**OOP פרק 4 - ירושה**

**4.1 מנגנונים בתכנות מונחה עצמים – ירושה:**

* **מבוא לפרקים 4 ו-5:**
  + נגדיר מחלקת אב שממנה מחלקות שונות יורשות – מחלקות אלו יקבלו את התכונות שלה והתנהגות שלה ויוכלו להוסיף ספציפיקציה לדברים שלהם.
  + חבילות – חלוקת הקוד
* **העמסה – Overloading:**
  + מה עושים כשיש לנו צורך בשיטה מסוימת אבל יש אפשרויות שונות לפרמטרים שלה?
  + העמסה הוא מנגנון שמאפשר לכתוב כמה בנאים למחלקה אבל עם רשימת פרמטרים שונה – אפשרי כל עוד רשימת הפרמטרים שונה בכל בנאי.
    - אפשרי גם לבנאים וגם לשיטות אחרות
  + אם השיטות נבדלות רק בערך ההחזרה וכל השאר אותו דבר (שם ורשימת פרמטרים) תהיה בעיית קומפילציה – חייבים להיות נפרדים במשהו שהוא לא ערך ההחזרה
  + לדוגמא – השיטה println היא העמסה – ניתן לתת לה פרמטרים שונים.

הירארכיית ירושה:

* **הכלה וירושה:**
  + ירושה מאפשרת להגדיר מערכת יחסים בין מחלקות שונות.
  + מערכת יחסים בין מחלקות:
    - יחס Has-a – למחקה א יש רכיב מסוג מחלקה ב (נקרא גם יחס הכלה, composition).
      * למחלקה א' יש אובייקט מהמחלקה ב'
      * ממומש בשימוש בdata-members
    - יחס Is-a – מחלקה א' היא סוג של מחלקה ב'
      * מחלקה א' היא מחלקה כללית ונגדיר מחלקה ב' שהיא סוג ספציפי של מחלקה א'
      * למחלקה ב' יש את כל הנתונים (תכונות ומתודות) של מחלקה א' אבל יש לו גם תכונות ומתודות משל עצמו
      * בא לידי ביטוי בירושה
    - יחס instance-of – ממומש ע"י יצירת מופע מחלקה
      * לא להתבלבל עם יחס is-a
      * מדובר ביצירת אובייקט ולא בירושה
  + נגיד שמחלקה A יורשת את מחלקה B אם A היא מסוג B
    - B היא הsub-class של A ו-A היא ה-super-class של A
    - ל-A יש את כל התכונות והמתודות של B אבל היא יכולה להוסיף גם תכונות ומתודות משל עצמה
    - ירושה מסומנת מהמילה השמורה extands
      * ברגע שהשתמשנו בה לא נצטרך להגדיר מחדש את השדות והמתודות שיש ב-A אלא פשוט נוכל להשתמש בהן ב-B
    - כלומר עבור B ניתן לקרוא לשדות ומתודות ממחלקה A וגם ממחלקה B
      * ניתן להסתכל על האובייקט מסוג B גם כאובייקט מסוג A
* **הירארכיית ירושה:**
  + תכונות של ירושה:
    - ירושה היא רקורסיבית – מחלקה A יורשת מB שיורשת מC וכן הלאה – אין הגבלה למספר הירושות
    - ירושה היא טרנזיטיבית – אם A יורשת מB וB יורשת מC אז A יורשת למעשה גם מC (בצורה עקיפה)
    - כל מחלקה יכולה להיות מחלקת אב של מספר בלתי מוגבל של מחלקות – יכולות לרשת ממנה כמה מחלקות

אולם כל מחלקה יכולה להיות מחלקת בן של מחלקה אחת בלבד – יכולה לרשת רק ממחלקה אחת

* + כל מחלקה בJAVA יורשת מהמחלקה java.lang.Object – מחלקה מיוחדת
    - ניתן לכתוב extands Object או אם נגדיר מחלקה בלי לכתוב זאת היא אוטומטית תירש ממנה, או ירושה בעקיפין (מחלקה שיורשת ממחלקה אחרת והיא כבר יורשת מObject)
    - המחלקה היחידה שלא יורשת ממנה היא המחלקה עצמה – המחלקה היחידה שאין לה מחלקת אב
* **איברים פרטיים של מחלקת האב:**
  + מצד אחד מחלקה שיורשת ממחלה אחרת מקבלת את כל הרכיבים שלה (כולל פרטיים) אבל מצד שני אמרנו שאף אחד אחר לא יכול לגשת לרכיבים פרטיים של מחלקה.
  + מחלקה יורשת לא יכולה לגשת לשדות הפרטיים של מחלקת האב (אם ננסה נקבל שגיאת קומפילציה), מצד שני השדות הציבוריים כן נגישים למחלקת הבן.
  + אבל השדות הפרטיים של מחלקת האב כן קיימים עבור מחלקת הבן פשוט אין לו גישה אליהם – כלומר ניתן לגשת למתודות פומביות של מחלקת האב שניגשות לשדות הפרטיים וזה יעבוד פשוט לא נוכל לגשת ישירות לשדות הפרטיים של מחלקת האב

נראות ודריסה:

* **Protected Visibility:**
  + Protected modifier – שדות ומתודות יכולות לקבל את הprotected modifier
    - זוהי למעשה אלטרנטיבה לפרטי ופומבי
  + ה protected modifier יכול לאפשר גישה למחלקות יורשות למשתנים שאותם הם יורשים, אבל להן בלבד (מחלקות לא יורשות לא יוכלו לגשת)
    - למשל משתנים שנגדיר כפרטיים ולא תהיה גישה לאף אחד גם לא למחלקה יורשת, נוכל לשנות לprotected וכך יהיה ניתן לגשת אליו רק ממחלקות יורשות.
  + נשתדל להימנע מprotected –
    - ברגע שנגדיר משהו כך הוא יהפוך להיות חלק מהAPI (בניגוד לפרטיים) - זה כן משהו שאנו חושפים לעולם, כל מחלקה חיצונית תוכל להסתמך על קיומם של השדות שנגדיר וכך תיווצר תלות.
  + נעדיף תמיד להגיד משהו כפרטי ולא כמוגן.
* **דריסה:**
  + Overriding – הרעיון של לקחת מחלקה, לרשת ממנה (להרחיב אותה) אבל לשנות את ההתנהגות שלה
    - כלומר לרשת מתודה מוגנת או פומבית ולשנות אותה.
  + למעשה צריך לממש מחדש את אותה המתודה תוך שימוש באותה חתימה כמו חתימת המתודה המחלקת האב.
    - כך כאשר ניקרא למתודה זו על אובייקט ממחלקת הבן תיקרא המתודה של מחלקת הבן ולא המתודה של מחלקת האב
      * כשנקרא למתודה על אובייקט של מחלקת האב היא תיקרא למתודה המקורית ממחלקת האב
  + קונספט super – מילה שמורה שמאפשרת לנו לגשת ממתודה שאנו דורסים אותה למתודה במחלקת האב
    - במקרה שנרצה לדרוס מתודה אבל כן להשתמש במתודה של מחלקת האב
    - נכתוב super.methodName() וזה יקרא למתודה ממחלקת האב
    - בקונסטרקטור ניקרא super() וזה יקרא לקונסטרקטור של מחלקת האב
  + נשים לב כי הקונסטרקטור של מחלקת בן צריך לקבל גם את הפרמטרים שדורש הקונסטרקטור של מחלקת האב.

בנאים:

* **מילת המפתח super ובנאים:**
  + חשוב לזכור ש-super() של הקונסטרקטור תמיד נקרא – בצורה ישירה או בצורה עקיפה.
    - תמיד נצטרך לקרוא לsuper כדי ליצור את מחלקת הבן כי אנו צריכים ליצור את השדות והשיטות שהוא מקבל ממחלקת האב
  + אם לא נקרא מפורשות לקונסטרקטור של מחלקת האב הוא ייקרא בעקיפין (מאוחרי הקלעים) לsuper() בלי פרמטרים – אם אין למחלקת האב קונסטרקטור ללא פרמטרים נקבל שגיאה
  + קודם ייקרא הקונסטרקטור של מחלקת האב ואז יתבצע הקונסטרקטור של מחלקת הבן.
  + אם אין לנו קונסטרקטור ללא פרמטרים במחלקת האב נהיה חייבים להוסיף קריאה לsuper עם הפרמטרים המתאימים

ירושה ופולימורפיזם:

* **ירושה ופולימורפיזם:**
  + Upcasting ופולימורפיזם עובדים בירושה בדיוק באותה צורה שהם עובדים עם ממשקים
  + נגדיר מחלקה A ומחלקות B ו-C שיורשות ממנה.
  + נוכל ליצור רפרנס לאובייקט מסוג A אך ליצור אובייקט של B או C – נקבל פולימורפיזם כלומר אובייקט של B או C ישמר כאובייקט מסוג A
    - נוכל לקרוא לפקודה שמוגדרת במחלקה A, אולם הקוד שירוץ הוא הקוד שנמצא במחלקה B/C (התוכן של מחלקת הבן שהאובייקט הוא למעשה שלה)
    - אולם לא נוכל לקרוא למתודות שנמצאות רק במחלקת הבן B/C כי סוג הרפרנס הוא A וב-A אין את המתודה הזו.
    - כלומר התוכן עצמו יהיה של מחלקת הבן (האובייקט האמיתי עליו נצביע) אולם נוכל לקרוא רק למתודות של A כי הרפרנס הוא מסוג A
      * הקוד שרץ הוא הקוד של סוג האובייקט הקונקרטי
      * מותר לנו להריץ רק מתודות ותכונות שיש למחלקה שהרפרנס הוא מהסוג שלה.
  + כלומר - אחרי המרה כלפי מעלה (לטיפוס של ממשק או של מחלקת אב), סוג המצביע קובע איזה שיטות אפשר להריץ, ואילו סוג העצם קובע איך הן ממומשות.

ירושה לעומת שימוש בממשק:

* **דוגמא ליתרונות של ירושה:**
  + נתבונן בדוגמא של קבצי שמע – נרצה ליצור קבצי שמע מסוג שיר ומסוג פודקאסט.
  + פתרון באמצעות ירושה – נעשה מחלקת אב של סטרים ובו נממש את המחלקה play, ניצור שתי מחלקות שיורשות ממנו ולכל אחת מהן תכונות נוספות, כאשר הן ירשו את המתודה Play
  + פתרון באמצעות ממשק – נגדיר ממשק של קובץ שניתן לנגן אותו ונגדיר שתי מחלקות שיממשו אותו. במקרה זה המימוש של Play יוגדר בנפרד בשתי המחלקות. כדי לפתור את כפל הקוד ניצור מחלקה נוספת שמממשת את הממשק והיא תממש את המתודה play. נוסיף למחלקות שיר ופודקאסט שדה של אובייקט מהחלקה הזו וכך הן יוכלו לקרוא לשיטה זו (ממשק + הכלה)
  + ירושה היא פתרון פשוט יותר וברור יותר מהפתרון של ממשק והכלה.
  + ירושה שימשה אותנו לשני צרכים –
    - מחזור קוד
    - פולימורפיזם – הנחנו שיכול להיות שלקוח ירצה להשתמש בשיר או פודקאסט מבלי לדעת באיזה סטרים ספציפי מדובר
  + אם היינו רוצים רק מחזור קוד יכולנו להשתמש בהכלה
  + כלומר ירושה היא כלי שמתאים כאשר נרצה גם מחזור קוד וגם פולימורפיזם.
  + במקרה שלנו הפצרון הראשון קריא יותר ומכיל פחות קוד

Casting:

* **Casting:**
  + המרה היא הפעולה שבלב הפולימורפיזם – התייחסות לאובייקט מסוג אחד כאובייקט מסוג אחר
    - כלומר לרפרנסים לאובייקטים
    - זה נקרא up-casting (אנחנו למעשה עולים בהיררכיה
    - הגדרה פורמאלית – סוג הרפרנס הוא או super-class או interface של האובייקט הקונקרטי
  + יש שני סוגים עיקריים של upcasting –
    - בלתי מפורש – הקומפיילר מחליט מה הסוג של הup-casting
      * לדוגמא A obj = new B() הקומפיילר ימיר את האובייקט מסוג B לאובייקט מסוג A
      * כלומר הקומפיילר יחליט מה סוג האובייקט
    - מפורש – נגיד לקומפיילר את הטיפוס הספציפי להמרה – A obj = (A) new B()
      * לרוב אין סיבה להשתמש בהמרה מפורשת כי הקומפיילר בדרך כלל יודע מה לעשות
  + Down-casting – סוג נוסף של המרה, קצת יותר מסובך. זוהי המרה בכיוון ההפוך.
    - חייב להיות מפורש – למשל A a=… ונרצה להפוך אותו ל-B b= (B) a אבל לא נוכל לכתוב את השורה B b = a
    - יכול להצליח – אם האובייקט הקונקרטי הוא באמת מטיפוס B – A a = new B() אז נוכל לעשות B b = (B) a

אבל אם האובייקט הקונקרטי הוא שונה נקבל שגיאה בזמן ריצה – A a = new C() ולא נוכל לכתוב B b = (B) a כי אין התאמה בין האובייקטים הקונקרטיים.

* + - מקרה נוסף שיכשל הוא המרה של אובייקט ממחלקה שונה לגמרי לסוג של מחלקה אחרת - הקומפיילר יזהה ונקבל שגיאת קומפילציה
    - המרה למטה יכולה להצליח רק אם עשינו לפני המרה למעלה – A a = new B() (המרה למעלה) ואז B b = (B) a (המרה למטה)
      * לא תמיד יעבוד אם אין התאמה בין האיברים הקונקרטיים מהם עברנו למעלה ואז למטה.
* **הבעייתיות בהמרה כלפי מטה:**
  + נעדיף לא להשתמש בהמרה מטה
  + יכול לגרום לבעיות – אם הכישלון יהיה הוא יהיה בזמן ריצה והרבה יותר קשה להתמודד איתו
  + כמעט תמיד יש דרך טובה יותר לעבוד מאשר להשתמש בהמרה מטה
  + סיבה נוספת שאנו לא נרצה להשתמש בכך היא גמישות – כשאנו משתמשים בהמרה מטה אנו עוברים ממשהו כללי למשהו ספציפי
    - הפגיעה בגמישות היא שכאשר נשתמש בהמרה מטה אנו למעשה מטפלים בפחות מקרים – פחות כללי
* **Instanceof:**
  + אופרטור של JAVA שמאפשר לנו לבדוק בזמן ריצה אם אובייקט הוא מופע של מחלקה מסוימת.
    - נקבל אמת או שקר על הביטוי A instanceof B
    - אם B יורשת מA או מממשת ממשק C נקבל עבור b instanceof A אמת ועבור b instanceof C גם כן אמת.

לעומת זאת נקבל שקר במקרים ההפוכים – a/c instanceof B

* + למעשה יכול לשמש אותנו לפני שאנו מבצעים המרה מטה.
  + זהו פתרון לא טוב –
    - עדיין ניצור קוד שמועד לבאגים
    - עדיין לא גמיש (לא מאפשר לנו לתמוך באפשרויות נוספות אלא במקרים ספציפיים)
  + האלטרנטיבה היא להשתמש בAPI כללי וכך לא נצטרך להשתמש בinstanceof – יותר פשוט וניתן להרחבה
* **סיכום המרות:**
  + המרה למעלה לא מפורשת - אם יש לנו A a ו-B b כאשר B יורשת מA נוכל לעשות a=b
    - אבל אי אפשר לעשות המרה למטה לא מפורשת b=a
  + אם נבצע a.method() אז יתבצע הקוד של המתודה שכתוב במחלקה B
* **equals ו-toString:**
  + כל מחלקה חדשה שאנו כותבים יורשת מהמחלקה Object של JAVA, ולכן אם יש לObject שיטה פומבית אז היא קיימת גם אצל כל מחלקה אחרת בJAVA.
  + שיטות שימושיות של Object:
    - toString – מחזירה ייצוג מחרוזתי של העצם
      * למעשה בהדפסה אנו קוראים לtoString - אם לא דרסנו אותה המימוש יהיה המימוש בObject
      * אם נרצה דרך מסוימת להפוך את האובייקט למחרוזת נדרוס את המתודה toString – נרצה לעשות זאת בכל מחלקה שבה נצטרך ייצוג מחרוזתי של אובייקט
    - equals – משווה את התוכן של שני עצמים ומחזירה אם הם שקולים מבחינתנו (לא נשתמש ב== כי הוא ישווה למעשה כתובות)
      * כל מחלקה יורשת שיטה זו
      * היא בודקת שקילות אבל לא תמיד נתייחס לשקילות בין אובייקטים באותה צורה
      * לכן נרצה לדרוס אותו – כאשר נדרוס תמיד צריך לבדוק שהאובייקט הנוסף הוא instanceof של המחלקה שלנו!
      * אם יש שדות שלא נבדוק בזמן דריסת המתודה אז הם למעשה לא חלק מהזהות של האובייקט ולכן אובייקטים יכולים להיות שקולים גם אם השדות האלו שונים
  + נשים לב שהשתמשנו ב-instanceof ב-equals, ושלמעשה כך ייראה כמעט כל מימוש של השיטה! instanceof הוא לא מקור כל הרוע, במיוחד כשהוא רק בודק אם עצם הוא מהסוג של המחלקה הנוכחית. לעומת זאת השימוש בו הוא נורת אזהרה אדומה כשמשתמשים בו כדי לבדוק את הטיפוס הקונקרטי מבין כמה אפשרויות במקום להשתמש בפולימורפיזם (למשל: אם העצם הוא ציפור תעשה כך, אם העצם הוא כלב תעשה אחרת וכן הלאה).
  + ניזכר כי יש שני סוגים של טיפוסים בJAVA – פרימיטיביים ואובייקטים.
    - פרימיטיביים הם לא אובייקטים, כלומר לא יורשים מObject ולכן אין להם את השיטה equals
    - תמיד נוכל להשוות פרימיטיביים עם ==

ירושת ממשקים:

* **ירושת ממשקים:**
  + האפשרות להרחיב ממשקים – interface A extends interface
  + שימושי כאשר נרצה להגדיר טיפוסים שונים של "חוזים" או התנהגויות שיחלקו אותן מתודות
    - כלומר הרחבת דרישות והאפשרות שמחלקות יממשו ממשקים עם דרישות יותר ספציפיות (למשל עם תוספות)
  + מחלקות שבוחרות לממש sub-interface חייבות לממש את המתודות של הsub-interface וגם את המתודות של ממשק האב.

**4.2 מיני עיצוב תוכנה – מבוא למודולים:**

חבילות:

* **חבילות:**
  + דרך לחלק את הקוד לחלוקות הגיוניות – לקחת מחלקות ממשקים ודברים דומים אחד לשני ולאחד אותם יחד
    - דומה לחלוקה לתיקיות על המחשב – מאפשר לשמור על סדר בקוד.
  + יתרון נוסף הוא חלוקת הקוד למודולים שונים
    - מאפשר לייצא רק חלק מהקוד
    - לאפשר לאנשים להשתמש רק בחלק מהקוד
  + שימוש בחבילות מאפשר להוסיף הרשאות לקוד שלנו – להגיד שלמשתנים שונים מותר לגשת בתוך החבילה אבל לא מחוץ לחבילה.
  + כשר אנו לא מציינים לפני איבר אף הרשאת נראות האיבר מקבל הרשאת ברירת מחדל – הרשאת חבילה (package) כלומר ניתן לגשת אליו רק מהחבילה הנוכחית (מכל הרכיבים שיש בחבילה) אבל לא מחוץ לה
  + דוגמא: the collection framework היא למעשה חבילה ב-java.util – הרכיבים שלה (ממשקים, מימושים, אלגוריתמים וכו') למעשה קשורים לוגית לאותה חבילה – כולם חולקים תכונה סמנטית – קשורים למבנה נתונים
    - הרכיבים שייכים לאותה חבילה וכולם חולקים את התכונה שהם עוסקים שבנושא של מבנה נתונים ולכן הם באותה חבילה.
    - כדי לשים מחלקה בחבילה מסוימת צריך לרשום בראש קובץ הJAV את השורה package <packName>;
    - ניתן לייבא חבילה – אם מחלקה לא נמצא בחבילה שלנו ונרצה להשתמש בו נצטרך לייבא אותו – import <packName>.A
      * אבל לא נוכל לגשת לשדות או מתודות ב-A אם הם לא מסומנים כפומביים
      * אם מחלקות נמצאות באותה חבילה לא נצטרך לייבא את המחלקה אם נרצה להשתמש בה בקובץ מחלקה אחרת כי הן שייכות לאותה חבילה
  + הרשאת package –
    - ניתן גם להגדיר מחלקות ללא שום modifier ואז תהיה להן הרשאת חבילה – מחלקות אלה ניתנות לגישה רק ממחלקות אחרות באותה החבילה
      * מחלקות מחבילות אחרות לא יוכלו לגשת אליהן
    - המחלקות הפומביות של חבילה מגדירות את הAPI של החבילה
* **סיכום:**
  + עד כה למדנו על 4 מגדירי נראות של איברים בתוך מחלקות –

איבר (שיטה או שדה) יכול להיות private, public, protected, ועכשיו למדנו הגדרת נראות רביעית: נראות חבילה (ברירת המחדל בג'אווה). איברים בנראות חבילה נגישים, ובכן, מכל מחלקה שבתוך החבילה.

* + לטיפוסים עצמם (למחלקות ולממשקים), יש רק שני מגדירי נראות אפשריים. טיפוס יכול להיות מוגדר בלי מגדיר נראות, למשל class Foo, ואז הוא בנראות חבילה וזמין רק בתוכה. אחרת, הוא יכול להיות פומבי, למשל public class Foo, ואז הוא זמין גם מחבילות אחרות.
    - טיפוס לא יכול להיות private או protected.
  + כמובן, הנראות של איברים במחלקה חסומים על ידי נראות המחלקה עצמה. למשל, אם המחלקה כולה בנראות חבילה, בוודאי שלא ניתן לראות אותה או אף איבר שלה (לרבות הפומביים) מחוץ לחבילה.
* **מודולים:**
  + Module – רכיב תוכנה:
    - מודול הוא אוסף של מחלקות וממשקים
    - למודול יש API – הAPI שלו הוא חלק מהמחלקות והממשקים שלו (החלק הפרטי הוא רק לשימוש פנימי של המודול)
      * הלקוח יודע רק מה המודול יודע לעשות ולא איך
  + בתוכניות גדולות יותר אנחנו קודם עושים אנקפסולציה למודולים ואז אנקספולציה למחלקות וממשקים

**4.3 ג'אווה סטרים – IDE:**

פרויקט חדש, ייבוא קבצים חלוקה לחבילות:

* **הקדמה ודוגמא עבור פרויקט איקס-עיגול:**
  + IDE היא למעשה סביבת עבודה
  + תיקיית src צריכה להכיל את קבצי הקוד שלנו
  + ניתן ליצור חבילות בתוך הsrc ואז הקבצים שקשורים לחבילה מסוימת יהיו בתיקייה שלה (יתווסף אוטומטית השורה הרצויה בתחילת הקובץ של השייכות לחבילה)
  + בחבילה main ניצור את השיטה main
  + בפעם הראשונה שנריץ נצטרך לבחור איזה main נרצה שהוא יריץ
  + קיצורי מקשים:
    - shift+F6 לשינוי שם
    - alt+enter להצעות תיקונים מצד סביבת העבודה
    - F2 לדילוג בין שגיאות הידור
    - ctrl+tab למעבר בין קבצים
    - ctrl+space לבקשה להשלמה אוטומטית
  + בעבודה עם חבילות מומלץ שכל טיפוס יהיה חלק מחבילה בעלת שם

הזנת cmd args והאנוטציה Override@:

* **ארגומנטים, override ועוד קיצורי מקשים:**
  + אם נרצה ארגומנטים של commandline נעשה edit configurations ונכתוב את הארגומנטים ב-parameters
  + @override – מציין שאנו דורסים שיטה אחרת, כך אם נעשה טעות בשורת ההצהרה של המתודה נקבל שגיאה
  + קיצורי המקשים שלמדנו:
    - sout להדפסה, ctrl+/ להערה או הוצאה מהערה
    - alt+insert להכנסת איברים
      * למשל לדריסת מתודות או למתודות ממשק שצריך לממש
    - ctrl+o למימוש שיטת ממשק או דריסה
    - ctrl+x/v ל-cut/paste של שורה
    - alt+shift+up/down להזזת שורות

GUI:

* **פתיחת חלון וציור עיגול:**
  + המחלקה הבסיסית שבJAVA שמייצגת חלון היא JFrame
    - נניח כי יצרנו אובייקט של המחלקה בשם frame
    - חלון הוא אוטומטית invisible ולכן כדי לראות את החלון נכתוב frame.setVisible(true)
    - בשימוש זה הריצה לא נסגרת אוטומטית ולכן נרצה לכתוב frame.setDefaultCloseOperation() – אם נלחץ ctrl+space נקבל פרמטרים שונים לאפשרויות.
      * אם נבחר EXIT\_ON\_CLOSE יצא בסגירת החלון (יסיים ריצה)
  + החלון נועד להיות קונטיינר שנועד להכיל דברים אחרים. אם נרצה שיכיל דבר בסיסי שיכיל רכיבים אחרים נוכל ליצור לו JPanel
    - נוסיף אותו לחלון עם frame.add(panel)
    - נוכל לבחור גודל של הפאנל נשתמש ב-panel.setPreferredSize(new Dimension(x,y)) – נגדיר את גודל החלון בעזרת אובייקט dimention
    - נגדיר לחלון שיתאים את הגודל שלו לגודל הרכיבים עם frame.pack()
  + אם נרצה לצייר על החלון – הפאנל מצייר באמצעות שיטה שלמעשה לא מציירת כלום. לכן נרצה להגדיר תת מחלקה של פאנל שתדרוס את המתודה ותצייר משהו
    - לשיטה קוראים paintComponent() שמקבל Graphics
    - נקרא לsuper למקרה שהוא עושה משהו חשוב
    - g.fillOval – מקבלת קואורדינטות ומציירת עיגול על המסך
    - הגדרת צבע העיגול עם g.setColor(Color.<>)
  + קיצור לבקשת תיעוד הוא ctrl+q
  + נשים לב כי ניתן להריץ את הקבצים מהcmd גם אם הם בIDE