**OOP – פרק 3**

**3.1 מנגנונים בתכנות מונחה עצמים - ממשקים ופולימורפיזם:**

* פולימורפיזם אומר שיש לנו קונספט של אובייקט אבל הוא רב צורתי – יכול לבוא לידי ביטוי בכל מיני צורות.
  + מבחינת הקוד הוא יכול להסתכל על אובייקט ולא אכפת לו מאיזה סוג הוא יהיה
* **מוטיבציה לממשקים – עולמה של אנה:**
  + ממשק = interface
  + ממשק הוא רשימה של שיטות
  + נגדיר ממשק עם המילה השמורה interface, שם הממשק וב{} נגדיר את השיטות הצריכות להיות לעצם מאיזושהי מחלקה כדי שיהיה ניתן להשתמש בו.
  + מחלקה צריכה לממש את הממשק – implements the interface
    - המחלקה תצהיר שהיא מממשת את הממשק – נוסיף בשורת ההצהרה

Class ClassName implements InterfaceName {}

* + - בתוך המחלקה חייבות להיות השיטות המוגדרות בממשק עם אותם נתונים
* **מדוע כדאי להשתמש בממשקים?**
  + ממשקים משמשים להגדרת סוג של חוזים / הסכמים שמחלקות מקבלות על עצמן.
  + בניגוד למחלקות שמייצגות משהו בעולם / בעולם התוכנה, ממשקים מגדירים איזושהי תכונה או משהו שמחלקות לוקחות על עצמן
  + דוגמאות –
    - Printable – ניתן להדפיס אותם
    - Comparable – ניתן להשוות אותם
    - Clonable – ניתן לשכפל אותם
  + הממשקים מדברים על מה עושים אך לא על איך עושים.
* ממשק הוא למעשה ציון של API מסוים, ולכן גם אם לא כתוב בממשק שהשיטה פומבית הכוונה היא שבמחלקה המממשת השיטות האלו חייבות להיות פומביות.
* **Upcasting – המרה כלפי מעלה:**
  + איך מעדכנים שיטה שתעבוד על כל אובייקט מחלקה שמממשת את המחלקה?
  + InterfaceName c – כל מופע של מחלקה שמממשת את הממשק
  + כלומר סוג הרפרנס הוא סוג הממשק והוא יכול להצביע על עצם שהוא מופע של מחלקה קונקרטית כל עוד המחלקה מממשת את הממשק.
  + למעשה יש לנו המרת טיפוס (מתקמפלת רק אם מוצהר שהמחלקה מממשת את הממשק) – InterName c = new ClassName()
    - בדומה אפשר לבצע השמה בין משתנה מסוג המחלקה המממשת ולהשם אותו במשתנה מטיפוס הממשק
    - כמו כן ניתן להגדיר שיטה שמקבלת משתנה מטיפוס הממשק ולשלוח לו משתנה מטיפוס המחלקה המממשת.
  + בגלל שהממשק הוא המכנה המשותף בין מחלקות אנו חושבים על הממשק כטיפוס מעל המחלקות שמממשות אותו, וההמרה מטיפוס נמוך לגבוה נקראת המרה מעלה – upcast
  + המטרה של רפרנס מסוג ממשק היא להיות מסוגל לקבל מופעי מחלקות שמממשות את הממשק.
  + יש לי רפרנס מסוג ממשק –
    - לעצם הקונקרטי עליו הוא מצביע בהכרח יש את השיטות המוצהרות בממשק והן בהכרח פומביות, כלומר נוכל להשתמש בהן
    - דרך רפרנס זה תמיד נוכל לקרוא לשיטות אלו.
    - שום דבר אחר לא מובטח על רפרנס זה ולכן לא נוכל לקרוא לשיטות אחרות.
    - כלומר מהרפרנס נוכל לקרוא רק לשיטות של הממשק!
  + UPCAST (המרה כלפי מעלה) - היא כאשר מצביעים על עצם דרך רפרנס מסוג של ממשק, אחרי UPCAST ניתן לקרוא רק לשיטות של הממשק.
  + המרה בכיוון ההפוך – המרה כלפי מטה, DOWNCAST – היא הרבה פחות שימושית.
* **מימוש מספר ממשקים -** מכיוון שממשק מתאר התנהגות מסוימת של עצם, אין מניעה עקרונית שעצם עם מספר התנהגויות יממש מספר ממשקים, אם כי נדיר לראות יותר משניים.
* **ממשקים בהסתכלות שונה:**
  + Interace revised – דרך אחרת להסתכל על ממשקים
  + איך ניתן לקחת ממשקים ולהסתכל עליהם כזוויות שונות של אותו אובייקט
  + דוגמא – חוקרת בחוג טכנולוגי:
    - יש לה הרבה ממשקים עם העולם – מול סטודנטים, חוקרים, עמיתים בפקולטות אחרות. ממשק נוסף שונה מול הסטודנטים כמרצה, ממשק מול הילדים כאמא, ממשק מול החברים.
    - כל ממשק מייצג סט תכונות שונה של החוקרת – ניתן להסתכל על כל תפקיד כממשק שונה.
    - למשל עבור התלמידים אם יבוא מרצה אחר להחליף אותה וישתמש באותו ממשק זה לא ישנה יותר מידי לסטודנטים.
    - כל ממשק קשור לתפקיד אר ואינו מתייחס לממשקים האחרים.
* **פולימורפיזם – רב צורתיות:**
  + פולימורפיזם בביולוגיה הוא כאשר יש זן עם תכונה ולה יש כמה אפשרויות שונות
    - יש אוכלוסייה מקומית מסוימת שלהם ניתן לקרוא באותו שם אך לכל אחד זה בא לידי ביטוי בצורה אחרת.
  + אותו דבר קורה עם עצמים – יש הרבה עצמים שממשים ממשק מסוים – נקבל ריבוי צורות של הממשק הזה, כל אחד יכול לממש את השיטות של הממשק בצורה אחרת אבל לכל יש את השיטות ועם כולם ניתן לקרוא לאותן שיטות.
  + המשמעות של פולימורפיזם היא ששני אובייקטים מטיפוס אותו ממשק יכולים להיות ממחלקות שונות אבל לנו זה לא ישנה – כשזה נוח נתייחס אליהם מטיפוס הממשק ואפילו לא נצטרך להכיר את המחלקה הקונקרטית.
  + האפשרות להיות אדיש למחלקות הקונקרטיות היא אבן יסוד בתכנות מונחה עצמים.
  + לדוגמא – ממשק של צורה שמכיל את השיטות היקף ושטח – ריבוע יממש שטח והיקף לפי נוסחאות של ריבוע אך מחלקת עיגול תממש אותן כפונקציה של רדיוס.

קיבלנו כי הממשק צורה הוא פולימורפי.

* + - לקוח של הממשק צורה – למשל בניין שיכיל שדה מטיפוס צורה שמכיל את הבסיס שלו והוא יכול להכיל אובייקט מטיפוס ריבוע או עיגול.
    - בחישוב הנפח של הבניין נשתמש בשיטה שטח של צורה והוא יכול בכל פעם, תלוי איזה אובייקט קונקרטי נמצא במשתנה מטיפוס צורה, לחשב את השטח בצורה שונה בגלל המימוש השונה של השיטה במחלקות שונות.
  + אתחול של רפרנס יכול לשנות מקצה לקצה את מה שמבצע הקוד.
  + ניתן ליצור מערך מטיפוס הממשק, למשל shape והוא יכול להכיל אובייקטים של צורות שונות – ריבעו, משולש, עיגול.
  + פולימורפיזם הוא עוד ביטוי של אבסטרקציה – הקוד מרוחק מהמימוש הקונקרטי – אנו עושים לעיגול וריבוע אבסטרקציה לצורה גיאומטרית.
  + אבסטרקציה ופולימורפיזם הם הבסיס של כתיבת קוד כללי.
* **הגדרה חלופית לפולימורפיזם:**
  + הגדרנו פולימורפיזם בתור המצב בו יש מספר סוגים שונים של עצמים שמממשים את הממשק
  + לעתים מגדירים פולימורפיזם בצורה מעט שונה, בתור המצב בו ניתן להסתכל על עצם של המחלקה המממשת באמצעות רפרנסים מסוגים שונים (על פי הממשקים אותם הוא מממש), ולכן לאותו עצם יש למעשה מספר "צורות" שונות. לדוגמה: לעיגול יש גם את הצורה Circle וגם את הצורה Shape, ועל כן, על פי הגדרה זו, הוא מרובה צורות (פולימורפי).
* **פולימורפיזם בשפות סקריפטינג:**
  + בשפה statically-typed, בה לכל עצם יש טיפוס מוגדר היטב בזמן כתיבת הקוד (כמו ג'אווה או C++), שתי הצורות שקולות: קיום פולימורפיזם במובן הראשון לגבי הממשק, גורר את קיום המובן השני לגבי העצם המממש, ולהיפך.
  + שפות dynamically-typed, כמו Python, Matlab או JavaScript, אין בזמן ריצה הרבה משמעות "לטיפוס", או לתבנית, מהם נוצר העצם. מה שחשוב עבור קוד נתון זה האם לעצם יש או אין את כל האיברים שהקוד עושה בהם שימוש. כל עוד לעצם יש את כל האיברים הדרושים, לקוד לא אכפת האם העצם נוצר מתבנית כלשהי, ואם כן, מהי. אם לעצם חסר אחד האיברים, הקוד יעלה שגיאת זמן ריצה. גם בסביבה כזו פולימורפיזם בהגדרתו הראשונה מתקיים, כי יש מגוון צורות של עצמים שקוד נתון יעבוד עבורם (כל העצמים בעלי האיברים הדרושים). מנגד, ההגדרה השנייה מאבדת את משמעותה בשפות עם טיפוסים דינמיים.
    - כל האמור בהקשר זה על שפות עם טיפוסים דינמיים נכון גם לגבי C++ Templates. טמפלייט יכול להניח שלטיפוס T הנתון ישנה למשל שיטה מסוימת, מבלי להניח שום דבר נוסף על הטיפוס של T. לצורך דיון זה, המצב שקול לשפה dynamically typed על אף שהטמפלייט מאומת כבר בזמן הידור.
* **תכנות לממשק, לא למימוש:**
  + אחד העקרונות הכי מרכזיים בתכנות מונחה עצמים – תכנות לממשק ולא למימוש – Program to Interface, Not Implementation.
  + העיקרון אומר שתמיד נשאף לכתוב קוד עבור הטיפוס הגבוה ביותר ועם המטרה הכללית ביותר.
    - למשל בדוגמא של אנה – נרצה לכתוב שיטה שסוחבת איזשהו חפץ שניתן להרים במקום לכתוב שיטה שסוחבת כדורי שלג.
  + אם נקפיד על תכנות לממשק – לכוון גבוה ועבור מטרה עם מקסימום כלליות – נרוויח המון.
  + יתרונות תכנות ממשק:
    - מניעת כפל קוד
    - שירות ליותר לקוחות – מתאים גם ללקוחות שאני עדיין לא מכיר
    - נוכל להחליף את המימוש והלקוח שמתכנת לממשק לא יצטרך לשנות את הקוד שלו.
  + לפעמים זה פשוט להכליל, אבל לפעמים לא תמיד יהיה ברור שניתן להכליל.
* **מימוש שונה במחלקות שונות – הקשר בין תכנות לממשק לבין אנקפסולציה:**
  + מה אם חייבים להבחין בין עצמים של מחלקות שונות?
  + בתכנות מונחה עצמים אנו חושבים בכובעים – כשנכתוב מחלקה אחת נחשוב שמישהו אחר אחראי על האחרות.
  + בדוגמא של עולמה של אנה – אם בסחיבת ציפור נרצה שהיא תעשה משהו מסוים כשסוחבים אותה וכשסוחבים זבוב הוא יעשה משהו אחר – האם צריך שתי פונקציות שונות למרות ששניהם ממשמים את הממשק carriable?
    - בתור המתכנת של המחלקה ילד אנו לא צריכים לכתוב קוד שונה עבור כל carriable – המתכנת של זבוב אחראי על התנהגותו ובדומה עבור מתכנת של ציפור.
    - פה נכנסת האנקפסולציה – מה שקשור להתנהגות הציפור ייכנס במחלקה ציפור ולא במחלקה של ילד.
    - בדוגמא זו נוכל להוסיף לממשק את השיטה התנהגות או שבתוך השיטות הנתונות בממשק נממש את ההתנהגות הרצויה בכך מחלקה.
  + אם צריך לעשות משהו שונה עבור כל מחלקה קונקרטית, נחזור אחורה ונבדוק איפה שברנו אנקפסולציה.
  + אם שברנו תכנות לממשק סביר שגם שברנו אנקפסולציה.

**3.2 עיצוב תוכנה – Factory:**

* **מחרוזות:**
  + מחרוזות בג'אווה הן עצמים, לכן משתנה מסוג מחרוזת הוא למעשה מצביע על מחרוזת ולא המחרוזת עצמה.
  + ההשוואה x==y עבור x,y מטיפוס מחרוזת היא ההשוואה האם שני המצביעים מצביעים על אותו עצם בזיכרון.
  + כדי לבדוק אם שתי מחרוזות השמורות בעצמים שונים שקולות נשתמש בשיטה equals של המחלקה string.
* **תבניות עיצוב:**
  + צריך להפעיל שיקול דעת בשביל לתכנן מבנה של תוכנה כך ששינויים ברכיב אחד לא ישפיעו על רכיב אחר, שלא יהיה כפל קוד ושהקוד יהיה כללי.
  + ישנם פתרונות שאחרים כבר חשבו עליהם עבור בעיות נפוצות.
  + פתרונות עיצוב שמתאימים לתרחישים נפוצים נקראים תבניות עיצוב.
  + חשוב להכיר תבניות עיצוב כי:
    - בעיצוב תוכנה שלמה יש הרבה על מה לחשוב, ולכן זה נוח להיעזר בפתרונות מוכנים לפחות לחלק מהבעיה.
    - ללמוד פתרונות עוזר ללמוד בעיות.
    - ז'רגון של תכנות מונחה עצמים (ותכנות בכלל) – מאפשר לתקשר רעיונות.
    - ניתן להשתמש באותה תבנית כללית בפרוייקטים שונים
    - מכיוון שמדובר בתבניות נפוצות הן השתכללו עם הזמן ולכן כדאי להשתמש בהם.
  + תבנית העיצוב הראשונה שנדבר עליה היא מפעל – Factory
* **מוטיבציה למפעל:**
  + דיברנו על הממשק shape והמחלקות ריבוע ועיגול שממשות אותו.
  + דיברנו גם על המחלקה בניין שמשתמשת באובייקט מטיפוס צורה.
    - Building מתכנת לממשק ולא למימוש
    - אם נוסיף צורות נוספות שיממשו את הממשק צורה בניין ידע להשתמש בהן.
  + מישהו צריך ליצור את הבניין ולספק את הצורה.
  + נניח שאנו מתכנתי סימולטור תכנון עירוני עם מחלקה buildingPlacer שיוצרת בניינים חדשים –
    - המשתמש מכניס את שם הצורה כקלט.
    - אנו נרצה ליצור בניין ולכן נתרגם את המחרוזת שהקיש המשתמש לאיזה אובייקט נשלח כצורה של הבניין – ריבוע עיגול וכו'
  + מה שקרה זה שבאמצע המחלקה יש קוד שתלוי במימושים ולא רק בממשק – קוד שמצומד לשינויים בצורות, כשיוסיפו צורות נצטרך להוסיף אותן גם כאן – מה יקרה אם נשכח להוסיף, או לא נדע שמישהו הוסיף צורות.
  + זאת כנראה לא בעיה מקומית – אפילו במחלקה יכולה להיות בעיה דומה – מה הסיכוי שכשיוסיפו צורות יזכרו לעדכן את כל המקומות שבהם משתמשים בהם.
  + המחלקה buildingPlacer מתוכנת לממשק עם חלק שמתייחס לשם של צורה קונקרטית והוא תכנות למימוש.
  + בכך יעזור המפעל.
* **מפעל:**
  + יש לנו ממשק פולימורפי עם כמה מימושים – הממשק קבוע ומה שמשתנה זה רשימת המחלקות שמשתמשות בו.
  + נרצה להיות תלויים רק במה שקבוע ולא במה שמשתנה – המימושים.
  + הבעיה שיצירת עצם צריכה לנקוב בשם המחלקה הספציפית.
  + לכן ניקח את החלק של יצירת האובייקטים ונייצא אותו למחלקה ייעודית שתפקידה ליצור את האובייקט, כך היא מאפשרת לאחרים לתכנת לפי ממשק ולא לפי מימוש. מחלקה זו תהיה מפעל.
  + המפעל יהיה היחיד שיצטרך להשתנות.
  + במקרה שלנו נקרא למפעל ShapeFactory ונוסיף לה שיטה שיוצרת שורה.
  + כך נוכל במחלקה יצירת אובייקט מסוג המפעל ובעזרתו תיצור את הבניין.
  + המפעל הוא היחיד שיצטרך להשתנות בשינוי רשימת המממשים של ממשק הצורה.
  + ניצור את המפעל כחלק מהשירות של צורות – כי אם לא אנו מאלצים את הלקוחות לקרוא לבנאים – תכנות למימוש. כאשר ניצור את המפעל נוסיף אותו כחלק משירות הצורות.
  + מפעל הוא עוד צורה של הסתרת מידע רק עבור קבוצת טיפוסים.
  + יש כאן גם אבסטרקציה – המפעל לא משתף איך הוא יוצר עיגול.
  + כל עוד המפעל מחזיר את מה שהלקוח רצה זה לא משנה איך הוא ייצר אותו, ואפילו לא משנה מופע של איזו מחלקה הוא מחזיר.
  + סיכום:
    - תבנית עיצוב היא פתרון עיצובי נפוץ לתרחיש נפוץ
    - התרחיש הנפוץ כרגע הוא ממשק פולימורפי (עם כמה מחלקות מממשות) וכמה לקוחות שמשתמשים בו. הלקוחות מעדיפים לתכנת לממשק (לא רוצים לקרוא בנאי למחלקה קונקרטית).
    - תבנית העיצוב מפעל היא דרך טובה לפתור בעיה זו – הלקוח אומר למפעל מה הוא צריך והמפעל קורא לבנאי של המחלקה הנכונה.
    - הלקוח צריך להכיר רק את המחלקה ואת המפעל, ואם תשתנה רשימת הצורות זה לא ישנה ללקוח.
* ראינו דוגמה לכמה מחלקות שממלאות יחד שירות מאוחד עבור לקוחות חיצוניים. קבוצה כזו נקראת מודול, ובהמשך נדבר על מודולים עוד רבות.
* **עיקרון הבחירה היחידה:**
  + תבנית העיצוב מפעל מתכתבת עם העיקרון העיצובי הידוע – עיקרון הבחירה היחידה (The Single Choice Principle).
  + לפי עיקרון זה אם יש משהו בתכנית שלנו עבורו תתכנה מספר אפשרויות, אז הרשימה המלאה של האפשרויות תופיע רק במקום אחד.
  + דוגמא לכך היא המחלקה ShapeFactory – המפעל הוא המקום היחיד בו מופיעה הרשימה המלאה של הצורות האפשריות – זה המקום היחיד בו נעשית בחירה בין האפשרויות.
  + עיקרון הבחירה היחידה רלוונטי גם במקרים אחרים של רשימת אלטרנטיבות שצריכה להופיע בקוד ולא רק עבור מפעל – נרצה שהרשימה תופיע במקום יחיד.
  + אם זה נראה קשה אז שווה לחזור ולשקול אם היינו יכולים לעצב את הדברים אחרת כך שהבחירה תוכל להתבודד.

**3.3 ג'אווה סטרים – ChatterBot 2.0:**

* **JShell:**
  + בשביל להשתמש בכלי זה צריך להתקין JAVA 9 ומעלה, ושתיקיית Bin של JAVA תהיה מוכרת ע"י הpath של הטרמינל.
    - אם זה מתקיים נוכל לכתוב בטרמינל jshell
  + נקבל shell בתחביר של JAVA
  + אפשר להשלים אוטומטית מילים עם טאב (ייתן כמה אופציות אם יש כאלו)
    - בכל שלב ישלים לתחילית המקסימלית
  + /vars ייתן לי את המצב של המשתנים שהגדרתי עד כה.
  + אם נכתוב רק / הוא ישלים לי איזה פקודות קיימות.
  + אפשר להגדיר שיטות (שלא נמצאות בתוך מחלקה) –
    - נכתוב שורת הצהרה רגילה של פונקציה, נכתוב { ואז כשנעשה אנטר הוא יזהה שאנחנו עדיין בתוך השיטה. כשנסגור עם } הוא יבין שסיימנו ליצור את השיטה.
    - נשתמש בשיטה כרגיל.
  + אם אנחנו מחזירים ערך ולא שמים אותו בשום מקום הjshell מגדיר משתנה אוטומטי שיכיל אותו
    - שם המשתנה יתחיל עם $ ולאחר מכן מספר
    - בכל פעם שיקרה מצב כזה ייתן לנו משתנה אחר
    - גם אם רק נכתוב מספר בלי לשים אותו במשתנה ונלחץ אנטר הוא יכניס אותו למשתנה אוטומטי.
  + ה-jshell שימושי מאוד כדי להבין אילו פקודות הן שימושיות בJAVA ואיך לכתוב ולהשתמש בדברים.
* **המרה וחלוקת שלמים – casting, integer division:**
  + המרות טיפוסים פרימיטיביים:
    - אי אפשר להשים double ל-int – לא מתקמפל
    - אבל ניתן לעשות המרה – נכתוב לפני ערך הdouble בסוגריים את (int)
      * בהמרה כזו ייחתך החלק השברי)
    - אם נרצה לעגל (הערך השלם הקרוב ביותר) יש שיטה ייעודים –

(int) Math.round(d)

* + חלוקות:
    - חלוקה של שני intים עם / תחזיר רק את החלק השלם, כלומר int
    - באופן כללי תוצאת חלוקה של שני טיפוסים התוצאה תהיה מהטיפוס המורכב יותר
      * למשל int וdouble תחזיר double
    - אם נכתוב מספר ואז בצמוד D זה יגיד לJAVA שהוא מטיפוס double – למשל 42D
    - אפשר גם להמיר בשורת החלוקה ואז תהיה קדימות להמרה ורק אחרי תתבצע החלוקה
* **Command Line Arguments:**
  + נקרא למחלקה EchoChatter (ובהמשך יהיו לנו סוגים שונים של צאטרים)
    - יש לה קבוע עם תחילית שמצפה בבקשות
    - בנאי פומבי שמקבל מערך של תגובות לבקשה לא חוקית
    - יש לה שיטה פומבית שמקבלת משפט ומחזירה את התשובה של הבוט אליו
  + trim() – חותכת רווחים בתחילת ובסוף מחרוזת
  + כעת נגדיר ממשק ל-chatterים:
    - נקרא לממשק Chatter
    - בתוכו נשים את השיטה String replyTo(String statement)
  + נוסיף ל- EchoChatterשהוא ממש את הממשק שכתבנו, ובאתחול הצ'טרים נהפוך את המערך של הצ'טרים להיות מטיפוס הממשק וכך נוכל להוסיף גם צ'טרים אחרים.
  + נרצה שהמשתמש יגדיר בהרצת התוכנית איזה סוג צ'טרים הוא רוצה – באמצעות הcommand line
  + Command line arguments –
    - כל תוכנית שרצה בcommandline יודעת לקבל משתנים של commandline
    - ב-Main אנו מקבלים מערך של מחרוזות שאנו קוראים לו args והוא יכיל את הcommand line arguments שקיבלה התוכנית בהרצה הנוכחית.
    - אורך args אומר לי כמה משתנים קיבלנו
* **Factory & String Format:**
  + נרצה מפעל שיקבל את הארגומנט הנוכחי וידע ליצור צ'טרים מהסוג שהמשתמש רצה
  + ניצור מחלקה ChatterFactory ובתוכה שיטה build שמקבלת מחרוזת ומחזירה Chatter מהטיפוס שקיבלה.
    - אם לא נצליח לייצר מופע נחזיר null
  + הערה על כמה שורות מתחילה ב\*/ ומסתיימת ב/\*
  + נשים לב שהשוואת מחרוזות עם == משווה אם הרפרנסים מצביעים לאותו עצם וכדי להשוות את המחרוזות עצמן נשתמש בequals (מתודה של String)
  + הדפסת שגיאה – System.err.println() – מדפיס דרך שגיאה
  + שרשור מחרוזות – ניתן להשתמש בשיטה String.format(“..%\_...”, arg1, arg2,..) –
    - נציין placeholder שבמקומו ייכנס המשתנה שניתן אחרי המחרוזת – נכתוב %\_ כאשר \_ יהיה אות שמציינת מאיזה טיפוס המשתנה
      * למשל s עבור מחרוזת
      * עבור int נכתוב d
    - המתודה תחזיר את המחרוזת לאחר שהכניסה את ערכי המשתנים במקומות המתאימים
* **Switch:**
  + במרים שיש לנו הרבה אפשרויות כאשר עבור כל אפשרות אנו רוצים לעשות משהו אחר נשתמש בswitch
  + סינטקס:

Switch (<on what>) {

Case <if equal to\_\_\_>:

<do…>

Break;

Case <2>:

<do…>

Break;

Default:

<…>

Break;

* + המילים השמורות:
    - Switch – מקבלת בסוגריים את מה שצריך לבחור את האפשרויות שלו
    - Case – אחריה נציין את אחת האפשרויות
    - Break – יוצא מהswitch, כלומר לא צריך את המקרה הבא
      * אם לא נשים הוא ימשיך לעבור על כל המקרים
    - Default – אם אף אחת מהאפשרויות לא מתאימה, כמו ה-else
  + אי אפשר לשים אחרי case משתנה – חייבים לשים קבוע כלשהו.
    - אם הגדרנו קבוע זה גם יעבוד – private static final…
  + אפשר להגדיר אחרי case כמה ערכים שונים עבורם נרצה לעשות אותו דבר – case “”, “”:
    - רק עבור JAVA 14 ומעלה
* **Switch and scopes:**
  + הסקופ של כל משתנה הוא הסוגריים המסולסלים שהוא הוגדר בו וכל הסוגריים המסולסלים שבפנים.
  + זאת הסיבה שהיינו צריכים להגדיר את המשתנה chatter מחוץ לswitch כדי שהוא יוכר כאשר נרצה להחזירו.
  + נרצה להגדיר משתנה בסקופ הכי קטן שאנו יכולים.
  + המשתנה הראשון שהגדרנו עם שם מסוים "ינצח" – למשל אם יש לנו שדה בשם מסוים ובאחת השיטות הגדרנו משתנה מקומי עם אותו שם בהתייחסות למשתנה בשם כזה נתייחס לשדה.
* **Responses to illegal request:**
  + ניצור קבוע STRING\_SEPERATOR כך שנקבל קלט מהמשתנה שיכיל את רשימת התגובות לבקשה לא חוקים עם הseperator המתאים
  + שיטה של מחרוזת – splite(separator) תפריד את המחרוזת לפי המפריד שהחזרנו לה ותחזיר רשימה של תתי הרשימות שהופרדו.
* **humanChatter, continue, break:**
  + humanChatter הוא צ'אטר שמקבל את מה שהוא צריך לענות מהמשתמש.
    - הוא מממש את chatter ולכן מכיל את השיטה replyTo.
    - משתמש בscanner לקבלת קלט מהמשתמש.
  + כל מה שצריך לעשות כדי לתמוך גם בו צריך להוסיף למפעל את הcase של human ל-switch
  + נרצה להגדיר שאם אחד הצ'אטרים לא אומר שום דבר (המחרוזת ריקה) אז נדלג עליו – נעשה זאת בעזרת הפקודה continue
  + Continue מדלג על יתר הבלוק קוד שיש בתוך הלולאה ועובר ישר לבדיקת התנאי הבא
    - ב-FOR הוא כן מבצע את השלב – למשל i++
  + equalsIgnoreCase – משווה ומתעלמת מcase
  + נעצור את ריצת הלולאה אם הstatement היא stop.
  + אפשר לעשות זאת עם הפקודה break שמפסיקה את הלולאה או עם תנאי ללולאה
* **silentChatter:**
  + גם הוא מממש את הממשק chatter
  + הוא מכיל 2 שדות:
    - המשפט – statement
    - ההסתברות להגיד את המשפט הנתון
  + ההסתברות והמשפט יינתנו בזמן יצירתו
  + נגריל מספר בין 0 ל-100, אם המספר שיצא קטן מההסתברות אז נחזיר את שדה המשפט, אחרת נחזיר מחרוזת ריקה
  + נגריל בעזרת random עם המתודה nextInt(100)
  + בקבלת קלט – אם ניח את nextInt ניקח רק את המספר אבל לא את הירידת שורה, ובשביל זה נצטרך להשתמש ב-nextLine
* **Basic toddler:**
  + צ'אטר תינוק שאוהב משפטים מסוימים – יש לו ביטוי אהוב, אם המשפט מכיל את הביטוי שהוא אוהב אז הוא אומר תגיד את <המשפט> שוב, אחרת אומר את המשפט עצמו.
  + מכיל שדה של likedPhrase
* **Recursive toddler:**
  + נגדיר שאם צ'אטר התינוק ממש לא אוהב את המשפט ששמע אז הוא ב50 אחוז צוחק ו50 אחוז שותק.
  + ניעזר ב-silentChatter – נוסיף שדה פרטי של chatter מסוג silentChatter שאומר haha בסיכוי של 50 אחוז.
  + אז במקרה שהמשפט לא מכיל את הביטוי האהוב של הצ'אטר תינוק אז נחזיר את מה שsilentChatter אומר
  + ניתן להמשיך – במקום להגיב ספציפית כמו silentChatter נוכל לבחור שיגיב לפי כל צ'אטר אפשרי אחר - נעשה את השדה כ-chatter כללי ונקבל מהמשתמש את הצ'אטר שנרצה להשתמש אם המשפט לא נמצא.
  + נכתוב במפעל בזמן יצירת צ'אטר התינוק למשתמש להקיש את שם הצ'אטר שירצה לתת עבור משפט שלא אוהב – אבל כדי להבין איזה צ'אטר לבנות נשתמש בשיטה build – כלומר נקרא רקורסיבית לפונקציה בה אנו נמצאים שיוצרת צ'אטר לפי בקשת המשתמש.
* **Decorators:**
  + דקורטור – מקשט – זאת תבנית עיצוב (design pattern) שבה מחלקה מממשת ממשק כלשהו בעל מתודה:

Class Decorator implements SomeInterface {

@override

Public void method() {}

}

* + בנוסף היא מעבירה קריאות למתודה שלה למתודה דומה של אובייקט אחר שגם הוא ממש את ה- SomeInterface:

class Decorator implements SomeInterface {

private SomeInterface innerInstance;

public Decorator(SomeInterface innerInstance) {

this.innerInstance = innerinstance;

}

@Override

public void method() {

innerInstance.method();

}

}

* + כמו כן, אחרי / לפני הקריאה למתודה של המופע הפנימי היא מוסיפה פונקציונליות משלה, כלומר "מקשטת" את פונקציונליות האובייקט הפנימי:

class Decorator implements SomeInterface {

private SomeInterface innerInstance;

public Decorator(SomeInterface innerInstance) {

this.innerInstance = innerinstance;

}

@Override

public void method() {

... additional “decorator” code …

innerInstance.method();

... additional “decorator” code …

}

}

* + דקורטורים יכולים גם לקשט דקורטורים אחרים ואפילו את עצמם.
  + עם מימוש של מספר דקורטורים ניתן להרכיב קומבינציות שונות ולקבל מספר אקספוננציאלי של התנהגויות חדשות.