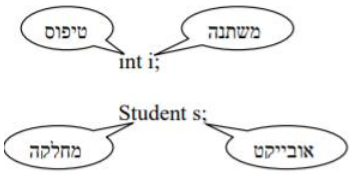
תן הגדרה למחלקה
מבנה לוגי המאגד בתוכו פונקציות ומשתני עצם תחת שם אחד, ממחלקה ניתן ליצור אובייקטים, שם המחלקה הוא שם של טיפוס חדש המוגדר ע"י המשתמש

תן הגדרה לאובייקט
אובייקט הוא משתנה מטיפוס המחלקה, אנו בונים אובייקט ע"י אתמול משתני העצם של המחלקה.



מהו ההבדל בין שיטה לפונקציה?
כאשר אומרים פונקצייה מתכוונים לפונקצייה סטטית, פונקצייה סטטית לא יכולה להשתמש במשתני עצם של המחלקה, אין צורך באובייקט כדי להפעיל פונקצייה סטטית אלא מספיק לכתוב את שם המחלקה ואז נקודה ושם הפונקציה. שיטה מופעלת על אובייקט ולשיטה יש גישה לכל משתני העצם של המחלקה אליו היא שייכת.

מהו ההבדל בין עצם המחלקה לבין משתנה סטטי?
ההבדל ראשון זה שמשתנה עצם נוצר בכל פעם שיוצרים אובייקט ומשתנה סטטי נוצר פעם אחת לפני הפעלת התוכנה ומשותף לכל האובייקט מטיפוס המחלקה
ההבדל השני הוא שניתן לגשת למשתנה סטטי גם מבלי ליצור אובייקט אלא ע"י שם המחלקה ולאחר מכן נקודה, לעומת זאת משתנה עצם ניתן לגשת אליו רק דרך אובייקט.

מהו ההבדל בין משתנה פרטי למשתנה גלובלי ?
משתנה פרטי – רק שיטות מהמחלקה שאנחנו נמצאים בה יכולים לגשים למשתני עצם או השיטות הפרטיות
משתנה גלובלי – כל המחלקות יכולות לגשת למשתני העצם או השיטות

תן הגדרה ל - Protected
ניתן לגשת למשתני Protected רק ממחלקות שיורשות ממני, מהמחלקה עצמה, ולמחלקות שנמצאות באותה חבילה

מה התפקיד של הבנאי?
ליצור אובייקט חדש ולאתחל את משתני העצם של המחלקה

מהו התפקיד של בנאי דפולטיבי ?
בנאי ללא ארגומנטים "נכתב" אוטומטים ע"י הקומפיילר אם לא נכתב אף בנאי אחר למחלקה, הבנאי הדחפולטיבי מאתחל את משתני העצם של המחלקה לערכים דיפולטייבים. 0 למספרים, $null$ למצביעים ו $false$ למשתנים בוליאנים

הסבר את המושג הורשה
כל תכונה ופעולה שהיו מוגדרים במחלקה המורישה מתקיימת באופן צא במחלקה הירושת, בשפת גאבה, ניתן לרשת לכל היותר ממחלקה אחת, כאשר אם לא יורשים משום מחלקה אז יורשים אוטומטית מהמחלקה $Object$, אך אין מניעה שמחלקה אחת תירש למס' מרובה של מחלקות. סוף שושלת הירושה נגמרת במחלקה סופית שהיא $final$.
יתרון: חסכון בקוד, דימוי העולם האמיתי – מקל על ההבנה, היכולת להרחיב קוד אשר נמצא כבר בספריות.

הסבר את המושג Overloading
שתי פונקציות עם אותו שם אבל עם ארוגמנטים שונים, כמות ארגומנטים שונה. ניתן להעמיס כל מתודה מס' פעמיים כל עוד יש שונה מהותי בין החתימה שלה לבין המתודות האחרות שהעומסו. שינוי בערך החזרה לא מספיק!

הסבר את המושג Override
לדרוס מתודה אשר קיימת במחלקה המורישה, אם נרשום מתודה עם חתימה זהה לחתימה של המתודה אשר נמצאת במחלקת האב אז המתודה הזאת תדרוס את התוכן של המתודה במחלקת האב, ניתן לדרוס מתודה פעם אחת בכל ירושה.

תן הגדרה למשתנה סופי
משתנה שאי אפשר לשנות את הערך שלו בזמן ריצה התוכנה
תן הגדרה לשיטה סופית
מתודה שלא ניתן לדריסה ע"י הירושים

הגדר את המושג פולימורפיזם
רב צורתיות, הרעיון הכללי הוא להתייחס לעצמים שונים בתור דברים דומים.
רב צורתיות מאפשר לנו לבצע פעולות מסוימות מבלי קשר ישיר לאובייקט עליו אנחנו מבצעים את הפעולה, כך שגם אם נשנה את האובייקט לאובייקט אחר זה לא ישנה את הדרך בה אנחנו מבצעים את הפעולה שלנו על האובייקט החדש, לדוגמא מערך של חתול,כלב,ציפור כאשר כל ממש את הממשק חיה, ניתן ליצור מערך של חיות וכל פעם להפעיל את המתודה $toString()$, תוצאת המתודה תיהיה כתלות האובייקט עליו אנחנו מפעילים את הפונקצייה.

מה ההבדל בין ממשק לבין מחלקה אבסטרקטית?
בממשק אנחנו רק מצהירים על השיטות ללא מימוש שלהם וללא שדות לעומס מחלקה אבסטקטית אשר יכולה להכיל שדות ומימוש של השיטות, בנוסף מחלקה יכולה לממש כמה ממשקים אבל לרשת רק ממחלקה אחת.

מה ההבדל בין מחלקה לבין מחלקה אבסטרקטית?
ממחלקה אבסטרקטית אין אפשרות ליצור אובייקטים לעמות מחלקה שממנה כן אפשר ליצור אובייקטים

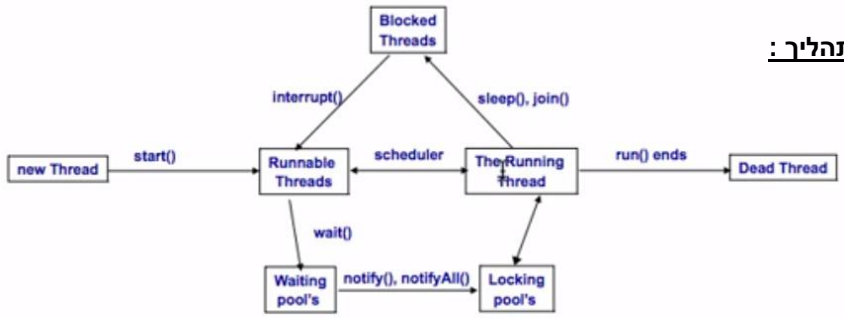
הסבר מהו המושג של Exception
מנגנון שמאפשר להודיע על שגיאה מבלי לעצור את התהליך

מהי המשמעות של המילה $Super$
כאשר אנחנו בתחום המחלקה היורשת, המילה השמורה $Super$ חושפת בפיננו את המשתנים והמתודות במחלקת הבסיס אותה אנחנו מרחיבים
$Super$ מאפשרת לנו גישה למתודות אשר דרסנו במתודה היורשת.

This מהי המשמעות של המילה
מצביע לאובייקט הנוכחי, משומש בעיקר בבנאים.

Thread תן הגדרה ל –
בתוך תהליך אחד ניתן להפעיל כמה תת תהליכים שעובדים במקביל או "כאילו" במקביל.

פרט על כל המצבים של תהליך :



New	קיים אובייקט של ה - thread, אך הוא לא רץ
$Runnable$	מצב בו ה – thread רץ
$Waiting$	מחכה עד ההודעה שהוא יכול להמשיך - notify
$Timed\ Waiting$	מחכה זמן מסויים עד שהוא יוכל להמשיך
$Terminated\ (Dead)$	סיים את חייו, סיים לעבוד

ניתן לקבוע את התדירות של התהליך באופן הבא :
`Thread.setPriority(int)`
כאשר הערכים הינם בין `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`

הסבר מהו ממשק
ממשק הוא הגדרת התנהגות ללא מימוש, מהין "חוזה" אשר מחייב כל מחלקה שרוצה לממש חוזה זה מחוייבת לממש את כל השיטות המפורטות בחוזה.

מהו מצב של Deadlock

מצב של קיפאון בו תהליך א' דורש משאב הנמצא ברשות תהליך ב'. תהליך ב' לא ישחרר את המשאב עד שיקבל את המשאב הדרוש לו המוחזק על ידי תהליך א'

מצב של קפאון, מצב בו שני תהליכים חוסמים אחד את השני ובכך נוצר מצב של קפאון. לדוגמא תהליך 1 מבקש את משאב א' אשר מוקצה לתהליך ב' אשר מבקש את משאב ב' אשר מוקצה לתהליך 1'

הסבר מהו Serialization

תהליך המרת האובייקט לזרם של ביטים, מעביר ממצב של אובייקטים לפורמט שניתן לאחסן אותו
חסרונות: לא נרשם בפורמט קריאה, בנוסף כאשר מוסיפים שדה לאחר כתיבה אז לא ניתן לקריאה.

הסבר מהו Synhronized

המילה השמורה Synchronized מבטיחה שבכל פעם רק תהליך יחיד יוכל להפעיל את קטע הבלוק על אובייקט ספציפי, כלומר, כל עוד הבלוק לא מסתיים אזי כל התהליכים האחרים ממתינים אשר שהתהליך אשר נכנס לבלוק יסיים את עבודו
ניתן להשתמש במילה Synchornized על מנת לנעול מתודות, כאשר נועלים מתודה אז האובייקט שבתוכו נמצאת המתודה ננעל, ניתן גם לייצר אובייקטים ולנעול אותם ע"י קטע בלוק { Synchronized(Object)

הגדר מהו תכנות גנרי

תכנות גנרי הוא הדרך לכתיבת תוכניות שאינן תלויות בטיפוסי המשתנים.

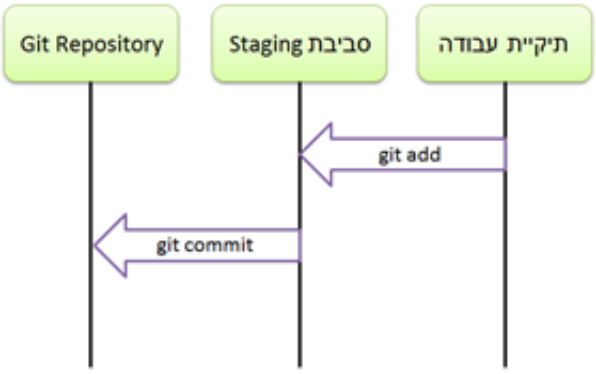
הסבר מהי המילה השמורה InstanceOf

כאשר נרצה לזהות האם אובייקט מסויים שייך למחלקה מסויימת, נוכל להשתמש במילה השמורה Instanceof , בדרך כלל נזדקק כאשר אובייקט מסויים אליו ניגש עשוי להיות שייך לכמה מחלקות אפשריות, ביניהן נרצה להבדיל.

ציינו מה ההבדל בין git ל github ופרטו מהן הפעוילות המרכזיות שעושים בgit

Git היא כלי לניהול גרסאות, בעזרת Git ניתן לחזור לגרסאות קדומות של הפרויקט שעליו אנחנו עובדים, כל פרויקט ב Git מכונה מאגר – Repository וכל מאגר יכול להכלול כמה קבצים ותקיות.
לעומת זאת **Github** הוא **ענן** לשמירת מאגרים של Git.
אפשריות שניתן לעשות דרך Git :

לאתחל רפוזיטורי בתקיה הנוכחית	git init
להוסיף את כל הקבצים שהשתנו מאז הקומיט האחרון אל תוך אזור הסטייג'ינג	git add --all
לבצע קומיט. כלומר, לאירוז גירסה חדשה.	git commit –m "some message"
להציג את היסטוריית הקומיטים שנעשו עד כה ברפוזיטורי הנוכחי	git log
להוסיף הפניה לרפוזיטורי המרוחק שבגיטהאב	git add remote origin YourGithubURL
להעלות את כל הרפוזיטורי שעשיתם לוקאלית אל הרפוזיטורי המרוחק (הגיטהאב שלכם)	git push origin master
להוריד את הרפוזיטורי שבענן אל המחשב שלכם	git pull origin master
להעתיק את תוכן הריפוזיורי של פרויקט כלשהו אל המחשב שלכם	git clone SomeGithubURLToCopy
"לקפוץ" לגרסא ישנה כלשהי. ניתן לראות את המספר המזהה של קומיטים ישנים ע"י פקודת הלוג.	git checkout someCommitID
הצגת השינויי בין 2 קומיטים. הקומיט הראשון צריך להיות הישן יותר.	git diff commitID commitID
יוצר ענף, שיתפצל החל מהקומיט הנוכחי עליו נמצאים בזמן הרצת הפקודה, בשם שתכתבו בפקודה עצמה.	git branch branchName
יציג את שמות כל הענפים הקיימים. כדי "לקפוץ" לענף יש להשתמש בפקודת צ'קאאוט עם שם הענף.	git branch
תצוגה גרפית של כל הקומיטים והענפים שברפוזיטורי הנוכחי	git log --graph –oneline --all



ב-Github יש יכולות נוספות מעבר ל Git ובפרט נושאים שקשורים לניהול הפרויקט, חלוקת משימות, ניהול Issues, ניהול Wiki, יצירת קובץ ReadMe וכו'

הגדר מהי מחלקה פנימית ואיזה סוגים של מחלקה פנימית יש:

יש 3 סוגים של מחלקה פנימית

1. מחלקה חברה
2. מחלקה סטטית
3. מחלקה אנונימית

על מנת ליצור מחלקה חברה :

```
A a = new A();
A.B b = a.new B();

public class A {
	int i;
	class B {
		int j = i; // חוקי!
	}
```

על מנת ליצור מחלקה סטטית :

```
A.B b = new A.B();

public class A {
	int i;
	static class B {
		int j = i; // לא חוקי!
	}
}
```

דוגמא למחלה אנונימית :

```
new Thread(new Runnable() {
@Override
  public void run() {
  // Text
  }
}).start();
```

הסבר מהו S.O.L.I.D

קוד ספגטי

- כל שינוי בקוד משפיע על הרבה חלקים בקוד.
- שינוי בקוד משפיע על אזורים לא קשורים בקוד.
- קוד לא פריק. לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שלשמם נכתב הקוד במקור.

עקרונות SOLID באים לתת קווים מנחים שיגרמו לנו להימנע מלכתוב קוד עם הבעיות הנ"ל.

S - Single Responsibility Principle	למחלקה צריך להיות תחום אחריות **אחד** (למחלקה צריכה להיות סיבה **אחת** להשתנות)
דוגמא: SumCalculator שיש לו גם אפשרות של הדפסה, אנחנו לא רוצים לשלב גם לוגיקה וגם gui ולכן נפצל ל2 מחלקות, מחלקה אחת תיהיה אותה SumCalculator רק בלי הפונקצייה של ההדפסה, וניצור מחלקה חדשה SamCalculatorOutput שמחזיקה בפנים אובייקט מטיפוס SamCalculator ובפנים יהיה אפשריות שונות של הדפסה	
O - Open/Closed Principle	מחלקה צריכה להיות **פתוחה** להוספות **וסגורה** לשינויים
דוגמא: יש לנו מחלקה SamCalculator שיש בה פונקציה של חישוב שטח עבור כל צורה (בודקים אם אובייקט הוא מטיפוס Circle וכו') הבעיה שנוצרת היא אם אנחנו רוצים להוסיף צורה שלא הייתה, המחלקה הזאת לא סגורה לשינויים כי משהו יהיה צריך להוסיף עוד בדיקה, הפתרון לבעיה היא ע"י ממשקים, נגדיר ממשק Shape שיהיה לו את getArea וכל צורה תממש את הממשק Shape וכך לא יהיה צורך לשינויים. סגורה לשינויים – לא צריך לגעת בה בעתיד פתוחה לשינויים – אם נצטרך להוסיף אובייקטים לא נצטרך לגעת בה בעתיד	

L - Liskov Substitution Principle	פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על אובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל
דוגמא: ריבוע שיורש ממלבן, מי שכתב את הפונקציה לא התחשב בכל האובייקטים מסוג הבן, לדוגמא פונקציה שקובעת רוחב ואורך לא זהה, אז בריבוע זה נופל. הפתרון הוא שלא כל האויבקטים תואמים לחיים במציאות ולכן לפעמיים לא צריך לעשות ירושה בין דברים.	
I- Interface Segregation Principle	יש לדאוג לממשקים מצומצמים: - לא לאלץ למחלקה לממש ממשק שאין לה צורך מלא בו. - לדאוג לכימוס מרבי של מידע.
דוגמא: ליצור ממשק של צורה עם פונקצייה של נפח, אז זה לא טוב כי יש צורות דו מימדיות, אז הפתרון זה ליצור 2 תתי ממשקים, כל ממשק מתאים לצורה דו מימדית או תלת מימדית. לגבי כימוס מרבי של מידע, לדוגמא יש מחלקה של איש קשר עם הרבה פרטים, ויש עוד מחלקה של שליחת הודעה ומחלקה של שיחת פלאפון, ברגע שאנחנו שולחים איש קשר הוא מקבל אובייקט עם מלא מידע שאין צורך ולא בטוח שאנחנו היינו רוצים לשלוח את המידע הזה ולכן נרצה לכנס את המידע, הפתרון לבעיה הוא להגדיר 2 ממשקים Emailer ו Dialer ואז Contact יממש את זה	
D- Dependency Inversion Principle	מחלקות height level לא צריכות להשתמש באופן ישיר במחלקות low level
מחלקות height level מחלקות שמתמשות במחלקות אחרות, לדוגמא מכונית היא מחלקה height level היא המכונית וגלגלים זה מחלקה low level . הכוונה היא שאוטו לא צריך להחזיק בתוכו גלגל כי בעתיד אולי הגלגל ישתנה אלא יחזיק **ממשק** מסוג גלגל, ככה שאין צורך לגעת בקוד של המכונית מאוחר יותר.	

אפשרות להדפיס אחרי הנקודה :

```
DecimalFormat df = new DecimalFormat("0.##");
System.out.println(df.format(5.55555));
System.out.printf("%.2f",5.55555);
```

מבני נתונים:

```
Set<Integer> set = new HashSet<Integer>();
List<Double> list = new ArrayList<Double>();
Synhronized!  - List<Integer> s = new Vector<Integer>();
```

נעדיף להשתמש ב List ולא ב<ArrayList<Double בגלל שאם נרצה לשנות בהמשך במקום ArrayList ל LinkedList אז זה יהיה פשוט ואם אחרת, יהיה תכונות של מערך שיכול להיות שהשתמשנו בהם.

```
HashMap<String,Integer> set = new HashMap<String,Integer>(); // <Key,Value>
/*
* K - the type of keys maintained by this map
* V - the type of mapped values
*/
set.put("Tzvi", 314977489);
set.put("Or", 311226617);
for(String s : set.keySet()) System.out.println(s);
for(int d : set.values()) System.out.println(d);
System.out.println(set);
System.out.println(set.containsKey("Tzvi"));
System.out.println(set.get("Tzvi"));
```

פלט:

```
Tzvi
Or
314977489
311226617
{Tzvi=314977489, Or=311226617}
true
314977489
```

תמיד נשתמש ב HashMap אלא אם כן נרצה את האיברים בצורה ממויינת ולכן נשתמש ב HashTree

תבניות עיצוב :

Singleton

מטרה: כיצד להגדיר מחלקה באופן שיאפשר יצירה של אוביקט אחד ויחיד ממנו
מימוש:
1. להגדיר משתנה סטטי מטיפוס עצמו להיות null
2. נגדיר בנאי פרטי
3. להגדיר מתודה getInstance() שמחזירה את האוביקט, כאשר בתוך המתודה יש בדיקה האם המשתנה null או לא, אם כן אז ניצור אחד חדש, בכל אופן נחזיר את המשתנה בסוף המתודה.

קוד:

```
public class A {
 private static A a = null;
 private A() {}
 private static A getInstance()
 {
        if(a == null) a = new A();
        return a;
 }
}
```

Thread - Safe :

```
 private static A getInstance()
 {
        if(a == null)
        {
                synchronized (A.class) {
                        if(a == null)
                                a = new A();
                }
        }
        return a;
}
```

Visitor

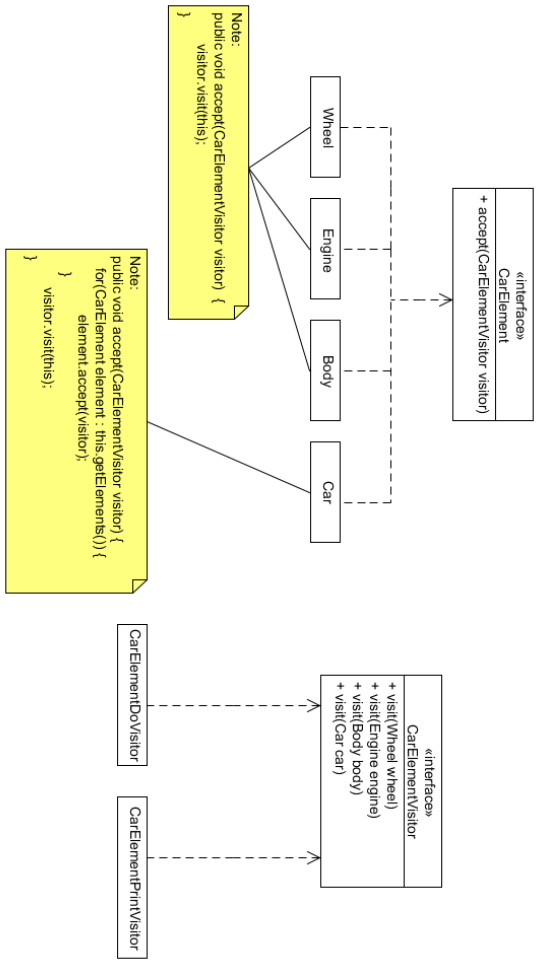
מטרה: מאפשר להגדיר פעולה על אובייקטים מבלי לשנות את המחלקה שלהן
דוגמא: הבעיה עם הצורות, לא אפשרי לשים בכל צורה גם כן לוגיקה ו-gui ולכן נרצה ליצור אובייקט שתפקידו להיות הצייר של הצורות.
מימוש:
1. ליצור ממשק DrawShape אשר כל צורה [Circle,Triangle,Square] יממשו, וזה על מנת שיוכלו לצייר אותם

```
public interface DrawShape {
        public void draw(Painter p);
}
public class { Square / Triangle / Circle } implements DrawShape{
        @Override
        public void draw(Painter p) {
                p.draw(this);
```

2. ליצור ממשק Painter עבור הצייר שתפקידו לצייר כל צורה

```
public interface Painter {
        void draw(Circle c);
        void draw(Square s);
        void draw(Triangle t);
```

3. ליצור צייר שתפקידו לצייר צורה




```
public class NicePainter implements Painter {

	@Override
	public void draw(Circle c) {
		System.out.println("Circle");
	}


	@Override
	public void draw(Square s) {
		System.out.println("Square");

	}

	@Override
	public void draw(Triangle t) {
		System.out.println("Triangle");

	}
```

4. לבצע ציור של הצורות באופן הבא :

```
	public static void main(String[] args) {
		Circle c = new Circle();
		Triangle t = new Triangle();
		Square s = new Square();
		DrawShape[] shapes = new DrawShape[3];
		NicePainter p = new NicePainter();
		shapes[0] = c;
		shapes[1] = t;
		shapes[2] = s;
		for(DrawShape shape : shapes)
			shape.draw(p);
```

Observer

מטרה: מהווה פתרון לבעיה התכנותית הבאה :
כיצד לאפשר לאוביקט אחד או יותר לקבל התראה כאשר מתרחש שינוי בState של האוביקט אשר אנחנו עוקבים אחריו

מימוש:

1. ליצור 2 ממשקים, ממשק אחד עבור Subject וממשק אחד עבור Observer (צופה)

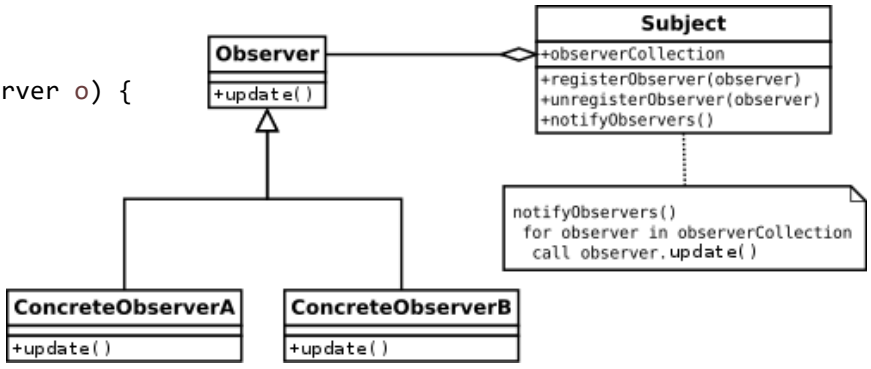
```
public interface Observer {
public void update(String msg);
}
```

```
public interface Subject {
public void register(Observer o);
public void unregister(Observer o);
public void notifyObservers();
}
```

2. ליצור 2 מחלקות, מחלקה ראשונה תיהיה לדוגמא תחנת החדשות והמחלקה השנייה תיהיה Ynet

```
public class NewsChannel implements Subject{
private List<Observer> observers = new ArrayList<Observer>();
private String msg;

@Override
public void register(Observer o) {
   observers.add(o);
}
```



```
 @Override
public void unregister(Observer o) {
   observers.remove(o);
}

public void SendMessage(String msg)
{
   this.msg = msg;
   notifyObservers();
}
```

```
3 public class Ynet implements Observer {
4     @Override
5     public void update(String msg) {
6         System.out.println("YNET: ");
7         System.out.println(msg);
8     }
9
```

```
@Override
public void notifyObservers() {
   for(Observer obs : observers)
   obs.update(msg);
   }
}
```

Decorator

מטרה : מאפשר לנו לשנות אובייקט בצורה דינמית, מאפשר הוספה התנהגות לאוביקט ללא השפעה על ההתהגות של אוביקטים אחרים מאותה מחלקה .

מימוש :

1. ליצור ממשק של פיצה, לממשק יהיה תיאור ומחיר

```
public interface Pizza {
       public String getDescripition();
       public double getCost();
}
```

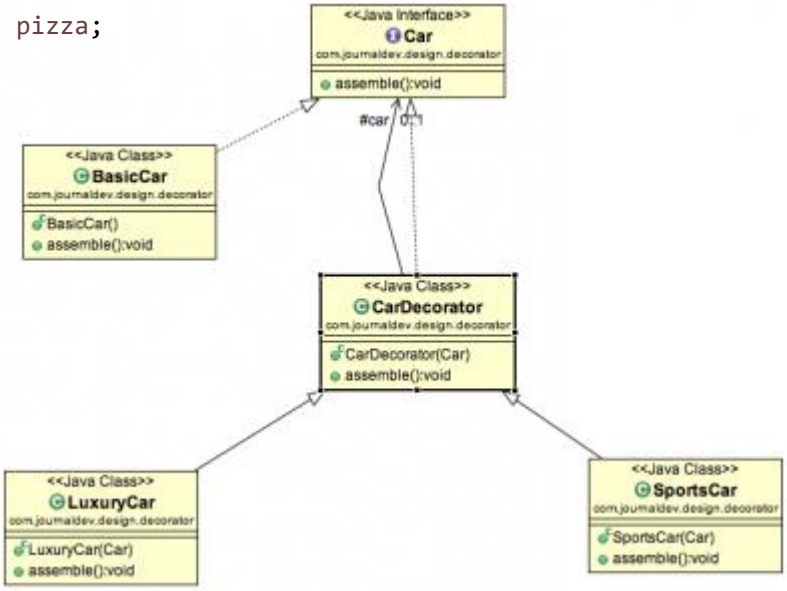
2. ליצור את השכבה הבאה, הבצק. ליצור מחלקה שממשת את הממשק פיצה

```
public class PlainPizza implements Pizza {

       @Override
       public String getDescripition() {
              return "Thin dought";
       }
       @Override
       public double getCost() {
              return 4.00;
       }
```

3. ליצור מחלקה אבסטקטית

```
abstract class ToppingDecorator implements Pizza {
       protected Pizza current;
       public ToppingDecorator(Pizza pizza)
       {
              current = pizza;
       }
}
```



4. ליצור תוספות חדשות שיורשות את המחלקה האבסטקטית :

```
public class TomatoSouce extends ToppingDecorator{
        public TomatoSouce(Pizza pizza) {
                super(pizza);
        }

        @Override
        public String getDescripition() {
                return current.getDescripition() + ",
Tomato Souce";
        }

        @Override
        public double getCost() {
                return current.getCost() + .35;
        }

}
```

```
public class Mozzarella extends ToppingDecorator{
        public Mozzarella(Pizza pizza) {
                super(pizza);
        }

        @Override
        public String getDescripition() {
                return current.getDescripition() +
", Mozzarella";
        }

        @Override
        public double getCost() {
                return current.getCost() + .50;
        }

}
```

5. לבסוף ליצור מחלקה בודקת:

```
public static void main(String[] args) {
Pizza basicPizza = new TomatoSouce(new Mozzarella(new PlainPizza()));
System.out.printf("Ingerdients: %s | Cost:
%f",basicPizza.getDescripition(),basicPizza.getCost());
```

פלט: `Ingerdients: Thin dought, Mozzarella, Tomato Souce | Cost: 4.850000`

Adapter

מטרה: מאפשר למחלקות לעבוד יחד למרות שיש בינהן ממשקים לא תואמים

בעיה: לדוגמא יש לנו תוכנית שרצה על Interface1 והשכרנו מתכנתים שיבצעו בשבילנו עבודה OtherDeveloperWork, הבעיה שהם לא מימשו את Interface1, אלא את Interface2 ולכן לא ניתן לרשום ככה :

```
Interface1 wa = new OtherDevelopersWork();
wa.Print();
```

ולכן, מה שצריך לעשות זה בעצם ליצור סוג של "מתאם" .

המתאם יראה ככה :

```
public class WorkAdapter extends OtherDevelopersWork implements Interface1
{
        @Override
        public void Print() {
                print();
        }
```

כאשר :

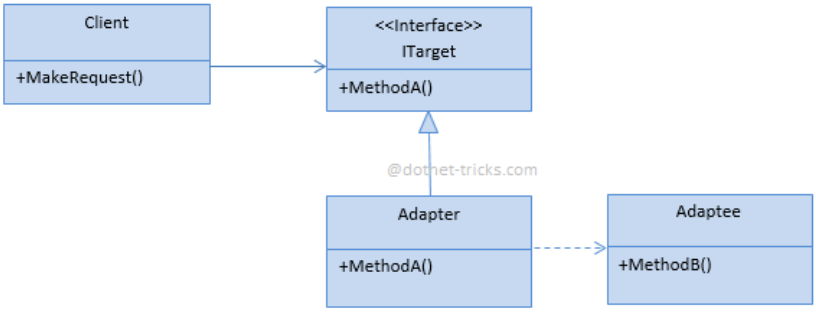
```
public interface Interface2 {
        void print();
}
```

```
public interface Interface1 {
        void Print();
}
```

```
public class OtherDevelopersWork implements Interface2 {
        public void print()
        {
                System.out.println("Other Work");
        }
```

ולכן כעת נוכל ליצור במקום ליצור אובייקט OtherDevelopersWork נוכל ליצור אובייקט מסוג WorkAdapter ולבצע את הפעולות הרצויות :

```
Interface1 wa = new WorkAdapter();
wa.Print();
```



Iterator

מטרה: כיצד לאפשר ליותר מתהליך אחד לעבור על מאגר כלשהו

בעיה: אם היינו מקבלים בחזרה מערך, אז קודם כל היה לנו גישה לדברים שלא היינו צריכים לקבל אליהם גישה, מעבר לזה, אם היינו מקבלים במקום מערך עץ אז היינו צריכים ללמוד איך להשתמש בעץ על מנת לעבור על האיברים

שימוש באיטרטור:

```
Iterator<Integer> it = list.iterator();
            while(it.hasNext())
            {
                  Integer _int = it.next();
                  System.out.print(_int);
             }
```

על מנת לממש את הממשק Iterator :

```
implements Iterator<Integer>
```

על מנת לממש את הממשק Iterable:

```
implements Iterable
```

לאחר שמחלקה ממשת את Iterable אז ניתן להשתמש בלולאה forEach

על מנת לכתוב בעצמנו :

1. נבנה צורה Shape
2. נבנה ממשק Iterator שיש לו את המתודות הבאות :
3. נבנה Iterator שעובר על צורות, הוא מקבל בבנאי Shape[] וממש את Iterator
4. נבנה את אוסף הצורות, נממש את הממשק Iterable על מנת שנוכל להשתמש בלולאה ForEach

```
 public class Shape
 {
        private String name;

        public Shape(String name)
        {
              this.name = name;
        }

        public String getName()
        {
              return name;
        }

        public void setName(String name)
        {
              this.name = name;
        }
```

```
public interface Iterator
{
//Returns true if the iteration has more elements.
      public boolean hasNext();
//Returns the next element in the iteration.
      public Shape next();
      public void remove();

}
```

```
public class ShapeCollector implements Iterable {
      private Shape[] list = new Shape[5];
      @Override
      public Iterator iterator() {
            return new ShapeIterator(list);
      }
```

```java
public class ShapeIterator implements Iterator {
	private Shape[] shapes;
	private int pos;
	public ShapeIterator(Shape[] shapes)
	{
		this.shapes = shapes;
	}
	@Override
	public boolean hasNext() {
		if(pos >= shapes.length || shapes[pos] == null)
			return false;
		else
			return true;
	}

	@Override
	public Shape next() {
		return shapes[pos++];
	}

	@Override
	public void remove() {
		if(pos <= 0) throw new IllegalStateException("Illegal Pos");
		if(shapes[pos - 1] != null)
		{
			for(int i = pos-1; i< shapes.length -1; i++)
				shapes[i] = shapes[i+1];

			shapes[shapes.length-1] = null;
		}
	}
}
```

<u>ההבדל בין הממשק Iterator לבין הממשק Iterable :</u>

נסתכל על Itrator כעל ממשק שהמטרה שלו לעזור לנו לעבור על אוסף ע"י המתודות hasNext(),next(),remove() , לעומת זאת Iterable זהו ממשק שכל מחלקה שממשת את הממשק מאלצת אותו להיות "Iterable" ובאופן כללי ניתן לשימוש ע"י הלולאה ForEach

State

<u>מטרה:</u> כיצד לגרום לכך שאופן פעולתו של אוביקט תשתה כאשר המצב שלו משתנה (מצב = ערכים וכו') מאפשר לעצם לשנות את התנהגותו כתוצאה של שינוי במצבו הפנימי. העצם יראה כאילו הוא החליף מחלקה!

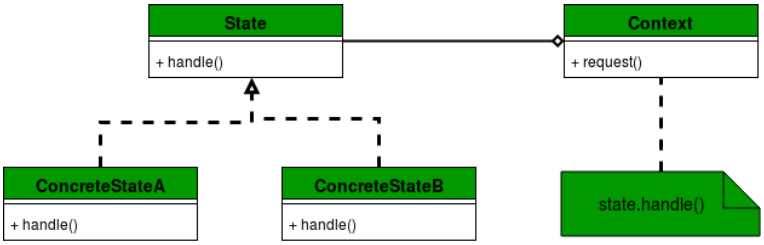
דוגמה:

מנורה אשר יכולה להיות דלוקה או כבויה

לדוגמא נסתכל על מכונית, יש מצב חסכוני ויש מצב ספורטיבי, אם נשמור בתוך מכונית את המצב אז לא מוכל לבצע שינויים, ולכן ניצור ממשק חדש שנקרא לו State שמצב ספורט ומצב חסכוני יממש אותו, בממשק State יש את המתודות האבסטקטיות הבאות :

```java
public interface State
{
	public abstract int getMaximumSpeed();
	public abstract void setSpeed(int num);
	public abstract int getSpeed();
	public abstract double getFuelConsumption();
}
```

לאחר מכן, במוכנית עצמה יהיה משתנה מטיפוס State ומשם נוכל להכניס ע"י Setters איזה מצב אנחנו רוצים. כל המתודות יפעלו על ה State ובכך יהיה ניתן לשנות באופן דינמי את התנהגות המכונית ביחס למצב הנוכחי.



Strategy

מטרה: כיצד לאפשר לאוסף של אלגורתמים שונים אשר מהווים פתרון לאותה בעיה להופיע בתוכנית באופן שיאפשר לנו להחליף אלגוריתם אחד באחר ללא שנאלץ לשנות את הקוד של שאר התוכנית יתר על המידה

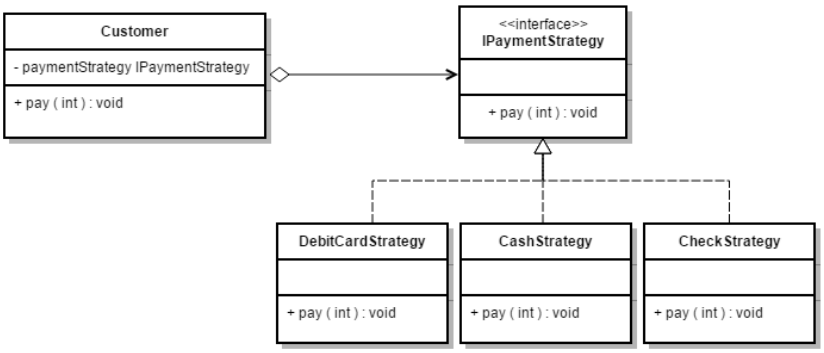
מאוד דומה ל State רק ש State הוא דינמי בעוד ש Strategy אין מצב , לכולם יש אותו מצב אבל יש דרכים אחרים לטיפול בבעיה ואנחנו נבחר דרך אחר ונמשיך איתה

מימוש: ליצור Strategy ואז ליצור לדוגמא BubbleSortStrategy ו HeapSortStrategy, כעת ניתן לבחור בין השניים בתוכנית הראשית, כאשר נחזיר `Strategy sort = new BubbleSortStrategy()` או לחלופין בשני.

הסבר על השוני :

in Strategy pattern, there are no states or all of them have same state. All one have is different ways of performing a task, like different doctors treat same disease of same patient with same state in different ways.

In state Pattern, subjectively there are states, like patient's current state(say high temperature or low temp), based on which next course of action(medicine prescription) will be decided.And one state can lead to other state, so there is state to state dependency(composition technically).

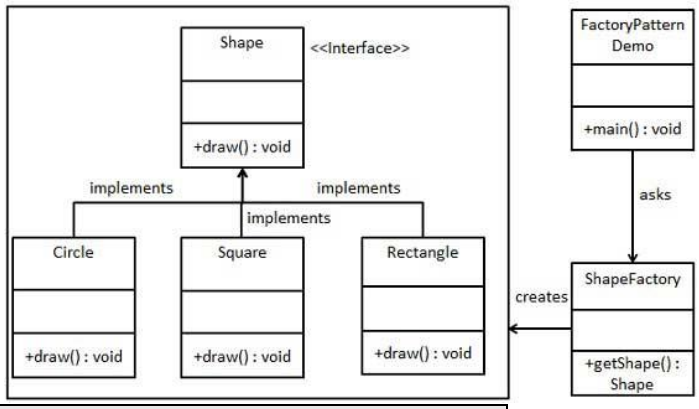


Factory

מטרה: כשנרצה שיטה שתחזיק אובייקט מאחת מכמה אפשריות כאשר כל האוביקטים בעלי מכנה משותף כלשהו, בדרך כלל נשמתמש בתבנית עיצוב פקטורי כשלא נידע באיזה שעה איזה מחלקה נצטרך, בנוסף זה דרך לסתיר את כל תתי המחלקות ותתי הבנאים, באופן כזה שהלקוח לא יהיה נגיש אליהם.

מימוש:

```
public class ShapeFactory {
	Shape CreateShape(String s)
	{
		Shape shape = null;
		if(s.equals("Circle"))
			shape = new Circle();
		else if(s.equals("Square"))
			shape = new Square();
		return shape;
	}
```



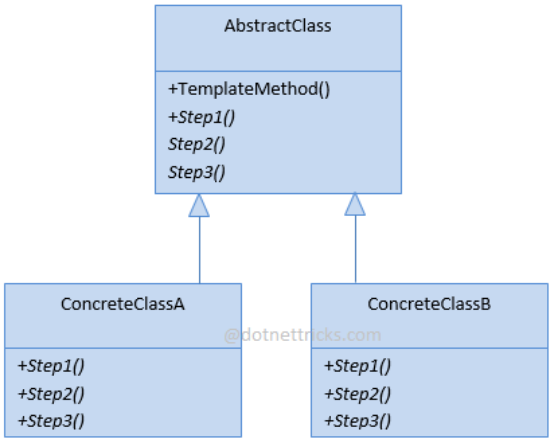
Template

מטרה: מגדיר שלד של אלגוריתם בפעולה, ומפנה חלק מן הצעדים לתת מחלקות. כך תת-מחלקות יכולות להגדיר מחדש צעדים מסוימים של האלגוריתם, ללא שינוי של המבנה שלו.

נשתמש כאשר:

- נרצה לממש את החלקים הקבועים של האלגוריתם, ולאפשר למחלקות היורשות לקבוע את השאר.
- נרצה לרכז את ההתנהגות המשותפת של תתי המחלקות במקום אחד, כדי למנוע שכפול קוד.

צבי מינץ סיכום מונחה עצמים



Composite

מטרה: גיבוש אובייקטים למבנה היררכי של עץ, איפשור התייחסות לאובייקטים כאובייקטים עצמאיים או באופן אחיד לצרוף של אובייקטים.

נשתמש כאשר נרצה שהלקוחות יוכלו להתעלם מההבדלים בין הרכבה של אובייקטים לאובייקטים בודדים, ויתייחסו לשניהם באופן אחיד

לרשום מחלקה של **Exception** :

```
public class FactorialException extends Exception {
        public FactorialException(){
              super();
          }
          public FactorialException(String message){
              super(message);
          }
}
```

על מנת להשתמש **בזמן** :

```
System.currentTimeMillis();
```

הסבר מהו Comperator וכיצד משתמשים בו

כאשר אנחנו רוצים למיין אוסף של עצמים מטיפוס מסוים, עלינו להגדיר כיצד משווים שני מופעים של המחלקה

ישנם 2 דרכים למיון :

דרך ראשונה:

1. להוסיף את הממשק Comparable

```
public class Coordinate implements Comparable<Coordinate> {
```

2. לממש את המתודה compareTo

```
@Override
      public int compareTo(Coordinate other) {
            return this._x - other.getX();
}
```

ניתן גם בעזרת

```
return Integer.compare(this._x, other.getX());
```

3. כעת ניתן לבצע מיון בעזרת Collection.Sort(List)

```
List<Coordinate> list = new LinkedList<Coordinate>();
            list.add(new Coordinate(1, 3));
            list.add(new Coordinate(0, 6));
            Collections.sort(list);
```

עבור כל פונקצייה x.CompareTo(y) :

- אם x > y אז להחזיר 1 או מספר חיובי
- אם x = y אז להחזיר 0
- אם x < y אז להחזיר -1 או מספר שלילי

דרך שנייה:

1. להוסיף מחלקה חדשה שתממש את Comperate

```
public class CoordinateComperator implements Comparator<Coordinate>{
	@Override
	public int compare(Coordinate first, Coordinate second) {
		return Integer.compare(first.getX(), second.getX());
	}
```

2. במיין להכניס את cmp אל Collection.sort(List,cmp)

```
List<Coordinate> list = new LinkedList<Coordinate>();
CoordinateComperator cmp = new CoordinateComperator();
list.add(new Coordinate(1, 3));
list.add(new Coordinate(0, 6));
Collections.sort(list,cmp);
```

כיצד ניתן לפתור בעיית סינכרון :

הגדרת הבעיה:

כאשר אנחנו מוסיפים איבר לקבוצה, זוהי **לא** פעולה **אטומית** אלא סדרה של פעולות. נדמיין את התרחיש הבא, תהליך ראשון מעוניין להוסיף איבר לקבוצה, הוא מקבל את האינדקס X לשים בו את הערך, אולם כרגע הוא מאבד את הזמן עיבוד והתהליך השני מקבל גם כן את האינדקס X הפנוי הבא בתור, התהליך השני מספיק להכניס את האיבר לתא X , כעת נגמר הזמן עיבוד עבור התהליך השני ולכן נחזור להתהליך הראשון, כעת הוא מבצע השמה עבור האיבר באינדקס X, קבלנו דריסה של איבר. במקום להכניס 2 איברים למערך הכנסו רק 1. נשים לב כי המקרה הגרוע ביותר עבור דריסה היא שבמקום 2X איברים עם 2 תהליכים, כל אחד ידרוס את האחר ולכן נבצע X השמות, בעוד שהמקרה הטוב זה שאין דריסות בכלל ולכן נוכל להכניס X2 איברים לקבוצה.

תרחיש שני (תוצאה שגויה):

1. Thread **א'** קורא מהזיכרון את הערך שבמקום x
2. Thread **א'** מגדיל את הערך ב-1

מתבצעת **החלפת הקשר**

3. Thread **ב'** קורא מהזיכרון את הערך שבמקום x
4. Thread **ב'** מגדיל את הערך ב-1
5. Thread **ב'** כותב למקום X את הערך החדש

מתבצעת **החלפת הקשר**

6. Thread **א'** כותב למקום X את הערך החדש

אפשרות 1:

```
Set<String> set = Collections.synchronizedSet(new HashSet<>());
List<String> set = Collections.synchronizedList(new ArrayList<String>());
Map<String,Double> set = Collections.synchronizedMap(new
HashMap<String,Double>());
```

אפשרות 2:

להחליף למבנה נתונים סינכרוני לדוגמא Vector

אפשרות 3:

לנעול אותם על אותו אובייקט (אפשר על המערך), לא לעשות במתודה Synchornized! כי כל אחד מהם על אובייקט נפרד!

אפשרות 4:

1. ליצור מתודה *לא סטטית* DoWork() שבתוכו יהיה כל התהליכים
2. מהמיין ליצור אובייקט מטיפוס MyMain main = new MyMain() ולקרוא ל main.DoWork()
3. להעביר את הSet לאזור הגלובאלי, ולבנות מתודה סינכרונית הבאה :

```
public class MyMain {
	private List<String> set = new ArrayList<>();
	private synchronized void add(String s)
	{
		set.add(s);
```

4. לשנות ל add(s) במקום set.add(s) בכל התהליכים

מהי פעולה אטומטית ?
פעולה אטומית הינה רצף של פקודות מכונה או פקודת מכונה יחידה שיבוצעו על ידי מעבד, **מבלי** שיכולה להיעשות החלפת הקשר בזמן ביצוען ומבלי שרכיב אחר במחשב יוכל להבחין בכל מצב ביניים אלא רק במצב ההתחלתי או הסופי של הפעולה.

מהו משתנה מסוג volatile ?
משמש כדי לציין שניתן להשתמש ולשנות את הערך של המשתנה ב threads-שונים, ללא volatile threadsשונים עלולים לראות ערכים שונים.

הורשה ובנאים :
במידה ויש מחלקה B שיורשת ממחלקת בסיס A
מתוך כל בנאי של B יש קריאה לבנאי של A, אם בA יש בנאי עם ארגומנטים בעוד שאין לו בנאי בלי ארגומנטים אז תיהיה שגיאת קומפלציה, כי כל פעם שיוצרים את B צריך ללכת לאחד הבנאים של A, ולכן אפשר לתקן עם 2 אפשריות, או להוסיף `super(i)` בבנאי של B או ליצור בנאי ריק ב-A

הגדר מהי מחלקה גנרית :
תכנות גנרי הוא הדרך לכתיבת תוכניות שאינן תלויות בטיפוסי המשתנים.

זכרון של String :
הפלט עבור

```
String s1 = "abc", s2 = "abc";
System.out.println(s1==s2);
```

הינו `true`
היות ו

```
String s1 = "abc",s2 = "a", s3 = s2 + "bc";
System.out.println(s1==s3);
```

הינו `false`

הסבר:
there is no way the compiler know if s1 and s2 would end up being the same value, At runtime, unless you call String.intern(), jvm won't check the string literal pool to see if the value is already there.

ולכן אם נשנה את זה לאופן הבא נקבל פלט `true` :

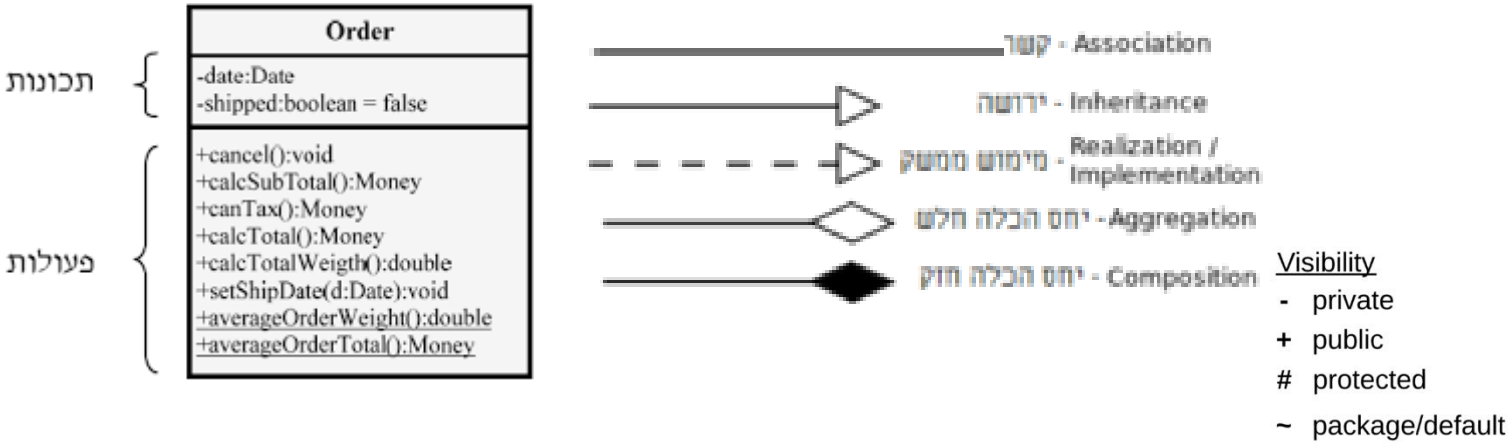
```
String s1 = "abc",s2 = "a", s3 = (s2 + "bc").intern();
System.out.println(s1==s3);
```

הפלט עבור

```
String a="test";
String b=new String("test");
if (a==b) ===> false
```

הינו `false`, נשים לב כי יצירת `new String` היא ב heap בעוד ש יצירה רגילה `"abc"=` היא ב `String` pool שגם כן נמצא ב heap אבל הוא במקום יעודי

הסבר מהי דיאגרת מחלקות:
היא סוג של תרשים סטטי המתאר את מבנה המערכת על ידי הצגת מחלוקתיה, תכונותיהן והקשרים בין המחלקות.
שפת UML - שפת התרשימים UML מאפשרת למדל מערכות תוכנה באמצעות תרשימים גרפיים.
החשיבות שיש לאפשרות ליצור מידול גרפי לפני שמושקעים המשאבים ומתחיל תהליך הפיתוח דומה לחשיבות שיש ליצירת מודל פיזי קטן על ידי ארכיטקט לפני שמתחילים בבניה בפועל.



הגדר מה זה תכנות מונחה עצמים:

תכנות מונחה עצמים זה סוג של תכנות אשר משתמשים בעצמים (אובייקטים) על מנת ליצור תוכניות במחשב, העקרון של תכנות מונחה עצמים דומה מאוד לצורה שמתנהל העולם, כאשר כל הדברים הם בעצים עצמים - קטגוריות וכל קטגוריה מחולקת לתת קטגוריה וכו'
דוגמא לכך זה לדוגמא המבחן שאתם קוראים הוא לא סתם מבחן, זה מבחן שנרשם בעזרת אובייקט עט שיש לו את הצבע שחור, שמוחזק ע"י אובייקט בנאדם בעל ת.ז 111111 וכו'..

Junit Testing :

Junit הינו מנגנון מובנה בשפת JAVA המאפשר כתיבת מודול בדיקות והרצתן ברצף בצורה נוחה ומהירה.

```
assertEquals("Text Error",expected,actual);
```

@Test	מסמל שהפונקציה היא פונקציית בדיקה
@BeforeEach	פונקציה שתרוץ לפני כל בדיקה. פונקציה זו יכולה להכין את סביבת הבדיקה (למשל: לקרוא את נתוני הקלט, לאתחל את המחלקה)
@AfterEach	פונקציה שתרוץ לאחר כל בדיקה. פונקציה זו יכולה לנקות את סביבת הבדיקה (למשל: למחוק נתונים זמניים, לשחזר ברירת מחדל). בנוסף יכולה לחסוך בזיכרון על ידי ניכוי מבני נתונים יקרים.
@BeforeAll	פונקציה זו מתבצעת פעם אחת, לפני תחילת כל הבדיקות. יכול לשמש לביצוע פעולות שדורשות זמן כגון חיבור למסדי נתונים. צריכה להיות מוגדרת כסטטית.
@AfterAll	פונקציה זו מתבצעת פעם אחת, לאחר שכל הבדיקות הסתיימו. יכול לשמש לביצוע פעולות ניקוי למשל התנתקות מבסיס נתונים. צריכה להיות מוגדרת כסטטית.
@Disabled	מתעלם מפונקציית הבדיקה. שימושי כאשר הקוד הבסיסי השתנה ומקרי הבדיקה עדיין לא אומצו או שזמן בדיקה זה יותר מידי גדול מלהיכלל

assertEquals	בודקת אם שני הביטויים שווים (מפעילה equals)
assertNotEquals	בודקת אם שני הביטויים שונים
assertSame	בודקת שלשני משתנים יש הפניה זהה
assertNotSame	בודקת שלשני משתנים יש הפניה שונה
assertFalse	מופעלת עם ביטוי בוליאני ובודקת אם הוא false
assertTrue	מופעלת עם ביטוי בוליאני ובודקת אם הוא true
assertNull	מופעלת על משתנה ובודקת שהוא null
assertNotNull	מופעלת על משתנה ובודקת שאינו null
fail	מכשילה את ה- test

כל השיטות יכולות לקבל כפרמטר ראשון מחרוזת, שתהווה תיאור הבעיה

הרשאות גישה:

הרשאה	גישה מאותה מחלקה	גישה מאותו package	גישה ממחלקה יורשת	גישה מכל מקום אחר
public	V	V	V	V
protected	V	V	V	X
שום דבר	V	V	X	X
private	V	X	X	X

קריאה וכתיבה לקובץ

גאבה מספקת מחלקות שמשמות לביצוע פעולות קלט ופלט, רוב המחלקות הן מחלקות שהאובייקטים מטיפוסן מייצגים Streams כלומר זרם, הכוונה היא לזרם של bytes אשר מהווה זרימה של נתונים, כמו למשל כתיבה למקדם מהווה זרימה של bytes של מידע רלוונטי עבור הלחיצות.

דבר ראשון בהירכיה יש את InputStream ו OutPutStream שניהן מחלקות אבסטקטיות שהן משמשות כבסיס למחלקות אחרות רבות, לדוגמא FileInputStream ו FileOutputStream
אובייקטים מטיפוסים אלה עושים שימוש בקבצים כמקור/יעד של bytes, בין הבנאים של שני המחלקות האלה ניתן למצוא גם בנאי שאמפשר לצייר את שם הקובץ, במידה והקובץ לא נמצא throws FileNotFoundException

צבי מינץ סיכום מונחה עצמים

FileInputStream	FileOutputStream
public FileInputStream (String name) בנאי שממש ליצירת אובייקט חדש מטיפוס FileinputStream, אשר יקשור לקובץ ששו נשלח כString. הקובץ חייב להיות קיים אחרת זורק FileNotfoundException	**public FileOutputStream(String name)** בנאי ליציירת אובייקט חדש מטיפוס FileOutPutStream אשר מקשור לname. אם השם לא קיים או לא ניתן לקרוא את הקובץ אז תזרק FileNotFoundException
public int read(byte []b) שיטה זו קוראת bytes מהinput stream ישירות אל תוך המערך b. מספר הbytes שהיא קוראת לא גדול מb.length, השיטה מחזירה את מספר בתים שנקראו או -1 אם אין יותר מידע שניתן לקרוא כיוון שהיא הגיע לסוף של הקובץ	**public FileOutputStream(String name, boolean append)** בנאי ליצירת אובייקט של FileOutputStream חדש שמקוקשר לקובץ ששמו name. אם הארגומנט השני שנשלח אל הבנאי true אז הbytes יכתבו אל סופו של הקובץ ולא מתחילתו, אם לא ניתן ליצור את הקובץ או שהקובץ לא ניתן לקריאה אז נזרקת שגיאה FileNotFoundException
public int read(byte []b, int off, int len) שיטה זו קוראת עד len בתים של נתונים מהinput stream הישר אל תוך המערך שהיא מקבלת, הbytes ננכתבים אל המערך החל מתא מספר off והשיטה מחזירה את מספר הבתים או -1 או נגמר הקובץ	**public void write (int b)** שיטה זו כותבת byte שערכו b אל הקובץ המקושר לאובייקט **public void write (byte[] b)** שיטה זאת כותבת b.length בתים ממערך b אל הקובץ שמקושר לאובייקט.
public long skip(long n) שיטה זו מקבל את מספר הבתים שעליהם עליה לדלג והיא מחזירה את מספר הבתים שעליהם היא דלגה בפועל, לעיתים זה קטן יותר ולעיתים זה שווה אפילו ל0	**public void write (byte[] b, int off, int len)** שיטה זו כותבת len בתים החל מהמיקום off אל הקובץ שמקשור לאובייקט
public int available() שיטה זה מחזירה את מספר הבתים שניתן לקרוא מבלי לגרום לאחת השיטות read להחסם	**public void close()** שיטה זו סוגרת את הFileOutputStream ומשחררת את כל משאבי המערכת שבהם האובייקט השתמש
public void close() מתודה לסגירת ה stream שמייצג האובייקט ושלחרור כל משאבי המערכת שבהם הוא השתמש	

המחלקות ObjectInputStream ו ObjectOutputStream
הן שתי מחלקות שמשמות לקריאה/כתיבה אובייקטים וגם לקריאה/כתיבה טיפוסים בסייסים. כדי שניתן יהיה לכתוב אובייקט לObjectOutPutStream על המחלקה שלו ליישם את interface Serializable במידה והממשק לא יושם אז תיזרק השגיאה NonSerializalizableException

ObjectOutputStream	ObjectInputStream
public ObjectOutputStream(FileOutputStream out) בנאי זה יוצר אובייקט חדש מטיפוס ObjectInputStream אשר דרכו ניתן לכתוב אל InputStream שהמצביע אליו נקלט בתוך הפרמטר out	**public ObjectInputStream(FileInputStreamin)** בנאי זה משתמש ליצירת קובץ חדש אשר יקשור ל InputStream שהמבציע שלו נשלח כבנאי
public final void writeObject(Object obj) שיטה זה משמשת לכתיבת אובייקט מסויים לאוביקט ObjectOutputStream שממנו היא הופעלה, יחד עם האוביקטים שכותים נכתב גם כל האוביקטים האחרים שמצביעים אליהם נמצבעים באובייקטים שכותבים	**public final Object readObject()** שיטה זו משתמש קראת אובייקט מתוך האובייקט ObjectInputStream שעליו היא הופעהל, המתודה מחזירה מצביע מטיפוס Object לאבוייקט שקראה, יש לעשות casting בהתאם למבציע שהתקבל
public void flush() שיטה זו גורמת לכתיבה כל הbytes שנאספו מbuffer ושטרם נשלחו בפועל.	**public int available()** שיטה זו מחזירה את מספר הבתים שניתן לקרוא מבלי לחכות לבתים נוספים שהגיעו
public void close() שיטה זו אחראית לסגירת הstream שמיוצג ע"י אובייקט ה ObjectoutPutStream	**public void close()** שיטה זו גורמת לסגירת האובייקט ObjectoutputStream אשר עליו היא הופעלה

המחלקות DataOutputStream וDataInputStream - שתי מחלקות אלה, מייצגות streams אשר ניתן להשתמש בהם כדי לכתוב/לקרוא ערכים מטיפוסים בסיסיים.

מהו JDBC

Java DataBase Connectivity זה API שמאפשר התחברות לבסיס נתונים מjavan ולבצע פעולות database

API – ספרייה של קוד, פקודות או פונקציות בהן יכולים המתכנתים לעשות שימוש פשוט מבלי להידרש לכתוב אותן בעצמם כדי שיוכלו להשתמש במידע של הירשום שממנו הם רוצים להשתמש לטובת הישום שלהם.

הגדר מהו SQLite ומה היתרונות והחסרונות שלו

SQLite היא מערכת לניהול בסיס נתונים אשר משתמש בשפה SQL

יתרונות של Sqlite זה שקובץ ההרצה שלו מאוד קומפקטי, אין צורך בהתקנה, תומך בנפחים גדולים של מידע, הוא חינמי וניתן להשתמש בו במגוון רחב של שפות, בפרט בגאבה.

החסרונות הם שהוא לא מאפשר גישה לרשת, אלא הכל נשמר על הדיסק הקשיח.

על מנת **להתחבר** לבסיס נתונים :

```
//Database Info
String DB_URL = "jdbc:sqlite:DB.db"; //DB is the File name
String JDBC_DRIVER = "org.sqlite.JDBC";
try {
Class.forName(JDBC_DRIVER); //Register JDBC driver

// Create connection to the database
// Connection connection
this.connection = DriverManager.getConnection(DB_URL);
this.st = connection.createStatement(); // Statement st
}
catch(Exception e) { e.printStackTrace(); }
```

פקודות:

ממשק שהמטרה של כל אובייקט שממש את הממשק הזה לאפשר שאילתות SQL ולהחזיר את התוצאות עבור השאילתה	Statement
פקודות להרצת שאילתה מול הבסיס נתונים	**executeQuery** – מחזיר ResultSet עבור השאילתה SELECT **executeUpdate** – מריצה שאילתה שמעדכנת את בסיס הנתונים, מחזיר את מספר השורות שבהם התבצע שינוי **execute** – מבצע כל סוג של שאילתה, מחזיר true אם תוצר השאלה הוא מסוג RS או False אחרת

Statement
ממשק שהמטרה של כל אובייקט שממש את הממשק הזה לאפשר שאילתות SQL ולהחזיר את התוצאות עבור השאילתה
דוגמא:
`String query = "SELECT * FROM STUDENTS WHERE ID='"+id+"'";`
`ResultSet rs;`
`rs = st.executeQuery(query);`
	prepareStatement יורשת מStatement ולכן רק באופן שליחת הפרמרטים לשאילתה שונה, ההרצה היא זהה, ניתן לעבוד עם prepareStatement ובמקרה זה מחליפים אך ורק הערכים לפרמטרים. שותלים את הפרמטרים באמצעות שיטות. כלומר במקום כל השאילתה תיהיה הצבה של ערך ?
	דוגמא:
	`String query = "INSERT INTO STUDENTS (NAME, ID) VALUES(?,?)";`
	`PreparedStatement pstmt = connection.prepareStatement(query);`
	`pstmt.setString(1, name);`
	`pstmt.setInt(2, id);`
	`pstmt.executeUpdate();`

- ResultSet כולל את התוצאות משאילתת ה-SELECT שבוצעה. התוצאות מסודרות במין סוג של טבלה, הדומה לטבלת מאגר הנתונים, כאשר מספר השורות הוא כמספר הרשומות	ResultSet

שעונות לתנאי של השאילתה ומספר העמודות כמספר השדות שביקש משפט השאילתה להציג. על מנת לשלוף מידע מהטבלה יש צורך בלהשתמש ב-getXXX(param) כאשר XXX מציין את טיפוס הנתונים לדוגמא Int,String וכו', param1 זה שם העמודה	
פקודות להרצת שאילתה מול הבסיס נתונים	**executeQuery** – מחזיר ResultSet עבור השאילתה Select **executeUpdate** – מריצה שאילתה שמעדכנת את בסיס הנתונים, מחזיר את מספר השורות שבהם התבצע שינוי **execute** – מבצע כל סוג של שאילתה, מחזיר true אם תוצר השאלה הוא מסוג RS או False אחרת

דוגמא לפקודות: (הדפסה של כל הבסיס נתונים Student)

```
ResultSet rs = st.executeQuery("SELECT * FROM STUDENTS");
while(rs.next())
                    {
                            int id = rs.getInt("ID");
                            String name = rs.getString("NAME");
                            ans += (counter++) + ":[" + name + "," + id +"]"
+ "\n";
                    }
```

לעדכן שם לפי ת.ז

```
String query = "UPDATE Students SET NAME = ? WHERE ID = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, name);
pstmt.setInt(2, id);
pstmt.executeUpdate();
```

להוסיף סטודנט חדש עם שם ות.ז

```
String query = "INSERT INTO STUDENTS (NAME, ID) VALUES(?,?)";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, name);
pstmt.setInt(2, id);
pstmt.executeUpdate();
```

לקבל שם לפי ת.ז

```
public String getNameByID(int id)
      {
            String query = "SELECT * FROM STUDENTS WHERE ID='"+id+"'";
            ResultSet rs;
            String name = "";
            try {
                  rs = st.executeQuery(query);
                  name = rs.getString("NAME");
            } catch (SQLException e) {
                  e.printStackTrace();
            }
            return name;
      }
```

הגדר מהו אוסף

אוסף הינו מבנה נתונים המאפשר לנו להחזיק כמות כלשהי של איברים בעלי מכנה משות

מהו XML

הוא תקן ליצוג נתונים במחשבים, שימוש בXML מקל על החלפת נתונים בין מערכות שונות שפועלות על גבי תשתיות שונות.
על מנת שמסמך XML יחשב ל Well-Formed צריך להתקיים :
1. שלכל תגית התחלה יש תגית סיום, ושכל צמד תגיות אשר יכונן בתוך צמד תגיות אחר, ישמור על המקיום המתאים של התגיות. (כמו חוק הסוגריים <><> או <<>>)
2. שיכלול מרכיב אחד ראשי

בתחילת כל מסמך XML יופיע משפט הקדמה אשר יציין כי זהו מסמך XML וגם כן יכיל הצהרה בנוגע לגרסא של ה XML שעל פיה המסמך כתוב , משפט הקדמה מתחיל ב ?>
בקובץ XML יכול להיות קטע מסוג CDATA שמטרצו לשלב טקסט שאין עומד בכללי ה well-formed למשל קוד java או טקס html שמעוניינים להציג בדפדפן
כל מידע שנמצא ב CDATA לא עובר בתהליך פענוח
קטע קוד כזה נראה ככה: <![CDATA[JAVA TEXT]]>

מהו JSON

פורמט טקסואלי הקריא לאדם אשר מיועד להעברת מבני מידע המורכבים מזוגות של מפתח-ערך. השימוש העיקר של פורט הוא להעברת מידע בין שרת לצרכן כתחליף לפורמט XML כי הוא פשוט יותר, JSON הוא מבנה נתונים לא תלוי שפה.
דוגמא לקוד JSON

```
{
    "firstName": "יעקב",
    "lastName": "ישראלי",
    "address": {
        "streetAddress": "13 ,רחוב המשעול",
        "city": "ירושלים",
        "state": "ישראל",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

מהו KML

KML זה פורמט של קובץ אשר המטרה שלו זה להראות מידע גאוגרפי סביב העולם באפלקציות מסוג Earth browsers כמו Google Earth לדוגמא.
KML מבוסס XML

Enums :

מטרה של enums זה לייצג קבוצה של קבועים בשפות תכנות, לדוגמא ימים בשבוע וכו' נשתמש ב enum כאשר אנחנו רוצים לדעת את כל האפשריות עבור ערכים בזמן קיפול.
יש 2 סיבות עקריות לשימוש בenums :
1. קל לקריאה וכתיבה
2. הקומפיילר תופס שגיאות
נשים לב כי למרות שלא מוציין שהמשתנים של enum הם סטטים וסופיים, הם אכן כאלה.
יתרונות מול Static final :

1. Type safety and value safety. 2. Guaranteed singleton.	1. Ability to use values in `switch` statement `case` statements without qualification.

3. Ability to define and override methods.	2. Built-in sequentialization of values via `ordinal()`. 3. Serialization by name not by value, which offers a degree of future-proofing. 4. `EnumSet` and `EnumMap` classes.

```
enum Color {
	RED(255,255,255), BLACK(233,233,233), YELLOW(200,200,200);
	private int red;
	private int green;
	private int blue;
	private Color (int r, int g, int b) {
		this.red = r; this.green = g; this.blue = b;
	}
	public String RGB()
	{
		return "(" + red +"," + green +","+ blue + ")";
	}
	public String toString()
	{
		String current = super.toString();
		return current.substring(0,1) +
current.substring(1).toLowerCase();
	}
}

public class MyMain{
public static void main(String[] args)
{
Color g =  Color.BLACK;
for(Color c : Color.values()) { System.out.println(c.ordinal()
+ " Color: " + c
+ " RGB: " + c.RGB()); }
}
}
```

להחליף בלוק של if-else :
בעזרת Enums :

```
enum Color {
	RED("red"){
		@Override
		public void doSomthing() {
			System.out.println("Im Red");
		}
	},
	YELLOW("yellow"){

		@Override
		public void doSomthing() {
			System.out.println("Im Yellow");
		}
	};
```

```
        private String _data;
        private Color(String s)
        {
                this._data = s;
        }
        public abstract void doSomthing();
        public static Color getByKey(String s)
        {
                for(Color c : values()) if(c._data.equals(s)) return c;
                throw new IllegalArgumentException();
        }

}

public class MyMain{
        public static void main(String[] args)
        {
                String s = (new Scanner(System.in)).nextLine();
                Color c = Color.getByKey(s);
                c.doSomthing();
        }
}
```

בעזרת Strategy / State , אם זה Instanceof אז Visitor

מה זה Reflaction:

Reflection הינו מנגנון לקבלת מידע על תוכן המחלקה: תכונות, שיטות, הרשאות וכד', מבלי שתהייה לנו גישה לקוד עצמו

מנגנון גנרי של שיקוף המידע

מנגנון זה מאפשר לנו להפעיל שיטות באופן דינאמי מבלי לדעת מראש מה יש בתוך המחלקה

הגדר מהו Iterator

Iterator הוא כלי המאפשר לעבור על כל איברי האוסף

לרשום Iterator :

כמחלקה אנונימית:

1. להוסיף

```
public class Container implements Iterable {
```

2.

```
        @Override
        public Iterator<Object> iterator() {
                return new Iterator<Object>() {
                        int pos = 0;
                        @Override
                        public boolean hasNext() {
                                return (pos >= 0 && _data[pos]!=null);
                        }

                        @Override
                        public Object next() {
                                return _data[pos++];
                        }
};
```

כמחלקה פנימית:

```
        public Iterator<Object> iterator() {
                return new ObjectIterator();
```

```
        }
        class ObjectIterator implements Iterator<Object>
        {
                int pos = 0;
                @Override
                public boolean hasNext() {
                        return (pos >= 0 && _data[pos]!=null);
                }

                @Override
                public Object next() {
                        return _data[pos++];
        }
}
```

לזכור למבחן:

- לרשום הערות /** **/
- אם יוצרים מבנה נתונים אז זה מטיפוס **ממשק**

- בעיות שיש עם מבנים לא סינכרונים:
  1. כשמוסיפים מכל תהליך מספר כלשהו אז לא מובטח המספר X כמות התהליכים
  2. ברגע שעוברים עם איטרטור על מבנה נתונים בעוד שמוסיפים איברים בתהליך יכולה להווצר שגיאה
  3. מחיקה של איברים במערך יכולה לגרום לגישה לערך שלא נמצא