סיכום oop

9 במרץ 2022

1 טיפוסי נתונים:

בג'אווה קיימים שני סוגי משתנים: 1 - משתנה פרמיטיבי ($int, double, char...$). 2 - מצביע לאובייקט כלשהו. ניתן להגדיר כמה מצביעים לאותו האובייקט, ושינוי שנבצע במצביע הראשון ישפיע על כולם.
$String$ הוא שונה מפני שזו מחלקה ולא טיפוס פרמיטיבי, אך ניתן לאתחל את המשתנה ללא יצירת $instance$.

1.1 $final$:

אם נגדיר משתנה בתור $final$ זה אומר שאי אפשר לשנות את ערכו במהלך כל זמן הריצה.
אם נגדיר מצביע אובייקט בתור $final$ אנו נוכל לשנות את האובייקט אך לא נוכל לשנות את המצביע להצביע על אובייקט אחר.
אם נגדיר מחלקה בתור $final$ אף מחלרה אחרת לא תוכל לרשת ממנה.
אם נגדיר מטודה בתור $final$ לא נוכל לדרוס אותה.

1.2 מערכים:

מערך הוא דרך של ג'אווה לשמור משתנים מסוג מסויים לדוגמא $int[]\ name\ =\ new\ int[]$
ניתן גם לאתחל אותם ישר ,כך: $int[]\ name\ =\ new\ int[]\{30, 10\}$
ניתן להגדיר את גודל המערך ע"י הכנסת מספר לתוך הסוגריים המרובעים.
כשנאתחלת מערך של $String$ אנו נקבל מערך שכולו $null$
מערכים דו מימדיים נגדיר כך $int[][]$ או כך $int[]\{\{3, 2\}, \{10\}\}$

2 $modifiers$ - מגדירים:

$modifiers$ מגדירים את השדה - האם הוא $private, public, static$:

2.1 $static$:

כאשר נגדיר שדה או מטודה בתור $static$ לצעשה אנו מגדירם אותו בתור"משתנה מחלקה" ומצהירים שהוא לא קשור למופע ספציפי אלא למחלקה.
בכדי לגשת למשתנה או מטודה שהוגדרו סטטים אנו ניגש אליהם דרך שם המחלקה ולא גרך מופע ספיפי - אחרת נקבל אזהרת קומפילציה.
מטודות סטטיות יכולות לגשת למשתנים סטטים בלבד משום שאינן מקושרות למופע ספציפי אלא למחלקה כולה, לכן אם נרצה לגשת למשתנה סטטי דרך מטודה, נצטרך להגדיר גם את המטודה כסטטית.

משתנה סטטי יאותחל במקרים הבאים:
1: כשנוצר מופע של **המחלקה.**
2: כשמבצעים השמה למשתנה הסטטי.
3: כשקוראים למטודה הסטטית.
4: כשעושים שימוש בממבר הסטטי.

2.2 $private$:

משתנה או מטודה שיוגדרו כפרטיים יהיו חשופים רק למחלקה בה הם נמצאים ולא מחלקות אחרות (גם לא מחלקות יורשות).
ככלל, אנו מעדיפים להגדיר שדות של המחלקה בתור $private$ כדי שלמשתמש לא תתאפשר גישה אליהם.
נוכל לאפשר גישהלדשדות על ידי מטודות של get/set .

2.3 $protected$:

באופן כללי מתפקד כ $private$, למעט העובדה שמחלקות יורדות יכולות לגשת לשדות או מטודות שהוגדרו כ $protected$.
באופן כללי נעדיף לא להשתמש ב $protected$ משום שהוא נכלל ב API.

3 דריסת מטודות:

3.1 $overriding$:

מספר כללים לדריסה של מטודות:
1: אם אנו דורסים מטודה - אנו צריכים לוודא שהמטודה החדשה לא מוגדרת כפרטית יותר מאשר המטודה הנדרסת.
2: לא ניתן לדרוס מטודה שמוגדרת כ $private$.
3: אם נשים את המילה השמורה $supper$ במטודה הדורסת אנו נממש את מה שמוגדר כ $supper$ כפי שהוא מוגדר במחלקת האב (דרך לשמור על השדות הרלוונטים לנו ממחלקת האב). ונוכל להוסיף עליה עוד פונקציונליות.
4: תמיד חייבים לקרוא לבנאי של מחלקת האב, בתוך המחלקה היורשת על ידי שימוש ב $supper$. אם לא נעשה זאת הבנאי הדיפולטיבי ייקרא דיפולטיבית. ואם לא קיים בנאי דיפולטיבי תיזרק שגיאת קומפלציה.

5: לא ניתן לדרוס שדות של המחלקה.

3.2 ניתן לדרוס מטודות בעלות אותו השם אפילו אם הן נמצאות יחד באותה המחלקה, אך זה מתאפשר במקרים הבאים:

1: הארגומנטים שונים בטייפ שלהם בשתי המטודות.
2: מספר הארגומנטים שונה בשתי המטודות.
3: סדר הארגומנטים שונה בן שתי המטודות - לא מומלץ לדרוס כך משום שיוצר בלבול.

4 המילה השמורה $this$:

בעזרת המילה $this$ ניתן לבצע מספר דברים.
1: לקרוא לקונסטרקטור אחר של המחלקה ולעדכן אותו דרך הקונסטרקטור החדש. כלומר נקבל מידע גרך קונ' b ונכניס אותו כך $this(other.real, other.img);$ כאשר הסוגרים מייצגות את הארגומנטים של קונ' a .
2: מאפשר לשלוח את המופע הנוכחי של האובייקט\המחלקה בתור פרמטר למטודה אחרת.

5 פולימורפיזם:

פולימורפיזם זהו עיקרון שאומר שנתן לממש אובייקט בתחתית העץ בעזרת $reference\ type$ של אובייקט אחר במעלה העץ.
לדוגמא - נוכל לממש כך $Animal\ myAnimal = myCow;$ מכיוון שפרה יורשת מחיה.
נשים לב כי באופן כה נוכל להריץ מטודות שקשורות למחלקה cow רק אם הן מוגדרות במחלקה $animal.$ (מה שיודפס אלו המטודות של המחלקה cow), אחרת תיזרק לנו שגיאת קומפילציה .
באופן כללי - ה $reference$ קובע מה מותר לנו להריץ, וה $object$ קובע מה ירוץ.
אך המקרה של שדות ומטודות סטטיות ה $reference$ בלבד קובע מה ירוץ

6 מחלקה אבסטרקטיות:

המוטיביה ליצור מחלקה אבסטרקטית היא - לאפשר שימוש בפולימורפיזם תוך שמירה על כך שאם נשכח לדרוס מטודה מסויימת תיזרק לנו שגיאת קומפילציה.
צד ניצור מחלקה אבסטרקטית: נממש כמו מחלקה רגילה אך לפניכן נשים את המילה השמורה $abstract$ כך : $public\ abstract\ class$
ואז ניצור מטודה כללית ריקה. ובכל תת מחלקה נצטרך לדרוס אותה , אם לא נממש אותה במחלקה היורשת נקבל שגיאת קומפילציה.
ההבדל העיקרי בין מח' זו למחלקות רגילות היא שלמחלקה אבסטרקטית אין $instans$ אך ניתן ליצור להן $reference.$

מספר הערות:
1: מחלקה אבסטרקטית שיורשת מחלקה אבסטרקטית אחרת לא מתחייבת לממש את כל המטודות של מחלקת האב.

2: מטודות שמוגדרות סטטיות או פרטיות לא יכולות להיות אבסטרקטיות.
3: אם ננסה לקרואלמטודה אבסטרקטית נקבל שגעאת קומפלציה.

7 ממשקים - $intarfaces$:

1: ממשק יכול להחזיק רק שני סוגים של משתנים: $final$, $static$, ומטודות $abstract$ (אין צורך להשתמש במילה $abstract$ אלא הכוונה לפונקציה לא ממושמשת) או $deafult$. השדות של האינטרפייס יכולים להיות $public$ בלבד.
2: ניתן להשתמש בהם רק עם משתנים שמוגדרים כ $final$ ומטודות $public$.
3: בדומה למחלקות אבסטרקטיות גם מ $interfaces$ לא ניתן ליצור אובייקטים והם נועדו כדי לעזור לנו לממש "חוזים" בין מחלקות.
4: כל ממשק יכול לרשת $extend$ מכמה ממשקים אחרים אך יכול לממש רק אחד ($implement$)
5:כדי לממש מחלקת ה $interface$ אנו צריכים להוסיף במחלקה היורשת את המילה השמורה $implements$ **ולא** נשתמש ב $extend$.
6: כל מחלקה יכולה לממש ($implement$) מספר ממשקים.
7:נשתמש ב $interface$לרוב כאשר אנו מגדירים מחלקה שעשה משהו - מחלקות עם הסיומת ble כלומר $printable, wtitable$ וכו'.. כלומר מחלקות שיש להן חוזה או התחייבות מסויימת.
8: ניתן גם לשים רפרנס מהטייפ של הממשק, אך נוכל לקרוא למטודות שמוכרות על ידי הממשק הספציי הזה בלבד.
מתי נממש: אם מחלקה אחת היא סוג של מחלקת האב, אזי נשתמש בירושה או במחלקה $abstract$. אך אם המחלקה היא סוג של חוזה מסויים בינה לבין מחלקת האב, אזי נגדיר את מחלקת האב כ $interface$. כשאנו לא יודעין במה לבחור תמיד נעדיף $interface$ מכיוון שניתן לרשת כמה מהם במקביל.

7.1 ממשק פונקציונלי - $functional\ interface$:

ממשק פונקציונלי זהו ממשק שמייצג **פונקציונליות** מסויימת (ולא מייצג נתונים) ומכיל פונקציה **אבסטרקטית אחת בלבד. או יחד עם מטודות שמוגדרות** $static$ **או** $deafult$ **.**
אחת הדרכים לממש ממשק פונקציונלי הוא באמצעות מחלקות אנונמיות.

הכלה - $compositio$:

נשמש ביחס הכלה כאשר נרצה לממש קוד אחר פעם נוספת אך ללא ירושה.
אנו נקרא למטודות של האובייקט ונממש אותן כשנצטרך. לכן ניתן לממש גם כמה אובייקטים במקביל.
נשתמש בהכלה כאשר נרצה למחזר קוד אך לא נצטרך להשתמש בקוד ב $polimorphisem$.
כיצד נממש: נשים במחלקה ה"יורשת" שדות מטיפוס של מחלקת האב.

8 $casting$:

זוהי דרך לקחת אובייקט מסוג מסויים ולממש אותו בתור אובייקט מסוג אחר.

קיימות שתי שיטות של $castimg$:

1: $up-casting$: נעלה מעלה במעלה העץ. כלומר - נקח אובייסט של כלב ונכניס אותו לתוך אובייקט של חיה, שהוא אובייקט כללי יותר.

2: $doun-casting$: נרד מטה בעץ. נקח אובייקט של חיה ונממש אותו בתור אובייקט של כלב. שהוא אובייקט ספציפי יותר.

באופן כללי אנו נעדי להיממע מ $doun-casting$ מכיוון שזה יכול ליצור לנו בעיה, בנוסף אנו תמיד שואפים להשתמש באובייקטים כליים יותר ולא באבייקטים ספציפים.

$instanceOf$:

זאת שיטה מובנית ב $java$ לבדו האם אובייקט מסויים הוא סוג של אובייקט אחר. אך אנו נעדיף להימנע משימוש בשיטה זו מפני שהיא מגבילה לנו את המודולריות של הקוד והיכולת לשינוי.

9 $design-paterns$:

9.1 האצלה$-delegation$:

דלגציה הוא דפוס עיצובי המתעסק במבנה הקוד שעוזר לנו לפתור בעיות מסויימות, והוא סוג של הרכבה בין הכלה לירושה.

כיצד נממש אותו: בתוך המחלקה שאיתה נרצה לעבוד - B, נגדיר אובייקט של המחלקה שאמורה להיות מחלקת האב - A.

אנו לא נממש את זה בצורה של ירושה.

1: ניצור את האובייקט של מחלקה A בתוך מחלקה B.

2: נגדיר משתנה בקונסטרקטור שיקבל את האובייקט A.

3: כשנרצה להשתמש במטודות של A ניצור מטודה במחלקה B ונקרא למטודה של Aכך: $a.foo()$

בך למעשה נממש את מחלקה A בתוך מחלקה B בלי לרשת אותה כלל.

9.2 $facade$:

דפוס עיצובי המתעסק במבנה של הקוד - הפשטת API מורכב של מלא מחלקות ל API יחיד ופשוט יותר.

מתי נממש: אם יש לנו קוד שמורכב ממלא מחלקות ולכל אחת מהן API שונה, אנו נרצה לבנות מחלקה ראשית עם API פשוט יותר, שמאגדת תחתיה את כל שאר המחלקות הקטנות. וכך נקבל API ברור ויחיד.

9.3 $factory$ מפעל:

זהו דפוס עיצובי העוסק ביצירה של אובייקטים.

למעשה אנו רוצים להפריד בין החלק בקוד שמייצר את האובייקטים לבין החלק שבו אנו משתמשים בהם. בדומה למפעל שמייצר דברים אצלנו יש את החלק בקוד שייצר את האובייקטים.

9.4 $singelton$:

זהו דפוס עיצובי העוסק גם כן ביצירה של אובייקטים.
העיקרון המנחה של הדפוס הוא - שיהיה לאובייקט שלנו מקום יחיד שבו הוא מופיע.
כיצד נממש:
1: ניצור אובייקט ונשמור אותו בתוך משתנה סטטי.
2: ניצור מטודה סטטית שתחזיר לנו את האובייקט.
3: נגדיר את ה $constractor$ של המחלקה בה או יוצרים את האובייקט כ $private$, כך שלא יוכלו לגשת אליו ממחלקות אחרות וליצור מופעים שלו.
אנו רוצים שהבנאי יוגדר כ $private$ מכיוון שאנו לא רוצים לאפשר ירושה.

9.5 $strategy$:

זהו דפוס עיצובי העוסק בהתנהגות.
כאשר יש לנו בקוד חלק שמתנהג אחרת, לדוגמא - קיימת מחלקה שעוסקת באלגוריתמים מסויימים אזי נפריד אותם ונממש כך.
כיצד נממש:
1: נגדיר מחלקה שתהיה המחלקה עם כל אלגוריתמי המיון שאנו רוצים לממש (אפשר לממש אותה כ $interface$ או כמחלקה $abstract$)
2: כל מחלקה תירש את המחלקה של האלגוריתמים ותתחייב ל APIשלה .
3: ניצור מחלקת $factory$ שמקבלת את מחלקת האלגוריתמים.
4: כשנרצה לממש אלגוריתם נעשה את זה דרך מחלקת ה $factory$.

10 $collection$:

$collection$ הוא אובייקט שמחזיק אובייקטים ומאפשר לעשות עליהם כל מיני מניפולציות. הם אובייקטים גנרים - $generic$ וכשיוצרים אותם אנו למעשה יוצרים אובייקט.
בג'אווה ניתן למש את הקולקשיין על ידי ספריה מיוחדת שבה יש $interface, implenents, algorithems$
1: $interface$ - מחזיק כל מיני מבני נתונים כגון רשימות, מפות וקבוצות.
2: $implements$ - מחזיק מבני נתונים יותר ספציפיים של מבני הנתונין הנ"ל - $hash-set, tree-set, linked-list$.
3: $algorithems$ - אלגוריתמי חיפוש ומיון.

10.1 מערכים:

מערכים היא דרך של ג'אווה לממש כל מיני מבני נתונים, אך זוהי דרך פרמיטיבית למדיי מפני שצריך להגדיר לפני השימוש את גודל המערך והוא לא ניתן לשינוי.

10.2 הרחבה על $collections$:

מתי נשתמש במבנה נתונים מסויים: אנו צריכים להסתכל על זמן הריצה שאנו מחפשים, מהן הפעולות אותן אנו רוצים לבצע והאם יש כפילויות באיברי המערך או לא. בנוסף אם יש מפתחות וערכים נבחר ב map

10.2.1 רשימות $lists$:

ברשימות אנו מאפשרים כפילויות של איברים, קיים סדר וניתן לגשת לאיבר ספציפי.
רשימה מקושרת: לוקחת פחות זיכרון מאשר רשימה רגילה, אך זמן הריצה שלה הוא $O(n)$

10.2.2 תור $queue$:

יש איבר שנמצא בראש התור, בד"כ נממש בצורת $FIFO$.
ניתן להוציא את האיבר הראשו מראש התור ע"י $pop()$ ולהכניס איבר לתור ע"י $poosh()$ ולדעת מי נמצא בראש התור ע"י $peek$.

10.2.3 קבוצה set:

זהו מבנה נתונים כמו רשימה אך הוא לא מאפשר כפל של איברים - כל איבר נמצא פעם אחת בלבד במערך.
$tree-set$: שומר את האיברים בצורה ממויינת ולכן זמן הריצה שלו הוא $O(log(n))$. יש צורך להכניס לו איברים ברי השוואה $comperable$ או לממש בבנאי שלו $comperator$ אחרת נקבל שגיאת זמן ריצה.
$hash-set$: שומר את האיברים ללא סדר. זמן ריצה של $O(1)$

10.2.4 map:

משמש כמילון בפייתון. מקבל ערך ומפתח כאשר כל מפתח יכול להימצא ב map פעם אחת בלבד.
$tree-map$: שומר את האיברים בצורה ממויינת ולכן זמן הריצה שלו הוא $O(log(n))$. יש צורך להכניס לו איברים ברי השוואה $comperable$ או לממש בבנאי שלו $comperator$ אחרת נקבל שגיאת זמן ריצה.
$hash-map$: שומר את האיברים ללא סדר. זמן ריצה של $O(1)$

10.3 $hash-table$ - טבלאות גיבוב:

טבלה שעוזרת לנו למיין מילים או ערכים לפי פונקיית האש המוגדרת מראש.
עדיף לדרוס את המטודה $hashCode()$ ואת המטודה $equals(object\ o)$ של האובייקט שאנו מוסיפים לאוסף. כדי שנקבל את התוצאה הרצויה.

10.4 $iterators$ - איטרטורים:

זהו אובייקט שלוקח מבנה נתונים ועובר על האיברים שלו אחד אחד . יכול לשמש למעבר על כל מבני הנתונים.

10.5 השוואות בין אובייקטים:

10.5.1 דריסת מטודת $equals()$ של מחלקת $object$:

כשנדרוס את המטודה $equals$ נצטרך לשים לב שהדברים הבאים נשמרים:

1: כל אובייקט שווה לעצמו.

2: סימטריה - אם $a = b$ אז גם $b = a$

3: טרנזטביות - אם $a = b$ וגם $b = c$ אז $a = c$.

4: אף אובייקט לא שווה ל $null$.

10.5.2 דריסת מטודת $hashCode()$ במחלקת obj:

כשנדרוס את המטודה הנ"ל נצטרך לשים לב ש:

1: המטודה מחזירה ערך האש שווה עבור ערים שווים.

2: השדות שלפיהם אנו בודקים את ערך ההאש יוגדרו כ $final$ בכדי שנוכל להשוות ללא שינוים.

בכדי שנוכל לסדר אובייקטים במערכים מסויימים ($tree$ למשל) נצטרך לדאוג לכך שהם יהיו ברי השוואה, ולדאוג לדרך שנוכל להשוות בניהם. כאשר האובייקטים הם מורכבים להשוואה יש שתי דרכים לעשות זאת:

10.5.3 נממש את ה $interface$ הבא הנקרא $comperable$.

נוכל לבצע השוואות בעזרתו. נצטרך להגדיר שהמחלקה יורשת אותו ($implement\ comperable$) ולדרוס את המטודה של ההשוואה $public\ int\ compareTo()$שמוגדרת בו (זאת מטודה שמחזירה 0 אם מתקיים שוויון ו 1 או 1- אחרת),
נעשה זאת כך: נקרא למטודה עם האנוטציה $@override$ ונגדיר את ערך ההחזרה שלה להיות האופן שבו אנו רוצים לבצע את ההשוואה בין שני האובייקטים תוך שימוש במטודה $compare()$.

10.5.4 ניצור $comperator$:

באותו האופן של החלק הקודם. רק שכאן נממש $interface$ הנקרא $comperator$ נירש אותו ונדרוס את המטודה $compare()$ ונחזיר ערך השוואה כרצוננו (כאן אין צורך להשתמה במטודה $compare()$ בכדי להשוות)

ההבדלים בין השיטות: ל $comperator$ (השיטה השניה) אין גישה לשדות פרטיים של המחלקה שעשוייסלהיות רלוונטים למיון. אך הוא יכול לשמש בכמה אופנים ותורם למחזור קוד כך שניתן לבחור בזמן ריצה איזו אסטרטגיית מיון לממש.

11 $exeptions$ - חריגות:

אנו נשתמש בחריגות כאשר נרצה לתפוס שגיאות זמן ריצה או שגיאות שנובעות כתוצאה מהכנסת קלט שגוי מהמשתמש. המטרה היא לזרוק את השגיאה חזרה במעלה העץ בכדי שהתוכנית לא תקרוס.

חריגה היא אובייקט, לכן אנו נצטרך ליצור חריגה חדשה לפני שנזרוק אחת - $new\ error$. ובמטודה שמשתמשת בחריגות נצטרך להוסיף את המילה השמורה $throw$.
אופציה נוספת היא להשתמש בבלוק $try - catch$ ואז לא נצטרך להשתמש במילה השמורה $throw$.
מנגנון החריגות עוזר לנו לפעפע מעלה את השגיאה ולהפריד בין החלק של הקוד לחלק של החריגות.
בנוסף, אנו יכולים לכתוב הודעת שגיאה ספציפית או להשתמש בשגיאות של ג'אווה.

11.1 מחלקת חריגות חדשה:

תמיד כשנרצה ליצור מחלקת $exeption$ חדשה נצטרל להגדיר בתוכה את מספר הגרסה כך:
$private\ static\ final\ long\ SerialVersionUID\ =\ 1L$

12 $packeges$ חבילות:

חבילות הן כמו תיקיות בג'אווה, כאשר שתי מחלקות נמצאות באותה החבילה הן יכולות לגשת אחת לשניה ללא ייבוא המחלקה האחרת לפניכן. אם הן לא נמצאות האוצה החבילה נצטרך לייבא אותן טרם השימוש.
אם המחלקה מוגדרת ללא $modifier - (public, protected...)$ אזי רק מחלקות שנמצאות איתה באותה החבילה יכולות לגשת אליה.
כשנשתמש ב $exeptions$ נעדיף להפריד כל חריגה לחבילה נפרדת.

אם מחלקה A לא נמצאת באותה החבילה עם מחלקה B אזי היא לא יכולה לגשת שדות של B גם אם היא תייבא את B, אם היא תנסה לעשות זאת תיזרק לנו שגיאת קומפילציה.

13 ביטויי $lambda$:

נין לממש פונציות בצורה הבאה $(intx, inty) - > returnx + y;$ וכך למיעשה נקבלאת החיבור בניהם.
$cimpera$ בדרך הבאה $Comparator < List > comparator = (s1, s2) - > Integer.compare(s1.length(), s2.length()));$.

14 $enums$:

הקונספט של $enums$ מאפשר לנו להגדיר מחלקה עם ערכים ספציפים.
נגדיר זאת כך: $public\ enum\ name\ \{type1, type2\}$ כאשר ב $type$ אנו שמים את הערכים שאנו רוצים לקבל - אלו יכולים להיות גם אובייקטים ספציפיים.
נשצש ב $enum$ כאשר כל הערכים ידועיים מראש.אם ננסה להכניס משתנה שלא הוגדר אנו נקבל שגיאת קומפלציה.
לא ניתן לשנות את הערכים בזמן ריצה.
בדומה לגנריות גם כאן זהו $type - safety$ שמעביר שגיאות זמן ריצה לקומפלציה.
ניתן לממש לולאות בעזרת $className.vakues()$

ל $enum$ יש קונסטרקטור $private$ או דיפולטיבי בלבד.
כ"כ לא ניתן ליצור מופעים של $enum$ מחוץ למחלקה.

15 מחלקות מקוננות - $nested-classes$:

מחלקה מקוננת זוהי מחלקה שנמצאת באוותו הקובץ עם מחלקה אחרת. ויש לה גישה לכל המשתנים של המחלקה העוטפת ולהיפך.

מדוע להשתמש במחלקות מקוננות:

1: זה שומר על החבאת המידע - האנקפסולציה. המחלקות יכולות לגשת אחת לשדות של השניה. וכן ניתן להגדיר את מחלקה המקוננת בתור $private$.

2: זה שומר על קוד מאורגן - הקוד מסביר את עצמו.

3: אם המחלקה המקוננת רלוונטית רק למחלקה שבה היא מקוננת. אזי נוכל להגדיר את המקוננת בתור $private$ ולא בתור $public$ באופן כללי נעדיף להגדיר מחלקות מקוננות בתור $private$.

מתי נשתמש במחלקות מקוננות:

1: כאשר אנו צריכים להגדיר מחלקות קטנות ואין טעם ליצור קובץ נוסף.

2: כאשר אנו מגדירים מחלקה בתור $pruvate$ אז נרצה שהיא תהיה מחלקה מקוננת.

15.1 סוגי מחלקות מקוננות:

15.1.1 מחלקה מקוננת סטטית:

זוהי מחלקה רגילה שמוגדרת ב $public\ static$, למעשה היינו יכולים לשין אותה בקובץ נפרד, אך מצורכי חבילות איחדנו אותן.
אין קשר בין המופעים של שתי המחלקות וכדי לקרוא למחלקה העוטפת נצטרך ליצור $instance$ חדש (מכיוון שיא סטטית היא לא יכולה לגשת לשדה במחלקה העוטפת שלא מקושר למופע ספציפי).

15.1.2 $inner-class$:

מחלקה פנימית שאינה $ststic$ והיא מקושרת ל $instance$ **ספציפי** של המחלקה העוטפת. ומיכוון שהיא מקושרת ל $instance$ ספציפי אזי היא לא יכולה להגדיר משתנים סטטים.

1 - $member\ class$: נצטרך ליצור $instance$ של המחלקה העוטפת ולאחר מכן ניצור $instance$ של המחלקה המקוננת בכדי להשתמש בה.

2 $local\ class$: מחלקה מקוננת הנמצאת בתוך מטודה, היא מתנהגת כמשתנה מקומי ומוגדרת רק בתוך המטודה. נשתמש בשיטה זו כאשר יש לנו מחלקה שרלוונטית רק בתוך המטודה.
מחלקה לוקאלית יכולה לגשת למשתנים של המטודה רק אם הם מוגדרים כ $final$. אחרת - תיזרק שגיאת קומפילציה.
בגלל שהיא מחלקה שרלוונטית רק בתוך המטודה , אזי אין שום משמעות ל $modifier$ שלה.
לא ניתן לשים $interface$ בתור מחלקות פנימיות.

לא יכולה להכיל משתנים סטטיים כי היא לא מחלקה סטטית.

15.1.3 $anonimus\ class$:

זוהי מחלקה אנונימית - ללא שם. למעשה אנו יוצרים $instance$ של $interface$.
ניתן בעזרתה לממש $interface$ או לרשת מחלקה אבסטרקטית.

16 מודולריות:

הרעיון של מודולריות הוא לפרק את הקוד לחלקים קטנים, וכך להפוך אותו לקל לתחזוק ולפירוק בין משימות או צוותים.
קיימים 4 עקרונות במודולריות:

16.1 $Decomposability$ - פריקות:

פירוק בעיה גדולה לתתי בעיות קטנות וטיפול בכל אחת מהן בנפרד. וחיבורן באופן פשוט לאר מכן.

16.2 $Composability$ - הרכבה:

חיבור כמה יחידות קוד קטנות לכדי יחידה אחת שלמה - לדוגמא שימוש בחבילות.
כל חלק אמור להיות אוטונומי ובלתי תלוי בחלקים האחרים.

16.3 $Understandability$ - מובנות:

כל חלק בקוד עומד בפני עצמו וניתן להסביר את העולה שלו שכמה מילים בודדות.

16.4 $Continuity$ - רציפות:

במידה ונרצה לשנות את הקוד, נצטרך לשנות יחידה אחת בלבד ולא את כל הקוד.

16.5 עיקרון ה $open - close$:

עיקרון זה אומר שהקוד אמור להיות סגור לשינויים אך פתוח להרחבה. כלומר ברגע שנרצה לשנות אותו זה יהיה קל לביצוע.

16.6 עיקרון ה $single - choise$:

עיקרון שאומר שאם נרצה לבצע שינוי בקוד נבצע אותו במקום יחיד, ולא נצטרך לבצע את השינוי בכמה מטודות כי אז זה מסרבל ומקשה על שינויים.

17 $streams$:

זוהי הדרך של ג,אווה לתקשר עם input, output שאנו מקבלים מהמשתמש כקלט.
קיימים שני סוגים של קלטים:
1: קלט שכתוב בערך סטרינגי כלומר קריאת קבצים, - נממש בעזרת ספריה מובנית בגאווה ע"י $writer,\ reader$.
2: קלט בינארי. נממש בעזרת ספריה מובנית בגאווה ע"י $outputStream,\ inputStream$. הן מחלקות אבסטרקטיות ויש מס' מחלקות שיורשות מהן.
בכדי לעבור עם כל אחת מהמחלקות הללו נצטרך ליצור אינסטנס שלהן ע"י שימוש ב new .
אנו נעדיף להשתמש בבלוקי $try - catch$ בכדי לתפוס חריגות של קובץ לא קיים או לא קריא.

17.1 $decorator$ - מקשט:

יצירת מחלקת $decorator$ שהיא למעשה המחלקה המקשטת - D של המחלקה שתקרא את הקבצים - R. היא צריכה להגדיר את אותו ה API של המחלקה הקוראת - R.
זהו למעשה שילוב של $delegation$ ו $wrraper$. המחלקה העוטפת תשמור לנו באפר עם הטקט שאנ רוצים לקרוא או לכתוב ואחרי שהבאפר יתמלא היא תתחיל לכתוב. כך למעשה אנו חוסכים בזמן של לכתוב כל פעם ישירות לקובץ ה $output$.
למחלקות מקשטות אין שימוש משל עצמן והן מייצגות מידע או פונקציונליות מסויימת.
כיצד נממש:
1: אנו יוצרים אובייקט של המחלקה שקוראת את הקובץ - R, במחלקה D.
2: כשנרצה לבצע פעולות נבצע אותן דרך האובייקט r של המחלקה R שנמצר במחלקה D.

18 $generics$:

שימוש בשיטה זו יעזור לנו העביר שגיאות זמן ריצה לשגיאות קומפילציה. שיטה זו באה להגדיר $type - sefty$ תכונה שהופכת את הקוד לנקי מבאגים.
ניתן להשתמש ב $generix$ גם במחלקות, מטודות ואובייקטים ולא רק ב $colections$
אך לא ניתן להכניס מערכים $(String[])$ ברשימות עם $generics$.
כיצד נממש: כשניצור אינסטנס של האובייקט נשים לייד ה $type$ סוגריים משולשים עם סוג משתנה אותו אנו רוצים שהאובייקט יקבל. לדוגמא - $private\ list < String >= new\ list <>$ כך למעשה הגדרנו שהרשימה שלנו תקבל משתנים סטרינגים בלבד.
ניתן לשים כמה סוגי משתנים בתוך הסוגריים המשולשים ולא רק 1 .

18.1 מטודות גנריות:

ניתן לשים משתנה גנרי גם בתוך מטודה.
אם נשנה את הטייפ שלו לטייפ שונה מהמשתנה הגנרי של המחלקה, אנו נקבל משתנה גנרי אחר.
אם המטודה מוגדרת כ $static$ אזי היא לא תקבל את המשתנה הגנרי אך אנו נוכל להגדיר לה משתנה גנרי.

18.2 מחלקות גנריות:

כשנרצה לממש מחלקה גנרית, נציב לייד השם של המחלקה $< T >$ וכך למעשה נייחד את המחלקה לקבל טייפ מסויים של אובייקטים.
לא ניתן להכניס בתור פרמטר גנרי טיפוס פרימיטיבי, לכן אם נרצה להכניס int נצטרך להשתמש ב $Integer$.
כלומר - שניצור $instance$ של המחלקה נשים בתוך הסוגריים המשולשים את ה $type$ של המשתנה אותו אנו רוצים שהמחלקה תקבל.
בנוסף אנו יכולים לכתוב ישירות לייד שם המחלקה את ה $type$ של האובייקט ואין צורך להשתמש ב $< T >$.
מחלקה מקוננת סטטית לא תקבל את המשתנה הגנרי של המחלקה העוטפת, אך ניתן להגדיר לה משתנה גנרי.

נשים לב כי אנו יכולים מבחינה קונספטואלית להוסיף רשימה מקושרת של סטרינגים, לרשימה מקושרת שמכילה אובייקטים.
אך $up-castig$ בניהם היא פעולה אסורה שתזרוק שגיאת קומפילציה, משום שקיימת היררכיה בין הטיפוסים, אך לא קיימת היררכיה בין המחלקות, והן אינווראנטות.
בכדי שנוכל לממש את זה הומצא הדבר הבא:

18.3 $wild-cards <?>$:

$wild-card$ ממומש באופן הבא: במקום לשים בסוגריים המשולשים את ה $type$ של המשתנה. נשים "?" וכך למעשה נגדיר את הרשימה להיות רשימה של אובייקטים שאין לנו עניין ב $type$ שלהם.
נשים לב! מכיוון שאנו לא יודעים מה הסוג של האובייקט, ניתן להוסיף לרשימה רק $null$. וכשנשלוף איברים אנו יכולים לשלוף **רק** אובייקטים.
הערה: $up-custing$ ו $wild-cards$ עובדים יחד.
פעולות אפשריות עם $wild-card$**:**
1: אם נרצה להגדיר מטודה שמקבלת אובייקטים אך אנו לא יודעים מה הסוג.
2: אם יש לנו רשימה של אובייקטים ואנו לא יודעים מה הסוג. אז נגדיר את המטודה עם $<?>$.
3: מכיוון שהרשימה הנ"ל מקבלת ערכי $null$ בלבד, אזי ניתן להפוך את זה לפיצ'ר במקרה שאנו לא רוצים להוסיף דבר לרשימה מלבד הערך $null$.
4: אם נרצה להשוות בין שתי רשימות שאנו לא יודעים את ה $type$ של המשתנים שלהן.
כיצד נממש: ניצור אינסטנס כך $LinkedList <?> = new\ LinkedList < Type >$ כאשר לתוך $Type$ אנו ציכים להכניס סוג ספציפי של משתנה. (רק $reference$ יכולים להשתמש ב "?" אך האובייקט הקונקרטי צריך להכיל משתנה ממשי) .

18.3.1 $extend$:

פעולה נוספת שניתן לבצע היא $LinkedList <? Extend\ Animal >$ כאשר $Animal$היא מחלקת אב שיורשות ממנה כמה מחלקות. נממש בצורה זו כאשר כל האובייקטים של הרשימה המקושרת הם אובייקטים שיורשים מ $Animal$.
בצורה זו ניתן גם לממש $up-custing$ לדוגמא $LinkedList <? Extend\ Animal > = \ new\ LinkedList < Dog >$

נשים לב כי סימן השאלה $<? \ Extend \ Animal>$ נחוץ כאן. אם לא נשים אוו תיותוצר לנו בעיה עם האינווראנטיות.

18.3.2 $super$:

באותו האופן ניתן לממש עם $super$ כך: $LinkedList <? \ super \ Animal>$ במקרה זה הפרמטר הגנרי יהיה מחלקות ש $Animal$ יורשת מהן.
נוכל לשלוף איברים של $object$
וניתן להוסיף לרשימה איברים של כל המחלקות שיורשות מ $Animal$

18.4 $erasure$:

לאחר שאנו משתמשים ב $generics$ ג'אווה משנה בחזרה את הערך ל $object$ כך שבעצם לא נשמרת כל הפרדה באחורי הקלעים. אך שיטה זו עוזרת לנו להבדיל בין סוגי המשתנים ושומרת לנו על הקוד מטעויות זמן ריצה.
לכן נצטרך לשים לב למספר דברים:
1: כשנרצה לשוות הין מחלקות נקבל שהן שוותאפילו שה $type$ הגנרי שלהן שונה בגלל פעולת ה $erasuer$
2: אין סיבה להשתמש ב $doun - custing$
3: אין צורד להשתמש ב $instanceOf$

19 ביטויים רגולרים:

ביטוי רגולרי הוא ערך שניתן להשוות אליו סטרינגים ולבדוק האם הם עומדים בתנאי הביטוי הרגולרי.

19.1 שיטות לחיפוש ביטוי רגולרי:

1: שיטה ראשונה היא השיטה החמדנית - היא מחפשת כמה שיותר מהביטוי ועושה $backtreacking$. כשהוא מגיע לסוף הביטוי הוא מחזיר האם הביטוי נמצא . **נממש ע"י** - נשים את הביטוי ב [] ואחריו לא נציב כלום.
2: השיטה הרכושנית - מחפשת את החלק המתאים בביטוי ולא עושה $backtreacking$. **נממש ע"י** - בוספת הסימן "+" אחרי הסוגריים המרובעים שבהם נמצא הביטוי
3: השיטה המינימליסטית - מחפשת רק את המינימום ההכרחי בביטוי **נממש ע"י -** נציב אחרי הסוגריים המרובעים "?"

מתי נשתמש בכל שיטה:
1: נשתמש בשיטה החמדנית כאשר יש חפיפה בין הביטויים. לדוגמא נרצה למצוא מילים עם סיומת ing
2: נשתמש בשיטה הרכושנית כאשר אין קשר בין הביטויים. לדוגמא חיפוש סיסמאות שמכילות מילים ומספרים.
3: נשתמש בשיטה המנימליסטית כאשר הרכושני לא רלוונטי והביטוי שצריך לחפש הוא קצר.

כללי אצבע לכתיבת ביטוי רגולרי:
1: נכתוב ביטויים קצרים
2: נמנע בשימוש באופרטורים כדוגמת - | . ואם נצטרך להשתמש בהם נציב את הצפוי בחלק הראשון של הביטוי.

3: נחפש בידיוק מה שצריך ולא נציב נקודות שמבטאות כל תו, אלא נמקד את החיפוש.
4: נשתדל לצמצם חיפושים ע"י שימוש בסוגריים.
5: כשאפשר נעדיף להשתמש בכמת הרכושני שלא משתמש ב $backtreacking$.
6: קיימת דרך חיפוש שנקראת $capturing$ שמשתמשת בסוגריים בשביל לחפש כמה פעמים הביטוי מופיע. נעדיף שלא להשתמש בבדיקה זו כי היא ארוכה.

19.2 כיצד נממש בג'אווה :

בג'אווה קיימות שתי מחלקות - $pattern$ ו $macher$.
המחלקה הראשונה מקבלת את הביטוי הרגולרי, והמחלקה השניה בודקת את הביטוי הרגולרי מול הערך הסטרינגי אותו אנו רוצים לבדוק.
כדי לקרוא לשתי המחלקות נצטרך ליצות $instance$ שלהן.
לאחר מכן נוכל לבדוק התאמה כך:
1: $mach.maches$ יחזיר לנו ערך בולאני אם הביטוי כולו מתאים או לא.
2: ניתן באמצעות המטודה $find()$ למצוא את החלק בסטרינג שמצאים לביטוי. ואף העתן לחתוך את החלקים המתאימים מהביטוי ע"י לולאה של $while(find())$ ואז להשתמש במטודות $start(),\ end()$ שיחזירו לו את האינדקס שבו הביטוי תואם מתחילתו עד סופו. **נשים לב:** אם נקרא למטודות $start(), end()$ והערך לא ימצא - תיזרק שגיאה.
3: באמצעות המטודה $lookingAt()$ אנו יכולים לבדוק התאמה לחלק מהביטוי.

20 $serializetion$ סרלזציה:

זוהי דרך בג'אווה לשמור אובייקטים. הערכים נשמרים כך שהמחשב יוכל לשמור אותם. כלומר לא בערך סטרינגי.
אם אנו רוצים לשמור אובייקטנצטרך להשתמש במילה השמורה - $serializable$ זהו למעשה $interface$ שנצטרך לממש.
בנוסף כל הסדות יצטרכו להיות $serializable$ או פרמיטביים.
הערה חשובה: כל אובייקט נשמר פעם אחת בלבד בכדי שנוכל להתמודד עם לולאות אינסופיות במקרה שבו כל שדה מוביל לעצמו שוב. לכן אם נשמור אובייקט בפעם השניה ישמר אליו מצביע. ואם נבצע שינוי באובייקט השני הוא **לא** ישמר אלא האובייקט הראשון ישמר. (נוכל לעקוף זאץ על ידי $out.reset$ ששוכחת את מה שנשמר עד כה או $out.close()$ שסוגר את ה $stream$)
תהליך הקריאה והכיבה מתבצע באמצעות $streamer$ ומשתמש במחלקות $input/output - stream$
אנו עובדים בשיטת $FIFO$ כלומר האובייקט שנכתב ראשון, יקרא ראשון.

אם לא נרצה ששדה מסויים ישמר : נגדיר אותו בתור $transient$. זוהי מילה שמורה שמסמלת שאנו לא רוצים לשמור את השדה הזה.
באותו האופן שדות שיוגדרו כ $static$ גם לא ישמרו מכיוון שהם שייכים למחלקה ולא לאובייקט ספציפי.
אובייקטים פרמיטביים: בכדי לשמור אובייקטים פרמטיביים אנו צריכים להשתמש ב $out.write\ type$ כאשר $type$ מייצג את סוג המשתנה. (משתנים פרמטביים נשמרים ללא בעיה)

גרסאות: מכיוון שלא ניתן לשנת סוג של שדה או את ההיררכיה, אנו נצטרך לעדכן את ה $serialVersionUID$ זוהי הגרסה שנשמרת על ידי ג'אווה בכל פעם. ואם לא שינינו את הגרסה נעדכן ידנית לאותו מספר גרסה. כך נוכל לשלוט באילו שינויים אנו רוצים להכניס לגרסה ואיזה לא.

21 $cloning$ - שכפול:

כדי לממש את ההעתקה נצטרך להשתמש ב $interface$ שנקרא $clonable$. ואחכ לממש את המטודה $clone()$ נשים לי כי המטודה הזאת מבצעת העתקה שטוחה $shellow\ copy$ ואם נרצה לבצע $deep\ coppy$ נצטרך לדרוס את המטודה הזאת.
דרך נוספת לבצע את ההעתקה היא להשתמש ב $copyConstructor$. ניצור אובייקט חדש ונבצע את ההעתקה עצמאית לאובייקט החדש עם המטודה $clone()$.

22 $reflections$:

קיים בג'אווה אובייקט בשם $Class$ שדרכו אפשר לגשת למחלקה ולשאול עליה מספר שאלות. כגון: מה שם המחלקה , אילו מטודות יש לה וכו'..
בנוסף אנו יכולים לגשת למטודות ושדות שמוגדרים כ $private$.
מאפשר לנו לכתוב תכנה גמישה יותר ולדבג בקלות, אך גם יכול לגרום לטעויות ולהרוס את הקוד, בנוסף זה מנוגד לעיקרון האינקפסולציה