**מצגת 9 – הורשה - Ingeritance**

הורשה – בJAVA ניתן לרשת אך ורק ממחלקה אחת ישירות

מחלקה מורישה = SuperClass (המחלקה הגבוהה ביותר בJAVA היא Object)

מחלקה יורשת (חדשה) = SubClass :

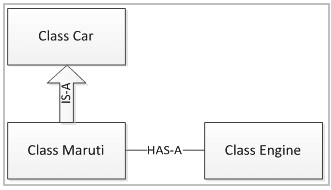
יכולה גם להיות SuperClass בעתיד.

כמעט בהכרח יותר ספציפית ומדוייקת מהSUPER שלה.

יתרונות : חוסך זמן, משפר את יכולות התחזוקה, השינוי והdebug. מפחית שכפולים קוד.

**האם כל מחלקה חדשה יורשת? כן! ממחקת Object גם אם לא הוצהר ככה (by defult)**

לא ניתן לרשת constructors! אפשר להשתמש אך ורק בעזרת super. אם לא השתמשת בsuper אז קיימת קריאת ברירת מחדל של constructor ריק (ללא פרמטרים), אם לא קיים כזה אז קוראים לבנאי של object (ללא פרמטרים) **לזכור: אם משתמשים בsuper זו חייבת להיות הפקודה הראשונה בconstructor.**

Is a & has a

עובד is a אדם

יונדאי is a רכב

אדם has a טלפון

רכב has a מנוע

Override – להגדיר מחדש פונקציה אשר ירשת אותה ממחלקת האב. חייבת להיות בעלת אותה חתימה.

כדאי לדעת: אם ביצעת override אתה עדיין יכול לגשת לפונקציה המקורית בעזרת קריאה דרך פונקציית האב. (superclass.function)

דרגות חופש של משתנים

Public (הכי חלש) – ניתן לגשת מכל מקום

Protected – ניתן לגשת אך ורק מהמחלקה עצמה, ממחלקות יורשות וממחלקות באותה חבילה.

Private – אך ורק מתוך המחלקה עצמה.

כדאי לדעת: אם מחלקה ירשה משתנה protected או public הוא חייב להישאר כזה.

בעיות מרכזיות עם protected

Subclass יכול לשנות משתנים מבלי להשתמש בset.

לפעמים נדרש שינוי בכל מחלקות הsub אם נעשה שינוי במחלקת האב. (לדוגמא שינוי משתנה לprivate פתאום)

המשתנים זמינים לכל המחלקות בתוך הpackage. לא בהכרח אנחנו רוצים את זה.

**כדאי לדעת:** הדרך הנכונה לתכנות היא ליצור משתני private ופונקציות public.

**שגיאות קומפילציה בהורשה:**

1. ניסיון גישה למשתנה private מחוץ למחלקה בלי get.
2. שימוש בsuper לא כשורה ראשונה בבנאי.

**מצגת 10 – פולימורפיזם – Polymorphism**

הרעיון המרכזי מאחורי פולימורפיזם הוא קריאה לפונקציה דומה על מחלקות שונות והפונקציה תדע לאן לגשת ואיך לפעול בהתאם לסוג המשתנה שקרא לה.

Superclass לא יכול להיות מיוחס כsubclass בגלל שהוא לא מהסוג שלו, אבל ההפך אפשרי.

יחס is-a מיוחס רק כלפי במעלה ההיררכיה ולא כפלי מטה.

**איך פונקציה יודעת למי להתייחס?** הפונקציה מחפשת מי קרא לה, היא משתמשת בפונקציה getClass (קיים במחלקת object) וכך היא יודעת למי לגשת.

Interface – על מנת לממש צריכים לבצע implements

* ניתן לממש כמה interface’s שרוצים[מפרידים בעזרת ","] (בשונה מהורשה בה אפשר רק מחלקה אחת).
* Interface מכריז מה הפונקציות הנדרשות למימוש אך לא חייב להכריז איך לממש אותם.
* מכיל **רק** מתודות אבסטרקטיות.
* מחלקה אשר מכילה את הinterface **חייבת** לממש בתוכה את כל הפונקציות שהיא מקבלת עם אותה חתימה בדיוק. (יש מקרה אחד בו לא חייבים להכריז על כל הפונקציות וזה אם המחלקה היא אבסטרקטית).
* כל מחלקה שמשתמשת בinterface **חייבת** להיות public.
* אם מחלקת אב implements an interface אז כל מחלקות הבן יש באופן אוטומטי is a relationship לinterface. ובכך מחלקות הבן מחויבות לממש את הפונקציונליות של הinterface.
* Interface בדרך כלל קיים בשביל להקצות פונקציות משותפות, גם למחלקות לא מקושרות (אין יחס הורשה כלשהו).
* **חשוב לדעת** – מחלקות לא קשורות אחת לשנייה (אין יחס ירושה בניהם) מקיימות פולימורפיזם בעזרת מימוש אותו interface.

Abstract Classes – (לא חייבת להיות מחלקה כזאת)

* מחלקה אשר לא ניתן ליצור אובייקט מסוג זה.
* לרוב ממומש רק בתור superclass.
* מחלקות אלה אינן "שלמות" ויותר מידי כלליות. הן מכילות הרבה מידע אשר צריך משותף לכל מחלקות הבן.
* מחלקות הבן שלהן חייבות להשלים את החלקים החסרים.
* לרוב מחלקות כאלה יכילו בתוכן לפחות פונקציה אחת אבסטרקטית.
* אם קיימת פונקציה אבסטרקטית אז גם המחלקה **חייבת** להיות אבסטרקטית.
* Constructors and static **לא יכולים** להיות אבסטרקטים!

Final methods

* מתודות במחלקת האב לא יכולות לעבור override במחלקות הבן.
* Private & static מרמזים על final. לא ניתן לשנות אותם במחלקות הבן גם כן.
* בגלל שלא ניתן לשנות final כל מחלקות הבן משתמשות בפונקציה המדויקת של מחלקת האב. מצמצם זמני ריצה.

Final Class

* לא ניתן לרשת ממחלקה כזו.
* כל המתודות בה הן Final בתור ברירת מחדל. (לדוגמא : String Class is Final)
* יצירת מחלקת Final מונעת ממשתמשים ליצור מחלקות בן ולעקוף את החוקים והפונקציות.(אבטחה גבוהה)

**בעיות קומפילציה**

1. הקומפיילר של JAVA לא מאפשר להפוך מחלקת אב למחלקת בן.(קיימת טכניקה המאפשרת את זה downcastin אך לא למדנו ולא מומלץ להשתמש בה).
2. ניסיון לייצר אובייקט מסוג אבסטרקטי
3. ניסיון לממש פונקציה אבסטרקטית של מחלקת האב, אלא אם מחלקת הבן גם היא אבסטרקטית.
4. ניסיון לרשת ממחלקה שהיא Final (String לדוגמא).
5. אי מימוש **כל** הפונקציות מהinterface במחלקה שהיא לא אבסטרקטית.

**מצגת 11 – Execptions**

חריגה – בעיה הנוצרת בזמן הרצת הפונקציה/תוכנה. הרעיון המרכזי הוא טיפול בחריגה מבלי למנוע מהתוכנה להפסיק לרוץ (אלא אם יש צורך בכך).

Throwable היא מחלקת האב של כל החריגות (exception) והשגיאות (error) בjava.(subclass of object)

**רק** מחלקות בת של Throwable יכולות לזרוק חריגות! למשל מחלקת exeption.

**כדאי לזכור** – החריגה תכנס לcatch הראשון שמתאים לה, לכן נתחיל מחריגות ספציפיות יותר ורק בסוף נשים חריגה כללית.

Stack trace – מציג מידע אודות השגיאה ומהיכן הגיעה. ניתן למצוא בפנים את שם השגיאה, מהיכן הגיעה, שרשור הקריאה שהוביל לשגיאה.

אם ברצונך לתפוס את החריגה ולטפל בה (כמובן **חייב** להכריז על Throwable) צריך להכניס את הפעולות לתוך try{} catch(exception){}

Try

* קטע קוד אשר **עלול** לזרוק חריגה. קטע קוד אשר צריך להיפסק אם חריגה קרתה בו.
* משתנים בתוך הtry הם משתנים מקומיים בלבד, הtry הופסק כך גם המשתנים.(כך גם לcatch).

**טוב לדעת** – אחרי try חייב לבוא לפחות catch אחד ו/או finally. (אין חובה לשים catch אחרי try אם יש finally).

**אסור לרשום קוד בין סיום הtry לתחילת הcatch.**

Multi-catch - 

**טוב לדעת** – אם לתוכנה/אפליקציה יש מספר threads אז זריקה שלא נתפסה תגרום לסגירת הthread הספציפי בלבד. אם קיים רק thread אחד אז התוכנית תסגר.

Throw clause – מרמז מה החריגות אשר יכולות לקרות בקוד.

Throws – מוכרז לפני גוף הפונקציה, כאשר קיימת חריגה אפשרית אשר אנו לא רוצים לטפל בה בקטע הקוד הנוכחי ומוכנים לשלוח אותה לטיפול במחלקת האב.

Finally – מתרחש גם אם נזרקה חריגה וגם אם לא, יבוצע מיד לאחר הtry או מיד לאחר הcatch. מתרחש אפילו אם קיים return/break/continue. קיים מקרה יחיד שבוא finally לא יתרחש וזה אם יש system.exit באחת מהשורות לפני.

**טוב לדעת** – אם נזרקת חריגה אשר לא נתפסת ולאחר מכן נזרקת חריגה נוספת בfinally אז החריגה הראשונה נמחקת ונשלחת אך ורק החריגה מתוך הfinally.

מומלץ להימנע לשים קטעי קוד אשר יכולים לזרוק חריגה בתוך הfinally.

**במידה וארעה חריגה שלא טופלה בתוך פונקציה, החריגה תתגלגל (במעלה ההיררכיה של הקריאה ו/או ההורשה) עד שתטופל.**

**סוגי חריגות נפוצים**

1. חריגה מגבולות המערך (ArrayIndexOutOfBounds)
2. הצבעה על ערך NULL (NullPointer)
3. InputMismatch – למשל כאשר scanner דורש int אך מקבל string.

**מצגת 14 – Strings, Characters and Regular Expressions**

String – מחלק האב המיוחסת לטקסט, בעלת הרבה בנים.

Constructors

* Strin() – יוצר מחרוזת ריקה באורך 0 ("").
* String(string) – המילה עצמה
* String(charArray) – רץ על מחרוזת של char’s ומחבר אותם לידי string.
* String(charArray, int1, int2) – מתחיל מint1 ומחזיר אורך של int2 משם.

Character - מוכנס בתוך גרש יחיד (') – מייצג את הערך המספרי של האות על פי טבלת Unicode.

String – מוכנס בתוך גרשיים (")

מתודות

* Length – אורך המחרוזת
* charAt – האות ביקום מדויק
* getChars – מחזיר מערך של האותיות 🡨 ניתן גם לקבל רק חלק מהמערך.
* equalsIgnoreCase – מתעלם אותיות גדולות או קטנות.
* comareTo
  + 0 🡨 אם הם שווים.
  + 1- 🡨 אם המחרוזת שקראה לפונקציה קטנה יותר.
  + 1 🡨 אם המחרוזת שקראה לפונקציה גדולה יותר.
* regionMatches 🡨 (x, str, y, z)
  + x 🡨 מאיפה להתחיל
  + str 🡨 עם מי להשוות
  + Y 🡨 מאיפה להתחיל בstr
  + Z 🡨 כמה אותיות להשוות
* startWith(string)/endsWith(string) – משווה את ההתחלה/סוף של מחרוזת למחרוזת אחרת.
* indexOf(str) – מוצא את המיקום הראשון שהstr קיים, אחרת מחזיר -1.
* indexOf(str, int) – על אותו עיקרון רק מאיפה להתחיל לחפש.
* lastIndexOf(str) – מחזיר את המיקום האחרון שהstr מופיע, אחרת מחזיר -1.
* lastIndexOf(str, int) – אותו עיקרון רק מאיפה להתחיל לרוץ אחורה.
* Substring
  + עם ערך אחד 🡨 מאיפה לרוץ עד הסוף
  + עם 2 ערכים 🡨 מאיפה לרוץ עד לאן (לא כולל).
* Concat – מחבר 2 מחרוזות ביחד (דומה ל "+") **אך** לא דורס את המחרוזת הקודמת.
* Replace(str, str2) – מחליף כל str בstr2 במחרוזת.
* toUpperCase – יוצר מחרוזת חדשה עם אותיות גדולות בלבד.
* toLowerCase – יוצר מחרוזת חדשה עם אותיות קטנות בלבד.
* Trim – יוצר מחרוזת חדשה בלי רווחים בהתחלה או בסוף.
* valueOf – מחליף ערכים לstring.

StringBuilder – דינאמי ומשתנה, מאוד טוב לניצול נכון של הזיכרון, גדל וקטן בהתאם למחרוזת בו. סוג של arrayList של תווים.

היתרון המרכזי שלו הוא שמירה על המחרוזת כפי שהיא, גם אם היא בתוך פונקציה/בלי return ועוד.

Constructors

* stringBuilder() – בלי תווים בפנים, בגודל 16.
* stringBuilder(int) – בלי תווים בפנים בגודל int.
* stringBuilder(str) – יוצר מערך של תווים בגודל של המחרוזת + 16.

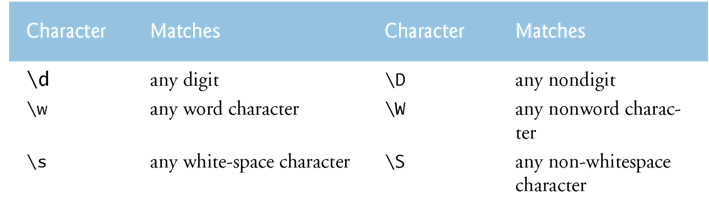
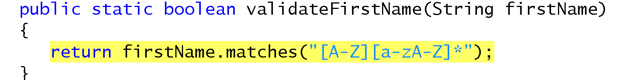
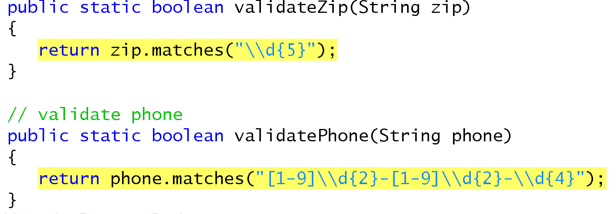
מתודות

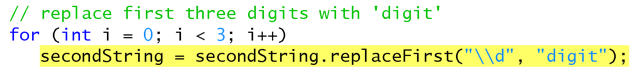
* toString() – מחזיר את המערך בצורה של string.
* Length – אורך המחרוזת הנוכחית בפנים.
* Capacity – אורך המערך (עם ערכים ריקים).
* setLength(int) – אם המילה ארוכה מהint אז חותכים את המילה וכל השאר זה null.
* charAt(int) – מחזיר את האות במיקום int.
* setCharAt(int, char) – מחליף במיקום int את האות עם char.
* getChars(x, y, arr, z)
  + x 🡨 מאיפה להתחיל
  + Y 🡨 עד איפה
  + Arr 🡨 לאן להכניס
  + z 🡨 מאיפה להתחיל להכניס בarr
* Reverse – הופך את המחרוזת
* Append(str) – מוסיף בסוף המחרוזת. ניתן לבצע שרשור apeend’s.
  + יכול לקבל כמה סוגים של משתנים כגון object, string, char[], char, Boolean, int, long, float, double.
* Insert(int, x) – מכניס את x במיקום int ומזיז את הערכים להיות אחריו.
  + X 🡨 יכול להיות הרבה סוגי משתנים כמו בappend.
* deleteCharAt(int) – מוחק במיקום int-1.
* Delete(x,y) – מוחק ממיקום x עד y-1.

Regular Expressions

"סט של חוקים" אשר יכולים לעזור לך לוודא שהinputאכן עומד בתקן הנדרש לך. מיכל characters וסימנים מיוחדים.

Matches – בודק ערך אל מול regular expression ומחזיר ערך בוליאני.

* נמצא בתוך [] (לדוגמא “[aeiou]” רק אותיות אלו)
* טווח מסומן ב"-" (לדוגמא “[A-Z]”)
* [^Z] – הכל **למעט** "Z" (שונה לגמרי מ[A-Y] אשר מאפשר רק אותיות A-Y בלי עוד סימנים, אפילו מותר "z")
* 
* “\*” – מצפה ל-0 או יותר.
* “+” – מצפה ל-1 או יותר.
* “|” – מצפה לאחד מהאפשרויות (לדוגמא “Hi (John|Jane)”).
* “?” – מצפה ל-0 או-1. **להיזהר עם "?", ימשיך לבדוק בתנאי הזה מבלי להמשיך כל עוד הוא מצליח!**
* ({n}) – מצפה לבדיוק n כאלה.
* ({n,m}) – מצפה לn עד m.
* ({n,}) – לפחות n.
* “\x” – מתייחס לתו x בלבד.
* replaceAll(x,y) – מחליף את כל הxים בy.
* replaceFirst(x,y) – מחליף את x בy. ניתן להכניס ללולאה ובכך להחליף יותר מהראשון, למשל 3 ראשונים

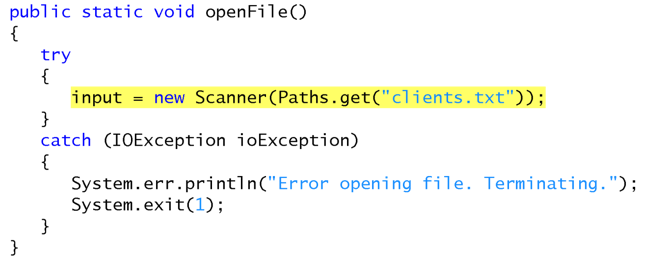
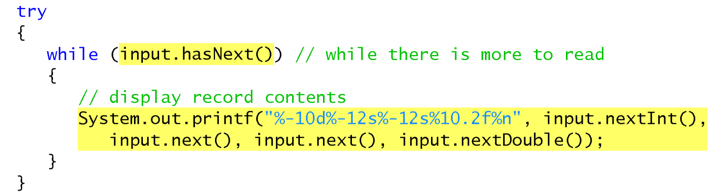


**חשוב לזכור** – regular expression **חייב** להיות String. אך ניתן לעשות השוואה עם הרבה סוגי משתנים.

**מצגת 15 – Files, Strams and Object Serialization**

File streams יכול לשמש לinput או output של data בתור bytes או characters.

Scanner - משמש לinput מהמקלדת או מקובץ.

* hasNext – קובע האם הגענו לסוף הקובץ (ctrl+z).
* משמש לקריאת קבצים
* 
* 

Path – מייצג את המקום של הקובץ או התיקייה. **לא פותח או מתעסק עם הקובץ עצמו!** (קיימים interfaces נוספים אשר דומים וגם הם מתעסקים רק במיקום כגון directoryStream, Paths, Files. כולם תחת java.nio).

Formatter – אם קיים קובץ אז פותח ואפשר לעבוד עליו. אם לא קיים קובץ אז יוצר כזה.

* מאפשר לעבוד על קובץ. למשל להכניס רשומות חדשות.
* **חשוב לסגור (close) בסוף השימוש (סוגר את הקובץ).**
* אם לא התבצע close אז הקובץ ייסגר אוטומטית כאשר התוכנית תיסגר.

בדרכים הנ"ל עדכון פרטים בקובץ גורר כתיבה מחדש של הקובץ עם הנתון החדש, דבר הלוקח זיכרון רב וזמן עיבוד ארוך.

Serialization

דרך לכתוב אובייקט שלם לקובץ. מכיל אוסף של bytes עם האובייקט עצמו והסוג שלו.

לאחר שאובייקט נשמר בעזרת serialized ניתן לקרוא אותו משם בעזרת deserialized.

objectInput/objectOutput – מאפשר לכתוב/לקרוא אובייקטים שלמים.

objectInputStream/objectOutputStream – **חובה לממש serializable insterface!!!**

* writeObject – לוקח אובייקט ומכניס אותו כמו שהוא. (מיוחס לoutput).
* readObject – מוציא את הקובץ. (מיוחס לinput).

**כדאי לדעת** – serializable לא מכילה מתודות כלל, היא אך ורק על מנת לתייג שהקבצים יכולים לעבור תהליך serialize or deserialize.

* כל משתנה במחלקות הממשות serializable חייב להיות serializable בעצמו. אחרת חייבים להכריז עליו כ”transient” בשביל שיתעלמו ממנו בעת הקריאה והכתיבה של האובייקטים.
* כל הprimitive-types הם serializable.

חריגות

* securityException – אם למשתמש אין גישה לערוך את הקובץ.
* fileNotFoundException – אם הקובץ לא קיים וגם לא ניתן לייצור אחד חדש.
* noSuchElementException – קורה אם הscanner מקבל משהו שונה ממה שהוא ציפה או שאין data.
* FormatterClosedException – אם סגרת את הformatter אך אתה מנסה לעבוד איתו.
* EOFException – אם readObject ניסה לקרוא אובייקט מעבר לסוף הקובץ.
* ClassNotFoundException – אם readObject לא מצליח לזהות את סוג האובייקט.

**מצגת 17 – Lambdas and Streams (Java8+)**

**Lambdas**

Functional programming - ציין מה אתה רוצה להשיג אבל לא איך להשיג אותו.

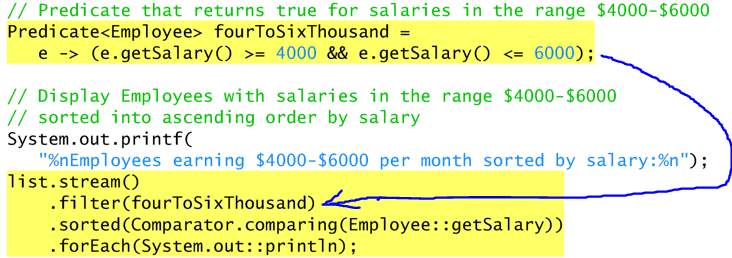
מתודות אנונימיות

פקודות למבדה "מבינות" לבד את סוג המשתנה הנדרש בזכות התוצאה הצפויה או בזכות הקלט הנכנס.

צורות כתיבה של פונקציה

*  - אנחנו הגדרנו שהמשתנה הוא מסוג int.
*  - הפונקציה תבין לבד מה סוג הפרמטרים לפי מה שנכניס.
*  - ניתן לרשום כך **רק** אם יש ביטוי אחד בלבד בגוף.
*  - אם יש רק פרמטר אחד אפשר להשמיט את הסוגריים.
*  - ניתן לרשום פונקציה ללא פרמטרים כלל.

ניתן לייצר למבדה אשר יהיה אפשר לממש אותה בעתיד מספר פעמים (שימוש בpredicate).

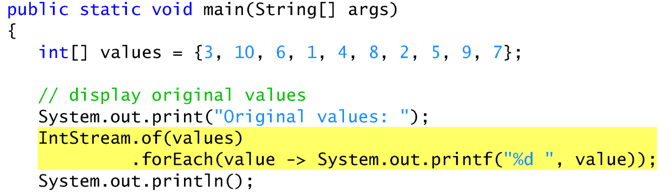
דוגמא : 

**Streams**

**מה זה?** – עוטף את הנתונים שלנו, מאפשר לנו לבצע פעולות פונקציונליות על המידע ובכך לבצע שרשור פעולות. (bulk functions).

קיימים סוגים שונים של input (מקבלים מערך כערך בסוגריים) כגון:

* IntStream(arr)

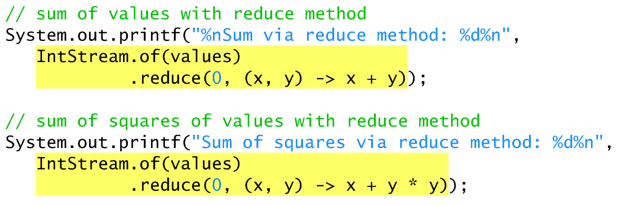


* LongStream(arr)
* DoubleStream(arr)

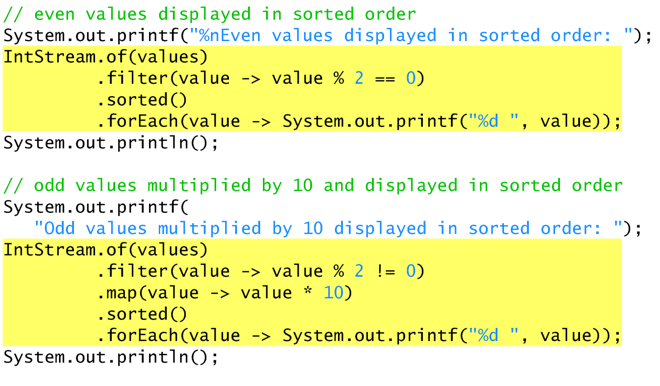
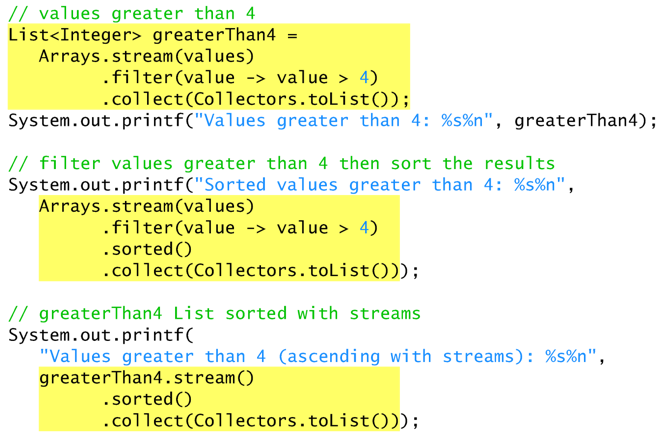
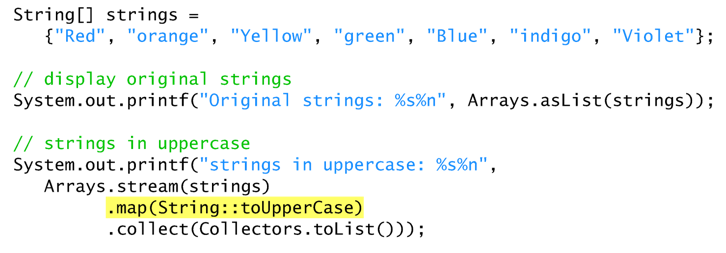
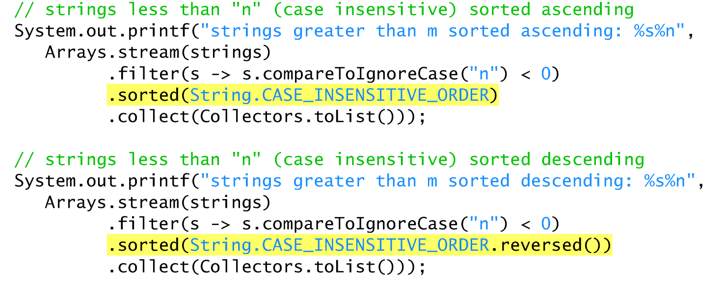
**חשוב לדעת -**  **במקום לבחור את סוג המשתנה של הסטרים אפשר לעשות Arrays אשר מתייחס לכל המשתנים בהתאים למה שהוא מקבל.**

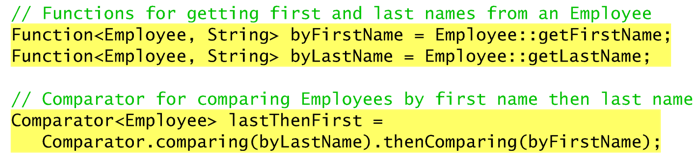
* אובייקטים אשר ממשמים interface Stream.
* מאפשר לך לבצע functional programming.
* לא בעל שטח אחסון, לא ניתן להשתמש בנתונים שלו שנית מפני שהוא לא משאיר העתק של הנתונים בזיכרון.
* **מחזיר stream חדש כתוצאה.**

פעולות נפוצות

* Filter(condition) – מחזיר רק מה שבתוך התנאי.
* Distinct() – מחזיר unique.
* Limit(int) – מחזיר int ערכים מתחילת הריצה.
* Map(x) – מחזיר stream חדש עם הערכים המבוקשים למה שרשום בx (יכול גם להיות פונקציה).
* Sorted – מחזיר stream חדש ממוין. (**כמובן** חייב להיות בעל אפשרות השוואה או שתבוצע השוואה ברירת מחדל של ג'אווה, למשל מחרוזות בסדר אלפבית).
* Range(x,y) – רוץ כint על הערכים מ-x עד y-1.
* rangeClosed(x,y) – כמו range רק עד y כולל.
* forEach()
* average() – ממוצע
* Sum() – סכימה של הערכים עצמם.
* count() – סכימה של מספר הערכים (לא הערך שלהם)
* max() - מקסימום
* min() - מינימום
* Reduce(x, function) – ביצוע פעולות מתמטיקה שונות. (x -מה הוא הערך ההתחלתי)
  + 
* Collect – יוצר new collection.
* toArray – יוצר new array.
* toList – הופך Stream<T> לList<t>.
* getAsDouble – מחזיר אובייקט מסוג double ואם לא ניתן זורק NoSuchElementException.
  + ניתן למנוע שגיאה זו בשימוש orElse.
* פעולות חיפוש
  + findFirst() – מצא את הערך הראשון (בדר"כ מתבסס על סדר פעולות שהיה לפני). **יתרון מרכזי:** מפסיק את הפונקציה ברגע שמוצא.
  + findAny() – מחזיר את כל הרשימה (בהתבסס על סדר פעולות שהיה לפני)

דוגמאות לפונקציות

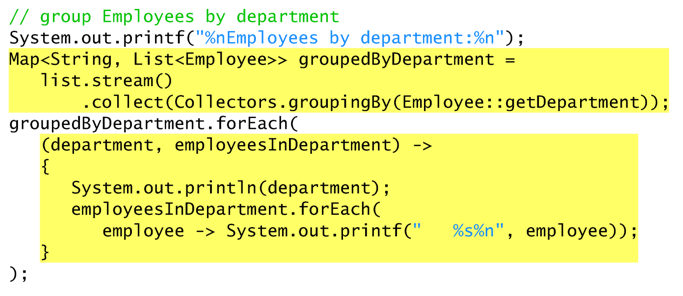
* 
* 
* 
* **שים לב! שימוש ב”::” בתוך הmap של המחרוזות.**
  + **שימוש ב:: מתבצע כאשר רוצים להשתמש בפונקציה מסוימת מתוך מחלקה, לדוגמא:** *Employee::getId.*
  + עוד דוגמא מעניינת : *System.out::printLn.*
* 
* **ניתן לבצע שרשור השוואת, להשוות לפי x ואז להשוות לפיy – שימוש במתודה thenComparing:**

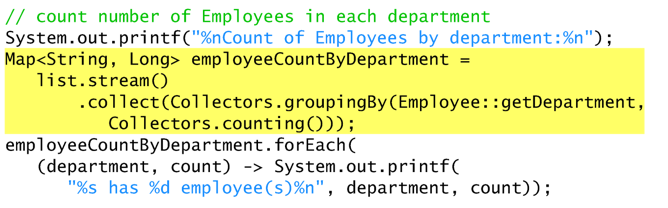


* **groupingBy – מבצע חלוקה כמו groupBy בSQL.**

**ניתן לראות שהחלק הראשון מבצע חלוקה ומכניס לMap חדש. החלק השני רץ על הMap ומדפיס (אין קשר בין הפונקציות)**

המשתנה *list* הינו arrayList של Employee.

דוגמא נוספת:



**מצגת 20 – Generic Classes and Methods**

הרעיון המרכזי הוא לייצר פונקציה אחת אשר יכולה לטפל בכמה סוגי משתנים, למשל הדפסה לסוגים שונים של מערכים. אחד מסוג int השני מסוג double והשלישי מסוג char.

ממומש בעזרת האות **T**. (<T>)

**מושג חשוב** **upper bound** – גבול עליון של משתנה, כלומר עד לאן הוא יכול לטפס במעלה ההיררכיה. בדרך כלל עד object אשר הוא הגבול העליון של כל המחלקות והמשתנים.

**טוב לדעת –**

* + T=type
  + E=element
  + K=key
  + V=value

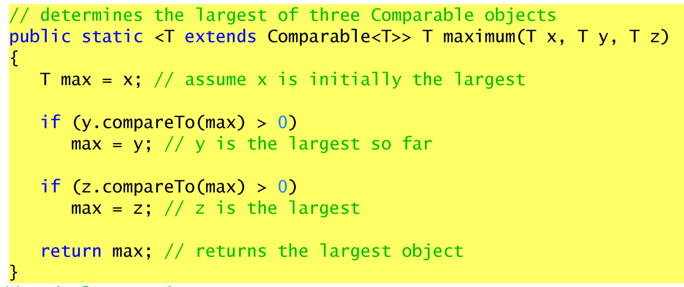
Erasure – הפעולה של החלפת הT בסוג המשתנה האמיתי. (ברירת המחדל היא החלפה למשתנה object).

* **עובדה מעניינת** – כחלק מהפעולה של erasure הוא מוסיף באופן אוטומטי casting לפני כל הפעולות הדורשות זאת.

דוגמה טובה לכך היא compareTo אשר מגיעה ממחלקת Comparable<**T**>.

**חשוב לזכור** – רק משתנים אשר ממשמים בעצם Comparable יכולים להיות בשימוש.

דוגמא :



פונקציות גנריות יכולות להידרס על ידי פונקציות גנריות חדשות, ופונקציות לא גנריות.

הקומפיילר מחפש את ההתאמה הטובה ביותר בשבילו בעת הריצה ואתה הוא בוחר בעת ההרצה.

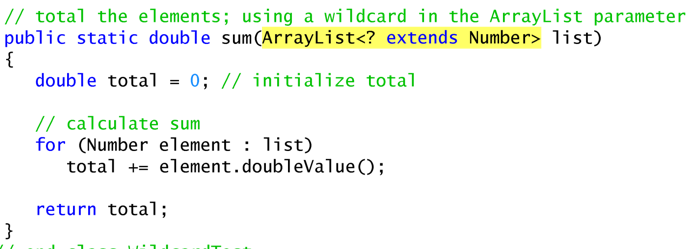
מחלקה **Number** – מחלקת אב של Integer ושל Double.

Wildcards

מסומן על ידי "?" (משתנה לא ידוע – unknown type).

למעשה מכריז שהפונקציה המבוקשת יכול להיות במחלקת האב ו/או באחת ממחלקות הבן שלו.

דוגמא לשימוש בפונקציית sum:



**מצגת 24 – Accessing Databases with JDBC**

JDBC למעשה מאפשר לחבר את הjava לdatabase.

Db\_url – “jdbc:derby:books”;

* Jdbc = פרוטוקול.
* Derby = תת-פרוטוקול. (קיימים עוד רבים כגון: mysql/oracle:thin/db2/sqlserver ועוד)
* Books = מיקום הDB הרצוי (שם הDB).

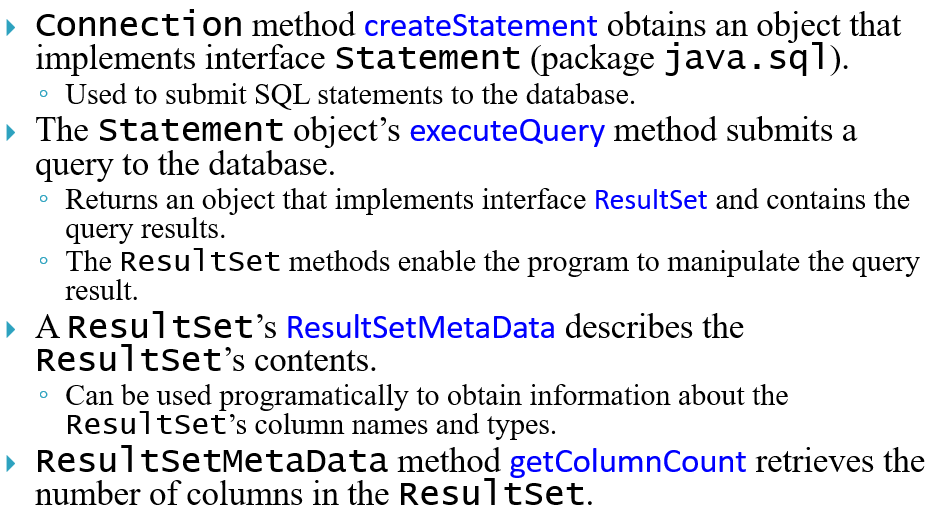
חיבור לDB : 

Deitel1 = שם משתמש deitel2 = סיסמא (של בעל הגישה לDB)

**סדר הפעולות בעת שימוש בDB (בעת סגירה סדר הפעולות הוא הפוך!)**

1. Connection
2. Statement
3. ResultSet – מכיל את תוצאות השאילתה
4. ResultSetMetaData – מספקת מידע על מבנה השאילתה

הסבר על הפקודות הדרושות בעת שימוש בSQL:



**כדאי לזכור**

* resultSet מתחיל משורה אחת לפני השורה הראשונה! SQLException קורה לרוב אם ניסת לגשת לresultSet מבלי לעשות resultSet.next לפני.
* **אין** דבר כזה column index 0! האינדקס הראשון הוא "1" (בשונה ממה שאנחנו יודעים על רשימות ומערכים).
* SQLException יכול לקרות גם כאשר אנחנו מנסים לבצע שינויים בresultSet לאחר שכבר סגרנו את הStatement.
* אם הדרייבר שלך לא תומך חזרה אחורה בResultSet תקבל שגיאה מסוג SQLFeatureNotSuportedException.
* דוגמא נוספת לשגיאה כזו – ניסיון לעשות update מבלי שהדרייבר תומך בכך.

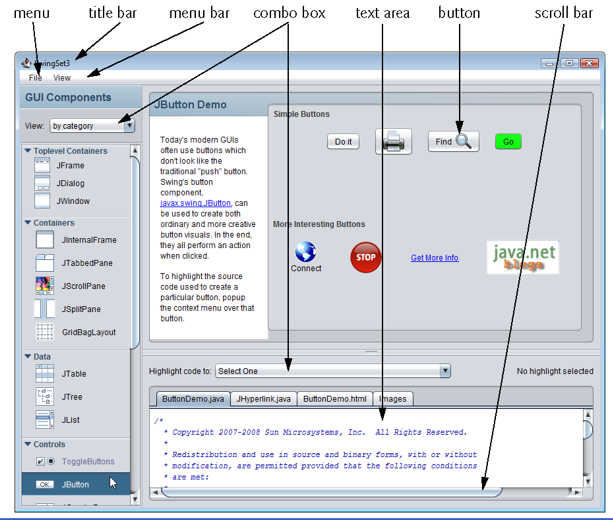
preparedStatements –

מאפשר ליצור שאילתות עם פרמטרים.

* מסומן ב-?
* מתחיל מספור מ"1" לפי סדר הסימני שאלה.
* יכול לשמש גם כהוספה וגם כקבלת מידע.
* סוג של פונקציית SQL שבונים לפני.

**מצגות 12+22 – GUI**

הצגה גרפית של java.



פקודות

* JOptionPane.showInputDialog(str) – מציג חלון