        * @return the newly created Seating
        */
        public Seating createSeating(Group group) {
                ...
        }

        /**
        * Creates and adds a Seating for the provided group, using the createSeating method.
        * @param group
        * @return true if a Seating was created and added, false otherwise.
        */
        public boolean addSeating(Group group) {
                ...
        }

        /**
        * Removes the seating for the provided table (number), if one exists
        * @param tableNum the number of the table to be removed
        */
        public void removeSeating(int tableNum) {
                ...
        }
}
```

```
// part 3

public class SimpleTable ... Table {

        public SimpleTable(int capacity) {
                ...
        }

        ...
}

/**
 * A table that consists of two other tables.
 */
public class CompositeTable ... Table {

        public CompositeTable(Table table1, Table table2, int lostCapacity) {
                ...
        }

        ...
}

// part 4

/**
 * Interface for listening to changes in Diner capacity
 */
public interface CapacityListener {
        /**
        * Called to inform that a Diner has changed capacity
        * @param diner
        */
        public void capacityChanged(Diner diner);
}

/**
 * Handles guests arriving at and departing from a Diner.
 */
public class GuestManager ... {

        public GuestManager(Diner diner) {
                ...
        }

        /**
        * Handles arriving groups, by either seating them immediately
        * (if possible) or putting them in queue. Those enqueued will
        * be seated when the Diner's (change in) capacity allows.
        * @param group
        */
        public void groupArrived(Group group) {
                ...
        }

        /**
        * Handles departing groups, by removing their seating.
        * @param tableNum the table where the group was seated
        */
        public void groupDeparted(int tableNum) {
                ...
        }

        ...
}
```

Appendix 2: Standard classes and methods

interface Iterable<T>

Iterator<**T**>	**iterator**() Returns an iterator over elements of type T.

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.
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 (true|false).
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double using **nextDouble()**.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int using **nextInt()**.
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.