TABLE OF CONTENTS

1. שפת JAVA היסטוריה 7
1.1. ידע כללי 7
1.2. תכנות מונחה עצמים - OBJECT ORIENTED PROGRAMMING 8
1.3. שלושת העקרונות - THE THREE OOP PRINCIPLES 8
1.4. ריכוזיות - ENCAPSULATION 9
1.5. הורשה - INHERITANCE 10
1.6. רב צורתיות - POLYMORPHISM 11
2. סוגי נתונים, משתנים ומערכים 13
2.1. סוג המרה ו CASTING 13
2.1.1. המרה אוטומטית 13
3. מחלקות 14
4. מבט נרחב על פונקציות ומחלקות 15
5. טעינת פונקציות 17
5.1. הבדל בין טעינת פונקציות 19
5.2. טעינת פונקציות עם נתונים פרימיטיביים 20
5.3. טעינה של ערכים מוחזרים 23
5.4. קונסטרקטור DEFAULT 23
5.5. מילת הייחוס THIS 24
5.6. קריאה מקונסטרקטור לקונסטרקטור 26
5.7. הבנת STATIC 27
5.8. איסוף "זבל" 27
6. חבילות וממשקים – PACKAGES & INTERFACES 31
6.1. חבילה – יחידת הספרייה - PACKAGE: THE LIBRARY UNIT 31
6.2. יצירת חבילות ייחודיות – CREATING UNIQUE PACKAGE NAME 33
6.3. שימוש בספרייה שאנו יוצרים 35
6.4. שימוש ב IMPORT על מנת לשנות התנהגות קוד 37
6.5. בקרת גישה 39
6.5.1. "ידידותי" 39
6.5.2. בקרת גישה - ציבורי **Public** 39
6.5.3. חבילת "ברירת המחדל" - **The"default package"** 40
6.5.4. בקרת גישה – פרטי **Private** 41
6.6. בקרת גישה PROTECTED 43
6.7. הפרדת יישום וממשקים 45
6.8. גישה למחלקות 46
7. הורשה 48
7.1. הורשה – היכרות 48
7.2. משתנה של מחלקת על המתייחס לאובייקט של תת מחלקה 52
7.3. שימוש בפונקצית ()SUPER 54
7.4. יצירת היראררכיה עם רמות 58

7.5. שימוש בקונסטרקטורים 61
7.6. שכתוב פונקציות – METHOD OVERRIDING 62
7.7. שליחת פונקציות דינאמית 65
7.8. מדוע לשכתב פונקציות 66
7.9. יישום שכתוב פונקציות 67
7.10. שימוש במחלקה בונה 69
7.11. שימוש ב FINAL בהורשה 73
7.11.1. מניעת שכתוב 73
7.11.2. מניעת הורשה 74
7.12. אובייקט המחלקה 74
7.13. UPCASTING 75
7.14. מדוע להשתמש ב UPCASTING 76

8. רב צורתיות – POLYMORPHISM 77

8.1. מודל UPCASTING 77
8.1.1. ומה קורה באם נשכח מהו סוג האובייקט? 78
8.2. מודל ה TWIST 80
8.2.1. פונקציה הקוראת לאובייקט עטוף 80
8.3. כיצד נקבע את ההתנהגות הנכונה? 81
8.3.1. הארכה – **Extensibility** 84
8.4. מודל DOWNCASTING 87
8.5. שכתוב VS טעינה 89

9. טיפול ביוצאי דופן – EXCEPTION HANDLING 90

9.1. שימוש ב TRY 91
9.2. שימוש ב CATCH 91
9.3. שימוש ב THROW 93
9.4. שימוש ב THROWS 94
9.5. שימוש ב FINALLY 95
9.6. ארגומנטים ביוצאי הדופן 97
9.7. יצירת יוצאי דופן משלנו 98
9.8. תפיסת כל טעות 101
9.9. זריקה נוספת של טעות 103
9.10. יוצאי דופן מובנים במערכת 105

10. קלט/פלט ב – JAVA 107

10.1. סטרים – STREAMS 107
10.2. מחלקת FILE 110
10.3. ספריות – DIRECTORIES 112
10.3.1. שימוש בפילטר שמות 113
10.4. מחלקת STREAM 115
10.4.1. קלט – **InputStream** 115
10.4.2. פלט – **OutputStream** 115
10.4.3. קובץ קלט – **FileInputStream** 115
10.4.4. קובץ פלט – **FileOutputStream** 117
10.4.5. יישום **StringBufferInputStream** 119
10.4.6. יישום **BufferedInputStream** 119
10.5. גישה אקראית לקובץ 121

11. טיפול ב STRING 124

11.1. קונסטרקטור ל STRING 124

11.2. אורך ה String 126
11.3. שרשור Strings 127
11.4. המרת String תוך שימוש ב ()toString 128
11.5. השוואת Strings 129

12. כלי עזר נוספים – JAVA UTILITIES 131

12.1. שימוש ב Vector 132
12.2. שימוש במחלקת Random 135
12.3. שימוש במחלקת Stack 137

13. מחלקת APPLET 139

13.1. אפלט בסיסי – Basic Applet 140
13.1.1. הוראות ל**applets** 140
13.1.2. יתרונות **applets** 140
13.1.3. מסגרת העבודה האפליקטיבית 141
13.2. מחלקת Applet 142
13.3. ארכיטקטורת Applet 143
13.4. הרצת applets מתוך דפדפן 144
13.5. שימוש ב Appletviewer 145
13.6. בדיקת applets 145
13.7. הרצת applet מתוך שורת פקודה 146
13.8. סדר טעינת ופריסת ה Applet 147
13.9. שימוש בפונקצית ()update 147
13.10. שימוש בפונקציות ב Applet 148
13.11. בקשה לצביעה מחודשת 149
13.12. שימוש בחלון הודעת המצב 153

14. מחלקת EVENT 154

14.1.1. טיפול באירוע עם עכבר 155
14.1.2. טיפול באירוע עם מקלדת 158
14.2. הבנת HTML & Applet Tags 160
14.3. העברת פרמטרים ב Applet 161
14.4. שימוש בפונקציות ()getCodeBase ו ()getDocumentBase 163
14.5. שימוש בממשק AppletContext ובפונקצית ()showDocument 164

15. חלונות, גרפיקה וטקסט תוך שימוש ב AWT וב SWING 165

15.1. מחלקות AWT 165
15.2. תכנות ראשוני בחלונות 165
15.2.1. רכיב - **Component** 166
15.2.2. תכולה - **Container** 166
15.2.3. פאנל - **Panel** 166
15.2.4. חלון - **Window** 166
15.2.5. מסגרת - **Frame** 166
15.2.6. חלון ריק - **Canvas** 166
15.3. עבודה עם חלונות ומסגרות 167
15.3.1. פונקציות נוספות לשימוש עם **Frame**: 167
15.4. מאורע WINDOW_DESTROY 168
15.5. יצירת חלון מסגרת בתוך Applet 169
15.6. טיפול במאורע של חלון עם מסגרת 172
15.7. יצירת חלון עצמאי 176

.16 פונקציות גרפיקה תוך שימוש ב JAVA2D 178

.16.1.1 ציור קווים 178
.16.1.2 ציור ריבוע 179
.16.1.3 ציור אליפסות ומעגלים 180
.16.1.4 ציור **Arc** 181
.16.1.5 ציור **Polygons** 182
.16.1.6 מתן מידות לגרפיקה 183
.16.2 עבודה עם צבעים 184
.16.2.1 פונקציות צבע 184
.16.2.2 איפוס ה **Mode** לעבודה 187
.16.3 שימוש בפונטים 189
.16.3.1 קביעת פונטים 189
.16.3.2 יצירה ושימוש בפונטים 190
.16.4 מחלקת FONT MATRICS 192
.16.5 מירכוז טקסט 194
.16.6 חלון CANVAS 195
CANVAS הנו רכיב המייצר לנו איזור לציור ובד"כ אנו נשתמש בו כבסיס לרכיבים חדשים. 195

.17 שימוש בשליטה, ניהול תוצאה ותפריטים תוך שימוש ב AWT ובמודול SWING 197

.17.1 יסודות שליטה ברכיבים 197
.17.2 הוספה/הסרה של שליטה 197
.17.3 תגובה לשליטה 198
.17.4 היכרות ל SWING 198
.17.5 מאורעות מבט מקרוב 199
.17.6 שימוש ב LABELS 200
.17.7 שימוש בכפתורים 202
.17.8 שימוש בכפתורי בחירה 204
.17.9 כפתורי RADIO 207
.17.10 רשימה DROP DOWN 210
.17.11 שימוש ברשימות 212
.17.12 שימוש בגלגלת 215
.17.13 שימוש בשדות טקסט 218
.17.14 שימוש באיזור טקסט 222
.17.15 שימוש בקביעת תצורה 224
.17.15.1 מערך זרימה – **FlowLayout** 224
.17.15.2 מערך גבול – **BorderLayout** 227
.17.15.3 מערך מרווח – **Inset Layout** 229
.17.15.4 מערך פאנל - **Panel Layout** 230
.17.15.5 מערך רשת – **Grid Layout** 231
.17.15.6 מערך כרטיס – **Card Layout** 233
.17.16 שימוש בתפריטים 236
.17.17 שימוש בקופסת דיאלוג 241
.17.18 שימוש בתפריטי POP UP 245

.18 תמונות – שימוש ב AWT 247

.18.1 יצירת אובייקט עם תמונה 247
.18.2 טעינת תמונות 248
.18.3 הצגת תמונה 248

.19 ENTERPRISE JAVABEANS 249

19.1. JavaBeans vs. EJBs 249
19.2. The EJB specification 250
19.3. EJB components 250
19.4. The pieces of an EJB component 251
19.5. EJB operation 252
19.6. Types of EJBs 252
19.7. Developing an EJB 253

20. תכנות ב MULTITHREADING 254

20.1.1. מודל ה **Thread** 254
20.1.2. תזמון **Threads** 254
20.1.3. סינכרון 255
20.1.4. הודעה – **Massages** 255
20.2. ה – Thread הראשי 256
20.3. יצירת Thread 258
20.3.1. יישום **Runnable** 258
20.3.2. הרחבת מחלקת **Thread** 261
20.4. יצירת Multiple Threads 262
20.5. שימוש בפונקצית isAlive() וב join() 264
20.6. שימוש בפונקצית suspend() וב – resume() 266
20.7. תזמון Threads 268
20.8. סינכרון – Synchronization 271
20.8.1. שימוש בפונקציה לסינכרון 271
20.8.2. שימוש בהצהרת סינכרון 274
20.9. תקשורת פנימית בין Threads 276
20.10. באג מסוג – Deadlock 279

21. תקשורת 281

21.1. תכנות מערכת 281
21.1.1. זיהוי מכונה 281
21.2. מודל Client/Server 284
21.2.1. בדיקת תכניות ללא תקשורת 284
21.2.2. שימוש ב **Port** 285
21.2.3. מהו **Socket**? 285
21.2.4. שירות למספר לקוחות 292
21.2.5. שימוש ב **Datagrams** 296
21.2.6. שימוש ב **URL** מתוך **Applet** 296
21.2.7. קריאת קובץ משרת 298
21.3. בסיס נתונים JDBC 299

22. שפת JAVA ושימוש ב- XML 301

22.1. מהו XML? 301
22.2. הבדלים בין XML ובין HTML 301
22.3. שפת XML לא עושה הכל 301
22.4. יצירת תגים ב XML 302
22.5. חוקי XML 302
22.6. שפה גמישה וחופשיה 302
22.7. XML כהשלמה ל HTML 303
22.8. הפרדת נתונים 303
22.9. העברות מידע 303

.22.10 שימוש XML ב B2B 303
.22.11 XML ושפות נוספות 303
.22.12 אלמנטים 305
.22.13 שימוש נכון בתגים 305
.22.14 שורש מסמכים 306
.22.15 אלמנטים המכילים מאפיינים 306
.22.16 תכונות 307
.22.17 שימוש ברווח 307
.22.18 הערות ב XML 307
.22.19 שורה חדשה ב XML 307
.22.20 אלמנטים והארכות ב XML 308
.22.21 יחס אלמנטים ב XML 309
.22.22 אלמנטים ותכולתם 310
.22.23 מתן שמות לאלמנטים 310
.22.24 אלמנטים VS מאפיינים 311
.22.25 אחסון נתונים באלמנט "ילד" 312
.22.26 חסרונות בשימוש מאפיינים 312
.22.27 דפדפנים שימושיים 312

.23 ולידציה 313

.23.1 שימוש ב- DTD 313
.23.2 DOCUMENT TYPE DECLARATIONS 315
.23.3 סכימות 316

.24 פרוטוקולים ב- XML 318

.24.1 פרוטוקול RSS 318
.24.2 פרוטוקול XML-RPC 318
.24.3 פרוטוקול SOAP 318
24.4. XML AS A MESSAGE FORMAT 318
.24.5 כתיבת XML 323
.24.6 OUTPUT STREAMS, WRITERS, AND ENCODINGS 329
.24.7 שימוש ב- XML-RPC 333
.24.8 שימוש בפרוטוקול SOAP 335

.25 קריאת קובץ XML 337

25.1. INPUTSTREAMS AND READERS 337
25.2. XML PARSERS 339
.25.2.1 בחירת API לטובת שימוש ב-XML 339
.25.3 מימוש SAX 339
.25.4 שימוש ב-DOM 342
.25.5 שימוש ב-JAXP 344

.26 שימוש ב- XSLT 346

.26.1.1 שימוש בערך הקיים ב-**Node** 347

בס"ד

הנדסת תוכנה - שפת Java

1. שפת Java היסטוריה

שפת Java פותחה והומצאה לראשונה ע"י סאן מיקרוסיסטמס פיתוחה החל בשנת 1991 בגרסאות ראשונות ובסיום פיתוח של 5 שנים לאחר שינוי גרסאות רבות הרי שהתוצר הייתה שפת התכנות Java. שפת התכנות הזו הנה הרחבה לשפות קיימות כגון C++ ומטרתה לתת פתרון לתכנות מונחה עצמים למשך שנים שפה זו שפותחה הוכרה ע"י ארגון הסטנדרטים העולמיים. בשנת 1996 אורגנה קבוצה שהכניסה את שפת Java לקבלת סטנדרטים של שפת תכנות. שפת Java הוכנסה ל – ANSI (American National Standard Institute) . הכנסת סטנדרטים חדשים לשפה הנו תהליך ארוך הלוקח בד"כ מספר שנים ולבסוף שפת Java הוכרה בשנת 1996 כשפה עם סטנדרטים של הנדסת תוכנה.

1.1. ידע כללי

מבנה ג'נרי – Generic Structure - מבנה זה מוגדר בכל שפות התוכנה כמבנה האב. מבנה זה מבוסס על הסטנדרטים של שפת C . אי לכך על סמך מבנה זה יש ביכולתך לפתח תוכניות.
סינטקס – Syntax – מוגדר כדקדוק של שפת התכנות (תחביר).

1.2. Object Oriented Programming - תכנות מונחה עצמים

גישת תכנות מונחה עצמים הוא בסיס ליבת שפת תכנות זו. תכנות מונחה עצמים הוא כל כך מהותי ב JAVA שיש צורך להבין מה הוא הרעיון החבוי בגישה שכזו לפני שאנו כותבים תוכניות למיניהם.
כידוע לנו כל תוכנית מחשב כוללת בתוכה שני רכיבים עיקריים שהם קוד ונתונים. על מנת לנהל רמת מורכבות גבוהה אנו נשתמש בגישה של תכנות מוחנה עצמים. הרעיון של תכנות מונחה עצמים הנו סידור התוכנית מסביב לנתונים הכוללים גישה לנתונים.
ובהסבר פשטני יותר, המחשבה בשפת תכנות זו הנה מתן פתרונות תיכנותיים ברמת חשיבת "אנוש" קרי, אדם בטבעו חושב ע"י אובייקטים ופותר בעיות כאלו ואחרות ע"י שימוש באובייקטים.
דוגמא: האדם אינו חושב על מכונית כעל מכלול של 1000 רכיבים אלה מצטיירת במוחו מחשבה על מכונית כאובייקט המכיל בתוכו את מאות תת האובייקטים.

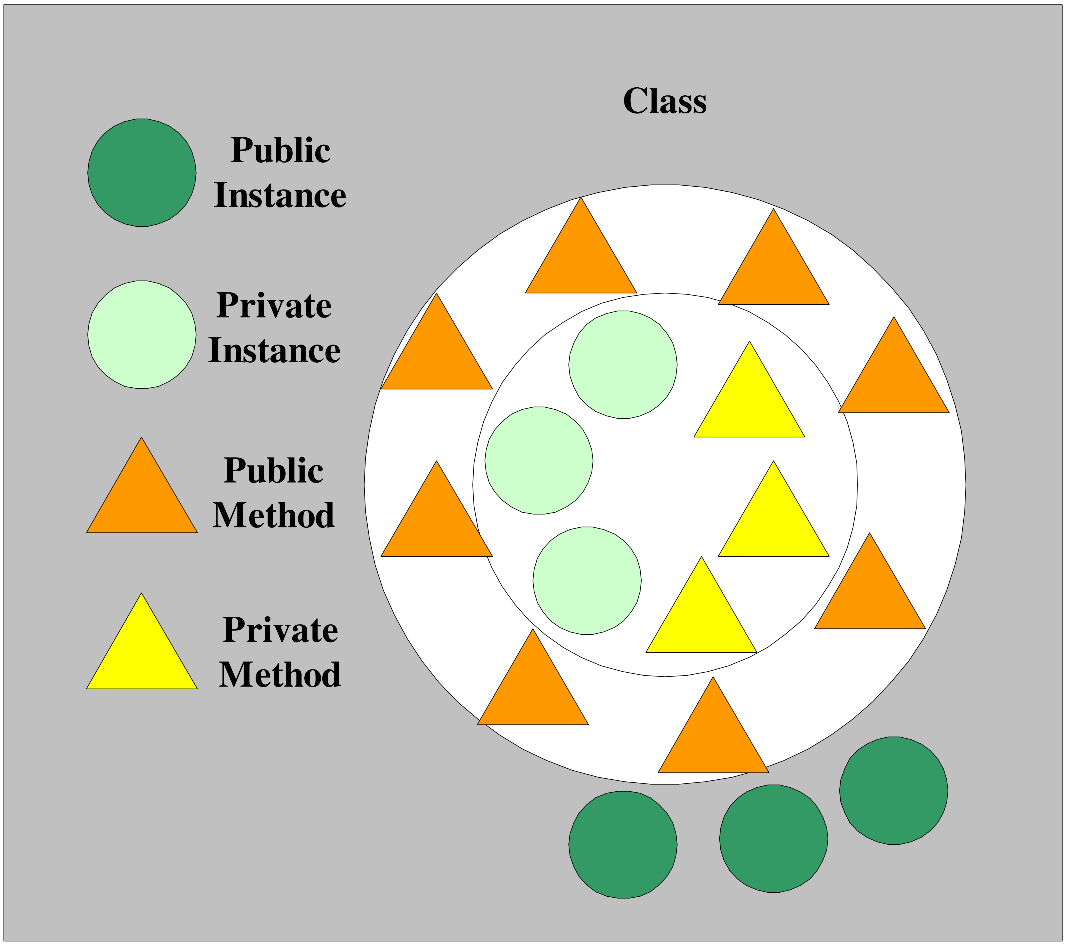
1.3. The Three OOP Principles - שלושת העקרונות

תכנות מונחה עצמים עוזר לנו לייצור מכניקה בה אנו נוכל לבצע מודל מוחה עצמים. שלושת העקרונות ב JAVA הנם:

1. Encapsulation - ריכוזיות
2. Inheritance - תורשתיות
3. Polymorphism – רב צורתיות

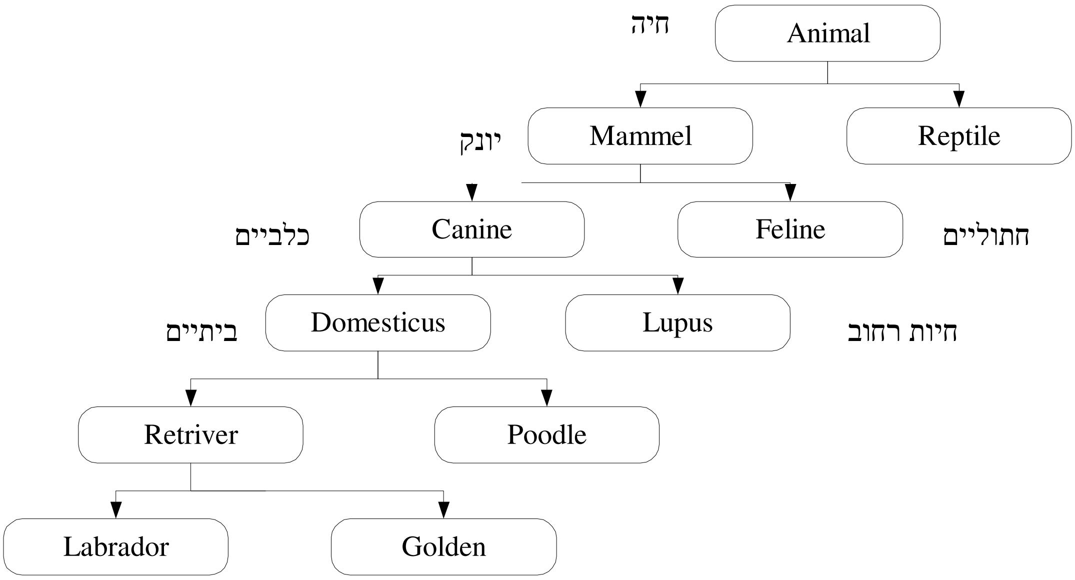
1.4. ריכוזיות - Encapsulation

זוהי מכניקה שמכילה ביחד קוד ונתונים ושומרת אותם ביחד כך שהפרעות חיצוניות או שימוש בקוד אינם נגישים. בשפת JAVA נוכל להתייחס לפעולה שכזו ע"י הגדרת מחלקה CLASS .
מחלקה בעצם קיומה מגדירה את מבנה הקוד והנתונים שמטבע הדברים יחלקו מספר פעולות עם אובייקטים נוספים. מהסיבה הזו אובייקטים בד"כ ייוחסו כמחלקות או instance .
אם כך ניתן לומר כי מחלקה הנה מבנה לוגי והאובייקט הנו המבנה האמיתי שמופעל בתוך המחלקה.
כאשר אנו ניצור מחלקות אנו נשתף קוד ונתונים שהם חלק בלתי נפרד מהמחלקה. לנתונים משותפים אלו אנו נקרא Members של המחלקה. הקוד שמפעיל את ה Members נקרא Method Members או רק Method .
לפיכך, על מנת שנתונים יוכלו לבצע קוד מסוים הרי שלא בהכרח כלל הנתונים אמורים להיות פעילים בתוך המחלקה. כתוצאה מכך, קיימת ב JAVA מכניקה שימושית להגדרת נתונים כללים או פרטיים.
הגדרות אלו מיוחסות כ Private Variables או Public Variable.
Public – גישה "ציבורית" למחלקה משמע, כל דבר חיצוני שמהמחלקה אמורה להשתמש בו.
Private – לשימוש נקודתי במחלקה בלבד לפיכך כל קוד חיצוני אינו יכול לגשת לקוד הנתון כפרטי.



1.5. Inheritance - הורשה

זהו תהליך שבו אובייקט אחד מכיל את המאפיינים של אובייקט אחר. זוהי תכונה חשובה מכיוון שהיא תומכת בקונספט של מחלקות היראכיות.
לדוגמא כלב מסוג גולדן רטריור הוא חלק ממחלקת כלבים, שהוא בעצם חלק ממחלקת יונקים שהוא בעצם מתחת מחלקת "על" של חיות.
ללא השימוש בהיררכיות היינו צריכים להגדיר לכל אובייקט את כלל המאפיינים. לפיכך ע"י שימוש בהיראכיה באובייקט יוגדרו רק אותם רכיבים ייחודיים שאנו צריכים.
נוכל לומר כי אובייקט מסוים יכול לרשת תכונות כלליות מההורה שלו.



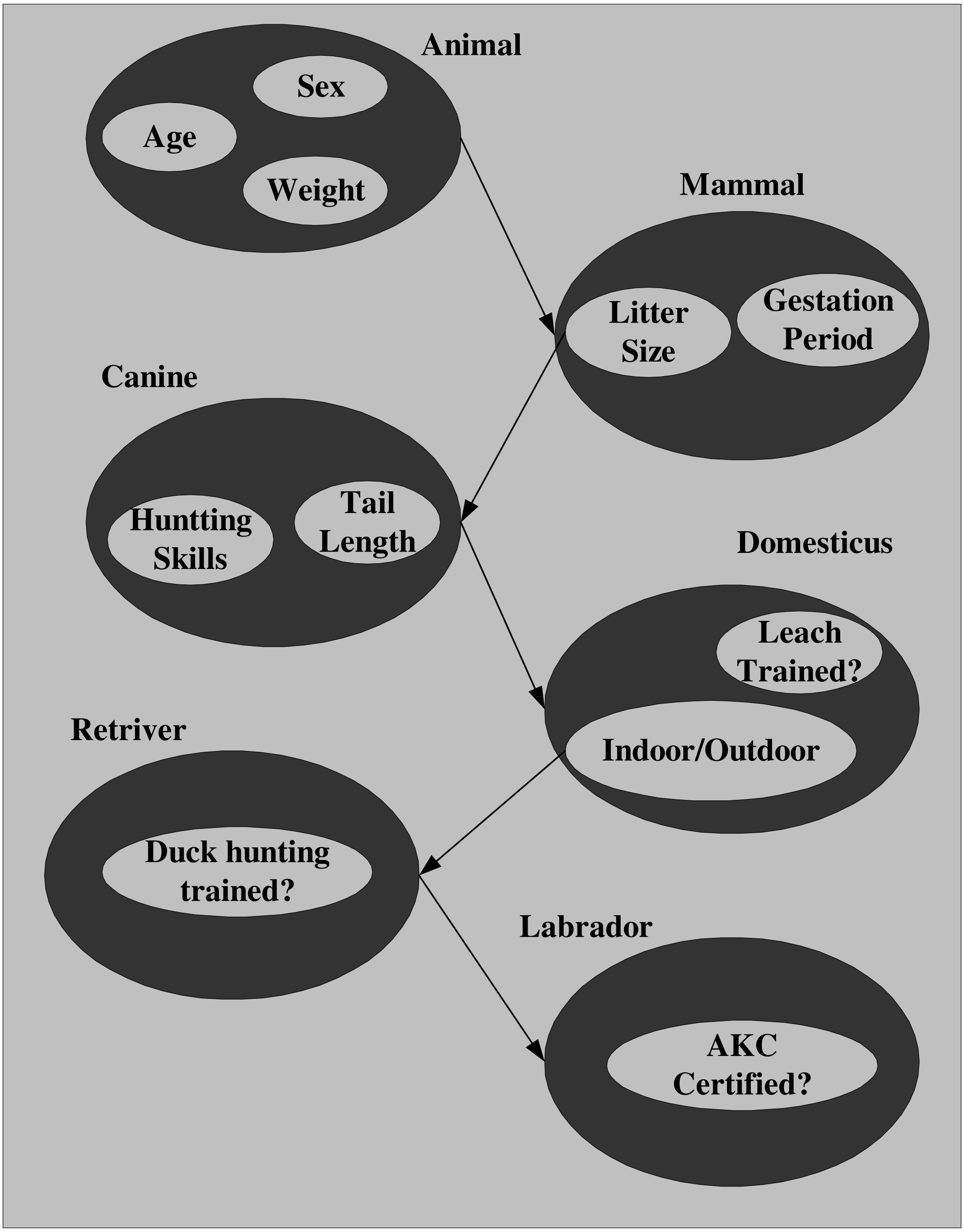
1.6. Polymorphism - רב צורתיות

הנה מילה יוונית וכוונתה "צורות רבות". תכונה זו מאפשרת לממשק אחד להיות שמיש לצורך פעולה של מחלקה מסוימת.

דוגמא:

ניקח בחשבון "מחסנית" בתוכנית. יכול להיווצר מצב שבו נצטרך 3 מחסניות. המחסנית הראשונה משמשת לערכים מספריים INT, השנייה למספרים עשרוניים FLOAT והשלישית לתווים. האלגוריתם שפועל על כל מחסנית הוא זהה למרות שהנתונים המאוכלסים הנם שונים.

בשפות שאינן מונחות עצמים במקרה שכזה נצטרך לייצור 3 מחסניות נפרדות.



לסיכום

כאשר שלושת התכונות שהוצגו עובדות ביחד בתוכנית הן יוצרות סביבת תכנות שתומכת בהרבה יותר מבנים מורכבים למתן פתרונות מהעולם האמיתי.

2. סוגי נתונים, משתנים ומערכים

2.1. סוג המרה ו Casting

במידה ויש לנו ידע משפות תכנות אחרות אזי אנו יודעים כי ניתן להעביר ערכים ממשתנה אחד למשתנה אחר מאותו סוג. במידה ושני סוגי המשתנים הנם תואמים אזי java תבצע לנו את ההמרה באופן אוטומטי. לדוגמא תמיד אפשרי לתת ערך מסוג int למשתנה מסוג long. אולם לא כול במשתנים הנם אותו הדבר ובמקרה שכזה לדוגמא המרה double למשתנה byte אזי כי נצטרך להשתמש ב Castאשר בעצם מבצע את ההתאמה בין הנתונים.

הבא נבחן את שני סוגי ההמרה.

2.1.1. המרה אוטומטית

ההמרה האוטומטית מתבצעת באופן הפשוט ביותר כאשר שני התנאים הבאים מתקיימים:

1. שני סוגי המשתנים תואמים
2. סוג המטרה הנו גדול יותר מסוג המקור.

המרה על יד שימוש ב Cast

על מנת לייצור התאמה בין שני סוגי משתנים שונים אנו חייבים להשתמש ב cast שזהו בעצם כלי לטיפול ביוצא דופן על מנת לבצע התאמת נתונים.

המבנה הכללי הבא הנו:

(target-type) value

כאשר:

Target type הנו המשתנה שאנו רוצים להמיר לדוגמא int ל byte. במידה וערך ה int הנו גדול יותר מערך ה byte באופן אוטומטי הקומפיילר יקצץ בשאר ה int על מנת להמירו ל byte.

דוגמא:

```
int a;
byte b;
//…..
b = (byte) a
```

דוגמא נוספת

```
// Casting
class Conversion
{
      public static void main (String args[])
      {
            byte a;
            int b = 125;
            double c = 233.456;

            System.out.println("\nConversion of int to byte");
            a = (byte) b;
            System.out.println("b and a" + a + "" + b);

            System.out.println("\nConversion of double to int");
            b = (int) c;
            System.out.println("b and c" + c + "" + b);

            System.out.println("\nConversion of double to byte");
            a = (byte) c;
            System.out.println("c and a" + c + "" + a);
      }
}
```

בנוסף יש לדעת כי במצב של הגדרת byte שהנם גדולים הקומפיילר יבצע קידום אוטומטי של המרה מ byte ל int.

3. מחלקות

בחלק זה נעסוק במחלקות.

תכנית ראשונה

```
//Uses of class called Box

class Box
{
	double width;
	double height;
	double depth;
}

class BoxTest
{
	public static void main (String args[])
	{
		Box mybox = new Box();
		double volume;

		mybox.width = 10;
		mybox.height = 20;
		mybox.depth = 15;

		volume = mybox.width * mybox.height * mybox.depth;

		System.out.println ("Volume is:" + volume);
	}
}
```

4. מבט נרחב על פונקציות ומחלקות

שני נושאים חשובים מבחינת אבטחת מידע הנם אתחול וניקוי. הרבה מאוד באגים נוצרים בשפות כגון C או C++ כאשר אנו כותבים תכנית ושוכחים לאתחל אותה. ובעיקר זה נכון לגבי ספריות אשר אין אנו מכירים אותן.
ניקוי הנה בעיה נקודתית מפאת הסיבה שקל לשכוח לנקות את הקוד כאשר אין אנו עוקבים אחריו. לפיכך המקור של נושאים אלו קרי אתחול וניקוי נשמרים וללא כל בעיה נוכל לגרום לבעיה של משאבי מערכת.
בשפת C++ הוצג הקונספט של קונסטרקטור שהוא בעצם פונקציה הנקראת אוטומטית כאשר אובייקט נוצר. בשפת Java אומץ הרעיון הזה ובנוסף לזה הוגדר גם אוסף ה"זבל" אשר אוטומטית משחרר משאבי מערכת כאשר אינם בשימוש יותר. בחלק זה נבחן את הנושא.
אתחול מאובטח על ידי שימוש בקונסטרקטורים.
אנו יכולים לדמיין לעצמנו לבנות פונקציה שתקרא initialize() לכל מחלקה שנבנה. שם הפונקציה הנו המפעיל את הפונקציה. לרוע המזל שיטה שכזו אומרת לנו כי אנו המשתמשים נצטרך לזכור את כלל הפונקציות שלנו בקוד.
לפיכך Java נותנת לנו כלי לאבטח את אותו אתחול על ידי שימוש בקונסטרקטור. במידה ויש למחלקה קונסטרקטור אזי הקריאה לקונסטרקטור מתבצעת באופן אוטומטי כאשר אובייקט נוצר. לפיכך נוכל לומר כי האתחול מאובטח.
השאלה הבאה הנה מה השם שניתן לפונקציה על מנת שנקרא לה. קיימים שני נושאים לדיון, האחד לשמור על אחדות של שמות ז"א יכול להיווצר מצב בו נקרא לפונקציה באותו שם וללא שימת לב נגדיר עוד פונקציה עם אותו שם. הנושא השני הוא מכיוון שהקומפיילר אחראי לקרוא לקונסטרקטור אזי הוא חייב לדעת לאיזו פונקציה לקרוא.
הפתרון המוצע בשפת C++ נראה הלוגי והנכון ביותר לעשות לפיכך אנו משתמשים בו גם ב Java .
הפתרון הנו ששם הקונסטרקטור זהה לשם המחלקה שאליה הוא שייך.

דוגמא

```java
// Demonstration of a simple constructor.

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
```

נשים לב כי כאשר נוצר אובייקט חדש בשם
new Rock();

האחסון ממוקם והקונסטרקטור נקרא ולפיכך מובטח לנו כי האובייקט יאותחל. שים לב כי שם הקונסטרקטור זהה לשם המחלקה!

כמו לכל פונקציה לקונסטרקטור ישנם ארגומנטים העוזרים לנו להגדיר כיצד האובייקט ייווצר. אם ניקח את הדוגמא לעיל הרי שנוכל להרחיבה ולומר כי הקונסטרקטור הבא הנו כללי ז"א מחזיק ערכים כללים שנוכל להשתמש בהם.

דוגמא

```java
//Constructors can have arguments.

class Rock2 {
  Rock2(int i) {
    System.out.println("Creating Rock number " + i);
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
}
```

שימוש בארגומנטים של קונסטרקטורים מאפשרים לנו לספק פרמטרים אשר נרצה לאתחל באוביקט מסוים.

לדוגמא אם למחלקה בשם Tree ישנו קונסטרקטור המכיל ערך int מספרי בודד נוכל לייצור אוביקט בצורה הבא:

```java
Tree t = new Tree(12);  // 12-foot tree
```

ובמידה ו Tree (int) הוא הקונסטרקטור היחידי אזי הקומפיילר לא יאפשר לנו לייצור אובייקט Tree אחר.

הקונסטרקטורים עוזרים לנו לייצור תכניות מובנות ולמנוע בעיות באגים. לדוגמא לא יצרנו שום פונקציה הנקראת initialize() שהיא בודדת מהגדרות המחלקה. ב Java ההגדרה והאתחול הנם שני קונספטים מאוחדים אין אפשרות שרק אחד יהיה בתכנית.

הקונסטרקטור הנו מבנה ייחודי של פונקציה מכיוון שאין לו כלל החזרת ערך אפילו לא Void. אם הייתה ביכולתנו האפשרות להחזיר ערכים אזי נוכל לומר כי היינו צריכים להגדיר לקומפיילר היכן אותם ערכים ובכך בעצם לא פתרנו בעיה.

5. טעינת פונקציות

אחד מהתכונות העיקריות בכל שפת תכנות הנה אפשרות שימוש בשמות. כאשר אנו יוצרים אובייקט אנו נותנים לו שם לאחסון ז"א בשם הזה האובייקט יתאכסן בזיכרון. פונקציה הנה שם לפעולה שכזו.
ובהחלט בהמשך התכנית נתייחס לכל האובייקטים והפונקציות על ידי מתן שמות. כאשר נשים לב כי שמות הגיוניים נותנים לנו הבנה טובה יותר שדל התכנית.
ב Java קיימת לנו האפשרות להגדיר פונקציה אחת או יותר באותה מחלקה החולקת את אותו שם מחלקה וכל עוד אותם פרמטרים הם שונים בפונקציות. כאשר זה המקרה נוכל לומר כי הפונקציות הנם נטענות Overloaded והתהליך הנו method overloaded.
בשפת Java (כמו גם CPP) ישנו פקטור המכריח את ביצוע טעינת השמות Method overloaded להתבצע והוא הקונסטרקטור.
מכיוון ששם הקונסטרקטור נקבע כבר על ידי שם המחלקה אזי נוכל לומר כי נוכל לייצור רק קונסטרקטור אחד. אבל מה יקרה באם נרצה לייצור אובייקט במספר דרכים?
לדוגמא, נניח שאנו בונים מחלקה שיכולה לאתחל את עצמה באופן סטנדרטי של קריאה מקובץ. אזי נצטרך 2 קונסטרקטורים האחד לצורך אי לקיחת ארגומנטים Default והאחר הלוקח String כארגומנט אשר הוא בעצם שם הקובץ אליו אנחנו מתייחסים.
נשים לב כי שניהם קונסטרקטורים ושניהם חייבים להכיל את אותו שם שהוא שם המחלקה. לפיכך נשתמש ב method overloading על מנת לתת פתרון לנושא שכזה.

דוגמא

```
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;
class Tree {
  int height;
  Tree() {// with no arguments
    System.out.println("Planting a seedling");
    height = 0;
  }
  Tree(int i) {// with arguments
System.out.println ("Creating new Tree that is "+ i + "
feet tall");
    height = i;
  }
  void info()
System.out.println ("Tree is " + height + " feet tall");
  }
  void info(String s) {
System.out.println (s + ": Tree is " + height + " feet
tall");
  }
  static void prt(String s) {
System.out.println(s);
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
}
```

הסבר

האובייקט Tree יכול להיווצר בקונסטרקטורים על ידי אחת משני האפשרויות, ללא ארגומנטים או עם ארגומנטים. על מנת לתמוך במקרה שכזה הגדרנו שני קונסטרקטורים האחד שלא לוקח ארגומנטים והרי הוא ה default והאחר הוא עם ארגומנטים. לאחר מכן שוב הגדרנו שני פונקציות info() נשים לב כי על ידי שימוש בטעינת פונקציות אנו משתמשים באותו שם של פונקציות שדבר זה מותר לביצוע.

5.1. הבדל בין טעינת פונקציות

במידה ולפונקציה ישנו את אותו שם אזי כיצד הקומפיילר מבדיל בינהם? החוקיות הנה פשוטה, כל פונקציה שהנה נטענת שוב חיבת לקבל את הארגומנטים הייחודיים שלה. אם נחשוב על זה אזי נוכל לומר כי הפתרון הוא לוגי כי הרי כיצד נוכל להבדיל בין שני פונקציות בעלות אותו שם מלבד סוג הארגומנט?
גם הבדל בין הארגומנטים מספיק לנו על מנת להבדיל בין שני פונקציות בעלות אותו שם.

דוגמא

```java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  static void print(String s, int i) {
  System.out.println("String: " + s + ", int:" + i);
  }
  static void print(int i, String s) {
  System.out.println("int: " + i +", String: " + s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
  }
}
```

הסבר

שני פונקציות ההדפסה print() מכילות ארגומנטים זהים אבל צורת הסדר היא שונה ולכן זה מה שעושה את ההבדל לקומפיילר.

5.2. טעינת פונקציות עם נתונים פרימיטיביים

משתנה פרימיטיבי יכול להיות מקודם באופן אוטומטי מקטן לערך גדול ותכונה זו יכולה להיות מבלבלת כאשר אנו עוסקים בטעינת פונקציות. בדוגמא הבאה נראה כיצד מועברים ערכים פרימיטיביים לפונקציה נטענת.

דוגמא

```
// Promotion of primitives and overloading.

public class PrimitiveOverloading
{
      // boolean can't be automatically converted
      static void prt(String s)
      {
            System.out.println(s);
      }

      void f1(char x) { System.out.println("f1(char)"); }
      void f1(byte x) { System.out.println("f1(byte)"); }
      void f1(short x) { System.out.println("f1(short)"); }
      void f1(int x) { System.out.println("f1(int)"); }
      void f1(long x) { System.out.println("f1(long)"); }
      void f1(float x) { System.out.println("f1(float)"); }
      void f1(double x) { System.out.println("f1(double)"); }

      void f2(byte x) { System.out.println("f2(byte)"); }
      void f2(short x) { System.out.println("f2(short)"); }
      void f2(int x) { System.out.println("f2(int)"); }
      void f2(long x) { System.out.println("f2(long)"); }
      void f2(float x) { System.out.println("f2(float)"); }
      void f2(double x) { System.out.println("f2(double)"); }
      void f3(short x) { System.out.println("f3(short)"); }
      void f3(int x) { System.out.println("f3(int)"); }
      void f3(long x) { prt("f3(long)"); }
      void f3(float x) { System.out.println("f3(float)"); }
      void f3(double x) { System.out.println("f3(double)"); }
      void f4(int x) { System.out.println("f4(int)"); }
      void f4(long x) { System.out.println("f4(long)"); }
      void f4(float x) { System.out.println("f4(float)"); }
      void f4(double x) { System.out.println("f4(double)"); }
      void f5(long x) { System.out.println("f5(long)"); }
      void f5(float x) { System.out.println("f5(float)"); }
      void f5(double x) { System.out.println("f5(double)"); }
      void f6(float x) { System.out.println("f6(float)"); }
      void f6(double x) { System.out.println("f6(double)"); }
      void f7(double x) { System.out.println("f7(double)"); }
      void testConstVal()
      {
            System.out.println("Testing with 5");
            f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
      }
      void testChar()
      {
            char x = 'x';
            System.out.println("char argument:");
            f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
      }
      void testByte()
      {
            byte x = 0;
            System.out.println("byte argument:");
```

```
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testShort()
    {
        short x = 0;
        System.out.println("short argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testInt()
    {
        int x = 0;
        System.out.println("int argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testLong()
    {
        long x = 0;
        System.out.println("long argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testFloat()
    {
        float x = 0;
        System.out.println("float argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    void testDouble()
    {
        double x = 0;
        System.out.println("double argument:");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    }
    public static void main(String[] args)
    {
        PrimitiveOverloading p =
            new PrimitiveOverloading();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
        p.testInt();
        p.testLong();
        p.testFloat();
        p.testDouble();
    }
}
```

הסבר

אם נסתכל תוצאת התכנית אזי נראה כי הקבוע שערכו 5 מיוחס כ int לפיכך הפונקציה שמשמשת אותנו לקחת פרמטרים הנה מסוג נתונים זה.
בכל שאר המקרים באם קיימים לנו ערכים אשר קטנים יותר מערכי הפונקציה הנטענת אזי כי הקומפיילר יקדם את אותם ערכים באופן עצמאי.

אם כן נשאלת השאלה מה יקרה באם הארגומנט גדול יותר מהצפוי להתקבל על ידי הפונקציה. התשובה ניתנת להלן:

```
	// Demotion of primitives and overloading.

public class Demotion
{
	static void prt(String s)
	{
		System.out.println(s);
	}

void f1(char x) { System.out.println("f1(char)"); }
void f1(byte x) { System.out.println("f1(byte)"); }
void f1(short x) { System.out.println("f1(short)"); }
void f1(int x) { System.out.println("f1(int)"); }
void f1(long x) { System.out.println("f1(long)"); }
void f1(float x) { System.out.println("f1(float)"); }
void f1(double x) { System.out.println("f1(double)"); }

void f2(char x) { System.out.println("f2(char)"); }
void f2(byte x) { System.out.println("f2(byte)"); }
void f2(short x) { System.out.println("f2(short)"); }
void f2(int x) { System.out.println("f2(int)"); }
void f2(long x) { System.out.println("f2(long)"); }
void f2(float x) { System.out.println("f2(float)"); }

void f3(char x) { System.out.println("f3(char)"); }
void f3(byte x) { System.out.println("f3(byte)"); }
void f3(short x) { System.out.println("f3(short)"); }
void f3(int x) { System.out.println("f3(int)"); }
void f3(long x) { System.out.println("f3(long)"); }

void f4(char x) { System.out.println("f4(char)"); }
void f4(byte x) { System.out.println("f4(byte)"); }
void f4(short x) { System.out.println("f4(short)"); }
void f4(int x) { System.out.println("f4(int)"); }

void f5(char x) { System.out.println("f5(char)"); }
void f5(byte x) { System.out.println("f5(byte)"); }
void f5(short x) { System.out.println("f5(short)"); }

void f6(char x) { System.out.println("f6(char)"); }
void f6(byte x) { System.out.println("f6(byte)"); }

void f7(char x) { System.out.println("f7(char)"); }

void testDouble()
	{
		double x = 0;
	System.out.println("double argument:");
		f1(x);f2((float)x);f3((long)x);f4((int)x);
		f5((short)x);f6((byte)x);f7((char)x);
	}
	public static void main(String[] args)
	{
		Demotion p = new Demotion();
		p.testDouble();
	}
	}
```

הסבר

בתרגיל זה הפונקציות לוקחות ערכים פרימיטיביים. במקרה שהארגומנט ארוך יותר נצטרך לבצע התאמה של נתונים.

5.3. טעינה של ערכים מוחזרים

נהוג לתהות מדוע שמות מחלקה וארגומנטים של פונקציות הנם ברשימה? מדוע איננו מפרידים בין פונקציות המכילות ערכים מוחזרים?
לדוגמא שתי הפונקציות להלן נפרדות בהגדרתן זו מזו:

```
void f() {}
int f() {}
```

פונקציות אלו עובדות כראוי אם הקומפיילר מזהה איזה ערך קיים לפונקציה בוודאות. אולם אפשר לקרוא לפונקציה ולהתעלם מערכה המוחזר. תכונה זו בדרך כלל מודגרת כקריאה לפונקציה על ידי מתן אפקט צדדי, מאחר שלא מעניין אותנו הערך המוחזר אלא אנו מעוניינים בקריאה לפונקציה בלבד. ולכן אם נקרא לפונקציה בצורה הבאה:

```
f();
```

אזי כי אז כיצד נוכל לדעת איך הקומפיילר קורא לפונקציה f()? וכיצד אנו נבחין בזה?
בגלל הבעיה הזו איננו יכולים להשתמש בערכים מוחזרים על מנת להבדיל בין פונקציות נטענות.

5.4. קונסטרקטור default

כפי שכבר הוזכר קונסטרקטור שהנו default איננו מכיל ערכים כלל והוא בעצם עוזר לנו לייצור אובייקט. במידה וניצור מחלקה אשר לה לא קיים קונסטרקטור, הקומפיילר ייצור לנו את default constructor.

דוגמא

```java
//DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird nc = new Bird(); // default!
  }
}
```

הסבר

השורה ;()new Bird יוצרת אובייקט חדש אשר משתמש ב default constructor למרות שלא רשמנו אחד שכזה. במידה ונרשום קונסטרקטור אזי הקומפיילר לא ייתן לנו אחד נוסף.

לצורך הדוגמא במידה ורשמנו קונסטרקטור אזי הקומפיילר ישתמש בקונסטרקטור הזה:

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

5.5. מילת הייחוס this

במקרה בו יש לנו שני אובייקטים מאותו סוג הנקראים a ו b נוכל לתהות כיצד נוכל לקרוא לאותה פונקציה תוך שימוש באובייקטים שונים?

דוגמא

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

ובמקרה שבו ישנה רק פונקציה אחת f() כיצד נוכל להבדיל בקריאה לאובייקטים?
על מנת לתת מענה לבעיה זו הרי שנוכל להשתמש במילת הייחוס this. אנו יכולים להשתמש במילת ייחוס זו אך ורק בתוך פונקציה אחת או יותר וללא הגבלה של מספר הפעמים בהם נשתמש. שימוש במילה זו מייצר התייחסות לאובייקט שלו הפונקציה קוראת. נוכל להתייחס לייחוס של המילה הזו כמו כל התייחסות אחרת לאובייקט. יש לזכור כי אם נקרא לפונקציה מתוך המחלקה הקיימת אזי כי לא נצטרך להשתמש במילת הייחוס הזו.
לפיכך נוכל לומר כי:

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

בתוך פונקצית pit() נוכל להגדיר את this.pit() אבל אין צורך בדוגמא זו. הקומפיילר מבצע זאת אוטומטית.

שימוש במילת הייחוס this יתבצע רק כאשר נרצה להשתמש בהתייחסות נקודתית לאותו אובייקט.

דוגמא

```
// Simple use of the "this" keyword.

public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
}
```

הסבר

בגלל שפונקצית ()increment מחזירה את ההתייחסות לאובייקט הנוכחי דרך שימוש במילת המפתח this .

5.6 קריאה מקונסטרקטור לקונסטרקטור

כאשר אנו כותבים מספר קונסטרקטורים קיימים מצבים בהם נרצה לקרוא מקונסטרקטור אחד למשנהו על ידי מניעה של שכפול קוד. נוכל לבצע זאת שוב בעזרת מילת הייחוס this.
באופן נורמלי כאשר נשתמש במילה זו נוצרת כפי שאמרנו התייחסות לאובייקט. בשימוש בקונסטרקטורים השימוש במילת הייחוס this מבצעת פעולה שונה כאשר אנו מתייחסים אליה. הפעולה הנה יצירת ייחודיות קריאה לקונסטרקטור המתאים לרשימת הארגומנטים. לפכיך נוכל לומר כי קיימת דרך קריאה ישירה לקונסטרקטורים.

דוגמא

```
// Calling constructors with "this."

public class Flower {
  int petalCount = 0;
  String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
System.out.println("Constructor w/ int arg only,
petalCount= " + petalCount);
  }
  Flower(String ss) {
System.out.println("Constructor w/ String arg only, s=" +
ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println(
      "default constructor (no args)");
  }
  void print() {
//!    this(11); // Not inside non-constructor!
System.out.println("petalCount = " + petalCount + " s =
"+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
}
```

הסבר

הקונסטרקטור Flower(String s, int petals) מראה כי בזמן שקראנו לקונסטרקטור אחד על ידי שימוש ב this לא נוכל לקרוא לעוד קונסטרקטור באותו זמן. בנוסף הקריאה לקונסטרקטור צריכה להתבצע ראשונה אחרת תהיה טעות קומפילציה.
בנוסף הדוגמא מראה לנו שימוש נוסף ב this. מאחר ששם הארגומנט הנו s ושם ה members של הנתונים גם s יכול להיות בלבול מסוים.
על ידי שימוש במילת this.s נוכל להתייחס ל member data.

בפונקצית ()print נוכל לראות כי לא נוכל לקרוא לקונסטרקטור מתוך הפונקציה.

5.7. הבנת Static

לאחר שהבנו מה היא מילת השיוך this נוכל להבין את משמעות השימוש במילת הייחוס static כאשר היא נמצאת בפונקציה. ז"א לא קיימת מילת this לאותה פונקציה המוגדרת כ static.
לא נוכל לקרוא לפונקציה שאיננה static מתוך פונקציה שהיא static אבל נוכל לקרוא לפונקציה המכילה static למחלקה עצמה ללא שום אובייקט. למעשה זהו השימוש במילת static. למעשה זהו אותו תהליך כמו בשפת C ששם בעצם הגדרנו פונקציה גלובאלית. ב Java לא נוכל להגדיר פונקציה גלובלית כמו ב C אבל נוכל להגדיר מילת static בפונקציה בתוך מחלקה המאפשרת גישה לפונקציות static ולשדות static.

5.8. איסוף "זבל"

בשפת java אין אנו צריכים לדאוג לנקות את הזיכרון מאובייקטים שאינם שימושיים. אם נזכור כיצד אובייקט מתנהג אזי כי נוכל לומר שברגע שלא השתמשנו באופרטור new על מנת להגדיר את האובייקט מבחינת הקומפיילר אובייקט זה ישחרר את מקומו בזיכרון.
ולכן ניתן לומר כי מאסף הזבל עובד בהתאם לאובייקטים שהוגדרו על ידי האופרטור. אם כן נשאלת השאלה מה קורה במצב בו מוגדר אובייקט בצורה "מיוחדת"?
על מנת לטפל במצב זה אנו נשתמש בפונקצית ()finalize אשר תוגדר במחלקה. אופי העבודה הנו פשוט, כאשר מאסף הזבל מוכן לעבודה הוא בראש ובראשונה ניגש לפונקצית ()finalize ורק לאחר מכן הוא ממשיך לשחרר את אותם אובייקטים שמוגדרים כ new.
בהשוואה לשפת ++c בד"כ פונקצית ()finalize הנה שימושית ל- destructor אשר הנה פונקציה שתמיד נקראת כאשר אנו הורסים אובייקט מסוים.
אבל חשוב לשים לב לדקות של שתי השפות ב ++c אובייקט תמיד נהרס בשפת java אובייקטים לא תמיד נאספים לזבל.
חשוב לזכור כי איסוף הזבל הנה תכונה העוזרת לנו לשחרר מקום בזיכרון ולכן חשוב לעקוב אחרי אובייקטים שלא יוגדרו ושלא ייאספו על יד מאסף הזבל.
בעזרת שימוש בפונקציה זו נוכל להדמות את התוכניות שלנו ל C על ידי טכניקה של שימוש בפונקציה ז"א על מנת לקרוא לתכניות אחרות שאינן כתובות ב java נעשה זאת בעזרת native methods.
שפות C ן ++C הנן השפות היחדיות התומכות ב native methods אבל מאחר שנוכל לקרוא לתכניות אחרות תוך שימוש בפונקציות אלו אזי כי כמובן נוכל להשתמש בהם ב java.
לדגומא בשפת C אנו ממקמים מקום בזיכרון תוך שימוש בפונקצית ()malloc ואם לא נקרא לפונקציית ()free המיקום שהוכרז יישאר כמות שהוא.
בשפות C ו ++C אנו חייבים לנקות את מקום הזכרון על ידי שימוש בפקודת delete. יצירת האובייקט בשפת ++C נעשית באותה צורה בה היא נעשית ב java על ידי הגדרת אופרטור new.
בניגוד לשפת ++C שפת java איננה מאפשרת לנו לייצור אובייקטים לואליים מהסיבה שנצטרך לעקוב אחריהם על מנת שיימחקו אלא ב java לא קיימת פקודת delete ופעולה זו מתבצעת על ידי מאסף הזבל.
אחת מהתכונות הבולטות של שימוש בפונקצית ()finalize הנה מעקב אחרי איסוף הזבל.

דוגמא

```
// Demonstration of the garbage
// collector and finalization

class Chair
{
	static boolean gcrun = false;
	static boolean f = false;
	static int created = 0;
	static int finalized = 0;
	int i;
	Chair()
	{
		i = ++created;
		if(created == 47)
			System.out.println("Created 47");
	}
	public void finalize()
	{
		if(!gcrun)
		{
			// The first time finalize() is called:
			gcrun = true;
			System.out.println(
				"Beginning to finalize after " +
				created + " Chairs have been created");
		}
		if(i == 47)
		{
			System.out.println(
				"Finalizing Chair #47, " +
				"Setting flag to stop Chair creation");
			f = true;
		}
		finalized++;
		if(finalized >= created)
			System.out.println(
				"All " + finalized + " finalized");
	}
}

public class Garbage
{
	public static void main(String[] args)
	{
		// As long as the flag hasn't been set,
		// make Chairs and Strings:
		while(!Chair.f)
		{
			new Chair();
			new String("To take up space");
		}
		System.out.println(
			"After all Chairs have been created:\n" +
			"total created = " + Chair.created +
			", total finalized = " + Chair.finalized);
		// Optional arguments force garbage
		// collection & finalization:
```

```
            if(args.length > 0)
            {
                  if(args[0].equals("gc") ||
                        args[0].equals("all"))
                  {
                        System.out.println("gc():");
                        System.gc();
                  }
                  if(args[0].equals("finalize") ||
                        args[0].equals("all"))
                  {
                        System.out.println("runFinalization():");
                        System.runFinalization();
                  }
            }
            System.out.println("bye!");
      }
}
```

הסבר התכנית

בתכנית זו אנו יוצרים אובייקטים רבים מסוג כיסא וממשיכים לייצור אובייקטים כאלו גם אחרי שמאסף הזבל מתחיל לרוץ. מאחר שמאסף הזבל רץ בכל זמן איננו יודעים היכן ומתי הוא יתחיל לרוץ ולפיכך יצרנו flag בשם garbageCollection על מנת שייתן לנו אינדיקציה. ה flag השני שנוצר הוא בעצם אומר לפונקצית ה main() להפסיק לייצור אובייקטים ברגע מסוים. שני ה flags קיימים בתוך פונקצית finalize() ובכך אנו מבטיחים לעצמנו שהתכנית תתבצע כראוי.

בנוסף מוגדרים שני משתנים נוספים מסוג static אשר הם נוצרים ומסתיימים תוך מתן אינדיקציה על האובייקטים הנוצרים עד אשר התנאי יתמלא ויתבצע במקרה שלנו 12 כסאות. ברגע שתנאי זה מתבצע אזי כי כלל האובייקטים משתחררים מתוך הזיכרון ומקומם שוב מוכן לשימוש.

נוכל לתהות על השורות הבאות:

```
while(!Chair.f)
            {
                  new Chair();
                  new String("To take up space");
            }
```

כיצד האובייקט נוצר שוב ושוב? ובכן כאשר פונקצית finalize() עובדת וכפי שהיא הוגדרה כאן יש לה סיום ולכן היא "מתעלמת" מההנחיה ב main().

בנוסף אנו יוצרים אובייקט String בכל פעם שהתכנית מבצעת גישה לפונקציה זו, הוספת פונקציה שכזו היא דרך למנוע ממאסף הזבל להיכנס לתוך הפונקציה.

כאשר אנו מרצים את התכנית בקומפיילר של JDK אנו נותנים את שורת הפקודה "gc," או "finalize," או "all.". הארגומנט gc יקרא לפונקצית System.gc( ). על ידי שימוש בארגומנט finalize אנו נכריח את הקומפיילר לקרוא ל System.runFinalization( ) אשר מאופי עבודתו גורם לאובייקטים להסתיים.

תוצאת התכנית תראה בצורה הבאה:

```
Created 12
Beginning to finalize after 3486 Chairs have been created
Finalizing Chair #12, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

בדוגמא זו נראה כיצד אנו מכריחים את הקומפיילר "להרוג" את אותם אובייקטים שאיננו רוצים ואו צריכים.

```
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

class Book
{
	boolean checkedOut = false;
	Book(boolean checkOut)
	{
		checkedOut = checkOut;
	}
	void checkIn()
	{
		checkedOut = false;
	}
	public void finalize()
	{
		if(checkedOut)
			System.out.println("Error: checked out");
	}
}

public class DeathCondition
{
	public static void main(String[] args)
	{
		Book novel = new Book(true);
		// Proper cleanup:
		novel.checkIn();
		// Drop the reference, forget to clean up:
		new Book(true);
		// Force garbage collection & finalization:
		System.gc();
	}
}
```

לסיכום

ראינו ובחנו כיצד מאסף הזבל עובד ב java. נזכור כי מאסף הזבל עובד על JVM.

6. חבילות וממשקים – Packages & Interfaces

נושא זה הנו חשוב מאוד לשימוש בספריות ומאידך גם למתכנת על מנת שיידע כי אין צורך להשתמש בכתיבת קוד כפול לשימוש בספריות. בנוסף יוצר הספרייה צריך לדעת כי שינוים ושדרוגים יכולים להיעשות בספרייה הנוצרת.
נושא זה יכול להיות מושג על ידי שימוש "נוח" Convention. לדוגמא, המתכנת שכותב ספרייה חייב להסכים לא להזיז פונקציות הקימות בספריות מאחר שזה יכול להשפיע על המתכנת. המצב ההפוך הנו קביל ויכול להתבצע.
במקרה של נתונים שהם "חברים" בספרייה, נשאלת השאלה כיצד יוצר הספרייה יכול לדעת איזה נתונים הנם נגישים למתכנת?
מקרה זה נכון גם לגבי פונקציות שהנם "חברות" ביישום המחלקה ולא פונקציות שהמתכנת ייצר. אם כן נשאלת השאלה מה קורה אם יוצר הספרייה רוצה להוציא מחוץ לקוד יישום ישן ולהחליפו בחדש? מכיוון ששינוי שכזה יכול להשפיע על כלל הקוד, לפיכך יוצר הספרייה הנו מוגבל ואינו יכול לייצור את מה שעולה על דעתו.
על מנת לפתור את הבעיה Java מספקת לנו את נתוני הגישה Access Specifiers על מנת לאפשר ליוצר הספרייה לומר מה נגיש למתכנת ומה אינו.
רמות הגישה הנתונות הנן:
Public – הרמה הגבוהה ביותר
Protected – הרמה הידידותית (אשר אין לה מילה שמורה)
Private – פרטי לחלוטין
מהחלק העליון שהוסבר ניתן לחשוב כיוצר ספרייה כי נרצה לשמור את כלל הנתונים שלנו כ Private ככל שניתן ולאפשר רק לפונקציות כלליות שנרצה שמתכנת אחר ישתמש בהם ברמת הקוד. כיוון חשיבה זה הנו מדויק, למרות שלמתכנתים בשפות אחרות כגון C בקרת הגישה איננה אותה בקרה כמו במקרה שלנו.
הקונספט של הספרייה, רכיביה והשליטה עליהם איננה מוחלטת. ישנם עדיין שאלות ונושאים שאינם סגורים מבחינה טכנית לדוגמא כיצד הרכיב הנו במעטפת ביחד עם הספרייה הרצויה.
התשובה הנה שנושא זה מטופל על ידי Package שהנה מילת מפתח, אי לכך נתוני בקרת השליטה מושפעים מכך שהאם המחלקה נמצאת באותה Package או ב Package אחר.
בחלק זה נבחן ונלמד כיצד רכיבי הספרייה מיושמים בחבילות ותוך כדי כמובן נשתמש בבקרת השליטה של התכנית.

6.1 חבילה – יחידת הספרייה - package: the library unit

חבילה הנה מה שנקבל כאשר נשתמש במילת המפתח import לדוגמא

```
import java.util.*;.
```

פקודה זו מביאה את כלל הספרייה ואת רכיביה הנלווים. רכיביה הנילוים קרי, באם נרצה להביא את מחלקת ArrayList אשר נמצאת ב java.util נוכל לבצע זאת על ידי אחת משתי הדרכים, הראשונה לתת את שמה המלא של הספרייה שהנו java.util.ArrayList (את זאת נוכל לבצע בלי שימוש ב import) והאחרת נוכל פשוט לקרוא ל ArrayList תוך שימוש במילת המפתח import.
אם נרצה להביא מחלקה בודדת נוכל לתת את שם המחלקה בתוך הצהרה של import דוגמא

```
import java.util.ArrayList;
```

כעת נוכל להשתמש ב ArrayList ללא שום תלות. אולם אף אחת משאר המחלקות ב java.util תהיה לנו זמינה.
הסיבה לשימוש או להבאה הנה מתן טכניקה פשוטה שיכולה לעזור לנו לנהל את טווח השמות

"name spaces". השמות של כלל המחלקות שאנו יוצרים מבודדות האחת מהשנייה. פונקציה בשם t() במחלקה A לא תהווה ניגוד לפונקציה t() במחלקה B. ומכאן שפונקציות יכולות להיות בעלות אותו שם במחלקות אחרות.

אם כן נשאלת השאלה מה לגבי שמות מחלקה? נניח כי ניצור מחלקה בשם stack המכילה את אחד הרכיבים להרצתה במחלקה אחרת? כאשר אנו יוצרים קוד ב java ומיישמים אותו באינטרנט מצב שכזה יכול לקרות ללא ידיעת המשתמש מאחר שמחלקות מוטענות אוטומטית בתהליך ריצת הקוד.
פוטנציאל זה הגורם לנו להתנגשויות שמיות הנה הסיבה מדוע חשוב לנו לשלוט על בקרת שמות על מנת לייצור ייחודיות בקוד תוך מתן אילוצים.
עד כה למדנו ועסקנו במחלקות נקודתיות קרי לא עסקנו ביבוא מחלקות או חבילות ומכאן נאמר כי השתמשנו ב "default package" אולם אם נרצה לייצור מצב בו ספריות של אפליקציות שונות רצות ממחשב אחד כמובן שנשתמש ב packages.

כאשר אנו יוצרים תכנית ב java היא נקראת "יחידת ההשלמה" או במינוח המקצועי compilation unit.
לכל יחידה שכזו אנו מוסיפים את הסיומת java. על מנת שהקובץ ירוץ בקומפיילר, ובתוך אותה יחידה אנו כוללים את בקרי הגישה לדוגמא יכולה להיות מחלקה שהנה public והיא בעצם נגישה לכלל הקוד לא רק באותו קובץ.
חשוב לזכור כי יכולה להיות רק מחלקה public אחת בלבד בכל יחידה שכזו. שאר המחלקות המוגדרות ביחידה הנם "מוחבאות" משאר העולם החיצוני מכיוון שאינן public והן מהוות תמיכה במחלקה הראשית שהנה public.
כאשר אנו נריץ קובץ java. נקבל את אותו תוצר בדיוק אבל עם סיומת class. לכל מחלקה בתוך יחידת הקובץ של java. לפיכך נוכל לומר כי לאחר הרצה של תכנית מסוימת תתקבל לנו יצירה של סיומות רבות של מחלקות.
במקרה של שימוש בקומפיילרים של שפות אחרות אזי נקבל גם סיומת obj. הקומפיילר בעצם מאחסן לנו את החבילות ומחלק אותם לבני מינם בהתאמה.
שיטת העבודה ב java הנה שונה. תכנית עבודה האמורה לרוץ הנה מכלול של קבצי class. אשר יכולות לבצע דחיסה/פריסה לתוך קובץ מסוג JAR (תוך שימוש ב Java's **jar** archiver).
האינטרפרטור של java הוא זה המאפשר לנו למצוא, לטעון, ולהריץ את הקבצים הללו.

בנוסף ספרייה הנה מכלול של קבצי class. לכל קובץ כפי שהוזכר יש רק מחלקה אחת שהיא public (אין הכרח שתהיה לנו מחלקה public אך שימוש זה מאוד נפוץ) ולפיכך ישנו רכיב אחד לכל קובץ. אם נרצה לומר כי כלל הרכיבים הכלולים ב java. וב class. משויכים יחדיו אזי כי נשתמש ב package. כאשר נצהיר כי:

```java
package mypackage;
```

בתחילת הקובץ אנו בעצם מצהירים כי "יחידת ההשלמה" קרי אותם קבצים שאנו אמורים לכלול על מנת להריץ את הקובץ נמצאים בחבילה בשם mypackage. נשים לב כי ההצהרה על שימוש ב package חייב להופיע בשורת הקוד הראשונה.
או במילים אחרות נוכל לומר כי המחלקה שהנה public הנמצאת בתוך החבילה mypackage ולאותה מחלקה ישנו שם כך שאם משהו ירצה להשתמש באותו שם של אותה המחלקה ולא לייצור קונפליקט במערכת אזי כי נשתמש במילת המפתח import תוך שילוב עם mypackae.
לדוגמא, נניח כי שם הקובץ שלנו הנו MyTest.java. שם זה אומר כי יכולה להיות אך ורק מחלקה public אחת בקובץ הזה ושם המחלקה הזו חייב להיות MyTest

```java
package mypackage;
public class MyTest {
```

ולצורך הדיון, אם משהו ירצה להשתמש בשם MyTest ולצורך העניין במחלקה public מתוך mypackage אזי כי נשתמש במילת השיוך import.
לדוגמא

```java
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

6.2. יצירת חבילות ייחודיות – Creating Unique Package Name

מאחר שחבילות אינן נדחסות לתוך קובץ בודד חבילה יכולה להיות מורכבת ממספר קבצי class. ולפיכך דברים יכולים להיעשות מורכבים יותר. על מנת למנוע זאת ישנו פתרון לוגי והוא לשים את קבצי class. המשויכים לחבילה מסוימת בתוך ספרייה בודדה ז"א נשתמש בהיראכיה של קבצים הקיימת במערכת ההפעלה. קיימת דרך נוספת והיא תוך שימוש ב jar .

אחסון החבילה תוך ספרייה בודדת אחת פותר לנו עוד שתי בעיות:

1. יצירת שם ייחודי לחבילה.
2. ומציאת אותן מחלקות שאינן שימושיות לאותו מקום.

דבר זה ניתן לביצוע על ידי מתן המסלול של קבצי ה class. לתוך מסלול ה package. הקומפיילר מאפשר לנו לבצע זאת אך החלק הראשון של ה package הנו בעצם שם דומיין האינטרנט של יוצר המחלקה הנשמרת. מאחר ששמות דומיין באינטרנט הנם ייחודיים באם נשתמש באותה צורה אזי כי מובטח לנו שה package תהיה ייחודית ולפיכך אף פעם לא יהיו לנו התנגשויות שמיות.
כמובן שאם אין לנו את הדומיין שלנו אזי נצטרך לפברק קומבינציה של יצירת שמות ייחודיים ל packages. אך אם נרצה להריך קוד java בבטחה אזי חשוב מאוד כי יהיה לנו דומיין משלנו.
החלק השני של פתרון הבעיה הוא פתרון שם ה package בתוך הספרייה במחשב שלך כך שכאשר תוכניות java ירוצו ויצטרכו לטעון את קבצי class. מיקום הספרייה יתבצע באופן דינמי.
האינטרפרטור של java מבצע את הפעולות הבאות:

1. מציאת משתנה סביבה על ידי שימוש ב CLASSPATH.(בד"כ ניתן לנו ע"י מערכת ההפעלה או הקומפיילר). CLASSPATH מכיל ספרייה אחת או יותר המשמשות כשורשים לחיפוש קבצי class.
2. בתחילת השורש האינטרפרטור לוקח את שם ה package ומחליף כל נקודה (.) בסימן קו נטוי (/) וכך הוא יוצר את המסלול מתוך שורש ה- CLASSPATH (דוגמא package my.name.test הופך להיות package my/name/test או package my\name\test או משהו אחר תלוי במערכת ההפעלה שלנו).
3. השלב הבא הנו שרשור הכניסות הקיימות ב CLASSPATH. בשלב זה CLASSPATH מחפש את קבצי class. על מנת לבצע התאמה עם השם שאנו יצרנו בכדי שהתכנית תרוץ

על מנת להבין את התהליך הבא נבחן את הדוגמא הבאה:
ניקח דומיין בשם iwatch.com . על ידי היפוך com.iwatch אנו מייצרים את השם הגלובלי הייחודי למחלקות שלנו. כלל הסיומות כגון com, edu, org וכו' קיימות בספריות java . נוכל לחלק זאת עוד ולהחליט כי אנו רוצים לייצור ספרייה בשם simple ולפיכך נסיים בכתיבת הקוד הבא:

```
package com.iwatch.simple;
```

כעת ה package הזה יכול להכיל תת קבצים, לדוגמא:

```
// Creating a package.
package com.iwatch.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.iwatch.util.Vector");
  }
}
```

כאשר אנו יוצאים את החבילות שלנו כפי שהוזכר ההצרה על חבילה חייב להיות בראש הקוד,לדוגמא:

```java
// Creating a package.
package com.iwatch.simple;

public class ListMe {
  public ListMe() {
    System.out.println(
      "com.iwatch.util.ListMe");
  }
}
```

נשים לב כי שתי הדוגמאות לעיל צריכות להיות מאוחסנות בתת ספרייה במערכת ההפעלה שלנו, לדוגמא:

```
C:\DOC\JavaT\com\iwatch\simple
```

כפי שהוזכר יצרנו חבילה בשם com.iwatch.simple אך נשאלת השאלה מה לגבי החלק הראשון של המסלול?
התשובה הנה שבחלק הראשון מטפלים לנו משתני ה CLASSPATH אשר קיימים על המחשב שלנו.

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

נוכל לראות כי CLASSPATH מכיל מספר אלטרנטיבות לחיפוש המסלול הנכון.

אולם כאשר נשתמש בקבצי jar אזי כי נצטרך לשים את שם קובך ה jar בתוך ה CLASSPATH.
לדוגמא לקובץ jar הקרוי apple.jar המסלול ייראה כך:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\apple.jar
```

ברגע שה CLASSPATH מיושם כמו שצריך הקובץ הבא יכול להיות מאוחסן בכל ספרייה:

```java
// Uses the library.
import com.iwatch.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    ListMe l = new ListMe();
  }
}
```

כאשר הקומפיילר נתקל בהצהרה של import הוא מתחיל את חיפושו בספריות המוגדרות על ידי CLASSPATH המחפש תת ספריות ב com\iwatch\simple ואז מחפש את הקבצים המקומפלים של השמות המתאימים (vector.class and ListMe.class) נשים לב כי לשתי המחלקות הפונקציות הנם public.

יש לשים לב כי ברגע שנתקין את הקומפיילר אזי כי CLASSPATH מותקן אתו יחדו אך יש לזכור כי בהתאם למחשב המותקן נצטרך לבצע התאמות של המסלולים.

התנגשות – Collisions
נשאלת השאלה מה קורה אם שתי ספריות מבצעות import תוך שימוש ב *. והן מכילות את אותו שם? לדוגמא:

```java
import com.iwatch.simple.*;
import java.util.*;
```

מאחר ש *.java.util מכיל את מחלקת Victory זה יכול לגרום להתנגשות בקוד. אולם, בכל זמן שלא נרשום קוד הגורם להתנגשות אזי הכל יהיה בסדר ולכן יש לשים לב לכתיבת הקוד הנכונה.
בדוגמא שלנו ההתנגשות יכולה לקרות ברגע שנכתוב את השורה הבאה:

```java
Vector v = new Vector();
```

לאיזה Victory הקומפיילר יתייחס? ובכן התשובה היא שלא נוכל לדעת ולפיכך גם המתכנת איננו יכול לדעת. ולפיכך הקומפיילר יוציא לנו הודעת שגיאה המכריחה אותנו לבצע שינוי. אם ברצוננו להשתמש ב Vector הסטנדרטי של java אזי כי נצטרך לומר:

```java
java.util.Vector v = new java.util.Vector();
```

מאחר שהגדרה זו שייכת ל CLASSPATH והיא מגדירה נקודתית את מיקומו של ה Vector אזי כי לא נצטרך להשתמש בהצהרת *import java.util אלא אם כן אנו משתמשים במשהו אחר מתוך *java.util

6.3. **שימוש בספרייה שאנו יוצרים**

לאחר שראינו כיצד הטכניקה עובדת נוכל להגדיר את הספריות שאנו רוצים על מנת למנוע שכפול קוד.
הבא נבחן את הדוגמא הבאה ובה ניצור חשבון עבור **System.out.println( )** על מנת להוריד קוד הדפסה. חלק זה יהיה מתוך חבילה הנקראת tools:

```java
// The P.rint & P.rintln shorthand.
package com.iwatch.tools;

public class P {
  public static void rint(String s) {
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
}
```

התכנית מראה לנו את האפשרות להדפיס String עם שורה חדשה (**Println( )**) וללא שורה חדשה (**Print( )**).

לאחר שהרצנו את התכנית הקובץ P.class יכול לשמש לנו בכל מקום במערכת על ידי שימוש במילת import , הבא נבחן את הדוגמא הבאה:

```
// Uses the tools library.
import com.iwatch.tools.*;

public class ToolTest {
  public static void main(String[] args) {
 P.rintln("Available from now on!");
 P.rintln("" + 100); // Force it to be a String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
  }
}
```

נשים לב כי את כלל האובייקטים אנו מכריחים להיות מיוצגים על ידי String. במקרה שלנו למעלה אנו מתחילים את הביטוי עם String ריק. אך הבא נשים לב כי ,**System.out.println(100)** עובד בלי שהוא נהפך ל string.
בעזרת תוספת של טעינה אנו נוכל להגיע למצב בו מחלקת P יכולה לבצע את הפעולה הזו.
ומעתה ואילך היכן שאנו מגיעים למצב בו אנו יוצרים כלי חדש העוזר לנו לקצר שורות קוד אזי כי נוכל להוסיף אותו ל package שלנו.

6.4. שימוש ב import על מנת לשנות התנהגות קוד

אחת התכונות החסרות ב java והקיימות דווקא ב C הנה השלמת התנייא המאפשרת לנו לשנות משפט switch על מנת להשיג התנהגות קוד שונה ללא שינוי של קוד אחר. הסיבה שתכונה שכזו איננה קיימת ב java היא מכיוון שתכונה זו הנה שימושית ב C על מנת לפתור בעיות של מערכות הפעלה שוות (בעיה זו איננה קיימת ב Java).
אולם, ישנם שימושים נכונים להתנייא זו. אחד השימושים היותר נפוצים הנו עבור שימוש ב debugging ז"א תהליך זה יכול להיות פעיל תוך כדי כתיבת הקוד.
הבא נבחן את הקוד הבא העוזר לנו לבצע debugging:

```
// Assertion tool for debugging.
package com.bruceeckel.tools.debug;

public class Assert {
  private static void perr(String msg) {
    System.err.println(msg);
  }
  public final static void is_true(boolean exp) {
    if(!exp) perr("Assertion failed");
  }
  public final static void is_false(boolean exp){
    if(exp) perr("Assertion failed");
  }
  public final static void
  is_true(boolean exp, String msg) {
    if(!exp) perr("Assertion failed: " + msg);
  }
  public final static void
  is_false(boolean exp, String msg) {
    if(exp) perr("Assertion failed: " + msg);
  }
}
```

אופי עבודת המחלקה הנו פשוט. המחלקה עובדת בצורה בוליאנית קרי אמת/שקר ולפי כך היא מבצעת את המבדקים. כמובן שישנם את הכלים היותר מתקדמים לטיפול בבעיות של יוצאי דופן.

כאשר נרצה להשתמש במחלקה הזו ולא בטיפול יוצאי הדופן שניתן לנו על ידי java אזי נוכל להוסיף את המחלקה הזו בצורה הבאה:

```
import com.bruceeckel.tools.debug.*;
```

על מנת להריץ את התכולה כך שנוכל לשלוח את הקוד, ניצור מחלקה נוספת בחבילה שונה באופן הבא:

```
// Turning off the assertion output
// so you can ship the program.
package com.bruceeckel.tools;

public class Assert {
  public final static void is_true(boolean exp){}
  public final static void is_false(boolean exp){}
  public final static void
  is_true(boolean exp, String msg) {}
  public final static void
  is_false(boolean exp, String msg) {}
}
```

במצב זה אם נשנה את הצהרת ה import הקודמת ל:

```
import com.bruceeckel.tools.*;
```

התכנית הבאה לעולם לא תדפיס לנו את מה שהמשתמש הכניס:

```
// Demonstrating the assertion tool.
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;

public class TestAssert {
  public static void main(String[] args) {
    Assert.is_true((2 + 2) == 5);
    Assert.is_false((1 + 1) == 2);
    Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
    Assert.is_false((1 + 1) == 2, "1 +1 != 2");
  }
}
```

על ידי שינוי החבילה המיובאת אנו בעצם משנים את הקוד שאמור להתבצע עליו debug.

לסיכום:
חשוב לזכור כי בכל פעם שאנו יוצרים package אנו בעצם מגדירים מבנה של ספריות כאשר אנו נותנים ל package שם. ה package חייבת להיות מוכלת בתוך הספרייה ונצביע עליה בעזרת שמה אשר חייבת להיות ספרייה שבה החיפוש מתבצע דרך CLASSPATH. שימוש התחלתי ב packages יכול להיות מבלבל, לפעמים נקבל טעויות של קישור מסלולים ומחלקות. ברגע שהודעות כאלו מתקבלות ננסה לבדוק את הצהרות ה package ולנסות להריצה בנפרד. ברגע שהיא רצה אזי נדע היכן הבעיה.

6.5. בקרת גישה

שפת java מאפשרת לנו לשלוט בתכניות שלנו ברמות גישה שונות על ידי שימוש בבקרי שליטה. אותם בקרי שליטה הנם public, private, protected. כאשר נשתמש בבקרי השליטה שלנו בתכניות הם יהיו מאוכלסים בתחילת ההגדרה של חבר במחלקה. כאשר כל בקר מאפשר את הגישה אליה הוא הוגדר ז"א בקרי הגישה הנם מוגדרים לטווח פעולתם בלבד. ובהתייחסות לשפת ++C הרי כי שם בקרי השליטה בתכנית שולטת מהיכן שהיא מוגדרת ועד אשר בקר שליטה אחר מחליף אותו.

6.5.1. "ידידותי"

נשאלת השאלה מה קורה באם אנו לא מגדירים בקרת גישה לתכנית כלל? הקומפיילר מספק לנו בקרת גישה אוטומטית הנקראת "ידידותית" friendly.
המשמעות הנה שכלל המחלקות באותה החבילה יוכלו לגשת לאותו חבר המוגדר כ"ידידותי" אבל לכלל המחלקות שמחוץ לחבילה ה"ידידותי" מופיע כפרטי private.
ולפיכך ניתן לומר כי אנו יכולים להתייחס לבקרה ה"ידידותית" כבקרת גישה לחבילה.
בקרת גישה "ידידותי" מאפשרת לנו לאחד מחלקות המשויכות לחבילה באם עצם פעולתם זהה ואו אנו צריכים אותם על מנת להריץ את החבילה.
בהרבה שפות תוכנה הדרך לארגן קבצים יכולה להיות מורכבת וסבוכה, לא זה המקרה ב java . בנוסף ייתכן מצב בו נרצה להוציא מחלקות אשר אין להם גישה למחלקות המוגדרות באותה חבילה, דבר זה מתאפשר.
המחלקה שולטת בקוד הגישה אליה ולחבריה ז"א קוד מחבילות אחרות לא יכול להופיע פתאום ולומר "אני כאן" ולצפות לראות נתונים מוגנים של אותה מחלקה.
הדרך היחידה לאפשר גישה לחבר במחלקה הנה:

1. לאפשר לחבר להיות ציבורי public
2. להשאיר את החבר במחלקה ללא הגדרה של שום בקרת גישה ובכך הקומפיילר ייתן לנו את ה default access
3. כפי שנראה בהורשה, מחלקה מורשת יכולה לגשת לנתונים המוגדרים כמוגנים protected כמו גם לחברים ציבוריים public במחלקה.
4. יצירת פונקציה אשר יכולה לקרוא ולשנות את ערכי בקרת הגישה (בד"כ שימושי מאוד ב OOP ותכונה זו הנה הכרחית כאשר אנו עוסקים ב Java Beans)

6.5.2. בקרת גישה - ציבורי Public

כאשר נשתמש במילת המפתח public אזי כי אנו אומרים לקומפיילר כי כלל החברים במחלקה המוגדרים כך הנם ציבוריים לשימוש בקוד לצורך כלשהו.
נניח כי אנו מגדירים חבילה בשם **dessert המכילה את "יחידת ההשלמה" הבאה:**

```
// Creates a library.
package t05.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
}
```

זכור כי הקובץ Cookie.java חייב להיות מאוחסן פיזית בתת ספרייה הנקראת dessert בספרייה תחת t05 (05 test) החייבת להיות מתחת לאחד מספריות ה CLASSPATH.
יש לזכור כי אנו אחראים לומר לקומפיילר היכן נמצאת החבילה אחרת הוא לא יחפש אותה ותתקבל טעות קומפילציה.

כעת אם ניצור תכנית המשתמש ב Cookie אזי נקבל:

```
// Uses the library.
import t05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
}
```

הסבר תכנית

אנו יכולים לייצור אובייקט בשם Cookie מאחר שהקונסטרקטור הנו public והמחלקה הנה public.
אולם החבר במחלקה (הפונקציה לעיל) bite() איננה נגישה מאחר והיא נמצאת בתוך Dinner.java
מאחר שלא הוגדרה לפונקציה זו שום בקרת גישה אזי כי הקומפיילר סיפק לנו את ה default.

6.5.3. חבילת "ברירת המחדל" - The"default package"

קיימת אופציה בה לא ניצור חבילה והקומפיילר יאפשר לנו לבצע זאת. הבא נבחן את המקרה הבא:

```
// Accesses a class in a
// separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
}
```

בקובץ השני באותה ספרייה נרשום:

```
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
}
```

הסבר התכנית

נשים לב כי Cake יכול לייצור אובייקט מסוג Pie ויכול בנוסף לקרוא לפונקצית f() שלו. הסיבה הנה
מכיוון ששתי הקבצים נמצאים באותה ספרייה ולא קיים להם שם חבילה שונה. ולפיכך כאשר אנו נתקלים
במצב כזה אזי כי מבחינת הקומפיילר הוא מגדיר זאת כ "default package".

6.5.4. בקרת גישה – פרטי Private

כאשר נשתמש במילת המפתח private אזי כי אנו ומרים לקומפיילר כי אף אחד אינו יכול לגשת לחבר הנקודתי הנמצא בתוך מחלקה או פונקציה.
מחלקות אחרות באותה חבילה גם אינן יכולות לגשת לאותם נתונים המוגדרים בפרטיים.
השימוש במילת מפתח זו ניתן להחלטת המתכנת בשימוש בו.

דוגמא

```
// Demonstrates "private" keyword.

class Sunday {
  private Sunday() {}
  static Sunday makeASunday() {
    return new Sunday();
  }
}

public class IceCream {
  public static void main(String[] args) {
//! Sunday x = new Sunday();// will not work
    Sunday x = Sunday.makeASunday();
  }
}
```

הסבר התכנית

אנו רואים שימוש במילת המפתח private. בתכנית זו איננו יכולים לייצור אובייקט חדש בשם Sunday() תוך שימוש בקונסטרקטור שלו מאחר שהוא מוגדר כפרטי. במקום זאת אנו קוראים לפונקציה makeASunday() על מנת לייצור אובייקט חדש במערכת.

בקרת גישה – מוגן Protected
על מנת שנבין מהו protected ניבחן את תהליך ההורשה.
שימוש במילת המפתח protected כחלק מתהליך ההורשה מאפשר לנו לקחת מחלקה קיימת ולהוסיף לה חברים חדשים למחלקה החדשה ובלי שום שינוי במחלקה הקיימת. בנוסף אנו יכולים לשנות את התנהגותו של חבר מסוים במחלקה. על מנת לבצע הורשה ממחלקה קיימת אנו נשתמש במילת המפתח extends ולצורך הדוגמא:

```
class Foo extends Bar {
```

שאר ההגדרות המחלקה הנן זהות.

במידה ואנו ניצור חבילה חדשה וממנה נרצה לבצע הורשה ממחלקה בחבילה אחרת אזי כי החברים היחידים אליהם תהיה לנו גישה הנם אלו המוגדרים כ public בחבילה המקורית. ולפעמים יוצר המחלקה ירצה לקחת חברים מסוימים מהמחלקה ובכך לאפשר גישה רוחבית יותר אבל לא לכלל החברים בתכנית. הבא נבחן את הדוגמא Cookie.java:

```java
// Can't access friendly member
// in another class.
import t05.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
System.out.println("ChocolateChip constructor");
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Can't access bite
  }
}
```

הסבר תכנית

נשים לב כי אחד הדברים המעניינים בהורשה הנו שאם פונקצית bite() הקיימת המחלקה Cookie אזי היא קיימת בכל מחלקה המורשת ממחלקת Cookie.

אבל מאחר שפונקצית bite() מכילה בקרת גישה ידידותית בחבילה אחרת אין אנו יכולים להשתמש בה במקרה הנדון.

כמובן שנוכל לעשות אותה ציבורית ובכך לפתור את הבעיה.

באם נשנה את הקוד וניצור מחלקה ציבורית אזי הקוד ייראה כך:

```java
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

נשים לב כי לפונקצית bite() עדיין קיימת בקרת גישה ידידותית בתוך חבילה בשם dessert אבל היא גם נגישה לכלל המחלקות אשר יבצעו הורשה ממנה.

6.6. בקרת גישה Protected

בקרת גישה זו עוסקת בעיקר בנושא ההורשה כאשר אנו מתייחסים למחלקה קיימת ואנו מוסיפים חברים חדשים למחלקה ללא שינוי למחלקה הקיימת אשר אנו מתייחסים אליה כמחלקת הבסיס.
אנו כמובן יכולים לשנות את ההתנהגות של חברים קיימים במחלקה על מנת לבצע הורשה ממחלקה קיימת.
על מנת לבצע הורשה אנו בפועל אומרים כי אנו מבצעים extends למחלקה קיימת באופן הבא:

```
class Foo extends Bar {
```

אם אנו יוצרים חבילה חדשה אנו יכולים להוריש רק מחבילה אשר לה קיימת בקרת גישה ציבורית. ואם נרצה לגשת לחברים בחבילה זו אזי כי נוכל לגשת בעזרת protected

משתנים ופונקציות המוגדרות כ- protected יכולים להיות נגישים רק מאותה מחלקה אליה הם שייכים יחד עם תתי המחלקות אליהם הם שייכים ובאותם מחלקות שבאותה חבילה.

Situation	**public**	**protected**	**default**	**private**
Accessible to class from same package?	yes	yes	yes	no
Accessible to class from different package?	yes	no, *unless it is a subclass*	no	no

דוגמא מסכמת

```
class AccessControl
{
      int x;//default access
      public int y;//public access
      private int z;//private access
//method for accessing z
      void setz (int i)
      {
            z=i;
      }
            int getz(){
            return z;
      }
}

class AccessControlTest
{
      public static void main (String args[])
      {
            AccessControl object = new AccessControl();
            object.x = 5;
            object.y = 10;
object.setz(50);//we must access z through its Function
            System.out.print("x, y and z:" + object.x + "" + object.y
+ "" + object.getz());
      }
}
```

6.7. הפרדת יישום וממשקים

בקרת גישה בד"כ מיוחסת כ implementation hiding . מעטפת של נתונים ופונקציות בתוך מחלקות תוך שילוב עם implementation hiding הנה ריכוזיות Encapsulation. התוצאה של ריכוזיות הנה נתונים מסוג מסוים עם מאפיינים והתנהגות.
בקרת גישה שמה לנו מחסומים בתוך סוג הנתונים משתי סיבות עיקריות:

1. מתן ההוראות לביצוע קרי מה אנו יכולים ואיננו יכולים לבצע בתכנית. אנו יכולים לבנות את הטכניקה שלנו בתכנית בצורה כזו שלא נצטרך לדאוג שהמתכנת יתייחס לקוד כחלק מהממשק .
2. הפרדת הממשק מהיישום.

באם מבנה התכנית הנו שימושי למספר תוכניות אבל איננו יכולים לבצע דבר מלבד לשלוח פקודות הגובלות בבקרת הגישה קרי public, private, protected .
ולפיכך אנו צריכים לזכור כי אנו עוסקים בתכנות מונחה עצמים בו מחלקה מוגדרת כמכלול אובייקטים.
כל אובייקט הכלול במחלקה מסוימת חולק את אותם מאפייני המחלקה והתנהגותה ובעצם המחלקה הנה ההגדרה לדרך בה כלל האובייקטים מאותו סוג יפעלו בצורה כזו או אחרת.
כאשר הוצגה לראשונה שפת תכנות מונחה עצמים, מילת המפתח class שימשה להצגה של נתונים מסוג חדש. אותה מילת מפתח הנה שימושית לכלל שפות התכנות מונחה העצמים ובעצם זהו העיקר של שפה מונחת עצמים לייצור נתונים מסוג חדש.
אם נרצה לייצור מחלקה אשר מגדירה את חבריה הציבוריים בתחילתה ולאחר מכן את חבריה המוגנים, פרטיים ואו ידידותיים אזי נוכל וכי היתרון של פולה שכזו הנו שהמשתמש יכול להבין את הלך התכנית.
לדוגמא:

```java
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
 }
```

חלק זה מתכנית מסוימת הנו קריא ומובן אך עדיין איננה קיימת ההפרדה המהותית בין הממשק לבין היישום. ז"א אנו עדיין רואים את הקוד של קובץ היעד.
הצגת הנתונים על ידי ממשק הנה בעצם העבודה של class browser שהנו בעצם כלי שעבודתו הנה לחפש את כלל המחלקות הנגישות ולהראות לנו מה נוכל לעשות איתן (ז"א אילו חברים נגישים). אם כך ניתן לסכם ולומר כי דפדפן הנו בעצם הממשק שמריץ את היישום שהוא הקוד.

6.8. גישה למחלקות

בשפת java נוכל להשתמש בבקרי הגישה גם לצורך הגדרות במחלקות קרי איזו מחלקה בתוך הספרייה תהיה נגישה ואיזו איננה כזו.
באם נרצה כי המחלקה תהיה נגישה לכלל אזי כי נגדירה כ public לדוגמא:

```
public class Widget {
```

במקרה הנדון, נניח כי הספרייה שיצרנו הנה בשם mylib אזי כי כל מתכנת יוכל לגשת למחלקה Widget באופן הבא:

```
import mylib.Widget;
```

או באופן הזה

```
import mylib.*;
```

בכל אופן יש לקחת בחשבון את הכללים הבאים:
1. רק מחלקה אחת ביחידת השלמה אחת יכולה להיות public . הרעיון לכך הוא מכיוון שכל יחידת השלמה מיוצגת על ידי מחלקה ציבורית אחת בזמן הקומפילציה. במידה וישנה יותר ממחלקה ציבורית אחת הקומפיילר ייתן לנו טעות.
2. שם המחלקה הציבורית חייב להיות תואם לשם הקובץ שנוצר ביחידת ההשלמה. לדוגמא מחלקה Widget חייבת להיות שמורה בשם Widget.java ולא יכולה להיות שמורה כ widget.java או כ WIDGET.java.
3. זה אפשרי אך לא אופייני לכלול יחידת השלמה ללא מחלקה public. במקרה שכזה שם הקובץ הנו כל שם שנבחר.

יש לשים לב כי מחלקה איננה יכולה להיות private או protected ולפיכך ניתן לומר כי קיימות לנו אך ורק שתי אופציות לבקרת גישה למחלקות:
1. Public
2. ידידותית

במידה ואיננו רוצים לתת גישה למחלקה מסוימת מה שניתן לבצע הוא תהליך של מתן בקרת גישה של private לקונסטרקטורים בתוך חבר שהנו static.

דוגמא

```
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Create a static object and
  // return a reference upon request.
  // (The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}

class Sandwich { // Uses Lunch
  void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void test() {
    // Can't do this! Private constructor:
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
}
```

7. הורשה

הורשה הנה אחת מתכונות ליבת המערכת של שפת תכנות זו מהסיבה שהיא מאפשרת לנו לייצור היראכיה ולסווג. על ידי שימוש בהורשה נוכל לייצור מחלקה כללית המכילה מאפיינים דומים לנתונים שונים. מחלקה זו יכולה להיות מורשת למחלקה אחרת.
בתחביר של java syntax מחלקה המורישה את מאפייניה נקראת Super Class ולפיכך תת המחלקה הנה מקרה נקודתי של מחלקת העל.

7.1. הורשה – היכרות

על מנת להוריש מחלקה ומאפייניה אנו נכלול את ההגדרה של מחלקה אחת בתוך השנייה על ידי שימוש במילת המפתח extends.
הצורה הכללית למחלקה המאפשרת הורשה הנה:

```
class sub class-name extends super class-name {
//body of class
}
```

הבא נבחן את הדוגמא הבאה

בדוגמא זו אנו יוצרים מחלקת על הנקראת MyClass1 ותת מחלקה הנקראת MyClass2. נשים לב כיצד מתבצעת ההורשה תוך שימוש במילת המפתח extends

```
/*
 A simple example of inheritance
*/
class MyClass1
{     // Declare class
      int x, y;

      void showxy()
      {
            System.out.println("x and y:" + x + "" + y);
      }
}

class MyClass2 extends MyClass1
{
      int z;
      void showz()
      {
            System.out.println("z:" + z);
      }

      void sum()
      {
            System.out.println("x+y+z:" + (x + y + z));
      }
}

class SimpleInherinte
{
      public static void main(String args[])
      {
            MyClass1 superObject = new MyClass1();
            MyClass2 subObject = new MyClass2();
            //we can use the super class itself
            superObject.x = 10;
            superObject.y = 20;
            System.out.println("Content of Super Object :");
            superObject.showxy();
            System.out.println();

            //note that the sub class has access to all public
members of its super class
            subObject.x = 5;
            subObject.y = 6;
            //superObject.z = 7;
            System.out.println("Content of subObject:");
            subObject.showxy();
            subObject.showz();
            System.out.println("Sum of x, y and z in sub Object:");
            subObject.sum();
      }
}
```

הסבר

בתכנית זו אנו רואים כיצד מחלקת העל הורישה את מאפייניה לתת המחלקה. יש לשים דגש על כך כי למרות שתת המחלקה מכילה את כלל החברים של מחלקת העל, היא איננה יכולה לגשת לנתונים המוגדרים כ protected.

דוגמא

בדוגמא זו נבחן את מחלקת Box ונשתמש במחלקת העל ובתת מחלקה אשר תאפשר למחלקת Box לקבל רכיב חישוב חדש בשם weight

```
/*
 Adding the weight feature into Box
*/
class Box
{      // Declare class
       double width;
       double height;
       double depth;

       Box (Box ob)
       {
              width = ob.width;
              height = ob.height;
              depth = ob.depth;
       }
       //constructor used when all dimensions specified
       Box(double w, double h, double d)
       {
              width =w;
              height =h;
              depth =d;
       }
       //constructor used when NO dimensions specified
       Box()
       {
              width=-1;//Use -1 to indicate an un initialsied box
              height=-1;
              depth=-1;
       }
       //constructor when cube is created
       Box(double len)
       {
              width=height=depth=len;
       }
       //compute and return volume
       double volume()
       {
              return height*width*depth;
       }
}
       //Here Box is extends to indicate weight
class BoxWeight extends Box
{
       double weight;//weight of the Box

       //construct a clone of an object
       BoxWeight(double w, double h, double d, double m)
       {
              width =w;
              height =h;
              depth =d;
              weight =m;
       }
}
```

```
//This class declare an object of type Box
class DemoBox
{
	public static void main(String args[])
	{

		BoxWeight mybox1 = new BoxWeight(10,20,15, 34.3);
		BoxWeight mybox2 = new BoxWeight(2,3,4,0.076);
		double vol;

		//get volume of first box
		vol = mybox1.volume();
		System.out.println("Volume in mybox1 is" + vol);
		System.out.println("Weight in mybox1 is" +
mybox1.weight);
		System.out.println();

		//get volume of second box
		vol = mybox2.volume();
		System.out.println("Volume in mybox2 is" + vol);
		System.out.println("Weight in mybox2 is" +
mybox1.weight);
		System.out.println();

	}
}
```

7.2. משתנה של מחלקת על המתייחס לאובייקט של תת מחלקה

משתנה התייחסות של מחלקת על יכול להיות שמיש לכל התייחסות של תת מחלקה היוצאת ממחלקת על. אנו נמצא את האספקט של הורשה כתכונה מאוד שימושית במגוון נרחב של מצבים נתונים.
על מנת להבין כיצד התהליך מתבצע בפועל הבא נבחן את הדוגמה הבאה:

```
/*
 Adding the weight feature into Box
*/
class Box
{      // Declare class
       double width;
       double height;
       double depth;

       Box (Box ob)
       {
              width = ob.width;
              height = ob.height;
              depth = ob.depth;
       }
       //constructor used when all dimensions specified
       Box(double w, double h, double d)
       {
              width =w;
              height =h;
              depth =d;
       }
       //constructor used when NO dimensions specified
       Box()
       {
              width=-1;//Use -1 to indicate an un initialsied box
              height=-1;
              depth=-1;
       }
       //constructor when cube is created
       Box(double len)
       {
              width=height=depth=len;
       }
       //compute and return volume
       double volume()
       {
              return height*width*depth;
       }
}
       //Here Box is extends to indicate weight
class BoxWeight extends Box
{
       double weight;//weight of the Box

       //construct a clone of an object
       BoxWeight(double w, double h, double d, double m)
       {
              width =w;
              height =h;
              depth =d;
              weight =m;
       }
}
```

```
//This class declare an object of type Box
class Reference
{
      public static void main(String args[])
      {

      BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.12);
            Box plainbox = new Box();
            double vol;

            //get volume of first box
            vol = weightbox.volume();
      System.out.println("Volume in weightbox is" + vol);
System.out.println("Weight of weightbox is" + weightbox.weight);
            System.out.println();

            //assign BoxWeight ref to box ref
            plainbox = weightbox;
            vol = plainbox.volume();
System.out.println(" Volume of plainbox is:" + vol);
      }
}
```

הסבר

בתרגיל זה weightbox מתייחס לאובייקטים של BoxWeight ו plainbox מתייחס לאובייקטים של Box.

7.3. שימוש בפונקצית ()super

עד כה ראינו כיצד יישמנו מחלקות בצורה הפשוטה ביותר. לדוגמא קונסטרקטור של מחלקה מסוימת מאתחל שדות מסוימים. ומכאן שלא ניצלנו את מיטב התכונות שקיימות בשפה. כמו גם בדוגמאות עד כה היה קיים שכפול קוד שאיננו יעיל.
ישנם מצבים בהם נרצה לייצור מחלקות על אשר שומרות את היישום לעצמם קרי שומרות את נתוני החברים כפרטיים. מאחר שתכונת הריכוזיות קיימת אזי כי ישנו פתרון לבעיה שכזו.
כאשר תת מחלקה צריכה להתייחס מיידית למחלקת העל שלה היא יכולה לעשות זאת תוך שימוש במילת המפתח super.
למילת מפתח זו ישנם שתי צורות כלליות:

1. האחת קוראת לקונסטרקטור של מחלקת העל
2. השנייה משמשת לגישה של חבר במחלקת העל אשר חבוי על ידי חבר בתת המחלקה.

<u>שימוש במילת המפתח super</u>

תת מחלקה יכולה לקרוא לקונסטרקטור של פונקציה על ידי הגדרה של מחלקת העל תוך שימוש בצורה הבאה של מילת המפתח super:

```
super (parameter list);
```

כאשר:
Super – הנה מילת מפתח לייחוס
Parameter List- הנה רשימת הפרמטרים שהקונסטרקטור צריך ממחלקת העל.

דוגמא

בדוגמא זו נבחן את השיטה הראשונה לשימוש במילת המפתח super. יהיה ניתן לראות כי פונקצית BoxWeight() קוראת לפונקצית super() עם פרמטרים של w,h,d על ידי כך הקונסטרקטור של פונקצית Box() נקרא, שהוא בעצם פעולתו מאתחל את width, height, depth תוך שימוש בערכיהם. BoxWeight איננו מאתחל את הערכים הללו יותר כך שהוא אמור לאתחל רק את ערכיו הייחודיים קרי weight.

כמו גם נראה בדוגמא זו כי פונקצית super() נקראת על ידי שלושה ארגומנטים. מאחר שקונסטרקטורים יכולים להיות נטענים ומועמסים, פונקצית super() יכולה להיקרא תוך שימוש בכל צורה שהוגדרה על ידי מחלקת העל. הקונסטרקטור המורץ יהיה הכלי להתאמת הארגומנטים.

```
/*
 A complete implementation using Box with constructors
*/
class Box
{      // Declare class
      private double width;
      private double height;
      private double depth;

      Box (Box ob)
      {
            width = ob.width;
            height = ob.height;
            depth = ob.depth;
      }
      //constructor used when all dimensions specified
      Box(double w, double h, double d)
      {
            System.out.println("Constructing Box");
            width =w;
            height =h;
            depth =d;
      }
      //constructor used when NO dimensions specified
      Box()
      {
            width=-1;//Use -1 to indicate an un initialsied box
            height=-1;
            depth=-1;
      }
      //constructor when cube is created
      Box(double len)
      {
            width=height=depth;
      }
      //compute and return volume
      double volume()
      {
            return height*width*depth;
      }
}
      //Here Box is extends to indicate weight
      class BoxWeight extends Box
      {
            double weight;//weight of the Box

            //construct a clone of an object
            BoxWeight(BoxWeight ob)
            {
                  super(ob);
```

```
            weight=ob.weight;
        }
        //constructor when all parameters are specified
        BoxWeight(double w, double d, double h, double m)
        {
            super(w, h, d);//call superclass constructor
            weight = m;
        }
        //default constructor
        BoxWeight()
        {
            super();
            weight=-1;
        }
        //constructor used when cube is created
        BoxWeight(double len, double m)
        {
            super(len);
            weight=m;
        }
    }

    //This class declare an object of type Box
    class SuperDemo
    {
        public static void main(String args[])
        {
            BoxWeight mybox1 = new BoxWeight(10,20,15, 34.3);
            BoxWeight mybox2 = new BoxWeight(2,3,4,0.076);
            BoxWeight mybox3 = new BoxWeight();
            BoxWeight mycube = new BoxWeight(3,2);
            BoxWeight myclone = new BoxWeight(mybox1);
            double vol;
            //get volume of first box
            vol = mybox1.volume();
            System.out.println("Volume in mybox1 is" + vol);
    System.out.println("Weight in mybox1 is" + mybox1.weight);
            System.out.println();
            //get volume of second box
            vol = mybox2.volume();
            System.out.println("Volume in mybox2 is" + vol);
    System.out.println("Weight in mybox2 is" + mybox1.weight);
            System.out.println();

            vol = mybox3.volume();
            System.out.println("Volume in mybox3 is" + vol);
    System.out.println("Weight in mybox3 is" + mybox1.weight);
            System.out.println();

            vol = myclone.volume();
            System.out.println("Volume in myclone is" + vol);
    System.out.println("Weight in mybox1 is" + mybox1.weight);
            System.out.println();

            vol = mycube.volume();
            System.out.println("Volume in mycube is" + vol);
    System.out.println("Weight in mycube is" + mybox1.weight);
            System.out.println();
        }
    }
```

דוגמא

בדוגמא זו נבחן את השיטה השנייה לשימוש במילת המפתח super. השימוש השני זהה במידה מסויימת לשימוש במילת הייחוס this מלבד שהוא כל הזמן מתייחס למחלקת העל של תת המחלקה שבה אנו משתמשים. לשימוש זה קיימת הצורה הכללית הבאה:

super.member

כאשר:

Member – משתנה או פונקציה כלשהי

דוגמא

```
/*
 A simple example of inheritance
*/
class MyClass1
{      // Declare class
       int x;
}
class MyClass2 extends MyClass1
{
       int x;

       MyClass2(int a, int b)
       {
              super.x =a;
              x=b;
       }

       void show()
       {
              System.out.println("x in Super Class:" + super.x);
              System.out.println("x in Super class:" + x);
       }
}

              class SuperTest
              {
                     public static void main(String args[])
                     {
                            MyClass2 subObject = new MyClass2(2,4);
                            subObject.show();
                     }
              }
```

7.4. יצירת היררכיה עם רמות

עד כה עסקנו בצורת היררכיה הכוללת רק מחלקות על ותת מחלקות. אולם נוכל לבנות היררכיה המכילה מספר רב של רמות הורשה ככל שנרצה. וכפי שכבר הוזכר זה מאוד נפוץ להשתמש בתת מחלקה כמחלקת על של מחלקה אחרת.
לדוגמא, נניח שיש לנו שלוש מחלקות A B C. מחלקה C יכולה להיות תת מחלקה של B ומחלקה B יכולה להיות תת מחלקה של A. כאשר מקרה כזה קורה תת המחלקה מורישה את מכלוליה.

דוגמא

```
/*
 A complete implementation using Box with constructors
*/
class Box
{      // Declare class
       private double width;
       private double height;
       private double depth;

       //Construct clone of an object
       Box (Box ob)
       {
              width = ob.width;
              height = ob.height;
              depth = ob.depth;
       }
       //constructor used when all dimensions specified
       Box(double w, double h, double d)
       {
              width =w;
              height =h;
              depth =d;
       }
       //constructor used when NO dimensions specified
       Box()
       {
              width=-1;//Use -1 to indicate an un initialsied box
              height=-1;
              depth=-1;
       }
       //constructor when cube is created
       Box(double len)
       {
              width=height=depth;
       }
       //compute and return volume
       double volume()
       {
              return height*width*depth;
       }
}
       //Here Box is extends to indicate weight
class BoxWeight extends Box
{
       double weight;//weight of the Box



       //construct a clone of an object
       BoxWeight(BoxWeight ob)
       {
```

```
		super(ob);
		weight=ob.weight;
	}
	//constructor when all parameters are specified
	BoxWeight(double w, double d, double h, double m)
	{
		super(w, h, d);//call superclass constructor
		weight = m;
	}
	//default constructor
	BoxWeight()
	{
		super();
		weight=-1;
	}
	//constructor used when cube is created
	BoxWeight(double len, double m)
	{
		super(len);
		weight=m;
	}
}

	//Add shipping Costs
class Shipping extends BoxWeight
{
	double cost;

	//constructor when all parameters are specified
	Shipping (Shipping ob)
	{ //pass object to constructor
		super(ob);
		cost = ob.cost;
	}
	//constructor when all parameters are specified
	Shipping (double w, double d, double h, double m, double c)
	{
		super(w, h, d, m);//call superclass constructor
		cost = c; //called to c variable
	}

	//default constructor
	Shipping()
	{
		super();
		cost = -1;
	}
	//constructor used when cube is created
	Shipping(double len, double c, double m)
	{
		super (len, m);
		cost = c;
	}
}

//This class declare an object of type Box
class ShippingDemo
{
	public static void main(String args[])
	{
		Shipping shipment1 = new Shipping(10, 15, 20, 10, 3.41);
```

```
            Shipping shipment2 = new Shipping(2, 3, 4, 0.76, 1.28);

            double vol;

            //get volume of first box
            vol = shipment1.volume();
            System.out.println("Volume of shipment1 is" + vol);
System.out.println("Weight in shipment1 is" + shipment1.weight);
            System.out.println("Shiping Costs: $" + shipment1.cost);
            System.out.println();

            //get volume of second box
            vol = shipment2.volume();
            System.out.println("Volume of shipment2 is" + vol);
System.out.println("Weight in shipment2 is" + shipment2.weight);
            System.out.println("Shiping Costs: $" + shipment2.cost);
            System.out.println();


      }
}
```

תוצאת התכנית

Volume of shipment1 is 3000
Weight in shipment1 is 10
Shiping Costs: $ 3.41

Volume of shipment2 is 24
Weight in shipment2 is 0.76
Shiping Costs: $ 1.28

הסבר

בגלל תהליכי הורשה, מחלקת Shipping יכולה לבצע שימוש בנתונים ממחלקות Box ו-BoxWeight תוך שימוש בלבד של הנתונים שאנו צריכים לביצוע החישוב.
חשוב להבין כי תהליך ההורשה מאפשר לנו שימוש חוזר ונשנה בקוד.
בדוגמא גם נוכל לראות כי פונקצית super() מתייחסת לקונסטרקטור של מחלקת העל הקרובה אליה ביותר. פונקצית super() במחלקת Shipping קוראת לקונסטרקטור במחלקת Box ופונקצית super() במחלקת Boxweight קורא לקונסטרקטור ב Box.

7.5. שימוש בקונסטרקטורים

כאשר אנו יוצרים היררכיה במחלקות נשאלת השאלה באיזה סדר הקונסטרקטורים נקראים בהיררכיה.
לדוגמא אם קיימת תת מחלקה בשם B ומחלקת על בשם A האם הקונסטרקטור של A נקרא לפני הקונסטרקטור של B או ההיפך?
התשובה היא שהקונסטרקטורים של המחלקות נקראים בצורה שהם נכתבו קרי ממחלקת העל לתת המחלקה.
מעבר לכך מאחר שפונקצית super() חיבת לרוץ ראשונה בהצהרה של תת מחלקה אזי שלפי סדר זה הקומפיילר עובד. במידה ופונקצית super() איננה בשימוש אזי נשתמש ב default constructor.

דוגמא

```java
/*
 Demonstration for Constructors
*/
class A
{      // Declare class
       A()
       {
              System.out.println("Inside A");
                                                                 }
}

class B extends A
{
       B()
       {
              System.out.println("Inside B");
                                                                 }
}

class C extends B
{
       C()
       {
              System.out.println("Inside C");
                                                                 }
}

       //This class declare an object of type Box
class Const
{
       public static void main(String args[])
       {
              C c = new C();
       }
}
```

תוצאת התכנית

Inside A
Inside B
Inside C

הסבר

כפי שניתן לראות הקונסטרקטורים נקראים על פי תהליך היווצרותם.

7.6. שכתוב פונקציות – Method Overriding

בהיררכיה של מחלקות כאשר לפונקציה ישנו את אותו שם ואותו סוג נתונים כמו שלפונקציה הנמצאת במחלקת העל אזי נוכל לומר כי הפונקציה בתת המחלקה הנה שכתוב Overriding.
כאשר פונקציה משוכתבת נקראת מתוך תת המחלקה היא תמיד מתייחסת לגרסה הנמצאת בתת המחלקה. הגרסה של הפונקציה הנמצאת במחלקת העל תהיה מוסתרת.

דוגמא

```java
/*
 methods Overriding
*/
class A
{      // Declare class
int i, j;

       A(int a, int b)
       {
              i=a;
              j=b;
       }
       //display i and j
              void show() {
              System.out.println("I and J:" + i + " " + j);
          }
}

class B extends A
{
       int k;

       B(int a, int b, int c)
       {
              super (a, b);
              k=c;
       }
       void show()
       {
              System.out.println("K:" + k);
       }
}

class Override
{
       public static void main(String args[])
       {
              B subOb = new B(1,2,3);
              subOb.show();
       }
}
```

תוצאת התכנית

K=3

הסבר

כאשר פונקצית show() קוראת לגרסה הנמצאת במחלקה B אזי הפונקציה במחלקה B תהיה שימושית. וכך המקרה לגבי שאר פונקציות ה show().

אם נרצה לגשת לגרסת התת מחלקה של פונקציה משוכתבת נוכל לעשות זאת על ידי שימוש ב super()
לדוגמא בתרגיל הקודם, בגרסה של מחלקה B הגרסה של מחלקת העל של פונקצית show() מבצעת שימוש בתת המחלקה עצמו. דבר זה מאפשר למשתנים להיות מוצגים.

דוגמא

```
/*
 methods Overriding
*/
class A
{      // Declare class
int i, j;

       A(int a, int b)
       {
             i=a;
             j=b;
       }
       //display i and j
             void show() {
             System.out.println("I and J:" + i + " " + j);
         }
}

class B extends A
{
       int k;

       B(int a, int b, int c)
       {
             super (a, b);
             k=c;
       }
       void show()
       {
             super.show();//this is calls A show()
             System.out.println("K:" + k);
       }
}

class Override
{
       public static void main(String args[])
       {
             B subOb = new B(1,2,3);
             subOb.show();
       }
}
```

תוצאת התכנית

I and J: 1 2
K:3

יש לזכור כי הפונקציה המשוכתבת חייבת להיות תואמת לפונקציה המקורית אחרת אם ננסה לשכתב אותן יקרה מצב של טעינת פונקציות. לדוגמא

```
/*
 methods with different type signature that overloaded NOT Overriding
*/
class A
{     // Declare class
int i, j;

      A(int a, int b)
      {
            i=a;
            j=b;
      }
      //display i and j
            void show() {
            System.out.println("I and J:" + i + " " + j);
        }
}

class B extends A
{
      int k;

      B(int a, int b, int c)
      {
            super (a, b);
            k=c;
      }
      void show(String msg)
      {
                        System.out.println(msg + k);
      }
}

class Override
{
      public static void main(String args[])
      {
            B subOb = new B(1,2,3);
            subOb.show("This is k:");
            subOb.show();
      }
}
```

תוצאת התכנית

This is k:3
I and J: 1 2

7.7. שליחת פונקציות דינאמית

עד כה ראינו כי שכתוב הנה תכונה קיימת ברוב שפות התכנות אולם לא עמדנו על חוזק מרכיביה. בחלק זה נדון ונראה כיצד שכתוב פונקציות מתבצע באופן דינמי.
באומרנו אופן דינאמי הכוונה הנה שפונקציה משוכתבת מתבצעת בזמן Run time ולא בזמן הרצת התכנית בפועל.
שליחת שכתוב פונקציות דינאמיות הנה אחת מתכונות רב הצורתיות ש Java מספקת לנו.
הבא נבחן את הדוגמא הבאה ע"י דיון אקדמי:
משתנה עם התייחסות במחלקת על יכול להתייחס לאובייקט בתת מחלקה. בעובדה זו הקומפיילר נעזר על מנת לפתור את שכתוב הפונקציה ב run time. כך זה מתבצע:
כאשר פונקציה משוכתבת נקראת ע"י התייחסות של מחלקת על, Java קובעת איזה סוג של גרסה של הפונקציה להריץ וזאת ייקבע בהתאם לאובייקט אליו היא משויכת או מתייחסת בזמן הקריאה. לפיכך ניתן לומר כי הקביעה נעשית בזמן run time. כאשר אובייקטים שונים מתייחסים לגרסאות משוכתבות שונות אזי גרסאות שונות של פונקציות ייקראו. במילים אחרות, ניתן לומר כי אנו מתייחסים לסוג האובייקט והוא בעצם זה שאומר לנו איזה גרסה משוכתבת של פונקציה תרוץ.
לפיכך אם מחלקת על מכילה פונקציה משוכתבת על ידי תת מחלקה אזי כאשר אובייקטים שונים יתייחסו דרך המשתנים המוגדרים במחלקת העל פונקציות שונות ירוצו.
בהשוואה לשפת C++ אזי נוכל לומר כי שכתוב פונקציות דומה לשימוש בפונקציות וירטואליות.

דוגמא

```
/*
 Dynamic method
*/
class A
{      // Declare class
       void callme()
       {
             System.out.println("Inside A");
       }
}

class B extends A
{
       //override callme()
       void callme()
       {
             System.out.println("Inside B");
       }
}

class C extends A
{
       //override callme()
       void callme()
       {
             System.out.println("Inside C");
       }
}
class DisOverride
{
       public static void main(String args[])
       {
             A a = new A();
             B b = new B();
             C c = new C();
             A Ref;//obtain reference of type A

             Ref=a;//Ref to a object
```

```
            Ref.callme();

            Ref=b;//Ref to b object
            Ref.callme();

            Ref=c;//Ref to c object
            Ref.callme();
      }
}
```

תוצאת התכנית

Inside A
Inside B
Inside C

הסבר

תכנית זו יוצרת מחלקת על אחת בשם A ושני תת מחלקות בשם B ו C. תת המחלקות B ו C משכתבים שוב את פונקציית callme() אשר הוגדרה במחלקת העל. בתוך פונקצית ה main() מוגדרים אובייקטים A B C . בנוסף ישנה התייחסות לאובייקטים אלו ע"י Ref.

7.8. מדוע לשכתב פונקציות

דיון אקדמי קצר יוצג בחלק זה על הסיבה לשכתוב פונקציות.
כפי שהוסבר שכתוב פונקציות מאפשר ל Java לתמוך ב run time polymorphism . רב צורתיות הנה תכונה חיונית לתכנות מונחה עצמים מסיבה אחת פשוטה: הוא מאפשר למחלקה כללית להגדיר פונקציות אשר יהיו כלליות לכל מה שנגזר מהן ובאותו זמן לאפשר לתת המחלקה להגדיר את הפונקציות שהן נקודתית צריכות.
שכתוב פונקציות הנה דרך נוספת ליישום Java הנקרא "ממשק אחד פונקציות רבות".
חלק מהיישום הנכון של רב צורתיות הנה ההבנה כי מחלקות על ותת מחלקות מספקות לנו היררכיה בתכנית. כאשר השימוש נעשה באופן הנכון מחלקת העל מספקות את כל האלמנטים שתת מחלקה יכולה להשתמש ישירות. בנוסף היא גם מגדירה את אותם פונקציות הנגזרות על מנת שאותה מחלקה תיישם אותן באופן עצמאי.
תכונה זו מאפשרת לתת המחלקה להיות גמישה להגדרות מקומיות ולפונקציות משלה.
לפיכך על ידי שילוב של הורשה עם שכתוב פונקציות מחלקת העל יכולה לספק לנו מבנה אחיד לפונקציות אשר כלל תת המחלקות ישתמשו בהן.
Dynamic run time polymorphism הנו אחד מהטכניקות בעיצוב שתכנות מונחה עצמים אשר עוזר לנו לבצע שימוש חוזר בקוד. היכולת של קוד הקיים בספריות לקרוא לפונקציות ולמשתנים שמניים ללא כל צורך בקומפילציה חדשה הנה כלי חשוב מאוד בתכנון הקוד.

7.9. יישום שכתוב פונקציות

הבא ניישם את מה שנכתב לעיל. בתכנית הבאה נייצר מחלקת על בשם Figure אשר מאכלסת מידות של אובייקטים דו מימדיים. בנוסף מחלקת העל מגדירה פונקציה בשם area() המחשבת את שטח האובייקטים.
בהמשך התכנית ניצור שתי תתי מחלקות למחלקת העל. המחלקה הראשונה הנה Rectangle והאחרת הנה Triangle . כאשר כל תת מחלקה תבצע שימוש בפונקצית area() לצורכה הפרטי.

דוגמא

```
/*
 Using run time Polymorphism
*/
class Figure
{      // Declare class
       double dim1;
       double dim2;

       Figure(double a, double b)
       {
              dim2 = a;
              dim2 = b;
       }

       double area()
       {
              System.out.println("Arae for Figure is Undefined");
              return 0;
       }
}

class Rectangle extends Figure
{
       Rectangle (double a, double b)
       {
              super(a, b);
       }

       //oerride area for rectangle
       double area()
       {
              System.out.println("Inside area for rectangle");
              return dim1*dim2;
       }
}

class Triangle extends Figure
{
       Triangle (double a, double b)
       {
              super(a, b);
       }

       //oerride area for Triangle
       double area()
       {
              System.out.println("Inside area for Triangle");
              return dim1*dim2/2;
       }
}
```

```
class FindAreas
{
	public static void main(String args[])
	{
		Figure f = new Figure(10, 10);
		Rectangle r = new Rectangle(9,5);
		Triangle t = new Triangle(10,8);

		Figure Ref;//obtain reference of type A

		Ref=r;//Ref to r object
		System.out.println("Area is" + Ref.area());

		Ref=t;
		System.out.println("Area is" + Ref.area());

		Ref=f;
		System.out.println("Area is" + Ref.area());
	}
}
```

תוצאת התכנית

Inside area for rectangle
Area is 45
Inside area for triangle
Area is 40
Inside area for undefined
Area is 0

7.10. שימוש במחלקה בונה

ישנם מצבים בהם נרצה להגדיר מחלקת על אשר מגדירה מבנה נתון מסוים ללא אספקת היישום הכללי לכל פונקציה. ז"א לפעמים נרצה לייצור מחלקת על אשר מגדירה מבנה כללי אשר תת המחלקות ישתמשו בו ובעצם כל תת מחלקה תדאג לנתונים שלה. מחלקה שכזו קובעת את אופי הפונקציה שתת המחלקה תיישם.
מצב כזה יכול לקרות כאשר מחלקת על איננה יכולה לייצור יישום בעל משמעות לפונקציה נתונה. כפי שנראה בהמשך אותה תופעה של יצירת פונקציה ללא משמעות איננה מצב יוצא דופן. לדוגמא, אם נסתכל בדוגמא הקודמת במחלקת Triangle אזי נוכל לומר כי לפונקצית area() אין כל משמעות באם היא תוגדר או לא מהסיבה שהיא מוגדרת כבר במחלקת העל. במקרה שכזה נרצה כמובן לאשרר כי תת המחלקה אכן שכתבה את הפונקציות אליה היא התייחסה.
הפתרון של שפת Java הנו שימוש בפונקציה בונה abstract method.
אנו יכולים לדרוש מהקוד בתת מחלקה מסוימת פונקציה מסוימת תהיה משוכתבת על ידי הדגרת מילת המפתח abstract.
פונקציות אלו לפעמים מיוחסות ל subclasser responsibility מכיוון שאין להם יישום מוגדר במחלקת העל. לפיכך תת מחלקה חייבת לשכתב אותם אולם אותה תת מחלקה איננה יכולה להשתמש בגרסה שהוגדרה המחלקת העל.

המבנה הכללי הנו:

```
abstract type name(parametr list);
```

כפי שניתן לראות מהתחביר לא מוגדר שום גוף לפונקציה.
כל מחלקה המכילה abstract אחד או יותר חייבת להיות מוגדרת כמחלקה שכזו. על מנת לבצע זו אנו נוסיף את מילת המפתח abstract לפני מילת ה class.
למחלקה מסוג זה לא קיימים אובייקטים, ולפיכך ניתן לומר כי אין באפשרות מחלקה שכזו להשתמש באופרטור new.
בנוסף אי אפשר להגדיר קונסטרקטור של abstract או abstract static method.

על מנת להבין כיצד התהליך מתבצע הבא נבחן את הדוגמא הבאה.

דוגמא

```
/*
 Abstarct
*/
abstract class A
{      // Declare class
       abstract void callme();

       //double methods are allowed in abstract
       void callmetoo()
       {
              System.out.println("Double Method");
       }
}

class B extends A
{
       //override callme()
       void callme()
       {
              System.out.println("Implementation of B");
       }
}

class AbstractTest
{
       public static void main(String args[])
       {
              B b = new B();

              b.callme();
              b.callmetoo();
       }
}
```

תוצאת התכנית

Double Method
Implementation of B

הסבר
נשים לב כי למחלקה A לא הוגדר שום אובייקט. כמו גם נשים לב כי בתוך אותה מחלקה מותר לנו לבצע כתיבה של מספר פונקציות במקביל.

למרות שמחלקה המוגדרת כ abstract איננה יכולה לכלול אובייקטים, היא אכן יכולה לכלול התייחסות לאובייקטים מכיוון ש Java run time polymorphism מיושם דרך התייחסות למחלקת העל.

הבא ניישם את הדוגמא עם מחלקת Figure.

דוגמא

```
/*
 Using abstract run time Polymorphism
*/
abstract class Figure // Declare abstract class
{
	double dim1;
	double dim2;

	Figure(double a, double b)
	{
		dim2 = a;
		dim2 = b;
	}
	//area is now an abstarct method
	abstract double area();
}
class Rectangle extends Figure
{
	Rectangle (double a, double b)
	{
		super(a, b);
	}

	//oerride area for rectangle
	double area()
	{
		System.out.println("Inside area for rectangle");
		return dim1*dim2;
	}
}

class Triangle extends Figure
{
	Triangle (double a, double b)
	{
		super(a, b);
	}

	//oerride area for Triangle
	double area()
	{
		System.out.println("Inside area for Triangle");
		return dim1*dim2/2;
	}
}


class AbstracTest
{
	public static void main(String args[])
	{
		//Figure f = new Figure(10, 10);//illigal now cause of 
abstract
		Rectangle r = new Rectangle(9,5);
		Triangle t = new Triangle(10,8);

		Figure Ref;//this is fine but NO object is created

		Ref=r;//Ref to r object
		System.out.println("Area is" + Ref.area());
```

```
            Ref=t;
            System.out.println("Area is" + Ref.area());

      }
}
```

תוצאת התכנית

Inside area for rectangle
Area is 45
Inside area for triangle
Area is 40

הסבר

ההערה בתוך פונקצית main() אומרת כי לא ניתן לנו להגדיר אובייקטים מסוג Figure מאחר שהיא מחלקת abstract. וכלל תת המחלקות של Figure חייבות לשכתב את פונקצית area().

7.11. שימוש ב Final בהורשה

למילת המפתח Final ישנם שלושה שימושים:

1. שימוש ליצירת שווה ערך של קבוע מסוים
2. מניעת שכתוב
3. מניעת הורשה

השימוש הראשון התבצע במילת Final תוך כדי כתיבת קוד שכבר עסקנו בו.

7.11.1. מניעת שכתוב

השימוש השני נעשה על מנת למנוע שכתוב של פונקציה. על מנת לבצע זאת הרי שנשתמש במילת המפתח final לפני שם הפונקציה.
פונקציות המוגדרות במילת המפתח final לא יכולות להיות משוכתבות.

דוגמא

```
/*
 using final
*/
class A {
    final void meth(){
            System.out.println("This is final Method");
      }
}

class B extends A
{
      //override callme()
      void meth() //ERROR!!!
      {
            System.out.println("Implementation of B");
      }
}
```

7.11.2. מניעת הורשה

כאשר נרצה למנוע ממחלקה לבצע הורשה נשתמש במילת המפתח final לפני הגדרת המחלקה. כמו כן צריך להיות ברור כי אין אפשרות לייצור מחלקה שהיא גם abstract וגם final

דוגמא

```
/*
 using final
*/
final class A {
    void meth(){
            System.out.println("This is final Method");
      }
}

class B extends A
{
      //override callme()
      void meth() //ERROR!!!Can not inherinte!!!
      {
            System.out.println("Implementation of B");
      }
}
```

7.12. אובייקט המחלקה

ישנה מחלקה אחת ייחודית ב java הנקראת Object. כל שאר המחלקות הנן נגזרות של מחלקה זו. ומכאן נאמר כי Object הנה מחלקת על לכל שאר המחלקות.
ומכאן משתנה ההתייחסות של Object יכול להתייחס לכל אובייקט בכל מחלקה שהיא. בנוסף מאחר שמערכים מיושמים כמחלקה משתנה מסוג Object יכול להתייחס גם למערך.
מחלקת Object מגדירה את הפונקציות הבאות ז"א אותן פונקציות הנם נתונות ומוכנות לעבודה בכל אובייקט שנגדיר.

Method	**Purpose**
Object clone()	Creates a new object that is the same as the object being cloned
boolean equals(object *object*)	Deterimine whether one object is equal to another
void finalize()	Called before an unused object is recycled
Class getClass()	Obtain the class of an object at run time
Int hashCode()	Returns the hash code associates with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void notifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returning a string that describe the object
void wait()	Waits on another thread of execution.

7.13. Upcasting

התכונה החשובה ביותר של הורשה הנה יחסי הגומלין בין המחלקה החדשה שיצרנו לבין מחלקת האב. נוכל לסכם ולומר כי יחסים אלו הנם יחסים של אב ובן קרי לבן קיימים את מאפייני האב.
הסבר זה אינו בא על מנת לצאת ידי חובת הסבר המינוח "הורשה" אלא ההפיך הוא הנכון, שפת java תומכת ישירות בהורשה.
לדוגמא, נחשוב כי קיימת לנו מחלקה בשם Instrument אשר מייצגת כלים מוסיקליים ותת המחלקה שלה נקראת Wind. מפאת ההגדרה של הורשה קרי כלל הפונקציות הנם אפשריות לשימוש על ידי תת המחלקה, ולכן כל הודעה שנרצה לשלוח ממחלקת העל יכולה להישלח גם מתת המחלקה.
ולדוגמתנו, אם למחקלת Instrument ישנה פונקציה בשם play() אזי גם למחלקת Wind תהיה פונקציה שכזו (כמובן היא מורשת אוטומטית). ומכאן נוכל לומר כי אנו אובייקט Wind הנו מסוג Instruments.

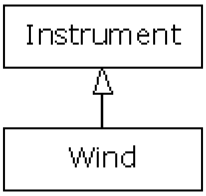
דוגמא

```
// Inheritance & upcasting.
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // body
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  public static void main(String args[]) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```

הסבר התכנית
המעניין בתכנית זו הנו השימוש בפונקצית tune() אשר מתייחסת למחלקת האב Instruments. אולם אם נסתכל על Wind.main() פונקצית tune() נקראת על ידי התייחסות ל Wind. בהתייחס לעובדה כי שפת תכנות זו הנה בעיקר בודקת את סוג הנתונים, נראה כי סוג זה של התייחסות הנו משונה כאשר פונקציה המקבלת סוג אחד של נתונים יכולה לקבל סוג אחר של נתונים.

7.14. מדוע להשתמש ב Upcasting

הסיבה בעיקרה הנה היסטורית, והיא מבוססת על הפעולה שבה מחלקה מורישה לתת המחלקות קרי קיימת היררכיה.
הדיאגרמה בהתייחס לדוגמא לעיל הנה



ניתן לראות כי בדיאגרמה ההתייחסות הנה בכיוון "עולה" ולא כמו הורשה רגילה בכיוון "יורד" ולכן הביטוי הנו Upcasting.
תכונה זו של Upcasting הנה בטוחה מהסיבה שאנו הולכים מסוג אחד ספיציפי של נתונים לסוג כללי של נתונים קרי, תת המחלקה הנה ה superset של מחלקת העל. תת המחלקה יכולה לכלול יותר פונקציות ממחלקת העל אבל היא חייבת להכיל לפחות את הפונקציות ממחלקת העל.
הדבר היחידי שיכול לקרות לתת המחלקה היורשת ממחלקת העל הנו "איבוד" פונקציות קרי חוסר שימוש באותן פונקציות המוגדרות במחלקת העל.

כמובן שקיימת תכונת Down Casting אך נסתכל על תכונה זו בהמשך.

8. רב צורתיות – Polymorphism

רב צורתיות הנה תכונה המאפשרת לנו הפרדה בין הממשק לבין היישום קרי בין השרת לבין הלקוח. תכונת הרב צורתיות מאפשרת לנו לשפר ולנהל את הקוד בצורה יעילה כך שאת התכונות אותן נרצה ליישם נוכל לבצע.
אם נזכור מהו בסיס OOP אזי כי נוכל לומר שתכונת ריכוזיות (Encapsulation) מייצרת לנו סוג נתונים חדשים על ידי שילוב מאפיינים והתנהגות. יישום חבוי מפריד את הממשק מהיישום על ידי מתן גישה פרטית private.
פונקצית polymorphic מאפשרת לנו לבצע פעולה בה סוג אחד של משתנים "יביע" סוג אחר של משתנים מוגדרים.
נשים לב כי התכונה הזו פועלת אך ורק באם הנתונים ואו המאפיינים הנם מאותו סוג.

8.1 מודל Upcasting

כבר הצגנו את תכונת ה Upcasting בחלק ההורשה קרי כבר ראינו כיצד אובייקט יכול להשתמש בסוג הנתונים שלו או בסוג הנתונים של מחלקת העל. כאשר אנו לוקחים התייחסות של אובייקט ומתייחסים אליה כהתייחסות למחלקת העל אזי כי אנו יוצרים Upcasting מכיוון שהצורה בה מחלקות מורשות מתנהגות.

דוגמא

```
// Inheritance & upcasting.

class Note
{
	private int value;
	private Note(int val) { value = val; }
	public static final Note
		MIDDLE_C = new Note(0),
		C_SHARP  = new Note(1),
		B_FLAT   = new Note(2);
}
class Instrument
{
	public void play(Note n)
	{
		System.out.println("Instrument.play()");
	}
}

	// Wind objects are instruments
	// because they have the same interface:
class Wind extends Instrument
{
	// Redefine interface method:
	public void play(Note n)
	{
		System.out.println("Wind.play()");
	}
}

public class Music
{
	public static void tune(Instrument i)
	{
		i.play(Note.MIDDLE_C);
	}
	public static void main(String[] args)
```

```
        {
                Wind flute = new Wind();
                tune(flute); // Upcasting
        }
}
```

הסבר התכנית

הפונקציה ( )Music.tune מקבלת התייחסות מ Instrument אבל בנוסף כל אשר קשור אליו. בפונקציית ()main נוכל לראות שתהליך זה קורה כהתייחסות ל Wind המועברת לפונקציית ()tune ללא שימוש ב cast אשר תהליך זה קביל.

הממשק ב Instrument חייב להיות קים ב Wind אשר מורש מ Instrument .

8.1.1. ומה קורה באם נשכח מהו סוג האובייקט?

עולה השאלה מה יקרה באם נשכח את סוג האוביקט שלנו. אולם שאלה זו תמוהה, כיצד נוכל לשכוח את סוג האובייקט שלנו? אולם זה התהליך כאשר אנו משתמשים ב Upcast . האומנם נוכל לבצע העברת ארגומנטים ללא שימוש ב Upcast ומדוע לא לבצע זאת?

שאלה זו מעלה לנו נקודה נוספת, אם נשתמש בהעברת ארגומנטים אזי כי נצטרך לייצור בכל פעם פונקציית ()tune חדשה לכל סוג אובייקט Instruments.

ובהנחת יסוד הנ"ל, הבא ניקח את התכנית ונוסיף שני כלים מוסיקליים חדשים:

```
// Overloading instead of upcasting.

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
}

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

class Wind extends Instrument {
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play()");
  }
}

class Brass extends Instrument {
  public void play(Note n) {
    System.out.println("Brass.play()");
  }
}
```

```java
public class Music2 {
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
  }
}
```

הסבר תכנית

תכנית זו עובדת אך ישנה הסתייגות רצינית. אנו חייבים לרשום מהו סוג הפונקציה הספציפית לכל תת מחלקה מסוג Instrument שנוסיף.

ומכאן זה אומר כי אנו רושמים שורות קוד נוספות, ובנוסף זה אומר כי אם נרצה להוסיף פונקציות כמו tune() או סוג חדש של Instrument יש לנו הרבה עבודה לבצע שאיננה יעילה. ובנוסף נתחשב בעובדה כי הקומפיילר לא ייתן לנו שום טעות אם נשכח לבצע שכתוב לפונקציות וכתוצאה מכך כל תהליך כתיבת התכנית הופך להיות בלתי מנוהל.

כמובן שזה יהיה יעיל יותר לכתוב פונקציה אחת אשר ממנה ננהל את כלל הנגזרות של מחלקת האב ולמעשה זוהי התכונה שרב צורתיות מציעה לנו.

8.2 מודל ה Twist

מהדוגמאות לעיל שמרנו את הקובץ של Instrument כקובץ Music.java. קובץ זה נותן לנו את התוצר של פונקצית Wind.play( ). ולמעשה זהו התוצר שאנו רוצים אבל לא נראה במבט ראשוני כי הפונקציה עובדת כראוי.
הבא נסתכל על פונקצית tune() המוגדרת בתכנית:

```java
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

אנו רואים כי הפונקציה מקבלת התייחסות של Instrument. אזי כי נשאלת השאלה כיצד הקומפיילר יכול לדעת שה Instrument הזה מתייחס ל Wind במקרה שבו קיימים לנו "כלים" נוספים מוגדרים?
התשובה היא שהקומפיילר לא יכול לדעת!.
על מנת להבין את הנושא לעומקו הבא נבחן את נושא ה Binding

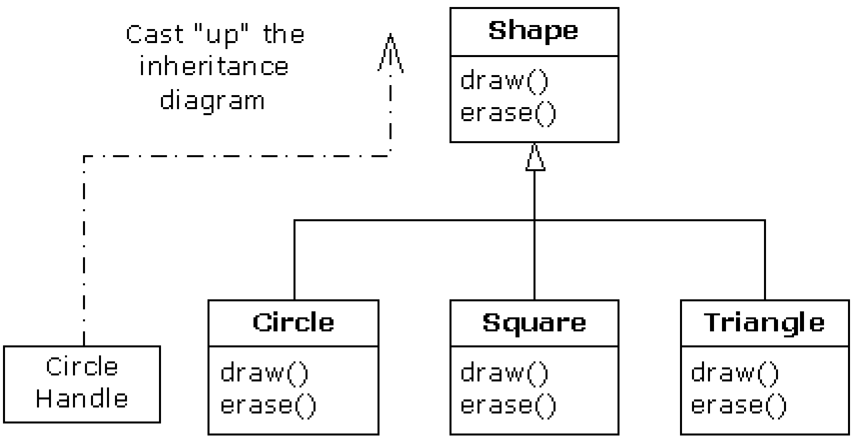
8.2.1 פונקציה הקוראת לאובייקט עטוף

קישור בין קריאה לפונקציה לגוף פונקציה נקרא בשפה המקצועית Binding. כאשר binding קץ לפני התכנית על ידי חבילה אחרת או על ידי תכנית אחרת אזי כי נוכל לקרוא לזה early binding ובהתייחס לשפת C או C++ אזי י לשפות אלו קיים רק מקרה אחד והנו early binding.
הבא ננתח את התכנית שלנו למעלה. להזכירכם, הקומפיילר איננו יודע לאיזו פונקציה לקרוא כאשר קיימת לו התייחסות אחת ל Instrument.
הפתרון לבעיה זו נקרא late binding קרי ה binding יתבצע במהלך ריצת התכנית ויתבסס על סוג האובייקט. המינוח late binding נקרא גם בשפה המקצועית dynamic binding או run-time binding.
כאשר אנו מיישמים את התכונה הזו ישנה טכניקה הקובעת מהו סוג האובייקט שנקבע בזמן ריצת התכנית והמתאים לפונקציה. ומכאן הקומפיילר איננו יודע עדיין את סוג האובייקט אבל הקריאה לפונקציה מבררת את הקריאה הנכונה לגוף הפונקציה על ידי ה binding.
כלל הפונקציות ב java משתמשות ב late binding אלא אם הוגדרו כ final. ואם זה המקרה נשאלת השאלה, מדוע להגדיר פונקציה כ final ?
כפי שכבר הוסבר, הגדרת פונקציה כ final מונעת את שכתובה ומעבר לכך היא איננה נותנת גישה ל dynamic binding.

8.3. כיצד נקבע את ההתנהגות הנכונה?

לאחר שאנו יודעים כי כלל הפונקציות ב java הנן binding אולי רב צורתיות תוך שימוש ב binding יוכל לעזור לנו לקבוע את סוג הנתונים הנכון לשימוש.
הדוגמא הקלאסית ב OOP הנה דוגמאת הצורה. דוגמא זו בד"כ שימושית מפאת הסיבה שקל לנו להבין ויזואלית (אם כי לא לחשוב כי תכנות OOP הנו בצורה ויזואלית).

דוגמאת הצורה וההסבר:
לדוגמא זו ישנה מחלקת על הנקראת Shape וממנה ישנם תת מחלקות שונות Circle, Square, Triangle וכו'.
דיאגרמת ההורשה הנה כדלהלן:



נשים לב כי Upcast יכול להתבצע בהצהרה פשוטה כגון:

```
Shape s = new Circle();
```

בדוגמא הזו, האובייקט מסוג Circle נוצר כתוצאה מהתייחסות מיידית המועברת ל Shape (אשר לכאורה נראה כטעות) מפאת הסיבה ש Circle הנו הורשה של Shape ולפיכך לא תיווצר טעות קומפילציה.
נניח כי נקרא לאחת מפונקציות מחלקות העל בצורה הבא:

```
s.draw();
```

בצורה זו שוב נצפה כי פונקצית draw() של מחלקת Shape תיקרא, אחרי הכל ההתייחסות הנה ל shape ולכן כיצד הקומפיילר יזהה מה לבצע?
ועדיין הפונקציה הנכונה נקראת Circle.draw( ) וזאת מכיוון השימוש ב binding שהנו חלק מ polymorphism.

דוגמא

הבא נבחן את התכנית המלאה לשימוש במחלקת shape.

```
	// Polymorphism in Java.

class Shape
{
	void draw() {}
	void erase() {}
}

class Circle extends Shape
{
	void draw()
	{
		System.out.println("Circle.draw()");
	}
	void erase()
	{
		System.out.println("Circle.erase()");
	}
}

class Square extends Shape
{
	void draw()
	{
		System.out.println("Square.draw()");
	}
	void erase()
	{
		System.out.println("Square.erase()");
	}
}

class Triangle extends Shape
{
	void draw()
	{
		System.out.println("Triangle.draw()");
	}
	void erase()
	{
		System.out.println("Triangle.erase()");
	}
}

public class Shapes
{
	public static Shape randShape()
	{
		switch((int)(Math.random() * 3))
		{
			default:
			case 0: return new Circle();
			case 1: return new Square();
			case 2: return new Triangle();
		}
	}
	public static void main(String[] args)
```

```
	{
		Shape[] s = new Shape[9];
		// Fill up the array with shapes:
		for(int i = 0; i < s.length; i++)
			s[i] = randShape();
		// Make polymorphic method calls:
		for(int i = 0; i < s.length; i++)
			s[i].draw();
	}
}
```

הסבר התכנית

מחלקת העל Shape הנה הממשק הכלי לשאר תת המחלקות הנגזרות ממנה קרי כלל הצורות נגזרות ונמחקות בסוף השימוש. תת המחלקות משכתבות את ההגדרות הקיימות במחלקת העל.

המחלקה הציבורית מכילה פונקציה מסוג static הנקראת randShape( ) המייצרת לנו את ההתייחסות לצורה ראנדומאלית לאובייקט שנבחר בכל פעם שאנו קוראים לה.

נשים לב כי פעולת upcasting קוראת בכל פעם שקיימת בהצהרה return אשר מתייחס ל Circle, Square או Triangle ושולח אותו מחוץ לפונקציה כערך מוחזר של Shape.

כך שבכל מקרה שאם נקרא לפונקציה לא נוכל לדעת מהו סוג הפונקציה מאחר שהיא תמיד מתייחסת ל Shape.

פונקצית main() מכילה מערך של התיחסות לצורות הנענות על ידי שימוש בפונקצית randShape( ).

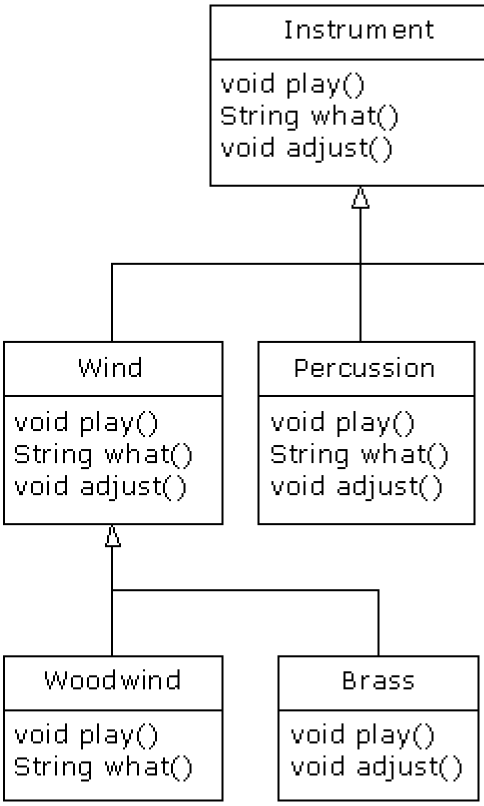
תוצאת התכנית הנה

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

מאחר שבכל פעם צורה נבחרת באקראיות תהיה לנו תוצאה שונה בכל ריצת תכנית.

8.3.1. Extensibility – הארכה

הבא נחזור לדוגמא עם המכשירים המוסיקליים. בגלל השימוש ברב צורתיות, נוכל להוסיף אובייקטים ככל שנרצה ללא שום שינוי בפונקצית ()tune.
בכתיבת קוד נכונה לתכנית הנקודתית כלל הפונקציות יעקבו אחרי הפונקציה הנ"ל והתקשורת תעשה רק דרך מחלקת העל.
לתכנית שכזו אנו קוראים בשפה המקצועית Extensible מהסיבה שאנו יכולים להוסיף פונקציונאליות חדשה על ידי הורשה של סוגי נתונים חדשים ממחלקת העל.
הבא נבחן את הדיאגרמה המתייחסת למחלקת Instrument:



נשים לב כי כלל המחלקות החדשות עובדות ללא שינוי בפונקצית()tune.

דוגמא

להלן היישום של הדיאגרמה לעיל

```
// An extensible program.
import java.util.*;

class Instrument
{
	public void play()
	{
		System.out.println("Instrument.play()");
	}
	public String what()
	{
		return "Instrument";
	}
	public void adjust() {}
}

class Wind extends Instrument
{
	public void play()
	{
		System.out.println("Wind.play()");
	}
	public String what() { return "Wind"; }
	public void adjust() {}
}

class Percussion extends Instrument
{
	public void play()
	{
		System.out.println("Percussion.play()");
	}
	public String what() { return "Percussion"; }
	public void adjust() {}
}

class Stringed extends Instrument
{
	public void play()
	{
		System.out.println("Stringed.play()");
	}
	public String what() { return "Stringed"; }
	public void adjust() {}
}

class Brass extends Wind
{
	public void play()
	{
		System.out.println("Brass.play()");
	}
	public void adjust()
	{
		System.out.println("Brass.adjust()");
	}
}

class Woodwind extends Wind
```

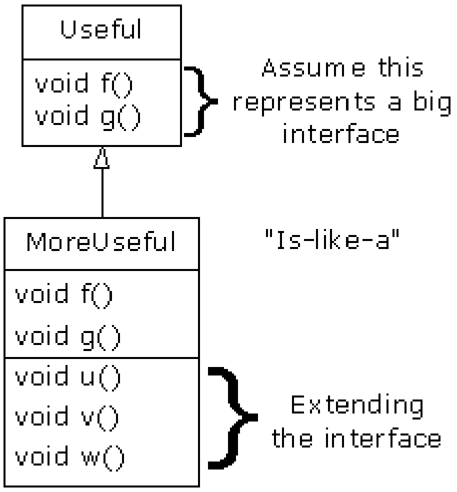
```
{
	public void play()
	{
		System.out.println("Woodwind.play()");
	}
	public String what() { return "Woodwind"; }
}

public class Music3
{
	// Doesn't care about type, so new types
	// added to the system still work right:
	static void tune(Instrument i)
	{
		// ...
		i.play();
	}
	static void tuneAll(Instrument[] e)
	{
		for(int i = 0; i < e.length; i++)
			tune(e[i]);
	}
	public static void main(String[] args)
	{
		Instrument[] orchestra = new Instrument[5];
		int i = 0;
		// Upcasting during addition to the array:
		orchestra[i++] = new Wind();
		orchestra[i++] = new Percussion();
		orchestra[i++] = new Stringed();
		orchestra[i++] = new Brass();
		orchestra[i++] = new Woodwind();
		tuneAll(orchestra);
	}
}
```

הסבר התכנית

הפונקציות החדשות הנן ()what אשר מחזירות לנו ערכי String עם התייחסות לפונקצית ()adjust.
בפונקצית ()main כאשר אנו נשים ערך מסויים בתוך מערך ה Instrument באופן אוטומטי יתבצע Upcasting.

8.4. מודל Downcasting

מאחר שאנו "מאבדים" את סוג המידע ברגע שאנו משתמשים ב Upcasting (קרי תזוזה כלפי מעלה בהיראכיה של ההורשה) על מנת שנוכל לקבל את המידע שאנו רוצים לגבי אובייקט מסוים ז"א תזוזה חזרה בתוך הורשה (=כלפי מטה) אזי כי נשתמש בתכונת downcasting.
אולם למרות שאנו יודעים כי תכונת upcast הנה בטוחה מחלקת העל איננה יכולה לקבל ממשק גדול יותר מתת המחלקה לפיכך כל הודעה שנעביר דרך ההורשה הנה "הודעה בטוחה".
אך בהשוואה ל downcast אנו איננו יודעים שהצורה (בדוגמא שלנו) היא אכן עיגול ומכאן זה אומר לנו כי קיימת האפשרות שתהיה כל צורה.



על מנת לפתור את הבעיה הנ"ל נוכל להניח כי קיימת אבטחת מידע שפעולת ה downcast תתבצע כראוי. בשפות כמו C++ אנו חייבים להצע פעולה מסוימת על מנת לקבל את סוג ה downcast אבל ב java כל ביצוע של cast נבדק.

אם כן למרות שזה נראה כאילו אנו עושים פעולה סטנדרטית לחלוטין אזי כי בזמן RTTI run-time type identification ה Cast נבדק.

דוגמא

```
// Downcasting & Run-time Type
// Identification (RTTI).
import java.util.*;

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
}
```

הסבר התכנית

כפי שאנו רואים בדיאגרמה מחלקת MoreUseful הנה מורישה ממחלקת Useful. מאחר שהיא מורישה מחלקה זו יכולה לבצע upcast למחלקת Usful. אנו רואים זאת בתכנית שאנו מאתחלים את מערך X בפונקציית main(). מאחר ששתי האובייקטים במערך הנם ממחלקת Useful נוכל לשלוח את פונקציות f() ו g() לשניהם, ואם ננסה לקוא לפונקציית u() נקבל טעות (פונקציה זו נמצאת במחלקת MoreUseful).

8.5. שכתוב VS טעינה

הבא נבחן את שתי הפולות שכתוב כנגד טעינה (המדובר הוא על פונקציות). בתכנית הבאה ההתייחסות לפונקצית play() משתנה על ידי שכתוב הפונקציה ז"א לא נשכתב את הפונקציה אלא נטעין אותה מחדש.

```
// Accidentally changing the interface.

class NoteX
{
	public static final int
		MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX
{
	public void play(int NoteX)
	{
		System.out.println("InstrumentX.play()");
	}
}

class WindX extends InstrumentX
{
	// OOPS! Changes the method interface:
	public void play(NoteX n)
	{
		System.out.println("WindX.play(NoteX n)");
	}
}

public class WindError
{
	public static void tune(InstrumentX i)
	{
		// ...
		i.play(NoteX.MIDDLE_C);
	}
	public static void main(String[] args)
	{
		WindX flute = new WindX();
		tune(flute); // Not the desired behavior!
	}
}
```

תוצאת התכנית

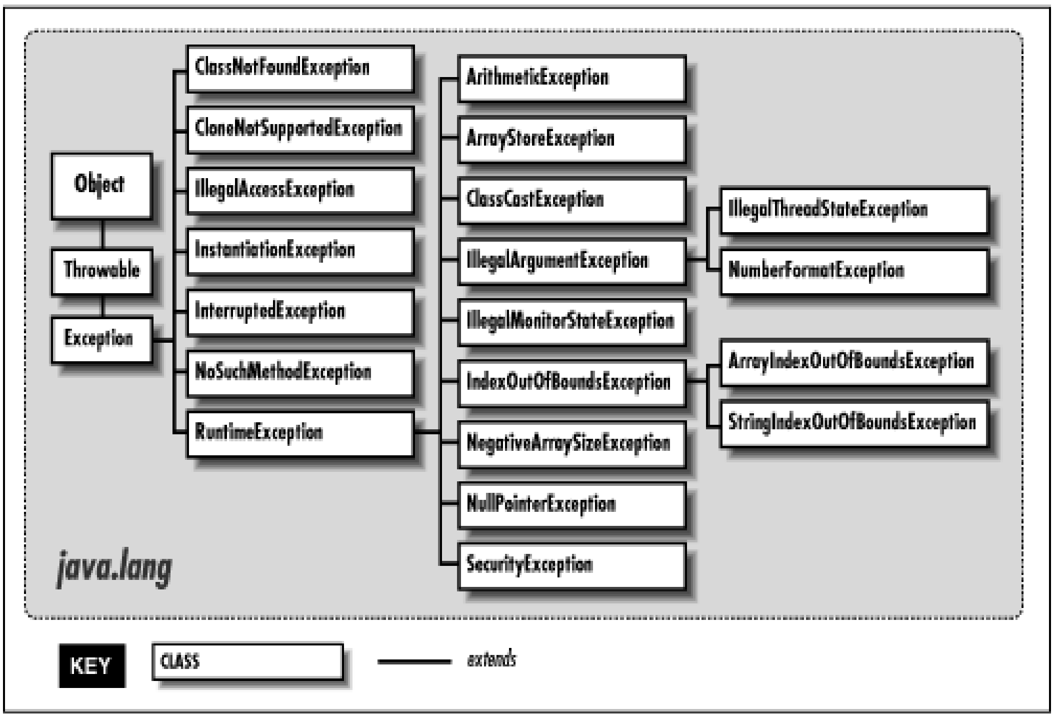
```
	InstrumentX.play()
```

9. Exception Handling – טיפול ביוצאי דופן

הזמן האידיאלי ביותר לתפוס טעויות הנו בזמן הקומפילציה אפילו לפני שאנו מריצים את התכנית. אולם לא כל הטעויות יכולות להיות מטופלות בזמן הקומפילציה ולכן נטפל בהם בזמן ריצת התכנית.
בהתייחסות לשפת C ולשפות דומות קיימות מספר דרכים על מנת לבצע טיפול ביוצאי הדופן ובטעויות אם כי חלק זה איננו נכלל באותם שפות כקונספט בפני עצמו.
בעיקר הטיפול בטעויות בשפות מסוג זה הנו דרך flag או ערכים המקימים תנאים מסוימים. תכונה זו של הוספת קוד לצורך בדיקות מהווה העמסה על הקוד הקיים ובסופו של דבר מסרבלת את התהליך.
אי לכך טיפול בטעויות מוכר בשפות תכנות רבות והקונספט כבר הוצג בשפות כגון C++, Pascal וכו'.
המילה exception כוונתה בקונטקסט הנידון "שאנו מבצעים יוצא דופן" קרי הטיפול באותם יוצאי דופן יוצר לנו מצב בו אנו מבצעים "ניקיון" בקוד המכיל טעויות. בנוסף, הטיפול בטעויות הנו כללי קרי אותה טעות נקודתית המטופלת תהיה מטופלת לאורך כלל התכנית באם קיימת אותה טעות מסוג זהה.
מכיוון שטיפול ביוצאי הדופן ובטעויות הנו כלי שהקומפיילר "מכריח" אותו לעבוד אזי כי נוכל להגיע לרמת נורמליזציה בקוד.
כל סוגי הטעויות הקיימים בקומפיילר הנם תת מחלקה של מחלקת העל Throwable. לפיכך ניתן לומר כי Throwable הנה הרמה הראשונה בהיררכיה של טיפול ביוצאי הדופן. מבנה ה Throwable הנו משתי תתי מחלקות אשר חולקים את הטעויות לתוך אזור ייחודי אחד.
בשפת התכנות java קיימים חמישה רכיבים עיקריים לטיפול ביוצאי דופן:

1. Try
2. Catch
3. Throw
4. Throws
5. Finally

הצורה הכללית של המחלקה הנה:



9.1. שימוש ב try

אם אנחנו נמצאים בתוך פונקציה ואנו רוצים לזרוק טעות מסוימת הפונקציה תפסיק את פעולתה כאשר אנו מבצעים פעולה של טיפול ביוצאי דופן. ובמידה ואיננו רוצים שתהליך זה יתבצע אזי כי נצטרך לכתוב בלוק קוד מיוחד בתוך הפונקציה ואשר יטפל ביוצא הדופן.
בלוק זה נקרא try בלוק מכיוון שאנו מנסים לקרוא למספר פונקציות בבלוק זה. את הבלוק הזה נרשום בעזרת שימוש במילת המפתח try.
דוגמא

```
try {
  // Code that might generate exceptions
}
```

באם נבדוק טעויות תוך כדי תכנות אשר אינן תומכות בתכונה זו של טיפול בטעויות נצטרך לבצע מספר שאילתות כל כך גדול אחרי כל פונקציה כך שניצור סרבול בקוד.
כאשר נשתמש בטיפול בטעויות אנו נשים הכל בתוך try בלוק שינסה לתפוס את יוצאי הדופן בקוד.
התוצר הנו קוד קל יותר לכתיבה ולמעקב.

9.2. שימוש ב catch

כמובן שאותה "זריקה" של קוד חייבת להסתיים היכן שהוא. "היכן שהוא" הנו מוגדר כמטפל ביוצא הדופן וקיים אחד שכזה לכל יוצא דופן שאנו רוצים "לתפוס". על מנת ל"תפוס" טעות נשתמש במילת המפתח catch.
נשים לב כי השימוש ב try וב catch מהווים לנו יחידה אחת.

הבא נבחן את הדוגמא הבאה:

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}
```

כל יחידה של catch הנה כמו פונקציה הלוקחת אך ורק ארגומנט אחד לטפל בו מסוג מסוים. המזהים identifiers המצויים בתוך הסוגריים של catch הנם בעצם הפרמטרים לזיהוי של ארגומנט ה catch.
אופי העבודה מאוד פשוט:
המטפל בטעות חייב להופיע מייד אחרי ה try בלוק. במידה וטעות נזרקת הטכניקה של טיפול בטעויות יוצאת "לצוד" את המטפל הראשון אם ארגומנט שמתאים לטעות. רק לאחר מכן הוא נכנס להצהרות ה catch ואז יוצא הדופן מיוחס במערכת כמטופל. החיפוש אחר המטפל בטעות הנכון נמשך עד אשר יימצא אחד מתאים במידה ונמצא הלופ נשבר באופן אוטומטי.

try & catch דוגמא לבלוק

```
//	Try and Catch clause
import java.util.Random;

class HandleError
{
	public static void main (String args[])
	{
		int a=0, b=0, c=0;
		Random r = new Random();

		for (int i=0; i<1000; i++)
		{
			try
			{
				b = r.nextInt();
				c = r.nextInt();
				a = 12345 / (b/c);
			}
			catch (ArithmeticException e)
			{
				System.out.println("Divide by Zero");
				a = 0; //set a to zero and continue
			}
			System.out.println("a:" + a);
		}
	}
}
```

9.3 שימוש ב throw

תנאי של יוצא דופן בסיסי הוא בעצם התנאי שמונע את המשך הטעות של הפונקציה או של חלק קוד מסוים. חשוב מאוד להבדיל בין תנאי שהנו יוצא דופן לבין תנאי שהנו קוד רגיל.
דוגמא פשוטה הנה פעולת חילוק. באם נרצה לבצע חילוק ב 0 אזי כי נצטרך לבדוק כי אכן אנו יכולים לבצע זאת.
כאשר אנו משתמשים ב throw מספר תהליכים מתבצעים:

1. דבר ראשון האובייקט היוצא דופן נוצר באותו אופן בו נוצרים אובייקטים ב java תוך שימוש ב new.
2. המסלול של ריצת הקוד נעצר וההתייחסות לאובייקט שהנו יוצא הדופן מוצא מחוץ לבלוק הקוד.
3. בנקודה זו טכניקת הטיפול ביוצא הדופן לוקחת שליטה על התכנית ומחפשת את הטיפול המתאים לאותה טעות.
4. המיקום המתאים שנמצא הוא בעצם ה exception handler שעבודתו הנה לטפל בבעיה כגון להמשיך את התכנית או להעביר את הבעיה הלאה.

בתור דוגמא קצרה הבא נבחן אובייקט הקרוי t. קיימת האפשרות שהעברנו התייחסות לאותו אובייקט שעדיין לא אותחל אזי נרצה לבדוק את ההתייחסות לפני שנקרא לפונקציה לרוץ. נוכל לשלוח את המידע על טעות זו לתוך מכלול רחב של יוצאי דופן ל ידי שניצור אובייקט המייצג את המידע הזה ואז "לזרוק" את המידע מחוץ לאותו מכלול. פעולה זו נקראת throwing an exception.

```
if(t == null)
  throw new NullPointerException();
```

פעולה זו "זורקת" את יוצא הדופן תוך התייחסות לאובייקט שנוצר.

דוגמא

```
//    Throws clause
class ThrowDemo
{
      static void demoproc()
      {
            try
            {
                  throw new NullPointerException("Demo");
            }
            catch (NullPointerException e)
            {

                  System.out.println("Inside throw");
                  throw e ;// rethrow the exception
            }
      }

      public static void main (String args[])
      {
            try
            {
                  demoproc();
            }
            catch (NullPointerException e)
            {
                  System.out.println("Re-Catch" + e);
            }
      }
}
```

9.4. שימוש ב throws

כל טעות או יוצא דופן שנרצה לזרוק בכל שלב שהוא בחיי התכנית אנו חייבים להודיע עליו לקומפיילר. במידה ופונקציה יכולה לגרום לטעות שהיא איננה יכולה לטפל בה היא חייבת להגדיר את התנהגותה של טעות זו כך שברגע שנקרא לפונקציה היא תהיה מוגנת מהטעות. ההודעה על הטיפול בטעות שכזו נעשה על ידי שימוש במילת המפתח throws ולאחריה מספר פרמטרים עם אותו פוטנציאל לטעות.
דוגמא

```
void f() throws TooBig, TooSmall, DivZero { //...
```

ובמידה ונאמר כי

```
void f() { // ...
```

אזי כי אנו אומרים ששום טעות איננה נזרקת מהפונקציה.

דוגמא

```
//    Throws clause
class ThrowsTest
{
      static void throwOne()throws IllegalAccessException
      {
            System.out.println("Inside throwOne");
            throw new IllegalAccessException("demo");
      }
      public static void main (String args[])
      {
            try
            {
                  throwOne();
            }
            catch (IllegalAccessException e)
            {
                  System.out.println("Catch" + e);
            }
      }
}
```

9.5. שימוש ב finally

כאשר יוצאי דופן נזרקים ריצת הפונקציות מתבצעת בצורה א-ליניארית כמובן תלוי כיצד הפונקציה נכתבה ולפעמים קיימים מצבים בהם יוצא הדופן ימשיך לרוץ ולגרום לופ אינסופי. דבר זה יכול להוות בעיה לדוגמא, במידה ופונקציה אמורה לפתוח קובץ מסוים ברגע שנכנסים לקובץ ולסגור אותו ברגע שיוצאים ממנו אזי לא נרצה שהקוד של טיפול ביוצאי הדופן יבצע עליו מבחן. אי לכך על ידי שימוש במילת המפתח finally נוכל לפתור בעיות מסוג זה.
Finally יוצרת לנו בלוק של קוד שירוץ אחרי ש try ו catch סיימו את פעולתם ולפני הקוד הבא לבלוק הבא של try ו catch.
Finally בלוק ירוץ ויוציא את התוצר לא משנה האם טעות נתפסה או לא.
במידה וטעות נתפסה אזי בלוק ה finally ירוץ למרות שלא נעשה כל שימוש ב catch התואם לטעות.
תהליך זה יכול להיות יעיל כאשר אנו נסגור קובץ כלשהו ונקפיא משאבים אחרים שיכולים להיות ממוקמים בתחילת הפונקציה עם כוונה להשתמש בהם לפני שהם מחזירים ערכים.
שימוש ב finally הנו אופציונאלי אולם כל הצהרה על try דורש לפחות הצהרה על catch או finally.

דוגמא

```java
//    Finally clause
class FinallyDemo
{
      static void throwA()
      {
            try
            {
                  System.out.println("Inside throwA");

                  throw new RuntimeException("Demo");
            }
            finally
            {
                  System.out.println("throwA finally");
            }
      }
      static void throwB()
      {
            try
            {
                  System.out.println("Inside throwB");
                  return;
            }
            finally
            {
                  System.out.println("throwB finally");
            }
      }
      static void throwC()
      {
            try
            {
                  System.out.println("Inside throwC");
            }
            finally
            {
                  System.out.println("throwC finally");
            }
      }

      public static void main (String args[])
      {
```

```
            try
            {
                  throwA();
            }
            catch (Exception e)
            {
                  System.out.println("Exception Caught");
            }
            throwB();
            throwC();
      }
}
```

9.6. ארגומנטים ביוצאי הדופן

כמו כל אובייקט ב java אנו תמיד ניצור יוצא דופן הנעזר באובייקט תוך שימוש באופרטור new אשר בעצם מאכלס לנו מקום וקורא לקונסטרקטור.
קיימים שני קונסטרקטורים ביוצאי הדופן הסטנדרטים:
1. Default constructor
2. String argument

דוגמא:

```
if(t == null)
  throw new NullPointerException("t = null");
```

שימוש במילת המפתח throw גורם למספר דברים להתבצע:
1. בד"כ קודם כל נשתמש באופרטור new על מנת לייצור אובייקט המייצג תנאי לטעות מסוג מסוים.
2. אנו נותנים את ההתייחסות לתשובה האפשרית ל throw
3. האובייקט למעשה "מוחזר" מתוך פונקציה אפילו אם אותו אובייקט איננו מתאים למה שהפונקציה אמורה לבצע.

בנוסף נוכל "לזרוק" כל אובייקט מסוג Throwable שנרצה. ובד"כ נבצע טיפול ביוצאי דופן למחלקות שונות בעלות טעויות שונות.
תכולת הטעות מיוצגת הן בקונטקסט שאנו מטפלים בו נקודתית והן נשלחת כמעין "הודעה" לכלל המחלקות כי קיים טיפול בטעות מסוימת.

9.7. יצירת יוצאי דופן משלנו

עד כה עסקנו בטיפול יוצאי הדופן הקיימים לנו ב java. לפעמים קיימים מצבים בהם נרצה אנו לייצור את אותם מטפלים ביוצאי הדופן שלנו.
על מנת לייצור את מטפלי הטעויות שלנו אנו מוכרחים לבצע הורשה מסוג קיים של מטפל ביוצא דופן, עדיף מטפל שיוצא דופן שיכול לענות לנו על הטעות שלנו (אם אפשר).
הדרך הטריוויאלית ביותר לייצור טיפול ביוצאי דופן חדש הנה לתת לקומפיילר לייצור default constructor עבורנו כך שלא נצטרך קוד רב לשימוש.

דוגמא

```
// Inheriting your own exceptions.
class SimpleException extends Exception {}

public class SimpleExceptionDemo {
  public void f() throws SimpleException {
System.out.println("Throwing SimpleException from f()");
    throw new SimpleException ();
  }
  public static void main(String[] args) {
    SimpleExceptionDemo sed =
      new SimpleExceptionDemo();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.err.println("Caught it!");
    }
  }
}
```

נזכור דבר חשוב – בד"כ הדבר החשוב ביותר לטיפול ביוצאי דופן כאשר אנו יוצרים אותם הנה שם המחלקה לה הם שייכים. הדוגמא לעיל הנה בהחלט מספקת לצורך ביצוע בדיקות.
בתכנית לעיל התוצאה תודפס ל console standard error ולו System.err ולא ל System.out ספרייה זו בעצם מטפלת בהדפסת הטעיות.

יצירת מחלקה לטיפול ביוצא דופן אשר לה קיים קונסטרקטור הלוקח String יתבצע באופן הבא:

```
// Inheriting your own exceptions.

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}

public class FullConstructors {
  public static void f() throws MyException {
    System.out.println(
      "Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println(
      "Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.err);
    }
  }
}
```

הקוד שהוספנו הנו מועט - הוספת שני הקונסטרקטורים המגדירים את הדרך ליצירת MyException.
בקונסטרקטור השני אנו משתמשים ב super.
מידע המעקב אחר הטיפול נשלח אל **System.err**.

תוצאת התכנית הנה

```
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:16)
        at
FullConstructors.main(FullConstructors.java:24)
Throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:20)
        at
FullConstructors.main(FullConstructors.java:29)
```

נוכל כמובן להוסיף קונסטרקטורים נוספים וחברים נוספים למחלקה:

```
// Further embellishment of exception classes.

class MyException2 extends Exception {
  public MyException2() {}
  public MyException2(String msg) {
    super(msg);
  }
  public MyException2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
  private int i;
}

public class ExtraFeatures {
  public static void f() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from h()");
    throw new MyException2(
      "Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace(System.err);
      System.err.println("e.val() = " + e.val());
    }
  }
}
```

הסבר

הוספנו משתנה I אשר ביחד עם הפונקציה והקונסטרקטור מבצע את פעולתו.

תוצאת התכנית הנה:

```
Throwing MyException2 from f()
MyException2
        at ExtraFeatures.f(ExtraFeatures.java:22)
        at ExtraFeatures.main(ExtraFeatures.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
        at ExtraFeatures.g(ExtraFeatures.java:26)
        at ExtraFeatures.main(ExtraFeatures.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
        at ExtraFeatures.h(ExtraFeatures.java:30)
        at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
```

מאחר שטיפול בטעיות הנה עוד סוג של אובייקט נוכל כמובן להרחיב את מגוון הטיפול ביוצאי הדופן.
נזכור כי שאר המתכנתים המשתמשים בקוד שלנו בד"כ רוצים לראות את הטיפול בטעויות ולא יותר.

9.8. תפיסת כל טעות

יש באפשרותנו להגדיר handler אשר תהיה באפשרותו לתפוס כל סוג של טעות. אנו נבצע זאת על ידי שימוש במחלקה הבסיסית הקיימת ששמה Exception.

דוגמא

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

הצהרה זו תתפוס כל יוצא דופן.
מאחר שמחלקה **Exception** הנה הבסיס לכלל יוצאי הדופן נוכל להשתמש בפונקציות מובנות הקיימות בקומפיילר ושהן מתחת מחלקת Throwable:

String getMessage( )
String getLocalizedMessage(**)** **-**Gets the detail message, or a message adjusted for this particular locale
String toString( ) - Returns a short description of the Throwable, including the detail message if there is one.
void printStackTrace( )
void printStackTrace(PrintStream)
void printStackTrace(PrintWriter)
Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown. The first version prints to standard error, the second and third prints to a stream of your choice (in Chapter 11, you'll understand why there are two types of streams).
Throwable fillInStackTrace( )
Records information within this **Throwable** object about the current state of the stack frames. Useful when an application is rethrowing an error or exception (more about this shortly).

In addition, you get some other methods from **Throwable**'s base type **Object** (everybody's base type). The one that might come in handy for exceptions is **getClass( )**, which returns an object representing the class of this object. You can in turn query this **Class** object for its name with **getName( )** or **toString( )**. You can also do more sophisticated things with **Class** objects that aren't necessary in exception handling.

להלן דוגמא המראה כיצד השימוש בפונקציות ה Exception מתבצע:

```
// Demonstrating the Exception Methods.

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("Here's my Exception");
    } catch(Exception e) {
      System.err.println("Caught Exception");
      System.err.println(
        "e.getMessage(): " + e.getMessage());
      System.err.println(
        "e.getLocalizedMessage(): " +
         e.getLocalizedMessage());
      System.err.println("e.toString(): " + e);
      System.err.println("e.printStackTrace():");
      e.printStackTrace(System.err);
    }
  }
}
```

תוצאת התכנית

```
Caught Exception
e.getMessage(): Here's my Exception
e.getLocalizedMessage(): Here's my Exception
e.toString(): java.lang.Exception:
   Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
 at ExceptionMethods.main(ExceptionMethods.java:7)
java.lang.Exception:
   Here's my Exception
 at ExceptionMethods.main(ExceptionMethods.java:7)
```

9.9. זריקה נוספת של טעות

לפעמים קיימים מצבים בהם נצטרך לבצע זרקה נוספת של טיפול בטעות במיוחד אם נשתמש ב Exception על מנת לתפוס טעות. מאחר שכבר קיימת לנו התייחסות במערכת לגבי הטעות אנו בפשטות יכולים לבצע זריקה נוספת.

דוגמא

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
}
```

זריקה מחודשת של טעות גורמת לטעות ללכת ל handler ברמה הבאה בקוד (נזכור כי הטיפול בטעויות מתבצע באופן היראכי).

להלן דוגמא לזריקה נוספת של טעות:

```
// Demonstrating fillInStackTrace()

public class Rethrowing {
  public static void f() throws Exception {
System.out.println("originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.err.println(
        "Inside g(), e.printStackTrace()");
      e.printStackTrace(System.err);
      throw e; // 17 The important line
      // throw e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String[] args) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.err.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace(System.err);
    }
  }
}
```

תוצאת התכנית

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
```

כאשר נסיר את ההערה משורות 17 ו 18 אזי כי נשתמש בפונקציית **fillInStackTrace( )**
והתשובה תהיה:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.g(Rethrowing.java:18)
        at Rethrowing.main(Rethrowing.java:24)
```

מאחר ששורה 18 הנה הנקודה החדשה לטיפול בטעות.

.9.10 יוצאי דופן מובנים במערכת

Runtime exceptions	
ArithmeticException	This exception is thrown to indicate an exceptional arithmetic condition, such as integer division by zero.
ArrayIndexOutOfBoundsException	This exception is thrown when an out-of-range index is detected by an array object. An out-of-range index occurs when the index is less than zero or greater than or equal to the size of the array.
ArrayStoreException	This exception is thrown when there is an attempt to store a value in an array element that is incompatible with the type of the array.
ClassCastException	This exception is thrown when there is an attempt to cast a reference to an object to an inappropriate type.
IllegalArgumentException	This exception is thrown to indicate that an illegal argument has been passed to a method
IllegalMonitorStateException	This exception is thrown when an object's wait(), notify(), or notifyAll() method is called from a thread that does not own the object's monitor.
IllegalStateException	This exception is thrown to indicate that a method has been invoked when the run-time environment is in an inappropriate state for the requested operation. This exception is new in Java 1.1.
IllegalThreadStateException	This exception is thrown to indicate an attempt to perform an operation on a thread that is not legal for the thread's current state, such as attempting to resume a dead thread.
IndexOutOfBoundsException	The appropriate subclass of this exception (i.e., ArrayIndexOutOfBoundsException or StringIndexOutOfBoundsException) is thrown when an array or string index is out of bounds
NegativeArraySizeException	This exception is thrown in response to an attempt to create an array with a negative size
NullPointerException	This exception is thrown when there is an attempt to access an object through a null object reference. This can occur when there is an attempt to access an instance variable or call a method through a null object or when there is an attempt to subscript an array with a null object
NumberFormatException	This exception is thrown to indicate that an attempt to parse numeric information in a string has failed.
RuntimeException	The appropriate subclass of this exception is thrown in response to a runtime error detected at the virtual machine level. Because these exceptions are so common, methods that can throw objects that are instances of RuntimeException or one of its subclasses are not required to declare that fact in their throws clauses
SecurityException	This exception is thrown in response to an attempt to perform an operation that violates

	the security policy implemented by the installed SecurityManager object
StringIndexOutOfBoundsException	This exception is thrown when a String or StringBuffer object detects an out-of-range index. An out-of-range index occurs when the index is less than zero or greater than or equal to the length of the string

Other exceptions	
ClassNotFoundException	This exception is thrown to indicate that a class that is to be loaded cannot be found
CloneNotSupportedException	This exception is thrown when the clone() method has been called for an object that does not implement the Cloneable interface and thus cannot be cloned
Exception	The appropriate subclass of this exception is thrown in response to an error detected at the virtual machine level. If a program defines its own exception classes, they should be subclasses of the Exception class
IllegalAccessException	This exception is thrown when a program tries to dynamically load a class (i.e., uses the forName() method of the Class class, or the findSystemClass() or the loadClass() method of the ClassLoader class) and the currently executing method does not have access to the specified class because it is in another package and not public. This exception is also thrown when a program tries to create an instance of a class (i.e., uses the newInstance() method of the Class class) that does not have a zero-argument constructor accessible to the caller
InstantiationException	This exception is thrown in response to an attempt to instantiate an abstract class or an interface using the newInstance() method of the Class class
InterruptedException	This exception is thrown to signal that a thread that is sleeping, waiting, or otherwise paused has been interrupted by another thread
NoSuchFieldException	This exception is thrown when a specified variable cannot be found. This exception is new in Java 1.1
NoSuchMethodException	This exception is thrown when a specified method cannot be found

10. קלט/פלט ב – Java

עד כה ראינו שימוש בפונקצית print אשר בעצם יצרה לנו פעולה של הדפסה למסך ובעצם לא ראינו יצד אנו מערבים בניית אפליקציות הרצות ב console כמו שפות ++C, C. הסיבה לכך פשוטה. מכיוון שרוב האפליקציות הכתובות ב java יהיו מעורבות עם אפליקציות אינטרנט.
אולם על מנת לתקשר עם רכיבים חיצוניים עלינו להבין את עומקה של סוגייא זו ובחלק זה נבחן שימוש נוסף בפונקציות וזאת על מנת קבל תמונה שלמה על שפת java.

10.1. סטרים – Streams

סטרים מוגדר כמבנה אשר מבצע אחד משתי הפעולות או מייצר או מקבל מידע. סטרים מקושר למכשיר פיזי על ידי java I/O. כלל הסטרימים מתנהגים באופן זההגם במקרה שבו המכשירים המחוברים אליו שונים. ולפיכך מבחינת java אותו מבנה מחלקות ופונקציות יכולות להיות מיושמות בכלל האפליקציות.
ומכאן לצורך הדיון ניתן לומר כי input stream יכול לייצור לנו סוגים שונים של input קרי קובץ בדיסק, מקלדת, או תקשורת לרשת.
כמו גם output stream יכול להתייחס לסוגים שונים קרי console, קובץ דיסק, או תקשורת לרשת.
Streams מוגדרים כ"דרך נקיה" על מנת לבצע פעולות I/O ללא הגדרה בכל פעם מחדש בקוד שלנו.

להלן טבלה המכילה את פונקציות ה input stream ופעולותיהם:

CLASS	FUNCTION	CONSTRUCTOR ARGUMENTS
ByteArray-InputStream	Allows a buffer in memory to be used as an InputStream	The buffer from which to extract the bytes.
As a source of data. Connect it to a FilterInputStream object to provide a useful interface.		
StringBuffer-InputStream	Converts a String into an InputStream	A String. The underlying implementation actually uses a StringBuffer.
As a source of data. Connect it to a FilterInputStream object to provide a useful interface.		
File-InputStream	For reading information from a file	A String representing the file name, or a File or FileDescriptor object.
As a source of data. Connect it to a FilterInputStream object to provide a useful interface.		
Piped-InputStream	Produces the data that's being written to the associated PipedOutput-Stream. Implements the "piping" concept.	PipedOutputStream
As a source of data in multithreading. Connect it to a FilterInputStream object to provide a useful interface.		
Sequence-InputStream	Converts two or more InputStream objects into a single InputStream.	Two InputStream objects or an Enumeration for a container of InputStream objects.
As a source of data. Connect it to a FilterInputStream object to provide a useful interface.		
Filter-InputStream	Abstract class which is an interface for decorators that provide useful functionality to the other InputStream classes.	

להלן טבלה המכילה את פונקציות ה output stream ופעולותיהם:

CLASS	FUNCTION	CONSTRUCTOR ARGUMENTS
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.	Optional initial size of the buffer.
To designate the destination of your data. Connect it to a FilterOutputStream object to provide a useful interface.		
File-OutputStream	For sending information to a file.	A String representing the file name, or a File or FileDescriptor object.
To designate the destination of your data. Connect it to a FilterOutputStream object to provide a useful interface.		
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated PipedInput-Stream. Implements the “piping” concept.	PipedInputStream
To designate the destination of your data for multithreading. Connect it to a FilterOutputStream object to provide a useful interface.		
Filter-OutputStream	Abstract class which is an interface for decorators that provide useful functionality to the other OutputStream classes. See Table 11-4.	See Table 11-4.

10.2. מחלקת File

למרות שכלל המחלקות מוגדרות על ידי java.io מחלקת File איננה מוגדרת. מחלקה זו פועלת ישירות על קבצים וקבצי מערכת. ולפיכך file איננו מגדיר כיצד הוא מקבל או מוציא מידע מקבצים מאוכלסים, מטרתו הנה לתאר את מאפייני הקובץ. מאחר באובייקט file משמש אותנו על מנת לקבל או להוציא מידע הקשור לקבצי דיסק כגון הרשאות, זמן, תאריך, מסלולי ספריות וכו' ומאחר ואיננו פועל על stream הוא איננו תת מחלקה של שתי המחלקות שהוזכרו.
קבצים מוגדרים כיעד וכמטרה לנתונים בתוך התכנית למרות שישנם הנחיות מפורשות ברמת אבטחת מידע כאשר אנו משתמשים בהם ב applet קבצים הם עדיין העיקר בתכניות מחשב.
ספרייה ב java מיוחסת כ file עם מאפיין נוסף שהנו שם הקובץ. נוכל לקבל את שם הקובץ תוך שימוש בפונקצית ()list.
המבנה הכללי הנו:

File(String directoryPath)

File(String directoryPath, String filename)

File(File dirObject, Dtring fileName)

כאשר:
directoryPath - נותן לנו את מסלול הקובץ.
filename - שם הקובץ
dirObject – הנו אובייקט מסוג file המגדיר את הספרייה.

דוגמא
בדוגמא זו ניצור 3 קבצים. הקובץ הראשון הנו אובייקט מסוג File והוא נבנה עם מסלול הספרייה שבה הוא נמצא. הקובץ השני כולל שני ארגומנטים האחד מסלולו והאחר שם הקובץ. הקובץ השלישי כולל את המסלול מקובץ f1 .

```
File f1 = new File("/");
File f2 = new File("/", "autoexec.bat");
File f3 = new File(f1, "autoexec.bat");
```

File מגדיר הרבה פונקציות העוזרות לנו לקבל את המאפיינים של אובייקט מסוג File. לדוגמא פונקצית ()getFile מחזירה לנו את שם הקובץ. פונקצית ()getParent מחזירה לנו את השם של ספריית האב ופונקצית ()exist מחזירה ערך true באם הקובץ נמצא אחרת היא תחזיר ערך false.
מחלקת File איננה סימטרית, ומכאן זה אומר כי קיימות לנו הרבה פונקציות בהם נבחן את מאפייני הקבצים.

דוגמא

בדוגמא זו נשתמש בפונקציות מובנות הקיימות במחלקת File.

```
//demonstrate of File class import java.io.File;
import java.io.File;

class FileTest
{
	static void P(String s)
	{
		System.out.println(s);
	}
	public static void main(String args[])
	{
		File f1 = new File("/java/COPYRIGHT");
		P("File name: " + f1.getName());
		P("Path:" + f1.getPath());
		P("Abs Path:" + f1.getAbsolutePath());
		P("Parent:" + f1.getParent());
		P(f1.exists() ? "exists" : "Does not exist");
		P(f1.canWrite() ? "is Write" : "is not write");
		P(f1.canRead() ? "is read" : "is not read");
		P("is " + (f1.isDirectory() ? " " : "might be a named 
pipe"));
		P(f1.isFile() ? "is normal file" : "is not normal");
		P(f1.isAbsolute() ? "is absolute" : "is not");
		P("File last Modified: " + f1.lastModified());
		P("File Size: " + f1.length()+ "Bytes");
	}
}
```

בנוסף מחלקת File מכילה שתי פונקציות אשר אינן יכולות להיקרא מספריות אלא רק בפורמטים של קבצים ייחודיים. להלן מבנה הקבצים:

boolean renameTo(File newName)

כאשר:
השם המוגדר בפונקציה הופך להיות שם הקובץ.הפונקציה תחזיר ערך true בזמן שהפעולה התבצעה וערך false בזמן שלא הייתה קיימת גישה לספריה לצורך שינוי שם הקובץ.

boolean delete()

פונקציה זו פועלת רק על קבצי אוביקטים פשוטים. היא איננה יכולה למחוק אפילו ספרייה ריקה. שוב על אותו רעיון של החזרת ערכים true באם מחיקה התבצעה וערך false באם אחרת.

10.3. ספריות – Directories

ספרייה הנה קובץ המכיל רשימה של קבצים אחרים וספריות אחרות. כאשר אנו יוצרים אובייקט מסוג File ואנו מתייחסים אלו כספריה פונקצית isDirectory() תחזיר ערך true. במקרה זהנוכל לקרוא לפונקצית list() על מנת לבנות את רשימת תכולת הקבצים.
קיימים שני מבנים לשימוש:

String[] list()

רשימת הקבצים מוחזרת אלינו כמערך של אובייקט string .

דוגמא

```
//Using directories

import java.io.File;

class DirFile
{
	public static void main(String args[])
	{
		String dirname = "/java";
		File f1 = new File(dirname);

		if(f1.isDirectory())
		{
			System.out.println("Directory od" + dirname);
			String s[] = f1.list();

			for(int i=0; i<s.length; i++)
			{
				File f = new File(dirname + "/" + s[i]);
				if(f.isDirectory())
				{
				System.out.println(s[i] + " is a directory");
				}
				else {
					System.out.println(s[i] + "is a file");
			}
		}
	}
	else
{
	System.out.println(dirname + "is not a directory");
		}
	}
}
```

10.3.1. שימוש בפילטר שמות

לעיתים נרצה להגביל את מספר הקבצים המופיעים לנו ברשימה תוך שימוש בפונקצית list() ולכלול רק את אותם קבצים שאנו רוצים למצוא. על מנת לבצע זאת נשתמש בתכונת פילטר. המבנה הכללי הנו:

String[] list(FilenameFilter FFObject)

כאשר:
FFObject – הנו האובייקט המיישם את FilenameFilter.

FilenameFilter – מגדיר רק פונקציה אחת accept() אשר קוראת רק פעם אחת לכל קובץ ברשימה.
המבנה הכללי הנו:

boolean accept(File directory, String filename)

הפונקציה מחזירה ערך של true במידה ושם הקובץ נמצא ברשימה אחרת היא תחזיר ערך false.

מחלקת OnlyExt מיישמת את FilenameFilter. שימושה הנו להראות את הקבצים המתאימים לרשימה שאנו שולפים תוך שימוש בפונקצית list().

דוגמא
דוגמא זו איננה עובדת אך היא מראה לנו את השימוש במבנה הקונסטרקטורים.

```
import java.io.*;

public class OnlyExt implements FilenameFilter
{
	String ext;

	public OnlyExt(String ext)
	{
		this.ext = "." + ext;
	}

	public boolean accept(File dir, String name)
	{
		return nam.endWith(ext);
	}
}
```

דוגמא

בדוגמא זו נחפש את הקבצים עם סיומת html

```
//demonstrate of File class import java.io.File;
import java.io.*;

class DirListOnly
{
	public static void main(String args[])
	{
		String dirname = "/java";

		File f1 = new File(dirname);

		FilenameFilter only = new OnlyExt("html");
			String s[] = f1.list(only);

		for(int i=0; i<s.length; i++)
		{
			System.out.println(s[i]);
		}
	}
}
```

10.4. מחלקת Stream

היישום למידע כקלט/פלט מתבצע תוך שימוש בהיררכיה הנמצאת בחבילת java.io. ברמה העליונה של ההיררכיה קיימות שני מחלקות עיקריות InputStream ו OutputStream . בנוסף קיימים לנו מספר תת מחלקות שהנם הורשה של מחלקות אלו. רק מחלקות File, FileDescriptor, RandomAccessFile, StreamTokenizer אינם תת מחלקות של הנ"ל.

10.4.1. קלט – InputStream

מחלקה זו הנה abstract המגדירה מודל של קלט streaming. כלל הפונקציות של מחלקה זו "יזרקו" טעות מסוג IOException.

10.4.2. פלט – OutputStream

מחלקה זו הנה abstract המגדירה מודל של פלט streaming. כלל הפונקציות של מחלקה זו יחזירו ערך void ו"יזרקו" טעות מסוג IOException.

10.4.3. קובץ קלט – FileInputStream

מחלקת FileInputStream יוצרת לנו InputStream אשר יכול לשמש אותנו לקריאת תכולה של קובץ.
מבנה הקונסטרקטורים הנו:

FileInputStream(String filepath)

FileInputStream(File fileObject)

שתי הפונקציות הללו יכולות ל"זרוק" לנו טעויות מסוג FileNotFoundException.
כאשר:
Filepath – הנו המסלול לקובץ.
FileObject – הנו האובייקט המתאר את הקובץ.

כאשר אנו יוצרים קובץ מסוג FileInputStream קובץ זה פתוח לקריאה. FileInputStream משכתב שש מהפונקציות המובנות במחלקת InputStream , פונקציות ()mark ו ()reset אינן משוכתבות.

דוגמא

הדוגמא הבאה מראה כיצד נקרא ביט בודד, מערך ביטים, ומערך משתנה של ביטים. בנוסף נשתמש בפונקצית available() על מנת לקבוע את מספר הביטים הנותרים. כמו גם נשתמש בפונקצית skip() על מנת לקפוץ מעבר לביטים שאיננו רוצים.

```
//FileInputStream

import java.io.*;

class FileInputStreamTest
{
	public static void main(String args[])throws Exception
	{
		int size;
		InputStream f = new
FileInputStream("FileInputStreamTest.java");

		System.out.println("Total Bytes are:" + (size =
f.available()));

		int n =size/40;
		System.out.println("First" + n + "Bytes of the file one
read() at a time");
		for(int i=0; i<n; i++)
		{
			System.out.println((char) f.read());
		}

		System.out.println("\nstill Available: " +
f.available());
		System.out.println("Reading the next " +n + "with one
read(b[])");
		byte b[] = new byte[n];
		if(f.read(b) != n)
		{
			System.out.println("Can not read" + n + "bytes");
		}
		System.out.println("new String(b, 0, 0, n)");
		System.out.println("\nStill Avaliable: " + (size =
f.available()));
		System.out.println("Skkiping half of remaining bytes with
skip()");

		f.skip(size/2);
		System.out.println("Still Available: " + f.available());
		System.out.println("Reading" + n/2 + "into the end of
array");
		if(f.read(b, n/2, n/2) != n/2)
		{
			System.err.println("Can not read" + n/2 + "bytes");
		}
		System.out.println(new String(b, 0, 0, b.length));
		System.out.println("\nStill Available" + f.available());
		f.close();
	}
}
```

תוצאת התכנית

```
Total Bytes are:1291 First32Bytes of the file one read() at a time /
/ F i l e I n p u t S t r e a m



 i m p o r t   j a v a
still Available: 1259 Reading the next 32with one read(b[]) new
String(b, 0, 0, n)
Still Avaliable: 1227 Skkiping half of remaining bytes with skip()
Still Available: 614 Reading16into the end of array .io.*;  class
not read" + n +
Still Available598
```

10.4.4. קובץ פלט – FileOutputStream

מחלקת FileOutputStream יוצרת לנו OutputStream אשר יכול לשמש אותנו לכתיבה של קובץ.
מבנה הקונסטרקטורים הנו:

FileOutputStream(String filepath)

FileOutputStream(File fileObject)

שתי הפונקציות הללו יכולות ל"זרוק" לנו טעויות מסוג IOException או SecurityException.
כאשר:
Filepath – הנו המסלול לקובץ.
FileObject – הנו האובייקט המתאר את הקובץ.

כאשר אנו יוצרים קובץ מסוג FileOutputStream קובץ זה אינו תלוי בקובץ הקיים.
FileOutputStream ייצר את הקובץ לפני שהוא יפתח אותו לתוצאה. במקרה שנרצה לפתוח קובץ קריאה בלבלד תהיה לנו טעות שגיאה.
כלל הפונקציות מחזירות ערך void.

דוגמא

בדוגמא זו ניצור buffer פשוט המכיל string ואז נשתמש בפונקציית getBytes() על מנת לבנות את מערך הביטים בהתאמה. ואז ניצור 3 קבצים.

```
//FileOutputStream

import java.io.*;

class FileOutputStreamTest
{
	public static void main(String args[])throws Exception
	{
		String source = "Now its the time to exam\n"
			+ "the I/O feature\n"
			+"of the Java System";
		byte buf[] = new byte[source.length()];
		source.getBytes(0, buf.length, buf, 0);

		OutputStream f0 = new FileOutputStream("file1.txt");
		for (int i=0; i<buf.length; i+=2)
		{
			f0.write(buf[i]);
		}
		f0.close();

		OutputStream f1 = new FileOutputStream("file2.txt");
		f1.write(buf);
		f1.close();

		OutputStream f2 = new FileOutputStream("file3.txt");
		f2.write(buf, buf.length-buf.length/4, buf.length/4);
		f2.close();
	}
}
```

בנוסף לשימוש ב FileInputStream וב FileOutputStream קיימים שימושים גםב Byte שהנם יישום דומה ליישומים שראינו.

10.4.5. יישום StringBufferInputStream

StringBufferInputStream דומה לקודמיו קרי byte רק שכאשר אנו נשתמש בו אזי כי ה buffer הפנימי הנו מערך string ואין שום תגובה למחלקת Output.
מבנה הקונסטרקטור הנו:

StringBufferInputStream(String str)

בתוך הקונסטרקטור אנו נשתמש באובייקט מסוג String.

יישום BufferedStream

BufferedStream מאריך את השימוש במחלקת פילטר ה stream על ידי הוספה של memory buffer לתוך I/O Stream. נוכל לבצע זאת ללא שום בעיה מכיוון שעל ה buffer אפשרי לבצע את מכלול הפעולות.

10.4.6. יישום BufferedInputStream

אחת משיטות האופטימיזציה הנה שימוש ב I/O buffering. Java מאפשרת לנו לעטוף כל InputStream לתוך buffered stream ולקבל את המופע המשופר.
מבנה הקונסטרקטורים הנו:

BufferedInputStream(InputStream *inputStream*)

BufferedInputStream(InputStream *inputStream*, int *bufSize*)

כאשר:
הקונסטרקטור הראשון יוצר buffered stream לגודל של 521 ביטים.
הקונסטרקטור השני גודל ה buffer מועבר דרך משתנה מספרי.

דוגמא

בדוגמא זו נשתמש בפונקציית mark() על לזכור היכן הכנסנו את ה input stream ולאחר מכן נשתמש בפונקציית reset() על מנת לחזור לאותו מיקום.

```
//Buffered Input

import java.io.*;

class BufferedInputTest
{
	public static void main(String args[])throws IOException
	{
		String s = "Now its the time to exam &\n"
			+ "the I/O feature\n"
			+"of the Java & System";
		byte buf[] = new byte[s.length()];
		s.getBytes(0, s.length(), buf, 0);

		ByteArrayInputStream in = new ByteArrayInputStream(buf);
		BufferedInputStream f = new BufferedInputStream(in);
		int c;
		boolean marked = false;
		while ((c = f.read()) != -1)
		{
			switch(c)
			{
				case'&':
					if(!marked)
					{
						f.mark(32);
						marked = true;
					}
					else
					{
						marked = false;
					}
					break;
				case';':
					if(marked)
						marked = false;
					System.out.print((char) c);
					break;
				case' ':
					if(marked)
					{
						marked = false;
						f.reset();
						System.out.print((char) c);
						break;
						default:
						if(!marked)
							System.out.print((char) c);
						break;
					}
			}
		}
	}
```

10.5. גישה אקראית לקובץ

RandomAccessFile מרכז את הגישה האקראית לקבצים. מחלקה זו איננה נגזרת משתי המחלקות העיקריות עליהן דיברנו.
מחלקה זו מיישמת את הממשק של DataInput ושל DataOutput אשר מוגדרות ב I/O.
מבנה הקונסטרקטורים הנו:

RandomAccessFile(File fileObject, String access)

RandomAccessFile(String filename, String access)

כאשר
FileObject – מגידר לנו את שם הקובץ לגשיה.

בנוסף בקרת אבטחה על הקבצים הנה מסוג read, write, exe. בעזרת פונקצית seek() נוכל להצביע על מיקום הקובץ הנכתב.

דוגמא

דוגמא מסכמת זו תכלול שימוש ברכיבי הקלט והפלט. כמו גם תכנית זו תחשב לנו את מספר המילים הנכתבות על ידי פונקציה.

```
//Word Counting

import java.io.*;

class WordCount
{
	public static int words=0;
	public static int lines=0;
	public static int chars=0;

	public static void wc(InputStream f) throws IOException
	{
		int c=0;
		boolean lastNotWhite = false;

		String whiteSpace = "\t\n\r";

		while ((c = f.read()) != -1)
		{
			chars++;
			if(c== '\n')
			{
				lines++;
			}
			if(whiteSpace.indexOf(c) != -1)
			{
				if (lastNotWhite)
				{
					words++;
				}
				lastNotWhite = false;
			}
			else
			{
				lastNotWhite = true;
			}
		}
	}

	public static void main(String args[])
	{
		FileInputStream f;
		try
		{
			if(args.length ==0)
			{
				wc(System.in);
			}
			else
			{
				for(int i=0; i<args.length; i++)
				{
					f = new FileInputStream(args[i]);
					wc(f);
				}
			}
```

```
            }
            catch (IOException e)
            {
                  return;
            }
            System.out.println(lines + "" +words + "" +chars);
      }
}
```

11. טיפול ב String

כפי שמוגדר בשפות תכנות רבות ואחרות String הנו סט של תווים. בשפות כמו C ו ++C היישום של String הנו כמערך תווים המחזיק ערכי 0. ב Java ההגדרה שונה. היישום של String הנו בעצם על ידי אובייקט מסוג String.
הגישה דרך אובייקט מאפשרת לנו גמישות מרבית של טיפול ב String כמו אובייקט. אם כי כאשר אנו יוצרים אובייקט באופן מאוד מפתיע אנו יוצרים string שאיננו יכול לשנות את המאפיינים המייצגים את אותו ה string. במראה ראשוני זה נשמע קצת מפתיע אך זה לא המקרה, נוכל לשנות את ה string על ידי שימוש באופרטור new.
String הקבוע במערכת הנו שמאלי ולא ניתן לשינוי. הגישה הזו נלקחה מפאת הסיבה שהטיפול הנו קל יותר. אם כן נשאלת השאלה ומה יקרה באם נרצה להזיז את ה string?
אזי כי למקרים שכאלו נשתמש באובייקטים הנקראים StringBuffer. שהנם מכילים Strings אשר יכולים לבצע שינוי לאחר שנוצרו.
שתי המחלקות הללו הנם מוגדרות ב java.lang ולפיכך הנן מתאפשרות לשימוש בכל תכנית. שתי הפונקציות הללו מוגדרות כ final קרי אף אחת מהמחלקות הללו איננה יכולה להיות תת מחלקה בהורשה.

11.1 קונסטרקטור ל String

מחלקת string תומכת במספר קונסטרקטורים. על מנת לייצור string ריק אזי כי נשתמש במבנה הבא:

```
String s = new string();
```

הגדרה כזו תיצור לנו אובייקט מסוג String ללא מאפיינים.

על מנת לייצור String שנוכל לאתחל אותו במערך של תווים נשתמש במבנה הבא:

```
String (char chars[])
```

דוגמא

```
char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
```

קונסטרקטור שכזה מאתחל את s עם התווים abc.

נוכל להגדיר תת תחום של מערך תווים כמאתחל על ידי שימוש במבנה הבא:

```
String(char chars[], int startIndex, int numChars)
```

דוגמא

```
Char chars[]= {'a', 'b', 'c', 'd', 'e', 'f'};
String s = new String (chars, 2,3);
```

מבנה שכזה מאתחל לנו את s עם התווים cde.

למרות שהתווים ב java משתמשים ב 16 ביט על מנת להציג את ה Unicode הסוג הרגיל לפורמט של String על האינטרנט הנו שימוש ב 8 ביטים המורכבים מ ASCII. מפאת הסיבה שתווי ASCII על בסיס 8 ביטים מאוד נפוצים מחלקת String מספקת לנו את הקונסטרקטורים הבאים על מנת לאתחל string הנתון ממערך byte.

המבנה הכללי הנו

String(byte asciiChars[], byte HighOrderByte)
String(byte asciiChars[], bye HighOrderByte, int startIndex, int numChars)

כאשר:
AsciiChars – מגדיר את מערך ה bytes
HighOrderByte – מייצג את הערך של הסדר מהגבוה לכל תו.

נזכור כי עבור טקסט ASCII עלינו להגדיר את הערך 0 ל highOrderByte.

דוגמא
בדוגמא זו נראה כיצד אנו משתמשים בקונסטרקטורים.

```
//Construct String from subset of char array

class SubString
{
      public static void main (String args[])
      {
            byte ascii[] = {65, 66, 67, 68, 69, 70};

            String s1 = new String(ascii, 0);
            System.out.println(s1);
            String s2 = new String(ascii, 0, 2, 3);
            System.out.println(s2);
      }
}
```

תוצאת התכנית

ABCDEF
CDE

נוכל בנוסף לבנות אובייקט מסוג String אשר יכיל את אותו רצף תווי של אובייקט אחר מסוג String.
המבנה הכלי הנו:

String (String strObject)

דוגמא

```
//Construct String from subset of char array

class SubString
{
      public static void main (String args[])
      {
            char c[] = {'J','A','V','A'};

            String s1 = new String(c);
            System.out.println(s1);
            String s2 = new String(s1);
            System.out.println(s2);
      }
}
```

תוצאת התכנית

JAVA
JAVA

11.2. אורך ה String

אורך ה string נקבע על פי מספר התווים שהוא מכיל. על מנת לדעת מה אורך התווים אנו נשתמש בפונקציית length()
המבנה הכללי הנו

int length()

דוגמא

```
Char chars[] = {'a','b','c'};
String s = new String(chars);
System.out.println(s.length());
```

.11.3 שרשור Strings

בכללי java איננה מאפשרת לאופרטורים להיות מופנים לאובייקט מסוג String. האופרטור היחידי המאפשר זאת הנו +. אופרטור זה מבצע שרשור של בתי strings או יותר והמייצר לנו אובייקט מסוג string.

דוגמא

```
//String concatenation

class StringCon
{
      public static void main (String args[])
      {
            String longStr = "This is have been " +
                  "a very long course" +
                  "That would be" +
                  "Completed this summer";
            System.out.println(longStr);
      }
}
```

.11.4 המרת Stringתוך שימוש ב toString()

כאשר java מבצעת המרה של נתונים לתוך הצגת string תוך ביצוע שרשור הוא מבצע זאת תוך קריאה לאחת מגרסאות הפונקציות הנטענות העוזרות לנו לבצע המרה על ידי שימוש בפונקצית valueOf() המוגדרת על ידי String.

פונקציה זו הנה נטענת לכלל סוגי ה string הפשוטים ולכלל האובייקטים מסוגו.

כאשר נטענים הסוגים הפשוטים הפונקציה מחזירה string המכיל את הטקסט הקריא על ידנו ושהנו אקוויואלנטי לערכים הנקראים.

כאשר פונקציה זו משתמשת באובייקטים הקריאה נעשית לפונקצית toString() על האובייקט.

לעת עתה הבא נבחן את פעולתה של פונקצית toString() מאחר שאנו יכולים לקבוע בעזרתה כיצד ה string יוצג לאובייקטים של המחלקות שאנו יוצרים.

כל מחלקה מיישמת בעצם את פונקצית toString() מכיוון שהיא מוגדרת כחלק בלתי נפרד מהאובייקט.

אולם, ה default ליישום של פונקצית toString() הוא מספק בהחלט לביצוע המטלה.

לרוב המחלקות שאנו יוצרים אנו נרצה לשכתב את פונקצית toString() על מנת להציג את מה שאנו רוצים. למרבה המזל נוכל לעשות זאת על ידי שימוש במבנה הבא:

String toString()

דוגמא

```
//Override toString() for Box class

class Box
{
	double width;
	double height;
	double depth;

	Box(double w, double d, double h)
	{
		width=w;
		height=h;
		depth=d;
	}

	public String toString()
	{
		return "Dimensions are" + width + "By" + depth + "By" +
height;
	}
}
class BoxTest
{
	public static void main (String args[])
	{
		Box b = new Box(10, 12, 14);
		String s = "Box b: " + b;//Concantenate Box Object

		System.out.println(b);
		System.out.println(s);
	}
}
```

11.5. השוואת Strings

מחלקת String מכילה תת פונקציות רבות העוזרות לנו לבצע את המטלות ביעילות המירבית. בחלק זה נבחן את השימוש בהשוואת Strings

השוואת strings נעשית על ידי שימוש בשתי פונקציות:
equals() – על מנת לבצע השוואה בין שתי Strings והמבנה הכללי הנו:

boolean equals(Object Str)

כאשר:
Str – מוגדר כאובייקט אשר עליו אנו מבצעים את ההשוואה עם אובייקט string אחר. ערך זה יחזיר true באם ההשוואה הנה מכילה את אותם ערכים ובמידה ולא אזי הערך המוחזר יהיה false.

על מנת לבצע השוואה אשר "מתעלמת" ממקרים של הבדלים אזי כי נשתמש בפונקצית equalsIgnoreCase() . שימוש בפונקציה זו גורמת למצב שבו השוואה של A-Z יהיה זהה ל a-z.
המבנה הכללי הנו:

boolean equalsIgnoreCase(Str Object)

כאשר:
Str – מוגדר כאובייקט אשר עליו אנו מבצעים את ההשוואה עם אובייקט string אחר. ערך זה יחזיר true באם ההשוואה הנה מכילה את אותם ערכים ובמידה ולא אזי הערך המוחזר יהיה false.

דוגמא

בדוגמא זו נראה שימוש בשתי הפונקציות הללו תוך שימוש בהשוואת Strings

```
//String Comparison

class SringComparison
{
      public static void main (String args[])
      {
            String s1 = "Hello";
            String s2 = "Hello";
            String s3 = "Nice One";
            String s4 = "HELLO";

System.out.println(s1 + "equals" + s2 + "" + s1.equals(s2));
System.out.println(s1 + "equals" + s3 + "" + s1.equals(s3));
System.out.println(s1 + "equals" + s4 + "" + s1.equals(s4));
System.out.println(s1 + "equalIgnoreCase" + s4 + "" +
s1.equalsIgnoreCase(s4));
      }
}
```

12. כלי עזר נוספים – Java Utilities

ספריית java מכילה לנו עשרות מחלקות ופונקציות לשימוש. מחלקות אלו משמשות אותנו בתוך ליבת שפת התכנות תוך שימוש בחבילות וגם כמובן תוך שימוש בתכניות שאנו כותבים.
השימושים הנפוצים כוללים אכלוס איסוף אובייקטים, מספרים ראנדומליים למינהם, שימוש בזמן ובתאריך, ושימוש בתווים.
מחלקות העזר הללו נמצאות בתוך חבילת java.util והן כוללות:

- BitSet
- Date
- Dictionary
- Hashtable
- Observable
- Properties
- Random
- Stack
- Stringtokenizer
- Vector

בחלק זה נבחן ונציג מספר מחלקות ופונקציות המשתמשות אותנו במהלך חיי העבודה היום יומיים. נבחן את הבאים:

1. Vector
2. Random
3. Stack

12.1. שימוש ב Vector

כפי שאנו יודעים מערכים הנם בעלי מקום קבוע מראש כאשר אנו מגדירים אותם ז"א לאחר ההגדרה איננו יכולים להגדילם או להקטינם. תכונה זו מגבילה אותנו לכך שאנו חייבים לדעת מראש מה הוא גודל המערך הרצוי לנו. אך לפעמים קיימים מצבים בהם לא נדע בדיוק מה גודל המערך עד אשר system run time תוודא בדיוק מהו גודלו של המערך.
על מנת לטפל במקרים שכאלו שפת Java מגדירה לנו מחלקה בשם Vector.
לצורך החיוני שלנו, Vector מוגדר מערך באורך המשתנה של אובייקט התייחסות ז"א Vector יכול באופן דינמי לגדול או לקטון.
Vectors נוצרים בעת אתחול ראשוני של גודל המערך. כאשר גודל המערך גודל ויוצא מההגדרה הראשונית ה Vector באופן אוטומטי מבצע את התאימות. כאשר האובייקטים מוזזים קרי מסיימים עבודתם ה Vector מקטין את עצמו.
להלן מבנה הקונסטרקטורים של ה Vector:

1. Vector()
2. Vector(int size)
3. Vector(int size, int incr)

הקונסטרקטור הראשון הנו default לכל Vector משמע גודלו הראשוני הנו מערך בגודל של 10.
הקונסטרקטור השני הנו Vector אשר גודלו ייקבע על פי מספר.
הקונסטרקטור השלישי יוצר Vector אשר תכולתו נקבעת על פי מספר ועל פי ההגדלה.

בכל שלושת הקונסטרקטורים, באתחול התחלתי ה Vector הנו ריק.
כל ה Vectors מתחילים עם תכולה ראשונית. ברגע שהגענו לתכולה זו בפעם הבאה שנרצה לאתחל אובייקט בתוך Vector באופן אוטומטי יתבצע חיפוש למקום בזיכרון על מנת למקם אובייקט ובנוסף נלקח מקום נוסף מהזיכרון שמא נרצה לאתחל אובייקטים נוספים. על ידי מיקום יותר מהדרוש לנו ה Vector מוריד את מספר המיקומים שחייבים להתבצע ז"א רק אותם אובייקטים שהוגדרו הם שימוקמו בזיכרון. תכונה זו חשובה מאחר ומדובר בזמן ריצה ובעצם זמן ריצת תכנית הנו זמן קומפילציה בפועל.
הכמות הדרושה לנו בכל פעם שאנו נרצה למקם אובייקטים חדשים נקבעת על ידי שימוש בהגדלה שאנו קובעים. במידה ואיננו מגדירים את ההגדלה אזי כי בכל cycle מתבצע מתן מקום כפול לכל vector.
בדרך כלל נרצה להגדיר את ערכי ההגדלה, נוכל לבצע זאת על ידי סוג הנתונים הבא, קרי:

```
int capacityIncrement;
int elementCount;
Object elementData[];
```

ערך ההגדלה נכולס בתוך capacityIncrement. מספר האלמנטים הנמצאים בפועל ב vector הנם מאוכלסים ב elementCount והמערך אשר מחזיק את ה vector מאוכלס ב elementData.

Vectors מגדירים מספר פונקציות אשר נוכל להשתמש בהם. הממשק לשימוש ב vector הנו מסוג Cloneable.
השימוש ב vectors הנו קל להפליא, ברגע שיצרנו vector נוכל להוסיף אלמנטים אליו על ידי קריאה לפונקצית addElement(). על מנת לדעת מיקום מדויק של אלמנט מסוים נוכל להשתמש בפונקצית elementAt() .
על מנת לקבוע את מספר האלמטים שה vector מכיל נשתמש בפונקצית contains().
על מנת לדעת מהו האלמנט הראשון ב vector נוכל להשתמש בפונקצית firstElement() וכמובן על מנת לדעת מהו האלמנט האחרון נשתמש בפונקצית lastElement().
נוכל כמובן לדעת מהו האינדקס של האלמנטים על ידי שימוש בפונקצית indexOf() ו lastIndexOf().
על מנת להוריד אלמנט מהמערך נוכל להשתמש בפונקצית removeElement() או removeElementAt().

דוגמא

בדוגמא זו נשתמש ב vector על מנת לאכלס ערכים מספריים שונים כאובייקטים.

```
//Vectors

import java.util.Vector;
import java.util.Enumeration;

class vectorTest
{
      public static void main (String args[])
      {
            Vector v = new Vector(3, 2); //Initial size 3, incr in 2

            System.out.println("Initial size" + v.size());
            System.out.println("Initial capacity:" + v.capacity());

            v.addElement(new Integer(1));
            v.addElement(new Integer(2));
            v.addElement(new Integer(3));
            v.addElement(new Integer(4));

System.out.println("Capacity after 4 additional" + v.capacity());
            v.addElement(new Double(6.23));
            System.out.println("Current Capacity" + v.capacity());

            v.addElement(new Integer(7));
            System.out.println("Current Capacity" + v.capacity());
            v.addElement(new Float(8.2));
            v.addElement(new Integer(10));

            System.out.println("Current Capacity" + v.capacity());

            v.addElement(new Integer(11));
            v.addElement(new Integer(12));

System.out.println("First Element" + (Integer)v.firstElement());
System.out.println("Last Element" + (Integer)v.lastElement());

            if(v.contains(new Integer(3)))
                  System.out.println("Vector contains 3");
            //Enumerate the elements in the vector

            Enumeration vEnum = v.elements();

            System.out.println("Elements in vector");
            while (vEnum.hasMoreElements())
                  System.out.println(vEnum.nextElement() + " " );
            System.out.println();
      }
}
```

תוצאת התכנית

```
Initial size0
Initial capacity:3
Capacity after 4
additional5
Current Capacity5
Current Capacity7
Current Capacity9
First Element1
Last Element12
Vector contains 3
Elements in vector 1  2  3  4  6.23  7  8.2  10  11  12
```

12.2. שימוש במחלקת Random

מחלקת random הנה מחולל מספרים ראנדומליים. מחלקה זו מגדירה שני קונסטרקטורים.
המבנה הכללי הנו:

Random()

Random (long seed)

הקונסטרקטור הראשון מחולל מספרים אשר הוגדרו בפעם הראשונה שיצרנו את הערכים.
הקונסטרקטור השני מאפשר לנו לייצור ערכים חדשים באופן ידני.

כאשר אנו נאתחל את אובייקט random עם ערכים ראשוניים אנו נגדיר את הערך ההתחלתי לרצף הראנדומלי.
באם נשתמש באותם ערכים ראשוניים שהגדרנו על מנת לאתחל אובייקט random אחר נקבל מבנה של אותו רצף.
באם נרצה לקבל רצף אחר נצטרך כמובן לאכלס משתנים אחרים. הדרך הקלה ביותר לעשות זאת הנה להשתמש בערכים הראשוניים של random.
הפונקציות המוגדרות על ידי מחלקת random הנם חמישה סוגים במספר:

1. ערכים מספריים קרי int יכולים להיות מורצים על ידי פונקצית nextInt()
2. ערכים מפריים ארוכים קרי long int יכולים להיות מורצים על ידי שימוש בפונקצית nextLong()
3. פונקציות מסוג nextFloat() ו nextDouble() מחזירות ערכים כשמם ובהתאמה בין 0.0 ל 1.0
4. לבסוף פונקצית nextGaussian() מחזירה ערך double הממורכז בערך 0.0 עם התפלגות נורמלית של 1.0 וכפי שאנו מכירים זאת כעקומת בל או כהתפלגות הנורמלית.

טבלת פונקציות:

Method	Description
double nextDouble()	Return the next double random number
float nextFloat()	Return the next float random number
double nextGaussian()	Return the next Gaussian random number
int nextInt()	Return the next int random number
long nextLong()	Return the next long random number
void setSeed(long new seed)	Sets the seed value to that specified by newseed

דוגמא

בדוגמא זו נשתמש בפונקציה אשר תראה לנו את חישוב ההתפלגות הנורמלית ובנוסף התכנית מחשבת ומראה לנו את מספר הערכים הנמצאים בין שתי סטיות התקן +- תוך שימוש בערך נוסף של 0.5 בכל קטגוריה. התוצאה תוצג על המסך בעזרת כוכבית.

```java
//Random

import java.util.Random;

class RandomTest
{
	public static void main (String args[])
	{
		Random r = new Random();
		double value;
		double sum =0;
		int Stam[] = new int[10];

		for(int i=0; i<100; i++)
		{
			value = r.nextGaussian();
			sum += value;

			double t = -2;
			for (int x=0; x<10; x++, t +=0.5)
				if(value < t)
				{
					Stam[x]++;
					break;
				}
		}

		System.out.println("Average of values:" + (sum/100));

		//Display the bell Curve (Normal Distribution)
		for (int i=0; i<100; i++)
		{
			for (int x=Stam[i]; x>0; x--)
				System.out.print("*");
			System.out.println();
		}
	}
}
```

תוצאת התכנית

```
Average of values:-0.10858379680225332
******
****
**************
***************
*********
*********************
**************
***********
*****
*
```

12.3. שימוש במחלקת Stack

מחלקה זו הנה תת מחלקה של מחלקת vector אשר מיישמת LIFO . למחלקה זו קיים אך ורק default constructor אשר מייצר מחסנית ריקה לחלוטין. מחלקה זו מפאת הסיבה שהיא מורשת ממחלקת vector ולאור פעילות ההורשה ב java אזי כי היא מכילה את כלל הפונקציות הקיימות במחלקת vector.
על מנת לשים אובייקט בתחילת המחסנית אנו נשתמש בפונקציה הקראת ()push. על מנת להזיז אובייקט מכלל המחסנית אנו נשתמש בפונקצית ()pop וכאשר אנו משתמשים בפונקציה זו יוצא דופן מסוג EmptyStackException נזרק החוצה.
נוכל להשתמש בפונקצית ()peek על מנת להחזיר ערך אבל לא על מנת להזיז את הערך העליון של האובייקט!
פונקצית ()empty מחזיקה ערך של true אם במידה ולא קיים כלום במחסנית.
פונקצית ()search תקבע באם אובייקט קיים במחסנית והאם הוא יחזיר ערך לפונקצית ה ()pop

דוגמא

בדוגמא זו ניצור מחסנית אשר "דוחפת" מספר Integers ואז "מקפיצה" אותם החוצה.

```
//Stack

import java.util.Stack;
import java.util.EmptyStackException;

class StackTest
{
	static void showpush(Stack STK, int a)
	{
		STK.push(new Integer(a));
		System.out.println("Push (" + a + " ) ");
		System.out.println("stack: " +STK);
	}

	static void showpop(Stack STK)
	{
		System.out.println("pop");
		Integer a = (Integer) STK.pop();
		System.out.println(a);
		System.out.println("stack: " + STK);
	}

	public static void main (String args[])
	{
		Stack STK = new Stack();

		showpush(STK, 42);
		showpush(STK, 66);
		showpush(STK, 88);
		showpop(STK);
		showpop(STK);
		showpop(STK);
		try
		{
			showpop(STK);
		}
		catch(EmptyStackException e)
		{
			System.out.println("Emoty Stack");
		}
	}
}
```

13. מחלקת Applet

המטרה העיקרית להגדרת GUI בספריות java הייתה על מנת לאפשר למתכנת לבנות אפליקציות שיוכלו לעבוד על כל פלטפורמה. המטרה הזו איננה הושגה בגרסת java 1.0. במקום פותחה גרסת AWT היוצרת לנו את מבנה ה GUI העובד על כל פלטפורמת עבודה. בנוסף הגרסה הוגבלה לשימוש רק בארבעה פונטים ואין באפשרותנו להרחיב את אותם פונטים גם אם מערכת ההפעלה שלנו מתוחכמת.
בנוסף AWT היא גרסה לבניית GUI שאיננה מבוססת תכנות מונחה עצמים והסיבה היא מכיוון שה AWT תוכננה ועוצבה בזמן קצר מאוד (חודש פיתוח).
המצב השתפר עם גרסת AWT 1.1 הכולל בתוכו מודולים של מאורעות אשר בעצם מייצג את גישת התכנות מונחה העצמים ביחד עם JavaBeans שהוא רכיב העוזר לנו לייצור תכנות ויזואלי.
גרסת java 2 כוללת בתוכה את מכלול הרכיבים ו AWT הוחלף ב(JFC)
Java Foundation Classes הכולל בתוכו את מגוון ה GUI אשר נקרא Swing.
אלו הם סטים עשירים בתכונות קלות ליישום ולשימוש Java Beans אשר בעצם בנויים על קונספט של dragged and dropped כמו גם תכנות על מנת לייצור GUI .
בחלק זה לא נכסה את כלל הספריות אלה נתמקד ב java swing 2. במידה ונרצה להשתמש ב AWT מסיבות תמיכה כגון דפדפן ישן ואו קוד ישן אזי כי נצטרך לבצע התאמות בהתאם.
בחלק זה נבחן כיצד אנו יוצרים applet בתכנות ישן לעומת יצירת GUI תוך שימוש ב swing כמו גם תוכניות שיכולות לרוץ הן מהדפדפן והן משורת פקודה.
ושוב אדגיש כי זה איננו מכיל את כלל ספריות ה swing אלא רק את מה שנצטרך על מנת להבין את הקונספט של בניית אפליקציות ב java.
באם נרצה להתמקד ב swing נוכל למצוא חומר נוסף ב (free) Java library documents in HTML format from java.sun.com.
ככל שנלמד על Swing נבין כי:

1. התכנות הנו במודולים לוגיים טובים יותר משפות אחרות כאשר Java Beans הנו המסגרת עבודה לספריית ה swing.
2. בניית GUI מתבצעת בצורה מהירה יותר המאפשרת לנו המתכנתים לכתוב קוד אשר עוזר לנו לעצב את הרכיבים בתכניות.
3. אם נשתמש אך ורק ברכיבי ה swing ולא נכתוב קוד התוצר יהיה ברמת קוד טובה מאוד.

Swing מכיל את כלל הרכיבים שאנו מצפים לראות בשפות מתקדמות ובעיצוב ממשקים מתקדם הכולל כפתורים, תמונות, עצים וטבלאות. היא בעצם ספרייה ענקית שעוצבה לתת מענה למטלות מורכבות ומסובכות אם כי במקרים בהם נרצה לבצע מטות מורכבות אזי כי נכתוב קוד. כמו גם קיימת האפשרות לחבר רכיבים האחד לשני והתוצר יעבוד.
בנושא מהירות תקשורת, כלל רכיבי Swing כתובים בשפת java כך שלא קיימת בעיה של התאמות על פלטפורמות שונות.
שימוש בלוח המקשים מתבצע באופן אוטומטי קרי נוכל להריץ אפליקציות ללא שימוש בעכבר וללא תכנות נוסף.
קיימת תמיכה בגרירה באופן הפשוט אנו עוטפים את האובייקט שלנו ב JscrollPane ומוסיפים אותו לטופס שלנו.
בנוסף קיימת התכונה של look and feel אשר עוזרת לנו לעצב את הממשקים בצורה הטובה ביותר.

13.1. אפלט בסיסי – Basic Applet

אחת ממטרות java הנה יצירת applets אשר הנם תוכניות קטנות הרצות מתוך הדפדפן. מכיוון שהם חייבות להיות מאובטחות (התכניות) applets הנם מוגבלים שאנו רוצים לפרסמם בנט. אולם applets הנם כלי מצוין לתכנות client .

13.1.1. הוראות ל applets

התכנות לביצוע applet הנו מוגבל קרי מערכת האבטחה של java עוקבת אחרי הפיתוח ברמה מסוימת.
אולם, נוכל תמיד לכתוב אפליקציות ככל שנרצה להרחיבן ואו לפתחן על מנת לנצל את מערכת ההפעלה שלנו.
עד כה עסקנו בכתיבת אפליקציות רגילות אשר רצו על console applications ללא שימוש בגרפיקה.
בעזרת שימוש ב swing נוכל לבנות אפליקציות GUI המותאמות לאפליקציות רגילות.
מטרת השימוש ב applet הנה בעצם "הרחבה" של אפשרויות דף האינטרנט במתן אפליקציות אינטראקטיביות.
מאחר שאנו מעונינים להריץ דפי אינטרנט מאובטחים יש לנקוט במספר אמצעים:

1. ה applet איננו יכול לרוץ על הדיסק המקומי.
2. זמן טעינת ה applet הנו ארוך.

13.1.2. יתרונות applets

בבניית applets קיימים לנו מספר יתרונות:

1. לא קיים עניין ההתקנה מכיוון ש applet רץ לבד ובעזרת ה API של הדפדפן.
2. אין צורך לדאוג שכתיבת קוד גרוע תגרום לנזק במערכת מסוימת מכיוון שנושא האבטחה הנו built in בתוך השפה.

מאחר ש applets הנם רכיבים שאוטומטית מבצעים אינטגרציה עם HTML אזי כי כל מערכת התומכת בדפדפן אינטרנט תומכת למעשה ב applets.

13.1.3. מסגרת העבודה האפליקטיבית

ספריות בד"כ הנם קבוצות הבנויות יחדיו לפי אופי עבודתם. לדוגמא הספרייה הסטנדרטית של java הנקראת String ביחד עם מחלקות ArrayList.
ספריות אחרות עוצבו כבלוק בונה על מנת לייצור מחלקות אחרות. קטגוריה מוחלטת של ספרייה נקראת בשפה המקצועית application framework שמטרתה הנה בעצם לעזור לנו לבנות אפליקציות ולהתנות לנו את התנהגותן על ידי אספקת המחלקות ואז לבצע קסטומיזציה של ההתנהגות לצרכינו אנו. מסגרת העבודה של האפליקציה מטע עבודתה תדאג לכלל הפונקציות בעבודה תוך כמובן טעינתם ואו שכתובם.
Applets בנויים תוך שימוש במסגרת עבודה מוגדרת. ההורשה מתבצעת ממחלקת Japplet ואז אנו משכתבים את הפונקציה הרצויה. קיימות מספר פונקציות השולטות ביצירה ובהרצה של applets על גבי דף אינטרנט והן:

Method	Operation
init( )	Automatically called to perform first-time initialization of the applet, including component layout. You'll always override this method.
start( )	Called every time the applet moves into sight on the Web browser to allow the applet to start up its normal operations (especially those that are shut off by stop( )). Also called after init( ).
stop( )	Called every time the applet moves out of sight on the Web browser to allow the applet to shut off expensive operations. Also called right before destroy( ).
destroy( )	Called when the applet is being unloaded from the page to perform final release of resources when the applet is no longer used

בעזרת המידע בטבלה לעיל נוכל לבחון את הדוגמא הבאה:

```
// Very simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
}
```

שים לב כי ל applet לא דרושה פונקצית main() ז"א על ידי הגדרת מסגרת העבודה שהיא בעצם הגורם להרצת ה applet כלל הפעולות נעשות. כל קוד שנרצה על מנת להטעין את ה applet נכנס בתוך פונקצית init().
בתכנית זו הפעילות היחידה המתבצעת הנה לשים טקסט על ה applet תוך שימוש במחלקת Jlabel. הקונסטרקטור למחלקה זו לוקח String ומשתמש בו על מנת לייצור את הטקסט label. בדוגמא זו הטקסט label קיים על הטופס.
פונקצית init() הנה האחראית לשים את כלל הרכיבים בטופס שלנו על ידי שימוש בפונקצית add().
לפונקציה זו לא נוכל לקרוא באופן פרטי וכך הוא שימושה ב AWT הישן. אולם שימוש ב swing דורש מאתנו לקרוא לפונקצית getContentPane( ) ולפיכך פונקצית add() הנה חלק מההתהליך.

13.2. מחלקת Applet

המחלקה applet מגדירה ומשתמשת במספר פונקציות שנראה בהמשך. המחלקה מספקת את מכלול הצרכים להרצת applets כגון תהליך התחלה, סיום וכדומה. בנוסף המחלקה מכילה פונקציות להצגת תמונות, וידאו וקליפים.
מחלקת Applet הנה הורשה של AWT ושל מחלקת Panel. מחלקת Panel הנה הורשה של מחלקת Container אשר הנה הורשה של מחלקת Component.
כלל המחלקות מספקות לנו את התמיכה שאנו צריכים על מנת לבנות חלונות ותמיכה בגרפיקה.

Method	Description
void destroy()	Called to the browser just before the applet is terminated.
AppletContext getAppletContext()	Return the context associated with the applet
String getAppletIno()	Return the String that describe the applet
AudioClip getAudioClip (URL)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by URL
AudioClip getAudioClip (URL url, String clip name)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by URL and having the name specified by clipName
URL getCodeBase()	Returns the URL associates with the applet
URL getDocuntationBase()	Returns the URL associates with the invoke applet
Image getImage(URL)	Returns an Image object that encapsulates the audio clip found at the location specified by URL
Image getImage(URL image Name)	Returns an Image object that encapsulates the audio clip found at the location specified by URL and having the name specified by ImageName
String getParameter(String param Name)	Returns the parameter associated with paramName. Null is returned if the specified parameter is not found.
String [] [] getParameterInfo()	Return String table that describe the parameters associates with the applets
void init()	
boolean isActive()	
void play(URL url)	Play audio
void play (URL url, String clipName)	Resize the applet according to the dimension associates with dim
void Resize(Dimension dim)	
void Resize(int width, int height)	
final void setStub(AppletStub stubObject)	
void showStatus(String str)	Applet status
void start()	Start
void stop()	Stop

13.3. ארכיטקטורת Applet

Applet הנה תכנית אשר בעצם מריצה חלונות. לפיכך הארכיטקטורה הנה שונה מאפליקציות המבוססות ורצות על סביבת דוס (כמו שעשינו עד כה).

נקודות חשובות לדיון הנם:

1. יש לזכור כי applet הנו event driven קרי אנו עובדים לפי מאורעות שאנו מגדרים.
2. המשתמש הסופי הוא בעצם האחראי לתפעל את ה applet .

מבנה שלד של Applet

ככלל כל ה applelts מכילים מספר פונקציות אשר מספקות את הטכניקה על מנת שהדפדפן יצליח להתממשק ל applet ולהריצו. ארבעת הפונקציות העיקריות הנן:

Init(), start(), stop(), destroy().

ישנה פונקציה נוספת עיקרית שהנה paint() והיא הורשה של Components אשר הנן חלק מ AWT.

להלן מבנה השלד:

```
//An Applet skelton
import java.awt.*;
import java.applet.*;
/*
 <applet code = "AppletSkelton" width=300 height=100>
 </applet>
*/

public class AppletSkelton extends Applet
{
      public void init()
      {
      }

      public void start()
      {
      }

      public void stop()
      {
      }

      public void destroy()
      {
      }

      public void paint(Graphics g)
      {
      }
}
```

למרות ששלד זה הנו ריק מתוכן אזי כי יש לנו תוצאה והנה applet ריק.

13.4. הרצת applets מתוך דפדפן

על מנת להריץ את התכנית הזו אנו נצטרך לשים אותה בתוך דף אינטרנט שניצור וכמובן בעזרת דפדפן התומך ב java.

על מנת לשים את ה applet בתוך דף HTML יש לפתוח את התג הבא, לדוגמא:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

בתוך התגים הללו אנו נכניס את קוד ה applets שלנו.
לדפדפנים מסוג Microsoft הטכניקה המפעילה את ה applets הנה ActiveX control ולדפדפן Netscape הטכניקה הנה plug-in.
בתוך קוד ה HTML שלנו אנו חייבים לכתוב קוד שיתמוך בשני בדפדפנים.

להלן הדוגמא:

```
<html><head><title>Applet1</title></head><hr>
<OBJECT
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM NAME="type" VALUE="application/x-java-
applet;version=1.2.2">
<COMMENT>
  <EMBED type=
    "application/x-java-applet;version=1.2.2"
    width="200" height="200" align="baseline"
    code="Applet1.class" codebase=".">
  <NOEMBED>
</COMMENT>
   No Need of Java Lang 2 support Applet
  </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
```

הסבר התכנית

תוצר התכנית נותן לנו את class. שהוא בעצם ה applet. מתן מידות של רוחב ואורך הנן מידות ה applets שיקבעו כאשר הוא יטען. (כמובן שהמידות הנן ב pixel). קיים מידע נוסף שנוכל לשים בתוך תג ה applet כגון מידע על חיפוש קובץ class. באינטרנט, מידע על align ומידע על הפרמטרים של ה applet על מנת לספק לנו אינפורמציה שה applet יכול לקבל. פרמטרים מופעים בצורה הבאה:

```
<param name="identifier" value = "information">
```

13.5. שימוש ב Appletviewer

בתוך ערכת JDK קיים רכיב אשר נקרא Appletviewer שבעצם אוסף את תג ה <applet> מחוץ לקוד ה HTML ומריץ אותו בצורה כזו באופן בודד. מכיוון ש Appletviewer מתעלם מכלל הקוד של HTML נוכל לשים את התגים הללו בקובץ מקור של java בצורה הבאה:

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

בצורה כזו אנו יכולים להריץ "appletviewer MyApplet.java" ובכך לא נצטרך לייצור קובץ HTML על מנת להריץ applet.
נוכל להוסיף את שורת הקוד ל applet שיצרנו ונקבל Applet1.java :

```
// Embedding the applet tag for Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
}
```

במצב זה נוכל להריץ את ה applet בצורה הבאה:

```
appletviewer Applet1b.java
```

13.6. בדיקת applets

תהליך הבדיקה הוא מאוד פשוט. לצורך כך נפתח את הדפדפן ונריץ את קובץ ה htm שיצרנו. ברגע שקובץ זה רץ הדפדפן מחפש את class. אשר מוגדר בקובץ ה applet.
כאשר נרצה לבחון את ה applet מחוץ לדפדפן התהליך קצת יותר מורכב. דבר ראשון עלינו להיות מחוברים לאתר אינטרנט.
מעבר לכך ב applet חייב להיות מותקן על השרת בספרייה שאנו נקבע תוך כמובן סנכרון ה IIS ובסיס הנתונים כך שאנו בודקים applet יש לוודא כי אכן מחקנו את המסלול למחשב המקומי שלנו מכיוון ש CLLASSPATH חפש אותו שם תחילה ולכן נשים לב כי ה applet מסונכרן.

13.7. הרצת applet מתוך שורת פקודה

ישנם מערכות הפעלה (כמו מקינטוש) שבהם לא קיימת שורת פקודה להרצה מסוימת ולכן נרצה להריץ את ה applet בתוך אפליקציה שאיננה "חלונאית".
ספריות ה swing מאפשרות לנו לבצע את הרצת האפליקציה כך שלא נצטרך מערכת הפעלה התומכת בחלונות אם כי המדובר הנו על מערכות הפעלה ישנות מבחינה טכנולוגית.
לפיכך נצטרך לבנות מחלקה אשר קוראת לחלונות או לך applet.
על מנת לייצור applet אשר יוכל לרוץ מתוך שורת פקודה אנו פשוט נוסיף פונקצית main() ל applet שאנו בונים שהוא בעצם יהיה instance ל applet בתוך הקוד שלנו.

דוגמא

בדוגמא זו ניצור קובץ בשם Applet1b.java אשר בעצם יכול לעבוד גם כאפליקציה וגם כ applet.

```
// An application and an applet.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1c extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
  // A main() for the application:
  public static void main(String[] args) {
    JApplet applet = new Applet1c();
    JFrame frame = new JFrame("Applet1c");
    // To close the application:
    Console.setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(100,50);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
}
```

13.8. סדר טעינת ופריסת ה Applet

חשוב מאוד להבין כיצד הפונקציות המוגדרות עובדות כאשר אנו טוענים ומשתמשים ב applets. כאשר ה applet נטען ה AWT קורא לפונקציות הבאות בסדר הבא:

1. init()
2. start()
3. paint()

כאשר ה applet מפסיק את פעולתו מתבצעות הפעולות הבאות:

1. stop()
2. destroy()

13.9. שימוש בפונקצית update()

ישנם מצבים בהם נרצה לשכתב את ה applet שלנו. נוכל לבצע זאת על ידי שימוש בפונקצית update() . פונקציה זו נקראת כאשר אנו מבקשים מה applet לבצע שינוי.
השימוש ה default הראשון הנו בעצם מילוי ה applet בצבע רקע תוך שימוש בפונקצית paint() . באם נמלא את ה applet בצבעים מראש אזי כי למשתמש לא תהיה בחירה לבצע שינויים.
דרך אחת לביצוע הנה לשכתב את פונקצית update() באופן שבו נוכל לבצע איתה מה שאנו רוצים ורק לאחר מכן לקרוא לפונקצית paint() .
מבנה השלד נראה כך:

```
public void update (Graphics g) {
//redisplay your window
}

public void paint (Graphics g) {
update (g);
}
```

13.10. שימוש בפונקציות ב Applet

למרות השימוש בפונקציות AWT על מנת לבצע את גרפיקת ה applet קיימים שימושים נוספים בפונקציות שנוכל לבצע על מנת לקבל את התוצר הרצוי לדוגמא נרצה לכתוב string לתוך ה applet.
על מנת להוציא string ל applet ישנו הצורך להשתמש בפונקצית drawString() אשר הוא חבר במחלקת Graphics.

המבנה הכללי הנו:

public void drawstring (String message, int x int y)

כאשר:
Massage – הנו ההודעה שאנו רוצים לכתוב על ה applet. ערכיו מתחילים כ default ב 0,0.
drawString() – איננה מזהה מאפייני קווים חדשים ז"א אם נרצה להתחיל שורה חדשה ליד השורה שיצרנו נצטרך לבצע זאת ידנית כך שנגדיר במדוייק את מקום משתני ה x,y. (ישנם טכניקות לבצע זאת בקלות)
על מנת לתת צבע רקע ל applet נשתמש בפונקצית setBackground() ועל מנת לתת צבע מקדים נשתמש בפונקצית setForeground().
פונקציות אלו מוגדרות על ידי מחלקת Components ולהן יש את המבנים הכללים הבאים:

void setBackground(Color newColor)

void setForeground(Color newColor).

דוגמא

```
//An Applet with using of Backgound and foregrond

import java.awt.*;
import java.applet.*;
/*
 <applet code = "Simple" width=300 height=50>
 </applet>
*/

public class Simple extends Applet
{
      String msg;

      //set the foreground and background colors
      public void init()
      {
            setBackground(Color.darkGray);
            setForeground(Color.cyan);
            msg = "Inside init()";
      }

      public void start()
      {
            msg += "Inside start()";
      }

      public void paint(Graphics g)
      {
            msg += "Inside paint()";
            g.drawString(msg, 10, 25);
      }
}
```

13.11. בקשה לצביעה מחודשת

ככלל applet כותב לחלון שלו רק כאשר מתבצע שימוש בפונקצית ()update או בפונקצית ()paint הנקראות על ידי ה AWT.
נשאלת שאלה מעניינת הכיצד ה applet יכול להשפיע על החלון עליו הוא מופעל לבצע שינוי כלשהו? התשובה הנה על ידי שימוש בפונקצית ()repaint. שים לב כי כל השינויים מתבצעים בזמן run time של ולכן כל שינוי המתבצע על ידי applet בעצם "מדווח" ישירות ל AWT.
פונקצית ()repaint הנה מוגדרת על ידי WAT והיא גורמת ל AWT לבצע שינויים בזמן ההרצה על ידי שימוש בפונקצית ()update.
לפונקצית ()repaint ישנם ארבע צורות בהן היא יכולה להופיע:

1. ()void repaint – צביעה מחודשת של כלל החלון
2. (void repaint(int left, int top, int width, int height – במקרה זה הקואורדינטות של החלק השמאלי המוגדרות על ידי המשתנים בפונקציה מוחלפים ומוגדרים במידות pixels.
3. (void repaint(long maxDelay – נותן עיכוב עד אשר יתבצע השימוש בפונקצית ()update
4. (void repaint(long maxDelay, int x, int y, int width, int height - נותן עיכוב עד אשר יתבצע השימוש בפונקצית ()update ובנוסף נותן קואורדינטות מדויקות לביצוע העיכוב.

דוגמא

```
//An Applet with using of Backgound and foregrond and changing colors

import java.awt.*;
import java.applet.*;
/*
 <applet code = "Banner" width=300 height=150>
 </applet>
*/

public class Banner extends Applet implements Runnable
{
	String msg = "Moving Banner Applet";
	Thread t = null;//thread definition

	//set the foreground and background colors
	public void init()
	{
		setBackground(Color.darkGray);
		setForeground(Color.cyan);
		t = new Thread(this);
		t.start();
		t.suspend();
	}
	public void start()//resume thread
	{
		t.resume();
	}
	//entry point for the thread
	public void run()
	{
		char ch;
		//display the banner until we will stop it
		for( ; ; )
		{
			try
			{
				repaint();
				Thread.sleep(250);
				ch = msg.charAt(5);
				msg = msg.substring (0, msg.length());
				msg += ch;
			}
			catch (InterruptedException e) {}
		}
	}

	// pause the banner
	public void stop()
	{
		t.suspend();
	}
	//stop thread when applet is terminated
	public void destroy()
	{
		if(t != null)
		{
			t.stop();
			t = null;
		}
	}
```

```
	public void paint(Graphics g)
	{
		g.drawString(msg, 60, 25);
	}
}
```

דוגמא

ניקח את הדוגמא לעיל ונשפר אותה בכך שנוסיף הודעה אשר תתחלף בכל פעם.

```
//An Applet with changing massage

import java.awt.*;
import java.applet.*;
/*
 <applet code = "BannerChange" width=300 height=150>
<param name = message  value = "On the web">
</applet>
*/

public class BannerChange extends Applet implements Runnable
{
	String msg;
	Thread t = null;//thread definition

	//set the foreground and background colors
	public void init()
	{
		setBackground(Color.darkGray);
		setForeground(Color.cyan);
		t = new Thread(this);
		t.start();
		t.suspend();
	}
	public void start()//resume thread
	{
		msg = getParameter("message");
		if (msg == null) msg ="Message Not Found";
		msg = "  " + msg;
		t.resume();
	}
	//entry point for the thread
	public void run()
	{
		char ch;
		//display the banner until we will stop it
		for( ; ; )
		{
			try
			{
				repaint();
				Thread.sleep(250);
				ch = msg.charAt(0);
				msg = msg.substring (1, msg.length());
				msg += ch;
			}
			catch (InterruptedException e) {}
		}
	}
```

```
        // pause the banner
        public void stop()
        {
                t.suspend();
        }
        //stop thread when applet is terminated
        public void destroy()
        {
                if(t != null)
                {
                        t.stop();
                        t = null;
                }
        }
        public void paint(Graphics g)
        {
                g.drawString(msg, 50, 30);
        }
}
```

13.12. שימוש בחלון הודעת המצב

בנוסף להצגת מידע בתוך ה applet נוכל כמובן להציג מידע על הסטטוס של החלון שבו ה applet רץ. נוכל לבצע זאת על ידי שימוש בפונקצית showStatus() עם ה string שנרצה לרשום. הודעת הסטטוס הנה יעילה במתן אינפורמציה למשתמש .

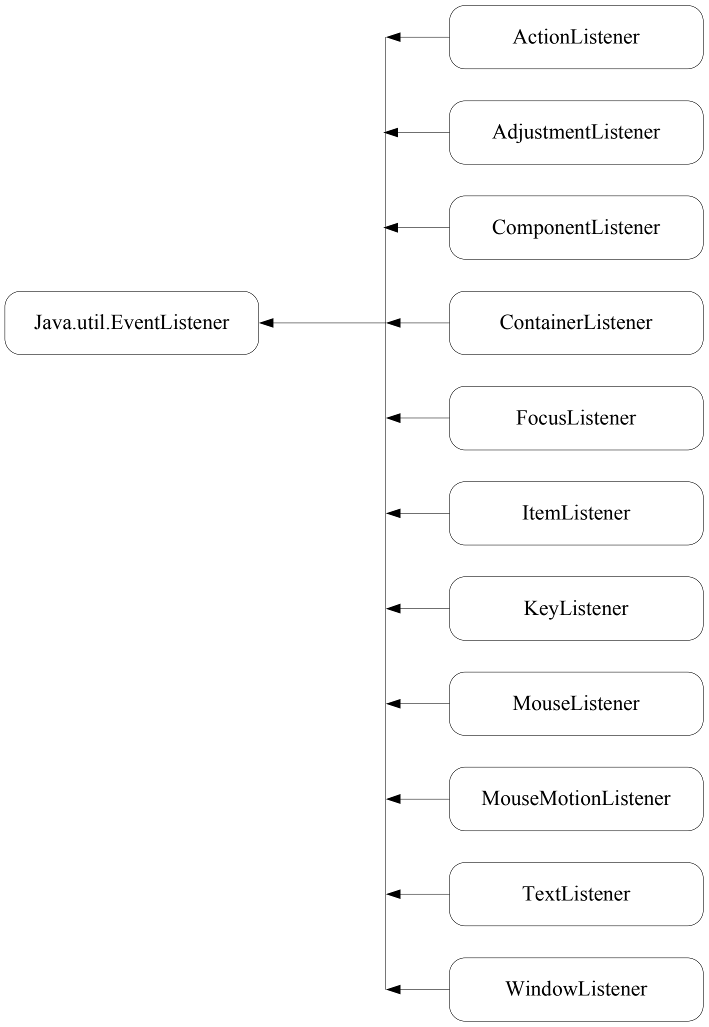
דוגמא

```
//An Applet with setStatus()

import java.awt.*;
import java.applet.*;
/*
 <applet code = "SetStatus" width=300 height=150>
 </applet>
*/

public class SetStatus extends Applet
{
	public void init()
	{
		setBackground(Color.darkGray);
	}

	public void paint(Graphics g)
	{
		g.drawString("This is Status applet", 10, 20);
		showStatus("The Status Windows");
	}
}
```

14. מחלקת Event

כלל האירועים מרוכזים (משתמשים בתכונת ריכוזיות) בתוך אובייקט בשם Event כאשר מחלקת Event הנה חלק מה AWT. מודל מחלקת ה Event מושתת בכללותו על GUI אשר נגזר ממאורעות אותן המשתמש מבצע. לדוגמא, האובייקט בעבודתו מגדיר מספר משתנים המתארים אירוע מסוים לדוגמא מיקום עכבר בצירי X Y, מקלדת וכדומה.
כאשר מעורבת מקלדת ובזמן אירוע שבו המשתמש ילחץ על כפתור מסוים אזי כי חלק מכפתורי המקלדת מוגדרים בתוך משתני key .
בנוסף אובייקט Event מגדיר מספר פונקציות וקבועים למינהם.
אופי הפעולה:
כאשר מתבצעת פעולת אינטרקציה מהמשתמש, מאורע נשלח באופן אוטומטי לתכנית. מאורע GUI מאוכלס באובייקט של מחלקה היורשת מ AWTEvent אשר הנה מחבילת java.awt.event.

להלן רשימת מאורעות מחבילת java.awt.event המתמשים ברכיבי Swing:



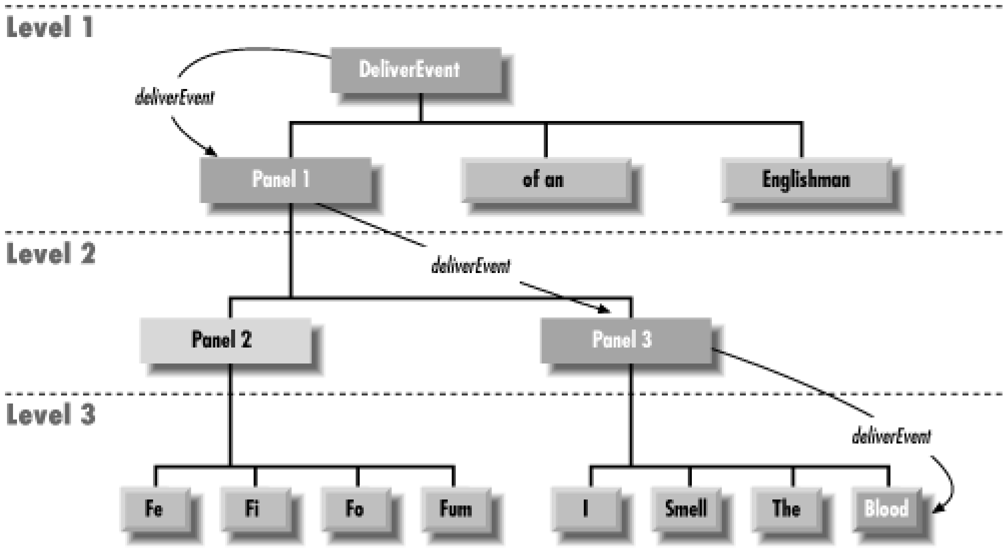
14.1.1. טיפול באירוע עם עכבר

הפונקציות הכללות בתוך Event בעצם מורשות על ידי מחלקת Applet כך שאנו יכולים לשלוט בעכבר ברגע שה Applet רץ ובל נשכח כי מחלקת Applet יורשת את מאפייניה ממחלקת Component.
כאשר התהליך מתבצע הפונקציה חייבת להחזיר ערך true ובמידה וה applet איננו יכול לטפל במאורע אזי כי יחזור לנו ערך false.
תהליך זה גורם לחלון לבצע פעילות מסוימת. באופן כללי, במידה ונשכתב את ה event handler נול לטפל במאורע ולהחזיר ערך true.
שתי הפונקציות החשובות ביותר הנן:
mouseUp()
mouseDown()

פונקצית mouseUp() נקראת כאשר כפתור עכבר נלחץ. פונקצית mouseDown() נקראת כאשר הכפתור הנלחץ משתחרר. Java אינה מפרידה בין כפתורי העכבר מכיוון שלא לכלל המערכות קיימים מספר כפתורים קבוע.

דוגמא

העברת המאורע מתבצעת באופן הבא



הטבלה הבאה מתארת את הפונקציות הקשורות למאורעות:

Event Types and Event Handlers	
Event Type	**Event Handler**
MOUSE_ENTER	mouseEnter()
MOUSE_EXIT	mouseExit()
MOUSE_MOVE	mouseMove()
MOUSE_DRAG	mouseDrag()
MOUSE_DOWN	mouseDown()
MOUSE_UP	mouseUp()
KEY_PRESS	keyDown()
KEY_ACTION	keyDown()
KEY_RELEASE	keyUp()
KEY_ACTION_RELEASE	keyUp()
GOT_FOCUS	gotFocus()
LOST_FOCUS	lostFocus()
ACTION_EVENT	action()

דוגמא

בדוגמא זו נבחן כיצד ה applet מטפל במאורע הקשור לעכבר. התכנית תראה את מיקום העכבר בקואורדינטות כאשר הוא יהיה על ה applet. בכל פעם שנלחץ על כפתור העכבר תופיע מילה "Down" ובכל פעם שהכפתור ישתחרר תופיע המילה "Up" כאשר נגרור את העכבר תופיע לנו * אשר בעצם תאתחל את ערכי X Y החדשים.

רק לאחר שקיימות הקואורדינטות החדשות אזי נשתמש בפונקצית paint() על מנת להראות את תוצאת התכנית.

```
//An Applet with mouse event handling

import java.awt.*;
import java.applet.*;
/*
 <applet code = "MEvent" width=300 height=100>
 </applet>
*/

public class MEvent extends Applet
{
      String msg = " ";
      int mouseX = 0, mouseY = 0;

      //handling button press on X Y
      public boolean mouseDown(Event evtObject, int x, int y)
      {
            mouseX = x;
            mouseY = y;
            msg = "Pressed Down";
            repaint();
            return true;
      }

      public boolean mosueUp(Event evtObject, int x, int y)
      {
```

```
            mouseX = x;
            mouseY = y;
            msg = "Released Up";
            repaint();
            return true;
      }

      //mouse move
      public boolean mouseMove(Event evtObject, int x, int y)
      {
            showStatus("Moving mouse at" + x + "," +y );
            return true;
      }

      // mouse drag
      public boolean mouseDrag(Event evtObject, int x, int y)
      {
            //save coordinates
            mouseX = x;
            mouseY = y;
            msg = "Released Up";
            showStatus("Dragging mouse at" + x + "," +y );
            repaint();
            return true;
      }
      //mouse enter
      public boolean mouseEnter(Event evtObject, int x, int y)
      {
            //save coordinates
            mouseX = 0;
            mouseY = 25;
            msg= "Mouse Enter";
            repaint();
            return true;
      }
      //mouse exit
      public boolean mouseExit(Event evtObject, int x, int y)
      {
            mouseX = 0;
            mouseY = 25;
            msg= "Mouse Left";
            repaint();
            return true;
      }

      public void paint (Graphics g)
{
g.drawString(msg, mouseX, mouseY);
}
}
```

14.1.2. טיפול באירוע עם מקלדת

בנוסף לשימוש בעכבר קיים שימוש במקלדת. השימוש נעשה על ידי פונקציות keyDown() בזמן לחיצה על המקלדת ופונקצית keyUp() בזמן שיחרור המקלדת.
על בסיס אותו רעיון גם בשימוש במקלדת קיימות פונקציות עיקריות וקבועים לשימוש.
להלן הטבלה המתארת את הפונקציות העיקריות:

Constants for Keys in Java

Constant	Event Type		Constant	Event Type
HOME	KEY_ACTION		F9	KEY_ACTION
END	KEY_ACTION		F10	KEY_ACTION
PGUP	KEY_ACTION		F11	KEY_ACTION
PGDN	KEY_ACTION		F12	KEY_ACTION
UP	KEY_ACTION		PRINT_SCREEN★	KEY_ACTION
DOWN	KEY_ACTION		SCROLL_LOCK★	KEY_ACTION
LEFT	KEY_ACTION		CAPS_LOCK★	KEY_ACTION
RIGHT	KEY_ACTION		NUM_LOCK★	KEY_ACTION
F1	KEY_ACTION		PAUSE★	KEY_ACTION
F2	KEY_ACTION		INSERT★	KEY_ACTION
F3	KEY_ACTION		ENTER (\n)★	KEY_PRESS
F4	KEY_ACTION		BACK_SPACE (\b)★	KEY_PRESS
F5	KEY_ACTION		TAB (\t)★	KEY_PRESS
F6	KEY_ACTION		ESCAPE★	KEY_PRESS
F7	KEY_ACTION		DELETE★	KEY_PRESS
F8	KEY_ACTION			

דוגמא

על בסיס אותו רעיון של שימוש בעכבר השימוש מתבצע על סמן ערכים בוליאניים אמת / שקר. הפעולה המתבצעת בקומפיילר הנה פעולת casting קרי המרה של כפתור מקלדת לסוג char.

```
//An Applet with mouse event handling

import java.awt.*;
import java.applet.*;
/*
 <applet code = "KeyTest" width=300 height=100>
 </applet>
*/

public class KeyTest extends Applet
{
      String msg = " ";

      //handling key press event
      public boolean keyDown(Event evtObject, int key)
      {
            msg += (char) key;
            repaint();
        showStatus("Key Pressed");
            return true;
      }
    //key release
      public boolean keyUp(Event evtObject, int key)
      {
            showStatus("Key Released");
            return true;
      }
      //key display
      public void paint (Graphics g)
      {
            g.drawString(msg, 20, 30);
      }
}
```

.14.2 הבנת HTML & Applet Tags

כאשר אנו יוצרים applets קיימת האפשרות להריץ מספר applets בדפדפן אם כי לא קיימת האפשרות להריץ מספר applet בתוך ה applet viewer.
בחלק זה נראה כיצד אנו מבצעים את הסנכרון ברמת HTML בין הקוד שלנו לבין הדפדפן.

להלן המבנה הכללי של דף HTML הכולל בתוכו Applet:

```
<Applet
[codebase=codebaseURL]
code=appletFile
[alt=text]
[name=applet name]
width=applet width in pixel
[align=alignment]
[vspace=pixel]
[hspace=pixels]
>
[<param name = attribute name VALUE = attribute value]
</Applet>
```

כאשר:
Applet – הנו תג המצוי בשפת HTML ומודיע לדפדפן כי קיים אובייקט.
Codebase- הנו מאפיין נוסף הקיים ומודיע לדפדפן היכן הURL של ה applet.
Code – מכיל את הקובץ class. כך שלדפדפן לא תהיה בעיית זיהוי קובץ
Alt – תג אופיונאלי הנותן לנו הצגת מידע קצרה
Name- שם שאנו נותנים ל applet
Width- תג מצוי ב HTML הנותן לנו את רוחב ה applet
Align – אופציה הנותנת לנו כיוון ה applet על המסך קרי ימינה, שמאלה ומרכז המסך.
Vspace & Hspace – המרווחים בין המסך לבין ה applet. שוב אלו תגי HTML הנתונים כאופציה.
Param name & value – תג HTML המאפשר לנו להגדיר ארגומנט ספציפי ל applet כל שאנו נריץ את ה applet הוא יבצע שימוש בפונקצית ()getParameter על מנת לגשת לארגומנט.

.14.3 העברת פרמטרים ב Applet

כפי שהזכרנו השימוש בתג applet בתוך HTML מאפשר לנו להעביר פרמטרים ל applet. על מנת לקבל פרמטרים אנו נשתמש בפונקצית getParameters() . שימוש בפונקציה זו מחזיר לנו את הערך המוגדר כפרמטר ומחזיר לנו אובייקט מסוג String.
לפיכך לשימוש בנתונים מספריים או בוליאניים נצטרך לבצע המרה של ה string לפורמט קריא.

דוגמא

```
//An Applet using parameters

import java.awt.*;
import java.applet.*;
/*
 <applet code = "Parametert" width=300 height=100>
<param name = fontName value = Times>
<param name = fontSize value = 122>
<param name = leading value = 2>
<param name = accountEnabled value = true>
</applet>
*/

public class Parameter extends Applet
{
      String msg = fontName;
      int fontSize;
      float leading;
      boolean active;
}

public void start()
{
String param;

fontName = getParameter ("fontName");
if(fontName = = null)
fontName = "Not Found";

param = getParameter("fontSize");
try
{
if (param != null)
fontSize =Integer.parseInt(param);
else
fontSize = 0;
}
catch (NumberFormatException e)
{
fontSize = -1;
}

param = getParameter("leading");
try
{
if(param != null)
leading = Float.valueOf(param).floatValue();
else
leading = 0;
}
catch (NumberFormatException e)
```

```
{
leading = -1;
}

param = getParameter("accountEnabled");
if (param != null)
active = boolean.valueOf(param).booleanValue();
}

public void paint(Graphics g)
{
g.drawString("Font Name: " + fontName, 0 , 10);
g.drawString("Font Size: " + fontSize, 0 , 20);
g.drawString("Leading : " + leading, 0 , 45);
g.drawString("Account Active : " + active, 0 , 58);
}
}
```

14.4. שימוש בפונקציות getCodeBase() ו getDocumentBase()

לעיתים קרובות נרצה להוסיף ל Applet שלנו תכונות כגון מדיה ואו טקסט. Java מאפשרת לנו לטעון קבצים שכאלו מתוך הספרייה שבה קוד ה HTML מריץ את ה Applet קרי document base והספרייה שבה מחלקת ה Applet נטענת קרי code base.
ספריות אלו מחזיקות אובייקטים מסוג URL על ידי הפונקציות getDocumentBase() ו getCodeBase() .
פונקציות אלו יכולות לשרשר את ה string עם שם הקובץ שאנו רוצים לטון. על מנת לטעון קובץ אחר פיזית אנו נצטרך להשתמש בפונקצית showDocument() אשר מוגדרת על ידי ממשק ה AppletContext שנדון עליו בהמשך.

דוגמא

```
//using setCodeBase() and setDocumentBase()

import java.awt.*;
import java.applet.*;
import java.net.*;

/*
 <applet code = "BaseFunctions" width=300 height=50>
 </applet>
*/

public class BaseFunctions extends Applet
{
      public void paint(Graphics g)
      {
            String msg;

            URL url = getCodeBase();
            msg = "Code base:" + url.toString();

            g.drawString(msg, 10, 20);

            url = getDocumentBase();
            msg = "Document base:" + url.toString();
            g.drawString (msg, 10, 40);
      }
}
```

14.5. שימוש בממשק AppletContext ובפונקציית showDocument()

אחת המתכונות המוצעות לנו על ידי Java הנה שימוש באנימציות או בקטעי וידאו למינהם המשתמשים את המשתמש באינטרנט על מנת לבצע מגוון פעולות לדוגמא קישוריות. על מנת לבצע קישוריות דרך ה Applet אנו נשתמש בפונקציית showDocument() המוגדרת בממשק ה AppletContext כאשר ממשק זה מכיל מידע על הרצת ה Applet.
בתוך ה Applet לאחר שהרצנו את תכולתו נוכל לקרוא לפונקציה נוספת בשם showDocument().
לפונקציה זו אין ערך מוחזר כך שיש להיות זהירים איתה מכיוון שלא נוכל לעקוב אחרי טעות שהתבצעה בפונקציה זו.
ולמעשה קיימות שתי פונקציות showDocument(URL) האחת נותנת לנו מידע על URL ז"א פונקציה זו מציגה את מה שקיים ב URL והפונקציה השנייה showDocument(URL where) מציגה את המקום המוגדר למסמך בחלון הדפדפן. הארגומנטים המשוייכים לפונקציה זו הנם "_self" "_parent" "_top" "_blank". נוכל כמובן להגדיר שם חדש אשר יגרום למסמך להיות מוצג בחלון חדש.

דוגמא

בדוגמא זו ה Applet ירוץ ויחכה עד אשר המשתמש ילחץ על כפתור העכבר. לאחר לחיצת המשתמש יוצג המידע שבתוך ה applet ואז נשתמש בשליטה הניתנת לנו על ידי הפונקציות להעברת מידע לקובץ הנקרא test.hmt. נשים לב כי הקובץ חייב להיות באותה ספרייה בה הapplet נמצא.

```
//using AppletContext and getCodebase()

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
 <applet code = "Demo" width=300 height=50>
 </applet>
*/
public class Demo extends Applet
{
	String msg;

	public void start()
	{
		msg = "Click mouse to view URL";
	}

	public boolean mouseDown(Event evtObject, int x, int y)
	{
		AppletContext ac = getAppletContext();
		URL url = getCodeBase();

		try
		{
			ac.showDocument (new URL(url + "test.htm"));
		}
		catch (MalformedURLException e)
		{
			showStatus(" URL NOT FOUND");
		}
		return true;
	}

	public void paint(Graphics g)
	{
		g.drawString(msg, 10, 20);
	}
}
```

.15 חלונות, גרפיקה וטקסט תוך שימוש ב AWT וב Swing

כפי שכבר הוסבר AWT נותן לנו תמיכה באפליקציות גרפיקה. בחלק זה נבחן את הנושא לעומקו תוך מתן דגש על בעיות בעיצוב GUI ופתרונן.
AWT מכיל מספר של מחלקות ופונקציות אשר בעזרתן נוכל לבצע את מכלול פעולותינו על מנת לייצור חלונות. תיאור מלא של AWT הנו חיבור בפני עצמו ועל כן אנו נתרכז בפעולות החשובות לנו בהנדסת תוכנה.

.15.1 מחלקות AWT

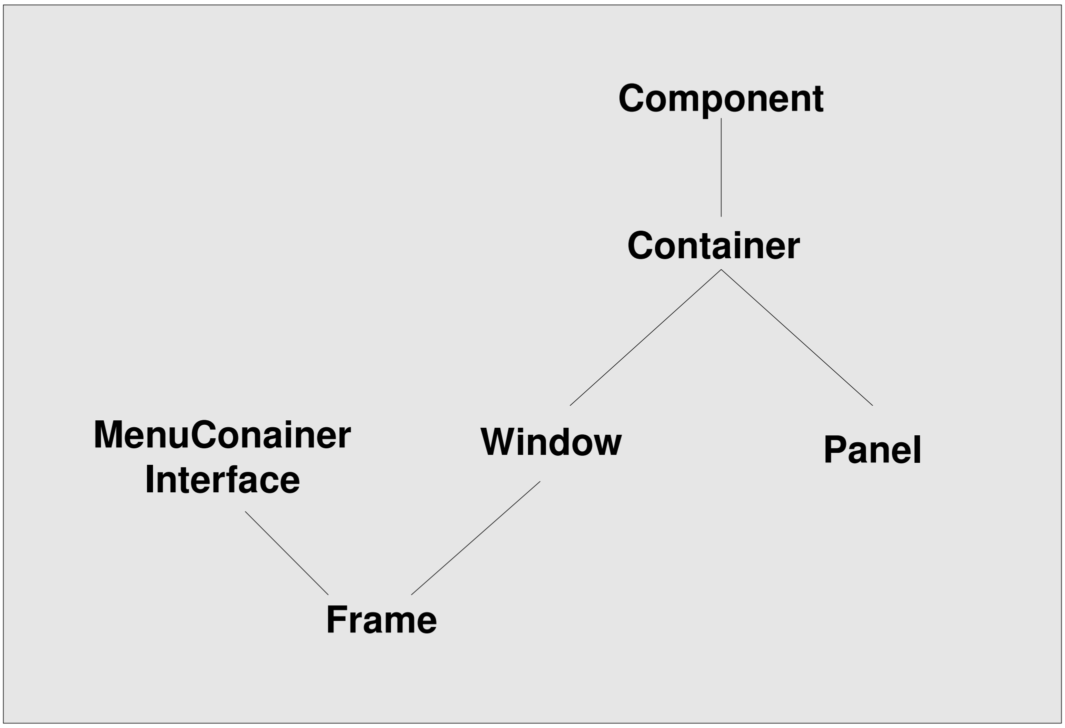
כלל מחלקות AWT מאוכלסות בחבילת java.awt . כלל המחלקות מתארות את השימוש ב AWT.
בנוסף לכלל המחלקות AWT מגדיר שני ממשקים:

1. LayoutManager
2. MenuContainer

שני ממשקים אלו מגדירים לנו את מאפייני הממשק אשר יהיו מיושמים על ידי כלל המחלקות שאנו נשתף בתכנית.

.15.2 תכנות ראשוני בחלונות

AWT מגדיר חלונות בהתאם להיררכיה של מחלקות המוסיפות פונקציונאליות בכל שלב כלשהו. שני החלונות העיקריים המוכרים הנם אלו הנגזרים מ Panel אשר משתמש ב applet ואלו הנגזרים מ Frame אשר יוצרות לנו חלון סטנדרטי.
מהדיאגרמה להלן נוכל להבין את אופי הפעולה של AWT ותת מחלקותיו:



15.2.1. רכיב - Component

בראש ההיררכיה של AWT נמצא Component אשר הנו בעצם מחלקת abstract המשתמשת בתכנות הריכוזיות של כלל המאפינים הויזואליים. כלל הממשקים המיוצגים על המסך המשתמש הנם תת מחלקה של Component.
מחלקה זו מכילה מאות פונקציות ציבוריות אשר אחראיות לניהול מאורעות כגון שימוש בעכבר או מקלדת.

15.2.2. תכולה - Container

מחלקת Container הנה תת מחלקה abstract של מחלקת Component. למחלקה זו קיימות תכונות נוספות ופונקציות נוספות המאפשרות למחלקת Component להיות מוכלים בתוכה.
מחלקה זו אחראית למערך התוצאה של הרכיבים המוכלים בה.

15.2.3. פאנל - Panel

מחלקת Panel הנה תת מחלקה מקבילית ל Container. מחלקה זו איננה מוסיפה שום פונקציות חדשות. באופי פעולתה היא מיישמת את מחלקת Container.
Panel הנה מחלקת העל למחלקת Applet. כאשר אנו מריצים Applet המצטייר למסך הוא מצטטיר בעזרת ה"משטח" של Panel.

15.2.4. חלון - Window

מחלקת window יוצרת top level window. חלון שכזה איננו מוכל בשום אובייקט קרי הוא נמצא פיזית ישירות במסך המשתמש. בד"כ איננו יוצרים חלון שכזה באופן ישיר אלא, אנו ניצור frame ובתוכו ירוץ החלון.

15.2.5. מסגרת - Frame

Frame משתמש בתכונת ריכוזיות קרי חלונות. הנה תת מחלקה של window שקיימים בו תפריטים, כותרות וכו'. כאשר אנו יוצרים אובייקט Frame בתוך applet הוא יכלול הודעת אזהרה למשתמש ותכולתה כי נוצר למשתמש חלון במסך.
במקרה שבו applet ירוץ מתוך הדפדפן של המשתמש ההודעה תהיה שונה קרי בדפדפנים החדשים היום ההודעה הנה פנימית.

15.2.6. חלון ריק - Canvas

למרות שזהו איננו חלק מההיררכיה ישנו חלון אחד נוסף שנרצה להציג כאן. חלון זה הנו Canvas. חלון זה משתמש בתכונת הריכוזיות ובעצם מרכז חלון ריק שעליו בעצם נוכל לצייר גרפיקה.

15.3. עבודה עם חלונות ומסגרות

לאחר שיצרנו applet החלון השימושי הבא הנו יווצר מתוך frame. אנו נשתמש בו על מנת לייצור חלון "ילד" בתוך ה applet אשר ישמש אותנו להרצת אפליקציות. השימוש מוגדר על ידי שני קונסטרקטורים אשר מבצעים פעולות שונות.
הצורה הכללית הנה:

```
Frame()
Frame(String title)
```

כאשר:
הקונסטרקטור הראשון יוצר חלון סטנדרטי שלא מכיל כותרת.
הקונסטרקטור השני יוצר חלון המכיל כותרת על ידי הגדרת string.

נשים לב כי איננו יכולים לקבוע את גודל החלון על פי צירי X Y אלא אנו נותנים את גודל החלון לאחר שהוא נוצר.

15.3.1. פונקציות נוספות לשימוש עם Frame:

- **מתן מידות לחלון:**

על מנת לתת מידות לחלון אשר יצרנו נוכל להשתמש על ידי פונקצית ()resize. המבנה הכללי הנו:

```
void resize(int newWidth, int newHeight)

void resize(Dimension newSize)
```

המידות החדשות של החלון מוגדרות על ידי newWidth ו newHeight או על ידי אובייקט בשם Dimension תוך שימוש ב newSize.
נשים לב כי המידות מוגדרות ב pixels.

כמובן שנוכל לדעת מהי מידת החלון על ידי שימוש בפונקצית ()size באופן הבא:

```
Dimension size()
```

פונקציה זו נותנת לנו את המידה של החלון המוכל בתוך שדות width ו height אשר הנם חלק מאובייקט Dimension.

- **הסתרה/מראה של חלון:**

לאחר שיצרנו חלון מסוג Frame , חלון זה אינו ויזואלי עד אשר ניצור את פונקצית ()show. על מנת להסתיר את החלון אנו נשתמש בפונקצית ()hide.
הפונקציות הנן כדלהלן:

```
void hide()
void show()
```

- **מתן כותרת לחלון:**

נוכל לשנות את כותרת מסגרת החלון תוך שימוש בפונקצית ()setTitle אשר לה קיים מבנה כללי של:

```
void setTitle(String newTitle)
```

15.4. מאורע WINDOW_DESTROY

כלל האובייקטים הנגזרים מ Component מקבלים מאורעות כגון לחיצה על עכבר או מקלדת. לפיכך כלל frame windows יכולות להיות פונקציות משוכתבות כגון mouseDown() או keyDown() כפי שתארנו בחלק ה Applet.
בנוסף ישנו מאורע אחד שנצטרך לטפל בו והוא סגירת החלון. על מנת להגיב למאורע זה ב applet שלנו יהיה חייב להיות משוכתב החלק המטפל במאורע קרי פונקצית handleEvent().
המבנה הכללי הנו:

boolean handleEvent(Event evtObj)

פונקצית handleEvent() הנה הרמה העליונה לחלון שלנו. כלל המאורעות המתייחסות לחלון שלנו עוברות דרך פונקצית handleEvent(). בכל פעם שאנו קוראים לפונקציה זו המכילה מאורע מסוים. אשר אנו משכתבים את פונקצית handleEvent() אנו מחזיקים ערך true במידה ואנו מטפלים במאורע. אחרת אנו נצטרך להעביר את המאורע דרך תת המחלקות דרך super.handleEvent().
בתוך פונקצית handleEvent() אנו יכולים לקבוע איזה מאורע יקרה על ידי הגדרה ב id של השדה הקשור למאורע המעביר את הארגומנט.
מחלקת Event מכילה מספר קבועים שהנם קוד ID למספר מאורעות כך שבכל פעם שמאורע קורה אזי כי רץ ה ID של המאורע.
להלן מבנה השלד המשוכתב של פונקצית handleEvent() המטפלת ב WINDOW_DESTROY:

```
public boolean handleEvent(Event evtObject)
	{
	if (evtObject.id == Event.WINDOW_DESTROY)
	{
	// respond to event
	return true;
	}
	return super.handleEvent(evtObject);
 }
```

15.5. יצירת חלון מסגרת בתוך Applet

יצירת חלון מסגרת מתוך applet הנה תהליך פשוט למדי. שלב ראשון אנו יוצרים תת מחלקה ל Frame. אנו משכתבים את כלל הפונקציות הסטנדרטיות לחלונות כגון init(), start(), stop(), paint() . אנו חייבים כמובן לשכתב גם את פונקצית handleEvent() אשר בעצם תסתיר את החלון כאשר נשתמש בפונקצית WINDOW_DESTROY.
לאחר שהגדרנו את תת המחלקה של Frame אנו יכולים לייצור אובייקט לאותו סוג מחלקה. אובייקט זה יגרום לחלון במסגרת להיות קיים אך ויזואלית לא נראה אותו עדיין. על מנת לראותו נשתמש בפונקצית show(). כאשר אנו יוצרים חלון קיימים ערכי default לגובה ולרוחב. נוכל כמובן להשתמש בפונקציות לשינוי מידות החלון resize().

דוגמא

בדוגמא זו אנו יוצרים applet אשר יוצר תת מחלקה של Frame הנקרא SampleFrame. חלון של תת מחלקה זו יווצר ברגע שנקרא לפונקצית init() של AppletFrame.
נשים לב בתכנית כי AppletFrame קורא לקונסטרקטור של Frame. קריאה זו גורמת לחלון להיווצר עם כותרת המועברת כ string.
בדוגמא זו נשכתב שוב את פונקציות start() ו stop() כך שהן מראות ומסתירות את חלון "הבן" בהתאמה. פעולה זו גורמת לחלון להיסגר.

```
//Creating a child frame window from within Applet

import java.awt.*;
import java.applet.*;

/*
 <applet code= "AppletFrame" width=300 height=50
 </applet>
 */

//Create a subclass of Frame
class SampleFrame extends Frame
{
	SampleFrame(String title)
	{
		super(title);
	}
	//Hide window when terminate by user
	public boolean handleEvent(Event evtObject)
	{
		if(evtObject.id == Event.WINDOW_DESTROY)
		{
			hide();
			return true;
		}
		return super.handleEvent(evtObject);
	}
	public void paint (Graphics g)
	{
		g.drawString("This is Frame Window", 10, 20);
	}
}

	//Create the Applet Window
public class AppletFrame extends Applet
{
	SampleFrame f;

	//Create a Frame Window
	public void init()
	{
		f = new SampleFrame("Frame Window");
		f.show();
		f.resize(250, 100);
	}

	// Remove fame window when Stopping the Applet
	public void stop()
	{
		f.hide();
	}

	//Show Frame window when starting Applet
```

```
	public void start()
	{
		f.show();
	}

	//Display msg in Applet
	public void paint (Graphics g)
	{
		g.drawString ("this is in Applet", 10, 20);
	}
}
```

15.6. טיפול במאורע של חלון עם מסגרת

מאחר ש Frame הנו תת מחלקה של Component הוא מוריש את כלל היכולות המוגדרות על ידי Component. ומכאן זה אומר שנוכל להשתמש ולנהל את החלון עם המסגרת שיצרנו כמו שאנו מנהלים חלון ראשי עם Applet. לדוגמא, נוכל לשכתב את פונקצית paint() להראות לנו תוצאה, ואז לקרוא לפונקצית repaint() כאשר נרצה לאתחל את החלון ואז נוכל לשכתב את כלל הפונקציות המטפלות במאורעות.
בשלב כלשהו שמאורע קורה בחלון הטיפול במאורע המוגדר על ידי החלון יכול להיקרא כך שכל מטפל בודד במאורע אחראי למאורע שלו.

דוגמא
בדוגמא זו ניצור חלון אשר יגיב למאורע עם עכבר. ה Applet הראשי מגיב גם למאורע עם העכבר.

```
//Creating a child frame window from within Applet

import java.awt.*;
import java.applet.*;

/*
 <applet code= "WindowEvents" width=300 height=50
 </applet>
 */

//Create a subclass of Frame
class SampleFrame extends Frame
{
      String msg = "";
      int mouseX=0, mouseY=10;
      int movX=0, movY=0;

      SampleFrame(String title)
      {
            super(title);
      }

      //Hide window when terminate by user
      public boolean handleEvent(Event evtObject)
      {
            if(evtObject.id == Event.WINDOW_DESTROY)
            {
                  hide();
                  return true;
            }
            return super.handleEvent(evtObject);
      }

      // Handle button Press
      public boolean mouseDown(Event evtObject, int x, int y)
      {
            //Save Coordinates
            mouseX = x;
            mouseY = y;
            msg = "Down";
            repaint();

            return true;
      }

      //Handle Press release
```

```
public boolean mouseUp(Event evtObject, int x, int y)
{
      //Save Coordinates
      mouseX = x;
      mouseY = y;
      msg = "Up";
      repaint();

      return true;
}

//Handle Mouse Move
public boolean mouseMove(Event evtObject, int x, int y)
{
      //Save Coordinates
      mouseX = x;
      mouseY = y;
      repaint(0, 0, 100, 20);

      return true;
}

// Handle mouse drag
public boolean mouseDrag(Event evtObject, int x, int y)
{
      //Save Coordinates
      mouseX = x;
      mouseY = y;
      movX = x;
      movY = y;
      msg = "*";
      repaint();

      return true;
}

//Handle mouse enter
public boolean mouseEnter(Event evtObject, int x, int y)
{
      //Save Coordinates
      mouseX = 0;
      mouseY = 24;
      msg = "Mouse Just Entered Sub Window";
      repaint();

      return true;
}

//Handle mouse exit
public boolean mouseExit(Event evtObject, int x, int y)
{
      //Save Coordinates
      mouseX = 0;
      mouseY = 24;
      msg = "Mouse Just left Sub Window";
      repaint();

      return true;
}

//
```

```
	public void paint (Graphics g)
	{
		g.drawString(msg, mouseX, mouseY);
		g.drawString("Mouse at" + movX + ", " + movY, 0, 10);
	}
}


	//Create the Applet Window
public class WindowEvent extends Applet
{
	SampleFrame f;
	String msg = "";
	int mouseX=0, mouseY=10;
	int movX=0, movY=0;


	//Create a Frame Window
	public void init()
	{
		f = new SampleFrame("Frame Window");
		f.show();
		f.resize(300, 250);
	}

	// Remove fame window when Stopping the Applet
	public void stop()
	{
		f.hide();
	}

	//Show Frame window when starting Applet
	public void start()
	{
		f.show();
	}

	//Handle button Press
	public boolean mouseDown(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = x;
		mouseY = y;
		msg = "Down";
		repaint();

		return true;
	}

	//handle button Release
	public boolean mouseUp(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = x;
		mouseY = y;
		msg = "Up";
		repaint();

		return true;
	}
```

```java
	//Handle Mouse Move
	public boolean mouseMove(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = x;
		mouseY = y;
		repaint(0, 0, 100, 20);

		return true;
	}

	//Handle mouse enter
	public boolean mouseEnter(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = 0;
		mouseY = 24;
		msg = "Mouse Just Entered Applet Window";
		repaint();

		return true;
	}

	//Handle mouse exit
	public boolean mouseExit(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = 0;
		mouseY = 24;
		msg = "Mouse Just left Applet Window";
		repaint();

		return true;
	}

	//Display msg in Applet
	public void paint (Graphics g)
	{
		g.drawString (msg, mouseX, mouseY);
		g.drawString ("Mouse at" + movX + ", " + movY, 0, 10);
	}
}
```

15.7. יצירת חלון עצמאי

למרות שיצירת Applet הנה הדרך הטובה ביותר על מנת להשתמש בתכונות AWT ניתן לייצור חלון כאפליקציה עצמאית. על מנת לבצע זאת אנו ניצור instance של חלון אשר נכלול בתוכו פונקצית main() .

דוגמא

בדוגמא זו ניצור חלון עצמאי המגיב למאורע של לחיצת עכבר או מקלדת.

```
//Creating AWT based Application

import java.awt.*;
import java.applet.*;


//Create rame window
public class AppWindow extends Frame
{
	String keymsg = "";
	String mousemsg = "";
	int mouseX=0, mouseY=0;

	//Handle keypress Event
	public boolean keyDown(Event evtObject, int key)
	{
		keymsg += (char) key;
		repaint();
		return true;
	}

	//Handle mouse down Event
	public boolean mouseDown(Event evtObject, int x, int y)
	{
		//Save Coordinates
		mouseX = x;
		mouseY = y;
		mousemsg = "Mouse Down at" + x + "," + y;
		repaint();

		return true;
	}

	//Close Window
	public boolean handleEvent(Event evtObject)
	{
		if (evtObject.id == Event.WINDOW_DESTROY)
			System.exit(0);
		return super.handleEvent(evtObject);
	}

	//Display key and mouse msgs
	public void paint (Graphics g)
	{
		g.drawString (keymsg, 0, 10);
		g.drawString (mousemsg, mouseX, mouseY);
	}
	//Create the Window
	public static void main(String args[])
	{
		AppWindow appwin = new AppWindow();
```

```
            appwin.resize(300, 200);
            appwin.setTitle ("AWT Base App.");
            appwin.show();
      }
}
```

16. פונקציות גרפיקה תוך שימוש ב Java2D

AWT תומך במגוון עשיר של פונקציות גרפיקה. כל הפונקציות המצויירות הנן "קרובות" לחלון. ז"א זה יכול להיות applet או חלון "ילד" של applet או אפליקציה בפני עצמה.
המצב הראשוני המצטייר לכל חלון הנו בקואורדינטות של 0, 0 כאשר אלו (=קואורדינטות) מצויינות ב pixels.
חלון הגרפיקה משתמש בתכונת הריכוזיות תוך שימוש במחלקת Graphics וניתן לבצע זאת ל ידי שתי דרכים:

1. כאשר הפונקציה מועברת ל applet והיא נקראת לדוגמא paint(), update().
2. כאשר נחזיר ערך לפונקצית getGraphics().

לצורך כלל הדוגמאות אנו נשתמש ב applet על מנת להציג גרפיקה אולם לכלל הטכניקות הקיימות הקוד זהה.
מחלקת Graphics מגדירה מספר פונקציות גרפיות. כאשר כל צורה יכולה להיות מצויירת ללא מילוי או מצוריית עם מילוי. האובייקטים מצוירים על ידי סדר הפונקציות הנקראות כאשר ה default הנו צבע שחור.
כאשר אובייקט מצטייר מחוץ לגבולות החלון שהגדרנו הוא באופן אוטומטי נחתך.
הבא נבחן את האלמנטים הקימים בגרפיקה AWT. יש לציין כי את אותם רכיבים ניתן כמובן לתכנת גם בעזרת מודל ה Swing.

16.1.1 ציור קווים

קווים ניתן לצייר על ידי שימוש בפונקצית drawLine(). המבנה הכללי הנו:

void drawLine(int startX, intStartY, int endX, intEndY)

כאשר פונקצית drawLine() הנה הפונקציה הנוכחית לשימוש בהגדרות X Y.

דוגמא

בדוגמא זו נצייר מספר קווים באופן פשוט

```
//Draw Line

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Lines" width=250 height=250>
  </applet>
*/

public class Lines extends Applet
{
      public void paint(Graphics g)
      {
            g.drawLine(0, 0 , 100, 100);
            g.drawLine(0, 100 , 100, 0);
            g.drawLine(40, 25 , 250, 180);
            g.drawLine(20, 150 , 400, 40);
            g.drawLine(5, 290 , 80, 29);
      }
}
```

16.1.2. ציור ריבוע

על מנת לצייר ריבוע נוכל להשתמש בשתי פונקציות עיקריות האחת drawRect() שתפקידה לצייר ריבוע ריק והאחרת fillRect() לצייר ריבוע מלא.
המבנה הכללי הנו:

void drawRect(int top, int left, int width, int height)

void fillRect(int top, int left, int width, int height)

כאשר הקצה העליון של הריבוע מיוצג על ידי top, left ומידות המרובע מיוצגות על ידי width, height.

על מנת לצייר ריבוע עם פינות מעוגלות אנו נשתמש בפונקציות drawRoundRect() וב fillRoundRect().
המבנה הכללי של הפונקציות הנו:

void drawRoundRect(int top, int left, int width, int height, intXDiam, intYDiam)

void fillRoundRect(int top, int left, int width, int height, intXDiam, intYDiam)

כאשר נשים לבכי לריבוע עם פינות מעוגלות אנו נוסיף את הגדרות העיגול על ידי הגדרה ב Xdiam וב Ydiam.

דוגמא

```
//Draw Rounded Rect

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Rect" width=250 height=250>
  </applet>
*/

public class Rect extends Applet
{
      public void paint(Graphics g)
      {
            g.drawRect(10, 10 , 60, 50);
            g.fillRect(100, 10 , 60, 50);
            g.drawRoundRect(190, 10 , 60, 50, 15, 15);
            g.fillRoundRect(70, 90, 140, 100, 30, 40);
      }
}
```

16.1.3. ציור אליפסות ומעגלים

על מנת לצייר אליפסה אנו נשתמש בפונקצית drawOval() . על מנת למלא את האליפסה אנו נשתמש בפונקצית fillOval().
המבנה הכללי הנו:

void drawOval(int top, int left, int width, int height)

void fillOval(int top, int left, int width, int height)

דוגמא

```
//Draw Elipes and Circle

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Elips" width=250 height=250>
  </applet>
*/

public class Elips extends Applet
{
      public void paint(Graphics g)
      {
            g.drawOval(10, 10 , 50, 50);
            g.fillOval(100, 10 , 75, 50);
            g.drawOval(190, 10 , 90, 30);
            g.fillOval(70, 90, 140, 100);
      }
}
```

16.1.4. ציור Arc

על מנת שנוכל לציר קשת נשתמש בפונקציות drawArc() וב fillArc(). המבנה הכללי הנו:

void drawArc(int top, int left, int width, int height, int StartAngle, int SweepAngle)

void fillArc(int top, int left, int width, int height, int StartAngle, int SweepAngle)

הקשת מוכלת בתוך ריבוע שאת הקוארדינאטות אנחנו קובעים. הקשת מצטיירת תוך שימוש ב StartAngle המוגדר על ידי SweepAngle .

דוגמא

```
//Draw Arcs

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Arc" width=250 height=250>
  </applet>
*/

public class Arc extends Applet
{
      public void paint(Graphics g)
      {
            g.drawArc(10, 40 , 70, 70, 0, 75);
            g.fillArc(100, 40 , 70, 70, 0, 75);
            g.drawArc(10, 100 , 70, 80, 0, 125);
            g.fillArc(100, 100, 70, 90, 0, 270);
            g.drawArc(200, 80, 80, 80, 0,180);
      }
}
```

16.1.5. ציור Polygons

על מנת לצייר פוליגוניים אנו נשתמש בפונקציות drawPolygon() וב fillPolygon(). המבנה הכללי הנו:

void drawPolygon (int x[], int y[], int numPoints)

void fillPolygon(int x[], int y[], int numPoints)

נוכל לראות מהמבנה הכללי כי אנו נצטרך לקבוע את נקודות תחילתו וסיומו של הפוליגון.

דוגמא

```
//Draw Polygon

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Poly" width=250 height=250>
  </applet>
*/

public class Poly extends Applet
{
      public void paint(Graphics g)
      {
            int xpoints[] = {30, 200, 30, 200, 30};
            int ypoints[] = {30, 30, 200, 200, 30};
            int num = 5;
            g.drawPolygon( xpoints, ypoints, num);
      }
}
```

16.1.6. מתן מידות לגרפיקה

קיימים מצבים בהם נרצה לקבוע את מידות הגרפיקה על המסך בתוך החלון המופיע. על מנת לבצע זאת, אנו נקרא בשלב ראשון לפונקציית ()size אשר נמצאת בתוך מחלקת Window ומהווה אובייקט ממנו. המבנה הכללי הנו:

Dimension size()

שאנו נשתמש בפונקציה זו אנו בעצם מרכזים את המידות בתוך אובייקט Dimension.
לאחר שיש לנו את מידות החלון נוכל להוציא לפועל את הגרפיקה שלנו באופן מדוייק.

על מנת להבין כיצד הטכניקה הזו עובדת הרי דוגמא. נתון לנו כי גודלו של applet הנו 200*200 pixels מרובע. ה applet מגדיל את אורכו ורוחבו בכל לחיצת כפתור על העכבר עד אשר הוא מגיע לגודל מקסימלי של 500*500.
בנקודה זו כאשר נלחץ שוב על העכבר גודל החלון יחזור להיות 200*200 והתהליך חוזר חלילה. בתוך החלון קיים ריבוע המצוייר בתוך הגבול הפנימי של החלון. בתוך המרובע X מצוייר כך שהוא ימלא את החלון.

דוגמא

```
//Resizing window dimensions

import java.awt.*;
import java.applet.*;

/*
  <applet code = "Resize" width=200 height=200>
  </applet>
*/

public class Resize extends Applet
{
      final int increment = 25;
      int max = 500;
      int min = 200;
      Dimension d;

      public void paint(Graphics g)
      {
            d = size();

            g.drawLine(0, 0 ,d.width-1, d.height-1);
            g.drawLine(0, d.height-1, d.width-1, 0);
            g.drawRect(0, 0 ,d.width-1, d.height-1 );
      }
      public boolean mouseUp (java.awt.Event evt, int x, int y)
      {
            int w = (d.width + increment) > max?min : (d.width +
increment);
            int h = (d.height + increment) > max?min : (d.height +
increment);
            resize(w, h);
            return true;
      }
}
```

16.2. עבודה עם צבעים

שפת java תומכת בצבעם בשיטה הישנה והבטוחה. מערכת AWT מאפשרת לנו להגדיר איזה סוג צבע שנרצה. לאחר שהגדרנו את הצבע מערכת AWT מבצעת התאמת חומרה לבדיקה. אי לכך אל לנו לדאוג לרמת ההתאמה המערכת מבצעת זאת באופן אוטומטי. התאמת הצבעים מתבצעת תוך שימוש במחלקת Color המשתמשת בתכונות הריכוזיות ובעצם מכילה את מכלול הצבעים. למחלקה זו קיימים מספר קבועים כפי שכבר ראינו לדוגמא Color.cayn.
נוכל כמובן לייצור את הצבעים שאנו רוצים על ידי הגדרות RGB תוך שימוש במבני הקונסטרקטורים הבא:

Color(int red, int green, int blue)

Color (int rgb Value)

Color (float red, float green, float blue)

כאשר:
בקונסטרקטור הראשון נוכל להגדיר את הצבעים על פי המספרים המוכרים לנו.
Color(200, 100, 150)

בקונסטרקטור השני ניקח int בודד המכיל את כלל RGB יחדיו. ה int מאורגן בצורה שכזו
Red 16 to 23 bit, green 8 to 15 bit, blue 0 to 7 bit

```
int newColor = (0xff0000 | (0xc0 << 16) | (0x00 <<8) | 0x00)
color darkRed = new Color (newColor);
```

בקונסטרקטור השלישי אנו לוקחים ערכים floats של RGB בין 0.0 ל 1.0.

16.2.1. פונקציות צבע

שימוש ב Hue, Saturation and Brightness

מודל ה HSB הנו אלטרנטיבה ל RGB להגדרת צבעים נקודתית. ה Hue מוגדר עם ערכים מספריים של בין 0.0 ל 1.0. ה Saturation מוגדר כסקלה של ערכים בין 0.0 ל 1.0 המייצגים צבעים בהירים יותר ל Hue. ה Brightness מיוצג על ידי ערכים מספריים בין 0.0 ל 1.0 כאשר 1 הנו ערך בהיר מאוד ו 0 הנו ערך שחור.
מחלקת Color נותנת לנו שימוש בשתי פונקציות על מנת לבצע המרה מ RGB ל HSB:

static int HSBtoRGB(float Hue, float Stauration, float Brightness)

static float[] RGBtoHSB(int red, int green, int blue, float values[])

שימוש בפונקציות getRed(), getBlue(), getGreen()

נוכל להשתמש בצבעים בודדים על ידי שימוש בפונקציות הנ"ל. המבנה הכללי הנו:

int getRed()

int getGreen()

int getBlue()

שימוש בפונקצית getRGB

על מנת לקבל מארז מלא של צבעי המסך נוכל להשתמש בפונקצית getRGB(). המבנה הכללי הנו:

```
int getRGB()
```

אתחול צבע גרפי לשימוש

ברירת המחדל הנה ציור הגרפיקה תוך שימוש בצבע ה foreground. נוכל לשנות את צבע הגרפיקה על ידי קריאה לפונקציה הגרפית setColor(). המבנה הכללי הנו

```
void setColor(Color newColor)

Color getColor()
```

דוגמא לשימוש בפונקציות שהוסברו:

```
// Demonstrating Colors
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ShowColors extends JFrame {
  public ShowColors()
  {
    super( "Using colors" );

    setSize( 400, 130 );
    show();
  }
  public void paint( Graphics g )
  {
    // set new drawing color using integers
    g.setColor( new Color( 255, 0, 0 ) );
    g.fillRect( 25, 25, 100, 20 );
    g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
    // set new drawing color using floats
    g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
    g.fillRect( 25, 50, 100, 20 );
    g.drawString( "Current RGB: " + g.getColor(), 130, 65 );
    // set new drawing color using static Color objects
    g.setColor( Color.blue );
    g.fillRect( 25, 75, 100, 20 );
    g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
    // display individual RGB values
    Color c = Color.magenta;
    g.setColor( c );
    g.fillRect( 25, 100, 100, 20 );
    g.drawString( "RGB values: " + c.getRed() + ", " +
      c.getGreen() + ", " + c.getBlue(), 130, 115 );
  }
  public static void main( String args[] )
  {
    ShowColors app = new ShowColors();

    app.addWindowListener(
      new WindowAdapter() {
        public void windowClosing( WindowEvent e )
        {
          System.exit( 0 );
        }
      }
    );
  }
}
```

16.2.2. איפוס ה Mode לעבודה

paint mode קובע לנו כיצד האובייקטים יצויירו על החלון במסך. ברירת המחדל הנה שכל תוצר חדש משכתב את הקיים. אולם ישנה אפשרות שהיה לנו אובייקט חדש מסוג XORed לתוך החלון תוך שימוש בפונקצית ()setXORMode. המבנה הכללי הנו:

void setXORMode (Color xorColor)

כאשר:
XorColor מגדיר את הצבע אשר יתממשק לחלון כאשר האובייקט יאותחל. היתרון בשימוש ב XOR הנו שאובייקט חדש תמיד יהיה ויזואלי ולא חשוב באם קיים אובייקט אחר.

על מנת לחזור למצב הראשוני אנו נשתמש בפונקצית ()setPaintMode. המבנה הכללי הנו:

void setPaintMode()

דוגמא

```
//Demonstare XOR Mode

import java.awt.*;
import java.applet.*;

/*
 <applet code = "XOR" width=300 height=200>
 </applet>
 */

public class XOR extends Applet
{
      int cshX=100, cshY=100;

      //mouse track
      public boolean mouseMove(Event evtObject, int x, int y)
      {
            cshX= x-10;
            cshY= y-10;
            repaint();
            return true;
      }

      //draw lines
      public void paint(Graphics g)
      {
            g.drawLine(0, 0, 100, 100);
            g.drawLine(0, 100, 100, 0);

            g.setColor(Color.blue);
            g.drawLine(40, 25, 250, 180);
            g.drawLine(75, 90, 400, 400);

            g.setColor(Color.green);
            g.drawRect(10, 10, 60, 50);
            g.fillRect(100, 10, 60, 50);

            g.setColor(Color.red);
            g.drawRoundRect(190, 10, 60, 50, 15, 15);
            g.fillRoundRect(70, 90, 140, 100, 30, 40);
```

```
            g.setColor(Color.cyan);
            g.drawLine(20, 150, 400, 40);
            g.drawLine(5, 290, 80, 19);

            //XOR MODE
            g.setXORMode(Color.black);
            g.drawLine(cshX-10, cshY, cshX+10, cshY);
            g.drawLine(cshX, cshY-10, cshX, cshY+10);
            g.setPaintMode();
      }
}
```

16.3. שימוש בפונטים

AWT תומך במספר פונטים רב. במרוצת השנים האחרונות החל משנת 95/94 ניתן דגש מיוחד על שימוש בפונטים וזאת על מנת לתת מראה נקי וברור למשתמש.
הפונטים נמצאים מרוכזים במחלקת Font וישנם מספר פונקציות לשימושם.

מחלקת Font מגדירה את המשתנים הבאים:

Variable	Meaning
String name	Name of the font
int size	Size of the font in points
int style	Font style

16.3.1. קביעת פונטים

כאשר אנו עובדים עם פונטים נרצה לדעת אילו פונטים שמישים לנו על המחשב. על מנת לקבל את המידע הזה נשתמש בפונקצית getFontList() המוגדרת במחלקת Toolkit. המבנה הכללי הנו:

String[] getFontList()

כאשר:
השימוש בתחביר זה מחזיר לנו מערך של strings המכיל את שמות הפונטים המצויים לנו במערכת.
מאחר ש getFontList() הנו חבר של Toolkit אנו נצטרך התייחסות לקרוא לו. נוכל להשתמש בהתייחסות על ידי שימוש בפונקצית getToolkit() אשר מוגדרת גם ב Component וגם ב Window.
ולפיכך נוכל להשתמש בפונקצית getToolkit() Toolkit .

דוגמא

```
//Fonts

import java.awt.*;
import java.applet.*;

/*
 <applet code = "Fonts" width=1700 height=50>
 </applet>
 */

public class Fonts extends Applet
{
	public void paint(Graphics g)
	{
		String msg = " ";
		String FontList[];

		FontList = getToolkit().getFontList();

		for(int i=0; i<FontList.length; i++)
			msg += FontList[i] + " ";

		g.drawString(msg, 5, 150);
	}
}
```

16.3.2. יצירה ושימוש בפונטים

על מנת לבחור פונט עלינו לבנות אובייקט מסוג פונט המתאר את הפונט. הקונסטרקטור של הפונט בצורתו הכללית הנו:

Font(String fontName, int fontStyle, int pointSize)

כאשר:
FontName הנם הפונטים הקיימים במערכת כגון Dialog, Times וכו' ובד"כ הם קיימים בכל מערכות ההפעלה.
FontStyle הנו הסגנון של הפונט שאותו נגדיר. נוכל להשתמש באחד מהשלושה הבאים: Font.BOLD, Font.PLAIN, Font.Italic או בשלושתם או לבצע הרכבה.

על מנת להשתמש בפונטים שאנו יצרנו אנו חייבים לבחור אותם תוך שימוש בפונקצית setFont() אשר מוגדרת על ידי Component. המבנה הכללי הנו:

void setFont(Font fontObject)

כאשר:
FontObject הנו האובייקט המכיל את הפונט שיצרנו.

דוגמא
בדוגמא זו נכלול את כל ברירות המחדל של הפונטים. בנוסף בכל פעם שנלחץ על העכבר הפונט ישתנה.

```
//Default Fonts

import java.awt.*;
import java.applet.*;

/*
 <applet code = "FontsTest" width=200 height=100>
 </applet>
 */

public class FontsTest extends Applet
{
      int next = 0;
      Font f;
      String msg;

      public void init()
      {
            f = new Font("Dialog", Font.PLAIN, 12);
            msg = "Dialog";
            setFont(f);
      }

      //switch fonts when mouse is clicked
      public boolean mouseDown(Event evtObject, int x, int y)
      {
            next++;
            switch(next)
            {
                  case 0:
                        f = new Font("Dialog", Font.PLAIN, 12);
                        msg = "Dialog";
                        break;
```

```
                        case 1:
                              f = new Font("Helvetica", Font.PLAIN, 12);
                              msg = "Helvetica";
                              break;

                        case 2:
                              f = new Font("TimesRoman", Font.PLAIN, 12);
                              msg = "TimesRoman";
                              break;

                        case 3:
                              f = new Font("Courier", Font.PLAIN, 12);
                              msg = "Courier";
                              break;
                  }

                  if (next == 3) next = -1;
                  setFont(f);
                  repaint();
                  return true;
            }

            public void paint(Graphics g)
            {
                  g.drawString(msg, 5, 150);
            }
}
```

.16.4 מחלקת Font Matrics

על מנת לשלוט במגוון הפונטים ובצוותיהם כגון רוחב, גובה וכדומה קיימת לנו מחלקה בשם FontMatrics המכילה מגוון פונקציות לשימוש.
על מנת להגדיר פונטים ישנם פרמטרים ראשוניים שנצטך לחת בחשבון והם:

Height	The top to bottom size of the tallest char in the font
Baseline	The line that the bottom char aligned
Ascent	The distance from the baseline to the top of the char
Descent	The distance from the baseline to the bottom of the char
Leading	The distance between the bottom of one line of text and the top of the next

עד עתה השתמשנו בפונקציה drawstring() אשר יצרה לנו את המערך התווי שאנו הגדרנו על ידי המיקום. אולם המיקום שפונקציה זו נותנתלנו הנו בקצה שמאל למעלה.
ולכן לצורך פתרון הנושא על מנת שנוכל למקם הן טקסט והן גרפיקה בתוך ה applet נשתמש במחלקת FontMatrics.

שימוש במחלקה זו עוזר לנו לקבוע את המרחק בין שורות טקסט מסויימות תוך כמובן קביעת אורכו של string מסוים.
בכל פעם שנרצה להכניס שורה חדשה אזי כי נצטרך להגדיר ציר Y חדש לצורך קביעת הגובה וציר X לצורך מתן אורכה של ההודעה.
על מנת לקבוע את המרחק בין השורות נוכל להשתמש בערך המוחזר על ידי פונקציה getLeading().
על מנת לקבוע את גובה הפונט נוכל להוסיף את הערך המוחזר לנו על ידי פונקציה getAscent() לערך המוחזר על ידי פונקציה getDescent().
על מנת לקבוע את גובה הפונט נשתמש בפונקציה getHeight().
על מנת לדעת היכן ערכי השורה האחרונה בכדי שנוכל לכתוב את השורה הבאה נשתמש בפונקציה stringWidth().

דוגמא

בדוגמא זו נראה כיצד אנו מייצרים מספר שורות ב applet תוך שינוי קואורדינטות בעת הצורך.

```
//Multiple lines

import java.awt.*;
import java.applet.*;

/*
 <applet code = "MultiLine" width=300 height=100>
 </applet>
 */

public class MultiLine extends Applet
{
      int curX=0, curY=0;

      public void init()
      {
            Font f = new Font("Helvetica", Font.PLAIN, 12);
            setFont(f);
      }
```

```
    public void paint(Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();

        nextLine("This is on line 1", g);
        nextLine("This is line 2", g);
        sameLine("This is on the same line", g);
        sameLine("This is too", g);
        nextLine("This is on line 3", g);
    }

    //function to handle next Line
    void nextLine(String s, Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();

        curY += fm.getHeight();
        curX = 0;

        g.drawString(s, curX, curY);
        curX = fm.stringWidth(s);
    }

    //    dispaly on the same line
    void sameLine(String s, Graphics g)
    {
        FontMetrics fm = g.getFontMetrics();

        g.drawString(s, curX, curY);
        curX += fm.stringWidth(s);
    }
}
```

16.5. מירכוז טקסט

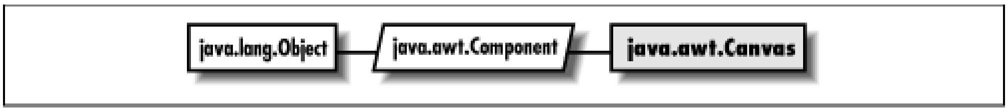
ניתן למרכז את הטקסט הקיים בחלון תוך שימוש בפונקציות שלמדנו ובחנו.

דוגמא

```
//Center Text

import java.awt.*;
import java.applet.*;

/*
 <applet code = "CenterText" width=300 height=100>
 </applet>
 */

public class CenterText extends Applet
{
	final Font f = new Font("Helvetica", Font.BOLD, 12);

	////

	public void paint(Graphics g)
	{
		Dimension d = this.size();

		g.setColor(Color.white);
		g.fillRect(0, 0, d.width, d.height);
		g.setColor(Color.black);
		g.setFont(f);
		drawCenteredString("This is Center", d.width,
d.height,g);
		g.drawRect(0, 0, d.width-1, d.height-1);
	}

	public void drawCenteredString (String s, int w, int h,
Graphics g)
	{
		FontMetrics fm = g.getFontMetrics();
		int x = (w - fm.stringWidth(s)) /2;
		int y = (fm.getAscent() + (h - (fm.getAscent()) +
fm.getDescent()))/2;

		g.drawString(s, x, y);

	}
}
```

16.6. חלון Canvas

יצירת חלון ריק כבר הוגדר תוך שימוש ב Canvas. ה Canvs הנו תת מחלקה של מחלקת Component אשר הנה תת מחלקה של מחלקת Object כמתואר בדיאגרמה להלן:



canvas הנו רכיב המייצר לנו איזור לציור ובד"כ אנו נשתמש בו כבסיס לרכיבים חדשים.
הגדרת המחלקה מתבצעת באופן הבא:

```
public class java.awt.Canvas
    extends java.awt.Component {

  // Constructors
  public Canvas();

  // Instance Methods
  public void addNotify();
  public void paint (Graphics g);
}
```

הקונסטרקטורים הקיימים הנם:

public Canvas()

כאשר:
קונסטרקטור זה יוצר לנו אובייקט מסוג Canvas.

הפונקציות לשימוש ב Canvas הנם:

public void addNotify()

כאשר:
פונקציה זו משכתבת את פונקצית Component.addNotify()

פונקציה נוספת הנה:

public void paint (Graphics g)

כאשר:
פונקציה ריקה האמורה להיות משוכתבת על מנת לאפשר לנו לצייר רכיב גרפי מסוים לדוגמא ציור של גרף מתמטי.

דוגמא

בדוגמא זו נראה שימוש במחלקת Canvas. הדוגמא משכתבת שוב את פונקצית paint() על מנת לצייר את שם ה applet ותקציר שירשם בתוכו. בנוסף אנו יוצרים פונטים וצבעים שאנו רוצים להשתמש בהם.

```
//Use of IntroCanvas subclass that ovverides paint()
//to draw the name of the applet and a brief

import java.awt;*.

 class IntroCanvas extends Canvas {
          private Color pink = new Color (255, 200, 200);
          private Color blue = new Color (150, 200, 255);
          private Color yellow = new Color (250, 220, 100);

private int w, h;

private int edge = 16;
private static final String title = "My Canvas Applet;"
private static final String name = "This is an Example of Canvas Use;"
private static final String context = "This will help you with your assignment;"
private Font namefont, titlefont, contextfont;

IntroCanvas} ()
setBackground(green;(
namefont = new Font("Times", Font.BOLD, 52);
titlefont = new Font("Times", Font.BOLD, 20);
contextfont = new Font ("Times", Font.PLAIN, 12);
}
}
```

17. שימוש בשליטה, ניהול תוצאה ותפריטים תוך שימוש ב AWT ובמודול Swing

GUI הנו חלק קריטי ממערכות תוכנה המשלבות אינטרקציה עם המשתמש.
בחלק זה נמשיך לבחון את תכונת הריכוזות הקיימת ב AWT תוך שימוש במודול ה Swing. כמו כן נבחן כיצד אנו מבצעים שימוש בכפתורים כגון כפתורי לחיצה או בחירה, תפריטים ו dialog box, וכו'.

17.1. יסודות שליטה ברכיבים

להלן רשימת הרכיבים הקיימים ב Swing כולל תיאור קצר:

Component	Description
JLable	איזור בו אנו מציגים טקסט אשר אי אפשר לערוך
JTextField	איזור בו המשתמש מכניס קלט מלוח המקשים. האיזור גם יכול להציג אינפורצמיה
JButton	איזור בו נוצר מאורע בזמן לחיצה
JCheckBox	רכיב GUI אשר אפשרי לבצע בו בחירה של אמת/שקר
JComboBox	רשימה "נופלת" הנותנת למשתמש אופציה לבחירת אחת מהאפשרויות.
JList	איזור בו רשימת ערכים מוצגת כאשר המשתמש יכול לבצע בחירה מתוך הרשימה.
JPanel	הנו container אשר רכיבים יכולים להיות ממוקמים.

17.2. הוספה/הסרה של שליטה

על מנת לכלול שליטה בחלון אנו חייבים להוסיף אותו לחלון. על מנת לבצע זאת בשלב ראשון אנו יוצרים instance של השליטה שאנו רוצים להוסיף ואז נוסיף אותו לחלון תוך שימוש בפונקצית add() אשר מוגדרת על ידי מחלקת Container.
לפונקציה זו קיימים שלושה סוגים:

Component add(Component compObject)

כאשר:
CompObject הנו ה instace של השליטה שאנו רוצים להוסיף. התייחסות ל compObject מוחזר לנו כערך.
לאחר שהשליטה התווספההיא באופן אוטומטי תהיה ויזואלית בכל פעם שחלון ה"אב" פועל.

לפעמים נרצה להוריד את השליטה בחלון מסוים. ניתן לבצע זאת על ידי שימוש בפונקצית remove() אשר מוגדרת במחקת Container. המבנה הכלי הנו:

void remove(Component Object)

כאשר:
Object מתייחס לאובייקט שאנו רוצים להרויד מהשליטה שלנו.

בנוסף על מנת להוריד את כל השליטה נוכל להשתמש בפונקצית removeAll().

17.3 תגובה לשליטה

מלבד labels שהנם מוגדרים כפקדים פסיביים כל שאר המאורעות נכנסים לעבודה כאשר המשתמש מבצע את פעולתו. לדוגמא כאשר המשתמש יחלץ על כפתור בחלון אזי כי המאורע יזהה את לחיצת הכפתור וייתן מענה למשתמש.
על מנת לטפל במאורע שכבר הוגדר אנו נצטרך לשכתב את המאורע על ידי שימוש בפונקצית action().
פונקציה זו מוגדרת על ידי מחלקת Component. המבנה הכללי הנו:

boolean action(Event evtObject, Object arg)

כאשר:
EvtObject מתאר את המאורע.
Args התייחסות למאורע.

מאחר שפונקצית action() נקראת לאחר שהמאורע כבר ביצע generated אנו חייבים לשכתב את פונקצית action() בתכנית שלנו. ומכאן הפונקציה המשוכתבת שלנו חייבת להחזיק ערך true באם היא מטפלת במאורע, אחרת היא תחזיר ערך false.

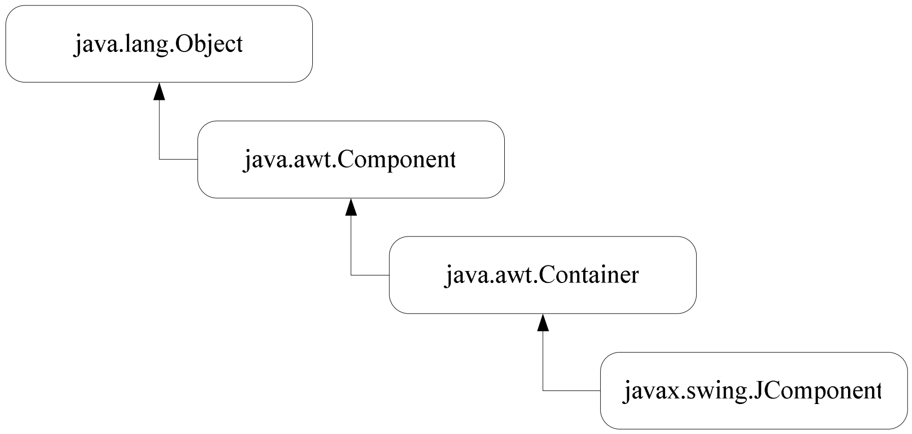
להלן מבנה השלד של שכתוב פונקצית action()

```
public boolean action (Event evtObject, Object arg)
{
if(////)
{
//handle the event
return true;
}
return false;
}
```

17.4 היכרות ל Swing

המחלקות אשר משמשות אותנו ליצירת GUI (תוארו לעיל) הנם חלק מרכיבי Swing GUI אשר נמצאים בחבילת javax.swing.
רכיבים אלו הנם החדשים ביותר הקימים ב Java כיום. רכיבי Swing נכתבים ומיושמים בשפת Java.
הרכיבים המקוריים שראינו עד עתה קרי AWT, הנם נמצאים בחבילת java.awt והנם מותאמים ליכולות הפלטפורמה הלוקאלית של המשתמש ולפיכך תוכניות Java הרצות בפלטפורמות שונות מיוצגים שונה עד לרמה של הצגת GUI שונה.
רכיבי ה Swing מאפשרים לנו להגדיר מצב של "ראה והרגש" (look & feel) לכל פלטפורמה כך שהמבנה הנו אחיד. בנוסף קיימת האפשרות לבצע שינויים של GUI בזמן ריצת התכנית.
רכיבי Swing בד"כ מיוחסים כקלים במשקלם, קרי הם נכתבים אך ורק ב Java כך שאינם צריכים לבצע שימוש בפלטפורמת המשתמש. לעומת זאת רכיבי AWT נשענים בעיקר על פלטפורמת הרצת התכנית כך שמטבעם הנם "כבדים" יותר בזמן ריצה. לכל רכיב שכזה ישנו peer מחבילת java.awt.peer אשר הנו אחראי לתגובה מהקוד לפלטפורמת השימוש. אך עם כל זאת עדיין קיימים רכיבי Swing אשר מיוחסים כ"כבדים" בעיקר תת המחלקה JFrame.

התרשים הבא מתאר את קשר ההורשה ההיררכית של מחלקות המגדירות מאפיינים והתנהגות אשר הנם אופיינים לרכיבי ה Swing השכיחים.



17.5. מאורעות מבט מקרוב

כבר ראינו כיצד אנו משתמשים במאורעות הקיימים במערכת שהנם נגזרים ממחלקת Event. בחלק זה נסתכל על מאורעות אלו שוב ונבחן את האינטגרציה בין המשתמש לבין המסך (בתוכניות שלנו).
מחלקת Event מגדירה שדות ומספרים קבועים על סמL ASCII קרי Constant ID.
בנוסף Event מגדירה גם 4 "מסות" Mask בביט:

Mask	Meaning
ALT_MASK	ALT is being pressed
CTRL_MASK	CRTL is being pressed
SHIFT_MASK	SHIFT is being pressed
META_MASK	META is being pressed

כאשר אנו מטפלים במאורעות הקשורים לשליטה בחלון, אנו חייבים לגשת לכלל השדות המוגדרים על ידי מחלקת Event. ברגע שנבצע זאת השליטה על התכנית תהיה קלה יותר. שתי השדות החשובים ביותר הנם target אשר מכיל התייחסות לאובייקט אשר מבצע את המאורע ושדה id אשר מכיל את הקבוע אשר מזהה את המאורע.

17.6. שימוש ב Labels

השליטה הקלה ביותר לשימוש הנה label.
Label הנו אובייקט מסוג מחלקת JLabel והוא מכיל string שיכול להיות מוצג על גבי החלון. כפי שהזכרנו labels הנם שלטים פסיביים שאינם תומכים בשום אינטראקציה עם המשתמש.
מבנה הקונסטרקטורים הנו:

```
JLabel()

JLable(String str)

JLable (String str, int how)
```

כאשר:
הקונסטרקטור הראשון יותר label ריק.
הקונסטרקטור השני יוצר label עם string המוגדר על ידי str. וברירת המחדל לכתיבה הנה שמאלה למעלה.
הקונסטרקטור השלישי יוצר label המכיל string המוגדר על ידי str תוך שימוש ביישור המוגדר על ידי how. ערך ה how יכול להיות אחד משלושת הקבועים הבאים Label.LEFT, Label.RIGHT, Lable.CENTER

נוכל כמובן לשנות את הטקסט המופיע על ה label תוך שימוש בפונקצית ()setText. וכמובן נוכל לראות את תכולת ה lable תוך שימוש בפונקצית ()getText.

מבנה הפונקציות הנו:

```
void setText(String str)

String getText()
```

נוכל לבצע יישור בתוך ה label על ידי שימוש בפונקצית ()setAlignment. על מנת לראות את מצב היישור הנוכחי נשתמש בפונקצית ()getAlignment.
מבנה הפונקציות הנו:

```
void setAlignment(int how)

int getAlignment
```

דוגמא

```
// Demonstrating the JLabel class.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LabelTest extends JFrame {
   private JLabel label1, label2;

   public LabelTest()
   {
      super( "Testing JLabel" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      // JLabel constructor with a string argument
      label1 = new JLabel( "Label with text" );
      label1.setToolTipText( "This is label1" );
      c.add( label1 );

     // JLabel constructor no arguments
      label2 = new JLabel();
      label2.setText( "Test Lable at bottom" );
      //label3.setIcon( bug );
      label2.setHorizontalTextPosition(
         SwingConstants.CENTER );
      label2.setVerticalTextPosition(
         SwingConstants.BOTTOM );
      label2.setToolTipText( "This is label3" );
      c.add( label2 );

      setSize( 156, 120 );
      show();
   }

   public static void main( String args[] )
   {
      LabelTest app = new LabelTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.7. שימוש בכפתורים

השימוש הנפוץ ביותר הנו כפתורי לחיצה. כפתור לחיצה הנו רכיב המכיל label והמבצע יישום כאשר אנו לוחצים עליו.
כפתורי לחיצה הנם תת מחלקה של Abstract Button שהנו חלק מחבילת javax.swing. פקודת פעולה על כפתור יוצרת ActionEvent כאשר פקודות אלו נוצרות בעזרת מחלקת JButton אשר יורשת ממחלקת AbstractButton.
מבנה הקונסטרקטורים הנו:

JButton()

JButton(String str)

כאשר:
הקונסטרקטור הראשון הנו כפתור ריק ללא תוית. (label)
הקונסטרקטור השני יוצר לנו כפתור עם תוית.

לאחר שהכפתור נוצר נוכל לתת לו תוית על ידי שימוש בפונקצית setLabel(). נוכל כמובן לראות איזו תוית קיימת כל הכפתור תוך שימוש בפונקצית getLabel().
מבנה הפונקציות הנו:

void setLable(String str)

String getLabel()

בכל פעם שהמשתמש ילחץ על הכפתור פונקצית action() נקראת. שדה ה target של המאורע מכיל התייחסות לכפתור המבצע את הפעולה.
האובייקט עם הפרמטרים מכיל התייחסות ל string אשר יהיה על הכפתור (=תוית).

דוגמא

בדוגמא זו נראה כיצד המשתמש לוחץ על כפתור מסוים. כאשר המשתמש יבחר כפתור וילחץ אזי כי תופיע הודעה שהמאורע אכן התבצע.

```
// Creating JButtons.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest extends JFrame {
   private JButton plainButton, fancyButton;

   public ButtonTest()
   {
      super( "Testing Buttons" );
      Container c = getContentPane();
      c.setLayout( new FlowLayout() );
      // create buttons
      plainButton = new JButton( "Simple Button" );
      c.add( plainButton );

     Icon bug1 = new ImageIcon( "bug1.gif" );
      Icon bug2 = new ImageIcon( "bug2.gif" );
      fancyButton = new JButton( "Fancy Button", bug1 );
      fancyButton.setRolloverIcon( bug2 );
      c.add( fancyButton );

      // create an instance of inner class ButtonHandler
      // to use for button event handling
      ButtonHandler handler = new ButtonHandler();
      fancyButton.addActionListener( handler );
      plainButton.addActionListener( handler );

      setSize( 275, 100 );
      show();
   }
   public static void main( String args[] )
   {
      ButtonTest app = new ButtonTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
 // inner class for button event handling
   private class ButtonHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         JOptionPane.showMessageDialog( null,
            "You pressed: " + e.getActionCommand() );
      }
   }
}
```

17.8. שימוש בכפתורי בחירה

רכיב ה GUI של ה Swing כולל בתוכו שלושה סוגים של כפתורי מצב:

1. JToggleButton
2. JCheckBox
3. JRadioButton

כפתור בחירה הנו סוג של כפתור המשמש אותנו על מנת לתת אפשרות בחירה למשתמש בצורה של on/off.
תכונה זו מכילה ציור של קופסא קטנה היכולה לאתחל סימון כלשהו.
לכל קופסא שכזו קיימת תוית label המתארת את הקופסא. נוכל לשנות את מצבה של הקופסא על ידי לחיצה לבחירה או לאי בחירה.
כפתורי בחירה יכולים להיות עצמאיים או כחלק מקבוצת כפתורים.

מבנה הקונסטרקטורים הנו:

JCheckbox()

JCheckbox(String str)

JCheckbox(String str, CheckboxGroup cbGroup, Boolean on)

כאשר:
הקונסטרקטור הראשון יוצר כפתור בחירה ריק לחלוטין כאשר מצב הסימון הנו "לא מסומן"
הקונסטרקטור השני יוצר כפתור בחירה עם תוית label המוגדרת על ידי str. ושוב מצב הכפתור הנו ב"לא מסומן".
הקונסטרקטור השלישי יוצר כפתור בחירה אשר התוית שלו מוגדרת על ידי str וקבוצתו מוגדרת על ידי cbGroup. במידה וכפתור הבחירה הזה איננו חלק מקבוצת כפתורים אזי כי ערכו של cbGroup יהיה 0.
הערך on נותן לנו את הערך להפעלת הכפתור קרי ערך true אומר כי כפתור הבחירה ירוץ עם סימון בפנים אחרת כפתור הבחירה יהיה ריק.

על מנת לקבל את מצב כפתור הבחירה הנוכחי נשתמש בפונקצית ()getState. על מנת לשנות את מצבו של הכפתור נוכל להשתמש בפונקצית ()setState.
בנוסף נוכל לדעת את ערך התוית של הכפתור הנוכחי על ידי שימוש בפונקצית ()getLabel. על מנת לשנות את התוית נוכל לבצע זאת על ידי שימוש בפונקצית ()setLabel.
מבנה הפונקציות הנו:

boolean getState()

void setState(Boolean on)

String getLabel()

void setLabel(String str)

אופי הפעולה:
בכל פעם שהמשתמש לוחץ על כפתור בחירה מופעלת פונקצית ()action. שדה ה target של המאורע מכיל את התייחסות של מצב כפתור הבחירה. ומכאן מצב זה יהיה אמת באם כפתור הבחירה נבחר אחרת מצבו יהיה שקר.
בד"כ כפתורי בחירה אינם מבצעים פעולה מיידית ובד"כ אנו משתמשים בהם בטפסים למינהם המצויים באינטרנט וזאת על מנת לאפשר למשתמש למלא את כלל המידע שאנו רוצים לקבל.

דוגמא

בדוגמא זו נראה כיצד כפתורי בחירה עובדים. נשים לב כי בכל פעם שאנו משנים את מצב הכפתור הסטטוס משתנה.

```
// Creating Checkbox buttons.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBoxTest extends JFrame {
   private JTextField t;
   private JCheckBox bold, italic;

   public CheckBoxTest()
   {
      super( "JCheckBox Test" );
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      t = new JTextField( "See How the style change", 20 );
      t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
      c.add( t );
      // create checkbox objects
      bold = new JCheckBox( "Bold" );
      c.add( bold );
      italic = new JCheckBox( "Italic" );
      c.add( italic );

CheckBoxHandler handler = new CheckBoxHandler();
      bold.addItemListener( handler );
      italic.addItemListener( handler );
      setSize( 355, 150 );
      show();
   }

   public static void main( String args[] )
   {
      CheckBoxTest app = new CheckBoxTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }

   private class CheckBoxHandler implements ItemListener {
      private int valBold = Font.PLAIN;
      private int valItalic = Font.PLAIN;

      public void itemStateChanged( ItemEvent e )
      {
         if ( e.getSource() == bold )
            if ( e.getStateChange() == ItemEvent.SELECTED )
               valBold = Font.BOLD;
            else
               valBold = Font.PLAIN;
```

```
         if ( e.getSource() == italic )
            if ( e.getStateChange() == ItemEvent.SELECTED )
               valItalic = Font.ITALIC;
            else
               valItalic = Font.PLAIN;

         t.setFont(
            new Font( "TimesRoman", valBold + valItalic, 18 ) );
         t.repaint();
      }
   }
}
```

17.9. כפתורי Radio

במקרים בהם נרצה להציג מידע מסוים אך נרצה כי למשתמש תהיה האפשרות לבחור רק אופציה אחת מתוך קבוצה מסוימת אזי כי נשתמש בטכניקת הקבוצות. ומכאן מכלל המידע שיוצג למשתמש רק אופציה אחת תוכל להבחר בכל פעם מחדש.
כפתורים אלו נקראים בשפה המקצועית radio buttons.
על מנת לייצור קבוצה שכזו בשלב ראשון עלינו להגדיר את הקבוצה ואז לשייך אליה את הכפתורים שאנו רוצים. קבוצה של כפתורי אלו הנה אובייקט של CheckBoxGroup. רק ברירת המחדל של הקונסטרקטור מוגדרת אשר בעצם יוצר לנו קבוצה ריקה.
נוכל לקבוע איזה כפתור יכלל בקבוצה על ידי שימוש בפונקצית ()getCurrent וכמובן שנוכל לבצע שינויים על ידי שימוש בפונקצית ()setCurrent.
מבנה הפונקציות הנו:

```
JCheckbox getCurrent()

void setCurrent(Checkbox which)
```

כאשר:
Which – מתייחס לכפתור שאנו רוצים לבחור.

דוגמא

```
// Creating radio buttons using ButtonGroup and JRadioButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButtonTest extends JFrame {
   private JTextField t;
   private Font plainFont, boldFont,
                italicFont, boldItalicFont;
   private JRadioButton plain, bold, italic, boldItalic;
   private ButtonGroup radioGroup;

   public RadioButtonTest()
   {
      super( "RadioButton Test" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      t = new JTextField( "See how the font style change", 25 );
      c.add( t );

      // Create radio buttons
      plain = new JRadioButton( "Plain", true );
      c.add( plain );
      bold = new JRadioButton( "Bold", false);
      c.add( bold );
      italic = new JRadioButton( "Italic", false );
      c.add( italic );
      boldItalic = new JRadioButton( "Bold/Italic", false );
      c.add( boldItalic );

      // register events
      RadioButtonHandler handler = new RadioButtonHandler();
      plain.addItemListener( handler );
      bold.addItemListener( handler );
      italic.addItemListener( handler );
      boldItalic.addItemListener( handler );

      // create logical relationship between JRadioButtons
      radioGroup = new ButtonGroup();
      radioGroup.add( plain );
      radioGroup.add( bold );
      radioGroup.add( italic );
      radioGroup.add( boldItalic );

      plainFont = new Font( "TimesRoman", Font.PLAIN, 14 );
      boldFont = new Font( "TimesRoman", Font.BOLD, 14 );
      italicFont = new Font( "TimesRoman", Font.ITALIC, 14 );
      boldItalicFont =
         new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
      t.setFont( plainFont );

      setSize( 300, 100 );
      show();
   }

   public static void main( String args[] )
   {
```

```
      RadioButtonTest app = new RadioButtonTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }

   private class RadioButtonHandler implements ItemListener {
      public void itemStateChanged( ItemEvent e )
      {
         if ( e.getSource() == plain )
            t.setFont( plainFont );
         else if ( e.getSource() == bold )
            t.setFont( boldFont );
         else if ( e.getSource() == italic )
            t.setFont( italicFont );
         else if ( e.getSource() == boldItalic )
            t.setFont( boldItalicFont );

         t.repaint();
      }
   }
}
```

‎.17.10 רשימה Drop Down

מחלקת Choice משמשת אותנו על מנת לייצור רשימה מסוג drop down. לפיכך ניתן לומר כי סוג זה של רשימה הנו סוג של תפריט.
מחלקת Choice מגדירה רק את ברירת המחדל של הקונסטרקטור אשר יוצר לנו רשימה ריקה.
על מנת להוסיף בחירה לרשימה אנו נשתמש בפונקצית ()addItem. המבנה הכללי הנו:

void addItem(String name)

כאשר:
Name – הנו התפריט שהוספנו.

על מנת לקבוע איזה תפריט נבחר נוכל להשתמש בפונקצית ()getSelectedItem או בפונקצית ()getSelectedIndex.
מבנה הפונקציות הנו:

String getSelectedItem()

int getSelectedIndex()

בנוסף נוכל לשנות את התפריטים הנוכחיים על ידי שימוש בפונקצית ()select עם אחד מהשניים או עם ערך מספרי שווה 0 או עם string שיתאים לשם ברשימה . (במידה ואיננו מתאים תהיה טעות).
מבנה הפונקציות הנו:

int countItem()

void select(int index)

void select(String name)

כאשר נתון לנו ה index אנו יכולים לדעת מהו השם הקשור לאותו אינדקס תוך שימוש בפונקצית ()getItem. המבנה הכללי הנו:

String getItem()

אופי הפעולה:
בכל פעם שהמשתמש בוחר אופציה מופעלת פונקצית ()action. שדה ה target של המאורע מכיל את התייחסות של מצב תפריט הבחירה.
בד"כ תפריטי בחירה אינם מבצעים פעולה מיידית ואנו משתמשים בהם על מנת לתת למשתמש מגוון רחב של אופציות כאלו ואחרות ואו לקבל ממנו מידע שאנו רוצים.

דוגמא

```
 //Demonstaration Selected Menu

import java.awt.*;
import java.applet.*;

/*
 <applet code = "SelMenu" width=300 height=180>
 </applet>
*/

public class SelMenu extends Applet
{
      String msg = "";
      Choice os, browser;

      public void init()
      {
            os = new Choice();
            browser = new Choice();

            //add lables
            os.addItem("Win Me");
            os.addItem("Win 2000");
            os.addItem("Win XP");

            browser.addItem("Netscape");
            browser.addItem("Microsoft");
            browser.addItem("AOL");
            browser.addItem("Apple");
            browser.addItem("Linux");

            add(os);
            add(browser);
      }
      //Repaint when status of check box change

      public boolean action(Event evtObject, Object arg)
      {
            if (evtObject.target instanceof Choice)
            {
                  repaint();
                  return true;
            }
            return false;
      }

      //Display current status of the check box

      public void paint(Graphics g)
      {
            msg = "Current OS: " ;
            msg += os.getSelectedItem();
            g.drawString(msg, 6, 120);

            msg = "Current Browser: ";
            msg += browser.getSelectedItem();
            g.drawString(msg, 6, 140);
      }
}
```

17.11. שימוש ברשימות

מחלקת List מאפשרת לנו לייצור רשימות קומפקטיות קטנות תוך מתן אופציה לגרירה. שלא כמו אובייקט Choice שראינו בשימושים הקודמים המאפשר לנו רק בחירה של אופציה אחת, אובייקט מסוג List מאפשר לנו לבנות רשימה שממנה נוכל לבחור מספר רב של אפשרויות.
מבנה הקונסטרקטורים הנו:

List()

List(int numRows, boolean multipleSelection)

כאשר:
הקונסטרקטור הראשון יוצר לנו רשימה שנוכל לבחור ממנה רק פריט אחד בכל זמן תון.
בקונסטרקטור השני ערך numRows הנו מספר השורות הויזואלי שאנו רוצים להכניס. במידה ומשתנה multipleSelection הנו true אזי כי המשתמש יכול לבחור שתי אופציות או יותר בזמן נתון. אחרת רק אופציה אחת תהיה אפשרית.

בנוסף, על מנת להוסיף בחירה בתפריט הרשימה נוכל להשתמש בפונקצית ()addItem. המבנה הכללי הנו:

void addItem(String name)

void addItem(String name, int index)

כאשר:
Name – הנו התפריט שהוספנו.
Index – מוסיף לנו את השורה החדשה המוגדרת באינדקס. האינדקס מתחיל ברך 0.

לרשימות המאפשרות בחירה בודדת נוכל לקבוע מהי ברירת המחדל על ידי שימוש בפונקצית ()getSelectedItem או בפונקצית ()getSelectedIndex.
מבנה הפונקציות הנו:

String getSelectedItem()

int getSelectedIndex()

לרשימות המאפשרות בחירה מרובה אנו חייבים להשתמש באחת מהפונקציות הבאות ()getSelectedItems או ב ()getSelectedIndexes.
להלן המבנה הכללי כיצד לייצור בחירה שכזו:

String[] getSelectedItems()

int [] getSelectedIndexes()

כאשר:
הפונקציה הראשונה מחזירה לנו מערך המכיל את שמות הפריטים הנבחרים.
הפונקציה השנייה מחזירה לנו מערך המכיל את האינדקסים הנבחרים.

על מנת לדעת מהו מספר הפריטים ברשימה נוכל להשתמש בפונקצית ()countItem. נוכל כמובן לשנות פריט כלשהו על ידי שימוש בפונקצית ()select המבוסס על ערך מספרי של 0.
מבנה הפונקציות הכללי הנו:

```
int countItem()

void select(int index)
```

כאשר נתון לנו ה index אנו יכולים לדעת מהו השם הקשור לאותו אינדקס תוך שימוש בפונקצית ()getItem. המבנה הכללי הנו:

```
String getItem()
```

אופי הפעולה:
בכל פעם ש List נקראת פונקצית ()action נקראת. שדה ה target של המאורע מכיל התייחסות לרשימות הקשורות לפעולה. האובייקט עם הפרמטרים מכיל את השמות החדשים לבחירה.
בנוסף, בכל פעם שפריט נבחר עם לחיצה אחת מאורע בשם LIST_SELECTED קורה.
על מנת לטפל במאורעות אלו אנו חייבים לשכתב את פונקצית ()handleEvent.

דוגמא

```
//Demonstaration Lists

import java.awt.*;
import java.applet.*;

/*
 <applet code = "ListTest" width=300 height=180>
 </applet>
*/

public class ListTest extends Applet
{
	String msg = "";
	List os, browser;

	public void init()
	{
		os = new List(4, true);
		browser = new List(4, false);

		//add lables
		os.addItem("Win Me");
		os.addItem("Win 2000");
		os.addItem("Win XP");

		browser.addItem("Netscape");
		browser.addItem("Microsoft");
		browser.addItem("AOL");
		browser.addItem("Apple");
		browser.addItem("Linux");

		add(os);
		add(browser);
	}
	//Repaint when status of check box change
	public boolean action(Event evtObject, Object arg)
	{
		if (evtObject.target instanceof List)
		{
			repaint();
			return true;
		}
		return false;
	}
	//Display current status of the check box
	public void paint(Graphics g)
	{
		int idx[];

		msg = "Current OS: " ;
		idx = os.getSelectedIndexes();
		for(int i=0; i<idx.length; i++)
			msg += os.getItem(idx[i]) + " ";
		g.drawString(msg, 6, 120);

		msg = "Current Browser: ";
		msg += browser.getSelectedItem();
		g.drawString(msg, 6, 140);
	}
}
```

17.12. שימוש בגלגלת

גלגלת משמשת אותנו על מנת לבצע בחירה מתוך רשימה מתמשכת המכילה ערכים מינימליים או מקסימאליים. גלגלות יכולים להופיע בצורה אנכית או אופקית.
גלגלת למעשה בנויה ממספר חלקים אינדיווידואליים. בכל סוף של גלגל ישנם חיצים המצביעים על כיוון הגלגלת והעוזרים למשתמש לקבוע היכן הוא רוצה לראות את הטקסט קרי מעלה או מטה.
הערכים העכשווים של הגלגלת נמדדים על ידי ערכים מינימליים או מאקסימליים הנתונים בתוך ה silder box לגלגלת.
הגלגלת יכולה להיות מוזזת על ידי המשתמש לקואורדינאטות חדשות. ורק לאחר מכן הגלגלת תקבל את ערכיה החדשים.
גלגלות מרוכזות בתוך מחלקת Scrollbar והמבנה הכללי הנו:

```
Scrollbar()

Scrollbar(int style)

Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

כאשר:

- הקונסטרקטור הראשון יוצר גלגלת אנכית
- הקונסטרקטור השני והשלישי יוצרים מאפשרים לנו לייצור את האוריינטציה של הגלגלת.
  במידה ו style הנו בעל ערך Scrollbar.VERTICAL אזי כי גלגלת אנכית תיווצר. וכמובן שאם style הנו בעל ערך של Scrollbar.HORIZONTAL אזי כי תיווצר לנו גלגלת אופקית.
  בקונסטרקטור השלישי הערך ההתחלתי המאותחל מועבר דרך initialValue. מספר היחידות המיוצגות מועבר על ידי thumbSize. ערכי המינימום והמקסימום מוגדרים ב int min/max.

במידה ואנו יוצרים גלגלת המשתמשת באחד שני הקונסטרקטורים הראשונים אזי כי נצטרך לתת את הפרמטרים שלנו תוך שימוש בפונקצית setValues() לפני השימוש. המבנה הכללי הנו:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

לפרמטרים הללו ישנם את אותם ערכים שיש לקונסטרקטור השלישי.

על מנת לקבל את ערכי הגלגלת נוכל להשתמש בפונקצית getValues() . על מנת לשנות ערך נוכחי אפשר להשתמש בפונקצית setValue(). מבנה הפונקציות הנו:

```
int getValue()

void setValues(int newValue)
```

כאשר:
NewValue – מגדיר את הערך החדש של הגלגלת.

הגלגלת הנמצאת בתוך ה scroll תהיה מושפעת מהערך החדש.

בנוסף נוכל לקבל את ערכי min/max על ידי שימוש בפונקציות getMinimum() ו-getMaximum(). המבנה הכללי הנו:

```
getMinimum()

getMaximum()
```

ברירת המחדל של הגלגלת בכל מאורע של up/down הנה שורה אחת בלבד. כמובן שנוכל לשנות זאת על ידי שימוש בפונקצית void setLineIncrement(int newIncr).
ברירת המחדל של דף מעלה/דף מטה הנו 10 ונוכל לשנות זאת על ידי שימוש בפונקצית setPageIncrement().

void setLineIncrement(int newIncr)

setPageIncrement()

אופי הפעולה:
מאורעות scroll bar אינם מועברים לפונקצית action(). מעבר לכך ל מנת שיבצעו את פעולותיהן הן חייבות לשכתב את handleEvent().
בכל פעם שאנו ניגשים ל scroll bar שדה ה target של המאורע מועבר, פונקצית handleEvent() תכיל התייחסות ל scroll bar שיוצר את המאורע.
שדה ה id יכיל את הערך המתאר את המאורע.

דוגמא

```
//Demonstaration Scroll bar

import java.awt.*;
import java.applet.*;

/*
 <applet code = "ScrollTest" width=300 height=180>
 </applet>
*/

public class ScrollTest extends Applet
{
	String msg = "";
	Scrollbar verticalScroll, horizontalScroll;

	public void init()
	{
		int width = Integer.parseInt(getParameter("width"));
		int height = Integer.parseInt(getParameter("height"));

		verticalScroll = new Scrollbar(Scrollbar.VERTICAL, 0, 1,
0, height);

		horizontalScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0,
1, 0, width);

		add(verticalScroll);
		add(horizontalScroll);
	}

	//Repaint when status of check box change
	public boolean handleEvent(Event evtObject)
	{
		if (evtObject.target instanceof Scrollbar)
		{
			repaint();
			return true;
		}
		return super.handleEvent(evtObject);
	}
	//Update scroll bars to reflect mouse
	public boolean mouseDrag(Event evtObject, int x, int y)
	{
		verticalScroll.setValue(y);
		horizontalScroll.setValue(x);
		repaint();
		return true;
	}
	//Display current status of the check box
	public void paint(Graphics g)
	{
		msg = "Vertical: " +  verticalScroll.getValue();
		msg += ", Horizontal: " + horizontalScroll.getValue();
		g.drawString(msg, 6, 160);

		//show current mouse drag cordinates
		g.drawString("*",
horizontalScroll.getValue(),verticalScroll.getValue());
	}
}
```

17.13. שימוש בשדות טקסט

מחלקת JTextField ומחלקת JPasswordField מיישמות שורה בודדת להכנסת טקסט ובד"כ בשפה המקצועית נקראת edit control. מחלקה זו מאפשרת למשתמש להכניס string ולערוך את הטקסט על ידי שימוש בחיצים הקיימים בלוח המקשים, או בעכבר.
מחלקת JTextField הנה הרחבה של מחלקת JTextComponent.
מבנה הקונסטרקטורים הנו:

JTextField()

JTextField(int numChars)

JTextField(String str)

JTextField(String str, int numChars)

כאשר:

- הקונסטרקטור הראשון יוצר לנו ברירת מחדל של שורת טקסט.
- הקונסטרקטור השני יוצר לנו שדה טקסט אשר ניתן להגדיר לו תכולת תווים על ידי intChars.
- הקונסטרקטור השלישי מאתחל את שדה הטקסט עם ה string שהגדרנו ב str.
- הקונסטרקטור הרביעי מאתחל את שדה הטקסט ומגדיר את רוחבו.

מחלקת TextField מגדירה לנו מספר פונקציות העוזרות לנו לנצל את תכונות מופע הטקסט. על מנת לקבל את ה string הקיים בטקסט נוכל להשתמש בפונקצית ()getText.
על מנת לשנות את ערכו נוכל להשתמש בפונקצית ()setText
המבנה הכללי הנו:

String getText()

void setText(String str)

המשתמש יכול לבחור חלק מהטקסט (=מנה) בתוך הטקסט הנתון. בנוסף נוכל לבחור טקסט על ידי שימוש בפונקצית ()select. נוכל לדעת מהו הטקסט הנבחר על ידי שימוש בפונקצית ()getSelected.
מבנה הפונקציות הנו:

String getSelectedText()

void select(int StartIndex, int EndIndex)

בנוסף נוכל לשלוט על תכולת הטקסט שהמשתמש יכניס תוך שימוש בפונקצית ()setEditable. נוכל לקבוע את העריכה על ידי שימוש בפונקצית ()isEditable. מבנה הפונקציות הנו:

boolean isEditable()

void setEditable(boolean canEdit)

בפונקציה הראשונה מחזירה ערך true במידה והטקסט שונה אחרת היא תחזיר ערך false.
בפונקציה השנייה משתנה canEdit הנו בוליאני.

קיימים מצבים בהם נרצה שהמשתמש יכניס טקסט אשר לא יהיה מוצג על המסך. על מנת לבצע זאת נוכל להשתמש בפונקצית ()setEchoCharacter. פונקציה זו מגדירה תו בודד אשר מחלקת TextField יכולה להציג אשר תווים מוכנסים.
נוכל לבדוק את שדה הטקסט לראות האם הוא נמצא ב mode זהעל ידי שימוש בפונקצית ()echoCharIsSet. נוכל לקבל את ערך ה echo על ידי קריאה לפונקצית ()getEchoChar. מבנה הפונקציות הנו:

```
void setEchoCharacter(char ch)

boolean echoCharIsSet()

char getEchoChar()
```

אופי הפעולה:
מאחר ששדות טקסט מייצגות את אופי העריכה בעזרת פונקציות משלהן בד"כ בתכנית שלנו לא תהיה תגובה למאורע פרטי קרי לחיצת מקש כלשהו. אולם, נרצה להגיב כאשר המשתמש יחלץ Enter. כאשר פעולה זו קוראת, פונקצית ()action נקראת. שדה ה target של המאורע מכיל התייחסות לשדה הטקסט כאשר פעולת Enter מתבצעת.

דוגמא

```
// Demonstrating the JTextField class.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextFieldTest extends JFrame {
   private JTextField text1, text2, text3;
   private JPasswordField password;

   public TextFieldTest()
   {
      super( "Testing JTextField and JPasswordField" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      // construct textfield with default sizing
      text1 = new JTextField( 10 );
      c.add( text1 );

      // construct textfield with default text
      text2 = new JTextField( "Enter text here" );
      c.add( text2 );

      // construct textfield with default text and
      // 20 visible elements and no event handler
      text3 = new JTextField( "Uneditable text field", 20
);
      text3.setEditable( false );
      c.add( text3 );

      // construct textfield with default text
      password = new JPasswordField( "Hidden text" );
      c.add( password );

      TextFieldHandler handler = new TextFieldHandler();
      text1.addActionListener( handler );
      text2.addActionListener( handler );
      text3.addActionListener( handler );
      password.addActionListener( handler );

      setSize( 325, 100 );
      show();
   }

   public static void main( String args[] )
   {
      TextFieldTest app = new TextFieldTest();

      app.addWindowListener(
```

```
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }

   // inner class for event handling
   private class TextFieldHandler implements
ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         String s = "";

         if ( e.getSource() == text1 )
            s = "text1: " + e.getActionCommand();
         else if ( e.getSource() == text2 )
            s = "text2: " + e.getActionCommand();
         else if ( e.getSource() == text3 )
            s = "text3: " + e.getActionCommand();
         else if ( e.getSource() == password ) {
            JPasswordField pwd =
               (JPasswordField) e.getSource();
            s = "password: " +
                new String( pwd.getPassword() );
         }

         JOptionPane.showMessageDialog( null, s );
      }
   }
}
```

17.14. שימוש באיזור טקסט

קיימים מצבים בהם שורה אחת של טקסט איננה מספיקה לנו. על מנת לתת פתרון לבעיה זו, JTextArea נותנת לנו מקום להכנסת מספר שורות של טקסט, תכונה הנקראת TextArea. מחלקת JTextArea יורשת ממחלקת JTextComponent.
מבנה הפונקציות הנו:

JTextArea()

JTextArea (int numLines, int numChars)

JTextArea (String str)

JTextArea (String str, int numLines , int numChars)

כאשר:
NumLines – מגדיר את הגובה של תכולת השורות.
NumChars – מגדיר את רוחב תכולת השורות.

TextArea הנה תת מחלקה של TextComponent. לפיכך היא תומכת בפונקציות getText(), setText() getSelectedText() select(), isEditable(), setEditable() כלל הפונקציות הללו הוסברו בחלק הנ"ל.

בנוסף מוגדרות לנו פונקציות נוספות:

void appendText(String str)

void insertText(String str, int index)

void replaceText(String str, int startIndex, intEndIndex)

כאשר:
הפונקציה הראשונה תלויה ב string המוגדר על ידי str
פונקציה שנייה מכניסה את ה string המועבר ב str באינדקס המוגדר.
פונקציה שלישית מבצעת החלפת טקסט.

אופי הפעולה:
אזורי טקסט הנם בעלי שליטה עצמאית. אין באפשרות התכנית שלנו לנהל ניהול נוסף את השליטה. אזורי טקסט מנהלים אך ורק מאורעות של got-focus או lost-focus.

דוגמא

```
// Copying selected text from one text area to another.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextAreaDemo extends JFrame {
   private JTextArea t1, t2;
   private JButton copy;

   public TextAreaDemo()
   {
      super( "TextArea Demo" );

      Box b = Box.createHorizontalBox();

      String s = "This is a demo string to\n" +
                 "illustrate copying text\n" +
                 "from one TextArea to \n" +
                 "another TextArea using an\n"+
                 "external event\n";

      t1 = new JTextArea( s, 10, 15 );
      b.add( new JScrollPane( t1 ) );

      copy = new JButton( "Copy >>>" );
      copy.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               t2.setText( t1.getSelectedText() );
            }
         }
      );
      b.add( copy );

      t2 = new JTextArea( 10, 15 );
      t2.setEditable( false );
      b.add( new JScrollPane( t2 ) );

      Container c = getContentPane();
      c.add( b );    // Box placed in BorderLayout.CENTER
      setSize( 425, 200 );
      show();
   }

   public static void main( String args[] )
   {
      TextAreaDemo app = new TextAreaDemo();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.15. שימוש בקביעת תצורה

עד כה כלל הרכיבים שהוצגו על המסך נעשו בעזרת השימוש בברירת המחדל של המערכת קרי המערכת ניהלה לנו את אופן הצגת הנתונים.
כפי שכבר הזכרנו ניהול התצוגה מתבצע באופן אוטומטי על ידי שימוש באלגוריתמים.
לכל אובייקט מסוג Container קיים מערך ניהול שהנו בעצם instance של המחלקה המיישמת את הממשק. מחלקה זו הנה LayoutManaer.
ניתן לבצע שינויים למחלקה זו על ידי פונקצית setLayout() . כאשר איננו מבצעים קריאה לפונקציה זו אזי כי מבחינת המערכת השימוש יהיה בברירת המחדל.
מבנה הפונקציה הכללי הנו:

```
void setLayout(LayoutManager layoutObject)
```

כאשר:
LayoutObject – הנו התייחסות למערך שאנו רוצים לקבל. באם נרצה להתעלם מניהול המערך ולבצע זאת באופן ידני אזי כי נעביר ערך 0 . במידה ואנו עושים זאת אנו נצטרך לתת את התצורה והמיקום של כל רכיב באופן ידני.
על מנת לבצע זאת נוכל להשתמש בפונקצית reshape() שהנה חלק ממחלקת Component.
ובד"כ נרצה להשתמש במערך הנתון לנו על ידי AWT.

כל מערך שומר מעקב אחר רשימה של רכיבים המאוכלסים על פי שמות כך שנהיול המערך מקבל מידע בכל פעם שאנו מוסיפים או מורידים רכיב כלשהו ל Container.
בכל רגע שה Conainer צריך לקבל מדות חדשות ניהול המערך מבצע זאת דרך פונקציות minimumLayoutSize() וב – prferredLauoutSize().
כל רכיב מנוהל על ידי המערך מכיל את פונקצית prefferedSize() וב – minimumSize(). פונקציות אלו מחזירות את הערכים והמידות שיהיו מיוצגות על ידי לצורך הצגת כל רכיב בנפרד.
ובכל מצב ניתן לומר כי באם נרצה אנו לבצע שינויים כלשהם במערך התצוגה אזי כי נצטרך לשכתב פונקציות.
שפת Java מגדירה מספר מחלקות לצורך LayoutManager , הבא נבחן אותם.

17.15.1. מערך זרימה – FlowLayout

מערך זה הנו ברירת מחדל של אחד המערכים הקיימים. מערך זה מיישם סגנון תוצאה פשוט הדומה לסגנון של טקסט בדף word.
תוך שימוש במערך זה רכיבים יהיו מסודרים מהפינה השמאלית העליונה ועד לפינה הימנית התחתונה.
כאשר אין מקום בשורה אחת נפתחת שורה חדשה. ישנם מירווחים בין כל רכיב מעלה/מטה כמו גם ימינה/שמאלה.
להלן מבנה הקונסטרקטורים:

```
FlowLayout()

FlowLayout(int how)

FlowLayout(int how, int horz, int ver)
```

כאשר:

הקונסטרקטור הראשון יוצר לנו את ברירת המחדל אשר יוצר רכיב מרכזי ומשאיר מרווחים של 5 pixels בין כל רכיב.

הקונסטרקטור השני נותן לנו להגדיר כיצד כל שורה תיושר. הערכים הקיימים למשתנה how הנם: FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT בהתאמה.

הקונסטרקטור השלישי נותן לנו להגדיר את המרווח האנכי והאופקי בין כל רכיב.

דוגמא

```
// Demonstrating FlowLayout alignments.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FlowLayoutDemo extends JFrame {
   private JButton left, center, right;
   private Container c;
   private FlowLayout layout;

   public FlowLayoutDemo()
   {
      super( "FlowLayout Demo" );

      layout = new FlowLayout();

      c = getContentPane();
      c.setLayout( layout );

      left = new JButton( "Left" );
      left.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               layout.setAlignment( FlowLayout.LEFT );

               // re-align attached components
               layout.layoutContainer( c );
            }
         }
      );
      c.add( left );

      center = new JButton( "Center" );
      center.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               layout.setAlignment( FlowLayout.CENTER );

               // re-align attached components
               layout.layoutContainer( c );
            }
         }
      );
      c.add( center );

      right = new JButton( "Right" );
      right.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
```

```
            {
               layout.setAlignment( FlowLayout.RIGHT );

               // re-align attached components
               layout.layoutContainer( c );
            }
         }
      );
      c.add( right );

      setSize( 300, 75 );
      show();
   }

   public static void main( String args[] )
   {
      FlowLayoutDemo app = new FlowLayoutDemo();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.15.2. BorderLayout – מערך גבול

מחלקת BorderLayout מיישמת מספר סגנונות. ברירת המחדל שלו הנה החל מרמתו העליונה של החלון. לתכונה זו קיימים 4 מידות קבועות של רוחב הנמצאות בקצה, ואיזור אחד גדול הנמצא במרכז. כל אחד מאיזורים אלו מיוחס על פי שם קרי North, South, East, West המייצגים את כלל הצדדים. המילה Center מיצגת את המרכז.
להלן מבנה הקונסטרקטורים:

BorderLayout()

BorderLayout(int how, int horz, int vert)

כאשר:
הקונסטרקטור הראשון יוצר לנו את ברירת המחדל.
הקונסטרקטור השני מאפשר לנו להגדיר את הערכים האופקיים והרוחביים.

כאשר אנו מוסיפים רכיבים אנו נשתמש בפונקצית add(). המבנה הכלי הנו:

Component add(String name, Component compObject)

דוגמא

```
// Demonstrating BorderLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BorderLayoutDemo extends JFrame
                              implements ActionListener {
   private JButton b[];
   private String names[] =
      { "Hide North", "Hide South", "Hide East",
        "Hide West", "Hide Center" };
   private BorderLayout layout;

   public BorderLayoutDemo()
   {
      super( "BorderLayout Demo" );

      layout = new BorderLayout( 5, 5 );

      Container c = getContentPane();
      c.setLayout( layout );

      // instantiate button objects
      b = new JButton[ names.length ];

      for ( int i = 0; i < names.length; i++ ) {
         b[ i ] = new JButton( names[ i ] );
         b[ i ].addActionListener( this );
      }

      // order not important
      c.add( b[ 0 ], BorderLayout.NORTH );  // North position
      c.add( b[ 1 ], BorderLayout.SOUTH );  // South position
      c.add( b[ 2 ], BorderLayout.EAST );   // East position
```

```
      c.add( b[ 3 ], BorderLayout.WEST );   // West position
      c.add( b[ 4 ], BorderLayout.CENTER ); // Center position

      setSize( 300, 200 );
      show();
   }

   public void actionPerformed( ActionEvent e )
   {
      for ( int i = 0; i < b.length; i++ )
         if ( e.getSource() == b[ i ] )
            b[ i ].setVisible( false );
         else
            b[ i ].setVisible( true );

      // re-layout the content pane
      layout.layoutContainer( getContentPane() );
   }

   public static void main( String args[] )
   {
      BorderLayoutDemo app = new BorderLayoutDemo();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.15.3. מערך מרווח – Inset Layout

לפעמים נרצה להשאיר מרווח בין הרכיב שלנו לבין החלון. על מנת לבצע זאת אנו נשכתב את פונקציית insets() המוגדרת על ידי Container. פונקציה זו מחזירה ערך insets שהנו אובייקט המכיל את מידות מעלה, מטה, שמאל, ימין.
מבנה הקונסטרקטור הנו:

Insets insets()

כאשר אנו משכתבים את הפונקציה הזו אנו חייבים להחזיר ערכים שהם new insets.

דוגמא

```
//Demonstaration BorderLayout with Insets
import java.awt.*;
import java.applet.*;
import java.util.*;

/*
 <applet code = "BorderTest" width=400 height=200>
 </applet>
*/
public class BorderTest extends Applet
{
      public void init()
      {
            setBackground(Color.cyan);

            //set layout
            setLayout(new BorderLayout());

            add("North", new Button("This is on top"));
            add("South", new Label("The footer msg"));
            add("East", new Button("Right"));
            add("West", new Button("Left"));

            String msg = "The reasonable man adapts\n" +
                  "the real time world\n\n\n" +
                  "aditude tryin to mind the gap\n" +
                  "in technologies over different " +
                  "places through the world";

            add("Center", new TextArea(msg));
      }

      //add insets
      public Insets insets()
      {
            return new Insets(10, 10, 10, 10);
      }
}
```

17.15.4. מערך פאנל - Panel Layout

GUI מורכב דורש כי כל רכיב ימוקם בדיוק במקומו הנתון. לעיתים רכיבים אלו כוללים מספר רב של פאנלים כאשר כל אחד מאלו מסודר במערך אחר. פאנלים נוצרים בעזרת מחלקת JPanel שהנה תת מחלקה של JComponent. מחלקת JComponents יורשת ממחלקת java.awt.Container כך שכל JPanel הנו Container.

דוגמא

```java
// Using a JPanel to help lay out components.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PanelDemo extends JFrame {
   private JPanel buttonPanel;
   private JButton buttons[];

   public PanelDemo()
   {
      super( "Panel Demo" );

      Container c = getContentPane();
      buttonPanel = new JPanel();
      buttons = new JButton[ 5 ];

      buttonPanel.setLayout(
         new GridLayout( 1, buttons.length ) );

      for ( int i = 0; i < buttons.length; i++ ) {
         buttons[ i ] = new JButton( "Button " + (i + 1) );
         buttonPanel.add( buttons[ i ] );
      }

      c.add( buttonPanel, BorderLayout.SOUTH );

      setSize( 425, 150 );
      show();
   }

   public static void main( String args[] )
   {
      PanelDemo app = new PanelDemo();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.15.5. Grid Layout – מערך רשת

מערך רשתי מצייר לנו את הרכיב בדו מימד. כאשר אנו מאתחלים את GridLayout אנו מגדירים את מספר השורות ואת מספר העמודות.

מבנה הקונסטרקטורים הנו:

GridLayout(int numRows, int numColums)

GridLayout(int numRows, int numColums, int horz, int vert)

כאשר:

הקונסטרקטור הראשון נותן לנו לייצור אתמערך הרשת עם מידות למספר השורות ומספר העמודות.

הקונסטרקטור השני נותן לנו לייצור את המרווחים האופקיים והאנכיים בין קוי הרשת.

דוגמא

```
// Demonstrating GridLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GridLayoutDemo extends JFrame
                            implements ActionListener {
   private JButton b[];
   private String names[] =
      { "one", "two", "three", "four", "five", "six" };
   private boolean toggle = true;
   private Container c;
   private GridLayout grid1, grid2;

   public GridLayoutDemo()
   {
      super( "GridLayout Demo" );

      grid1 = new GridLayout( 2, 3, 5, 5 );
      grid2 = new GridLayout( 3, 2 );

      c = getContentPane();
      c.setLayout( grid1 );

      // create and add buttons
      b = new JButton[ names.length ];

      for (int i = 0; i < names.length; i++ ) {
         b[ i ] = new JButton( names[ i ] );
         b[ i ].addActionListener( this );
         c.add( b[ i ] );
      }

      setSize( 300, 150 );
      show();
   }

   public void actionPerformed( ActionEvent e )
   {
      if ( toggle )
         c.setLayout( grid2 );
      else
         c.setLayout( grid1 );
```

```
      toggle = !toggle;
      c.validate();
   }

   public static void main( String args[] )
   {
      GridLayoutDemo app = new GridLayoutDemo();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.15.6. מערך כרטיס – Card Layout

מערך זה הנו ייחודי בכך שהוא מכיל מספר מערכים יחדיו. לכל מערך נוכל להתייחס ככרטיס אחד ולמספר מערכים נוכל להתייחס כמספר כרטיסים. נוכל להכין מספר כרטיסים ולהכינם לשימוש.
הקונסטרקטורים הנם:

CardLayout()

CardLayout(int horz, int vert)

כאשר:
- הקונסטרקטור הראשון הנו ברירת המחדל.
- הקונסטרקטור השני מאפשר לנו להגדיר את המרווחים האופקיים והאנכיים בין הרכיבים.

שימוש במערך שכזה דורש יותר עבודה ממערך אחר. מערך כרטיס בד"כ מוחזק כאובייקט מסוג Panel אשר חייב להכיל CardLayout הנבחר כמנהל המערך.
ולפיכך אנו חייבים לייצור panel לכל כרטיס. לאחר מכן אנו מוסיפים את הרכיבים שיצרנו לכל כרטיס ל panel המתאים. לאחר מכן אנו מוסיפים את אותם panels לכל CardLayout.
ולבסוף, אנו מוסיפים את ה panel הנוכחי ל applet panel.
לאחר שביצענו את הצעדים הללו אנו חייבים לתת דרך למשתמש לבחור בין כרטיסים. דרך אחת מוכרת הנה לכלול כפתורי לחיצה בכל כרטיס אחד.
כאשר אנו מוסיפים panel נשתמש בפונקצית add() לפי המבנה הבא:

Component add(String name, Component panelObject)

כאשר:
Name – הנו שמו של הכרטיס שלה panel המוגדר ב panelObject.

לאחר שיצרנו את המערך הראשוני התכנית שלנו מפעילה את הכרטיס על ידי קריאה לאחת מהפונקציות CardLayout הבאות:

void first(Container deck)

void last(Container deack)

void next(Container deck)

void previous(Container deck)

void show(Container deck, String cardName)

כאשר:
Deck – הנו ההתייחסות ל container אשר מחזיק את הכרטיס ו cardName הנו שם הכרטיס.

שאר הפונקציות הנם בעצם קריאות לכרטיסים קרי ראשונת הבא, אחרון וכו'.

דוגמא

```
// Demonstrating CardLayout.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CardDeck extends JFrame
                      implements ActionListener {
   private CardLayout cardManager;
   private JPanel deck;
   private JButton controls[];
   private String names[] = { "First card", "Next card",
                              "Previous card", "Last card" };

   public CardDeck()
   {
      super( "CardLayout " );

      Container c = getContentPane();

      // create the JPanel with CardLayout
      deck = new JPanel();
      cardManager = new CardLayout();
      deck.setLayout( cardManager );

      // set up card1 and add it to JPanel deck
      JLabel label1 =
         new JLabel( "card one", SwingConstants.CENTER );
      JPanel card1 = new JPanel();
      card1.add( label1 );
      deck.add( card1, label1.getText() ); // add card to deck

      // set up card2 and add it to JPanel deck
      JLabel label2 =
         new JLabel( "card two", SwingConstants.CENTER );
      JPanel card2 = new JPanel();
      card2.setBackground( Color.yellow );
      card2.add( label2 );
      deck.add( card2, label2.getText() ); // add card to deck

      // set up card3 and add it to JPanel deck
      JLabel label3 = new JLabel( "card three" );
      JPanel card3 = new JPanel();
      card3.setLayout( new BorderLayout() );
      card3.add( new JButton( "North" ), BorderLayout.NORTH );
      card3.add( new JButton( "West" ), BorderLayout.WEST );
      card3.add( new JButton( "East" ), BorderLayout.EAST );
      card3.add( new JButton( "South" ), BorderLayout.SOUTH );
      card3.add( label3, BorderLayout.CENTER );
      deck.add( card3, label3.getText() ); // add card to deck

      // create and layout buttons that will control deck
      JPanel buttons = new JPanel();
      buttons.setLayout( new GridLayout( 2, 2 ) );
      controls = new JButton[ names.length ];

      for ( int i = 0; i < controls.length; i++ ) {
         controls[ i ] = new JButton( names[ i ] );
         controls[ i ].addActionListener( this );
         buttons.add( controls[ i ] );
      }
```

```
      // add JPanel deck and JPanel buttons to the applet
      c.add( buttons, BorderLayout.WEST );
      c.add( deck, BorderLayout.EAST );

      setSize( 450, 200 );
      show();
   }

   public void actionPerformed( ActionEvent e )
   {
      if ( e.getSource() == controls[ 0 ] )
         cardManager.first( deck ); // show first card
      else if ( e.getSource() == controls[ 1 ] )
         cardManager.next( deck );  // show next card
      else if ( e.getSource() == controls[ 2 ] )
         cardManager.previous( deck );  // show previous card
      else if ( e.getSource() == controls[ 3 ] )
         cardManager.last( deck );  // show last card
   }

   public static void main( String args[] )
   {
      CardDeck cardDeckDemo = new CardDeck();

      cardDeckDemo.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }
}
```

17.16. שימוש בתפריטים

חלון יכול להכיל תפריטים. תפריט יכול להכיל רשימה וממנה המשתמש יכול לבצע בחירה. כל בחירה שכזו מכילה בתוכה תת בחירה.
הקונספט המיישם זאת הנו תוך שימוש במחלקות JMenu ו JMenuBar ו JMenuItem.
באופן כללי, JMenuBar מכיל אחד או יותר אובייקטים מסוג Menu . כל אובייקט מסוג Menu מכיל אובייקטי JMenuItem. כאשר כל JMenuItem מייצג משהו שיכול להיבחר על ידי המשתמש.
מאחר ש JMenu הנו תת מחלקה של JMenuItem היררכיה של תת תפריטים יכולה להיווצר.
בנוסף אפשרי גם לייצור תפריטי בחירה, אשר אלו אופציות מסוג JCheckboxMenuItem ולהם תהיה האפשרות למשתמש לסמנם כאשר ייבחרו.
על מנת לייצור תפריט עלינו לייצור instance של JMenuBar. מחלקה זו רק מגדירה לנו את הקונסטרקטור עם ברירת המחדל.
לאחר מכן, עלינו לייצור instance של JMenu אשר יגדירו את הבחירה הנעשית.
מבנה הקונסטרקטורים הנו:

```
JMenu(String optionName)

JMenu(String optionName, boolean removable)
```

כאשר:
OptionName – הנו מגדיר את שם התפריט הנבחר
Removable – במידה וערכו הנו true אזי כי החלון שיופיע יכול להיות בעל תזוזה. אחרת הוא יישאר סטטי.

תפריטים אינדיווידואליים הנם מסוג MenuItem. המבנה הכללי הנו:

```
JMenuItem(String itemName)
```

כאשר:
ItemName – מייצג את שם התפריט

נוכל כמובן לנתק תפריט כלשהו על ידי שימוש בפונקצית disable(). והפעולה ההפוכה הנה enable().
נוכל לברר מהו מצב התפריט על ידי פונקצית isEnabled() .
מבנה הקונסטרקטורים הנו:

```
void disable()

void enable()

void isEnabled()
```

נוכל לשנות את שם התפריט על ידי קריאה ל setLabel(). וכמובן נוכל לדעת מהו שם התפריט תוך שימוש בפונקצית getLabel().
מבנה הקונסטרקטורים הנו:

```
void setLabel(String newName)

String getLabel()
```

נוכל לייצור תפריטי בחירה על ידי שימוש בתת מחלקה של ()MenuItem הנקראת ()CheckboxMenuItem. המבנה הכללי הנו:

JCheckboxMenuItem(String itemName)

כאשר:
ItemName – הנו השם המופיע בתפריט.

פריטים נבחרים מפעילים תגובה של כפתורי toggles. בכל פעם שתפריט נבחר מצבו משתנה. נוכל לראות את הסטטוס של תפריט הבחירה על ידי קריאה לפונקצית ()getState. נוכל לשנות את מצב התפריט תוך שימוש בפונקצית ()setState. מבנה הפונקציות הנו:

boolean getState()

void setState(boolean checked)

במידה ותפריט נבחר אזי פונקצית ()getState תחזיר ערך true. אחרת היא תחזיר ערך false. על מנת לבדוק פריט נעביר ערך true לפונקצית ()setState. על מנת לנקות תפריט נעביר false.

לאחר שיצרנו תפריט אנו חייבים להוסיף אותו לאובייקט מסוג Menu על ידי שימוש בפונקצית ()add והמבנה הכללי הנו:

JMenuItem add(MenuItem item)

לאחר שהוספנו פריט לאובייקט מסוג Menu נוכל להוסיף אובייקט לתפריט על ידי גרסה נוספת של פונקצית ()add. המבנה הכללי הנו:

JMenu add(Menu menu)

כאשר:
Menu – הנו התפריט שהוספנו.

אופי הפעולה:
תפריטים יוצרים מאורע רק כאשר הם נבחרים על ידי JMenuItem. הם אינם יוצרים מאורע כאשר אנו ניגשים לתפריט על מנת לראותו בלבד.
בכל פעם שפריט נבחר פונקצית ()action נקראת. שדה ה target של המאורע מכיל התייחסות לפעולה הרצויה. האובייקט המכיל את הפרמטרים מכיל התייחסות ל String המופיע בתפריט.

דוגמא

```
// Demonstrating menus
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MenuTest extends JFrame {
   private Color colorValues[] =
      { Color.black, Color.blue, Color.red, Color.green };
   private JRadioButtonMenuItem colorItems[], fonts[];
   private JCheckBoxMenuItem styleItems[];
   private JLabel display;
   private ButtonGroup fontGroup, colorGroup;
   private int style;

   public MenuTest()
   {
      super( "Using JMenus" );

      JMenuBar bar = new JMenuBar();  // create menubar
      setJMenuBar( bar );  // set the menubar for the JFrame

      // create File menu and Exit menu item
      JMenu fileMenu = new JMenu( "File" );
      fileMenu.setMnemonic( 'F' );
      JMenuItem aboutItem = new JMenuItem( "About..." );
      aboutItem.setMnemonic( 'A' );
      aboutItem.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               JOptionPane.showMessageDialog( MenuTest.this,
                  "This is an example\nof using menus",
                  "About", JOptionPane.PLAIN_MESSAGE );
            }
         }
      );
      fileMenu.add( aboutItem );

      JMenuItem exitItem = new JMenuItem( "Exit" );
      exitItem.setMnemonic( 'x' );
      exitItem.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               System.exit( 0 );
            }
         }
      );
      fileMenu.add( exitItem );
      bar.add( fileMenu );    // add File menu

      // create the Format menu, its submenus and menu items
      JMenu formatMenu = new JMenu( "Format" );
      formatMenu.setMnemonic( 'r' );

      // create Color submenu
      String colors[] =
         { "Black", "Blue", "Red", "Green" };
      JMenu colorMenu = new JMenu( "Color" );
      colorMenu.setMnemonic( 'C' );
```

```
      colorItems = new JRadioButtonMenuItem[ colors.length ];
      colorGroup = new ButtonGroup();
      ItemHandler itemHandler = new ItemHandler();

      for ( int i = 0; i < colors.length; i++ ) {
         colorItems[ i ] =
            new JRadioButtonMenuItem( colors[ i ] );
         colorMenu.add( colorItems[ i ] );
         colorGroup.add( colorItems[ i ] );
         colorItems[ i ].addActionListener( itemHandler );
      }

      colorItems[ 0 ].setSelected( true );
      formatMenu.add( colorMenu );
      formatMenu.addSeparator();

      // create Font submenu
      String fontNames[] =
         { "TimesRoman", "Courier", "Helvetica" };
      JMenu fontMenu = new JMenu( "Font" );
      fontMenu.setMnemonic( 'n' );
      fonts = new JRadioButtonMenuItem[ fontNames.length ];
      fontGroup = new ButtonGroup();

      for ( int i = 0; i < fonts.length; i++ ) {
         fonts[ i ] =
            new JRadioButtonMenuItem( fontNames[ i ] );
         fontMenu.add( fonts[ i ] );
         fontGroup.add( fonts[ i ] );
         fonts[ i ].addActionListener( itemHandler );
      }

      fonts[ 0 ].setSelected( true );
      fontMenu.addSeparator();

      String styleNames[] = { "Bold", "Italic" };
      styleItems = new JCheckBoxMenuItem[ styleNames.length ];
      StyleHandler styleHandler = new StyleHandler();

      for ( int i = 0; i < styleNames.length; i++ ) {
         styleItems[ i ] =
            new JCheckBoxMenuItem( styleNames[ i ] );
         fontMenu.add( styleItems[ i ] );
         styleItems[ i ].addItemListener( styleHandler );
      }

      formatMenu.add( fontMenu );
      bar.add( formatMenu );  // add Format menu

      display = new JLabel(
         "Sample Text", SwingConstants.CENTER );
      display.setForeground( colorValues[ 0 ] );
      display.setFont(
         new Font( "TimesRoman", Font.PLAIN, 72 ) );

      getContentPane().setBackground( Color.cyan );
      getContentPane().add( display, BorderLayout.CENTER );

      setSize( 500, 200 );
      show();
   }
```

```
   public static void main( String args[] )
   {
      MenuTest app = new MenuTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }

   class ItemHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         for ( int i = 0; i < colorItems.length; i++ )
            if ( colorItems[ i ].isSelected() ) {
               display.setForeground( colorValues[ i ] );
               break;
            }

         for ( int i = 0; i < fonts.length; i++ )
            if ( e.getSource() == fonts[ i ] ) {
               display.setFont( new Font(
                  fonts[ i ].getText(), style, 72 ) );
               break;
            }

         repaint();
      }
   }

   class StyleHandler implements ItemListener {
      public void itemStateChanged( ItemEvent e )
      {
         style = 0;

         if ( styleItems[ 0 ].isSelected() )
            style += Font.BOLD;

         if ( styleItems[ 1 ].isSelected() )
            style += Font.ITALIC;

         display.setFont( new Font(
            display.getFont().getName(), style, 72 ) );

         repaint();
      }
   }
}
```

17.17. שימוש בקופסת דיאלוג

לפעמים נרצה לייצור קופסת דיאלוג אשר תכיל בתוכה פקדי שליטה נוספים. Dialog box הנם תמיד ה"ילדים" של חלון האב המפעיל אותם. בנוסף לא קיימים בהם תפריטים.
Dialog boxes הנם אובייקטים מסוג Dialog ומבנה הקונסטרקטורים הו:

Dialog(Frame parenrWindow, boolean mode)

Dialog(Frame parentWindow, String title, boolean mode)

דוגמא

הבא נבחן את הדוגמא הקודמת ונוסיף בה את תכונת הדיאלוג קיימת. נשים לב כי אנו משתמשים בפונקציית dispose() אשר הנה פונקציה של Windows.

```
//illustrate Menus

import java.awt.*;
import java.applet.*;

/*
 <applet code = "DialogDemo" width=250 height=150>
 </applet>
 */

//create subclass of dialog
class SampleDialog extends Dialog
{
	SampleDialog(Frame parent, String title)
	{
		super(parent, title, false);
		setLayout(new FlowLayout());
		resize(300,180);

		add(new Label("Press this B"));
		add(new Button("Cancel"));
	}

	//Remove dialog box when user terminate it
	public boolean handleEvent(Event evtObject)
	{
		if(evtObject.id==Event.WINDOW_DESTROY)
		{
			dispose();
			return true;
		}
		return super.handleEvent(evtObject);
	}

	//Button
	public boolean action(Event evtObject, Object arg)
	{
		if(evtObject.target instanceof Button)
		{
			if(arg.equals("Cancel"))
			{
				dispose();
				return true;
			}
```

```
                        return false;
                }

                public void paint(Graphics g)
                {
                        g.drawString("This is Dialog Box", 10, 50);
                }
}

        //create subclass of Frame
        class MenuFrame extends Frame
        {
                String msg = "";
                CheckboxMenuItem debug, test;

                MenuFrame(String title)
                {
                        super(title);

                        //create menu bar and add it to frame
                        MenuBar mbar = new MenuBar();
                        setMenuBar(mbar);


                //create the menu item
                Menu file = new Menu("File");
                file.add(new MenuItem("New"));
                file.add(new MenuItem("Open"));
                file.add(new MenuItem("Close"));
                file.add(new MenuItem("Test"));
                file.add(new MenuItem("-"));
                file.add(new MenuItem("Exit"));
                mbar.add(file);

                Menu edit = new Menu("Edit");
                edit.add(new MenuItem("Cut"));
                edit.add(new MenuItem("Copy"));
                edit.add(new MenuItem("Paste"));
                edit.add(new MenuItem("Exit"));
                Menu sub = new Menu ("Special");

                sub.add(new MenuItem("First"));
                sub.add(new MenuItem("Second"));
                sub.add(new MenuItem("Third"));
                edit.add(sub);

                //checkable menus
                debug = new CheckboxMenuItem("Debug");
                edit.add(debug);
                test = new CheckboxMenuItem("Test");
                edit.add(test);

                mbar.add(edit);
        }
        //Hide window when terminate by the user
        public boolean handleEvent (Event evtObject)
        {
                if (evtObject.id == Event.WINDOW_DESTROY)
                {
                        hide();
                        return true;
```

```
            }
            return super.handleEvent(evtObject);
      }

            //display user choices
      public boolean action(Event evtObject, Object arg)
      {
            if(evtObject.target instanceof MenuItem)
            {
                  msg = "You Selected";

                  //active dialog box
                  if(arg.equals("New"))
                  {
                        msg += "New";
                        SampleDialog d = new SampleDialog(this, "New
Dialog Box");
                        d.show();
                  }

                  //try defining other dialog boxes
                  else if(arg.equals("Open"))
                        msg += "Open";
                  else if(arg.equals("Close"))
                        msg += "Close";

                  else if(arg.equals("Exit"))
                        msg += "Exit";

                  else if(arg.equals("Edit"))
                        msg += "Edit";

                  else if(arg.equals("Cut"))
                        msg += "Cut";

                  else if(arg.equals("Copy"))
                        msg += "Copy";

                  else if(arg.equals("Paste"))
                        msg += "Paste";

                  else if(arg.equals("First"))
                        msg += "First";

                  else if(arg.equals("Second"))
                        msg += "Second";

                  else if(arg.equals("Third"))
                        msg += "Third";

                  else if(arg.equals("Debug"))
                        msg += "Open";

                  else if(arg.equals("Test"));
                  msg += "Test";

                  repaint();
                  return true;
            }
            return false;
      }
```

```
	public void paint(Graphics g)
	{
		g.drawString(msg, 10, 140);

		if(debug.getState())
			g.drawString("Debug is on", 10, 160);
		else
			g.drawString("Debug is on", 10, 160);
		if (test.getState())
			g.drawString("Testing is on", 10, 180);
		else
			g.drawString("Testing is off", 10, 180);
	}
}

	//Create Frame Window

public class MenuDemo extends Applet
{
	Frame f;

	public void init()
	{
		f = new MenuFrame("Menu Tets");
		int width = Integer.parseInt(getParameter("width"));
		int height = Integer.parseInt(getParameter("height"));

		resize(width, height);

		f.show();
	}

	public void start()
	{
		f.show();
	}

	public void stop()
	{
		f.hide();
	}
}
```

17.18. שימוש בתפריטי Pop Up

מרבית האפליקציות כיום מורכבות ממתן אופציות נרחבות למשתמש. אחת מאלו הנה תפריטים "קופצים" אשר אנו יכולים לבצע העברת שליטה ומידע דרכם. ב Swing תפריטים אלו נוצרים על ידי מחלקת JPopupMenu שהנה תת מחלקה של JComponent.

דוגמא

```
// Demonstrating JPopupMenus
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class PopupTest extends JFrame {
   private JRadioButtonMenuItem items[];
   private Color colorValues[] =
      { Color.blue, Color.yellow, Color.red };

   public PopupTest()
   {
      super( "Using JPopupMenus" );

      final JPopupMenu popupMenu = new JPopupMenu();
      ItemHandler handler = new ItemHandler();
      String colors[] = { "Blue", "Yellow", "Red" };
      ButtonGroup colorGroup = new ButtonGroup();
      items = new JRadioButtonMenuItem[ 3 ];

      // construct each menu item and add to popup menu; also
      // enable event handling for each menu item
      for ( int i = 0; i < items.length; i++ ) {
         items[ i ] = new JRadioButtonMenuItem( colors[ i ] );
         popupMenu.add( items[ i ] );
         colorGroup.add( items[ i ] );
         items[ i ].addActionListener( handler );
      }

      getContentPane().setBackground( Color.white );

      // define a MouseListener for the window that displays
      // a JPopupMenu when the popup trigger event occurs
      addMouseListener(
         new MouseAdapter() {
            public void mousePressed( MouseEvent e )
               { checkForTriggerEvent( e ); }

            public void mouseReleased( MouseEvent e )
               { checkForTriggerEvent( e ); }

            private void checkForTriggerEvent( MouseEvent e )
            {
               if ( e.isPopupTrigger() )
                  popupMenu.show( e.getComponent(),
                                  e.getX(), e.getY() );
            }
         }
      );

      setSize( 300, 200 );
      show();
   }
```

```
   public static void main( String args[] )
   {
      PopupTest app = new PopupTest();

      app.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
               System.exit( 0 );
            }
         }
      );
   }

   private class ItemHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         // determine which menu item was selected
         for ( int i = 0; i < items.length; i++ )
            if ( e.getSource() == items[ i ] ) {
               getContentPane().setBackground(
                  colorValues[ i ] );
               repaint();
               return;
            }
      }
   }
}
```

18. תמונות – שימוש ב AWT

בחלק זה נבחן את השימוש בתמונות כחלק מתכונות ה AWT. התמיכה בתמונות נמצאת בתוך ספריות java.awt.image.
Images הנם אובייקטים של מחלקת Image אשר הנה חלק מחבילת java.awt אשר כוללים מספר מחלקות לשימוש והן:

ColorModel
CropImageFilter
DirectColorModel
FilteredImageSource
ImageFilter
IndexColorModel
MemoryImageSource
PixelGarbber
RGBImageFilter
This Interfaces are defined
ImageConsumer
ImageObserver
ImageProducer

18.1 יצירת אובייקט עם תמונה

לאור העובדה שתמונות חייבות להצטייר על החלון שאנו מבצעים עליו את הפעילות למחלקת Image אין מספיק מידע לגבי הנושא ולכן למחלקת Component שהנה חלק מ java.awt ישנה פונקציה אשר עוזרת לתמונה להצטייר. פונקציה זו הנה createImage().
המבנה הכללי הנו:

```
Image cretaeImage(ImageProducer imgProd)

Image cretaeImage(int width, int height)
```

כאשר:
הקונסטרקטור הראשון מחזיר לנו את התמונה המיוצרת על ידי imgProd אשר הנו אובייקט של מחלקה המיישמת את ImageProducer.
הקונסטרקטור השני מחזיר תמונה ריקה.

דוגמא

```
Canvas c = new Canvas();
Image test = c.sreateImage(200, 200);
```

18.2 טעינת תמונות

הדרך השנייה בה נוכל להציג תמונה הנה דרך טעינתה ממקום מסוים. על מנת לבצע זאת נשתמש בפונקצית ()getImage אשר מוגדרת על ידי מחלקת Applet.
מבנה הקונסטרקטורים הנו:

Image getImage(URL url)

Image getImage(URL url, String imageName)

כאשר:
הקונסטרקטור הראשון מחזיר לנו אובייקט המרכז את התמונה הממוקמת על ידי URL.
הקונסטרקטור השני מחזיר לנו את מיקום התמונה המוגדרת על ידי URL ושם התמונה מוגדר על ידי שם התמונה.

18.3 הצגת תמונה

לאחר שקיימת לנו התמונה נוכל להציג אותה תוך שימוש בפונקצית ()drawImage אשר הנה חבר במחלקת Graphics. לפונקציה זו קיימות צורות רבות לשימושים אך אנו נשתמש במבנה הבא:

boolean drawImage(Image imgObj, int left, int top, ImageObserver imgObj)

כאשר:
קונסטרקטור זה מציג לנו את התמונה המועברת כאובייקט ב imgObj עם מידות עליונות של צד שמאל.
ImgObj הנו התייחסות למחלקה אשר מיישמת את ImageObserver .

בעזרת שימוש בפונקציות ()getImage ו ()drawImage נוכל באופן פשוט להציג תמונות אשר יוצגו ב applet.

דוגמא

```
/*
<applet code = "SimplePic" width=150 height=150>
<param name = "img" value="test.jpeg">
</applet>
 */

import java.awt.*;
import java.applet.*;

public class SimplePic extends Applet
{
	Image img;

	public void init()
	{
		img = getImage(getDocumentBase(), getParameter("img"));
	}

	public void paint (Graphics g)
	{
		g.drawImage(img, 0, 0, this);
	}
}
```

19. Enterprise JavaBeans

נניח כי אנו רוצים לפתח יישום עם מספר שכבות על מנת לראות ולעדכן רשומות בבסיס הנתונים תוך שימוש בממשק אינטרנט.
נוכל לכתוב יישום תוך שימוש ב-JDBC את ממשק המשתמש נוכל לכתוב תוך שימוש ב-JSP/Servlets והמערכת יכולה להיות מבוזרת תוך שימוש ב-CORBA.
אך ישנם אלמנטים שעלינו לקחת בחשבון כאשר אנו מפתחים מערכת מבוזרת:
Performance – האובייקט המבוזר שאנו יוצרים חייב להיות בעל ביצועים בהכרח מאחר והם באופן פוטנציאלי יכולים לשמש מספר לקוחות ביחד.
Scalability – האובייקט חייב להיות בעל יכולת scalable משמע כי מספר המופעים של האובייקט יכולים לגדול ולעבור לשרת אחר ללא צורך בשינויים בקוד.
Security – לעיתים נרצה לבצע שימוש בהרשאות מצד הלקוח והשרת. באופן אידיאלי נרצה להוסיף משתמשים ותפקידים במערכת ללא צורך בקומפילציה מחודשת.
Distributed Transactions – אובייקט מבוזר חייב להיות בעל יכולת להתייחס לטרנזקציה באופן שקוף. לדוגמא, אם אנו עובדים עם 2 בסיסי נתונים נפרדים עלינו להיות מוכנים לפעולת עדכון באופן מקביל יחד עם אותה טרנזקציה.
Reusability – האובייקט המבוזר האידיאלי יכול להיות מועבר לסביבת עבודה אחרת.
Availability – באם אחד מהשרתים קורס לקוחות אמורים לעבור באופן אוטומטי לשרת אחר.

על מנת לתת מענה למכלול האלמנטים SUN הציגה את הקונספט של Enterprise JavaBeans specification (EJB).
EJB מתארים מודל רכיבים בצד השרת אשר נותן מענה למכלול האלמנטים שהוצגו לעיל.

19.1. JavaBeans vs. EJBs

מאחר והשמות זהים ישנו צורך בהבהרה בין JavaBeans ובין EJB.
בזמן ששניהם חולקים את אותה לוגיקה מבחינת יישום יחד עם design patterns שניהם נותנים יכולת פתרון אחרת.
הסטנדרטים המוגדרים ב- JavaBeans עוצבו על מנת לבצע שימוש חוזר ברכיבים שהינם נמצאים ב-IDE ואשר מוגדרים ויזואליים.
EJB מגדיר את מודל הרכיבים על מנת לפתח קוד בצד השרת. מאחר ו-EJB יכולים לרוץ בצד השרת ועל גבי שרתים שונים לא ניתן לבצע ספריות ואו רכיבים ויזואליים תוך שימוש ב- AWT ואו ב-Swing.

19.2. The EJB specification

EJB מתאר את מודל הרכיבים בצד השרת. הוא מגדיר 6 תפקידים אשר אנו משתמשים בהם על מנת לבצע פעולות פיתוח ומימוש כמו כן הוא מגדיר את הרכיבים במערכת.
להלן טבלה המתארת את התפקידים ואת אחריותם:

Role	Responsibility
Enterprise Bean Provider	The developer responsible for creating reusable EJB components. These components are packaged into a special jar file (ejb-jar file).
Application Assembler	Creates and assembles applications from a collection of ejb-jar files. This includes writing applications that utilize the collection of EJBs (e.g., servlets, JSP, Swing etc. etc.).
Deployer	Takes the collection of ejb-jar files from the Assembler and/or Bean Provider and deploys them into a run-time environment: one or more EJB Containers.
EJB Container/Server Provider	Provides a run-time environment and tools that are used to deploy, administer, and run EJB components.
System Administrator	Manages the different components and services so that they are configured and they interact correctly, as well as ensuring that the system is up and running.

19.3. EJB components

EJB הנם רכיבים רב פעמיים אשר באים מה- business logic. רכיבים אלו מאפשרים להיות נישאים יחד עם שירותים אחרים כגון אבטחה, cache וטרנזקציות.

EJB Container & Server

רכיב זה הינו run-time environment אשר מכיל ומריץ רכיבי EJB. הקונטיינר אחראי להגדרה בצד הלקוח. הקונטיינר מספק לנו low-level של EJB כולל טרנזקציות מבוזרות, אבטחה, ניהול אורך חיים, caching, threading, וניהול sessions.
שרת ה-EJB מוגדר כשרת היישום הכולל ויכול להריץ אחד או יותר רכיבי EJB.

Java Naming and Directory Interface (JNDI)

JNDI משמש אותנו על מנת לתת שמות לרכיבי ה-EJB אשר קיימים ברשת יחד עם קונטיינרים אחרים כמו טרנזקיות.

Java Transaction API/Java Transaction Service (JTA/JTS)

רכיבי JTA יחד עם JTS משמים אותנו על מנת לבצע טרנזקציות API. בצד ספק ה-EJB הוא יכול לבצע שימוש ב-JTS על מנת לייצור קוד לטרנזקציה למרות שקונטיינר ה-EJB בד"כ מממש טרנזקציות לרכיבי ה-EJB.

מצד הלקוח (the deployer) הוא יכול להגדיר את מאפייני הטרנזקציות של רכיב ה-EJB בזמן ריצה. קונטיינר ה-EJB אחראי לטפל בטרנזקציות ולא משנה האם הן מקומיות או מבוזרות.

CORBA and RMI/IIOP

ניתן לבצע שימוש בפרוטוקול CORBA לדוגמא ניתן ליישם זאת ע"י שימוש ב-JTS יחד עם JNDI המתייחסים לשירותי CORBA ומיישמים את ה-RMI בשכבה העליונה של של פרוטוקול CORBA/IIOP

19.4. The pieces of an EJB component

EJB מכיל מספר רכיבים, להלן:

Enterprise Bean

זהו רכיב מסוג מחלקה. הוא מיישם את הממשק של ה-enterprise bean ומספק יישום של פונקציות עסקיות אשר הרכיב אמור לבצע. המחלקה לא מיישמת קוד של authorization, authentication, multithreading, or transactional

Home interface

לכל Enterprise Bean שאנחנו יוצרים חייב להיות Home interface. מבחינת אופי הפעולה ה-Home interface משמש אותנו כמפעל של ה- EJB.

Remote interface

ממשק זה הנו ממשק JAVA אשר משפיע על הפונקציות של ה-Enterprise Bean אשר אנו רוצים לחשוף לעולם החיצוני.

Deployment descriptor

זהו קובץ מסוג XML המכיל מידע על ה- EJB. שימוש בפורמט זה מאפשר לנו לבצע שינויים מהירים במאפייני ה-EJB.

הקובץ שניתן לשינויים מכיל את האלמנטים הבאים:

1. The Home and Remote interface names that are required by your EJB
2. The name to publish into JNDI for your EJBs Home interface
3. Transactional attributes for each method of your EJB
4. Access Control Lists for authentication

EJB-Jar file

זהו קובץ java jar רגיל המכיל את ה- EJB יחד עם Home and Remote interfaces כמו כן את תיאור המימוש.

19.5. EJB operation

ברגע שיש לנו קובץ EJB-Jar המכיל את ה-bean יחד עם ה-Home and Remote interface ועם ה-deployment descriptor אנו יכולים לחבר את כלל האלמנטים ביחד ואז לבצע בהם שימוש בתהליך הקונטיינר.
לדוגמא, 5 לקוחות יכולים לבקש בו זמנית את היצירה של EJB תוך שימוש בממשק - Home and Remote interface ולפיכך הקונטיינר יגיב ע"י יצירה של EJB וחלוקתו לאותם לקוחות.

19.6. Types of EJBs

ישנם שני סוגי Beans עיקריים - *Session Beans* and *Entity Beans*

Session Beans

ניתן להשתמש בסוג זה על מנת להציג Use cases או תהליך עבודה מצד הלקוח. הם מייצגים פעולה או נתונים המשכיים אך לא את הנתונים עצמם!
ישנם שני סוגי Sesion beans והינם: *Stateless* and *Stateful*
כלל ה- Session beans חייבים ליישם את interface **javax.ejb.SessionBean**
Stateless Session Beans – הינם הסוג הפשוט ליישום EJB. אינם מחזיקים כלל conversational state עם הפונקציות בצד הלקוח כך שניתן לבצע בהם שימוש חוזר בצד השרת מאחר ואינם מבצעים פעולת cache. כאשר אנו מבצעים שימוש בסוג זה כלל ה-state חייבים להיות מאוחסנים מחוץ ל-EJB.
Stateful Session Beans – סוג זה מחזיק את state between invocations. מבחינת מיפוי היחס הינו אחד לאחד ללקוח וקיימת האפשרות לשמור state בתוך עצמם.
קונטיינר ה-EJB אחראי לבצע pooling ו- cache ל- Stateful Session Beans אשר ניתן לבצע ע"י *Passivation* או *Activation*.
במידה והקונטיינר קורס הנתונים של כלל ה-Stateful Session Beans יכולים להיאבד.

Entity Beans

רכיבים אלו מייצגים persistent data ואת התנהגות נתונים אלו. ניתן לחלוק את ה-Entity Beans בין לקוחות רבים באותו אופן בו ניתן לחלוק נתונים מבסיס הנתונים.
הקונטיינר אחראי לביצוע caching יחד עם שמירה על שלמות ה- Entity Beans. אורך החיים של ה-Entity Bean ארוך יותר מאשר הקונטיינר כך שאם הקונטיינר קורס ניתן למצוא את הנתונים השייכים ל-Entity Bean.
ישנם שני סוגים של Entity Bean:
1. Container Managed persistence
2. Bean-Managed persistence

Container Managed Persistence (CMP) – ל-CMP יש את ה- persistence מיושם בשמו ע"י קונטיינר ה-EJB. דרך מאפיינים המוגדרים ב- deployment descriptor קונטיינר ה-EJB מבצע מיפוי למאפייני ה- Entity Bean על מנת לבצע אכסון נתונים (בד"כ לבסיס הנתונים)
CMP מורידים את זמן הפיתוח של ה- EJB כמו כן את רמת הקוד.
Bean Managed Persistence (BMP) – ל-BMP יש את ה- - persistence מיושם ע"י ה-Enterprise Bean Provider. ולפיכך ה- Enterprise Bean Provider אחראי ליישם את הלוגיקה על מנת לייצור EJB חדש או לעדכן מאפיינים קיימים ב- EJB. בד"כ נצטרך לכתוב קוד JDBC. בתהליך BMP המתכנת אחראי על ניהול תהליך ה- Entity Bean persistence

19.7. Developing an EJB

כפי שהוזכר רכיב EJB חייב לכלול לפחות מחלקה אחת לטובת EJB ושתי ממשקים Remote and Home interfaces. להלן מדריך ליצירת remote interface:

1. The remote interface must be public.
2. The remote interface must extend the interface javax.ejb.EJBObject.
3. Each method in the remote interface must declare java.rmi.RemoteException in its throws clause in addition to any application-specific exceptions.
4. Any object passed as an argument or return value (either directly or embedded within a local object) must be a valid RMI-IIOP data type (this includes other EJB objects).

להלן דוגמא פשוטה ליישום:

```
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
  public long getPerfectTime()
    throws RemoteException;
} ///:~
```

20. תכנות ב Multithreading

שלא כמו שפות תכנות אחרות, שפת java נותנת לנו תמיכה בתכנות ב Multithreading. תכנות שכזה מכיל שני חלקים או יותר של תוכנית היכולים לרוץ במקביל. כאשר כל חלק כזה מתכנית נקרא thread ולכל thread מוגדר מסלול ריצה בנפרד. ולפיכך ניתן לומר כי Multithreading הנה צורה של Multitasking.

המינוח Multitasking מוכר לנו מכיוון שתכונה זו תומכת ברוב מערכות ההפעלה המתקדמות. אולם ישנם שני סוגים עיקריים ל Multitasking :

1. משימה המבוססת על תהליך – זוהי תכונה המאפשרת למחשב שלנו להריץ שני תכניות ויותר במקביל.
2. משימה המבוססת על thread – כאשר ה thread מוגדר כיחידת קוד קטנה ז"א תכנית בודדת אחת יכולה להריץ שני תהליכים או יותר במקביל.

ולפיכך ניתן לומר כי שימוש בסוג השני דורש מעט יותר משאבים.

Multithreading מאפשר לנו לכתוב תכניות יעילות אשר יכולות לנצל במידה כמעט אופטימלית את ה CPU וזאת מפאת הסיבה כי זמן ה"מנוחה" נשמר למינימום. תכונה זו חשובה ביותר על מנת לאפשר כתיבת תוכניות אינטראקטיביות במכלול מערכתי של רשתות תקשורת.

20.1.1. מודל ה Thread

java run time system תלוי ב threads בהרבה דברים וכל אותן מחלקות וספריות המעוצבות לצורך התמיכה בתכונה זו לוקחות חלק בהפעלת threads. ולמעשה java משתמשת ב threads על מנת לתת לנו סביבה א-סינכרונית. (שוב ניצולת CPU)

היתרון של Multithreading הנו שעיקר ה loop/polling (=תהליך הרצת קובץ במערכת) איננו "קיים" לכאורה. Thread אחד יכול לעצור ללא שום תלות בחלק אחר של התכנית.

בנוסף Multithreading מאפשר לנו שליטה גם באנימציה מסויימת כך שתהליכים אכן מתבצעים במקביל ושוב תוך ניצולת כמעט אופטימלית של ה CPU.

ל thread קיימים מספר מצבים:

- Running
- Ready to run
- Suspended
- Resumed
- Blocked

20.1.2. תזמון Threads

java נותנת לנו אפשרות תזמון לכל thread הקובעת כיצד יתנהגותו תהיה בהתייחס לאחרים. תיזמון threads הנם משתנים מסוג int אשר מגדירים לנו את התזמון בין thread אחד למשנהו.

כערך מוחלט, תזמון איננו בעל משמעות, ומכאן thread אשר לו מספר תזמון גבוה יותר לא רץ מהר יותר מאשר thread עם מספר נמוך יותר. אולם מתן התזמונים ל threads עוזר לנו להחליט מתי להחליף threads. פעולה זו נקראת בשפה המקצועית Context Switch.

החוקיות על מנת לקבוע תזמונים הנה:

1. כאשר thread מבקש שליטה – מצב זה קורה בד"כ כאשר אנו נשתמש ב sleeping, blocking, pending I/O וכו'. במקרה זה כלל ה threads האחרים נבחנים, והאחדבעל התזמון הגבוה ביותר ייכנס לעבודה.
2. כאשר thread בעל תזמון גבוה נכנס לאחר – במקרה זה ה thread בעל העדיפות הגבוה יותר ייכנס לעבודה ולא משנה מה ה thread בעל העדיפות הנמוכה יותר עושה או עשה תהליכו של זה פשוט מפסיק. פעולה זו נקראת בשפה המקצועית Preemptive multitasking.

במקרה בהם שני threads הנם בעלי אותו תזמון אזי כי המקרה מעט מורכב. למערכות מתקדמות כיום במקרה שכזה הפתרון הנו "חיתוך" אוטומטי של זמן ריצה.

20.1.3. סינכרון

מאחר ש Multithreading מוצג כתכונה הגורמת לנו לקבל תכנית שהנה א-סינכרונית, חיבת להיות דרך בה נוכל לבצע סינכרון כאשר אנו נצטרך זאת. לדוגמא, במידה ונרצה ששני threads יתקשרו בינהם ויחלקו מבנה נתונים מסוים כיצד נוכל לסנכרן בינהם?
לצורך פתרון הבעיה java מיישמת את מודל ה monitor.
ה monitor הנו שלט בקרה (כמו קופסא קטנה וירטואלית) היכולה להכיל רק thread אחד. ברגע שה thread נכנס לתוך ה monitor כל שאר ה threads מחכים עד אשר הוא יסיים את פעולתו.
לא קיימת מחלקת Monitor ובמקום לכל אובייקט הקיים monitor משלו יכול לקבל ולבצע סינכרון כאשר יידרש.
ברגע ש thread נמצא בתוך פונקצית הסנכרון אף thread אחר לא יכול לקרוא לסנכרון מאותו סוג אובייקט.

20.1.4. הודעה – Massages

ברגע שהתכנית שלנו מחולקת למספר threads **אנו נצטרך להגדיר כיצד הם יתקשרו האחד עם השני**. נוכל לבצע זאת על ידי קריאה לפונקציות מוגדרות מראש שלכלל האובייקטים יש. מערכת ההודעות ב java מאפשאת ל thread להיכנס לפונקציה המטפלת בסינכרון ואז לחכות עד אשר threads אחרים יודיעו כי thread זה הממתין "יוצא" מחוץ לפונקציה.

מחלקת Thread וה- Runnable Interface
Multithreading בנוי על מחלקת Thread כולל פונקציות וכמובן הממשק להרצה Runnable. מחלקת Thread מרכזת את ההרצה של ה threads. מאחר שאיננו יודעים לצפות במדויק מתי זמן הריצה של ה threads אזי כי נשתמש לצורך הרצה במחלקת Thread.
על מנת לייצור thread חדש במערכת נוכל לבצע זאת על ידי אחת משתי הפעולות הבאות:

1. להרחיב את מחלקת Thread (=הורשה)
2. יישום של ממשק Runnable

מחלקת Thread מגדירה מספר רב של פונקציות, הבא נסתכל על הפונקציות העיקריות אשר ישמשו אותנו:

Method	Meaning
getName	Obtain thread name
getPriority	Obtain thread priority
isAlive	Determine if thread is still running
join	Wait for a thread to terminate
resume	Resume exe of a suspended thread
run	Entry point for a thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method
suspend	Suspend a thread

20.2. ה – Thread הראשי

כאשר אנו מריצים תכנית ב java קיים בה כבר thread אחד. ובד"כ אנו מתייחסים אליו כאל main thread של התכנית שלנו מכיוון שהוא הראשון שרץ בתכנית.
ה main thread חשוב משתי סיבות:

1. ממנו אנו נקרא ל threads אחרים
2. זהו ה thread המסיים את כלל התהליכים.

למרות ש thread זה נוצר באופן אוטומטי אנו יכולים לשלוט בו תוך שימוש במחלקת Thread על ידי אובייקט. על מנת לעשות זאת אנו חייבים להחזיר התייחסות על ידי קריאהלפונקצית currentThrad() אשר הנה public static של מחלקת Thread.
המבנה הכללי הנו:

static Thread currentThread()

כאשר הפונקציה מחזירה התייחסות ל thread אשר נקרא. לאחר שיש לנו התייחסות ל thread זה נוכל לשלוט בו.

דוגמא

```
//Controlling the main Thread

class CurrentThreadTest{
      public static void main(String args[])
      {
            Thread t = Thread.currentThread();

            System.out.println("Current thread:" + t);

            //change the name of the thread
            t.setName("My Thread");
            System.out.println("After name change:" + t);

            try
            {
                  for(int n=5; n>0; n--)
                  {
                        System.out.println(n);
                        Thread.sleep(1000); //MilliSeconds
                  }
            }
            catch (InterruptedException e){

                  System.out.println("Main thread interrupted");
                   }
      }
      }
```

תוצאת התכנית

```
Current thread:Thread[main,5,main]
After name change:Thread[My Thread,5,main]
5
4
3
2
1
```

הסבר התכנית

בתכנית זו התייחסות ל current thread נעשית על ידי שימוש בפונקצית ()currentThread ואנו מאכלסים משתנה בשם t. לאחר מכן התכנית מציגה לנו מידע על ה thread.
לאחר מכן אנו משתמשים בפונקצית ()setName על מנת לשנות את שם ה thread שאנו רוצים. לאחר מכן אנו משתמשים בלולאת for שבה אנו סופרים את מרווח הזמן של שנייה אחת בין כל מופע .
לבסוף אנו משתמשים בפעולת try/catch על מנת לבדוק כי אכן פונקצית ()sleep מבצעת את פעולתה ולא גורמת לנו לטעות בתכנית.
הבא נשים לב לשורה הבאה [**My Thread,5,main**]After name change:Thread
שורה זו מדפיסה לנו את שם ה thread, תזמונו, ושם קבוצתו בהתאמה. (כאשר ברירת המחדל של שם הקבוצה ושל ה thread הנם בשם main).
Thread group הנו מבנה נתונים השולט בהצהרות של אוסף threads. תהליך זה מנוהל על ידי sysem run time .

20.3. יצירת Thread

באופן הכי פשוט אנו יכולים לייצור thread על ידי אתחול של אובייקט מסוג Thread. נוכל לבצע זאת על ידי שתי דרכים:

1. ליישם את ממשק Runnable
2. נוכל להרחיב את מחלקת Thread (=הורשה)

הבא נבחן את שתי הצורות הללו.

20.3.1. יישום Runnable

הדרך הקלה ביותר לייצור thread הנה לייצור מחלקה המיישמת את ממשק ה Runnable. ממשק Runnable בונה יחידה של exe קוד. אנו יכולים לבנות thread על כל אובייקט המיישם את Runnable. על מנת ליישם Runnable אנו חייבים ליישם את פונקצית run() במחלקה. המבנה הכללי הנו:

public abstract void run()

בתוך פונקצית run() אנו מגדירים את הקוד המכיל את ה thread החדש. חשוב להבין כי פונקציה זו יכולה לקרוא לפונקציות אחרות, כמו גם להשתמש במחלקות אחרות ולהגדיר משתנים כמו ה main thread.
ההבדל היחידי הנו שפונקציה זו מגדירה את נקודת ההתחלה ל thread אחר כאשר ה thread הזה יסתיים שפונקצית run() חוזרת.
לאחר שיצרנו מחלקה המיישמת את Runnable אנו נאתחל אותה עם אובייקט מסוג Thread מתוך המחלקה. Thread מגדיר לנו מספר קונסטרקטורים. האחד שנתמש בו מוגדר להלן:

Thread(Runnable threadObject, String threadName)

כאשר:
ThreadObject – הנו instance של המחלקה המיישמת את Runnable .
ThreadName – שם ה thread החדש.

לאחר שיצרנו את ה thread החדש נוכל להריץ אותו על ידי שימוש בפונקצית start() המוגדרת במחלקת Thread. מביחנת עבודה, פונקצית strat() מריצה קריאה לפונקצית run() .
המבנה כללי הנו:

synchronized vois start()

דוגמא

```
//Implementing Runnable

class NewThread implements Runnable
{
	Thread t;

	NewThread()
	{
		//create second thread
		t = new Thread(this, "Test Thread");
		System.out.println("Child thread:" +t);
		t.start(); //start the thread
	}

	public void run()
	{
		try
		{
			for(int n=5; n>0; n--)
			{
				System.out.println("Child thread:" + n);
				Thread.sleep(500);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println("Child Interrupted");
		}
		System.out.println("Existing Child Thread");
	}
}


	class ThreadTest
	{
		public static void main(String args[])
		{
			new NewThread(); //create a new thread

			try
			{
				for(int n=5; n>0; n--)
				{
					System.out.println("Main thread:" + n);
					Thread.sleep(1000);
				}
			}
			catch (InterruptedException e)
			{

				System.out.println("Main Thread
Interrupted");
			}
			System.out.println("Exisitng Main Thread");
		}
	}
```

תוצאת התכנית

```
Child thread:Thread[Test Thread,5,main]
Main thread:5
Child thread:5
Main thread:4
Child thread:4
Main thread:3
Child thread:3
Main thread:2
Child thread:2
Main thread:1
Child thread:1
Exisitng Main Thread
Existing Child Thread
```

הסבר התכנית

השימוש במילת הייחוס this מצביע כי אנו נרצה לייצור thread חדש ולקרו לפונקצית run() עם האובייקט הנוכחי.

נשים לב כי השתמשנו בפונקצית start() אשר מתחילה להפעיל את ה thread הראשון עם פונקצית run().

נקודה אחרונה לדגש, כפי שהוסבר ה main thread הנו האחרון לרוץ ולכן הוא חייב להיות מוגדר כאחרון בתכנית. במידה וה main thread מסיים לפני שה child thread מסיים הקומפיילר יכנס למצב של "תלות" קרי קפאון.

20.3.2. הרחבת מחלקת Thread

הדרך השנייה ליצור thread הנה לייצור מחלקה חדשה אשר מרחיבה את מחלקת Thread ואז לייצור unstance של המחלקה. המחלקה המורחבת חיבת לשכתב את פונקצית run() אשר מגדירה את נקודת הכניסה ל thread החדש. בנוסף אנו חייבים לקרוא לפונקצית start() על מנת להתחיל להריץ את ה threads.

דוגמא

```
//Extending Thread

class NewThread extends Thread
{
	NewThread()
	{
		//create second thread
		super("Test Thread");
		System.out.println("Child thread:" +this);
		start(); //start the thread
	}

	public void run()
	{
		try
		{
			for(int n=5; n>0; n--)
			{
				System.out.println("Child thread:" + n);
				Thread.sleep(500);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println("Child Interrupted");
		}
		System.out.println("Existing Child Thread");
	}
}

class ThreadTest
{
	public static void main(String args[])
	{
		new NewThread(); //create a new thread

		try
		{
			for(int n=5; n>0; n--)
			{
				System.out.println("Main thread:" + n);
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{

			System.out.println("Main Thread Interrupted");
		}
		System.out.println("Exisitng Main Thread");
	}
}
```

.20.4 יצירת Multiple Threads

עד כה עסקנו ביצירת thread בודד שאנו יצרנו וב main thread הקיים לנו במערכת. אולם יש באפשרותנו להגדיר מספר threads כל שנרצה בתכניתנו.

דוגמא

דוגמא זו ניצור שלושה Threads

```
//Extending Thread

class NewThread implements Runnable
{
	String name;
	Thread t;

	NewThread(String threadname)
	{
		name = threadname;
		t = new Thread(this, name);
			System.out.println("New thread:" + t);
		t.start(); //start the thread
	}

	// entry point for thread
	public void run()
	{
		try
		{
			for(int n=5; n>0; n--)
			{
				System.out.println(name + ":" + n);
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println(name + "Interrupted");
		}
		System.out.println(name + "Existing Thread");
	}
}


class MultiThread
{
	public static void main(String args[])
	{
		new NewThread("One"); //create a new thread
		new NewThread("Two");
		new NewThread("Three");

		try
		{
			Thread.sleep(10000);
		}
		catch (InterruptedException e)
		{
			System.out.println("Main Thread Interrupted");
		}
		System.out.println("Exisitng Main Thread");
```

```
      }
}
```

תוצאת התכנית

```
New thread:Thread[One,5,main]
New thread:Thread[Two,5,main]
New thread:Thread[Three,5,main]
One:5
Two:5
Three:5
One:4
Two:4
Three:4
One:3
Two:3
Three:3
One:2
Two:2
Three:2
One:1
Two:1
Three:1
OneExisting Thread
TwoExisting Thread
ThreeExisting Thread
Exisitng Main Thread
```

20.5. שימוש בפונקצית ()isAlive וב ()join

עד כה ראינו כיצד אנו מטפלים ב threads תוך מתן שהות ארוכה מספיק על מנת שכלל ה threads ירוצו. שיטה זו יכולה להיות מטרידה ובלתי יעילה ובנוסף עולה גם השאלה כיצד thread אחד יכול לדעת שהאחר סיים לרוץ?
לצורך הפתרון נוכל להשתמש בפונקציות הכלולות במחלקת Thread. כשלב ראשון אנו קוראים לפונקצית ()isAlive במבנה הכללי הבא:

final boolean isAlive() throws InterruptedException

כאשר הפונקציה מחזירה לנו ערך true באם ה thread עדיין רץ. אחרת היא תחזיר לנו ערך false.

ובד"כ נוכל להשתמש פונקצית ()join במבנה הכללי הבא:

final boolean join() throws InterruptedException

כאשר פונקציה זו מחכה עד אשר thread מסיים את עבודתו.

דוגמא

```
//Using isAlive() and join()

class NewThread implements Runnable
{
	String name;
	Thread t;

	NewThread(String threadname)
	{
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread:" + t);
		t.start(); //start the thread
	}

	// entry point for thread
	public void run()
	{
		try
		{
			for(int n=5; n>0; n--)
			{
				System.out.println(name + ":" + n);
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println(name + "Interrupted");
		}
		System.out.println(name + "Existing Thread");
	}
}


class JoinTest
{
```

```
      public static void main(String args[])
      {
            NewThread object1 = new NewThread("One"); //create a new
thread
            NewThread object2 = new NewThread("Two");
            NewThread object3 = new NewThread("Three");

            System.out.println("Thread one is alive:" +
object1.t.isAlive());
            System.out.println("Thread two is alive:" +
object2.t.isAlive());
            System.out.println("Thread three is alive:" +
object3.t.isAlive());

            //wait for threads to finish
            try
            {
                  System.out.println("wait for threads to finish");
                  object1.t.join();
                  object2.t.join();
                  object3.t.join();
            }
            catch (InterruptedException e)
            {
                  System.out.println("Main Thread Interrupted");
            }
            System.out.println("Thread one is alive:" +
object1.t.isAlive());
            System.out.println("Thread two is alive:" +
object2.t.isAlive());
            System.out.println("Thread three is alive:" +
object3.t.isAlive());
            System.out.println("Exisitng Main Thread");
      }
}
```

תוצאת התכנית

```
New thread:Thread[One,5,main]
New thread:Thread[Two,5,main]
New thread:Thread[Three,5,main]
Thread one is alive:true
Thread two is alive:true
Thread three is alive:true wait for threads to finish
One:5
Two:5
Three:5
One:4
Two:4
Three:4
One:3
Two:3
Three:3
One:2
Two:2
Three:2
One:1
Two:1
Three:1
OneExisting Thread
TwoExisting Thread
ThreeExisting Thread
```

```
Thread one is alive:false
Thread two is alive:false
Thread three is alive:false
Exisitng Main Thread
```

20.6. שימוש בפונקצית suspend() ו‎ב – resume()

לפעמים נרצה להשהות את עבודת ה thread עד אשר פעולה אחרת תתבצע. לאחר ההשהייה אנו נרצה להפעיל שוב את ה thread.

על מנת לבצע זוג זה של פעולות אזי כי נשתמש בשתי פונקציות במבנה הכללי הבא:

final void resume()

final void suspend()

דוגמא

```
//Using isAlive() and join()

class NewThread implements Runnable
{
	String name;
	Thread t;

	NewThread(String threadname)
	{
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread:" + t);
		t.start(); //start the thread
	}

	// entry point for thread
	public void run()
	{
		try
		{
			for(int n=10; n>0; n--)
			{
				System.out.println(name + ":" + n);
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println(name + "Interrupted");
		}
		System.out.println(name + "Existing Thread");
	}
}


class SuspendTest
{
	public static void main(String args[])
	{
		NewThread object1 = new NewThread("One"); //create a new
thread
		NewThread object2 = new NewThread("Two");
```

```
            //wait for threads to finish
            try
            {
                  Thread.sleep(1000);
                  object1.t.suspend();
                  System.out.println("Suspending Thread One");
                  Thread.sleep(1000);
                  object1.t.resume();
                  System.out.println("Resuming Thread One");

                  object2.t.suspend();
                  System.out.println("Suspending Thread Two");
                  Thread.sleep(1000);
                  object2.t.resume();
                  System.out.println("Resuming Thread Two");
            }
            catch (InterruptedException e)
            {
                  System.out.println("Main Thread Interrupted");
            }

            //wait for thead to finish
            try
            {
                  System.out.println("wait for thead to finish");
                  object1.t.join();
                  object2.t.join();
            }
            catch(InterruptedException e)
            {

                  System.out.println("Exisitng Main Thread");
            }
            System.out.println("Exisitng Main Thread");
      }

}
```

תוצאת התכנית

```
New thread:Thread[One,5,main]
New thread:Thread[Two,5,main]
One:10
Two:10
Suspending Thread One
Two:9
Resuming Thread One
Suspending Thread Two
One:9
Resuming Thread Two
wait for thead to finish
One:8
Two:8
One:7
Two:7
One:6
Two:6
One:5
Two:5
One:4
Two:4
One:3
```

```
Two:3
One:2
Two:2
One:1
Two:1
OneExisting Thread
TwoExisting Thread
Exisitng Main Thread
```

.20.7 תזמון Threads

תכונת התזמון עוזרת לנו על מנת להחליט איזה thread ירוץ בזמן נתון. תיאורתית עדיפות גבוהה ל thread אומר כי הוא יקבל יותר זמן CPU מאשר thread עם עדיפות נמוכה. הזמן שה thread יקבל בד"כ תלוי במספר פקטורים נוספים.
Thread בעל עדיפות גבוהה יותר יכול גם להיות מיוחס כבעל עדיפות נמוכה לדוגמא כאשר Thread בעל עדיפות נמוכה רץ ו thread בעל עדיפות גבוהה צריך להיכנס לעבודה אזי הוא יקבל עדיפות נמוכה. תיאורתית thread עם עדיפות זהה צריך לקבל גישה זהה ל CPU. אבל במצב זה אנו צריכים להיות זהירים. יש לזכור ש java עוצבה לעבודה בסביבות רבות. חלק מסביבות עבודה אלו מיישמות multitasking שונה מסביבות אחרות. לצורך האבטחה, threads אשר להם אותה עדיפות צריכים לשחרר את שליטתם אחת לפרק זמן כלשהו. פעולה שכזו תבטיח לנו שכלל ה threads יקבלו הזדמנות לרוץ מתחת למערכת הפעלה שאיננה מתנגשת עם threads.
מעשית, כלל ה threads ירוצו מהסיבה הפשוטה שהנם עובדים על ה CPU. לסוג זה של threads אנו נרצה לשלוט על מנת שאחרים ירוצו.
נוכל לתת עדיפות ל thread תוך שימוש בפונקצית setPriority() שהנה חלק ממחלקת Thread. המבנה הכללי הנו:

final void setPriority(int level)

כאשר:
Level – הנה העדיפות החדשה המוגדרת ל thread. הערך הניתן ל level חיב להיות בטווח של MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY אשר כברירת מחדל אנו נקבל את הערך 5. עדיפויות אלו מוגדרים כ final במחלקת Thread.

נוכל כמובן לדעת מהו הסטטוס הנוכחי של thread מסוים תוך שימוש בפונקצית final void getPriority().

דוגמא

בדוגמא זו נבחן כיצד שני threads בעדיפויות שונות אינם רצים על אותה מערכת הפעלה. Thread אחד מקבל עדיפות של 7 והאחר מקבל עדיפות של 3. ה threads מתחילים לרוץ למשך זמן של 10 שניות. כאשר כל אחד סופר מספר loops. לאחר זמן זה ה thread הראשי מסיים את כלל התכנית.

```
//Thread Priorities

class clicker implements Runnable
{
	int click = 0;
	Thread t;
	private boolean running = true;

	public clicker(int p)
	{
		t=new Thread(this);
		t.setPriority(p);
	}

	public void run()
	{
		while (running)
		{
			click++;
		}
	}

	public void stop()
	{
		running = false;
	}

	public void start()
	{
		t.start();
	}
}

class HighLowPri
{
	public static void main(String args[])
	{
		Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
		clicker high = new clicker(Thread.NORM_PRIORITY + 2);
		clicker low = new clicker(Thread.NORM_PRIORITY - 2);

		high.start();
		low.start();

		try
		{
			Thread.sleep(1000);
		}
		catch(InterruptedException e)
		{
			System.out.println("Main thread interrupted");
		}

		low.stop();
		high.stop();
```

```
            //wiat for child Thread to terminate
            try
            {
                  high.t.join();
                  low.t.join();
            }
            catch(InterruptedException e)
            {
                  System.out.println("Interrupted Exception caught");
            }
            System.out.println("Low Priority Thread: " + low.click);
            System.out.println("High Priority Thread: " +
high.click);
      }
}
```

תוצאת התכנית

נשים לב כי התכנית רצה על מערכת Windows XP ונכתבה ב Visual J#. ולכן תוצאת התכנית תהיה שונה ממערכות הפעלה אחרות.
תוצאת התכנית תלויה במהירות המעבד שלנו מכיוון שאנו מריצים thread המקבל זמן CPU. במקרה של תכנית זו התשובה מראה כי ה thread בעל העדיפות העליונה קיבל 100% זמן CPU לעומת ה thread עם העדיפות הנמוכה. זאת מכיוון שה thread בעל העדיפות הגבוהה מניע את ה CPU.

```
Low Priority Thread: 0
High Priority Thread: 317685241
```

20.8. סינכרון – Synchronization

כאשר שניים או יותר threads צריכים גישה על מנת לחלוק מקור מסוים נצטרך להגדיר זאת ולהבטיח כי אותו מקור יהיה לצורך שימוש לכלל ה Threads כל אחד בזמן ריצתו. על מנת לבצע פעולה זו אנו נשתמש בתכונת הסינכרון.
המפתח לסינכרון מבוסס על הקונספט של ה monitor שהנו אובייקט המשמש לנו כנעילה ומכאן רק thread אחד יכול להיות מוכל בתוך monitor בזמן נתון. ומכאן כאשר thread מבצע נעילה אזי כי הוא נכנס ל monitor. כל שאר ה threads המנסים להיכנס ל monitor יהיו בהמתנה עד אשר אותו אחד שרץ יסיים את תפקידו.
בהשוואה לשפות c,c++ ידוע לנו כי השימוש ב monitor הנו "טריקי" מכיוון שרוב השפות אינן תומכות בתכונה זו ומכאן שבשפות תכנות אחרות התכנית שלנו צריכה לאתחל נתונים פרימיטיביים על מנת לבצע סינכרון.
ב java ניתנת לנו האפשרות לבצע סינכרון על ידי שתי דרכים:

1. שימוש בפונקציה לסנכרון
2. הצהרה על סנכרון

20.8.1. שימוש בפונקציה לסינכרון

סינכרון הנו קל ב java מכיוון שלכלל האובייקטים ישנה את ההשפעה על ה monitor. על מנת להיכנס ל monitor אנו בסך הכל נקרא לפונקציה אשר מכילה את מילת המפתח. בזמן ש thread נמצא בתוך סינכרון כל שאר ה threads המנסים לקרוא לו באותו זמן ימתינו עד אשר יסתיים הסינכרון. על מנת לצאת מה monitor אנו בסך הכל נחזור מהפונקציה.
על מנת להבין כיצד התהליך מתבצע נתחיל עם דוגמא פשוטה **שאיננה** משתמשת בסנכרון ונראה כיצד קיים הצורך בתכונה זו.

דוגמא

בתכנית זו קיימים שלוש מחלקות הראשונה נקראת Callme ויש לה פונקציה הנקראת call(). פונקציה זו לוקחת String הנקרא msg. פונקציה זו מנסה להדפיס את ה msg בתוך סוגריים מרובעות. הדבר המעניין הנו שלאחר ההדפסה היא קוראת ל Thread.sleep(1000) אשר בעצם עוצר את ה Thread הנוכחי לשנייה אחת.

הקונסטרקטור של המחלקה השנייה נקרא Caller. הוא לוקח התייחסות למחלקת Callme ובנוסף String אשר מאוכלס ב target וב msg בהתאמה. הקונסטרקטור יוצר לנו thread חדש אשר יקרא לאובייקט המכיל את פונקצית run() . ה thread מתחיל לעבוד מיידית. פונקצית run() של ה Caller קוראת לפונקצית call() ב target ומעבירה את הודעת ה String. לבסוף מחלקת Synch מתחילה על ידי יצירת instace של Callme ושלוה instance של Caller כאשר לכל אחד ישנה הודעה ייחודית משלו.

```
//a non Synchronized program

class Callme
{
	void call(String msg)
	{
		System.out.print("[" + msg);
		try
		{
			Thread.sleep(1000);
		}
		catch (InterruptedException e)
		{
			System.out.println("Interrupted");
		}
		System.out.println("]");
	}
}

	class Caller implements Runnable
	{
		String msg;
		Callme target;
		Thread t;

		public Caller(Callme targ, String s)
		{
			target = targ;
			msg = s;
			t = new Thread(this);
			t.start();
		}

		public void run()
		{
			target.call(msg);
		}
	}

class Synch
{
	public static void main(String args[])
	{
		Callme target = new Callme();
		Caller object1 = new Caller(target, "Hello");
		Caller object2 = new Caller(target, "Synchronized");
		Caller object3 = new Caller(target, "World");
```

```
            //wait for child Thread to terminate
            try
            {
                  object1.t.join();
                  object2.t.join();
                  object3.t.join();
            }
            catch(InterruptedException e)
            {
                  System.out.println("Interrupted");
            }
      }
}
```

תוצאת התכנית

```
[Hello[Synchronized[World]
]
]
```

הסבר התכנית

אנו רואים כיצד שלושת ה threads מתחרים על מקום ריצה קרי אינם מסונכרנים, ואנו רואים זאת בתוצאת התכנית.

על מנת לבצע סינכרון בתכנית אנו נוסיף את מילת המפתח synchronized לפונקצית Callme והתכנית תהיה מסונכרנת.

```
class Callme
{
      synchronized void call(String msg)
      {
            System.out.print("[" + msg);
            try
            {
                  Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                  System.out.println("Interrupted");
            }
            System.out.println("]");
      }
}
```

20.8.2. שימוש בהצהרת סינכרון

הדרך האחרת והיעילה יותר על מנת לייצור סינכרון בתוכניות הנה שימוש בהצהרה על סינכרון. מקרה שכזה פותר לנו בעיות בהם לא נוכל לכתוב פונקצית סינכרון מפאת שהתכנית נכתבה על ידי צד שלישי לצורך העניין.
על מנת לבצע זאת נוכל להשתמש במבנה הבא:

```
synchronized (object){
//statement to be synchronized
}
```

דוגמא

ניקח לצורך העניין את הדוגמא שבה השתמשנו בחלק הקודם ונוסיף סינכרון לפונקצית run().

```
//Synchronized Statement

class Callme
{
	void call(String msg)
	{
		System.out.print("[" + msg);
		try
		{
			Thread.sleep(1000);
		}
		catch (InterruptedException e)
		{
			System.out.println("Interrupted");
		}
		System.out.println("]");
	}
}

class Caller implements Runnable
{
	String msg;
	Callme target;
	Thread t;

	public Caller(Callme targ, String s)
	{
		target = targ;
		msg = s;
		t = new Thread(this);
		t.start();
	}

	public void run()
	{
		synchronized (target)
		{
			target.call(msg);
		}
	}
}
```

```
class Synch
{
	public static void main(String args[])
	{
		Callme target = new Callme();
		Caller object1 = new Caller(target, "Hello");
		Caller object2 = new Caller(target, "Synchronized");
		Caller object3 = new Caller(target, "World");

		//wait for child Thread to terminate
		try
		{
			object1.t.join();
			object2.t.join();
			object3.t.join();
		}
		catch(InterruptedException e)
		{
			System.out.println("Interrupted");
		}
	}
}
```

20.9. תקשורת פנימית בין Threads

כפי שכבר הזכרנו Multithreading הנו תחליף מצויין לתכנות ב Loops כאשר השימוש הנו חלוקה לוגית של תהליכים. היתרון השני בשימוש ב threads הנו ב Polling.
Polling בד"כ מיושם על ידי Loop הבודק מצב מסוים לביצוע התנייא. ברגע שהתנאי אמת פעולה מסוימת מתבצעת. פעולה זו מבזבזת לנו משאבי מערכת ו CPU.
על מנת למנוע מצב של Polling קיים לנו ב java תהליך פנימי לתקשור בין Threads תוך שימוש בפונקציות wait(), notify(), notifyAll().
פונקציות אלו מיושמות כ final במחלקת Object כאשר כלל השלושה יכולות להיקרא בתוך סינכרון.
השימוש בפונקציות הנו פשוט:
wait() – אומר ל thread הקורא לוותר על ה monitor וללכת "לישון" עד אשר thread אחר ייכנס ל monitor ויקרא לפונקצית notify()
notify() – "מעיר" את ה thread הראשון אשר נקרא על ידי wait() באותו אובייקט
notifyAll() - "מעיר" את ה thread הראשון אשר נקרא על ידי wait() באותו אובייקט. ה thread בעל העדיפות הגבוהה ביותר ירוץ.

פונקציות אלו מוגדרות על ידי מחלקת Object באופן הבא:

final void wait()

final notify()

final notifyAll()

דוגמא

```
//Synchronized Statement

class Q
{
	int n;
	boolean valueSet = false;

	synchronized int get()
	{
		if(!valueSet)
			try
			{
				wait();
			}
			catch(InterruptedException e)
			{
				System.out.println("Interrupted Exception 
caught");
			}
		System.out.println("Got: " +n);
		valueSet = false;
		notify();
		return n;
	}

	synchronized void put(int n)
	{
		if(!valueSet)
			try
			{
				wait();
			}
			catch(InterruptedException e)
			{
				System.out.println("Interrupted Exception 
caught");
			}
		this.n = n;
		valueSet = true;
		System.out.println("Put: " +n);
		notify();
	}
}

class Producer implements Runnable
{
	Q q;
	Producer(Q q)
	{
		this.q = q;
		new Thread(this, "Producer").start();
	}

	public void run()
	{
		int i = 0;

		while(true)
		{
```

```
                        q.put(i++);
                }
        }
}

class Consumer implements Runnable
{
        Q q;
        Consumer(Q q)
        {
                this.q = q;
                new Thread(this, "Consumer").start();
        }

        public void run()
        {
                int i = 0;

                while(true)
                {
                        q.get();
                }
        }
}

class PCFixed
{
        public static void main(String args[])
        {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);

                System.out.println("Press Control-C to stop");
        }
}
```

20.10. באג מסוג – Deadlock

ישנה טעות מסוג מיוחד המתייחסת ישירות ל multitasking . טעות זו נקראת deadlock והיא נעשית כאשר לשני threads ישנו מעגל תלויות בזוג אובייקטים.

דוגמא, נניח כי אובייקט אחד ייכנס ל monitor , נקרא לאובייקט זה X ואובייקט אחר נכנס ל monitor ונקרא לו לצורך העניין Y. במידה ו thread X מנסה לקרוא לפונקצית סינכרון אשר נמצאת ב Y הוא ייחסם (כצפוי).

אולם, אם thread Y ינסה לקרוא ל X הוא יחכה לעד מאחר שעל מנת להיכנס ל X אנו חייבים לשחרר אותו.

באג מסוג זה הנו קשה לפתרון משתי סיבות:

1. הוא קורה לעיתים נדירות
2. הוא חייב להכיל שניים או יותר threads ואובייקטים מסונכרנים.

על מנת להבין את מהות הבאג הבא נבחן את הדוגמא הבאה.

דוגמא

```
//Deadlock Bug

class A
{
	synchronized void foo(B b)
	{
		String name = Thread.currentThread().getName();
		System.out.println(name + "entered A.foo");

		try
		{
			Thread.sleep(1000);
		}
		catch(InterruptedException e)
		{
			System.out.println("A interrupted");
		}

		System.out.println(name + "trying to call B.last()");
		b.last();
	}

	synchronized void last()
	{
		System.out.println("Inside A.last");
	}
}

class B
{
	synchronized void bar(A a)
	{
		String name = Thread.currentThread().getName();

		System.out.println(name + "entered B.bar");

		try
		{
			Thread.sleep(1000);
		}
		catch(InterruptedException e)
```

```
            {
                  System.out.println("B interrupted");
            }
            System.out.println(name + "trying to call A.last()");
            a.last();
      }


      synchronized void last()
      {
            System.out.println("Inside A.last");
      }
}

class DeadLock implements Runnable
{
      A a = new A();
      B b = new B();

      DeadLock()
      {
            Thread.currentThread().setName("MainThread");
            Thread t = new Thread(this, "Racingthread");
            t.start();

            a.foo(b);
            System.out.println("Back in main thread");
      }

      public void run()
      {
            b.bar(a);
            System.out.println("Back in other thread");
      }

      public static void main (String args[])
      {
            new DeadLock();
      }
}
```

תוצאת התכנית

```
MainThreadentered A.foo
Racingthreadentered B.bar
MainThreadtrying to call B.last()
Racingthreadtrying to call A.last()
```

21. תקשורת

בזמנים עברו המתכנת היה צריך לדעת מגוון רב של סביבות עבודה ולעיתים גם כיצד חומרה מתנהגת ופרוטוקולי תקשורת למינהם.
אולם הרעיון המרכזי בחלוקת מערכת אינו קשה כל כך והוא מחולק באופן מאוד לוגי בספריות java:

- קבלת מידע ממחשב מסוים והעברתו למחשב אחר מתבצעת על ידי תכנות מערכת בסיסי.
- קישור לבסיס הנתונים נעשה דרך שימוש ב Java DataBase Connectivity (JDBC) שהנו בעצם הרכב של פלטפורמת SQL.
- אספקת שירותי נט מתבצעת דרך Java's servlets and Java-Server Pages (JSPs)
- הרצת פונקציות java בשליטה מרחוק כאשר האובייקטים נמצאים במחשב המרוחק מתבצעת על ידי Java's Remote Method Invocation (RMI)
- שימוש בקוד שכתוב בשפות אחרות הרצות על ארכיטקטורות שונות מתבצע בעזרת Common Object Request Broker Architecture (CORBA)
- בידוד המבנה העסקי הלוגי מנושאי התחברות במיוחד התקשרות לבסיס נתונים, ניהול ואבטחת מידע נעשה על ידי Enterprise JavaBeans (EJBs). כאשר EJB איננה ארכיטקטורה של חלוקת משאבים אולם התוצר של האפליקציות הנו בד"כ שימוש בארכיקטורת client server
- הוספה והסרה של רכיבים לרשת המייצגים מערכת לוקאלית נעשה על ידי Java's Jini.

21.1. תכנות מערכת

אחת מיתרונות java הנו תכנות מערכת קל יחסית ולא בכדי ספריות java מסוגלות ל"הבין" מהו הקובץ היוצא ומהותו מו גם תקשורת בין מחשבים מרוחקים.
ככל שניתן תוך מתן דגש למערכות התכנות נבנה בכדי להתחשב ב JVM שעליו מותקנת התוכנה ומכאן שאנו נשתמש בקבצים על מנת לעשות זאת ולמעשה ה"רכיב" הפיזי של תקשורת מערכת הנו ה socket עם אובייקט מסוג stream ולפיכך כאשר נסיים את כתיבת התכניות אנו נשתמש באותן פונקציות שבהם השתמשו ב stream.
בנוסף, אנו ראה את השימוש בתכונת ה multithreading.

21.1.1. זיהוי מכונה

על מנת לומר למכונה פקודה כלשהי ממכונה אחרת ולהיות בטוח כי אנו מחוברים למכונה הנכונה חייבת להיות דרך לבצע זאת. בזמנים קדומים השתמשו בשמות ייחודיים לכל מכונה.
מכיוון ש java עובדת בסביבת אינטרנט אשר דורשת דרך בה נזהה בייחודיות את המכונה אליה אנו מתקשרים אנו נשתמש בכתובת IP הקיימת בשתי צורות:

1. המוכרת יותר הנה DNS (Domain Name System) – שם הדומיין שלי הנו iwatch.com ובאם יש לי מחשב שנקרא AviadTest בדומיין שלי שם הדומיין שלי יהיה AviadTest.iwatch.com. זוהי בדיוק אותה לוגיקה לשימוש במערכת מיילים ובד"כ תכונה זו שימושית בעולם האינטרנט.
2. האופציה הפחות מוכרת הנה שימוש במספר IP לדוגמא 198.233.13.56

בשני המקרים כתובת ה IP מיוצגת פנימית על ידי 32 ביט (כך שאף אחד לא יכול לעבור 255) ובכך נקבל אובייקט מיוחד של java המייצג את המספרים הללו באחד משתי הצורות שהוצגו תוך שימוש בפונקצית static InetAddress.getByName( ) הנמצאת בחבילת java.net.

למחלקת InetAddress לא קיימים קונסטרקטורים ויזואליים. על מנת לייצור אובייקט מסוג InetAddress אנו חייבים להשתמש באחת מהפונקציות הנמצאות לרשותנו. במקרה שלנו 3 הפונקציות לשימוש הנן:

static InetAddress getLocalHost()

static InetAddress getByName (String hostName)

static InetAddress [] getAllName()

כאשר:
הפונקציה הראשונה מחזירה לנו את אובייקט InetAddress אשר מייצג local host.
הפונקציה השנייה מחזירה לנו את כלל השמות. במידה ופונקציה זו איננה מסוגלת לפתור את כלל הכתובות אזי כי נזרקת לנו טעות מסוג UnknownHostException
הפונקציה השלישית שימושית בעולם האינטרנט מהסיבה הפשוטה ששם אחד יכול לייצג מספר מכונות. פונקציה זו מחזירה מערך של כתובות. במידה ואיננו יכול לפענח את אחד השמות אזי כי תזרק גם טעות מסוג UnknownHostException.

תוצאת האובייקט מסוג InetAddress אשר נוכל להשתמש בו על מנת לבנות את ה socket .

לצורך הבנה, נניח כי אנו מחוברים לאינטרנט בבית דרך ספק אינטרנט. בכל פעם שאנו מתחברים אנו מקבלים כתובת IP זמנית. אבל בזמן החיבור שלנו יש לנו כתובת IP ככל כתובת קבועה.
אם משהו מתחבר למכונה שלך תוך שימוש בכתובת ה IP אזי כי הוא יכול להתחבר לשרת אינטרנט או לשרת FTP אשר רצות על המכונה שלנו.
כמובן שהמתחבר אליך יצטרך לדעת מהי כתובת ה IP שלך. ומאחר שכיום כתובות IP הנן כתובות משתנות הכיצד נוכל לדעת מהי הכתובת שלנו בכל פעם שאנו מתחברים?
הבא נבחן את התכנית הבאה בה נשתמש בפונקציית ( )InetAddress.getByName על מנת לייצר לנו את כתובת ה IP. על מנת להשתמש בתכנית זו הנך חייב לדעת את שם המחשב שלך.

```java
// Finds out your network address when
// you're connected to the Internet.
import java.net.*;

public class WhoAmI
{
	public static void main(String[] args)
		throws Exception
	{
		if(args.length != 1)
		{
			System.err.println(
				"Usage: WhoAmI MachineName");
			System.exit(1);
		}
		InetAddress a =
			InetAddress.getByName(args[0]);
		System.out.println(a);
	}
}
```

ובכל מקרה תוצאת התכנית תהיה שם המחשב שלך עם כתובת IP בצורה הבאה:

```
peppy/199.190.87.75
```

הבא נבחן את השימוש בפונקציות הנ"ל:

```
//Demonstration of Internet Address

import java.net.*;

class InternetAdd
{
	public static void main (String args[])throws
UnknownHostException
	{
		InetAddress Address = InetAddress.getLocalHost();
		System.out.println(Address);
		Address = InetAddress.getByName("msn.com");
		System.out.println(Address);
		InetAddress Add[] =
InetAddress.getAllByName("www.walla.co.il");
		for (int i=0; i<Add.length; i++)
			System.out.println(Add[i]);
	}
}
```

תוצאת התכנית

```
your-c36yanb4fd/62.0.137.139
msn.com/207.68.172.246
www.walla.co.il/192.118.82.140
```

21.2. מודל Client/Server

הנקודה העיקרית במערכת תקשורת הנה לאפשר לשתי מחשבים או יותר לתקשר בינהם. ברגע שמחשב אחד מוצא את האחר נוצר מצב של תקשורת. אם כן נשאלת השאלה כיצד מחשב אחד ימצא את האחר?
הפתרון הנו מודל Client/Server כאשר השרת הנו המחשב המקומי והלקוח הנו המחשב המתקשר אליו. ההבדל בינהם חשוב רק בזמן שהלקוח מנסה לתקשר עם השרת. בזמן שהם מתוקשרים אזי כי קיימת לנו תקשורת דו סטרית וזה כלל לא משנה ברגע נתון זה איזה מחשב שרת ואיזה לקוח.
אם כך עבודת השרת הנה לעקוב אחר ההתקשרות וזה נוצר בעזרת אובייקט מיוחד שאנו יוצרים. עבודת הלקוח הנה לבצע את ההתקשרות עם השרת בעזרת אותו אובייקט שאנו יוצרים.
ברגע שהתקשורת נעשתה אזי כי התקשורת נהפכת לאובייקט I/O stream ומכאן נוכל להתייחס לחיבור כמו שאנו קוראים מקובץ כלשהו.
לפיכך ניתן לומר כי לאחר ההתקשרות אנו נשתמש בתכונות I/O אשר הוצגו בפרקים קודמים. זוהי בעצם אחת התכונות הנחמדות בתקשורת תוך שימוש ב java .

21.2.1. בדיקת תכניות ללא תקשורת

ישנם מצבים בהם לא תהיה לנו מכונת לקוח, שרת או מערכת לבדיקת התכניות שלנו. לצורך פתרון הבעיה קיימת תכונה הנקראת localhost והיא בעצם דימוי של IP ללא צורך במערכת. הדרך הג'נרית על מנת לייצור סוג זה של התקשרות הנה במבנה הבא:

```
InetAddress addr = InetAddress.getByName(null);
```

כאשר:
אם נתייחס לפונקצית getByName עם ערך 0 אזי כי ברירת המחדל הנה שימוש ב localhost.
ה InetAddress הנה הכתובת שבה נשתמש על מנת להתייחס למכונה מסויימת כאשר אי אפשר לשנות את תכולת זו.
הדרך היחידה לייצור InetAddress הנה דרך אחת המחלקות הנטענות שהנן static של פונקצית getByName( ), getAllByName( ), getLocalHost( ) .

נוכל בנוסף לייצור כתובת לוקאלית על ידי טיפול ב String:

```
InetAddress.getByName("localhost");
```

או כמובן נוכל לתת את כתובת ה IP הפנימית המצויה:

```
InetAddress.getByName("127.0.0.1");
```

השימוש בכל אחת מהפונקציות הנ"ל נותנת לנו את אותה תוצאה.

21.2.2. שימוש ב Port

כתובת IP איננה ייחודית מספיק על מנת לזהות שרת מסויים מאחר שהרבה שרתים יכולים להיות קיימים על מכונה אחת.
כל כתובת IP יכולה להכיל ports וכאשר אנו מתקינים client על השרת שלנו לדעת מהו ה port שבו גם השרת וגם הלקוח מתקשרים בינהם.
ה port איננו מיקום פיזי במכונה אלא הנו מבנה תוכנה. הלקוח יודע כיצד להתקשר למכונה דרך כתובת IP אך כיצד הלקוח מתקשר עם השירות הנכון שהוא רוצה?
במקום זה נכנסים מספרי ה ports ברמה השנייה של הכתובת.
הרעיון הנו שכאשר נבקש port מסוים אזי כי אנו מבקשים את השירות שה port הזה מקושר אליו.
לדוגמא, משך הזמן ביום הנו שירות. באופן טיפוסי כל שירות "מודבק" ל port מסוים הנמצא על שרת מסוים וזוהי אחריות הלקוח לדעת איזה שירות נמצא על איזה שרת.
שירותי המערכת מאפשרים לנו שימוש ב port החל מ 1 וכלה ב 1024. הדוגמא הראשונה לשימוש תהיה port בעל מספר 8080 שהנו ה port היותר מוכר לנו מסביבות אינטרנט.

21.2.3. מהו Socket?

ה socket הנו בעצם בנוי מתוכנה המשמשת אותנו לייצג "terminals" של התקשרויות בין שתי מכונות.
בזמן התקשרות נתון, ישנו socket על כל מכונה (באופן תיאורתי נוכל לחשבו על כבל מקשר בין שתי המכונות).
כמובן שמבנה החומרה איננו ידוע – בל נשכח כי כל העניין הנו בעצם יצירת תקשורת מבוססת תוכנה ולא חומרה!
בשפת java אנו יוצרים socket על מנת לייצור התקשרות למכונה אחרת ואז אנו מקבלים את ה InputStream ואת ה OutputStream מה socket על מנת לאפשר לנו להתייחס להתקשרות כאובייקט מסוג I/O stream.

ישנם שתי stream-based socket classes:
1. ServerSocket – שהשרת משתמש בו על מנת "לשמוע" תקשורת
2. Socket – שהלקוח משתמש על מנת לבצע את התקשורת.

לאחר שהלקוח ביצע את התקשרות ה Socket השרת ServerSocket מחזיר דרך שימוש בפונקצית accept() תגובה אשר אומרת באיזה דרך תתבצע ההתקשרות.
ומכאן קיים לנומצב של Socket to Socket .
בנקודה זו אנו משתמשים בפונקצית getInputStream( ) וב getOutputStream( ) על מנת לקבל תגובה של אובייקטים מסוג InputStream ושל OutputStream לכל Socket.
את התוצר אנו חייבים להכניס לתוך פורמט של מחלקות buffer.
אולם, ה ServerSocket יוצר לנו רכיב פיזי של "שרת" או שרת מכונה. כאשר אנו יוצרים ServerSocket אנו נותנים לו רק את מספר ה port. איננו צריכים לתת כתובת IP מפאת הסיבה שהכתובת כבר נמצאת במכונה המתקשרת.
אולם כאשר אנו יוצרים Socket אנו חייבים לתת את שניהם גם את כתובת ה IP וגם את מספר ה port אשר אליו אנו מנסים להתחבר. (פונקצית ServerSocket.accept( ) מחזירה לנו את כלל המידע).

דוגמא ליצירת שרת / לקוח

דוגמא זו בונה את השימוש הפשוט ביותר בארכיטקטורת שרת לקוח תוך שימוש ב Sockets.
עבודת השרת:
כל מה שהשרת עושה הוא לחכות לחיבור ואז משתמש ב Socket על מנת לייצור תקשורת ליצירת InputStream ו – OutputStream ואלו אנו נמיר ל Reader ול Writer ואז נעטוף את האובייקטים ב BufferedReader וב - PrintWriter. לאחר מכן הכל נקרא מתוך ה BufferedReader ומתבצע echo ל – PrintWriter עד אשר מתקבלת השורה END אשר בסופו של דבר סוגרת את התקשרות.
עבודת הלקוח:
הלקוח מבצע את התקשורת לשרת ואז יוצר את ה OutputStream ומבצע את אותה עטיפה כמו בשרת. שורות טקסט נשלחות דרך PrintWriter. הלקוח יוצר בנוף את ה InputStream על מנת לדעת מהי תגובת השרת.

גם השרת וגם הלקוח משתמשים באותו מספר port כאשר הלקוח משתמש בכתובת IP מקומית על מנת להתקשר לשרת באותה מכונה כך שלא צריך לבצע בדיקה על גבי הרשת (ז"א התכנית תרוץ לוקאלית).

צד השרת

```
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer
{
	// Choose a port outside of the range 1-1024:
	public static final int PORT = 8080;
	public static void main(String[] args)
		throws IOException
	{
		ServerSocket s = new ServerSocket(PORT);
		System.out.println("Started: " + s);
		try
		{
			// Blocks until a connection occurs:
			Socket socket = s.accept();
			try
			{
	System.out.println("Connection accepted: "+ socket);
			BufferedReader in =
			new BufferedReader(
			new InputStreamReader(
			socket.getInputStream()));
				// Output is automatically flushed
				// by PrintWriter:
			PrintWriter out =
			new PrintWriter(
			new BufferedWriter(
			new OutputStreamWriter(
			socket.getOutputStream())),true);
				while (true)
				{
					String str = in.readLine();
					if (str.equals("END")) break;
					System.out.println("Echoing: " + str);
					out.println(str);
				}
				// Always close the two sockets...
			}
			finally
			{
				System.out.println("closing...");
				socket.close();
			}
		}
		finally
		{
			s.close();
		}
	}
}
```

הסבר תכנית

ניתן לראות כי ServerSocket איננו צריך כתובת IP. כאשר אנו קוראים לפונקצית accept() הפונקציה חוסמת עד אשר הלקוח מתקשר. לאחר שנוצרה ההתקשרות, פונקצית accept() חוזרת עם אובייקט Socket אשר מייצג את ההתקשרות.

האחריות לנקות את ה sockets נלקחת בחשבון. באם הקונסטרקטור של ServerSocket נופל התכנית מפסיקה.

במקרה זה פונקצית main() זורקת יוצא דופן לטיפול כך שאין צורך להשתמש ב try. ובמקרה השני באם הקונסטרקטור של ServerSocket מצליח אזי כי כלל הקריאות צריכות להתבצע ולכן השימוש ב try/finally.

אותה לוגיקה מתקיימת גם ל Socket.

צד הלקוח

החלק השני של התכנית נראה בדיוק כמו פתיחה ואו סגירה של תכנית תוך שימוש ב InputStream וב OutputStream אשר נוצרים מתוך אובייקט ה Socket.
שני האוביקטים ה InputStream וה OutputStream מומרים לאובייקטים של Writer ושל Reader תוך שימוש במחלקות "המרה" קרי InputStreamReader ו – OutputStreamWriter ישירות.
בנוסף נוכל להשתמש במחלקות **InputStream** וב - **OutputStream** אשר נמצאות בגרסת java 1.0 כאשר בתוצאה ישנו יתרון מהותי של שימוש בגישת ה Writer. יתרון זה מופיע עם **PrintWriter אשר יש לו קונסטרקטור שנטען מחדש הלוקח ארגומנט שני, ו flag בוליאני המורה על ביצוע flush אוטומטי בכל פקודת println() בלבד.**
בכל פעם שאנו כותבים ל Out ה Buffer שלו חייב לבצע Flush על מנת שהמידע יעבור דרך הרשת.
ביצוע Flushing הנו חשוב מאוד בדוגמא הנוכחית מכיוון שהלקוח והשרת כל אחד מהם בנפרד מחכה לאותה שורת קוד מהצד השני על מנת להמשיך. במידה ופעולת Flushing איננה מתבצעת המידע לא יעבור דרך המערכת עד אשר ה buffer מלא ובאשר יגרום לנו בעיות במערכת.
כאשר אנו כותבים תוכניות למערכת תקשורת אנו צריכים להיות זהירים עם שימוש ב flush אוטומטי.
בכל פעם שאנו מבצעים flush ל buffer חבילת מידע חייבת להיארז ולהישלח. במקרה שלנו זה בדיוק מה שאנו רוצים מכיוון שחבילת המידע הכוללת את שורת הקוד איננה נשלחת אזי כי התקשורת לא תתבצע בין השרת ובין הלקוח.
במילים אחרות, סוף השורה הנה סוף ההודעה אך במקרים רבים הודעות אינן מיוחסות על פי מספר השורות ולפיכך יהיה יותר יעיל לא להשתמש ב flush אוטומטי ובמקום ניתן ל built in buffering להחליט מתי לבנות ולשלוח את חבילת הנתונים.
נשים לב כי כלל ה streams שאנו פותחים אלו הם שמבצעים buffer.
הלולאה האינסופית של while קוראת שורות מה BufferedReader in וכותבת מידע ל System.out ול - PrintWriter out.
כאשר הלקוח שולח את שורת ה END התכנית יוצאת מה Loop וסוגרת את ה Socket.

```java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr = InetAddress.getByName(null);
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket = new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
}
```

הסבר התכנית

בפונקצית ()main נוכל לראות את שלושת הדרכים על מנת לייצור **InetAddress של כתובת ה IP המקומית תוך שימוש ב null, localhost, 127.0.0.1 .**

כאשר מודפסת InetAddress התוצאה הנה:

```
localhost/127.0.0.1
```

נשים לב כי את ה Socket שאנו יוצרים הנקרא socket בתכנית אנו יוצרים גם עם InetAddress וגם עם מספר ה port.

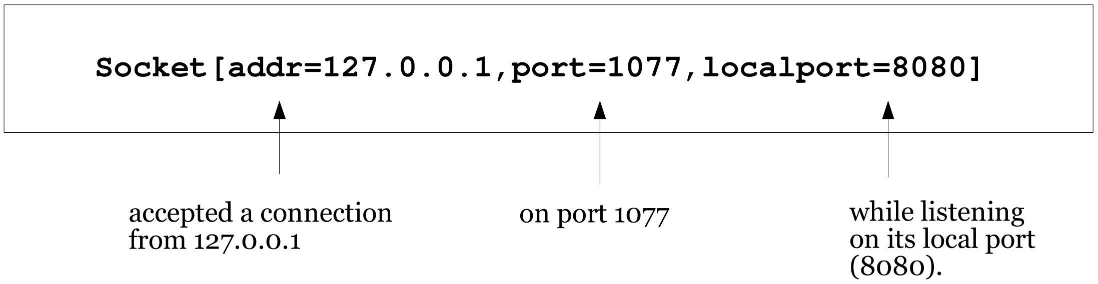
על מנת להבין מה קורה כאשר אנו מדפיסים את אחד מהאובייקטים של Socket נזכור כי התקשורת לאינטרנט מתבצעת ייחודית על פי הפרמטרים הבאים:

clientHost, clientPortNumber, serverHost, and serverPortNumber

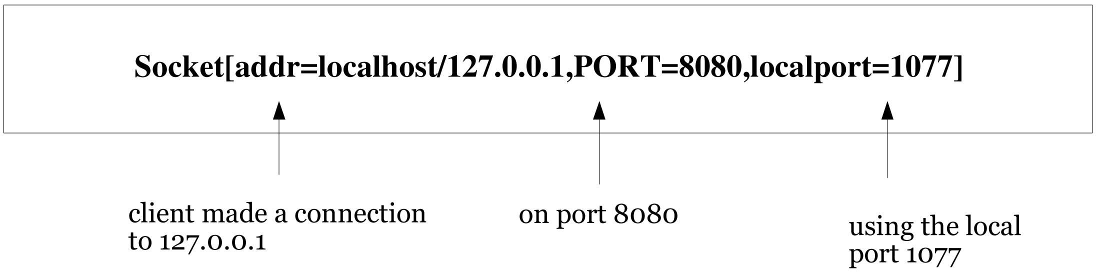
כאשר השרת "מתעורר" הוא לוקח את מספר ה port שנתנו לו בדומיין הלוקאי 8080. כאשר הלקוח "מתעורר" הוא ממוקם על ה port הבא בתור של המכונה הקיימת במקרה זה 1077.

על מנת שהנתונים ינועו בין השרת לבין הלקוח כל צד אמור לדעת מתי לשלוח אותם. ולפיכך במהלך תהליך התקשורת לשרת הידוע, הלקוח שולח את הכתובת המוחזרת על מנת שהשרת אן לשלוח את הנתונים.

Server Side



Client Side



נשים לב כי בכל פעם שאנו מפעילים את לקוח חדש מספר ה port המקומי עולה. לדוגמא התחלנו עם מספר 1025 ועולה הלאה עד אשר אנו מבצעים restart למכונה שלנו.
במערכות UNIX המספר עולה עד אשר מגיע לגבולו ואז באופן אוטומטי מחפש את מספר ה port הנמוך יותר.
ברגע שיצרנו אובייקטים מסוג Socket התהליך להפוך את האובייקט ל BufferedReader ול PrintWriter הנו כמו בצד השרת.

21.2.4. שירות למספר לקוחות

עד כה ראינו כיצב אנו מטפלים המקרה בודד בו קיים רק שרת אחד. מקרה שכזה הנו מקרה קצה ולא תואם את המציאות בד"כ.
על מנת לטפל בנושא זה הפתרון הנו שימוש ב Multithreading. המבנה הכללי הנו יצירת ServerSocket בודד בשרת ואז לקרוא לפונקצית accept( ) על מנת לחכות עד אשר התקשרות חדשה תתבצע. כאשר הפונקציה חוזרת אנו לוקחים את תגובת ה Socket ואנו משתמשים בה על מנת לייצור thread חדש אשר באופי עבודתו ישרת את הלקוח החדש. והתהליך חוזר חלילה.

דוגמא

```java
// A server that uses multithreading
// to handle any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in = new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        socket.close();
```

```
          } catch(IOException e) {
            System.err.println("Socket not closed");
          }
        }
      }
    }

    public class MultiJabberServer {
      static final int PORT = 8080;
      public static void main(String[] args)
          throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Server Started");
        try {
          while(true) {
            // Blocks until a connection occurs:
            Socket socket = s.accept();
            try {
              new ServeOneJabber(socket);
            } catch(IOException e) {
              // If it fails, close the socket,
              // otherwise the thread will close it:
              socket.close();
            }
          }
        } finally {
          s.close();
        }
      }
    }
```

הסבר התכנית

ה thread בשם ServeOneJabber לוקח את אובייקט ה Socket המיוצר על ידי פונקצית accept() שנמצאת בפונקצית ה main() בכל פעם שלקוח חדש מבצע התחברות. ואז כמו בדוגמאות הקודמות הוא יוצר BufferedReader ואובייקט PrintWriter שהנו flash אוטומטי המשתמש ב Socket. לבסוף, הפונקציה start() נקראת אשר מבצעת אתחול ל thread שקורא לפונקצית run().
ומכאן התהליך זהה בעצם לכל יצירת תקשורת חדשה כמו גם התהליך פעיל עד אשר תקרא שורת ה END.

לכלל התכניות יש לשים דגש על "ניקוי" ה socket ולטפל במצב זה כראוי. בדוגמא לעיל, ה socket נוצר מחוץ ל ServeOneJabber כך שהאחריות יכולה להתחלק. במידה והקונסטרקטור ServeOneJabber נופל הוא בסך הכל יזרוק יוצא דופן (=טעות) אשר תנקה את ה thread. ובמידה והקונסטרקטור מצליח אזי כי אובייקט ServeOneJabber לוקח אחריות לניקוי ה thread בתוך פונקצית run().
נשים לב לפשטות של MultiJabberServer. כמו בעבר ה ServerSocket נוצר ואז פונקצית accept() נקראת על מנת לבצע התקשרות. אך הפעם הערך המוחזר של הפונקציה הנ"ל מועבר לקונסטרקטור ל ServeOneJabber אשר יוצר thread חדש המטפל בתקשורת. כאשר התקשורת מפסיקה אזי כי הthread פשוט יעלם.
במידה והתוצר של ServerSocket נופל אזי כי תיזרק טעות דרך פונקצית ה main(). ובמידה והתוצר מצליח טיפול מסוג try-finally מבטיח לנו ניקיון תקשורת. הלולאה הפנימית של try-catch עובדת אך ורק במקרה של נפילת הקונסטרקטור ServeOneJabber .
במידה והקונסטרקטור מצליח אזי כי ה thread בשם ServeOneJabber יסגור את ה socket הקשור אליו.

על מנת לבדוק האם השרת מטפל במספר לקוחות אזי כי נבחן את התכנית הבאה. נשים לב כי מקסימום מספר ה threads האפשרי נקבע על ידי final int MAX_THREADS

```
// Client that tests the MultiJabberServer
// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread
{
	private Socket socket;
	private BufferedReader in;
	private PrintWriter out;
	private static int counter = 0;
	private int id = counter++;
	private static int threadcount = 0;
	public static int threadCount()
	{
		return threadcount;
	}
	public JabberClientThread(InetAddress addr)
	{
		System.out.println("Making client " + id);
		threadcount++;
		try
		{
			socket =
				new Socket(addr, MultiJabberServer.PORT);
		}
		catch(IOException e)
		{
			System.err.println("Socket failed");
			// If the creation of the socket fails,
			// nothing needs to be cleaned up.
		}
		try
		{
			in =
				new BufferedReader(
				new InputStreamReader(
				socket.getInputStream()));
			// Enable auto-flush:
			out =
				new PrintWriter(
				new BufferedWriter(
				new OutputStreamWriter(
				socket.getOutputStream())), true);
			start();
		}
		catch(IOException e)
		{
			// The socket should be closed on any
			// failures other than the socket
			// constructor:
			try
			{
				socket.close();
			}
			catch(IOException e2)
			{
				System.err.println("Socket not closed");
```

```
                        }
                }
                // Otherwise the socket will be closed by
                // the run() method of the thread.
        }
        public void run()
        {
                try
                {
                        for(int i = 0; i < 25; i++)
                        {
                                out.println("Client " + id + ": " + i);
                                String str = in.readLine();
                                System.out.println(str);
                        }
                        out.println("END");
                }
                catch(IOException e)
                {
                        System.err.println("IO Exception");
                }
                finally
                {
                        // Always close it:
                        try
                        {
                                socket.close();
                        }
                        catch(IOException e)
                        {
                                System.err.println("Socket not closed");
                        }
                        threadcount--; // Ending this thread
                }
        }
}
public class MultiJabberClient
{
        static final int MAX_THREADS = 40;
        public static void main(String[] args)
                throws IOException, InterruptedException
        {
                InetAddress addr =
                        InetAddress.getByName(null);
                while(true)
                {
                        if(JabberClientThread.threadCount()
                                < MAX_THREADS)
                                new JabberClientThread(addr);
                        Thread.currentThread().sleep(100);
                }
        }
}
```

הסבר התכנית

ניתן לראות כי הקונסטרקטור JabberClientThread לוקח את InetAddress ומשתמש בו על מנת לייצור התקשרות חדשה.
יש לשים לב לתבנית הפעולה, ה Socket תמיד משמש אותנו על מנת לייצור אובייקט כמעין Reader/Writer (InputStream and/or OutputStream=) אשר הנה הדרך היחדיה שאפשר להשתמש ב socket.
נשים לב שוב כי פונקצית ()start מבצעת אתחול וקוראת לפונקצית ()run . ברגע זה הודעות נשלחות בין השרת לבין הלקוח אולם יש לשים דגש על כך כי ל thread ישנו אורך חיים מסויים עד אשר הוא מסיים את פעולתו (=הצלחה ואו כשלון).
נשים לב לניקיון ה socket אחד שכזה מתבצע אם הקונסטרקטור נפל לאחר שנוצר ה socket אך לפני שהקונסטרקטור משלים את עבודתו.
אחרת, האחריות לקריאת פונקצית ()close ניתנת לאחריותה של פונקצית ()run.
ה threadcount סופר לנו ומבצע מעקב אחר מספר האובייקטים שנוצרים JabberClientThread. הוא מבצע תהליך של עלייה כחלק מהקונסטרקטור (עלייה מספרית) ומבצע ירידה מספרית כאשר פונקצית ()run מסיימת.

21.2.5. שימוש ב Datagrams

עד כה ראינו שימוש בפוטוקול TCP/IP אשר מעוצב ומבטיח לנו כי הנתונים יגיעו ליעדם. פרוטוקול זה מאפשר שליחה מחודשת של נתונים אבודים וכמובן שתהליך העברת הנתונים מתבצע דרך מספר נתיבים.
כל היתרונות הללו עולים כסף – והרבה!
למרות הכל זהו הפרוטוקול השמיש ביותר בתעשיות המתקדמות.

קיים פרוטוקול נוסף בשם User Datagram Protocol (UDP) אשר אינו מבטיח כי חבילות המידע יועברו באותו סדר בו הם נוצרו. פרוטוקול זה נקרא בשפה המקצועית "פרוטוקול שאינו אמין" שלכאורה נשמע רע אך למרות שהוא מהיר יותר הוא יעיל.
ישנם אפליקציות מסוימות כמו אותות מוסיקה אשר שימוש בסוג זה של פרוטוקול לא יכול להיות קריטי עד כדי כך.

21.2.6. שימוש ב URL מתוך Applet

יש באפשרותנו להציג כל URL ולתקשר דרך ה Applet שיצרנו. נוכל לבצע זאת על ידי שימוש בשורת הפקודה הבאה:

```
getAppletContext().showDocument(url);
```

דוגמא

בדוגמא זו נשתמש במודל ה swing.

```
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class ShowHTML extends JApplet {
  JButton send = new JButton("Go");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    send.addActionListener(new Al());
    cp.add(send);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // This could be a CGI program instead of
        // an HTML page.
        URL u = new URL(getDocumentBase(),
          "FetcherFrame.html");
        // Display the output of the URL using
        // the Web browser, as an ordinary page:
        getAppletContext().showDocument(u);
      } catch(Exception e) {
        l.setText(e.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new ShowHTML(), 100, 50);
  }
}
```

21.2.7. קריאת קובץ משרת

עד כה קראנו קבצים הנמצאים בצד השרת. בדוגמא זו נראה כיצד אנו יכולים לקרוא קבצים הנמצאים בצד הלקוח.

```
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class Fetcher extends JApplet {
  JButton fetchIt= new JButton("Fetch the Data");
  JTextField f =
    new JTextField("Fetcher.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    fetchIt.addActionListener(new FetchL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(fetchIt);
  }
  public class FetchL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
        t.setText(url + "\n");
        InputStream is = url.openStream();
        BufferedReader in = new BufferedReader(
          new InputStreamReader(is));
        String line;
        while ((line = in.readLine()) != null)
          t.append(line + "\n");
      } catch(Exception ex) {
        t.append(ex.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new Fetcher(), 500, 300);
  }
}
```

21.3. בסיס נתונים JDBC

כיום בטכנולוגיות הקיימות משוערך כי לפחות מחציתן עובדות בארכיטקטורת שרת/לקוח ומכאן אחד היתרונות המהותיים ב java הנה היכולת לבנות בסיס נתונים עצמאי המושתת על ארכיטקטורה זו.
אחת הבעיות היותר מהותיות בבסיסי נתונים הנה היכולת לתקשר בין בסיסי נתונים שונים. למרות שקיימת שפת SQL התומכת במרבית בסיסי הנתונים עדיין קיימת בעית התקשרות בין בסיסי נתונים של חברות שונות כגון מייקרוסופט כנגד אורקל.
JDBC עוצב על מנת לתת פתרון לבעיה מסוג זה. אולם עדיין קיימת האפשרות לבצע קריאה מסויימת מתוך JDBC לאותו בסיס נתונים שנרצה.
אחד המקומות בהם אנו מהנדסי התוכנה נצטרך להשתמש בשפת SQL הנה כאשר נשתמש בהצהרת CREATE TABLE כאשר אנו יוצרים טבלה חדשה ומגדירים את סוג ה SQL לכל עמודה בטבלה.
למרבה הצער ישנו הבדל מהותי בין יצירת הטבלאות בשפת SQL לבין מוצרים שונים הקיימים בשוק.
בסיסי נתונים שונים התומכים בסוג SQL באותה סמנטיקה ומבנה יכולים לתת לנו שמות אחרים לאותם סוגים. בסיסי הנתונים העיקריים תומכים ב SQL ובסוגי נתוניו החל מערכים בינאריים גדולים: באורקל סוג זה נקרא LONG RAW בסיס נתונים של Sybase קורא לסוג זה IMAGE בסיס נתונים Informix קורא לסוג זה BYTE ובסיס נתונים DB2 קורא לסוג זה LONG VARCHAR FOR BIT DATA.
תאימות הנו נושא חשוב ביותר לצורך הרצת קבצים וזאת מכיוון שמשתמשים שונים מריצים קבצים על מערכות שונות.
JDBC כמו הרבה API המצויים ב Java עוצב למתן פתרונות. הפונקציה שאנו קוראים לה על מנת לבצע את הפעולה באופן לוגי מבצעת אותה קרי קישור לבסיס הנתונים, יצירת הצהרה והרצת השאילתא.
על מנת לאפשר שאכן הפלטפורמה תהיה בלתי תלויה JDBC מספק לנו driver manager אשר באופן דינמי מחזיק את כלל אובייקטי הדריברים אשר בסיס הנתונים יצטרך.
ולפיכך באם בחברה מסוימת קיימים מספר בסיסי נתונים שונים לדוגמא 5 שכאלה אזי כי נצטרך 5 אובייקטים.
אובייקט הדרייבר רושם את עצמו ביחד עם מנהל הדריברים בזמן הטעינה, ואנו יכולים להכריח את הטעינה להשתמש ב **Class.forName( )**.

על מנת לפתוח בסיס נתונים עלינו להגדיר בסיס נתונים URL המגדיר את:

1. שאנו משתמשים ב JDBC עם JDBC
2. את תת הפרוטוקול קרי שם הדרייבר או שם בסיס הנתונים להתקשרות. מאחר שפיתוח JDBC נעשה על בסיס ODBC תת הפרוטוקול הראשון המצוי נקרא JDBC-ODBC bridge המוגדר על ידי ODBC
3. זיהוי בסיס הנתונים קרי כיצד נזהה את בסיס הנתונים במבנה הלוגי שלו כגון מיקום הימצאו, טבלאות, שאילתות וכו'.

כלל המידע הנ"ל מוכל לתוך String אחד הנקרא "database URL“. לדוגמא על מנת להתקשר דרך תת פרוטוקול של ODBC לבסיס נתונים המזוהה בשם people נרשום את קוד ה URL הבא:

```
String dbUrl = "jdbc:odbc:people";
```

במידה ואנו מתקשרים בתוך רשת, ה URL של בסיס הנתונים יכיל את מידע התקשורת המזהה את המכונה הנכונה ברשת.
לדוגמא

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

בדוגמא זו אנו רואים את השימוש בקישור URL לבסיס הנתונים. הקוד מורכב משני חלקים קרי החלק הראשון הנו "jdbc:rmi://192.168.170.27:1099/" והוא משתמש ב RMI על מנת לבצע את התקשורת לבסיס הנתונים המרוחק הנמצא ב Port 1099 בכתובת IP 192.168.170.27. ואילו החלק השני של ה URL "jdbc:cloudscape:db" מעביר את הקונפיגורציה של תת הפרוטוקול ואת שם בסיס הנתונים. חלק זה יעבוד אך ורק אם החלק הראשון הצליח במשימתו.

כאשר אנו מוכנים להתקשר לבסיס הנתונים שלנו אנו נקרא לפונקציה static בשם DriverManager.getConnection( ) שהנה פונקצית מערכת ובעזרתה נעביר את ה URL של בסיס הנתונים, שם המשתמש, והסיסמא על מנת לגשת לבסיס הנתונים. כאשר אנו מבצעים זאת אנו מקבלים בחזרה את אובייקט התקשורת שאנו נשתמש בו על מנת לבצע מניפולציות ושאילתות בבסיס הנתונים.

בדוגמא הבא נראה כיצד אמ-נו פותחים תקשורת בין בסיס נתונים המכיל מידע התקשרות והמחפש את האדם על פי שם משפחתו המופיע לנו בשורת הפקודה. התכנית בוחרת רק את אותם אנשים להם קיימת כתובת מייל ואז התכנית מדפיסה לנו את המתאימים לבחירה שלנו.

```java
// Looks up email addresses in a
// local database using JDBC.
import java.sql.*;

public class Lookup {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    // Load the driver (registers itself)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, user, password);
    Statement s = c.createStatement();
    // SQL code:
    ResultSet r =
      s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // Capitalization doesn't matter:
      System.out.println(
        r.getString("Last") + ", "
        + r.getString("fIRST")
        + ": " + r.getString("EMAIL") );
    }
    s.close(); // Also closes ResultSet
  }
}
```

22. שפת Java ושימוש ב- XML

22.1. מהו XML?

- המינוח המלא הנו **EX**tensible **M**arkup **L**anguage
- שפה זו הוצגה בשנת 1998.
- שפה זו בדיוק כמו שפת HTML והנה שפת סימונים.
- שפת XML עוצבה על מנת לתת לנו פתרון לתיאור נתונים.
- התגים שבהם אנו משתמשים ב XML אינם מוגדרים מראש. עלינו להגדיר את התגים שלנו.
- בשפת XML אנו משתמשים ב Document Type Definition (DTD) או ב XML Schema על מנת לתאר את הנתונים.
- XML עם DTD או עם Schema הנו מעוצב להיות בעל תיאור עצמאי.
- שפת XML נתמכת בארגון הסטנדרטים העולמיים של W3C.

DIFFERENCES	DTD'S	SCHEMAS
LANGUAGE	DTD's are written using EBNF (Extended Backus-Naur Form)	Schemas are written using XML.
DATA CONSTRAINTS	DTD's have minimal data constraints.	Schemas allow more specific constraints
USER DEFINED TYPES	DTD's limit to a fixed set of content models	Schemas provide greater flexibility and expressing content.

22.2. הבדלים בין XML ובין HTML

- XML אינו תחילף לשפת HTML.
- XML ו HTML עוצבו למטרות שונות:
  - XML עוצבה על מנת לתאר נתונים במטרה לומר מה הנתונים מייצגים.
  - HTML עוצבה על מנת להציג נתונים בדגש על מראה הנתונים והצגתם.

ניתן לומר כי HTML עוצבה על מנת להציג מידע ואילו XML עוצבה על מנת לתאר את הנתונים.

22.3. שפת XML לא עושה הכל

שפת XML לא עוצבה לתת לנו מענה כולל. אולי קצת קשה לנו להבין זאת אבל שפת XML לא תתן לנו מענה כולל על הכל. XML נוצרה על מנת לתת לנו מבנה, אחסון ולשליחה של מידע.
להלן דוגמא לשליחת הערה מ Tove ל Jani

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

הסבר:
נשים לב כי ל note יש את הכותרת הפותחת ואת סגירת התגים. בנוסף קיים המידע על שם השולח ושם המקבל. יחד עם זאת מסמך זה אינו מבצע את הכל, קרי זה ס"ה מידע נקי שעטוף בתגים של XML. על מנת שנוכל לשלוח זאת עלינו לכתוב קוד שישלח את המידע ויציג אותו.

22.4. יצירת תגים ב XML

כל עת שאנו נעקוב אחרי חוקי ה XML יהיה לנו קל לכתוב תגים חדשים.
תגים של XML נראים אותו דבר כמו תגים של HTML.
להלן דוגמא:

```
<book_title>XML PROGRAMMING WITH JAVA </book_title>
<author>Mark

Wilson</author>
<publisher>Manning

Publications</publisher>
<copydate>2000</copydate>
<retailprice>$1,000,000</retailprice>
<ISBN>

1884777872</ISBN>
```

22.5. חוקי XML

קיימים לנו 10 חוקים לכתיבת תגים XML. להלן החוקים:
1. יש לכתוב תגי XML שיהיו שימושיים באינטרנט.
2. על תגי XML לתמוך במגוון אפליקציות.
3. על תגי XML להיות תומכים ב SGML.
4. המטלה של כתיבת תוכנית אמורה להיות קלה לביצוע ולתהליך.
5. עלינו לשמור על מספר התכונות האופטימלי ב XML לרמה מינימלית.
6. מסמכי XML אמורים להיות בנויים באופן קריא וברור.
7. עיצוב מסמכי XML אמור להיות מהיר.
8. עיצוב המסמכים אמור להיות פורמלי.
9. יש לשמור על רמה של פשטות.
10. קיצור של קוד ותגי XML אינו בחשיבות על.

22.6. שפה גמישה וחופשיה

כפי שכבר אמרנו תגי XML אינם מוגדרים מראש. עלינו לייצור את התגים שלנו. לצורך השוואה עם HTML, עלינו להגדיר את התגים מראש. כותב הקוד בשפת HTML יכול להשתמש רק בתגים שבהם השפה כתובה.
שפת XML מאפשרת לנו לייצור תגים משלנו ולייצור מבנה מסמכים משלנו. למשל התגים בדוגמא הקודמתת <from><to> אינם תגים קיימים בשפת HTML.

22.7. XML כהשלמה ל HTML

חשוב להבין כי שפת XML איננה מחליפה את שפת HTML. בפיתוחי אינטרנט עתידיים סביר להניח כי שימוש בשפת XML יתבצע על מנת לתאר נתונים בעוד שהשימוש בשפת HTML יעשה על מנת להציגם.
הדרך הטובה ביותר להציג את שפת XML הנה:
XML הנה שפה עצמאית לכלל הפלטפורמוטת, תוכנות וחומרה והנה כלי להעברת מידע.

22.8. הפרדת נתונים

כאשר אנו משתמשים בשפת HTML על מנת להציג נתונים, הנתונים מאוחסנים בתוך ה HTML שלנו. בשפת XML כאשר אנו מציגים נתונים, הנתונים יכולים להיות מאוחסנים בקבצי XML שונים. בדרך זו אנו יכולים להתרכז בבניית HTML נכון ולהיות בטוחים כי שינויים מהותיים לא יידרשו מאיתנו זמן נוסף.
XML יכול להיות מאוחסן בתוך HTML כ "data island".

22.9. העברות מידע

בעזרת XML אנו יכולים להעביר ולקבל מידע ממערכות שונות בעלות פלטפורמות עבודה שונות ללא קשר לשפות התכנות, חומרה קיימת או שינויי תוכנה קיימת.
נשים לב כי אחת מהמטלות הקריטיות בעולם פיתוח התוכנה הנה תאימות נתונים. כך שאנו משתמשים בשפת XML לא קיימת לנו תאימות נתונים.

22.10. שימוש XML ב B2B

כבר היום קיימים שימושים בתעשיית האינטרנט תוך שימוש בשפת XML. שפה זו תופסת תאוצה כאחת מהשפות העיקריות להחלפת מידע כגון מידע עסקי, פיננסי ועוד.
מאחר ששפת XML מאוחסת כטקסט Text Plain תכונה זו מאפשרת לנו חופשיות ממוצרי חומרה ותוכנה קיימים אצל הלקוח.

22.11. XML ושפות נוספות

אחת מהשפות הידועות כיום הנה WML (Wireless Markup Language). שפה זו מאפשרת כתיבת אפליקציות על גבי מכשירים סלולריים. שפה זו כתובה בעזרת XML.
דוגמא למסמך XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

הסבר:
שורה ראשונה הנה הגדרת המסמך להיות מסוג XML. הגדרה זו מגדירה את הגרסה ואת שפת הEncoding לשימוש במסמך. בדוגמא זו הגרסה הנה מספר 1, ואנו משתמשים בשפה לטינית/אירופאית.

השורה הבאה:
מתארת לנו את אלנט השורש של המסמך. אנו יכולים לחשוב על זאת כ"מסמך זה הנו "note"

```
<note>
```

4 השורות הבאות:
שורות אלו מתארות את ה"ילדים" של שורש המסמך (to, from, heading, and body)

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
```

ולבסוף:
סגירת התג.

```
</note>
```

22.12. אלמנטים

כלל האלמנטים בהם נשתמש צריכים להיות סגורים בסוף שימוש.
לדוגמא בשפת HTML אם אנו מבצעים שימוש באלמנט של <P> אזי כי נוכל להשאיר פתוח והדפדפן יזהה זאת ללא בעיה. בשפת XML כלל האלמנטים אמורים להיות סגורים, אם נגדיר שימוש ב <P> אזי כי נצטרך לסגור את התג.
להלן דוגמא:
דוגמא ראשונה מראה שימוש באלמנט <P> בשפת HTML. שימוש זה הוא נכון

```
<p>This is a paragraph
<p>This is another paragraph
```

דוגמא שנייה מראה לנו שימוש בתג <P> בשפת XML. יש לשים לב כי כאשר פתחנו תג עלינו לסגור אותו.

```
<p>This is a paragraph</p>
<p>This is another paragraph</p>
```

* שפת XML הנה מוגדרת כשפת case sensitive. ויש לשים לב בעת הכתיבה. לדוגמא:
השורה הראשונה מראה לנו טעות.
בעוד שהשורה השנייה תקנית.

```
<Message>This is incorrect</message>

<message>This is correct</message>
```

22.13. שימוש נכון בתגים

שימוש לא נכון ב nesting בשפת XML יכול לייצור לנו טעות. לעומת שפת HTML שהשימוש מותר ולא ממש משנה לנו היכן אנו כותבים את התגים, בשפת XML זה קריטי.

דוגמא:
בדוגמא הראשונה ישנו שימוש בשפת HTML, נשים לב כי התגים לא בהכרח בסדר מסויים.

```
<b><i>This text is bold and italic</b></i>
```

בדוגמא השנייה ישנו שימוש בשפת XML, נשים לב כי עלינו להגדיר במדוייק את התגים.

```
<b><i>This text is bold and italic</i></b>
```

22.14. שורש מסמכים

כלל מסמכי XML אמורים להיות עם שורש. כל שאר האלמנטים בהם נשתמש יהיו בתוך השורש.

דוגמא:
נשים לב כי השורש שלנו הוא <root> ואילו אלמנטים נוספים הנם <child><subchild>

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

22.15. אלמנטים המכילים מאפיינים

נוכל לזכור משפת HTML את הקוד הבא:
<IMG SRC="computer.gif">
מאפיין ה SRC נותן מידע נוסף לגבי התמונה. בשפות XML ו HTML מאפיינים מכילים מידע נוסף על האלמנטים.

דוגמא:

```
<img src="computer.gif">
<a href="demo.asp">
```

מאפיינים בד"כ נותנים לנו מידע שאינו חלק מהנתונים. בדוגמא להלן file type אינו רלוונטי לנתונים יחד עם זאת הוא חשוב לתוכנה שרוצה לעשות מניפולציה על האלמנט.

```
<file type="gif">computer.gif</file>
```

סגנון מרכאות
ערכי מאפיינים חייבים להיות סגורים במרכאות, אך נוכל להשתמש במרכאות כפולות או בודדות.

דוגמא:

```
<person sex="female">
```

או נוכל לכתוב גם כך:

```
<person sex='female'>
```

נשים לב כי אם ערך המאפיין עצמו מכיל מרכאות כפולות זה הכרחי להשתמש במרכאות בודדות לפי הדוגמא להלן:

```
<gangster name='George "Shotgun" Ziegler'>
```

וההיפך, אם ערך המאפיין עצמו מכיל מרכאות בודדות זה הכרחי להשתמש במרכאות כפולות לפי הדוגמא להלן:

```
<gangster name="George 'Shotgun' Ziegler">
```

22.16. תכונות

בשפת XML אלמנטים יכולים לכלול תכונות כגון שמות וערכים בדיוק כמו שפת HTML. בשפת XML הערך של התכונה חייב להיות תמיד במרכאות.

דוגמא:
נשים לב לדוגמא הראשונה. דוגמא זו איננה נכונה מכיוון שיש לנו אלמנט שהנו תאריך וערכי התארים אינם במרכאות.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note date=12/11/2002>
<to>Tove</to>
<from>Jani</from>
</note>
```

דוגמא שנייה, נשים לב כי הוספנו לערכים מרכאות ובמצב כזה הקוד תקין.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note date="12/11/2002">
<to>Tove</to>
<from>Jani</from>
</note>
```

22.17. שימוש ברווח

שלא כמו HTML משפט כגון:
Hello my name is Tove
ייתן לנו תוצר של
Hello my name is Tove

בעוד שבשפת XML אסור לנו לייצור מרווחים ללא שימוש.

22.18. הערות ב XML

כתיבת הערות מתבצעת בדיוק כמו שפת HTML באופן הבא:
<!—This is comment-->

22.19. שורה חדשה ב XML

שורה חדשה תמיד תהיה כברירת מחדל בצד שמאל. באפליקציות חלונאיות, שורה חדשה בד"כ מאוחסנת כזוג מאפיינים כגון carriage return (CR) and line feed (LF) וכך המקרה לגבי XML. תמיד שורה חדשה תהיה ב LF כמו ב UNIX.

22.20. אלמנטים והארכות ב XML

אחד היתרונות הבולטים בשפת XML הנה היכולת לבצע הוספות למסמכים קיימים ללא בעיות מורכבות.

דוגמא
הבא נראה את שורות הקוד הבאות

```
<note>
<to>Tove</to>
<from>Jani</from>
<body>Don't forget me this weekend!</body>
</note>
```

הבא נחשוב כי יצרנו אפליקציה שנתנה לנו את התוצר הבא:

MESSAGE

To: Tove
From: Jani

Don't forget me this weekend!

כעת, עלינו להוסיף מידע למסמך, ונשאלת השאלה האם האפליקציה תיפול או תפסיק לרוץ?

```
<note>
<date>2002-08-01</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

התשובה היא לא. האפליקציה עדיין יכולה למצוא את האלמנטים הקיימים בה ולייצור את התוצר.

22.21. יחס אלמנטים ב XML

על מנת שנוכל להבין את הטרמינולוגיה של XML עלינו להבין את היחס בין האלמנטים בדף ה XML כמו גם עלינו להבין כיצד תכולת האלמנטים מתוארת.

דוגמא:
נניח כי אנו מתארים את הספר הבא:

My First XML

Introduction to XML

- What is HTML
- What is XML

XML Syntax

- Elements must have a closing tag
- Elements must be properly nested

נניח כי מסמך ה XML להלן מתאר את הספר:

```
<book>
<title>My First XML</title>
<prod id="33-657" media="paper"></prod>
<chapter>Introduction to XML
<para>What is HTML</para>
<para>What is XML</para>
</chapter>

<chapter>XML Syntax
<para>Elements must have a closing tag</para>
<para>Elements must be properly nested</para>
</chapter>

</book>
```

הסבר:
שורש המשמך הנו <book>. הילדים הנם Title, prod, chapter. לפיכך נוכל לומר כי book הנו ההורה של שאר הילדים.

22.22. אלמנטים ותכולתם

אלמנט ב XML הנו מוגדר ככל דבר החל מהגדרתו הראשונה וכלה בסגירת ההגדרה בתג.
אלמנט יכול להכיל תכולה נקודתית, מעורבת (טקסט ומספרים) או פשוטה(טקסט) ואפילו ריקה (לא מכיל מידע). כמובן שאלמנט יכול לכלול מאפיינים.
אם נסתכל על הדוגמא הקודמת אזי נוכל לומר כי, ל book ישנו אלמנט מכיל מכיוון שהוא מכיל אלמנטים אחרים. Cahpter מכיל תכולה מעורבת מכיוון שהוא מכיל טקסט ואלמנטים אחרים. Para מכיל תכולה פשוטה מכיוון שזה רק טקסט ואילו Prod תכולתו ריקה (לא מכיל מידע).

22.23. מתן שמות לאלמנטים

קיימים מספר חוקים למתן שמות לאלמנטים. להלן החוקים:

1. שמות יכולים להכיל אותיות, מספרים ומאפיינים אחרים.
2. שמות לא יכולים להתחיל במספר או במאפיינים וסמלים אחרים.
3. לשמות אסור להתחיל באותיות XML.
4. שמות לא יכולים להכיל רווח.

כאשר אנו נפתח אפליקציה ויהיה קיים צורך לשימוש ב XML נעבוד לפי הכללים הבאים:
כל שם יכול לשמש אותנו, לא קיימים שמות שמורים, אך הרעיון הוא לתת שמות שמתארים פעולה.
דוגמא:
<first_name>, <last_name>.
יש להמנע משימוש ב"-" וב".".
שמות אלמנטים יכולים להיות כל מה שרק נרצה אך שוב נרצה לשמור על פשטות ולוגיקת פיתוח.
לדוגמא שמות אלמנטים יהיו:
<book_title> not like this: <the_title_of_the_book>.

מסמכי XML מגיבים לשימוש ב DB לפיכך שימוש נכון בלוגיקה שמית הוא קריטי, לא ממציאים שמות!

22.24. אלמנטים VS מאפיינים

בחלק זה נעשה השוואה בין שימוש באלמנטים לשימוש במאפיינים. ננסה לאמוד את השימוש הנכון בעת הצורך. אנו יודעים כי נתונים יכולים להיות מאוכלסים אצל ה"ילד" או בתוך מאפיין.

הבא נבחן את הדוגמאות הבאות:

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

בדוגמא הראשונה, sex הנו מאפיין. בדוגמא השנייה sex הנו "ילד". שתי דוגמאות אלו ייתנו לנו את אותו תוצר!
לא קיימים חוקים מתי נשתמש במאפיינים ומתי נשתמש ב"ילד" כאלמנט. מניסיוני אני יכול לומר כי מאפיינים יכולים להיות מטופלים יפה ב HTML אך ב XML עלינו להמנע מהם ככל שניתן. הצעתי היא שימוש יותר באלמנטים "ילדים".

22.25. אחסון נתונים באלמנט "ילד"

כפי שאמרתי לעיל, הדרך העדיפה היא אכסון נתונים באלמנט "ילד". הבא נבחן את הדוגמאות הבאות:

דוגמא ראשונה אנו משתמשים במאפיין שהנו תאריך:

```
<note date="12/11/2002">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

בדוגמא השנייה אלמנט התאריך שימושי שוב:

```
<note>
<date>12/11/2002</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

דוגמא שלישית והמוצעת על ידי הוא השימוש הבא:

```
<note>
<date>
  <day>12</day>
  <month>11</month>
  <year>2002</year>
</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

22.26. חסרונות בשימוש מאפיינים

1. מאפיינים אינם יכולים לכלול ערכים מרובים.
2. מאפיינים אינם גמישים לשינויים.
3. מאפיינים לא יכולים לתאר מבנים.
4. קשה יותר לשלוט במאפיינים ע"י תוכנית.
5. קשה לבצע בדיקות של מאפיינים מול DTD אשר מאפשר לנו שימוש באלמנטים של מסמך XML.

22.27. דפדפנים שימושיים

- Internet Explorer 6
- Firefox 1.0.2
- Mozilla 1.7.8
- Opera 8
- Netscape 6

23. ולידציה

אחד מהיתרונות המהותיים ב-XML הינה הצבת תנאים מוקדמים על מנת לקרוא ואו לקבל נתונים.
XML מאפשר לנו לומר כי כל אלמנט Order חייב לכלול בדיוק אלמנט customer אחד וכך שלכל customer ישנו אלמנט id המכיל שם שמוגדר ב-XML, וכך שלכל אלמנט ShipTo חייב להיות אלמנט אחד או יותר של streets, city, state ו- zip.
בדיקת מסמך ה-XML כנגד הרשימה הזו נקראת אימות. אימות אינו הכרחי אך רצוי.
קיימת שפה נוספת בה ניתן להגדיר תנאים כאלה ואחרים, שפה זו מיוחסת כ-Schema Language והמסמך המכיל את האילוצים נקרא schemas.
שפת ה- DTD (Document Type Definition) הינה השפה היחידה הבנויה לתוך XML parsers והיא כחלק מסטנדרט XML.
אולם, קיימות סכימות נוספות אשר יכולות להיות שימושיות.

23.1. שימוש ב- DTD

DTD מתמקד במבנה האלמנטים של המסמך. אנו מגדירים את האלמנטים שעל המסמך לכלול כמו גם איזה סדר האלמנטים יהיו ומה הם מאפייניהם.
הגדרת אלמנט:
ע"מ שנוכל להגדיר בהתאם ל-DTD כל אלמנט חייב לכלול את ההגדרה של מילת הייחוס ELEMENT.
לדוגמא, הגדרת אלמנט האומר כי האלמנטים Name מכילים PCDATA# קרי זה טקסט ולא אלמנט מסוג בן.

```
<!ELEMENT Name (#PCDATA)>
```

לאלמנטים אשר יכולים להיות בנים מוגדרים ע"י רשימת שמות של ילדיהם בסדר המופרד ע"י פסיק.
לדוגמא, ההגדרה הבאה לאלמנט אומרת כי אלמנט order מכיל אלמנט customer, אלמנט product, אלמנט Subtotal, אלמנט Tax, אלמנט Shipping ואלמנט Total ומכאן שניתן להציג זאת באופן הבא:

```
<!ELEMENT Order (Customer, Product, Subtotal, Tax, Shipping, Total)>
```

הרשימה הנמצאת בסוגריים והמעידה על תכולתו של אלמנט מסוים נקראת Content Model.
נוכל לשים **סימן שאלה** אחרי שם אלמנט על מנת להצביע כי האלמנט הינו **אופציונלי** משמע 0 או 1 יקרה לאלמנט זה.
נוכל לשים כוכבית לאחר שם האלמנט על מנת להצביע כי 0 או יותר מופעים יכולים להיות מאותו אלמנט.
נוכל לשים את סימן הפלוס לאחר שם האלמנט על מנת להצביע כי 1 או יותר מופעים של האלמנט חייבים לקרות במיקום.
לדוגמא, הגדרת האלמנט הבאה אומרת כי אלמנט ShipTo חייב לכלול 0 או 1 מופעים של אלמנטים GiftReceipt, חייב להיות מופע 1 או יותר של אלמנט Street וחייב להיות בדיוק מופע 1 בלבד של אלמנט City, State, Zip

```
<!ELEMENT ShipTo (GiftRecipient?, Street+, City, State, Zip)>
```

אפשרי להשתמש במפריד שהנו קו ורטיקאלי במקום שימוש בפסיק על מנת להצביע כי אלמנט אחד או אחר יכולים להופיע. ניתן לקבץ קבוצה של אלמנטים בסוגריים על מנת להצביע כי עלינו להתייחס לקבוצה כיחידה אחת. אפשרי להשתמש ב *,?,+ לקבוצה על מנת להצביע כי 0 או יותר, 0 או 1, או 1 או יותר מהקבוצות הללו אמורות להופיע.
לבסוף, ניתן להחליף את כלל ה- content model במילת הייחוס EMPTY על מנת להצביע כי האלמנט לא מכיל תוכן כלל.

DTD מגדיר לנו איזה מאפיינים יכולים להופיע ובאילו אלמנטים. כל מאפיין מוגדר ב- ALLLIST אשר מגדיר את הבא:

- את האלמנט אליו המאפיין שייך.
- את שם המאפיין.
- את סוג המאפיין.
- את ערך ברירת המחדל של המאפיין.

לדוגמא, ההצהרה הבאה אומרת כי לאלמנט Customer חייב להיות מאפיין id יחד עם סוג id:

```
<!ATTLIST Customer id ID #REQUIRED>
```

DTD מגדיר 10 סוגים של מאפיינים:

1. CDATA - כל string של טקסט. ערך ברירת המחדל למאפיינים לא מוגדרים הינו מסמך שאינו זמין.
2. NMTOKEN - string אשר נוצר עם אחד או יותר שמות תווים חוקיים ב-XML . שם יכול להתחיל עם מספר.
3. NMTOKENS- מרווח המפריד בין שמות
4. ID - שם ב-XML אשר מצביע על ייחודיות של מאפיין.
5. IDREF - שם ב-XML המשתמש בערכו של מאפיין ה-ID על מספר אלמנטים במסמך.
6. IDREFS - מרווח המפריד שמות XML אשר משתמשים במאפיין ID במיקום כלשהו במסמך.
7. ENTITY - שם היישות בהגדרת DTD
8. ENTITIES - מרווח של שמות יישויות מוגדרות ב- DTD.
9. NOTATION - שם ההערה המוגדר ב- DTD
10. Enumeration - רשימה של ערכים חוקיים עבור מאפיין המופרד ע"י קווים ורטיקאליים.

DTD מאפשר 4 ערכי ברירת מחדל למאפיינים:

1. #REQUIRED - לכל אלמנט במסמך המתבקש ישנו ערך.
2. #IMPLIED - כל אלמנט במסמך המתבקש יכול להציג ערך ברירת מחדל. (לא חובה)
3. #FIXED "*value*" - המאפיין תמיד יכלול ערך קבוע ויוצג במרכאות בודדות או כפולות.
4. "*value*" - כברירת מחדל למאפיין ישנו ערך המוגדר ב-DTD ע"י מרכאות בודדות או כפולות.

להלן דוגמא המציגה מסמך Order תוך שימוש בתגים שהוצגו:

```
<!ELEMENT Order (Customer, Product+, Subtotal, Tax, Shipping, Total)>
<!ELEMENT Customer (#PCDATA)>
<!ATTLIST Customer id ID #REQUIRED>
<!ELEMENT Product (Name, SKU, Quantity, Price, Discount?,
                   ShipTo, GiftMessage?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT SKU (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ATTLIST Price currency (USD | CAN | GBP) #REQUIRED>
<!ELEMENT Discount (#PCDATA)>
<!ELEMENT ShipTo (GiftRecipient?, Street+, City, State, Zip)>
<!ELEMENT GiftRecipient (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City   (#PCDATA)>
<!ELEMENT State  (#PCDATA)>
<!ELEMENT Zip    (#PCDATA)>
<!ELEMENT GiftMessage (#PCDATA)>
<!ELEMENT Subtotal (#PCDATA)>
<!ATTLIST Subtotal currency (USD | CAN | GBP) #REQUIRED>
<!ELEMENT Tax (#PCDATA)>
<!ATTLIST Tax currency (USD | CAN | GBP) #REQUIRED
              rate CDATA "0.0"
>

<!ELEMENT Shipping (#PCDATA)>
<!ATTLIST Shipping currency (USD | CAN | GBP) #REQUIRED
                   method   (USPS | UPS | Overnight) "UPS">
<!ELEMENT Total (#PCDATA)>
<!ATTLIST Total currency (USD | CAN | GBP) #REQUIRED>
```

Document Type Declarations .23.2

מסמכים XML כוללים את ההגדרה של DTD.

```
<!DOCTYPE Order SYSTEM "order.dtd">
```

את ההגדרה יש למקם באופן הבא:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Order SYSTEM "order.dtd">
<Order>
   ...
```

23.3. סכימות

סכימות באות לתת מענה על מספר אלמנטים אשר לא קיימים ב-DTD. ראשית, סכימות נכתבות בתחביר XML תוך שימוש בתגיות, אלמנטים ומאפיינים. שנית, סכימות מבוססות על Namespace.
בנוסף, ניתן לייחס סוג נתונים לסכימות כגון תאריך, מספר, וכו לאלמנטים ולבצע ולידציה.
בדוגמא להלן נראה את הסכימה למסמך Order. מסמך Order.dtd משתמש בהצהרה של ELEMENT הנקראת Order.xsd תוך שימוש ב- xsd:element.
בנוסף order.dtd מבצע שימוש בהצהרת ALLLIST order.xsd משתמש באלמנט- xsd:attribute
בנוסף סכימה יכולה להכיל תנאים, לדוגמא אם אלמנטים כגון Tax, Shipping יכילו נתונים שאינם מספריים אזי כי תהיה טעות אשר ניתן לראותה, DTD לא יכול לגלות טעות מסוג זה.

להלן הסכימה:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Customer">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:ID"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
         </xsd:element>
        <xsd:element name="Product" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name"     type="xsd:string"/>
              <xsd:element name="SKU"
               type="xsd:positiveInteger"/>
              <xsd:element name="Quantity"
               type="xsd:positiveInteger"/>
              <xsd:element name="Price"    type="MoneyType"/>
              <xsd:element name="Discount" type="xsd:decimal"
                           minOccurs="0"/>
              <xsd:element name="ShipTo">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="GiftRecipient"
                     type="xsd:string"
                     minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:element name="Street"
                     type="xsd:string"/>
                    <xsd:element name="City" type="xsd:string"/>
                    <xsd:element name="State"
                     type="xsd:string"/>
                    <xsd:element name="Zip" type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="GiftMessage" type="xsd:string"
                           minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
```

```
        <xsd:element name="Subtotal" type="MoneyType"/>
        <xsd:element name="Tax">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="MoneyType">
                <xsd:attribute name="rate" type="xsd:decimal"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Shipping">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="MoneyType">
                <xsd:attribute name="method" type="xsd:string"/>
              </xsd:extension>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Total" type="MoneyType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="MoneyType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

</xsd:schema>
```

24. פרוטוקולים ב- XML

24.1. פרוטוקול RSS

פרוטוקול זה משמש אותנו על מנת להחליף headlines יחד עם abstracts בין אתרי חדשות שונים. כיום הוא קיים בגרסה 0.9.1 ובגרסה 1.0

24.2. פרוטוקול XML-RPC

פרוטוקול זה תומך ב- remote procedure calls על גבי האינטרנט על ידי העברת שמות הפונקציות והארגומנטים המוכלים במסמך ה-XML והעוברים על גבי HTTP.

24.3. פרוטוקול SOAP

בזמן שפרוטוקול XML-RPC מבצע שימוש אך ורק באלמנטים פרוטוקול SOAP מוסיף מאפיינים יחד עם namespaces.

24.4. XML as a Message Format

אחד מהעקרונות השימושיים ביותר ב- XML הינה החלפת נתונים בין מערכות הטרוגניות. בהינתן כל אוסף של נתונים נוכל כמובן לייצר קבצי XML. ומכאן ניתן לומר כי XML מגשר לנו על הקושי של העברת נתונים בין מערכות שונות טכנולוגית.

Table 2.1. Primitive Data Types defined in the W3C XML Schema Language

Type	Meaning
`xsd:string`	The schema equivalent of #PCDATA, any string of Unicode characters that may appear in an XML document
`xsd:boolean`	true, false, 1, 0
`xsd:decimal`	A decimal number such as 44.145629 or -0.32, with an arbitrary size and precision; similar to the `java.math.BigDecimal` class
`xsd:float`	The 4-byte IEEE-754 floating point number which best approximates the specified decimal string, same as Java's `float` type
`xsd:double`	The 8-byte IEEE-754 floating point number which best approximates the specified decimal string, same as Java's `double` type
`xsd:integer`	An integer of arbitrary size, similar to the `java.math.BigInteger` class
`xsd:nonPositiveInteger`	An integer less than or equal to zero
`xsd:negativeInteger`	An integer strictly less than zero
`xsd:nonNegativeInteger`	An integer greater than or equal to zero
`xsd:long`	An integer between -9223372036854775808 and +9223372036854775807 inclusive; equivalent to Java's `long` primitive data type
`xsd:int`	An integer between -2147483648 and 2147483647 inclusive; equivalent to Java's `int` primitive data type
`xsd:short`	An integer between -32768 and 32767 inclusive;

Type	Meaning
	equivalent to Java's `short` primitive data type
`xsd:byte`	An integer between -128 and 127 inclusive; equivalent to Java's `byte` primitive data type
`xsd:unsignedLong`	An integer between 0 and 18446744073709551615.
`xsd:unsignedInt`	An integer between 0 and 4294967295
`xsd:unsignedShort`	An integer between 0 and 65535
`xsd:unsignedByte`	An integer between 0 and 255
`xsd:positiveInteger`	An integer strictly greater than zero
`xsd:duration`	A length of time given in the ISO 8601 extended format: `PnYnMnDTnHnMnS`. The number of seconds can be a decimal or an integer. All the other values must be non-negative integers. For example, `P1Y2M3DT4H5M6.7S` is one year, two months, three days, four hours, five minutes, and 6.7 seconds.
`xsd:dateTime`	A particular moment of time on a particular day up to an arbitrary fraction of a second in the ISO 8601 format: *CCYY-MM-DD*T*hh*:*mm*:*ss*. This can be suffixed with a `Z` to indicate coordinated universal time (UTC) or an offset from UTC. For example, Neil Armstrong set foot on the moon at `1969-07-20T21:28:00-06:00` by the clock in Houston mission control which is also known as `1969-07-21T02:28:00Z`
`xsd:time`	A certain time of day on no particular day in the ISO 8601 format: *hh*:*mm*:*ss.sss*. A time zone specified as an offset from UTC is optional. For example, on most days I wake up around `07:00:00.000-05:00` and go to bed around `23:30:00.000-05:00`.
`xsd:date`	A particular date in history given in ISO 8601 format: *YYYYMMDD*; e.g. `20010706` or `19690920`.
`xsd:gYearMonth`	A certain month in a certain year; e.g. `2001-12` or `1999-03`.
`xsd:gYear`	A year in the Gregorian calendar ranging from 0001 to 2001 to 9999, 10000, 10001 and beyond. Earlier dates can be represented as -0001, -0002, -0003, and so forth back to the Big Bang. There is no year zero, however.
`xsd:gMonthDay`	A specific day of a specific month in no particular year in the form `--02-28`. For example, Christmas comes on `--12-25`.
`xsd:gDay`	A particular day of no particular month in the form ---01, ---02, ---03, through ---31
`xsd:gMonth`	A particular month in no particular year in the form --01--, --02--, --03--, through --12--
`xsd:hexBinary`	Hexadecimal encoded binary data; each byte of the data

Type	Meaning
	is replaced by the two hexadecimal digits that represent its unsigned value
`xsd:base64Binary`	Base-64 encoded binary data
`xsd:anyURI`	An absolute or relative URL or a URN
`xsd:QName`	An optionally prefixed XML name such as `SOAP-ENV:Body` or `Body`. Unprefixed names must be in the default namespace.
`xsd:NOTATION`	The name of a notation declared in the current schema
`xsd:normalizedString`	A string in which carriage returns (\r), line feeds (\n) and tab (\t) characters should be treated the same as spaces
`xsd:token`	A string in which all runs of white space should be treated the same as a single space
`xsd:language`	An RFC 1766 language identifier such as en, fr-CA, or i-klingon
`xsd:NMTOKEN`	An XML name token
`xsd:NMTOKENS`	A white space separated list of XML name tokens
`xsd:Name`	An XML name
`xsd:NCName`	An XML name that does not contain any colons; that is, an unprefixed name
`xsd:ID`	An NCName which is unique among other things of ID type in the same document
`xsd:IDREF`	An NCName used as an ID somewhere in the document
`xsd:IDREFS`	A whitespace separated list of IDREFs
`xsd:ENTITY`	An NCName that has been declared as an unparsed entity in the document's DTD
`xsd:ENTITIES`	A white space separated list of ENTITY names

הדוגמא להלן מראה שימוש בחלק מהתגיות המוצגות בטבלה בחלקים שונים במסמך. הבא נבחן את הקוד:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Customer id="c32" xsi:type="xsd:string">Chez Fred</Customer>
  <Product>
    <Name xsi:type="xsd:string">Birdsong Clock</Name>
    <SKU xsi:type="xsd:string">244</SKU>
    <Quantity xsi:type="xsd:positiveInteger">12</Quantity>
    <Price currency="USD"  xsi:type="xsd:decimal">21.95</Price>
    <ShipTo>
      <Street xsi:type="xsd:string">135 Airline Highway</Street>
      <City xsi:type="xsd:string">Narragansett</City>
      <State xsi:type="xsd:NMTOKEN">RI</State>
      <Zip xsi:type="xsd:string">02882</Zip>
    </ShipTo>
  </Product>
  <Product>
    <Name xsi:type="xsd:string">Brass Ship's Bell</Name>
    <SKU xsi:type="xsd:string">258</SKU>
    <Quantity xsi:type="xsd:positiveInteger">1</Quantity>
    <Price currency="USD" xsi:type="xsd:decimal">144.95</Price>
    <Discount xsi:type="xsd:decimal">.10</Discount>
    <ShipTo>
      <GiftRecipient xsi:type="xsd:string">
        Samuel Johnson
      </GiftRecipient>
     <Street xsi:type="xsd:string">271 Old Homestead Way</Street>
      <City xsi:type="xsd:string">Woonsocket</City>
      <State xsi:type="xsd:NMTOKEN">RI</State>
      <Zip xsi:type="xsd:string">02895</Zip>
    </ShipTo>
    <GiftMessage xsi:type="xsd:string">
      Happy Father's Day to a great Dad!

      Love,
      Sam and Beatrice
    </GiftMessage>
  </Product>
  <Subtotal currency='USD' xsi:type="xsd:decimal">
    393.85
  </Subtotal>
  <Tax rate="7.0"
       currency='USD' xsi:type="xsd:decimal">28.20</Tax>
  <Shipping method="USPS" currency='USD'
            xsi:type="xsd:decimal">8.95</Shipping>
  <Total currency='USD' xsi:type="xsd:decimal">431.00</Total>
</Order>
```

מסמך יכול לכלול סכימה על מנת להצביע על סוגו.

באופן מאוד פשוט ניתן לכתוב תוכניות המבוססות על קבצי XML.
בשפת Java המימוש הוא פשוט יותר מאחר וישנה תמיכה מלאה ב-UNICODE. אין צורך לדעת API מיוחד כגון DOM או SAX או JDOM. כל מה שעלינו לדעת הוא כיצד לבצע שימוש נכון בפונקצית ההדפסה System.out.println().
באם נרצה לאחסן את הקובץ XML שלנו נוכל להשתמש במחלקת FileOutputStream.
בחלק זה נפתח את קוד פיבונצ'י ונכניס אותו למסמך XML.
המפתח העיקרי שעלינו לזכור הינו שנתונים מגיעים ממקורות שונים ומאוכלסים בקובץ XML. נבצע שימוש במחלקות כגון OutputStreamWriter, String, HTTPServlet.
קוד פיבונצ'י פותח בשנת 1200 לספירה ובא לבחון את השאלה כמה זוגות ארנבים יכולים להיוולד מזוג מסוים. על מנת לתת מענה לשאלה, הנחת היסוד באלגוריתם הינה שמשך ההריון הוא חודש.
הארנבות אינן מענייני דיומא שלנו אלא המתמטיקה. כל מספר הוא מסוג int אשר נוצר כתוצאה של חיבור שני הזוגות הקודמים לו. שני המספרים הראשונים הינם 1.
ולגבי היישום – מאוד פשוט לחשב את נוסחת פיבונצ'י ע"י לולאת for פשוטה באופן הבא:

```
int low = 1;
    int high = 1;
    for (int i = 1; i <= 40; i++) {
      System.out.println(low);
      int temp = high;
      high = high+low;
      low = temp;
    }
```

מאחר ובאופן טבעי מספרי פיבונצ'י גודלים באופן מהיר אזי כי עלינו להרחיב את גבולות סוג המספר קרי לבצע שימוש ב-int גדול יותר ולצורך כך נשתמש ב- BigInt

להלן הקוד:

```
import java.math.BigInteger;

public class FibonacciNumbers {

  public static void main(String[] args) {

    BigInteger low  = BigInteger.ONE;
    BigInteger high = BigInteger.ONE;
    for (int i = 1; i <= 10; i++) {
      System.out.println(low);
      BigInteger temp = high;
      high = high.add(low);
      low = temp;
    }

  }

}
```

להלן התוצאה

```
1
1
2
3
5
8
13
21
34
55
```

24.5. כתיבת XML

נניח כי נרצה לייצא את הנתונים שהתקבלו בפורמט XML. להלן התוצאה:

```
<?xml version="1.0"?>
<Fibonacci_Numbers>
  <fibonacci>1</fibonacci>
  <fibonacci>1</fibonacci>
  <fibonacci>2</fibonacci>
  <fibonacci>3</fibonacci>
  <fibonacci>5</fibonacci>
  <fibonacci>8</fibonacci>
  <fibonacci>13</fibonacci>
  <fibonacci>21</fibonacci>
  <fibonacci>34</fibonacci>
  <fibonacci>55</fibonacci>
</Fibonacci_Numbers>
```

להלן התוכנית אשר תבצע את קוד פיבונצ'י כמסמך XML:

```
import java.math.BigInteger;


public class FibonacciXML {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      System.out.println("<?xml version=\"1.0\"?>");
      System.out.println("<Fibonacci_Numbers>");
      for (int i = 0; i < 10; i++) {
        System.out.print("  <fibonacci>");
        System.out.print(low);
        System.out.println("</fibonacci>");
        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      System.out.println("</Fibonacci_Numbers>");

  }

}
```

נוכל כמובן לבצע שימוש בשמות קבועים עבור שמות האלמנטים באופן הבא:

```
import java.math.BigInteger;


public class FibonacciConstants {

  public final static String rootElementName
   = "Fibonacci_Numbers";
  public final static String fibonacciElementName = "fibonacci";
  public final static String xmlDeclaration
   = "<?xml version=\"1.0\"?>";

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      System.out.println(xmlDeclaration);
      System.out.println("<" + rootElementName + ">");
      for (int i = 0; i < 10; i++) {
        System.out.print("  <" + fibonacciElementName +">");
        System.out.print(low);
        System.out.println("</" + fibonacciElementName +">");
        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      System.out.println("</" + rootElementName + ">");

  }

}
```

כעת נניח כי אנו רוצים להוסיף מאפיינים לאלמנטים במסמך.
לדוגמא נרצה להוסיף את האלמנט Fibonacci לכל מאפיין index אשר מגדיר מהו המספר.
להלן הקוד:

```
import java.math.BigInteger;


public class FibonacciAttributes {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      System.out.println("<?xml version=\"1.0\"?>");
      System.out.println("<Fibonacci_Numbers>");
      for (int i = 1; i <= 10; i++) {
        System.out.print("  <fibonacci index=\"" + i + "\">");
        System.out.print(low);
        System.out.println("</fibonacci>");
        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      System.out.println("</Fibonacci_Numbers>");

  }

}
```

נוכל כמובן לבצע שימוש במרכאות " " כפולות במקום "\ באופן הבא:

```
<?xml version="1.0"?>
<Fibonacci_Numbers>
  <fibonacci index="1">1</fibonacci>
  <fibonacci index="2">1</fibonacci>
  <fibonacci index="3">2</fibonacci>
  <fibonacci index="4">3</fibonacci>
  <fibonacci index="5">5</fibonacci>
  <fibonacci index="6">8</fibonacci>
  <fibonacci index="7">13</fibonacci>
  <fibonacci index="8">21</fibonacci>
  <fibonacci index="9">34</fibonacci>
  <fibonacci index="10">55</fibonacci>
</Fibonacci_Numbers>
```

עד עתה תוכניות Java יצרו לנו מסמך XML שלא היה מאומת. על מנת לאמת מסמך XML נצטרך לבצע DTD – להלן הקוד:

```
import java.math.BigInteger;


public class ValidFibonacci {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      System.out.println("<?xml version=\"1.0\"?>");
      System.out.println("<!DOCTYPE Fibonacci_Numbers [");
      System.out.println(
       "  <!ELEMENT Fibonacci_Numbers (fibonacci)*>");
      System.out.println("  <!ELEMENT fibonacci (#PCDATA)>");
      System.out.println(
       "  <!ATTLIST fibonacci index NMTOKEN #REQUIRED>");
      System.out.println("]>");
      System.out.println("<Fibonacci_Numbers>");
      for (int i = 0; i < 10; i++) {
        System.out.print("  <fibonacci index=\"" + i + "\">");
        System.out.print(low);
        System.out.println("</fibonacci>");
        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      System.out.println("</Fibonacci_Numbers>");

  }

}
```

להלן התוצאה הכוללת את סוג המסמך:

```
<?xml version="1.0"?>
<!DOCTYPE Fibonacci_Numbers [
  <!ELEMENT Fibonacci_Numbers (fibonacci)*>
  <!ELEMENT fibonacci (#PCDATA)>
  <!ATTLIST fibonacci index NMTOKEN #REQUIRED>
]>
<Fibonacci_Numbers>
  <fibonacci index="0">1</fibonacci>
  <fibonacci index="1">1</fibonacci>
  <fibonacci index="2">2</fibonacci>
  <fibonacci index="3">3</fibonacci>
  <fibonacci index="4">5</fibonacci>
  <fibonacci index="5">8</fibonacci>
  <fibonacci index="6">13</fibonacci>
  <fibonacci index="7">21</fibonacci>
  <fibonacci index="8">34</fibonacci>
  <fibonacci index="9">55</fibonacci>
</Fibonacci_Numbers>
```

צירוף סכימה למסמך היא פעולה פשוטה כל שעלינו לעשות הוא למקם את xmlns:xsi ואת xsi:noNamespaceSchemaLocation בנתיב האלמנט.

נניח כי במקום להשתמש במספרי פיבונצ'י נרצה להשתמש במילון המינוחים הנתון של XML. המינוח הינו Name Space. לכלל שמות האלמנטים ישנו תחביר לדוגמא xmlns:mathml ובנתיב האלמנט תחביר mathml מתחבר ל- http://www.w3.org/1998/Math/MathML.
לכל מספר פיבונצ'י הקיים ב- mathml:mrow מחולק לאלמנט mathml:mi והאלמנט mathml:mo והאלמנט mathml:mn.

להלן הקוד:

```
<?xml version="1.0"?>
<mathml:math xmlns:mathml="http://www.w3.org/1998/Math/MathML">
  <mathml:mrow>
    <mathml:mi>f(1)</mathml:mi>
    <mathml:mo>=</mathml:mo>
    <mathml:mn>1</mathml:mn>
  </mathml:mrow>
  <mathml:mrow>
    <mathml:mi>f(2)</mathml:mi>
    <mathml:mo>=</mathml:mo>
    <mathml:mn>1</mathml:mn>
  </mathml:mrow>
  <mathml:mrow>
    <mathml:mi>f(3)</mathml:mi>
    <mathml:mo>=</mathml:mo>
    <mathml:mn>2</mathml:mn>
  </mathml:mrow>
  <mathml:mrow>
    <mathml:mi>f(4)</mathml:mi>
    <mathml:mo>=</mathml:mo>
    <mathml:mn>3</mathml:mn>
  </mathml:mrow>
</mathml:math>
```

להלן תוכנית אשר מייצרת מסמך MathML

```
import java.math.BigInteger;


public class MathMLFibonacci {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      System.out.println("<?xml version=\"1.0\"?>");
      System.out.println(
        "<mathml:math "
        + "xmlns:mathml=\"http://www.w3.org/1998/Math/MathML\">"
      );
      for (int i = 1; i <= 10; i++) {
        System.out.println("  <mathml:mrow>");
        System.out.println("    <mathml:mi>f(" + i
         + ")</mathml:mi>");
        System.out.println("    <mathml:mo>=</mathml:mo>");
        System.out.println("    <mathml:mn>" + low
         + "</mathml:mn>");
        System.out.println("  </mathml:mrow>");
        BigInteger temp = high;
        high = high.add(low);
        low = temp;
      }
      System.out.println("</mathml:math>");

  }

}
```

Output Streams, Writers, and Encodings .24.6

לעיתים לא נרצה לייצר קובץ XML בהדפסה החוצה אלא נרצה לכתוב את הקובץ לרשת נתונה. מסמכי XML הינם מבוססי טקסט והטקסט עצמו בנוי מתווים מסוג UNICODE.

כאשר אנו כותבים את התווים לתוך stream עלינו לבחור את התו המגדיר את ההמרה כגון UTF-8 או UTF-16. הדרך בה Java מבצעת המרה היא תוך שינוי של OutputStreamWriter ל-OutputStream.

נניח כי קיים לנו קובץ בשם fibonacci.xml. ראשית נרצה לפתוח אותו תוך שימוש ב-FileOutputStream באופן הבא:

OutputStream fout = new FileOutputStream("fibonacci.xml");

ואז נוכל לחבר את ה-BufferedOutputStream יחד עם ה-OutputStreamWriter ונעביר את קוד ההמרה בקונסטרטור ()OutputStreamWriter.

לדוגמא:

נניח כי בחרנו ISO-8859-1, Latin-1 encoding ולפיכך הוא מבצע שימוש בשם הקיים ב-Java שהנו "8859_1" ומכאן נוכל לומר כי:

OutputStreamWriter out = new OutputStreamWriter(bout, "8859_1");

לבסוף נכתוב את התוצר ב- `OutputStreamWriter` ונוודא כי אנו מבצעים שימוש ב-ENCODING הנכון.

להלן קוד הכותב קובץ XML:

```
import java.math.BigInteger;
import java.io.*;


public class FibonacciFile {

  public static void main(String[] args) {

      BigInteger low  = BigInteger.ONE;
      BigInteger high = BigInteger.ONE;

      try {
        OutputStream fout= new FileOutputStream("fibonacci.xml");
        OutputStream bout= new BufferedOutputStream(fout);
        OutputStreamWriter out
         = new OutputStreamWriter(bout, "8859_1");

        out.write("<?xml version=\"1.0\" ");
        out.write("encoding=\"ISO-8859-1\"?>\r\n");
        out.write("<Fibonacci_Numbers>\r\n");
        for (int i = 1; i <= 10; i++) {
          out.write("  <fibonacci index=\"" + i + "\">");
          out.write(low.toString());
          out.write("</fibonacci>\r\n");
          BigInteger temp = high;
          high = high.add(low);
          low = temp;
        }
        out.write("</Fibonacci_Numbers>\r\n");

        out.flush();  // Don't forget to flush!
        out.close();
      }
```

```
      catch (UnsupportedEncodingException e) {
        System.out.println(
         "This VM does not support the Latin-1 character set."
        );
      }
      catch (IOException e) {
        System.out.println(e.getMessage());
      }

  }

}
```

טבלת קידוד:

XML Name	Java Name	First supported in Java	Scripts and Languages
ISO-8859-1	8859_1	1.1	Latin-1: ASCII plus the accented characters needed for most Western European languages including Albanian, Basque, Breton, Catalan, Cornish, Danish, Dutch, English, Estonian, Faroese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish, Italian, Latin, Luxemburgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Sorbian, Spanish, and Swedish as well as many non-European languages written in the Latin alphabet such as Swahili and Malaysian
ISO-8859-2	8859_2	1.1	Latin-2: ASCII plus the accented characters needed for most Central European languages including Albanian, Croatian, Czech, Finnish, German, Hungarian, Latin, Polish, Romanian, Slovak, Slovenian, and Sorbian
ISO-8859-3	8859_3	1.1	Latin-3: ASCII plus the accented characters needed for most Southern European languages including English, Esperanto, Finnish, French, German, Italian, Latin, Maltese, Portuguese, and Turkish
ISO-8859-4	8859_4	1.1	Latin-4: ASCII plus the accented characters needed for most Northern European languages including Danish, English, Estonian, Finnish, German, Greenlandic, Latin, Latvian, Lithuanian, Norwegian, S□mi, Slovenian, and Swedish

ISO-8859-5	8859_5	1.1	ASCII plus Cyrillic
ISO-8859-6	8859_6	1.1	ASCII plus Arabic
ISO-8859-7	8859_7	1.1	ASCII plus Greek
ISO-8859-8	8859_8	1.1	ASCII plus Hebrew
ISO-8859-9	8859_9	1.1	Latin-5: same as Latin-1 except the Turkish letters Ğ, ğ, İ, ı, Ş, and ş take the place of the Icelandic letters þ, Þ, ý, Ý, Ð, and ð
ISO-8859-13	ISO8859_13	1.3	Latin-7: ASCII plus the accented characters needed for most Baltic languages including Latvian, Lithuanian, Estonian, and Finnish, as well as English, Danish, Swedish, German, Slovenian, and Norwegian.
ISO-8859-15	ISO8859_15_FDIS	1.2	Latin-9: same as Latin-1 but with the Euro sign € instead of the international currency symbol ¤. It also replaces the infrequently used symbol characters ¦, ¨, ´, ¸, ¼, ½, and ¾ with the infrequently used French and Finnish letters Š, š, Ž, ž, Œ, œ, and Ÿ.
UTF-8	UTF8	1.1	The default encoding of XML documents; each Unicode character is represented in between 1 and 4 bytes.
UTF-16	UnicodeBig or UnicodeLittle	1.2	An encoding of Unicode in which characters in the Basic Multilingual Plane are encoded in two bytes, and all other characters are encoded as two two-byte surrogates
ISO-10646-UCS-2	N/A	N/A	A straightforward encoding in which each Unicode character is represented as a two-byte integer; cannot represent characters outside the Basic Multilingual Plane
ISO-10646-UCS-4	N/A	N/A	A straightforward encoding in which each Unicode character is represented as a four-byte integer
ISO-2022-JP	JIS	1.1	Japanese
Shift_JIS	SJIS	1.1	Japanese
EUC-JP	EUCJIS	1.1	Japanese
US-ASCII	ASCII	1.2	English

GBK	GBK	1.1	Simplified Chinese
Big5	Big5	1.1	Traditional Chinese
ISO-2022-CN	ISO2022CN	1.1	Traditional Chinese
ISO-2022-KR	ISO2022KR	1.1	Korean

24.7. שימוש ב- XML-RPC

XML-RPC request הנו מסמך XML הנשלח על גבי רשת עם socket. על מנת לכתוב שירות מסוג זה עלינו לבקש מהמשתמש את הנתונים שברצונו להעביר, לעטוף אותם כקובץ XML, ולכתוב את ה-URL המצביע לשרת. התגובה מהשרת תגיע כקובץ XML בחזרה.

להלן קוד ה- XML:
מסמך זה מבקש את הערך 23 מקוד פיבונצ'י

```
<?xml version="1.0"?>
<methodCall>
  <methodName>calculateFibonacci</methodName>
  <params>
    <param>
      <value><int>23</int></value>
    </param>
  </params>
</methodCall>
```

הנה תגובת השרת:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>28657</double></value>
    </param>
  </params>
</methodResponse>
```

התגובה חוזרת אלינו כמשתנה מסוג double מאחר ומשתנים מסוג int מוגבלים בפרוטוקול XML-RPC זאת ועוד פרוטוקול זה לא נותן ערך מספרי מקסימלי לערכי double.
הלקוח צריך לקרוא את המספר שהוכנס ע"י המשתמש לעטוף את המספר כקובץ XML ולשלוח את הבקשה לשרת תוך שימוש ב- HTTP POST.
ב-Java הדרך הקלה ביותר להעביר מסמך הינה שימוש במחלקת java.net.HttpURLConnection

להלן קוד המראה כיצד לבצע התקשרות בין פרוטוקול XML-RPC תוך שימוש ב- URLConnection.
לצורך הרצה יש לבנות Servlet:

```
import java.net.*;
import java.io.*;


public class FibonacciXMLRPCClient {

  private static String defaultServer
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println(
       "Usage: java FibonacciXMLRPCClient index serverURL");
      return;
    }

    String index = args[0];

    String server;
    if (args.length <= 1) server = defaultServer;
```

```
    else server = args[1];

    try {
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();
      OutputStreamWriter wout = new OutputStreamWriter(out, "UTF-8");

      wout.write("<?xml version=\"1.0\"?>\r\n");
      wout.write("<methodCall>\r\n");
      wout.write(
       "  <methodName>calculateFibonacci</methodName>\r\n");
      wout.write("  <params>\r\n");
      wout.write("    <param>\r\n");
      wout.write("      <value><int>");
      wout.write(index);
      wout.write("</int></value>\r\n");
      wout.write("    </param>\r\n");
      wout.write("  </params>\r\n");
      wout.write("</methodCall>\r\n");

      wout.flush();
      out.close();

      InputStream in = connection.getInputStream();
      int c;
      while ((c = in.read()) != -1) System.out.write(c);
      System.out.println();
      in.close();
      out.close();
      connection.disconnect();
    }
    catch (IOException e) {
      System.err.println(e);
      e.printStackTrace();
    }

  }  // end main

}
```

24.8. שימוש בפרוטוקול SOAP

שימוש בפרוטוקול זה זהה באופי המימוש לשימוש בפרוטוקול XML-RPC כל שעלינו לעשות הוא לשנות את התחביר על מנת שיתאים להתחברות פרוטוקול SOAP.
להלן הבקשה מהשרת:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <SOAP-ENV:Body>
    <calculateFibonacci
      xmlns="http://namespaces.cafeconleche.org/xmljava/ch3/"
      type="xsi:positiveInteger">10</calculateFibonacci>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

תגובת השרת הינה עם רשימה של ערכי פיבונצ'י הקיימים בתוך המעטפת:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" />
  <SOAP-ENV:Body>
    <Fibonacci_Numbers
      xmlns="http://namespaces.cafeconleche.org/xmljava/ch3/">
      <fibonacci index="1">1</fibonacci>
      <fibonacci index="2">1</fibonacci>
      <fibonacci index="3">2</fibonacci>
      <fibonacci index="4">3</fibonacci>
      <fibonacci index="5">5</fibonacci>
      <fibonacci index="6">8</fibonacci>
      <fibonacci index="7">13</fibonacci>
      <fibonacci index="8">21</fibonacci>
      <fibonacci index="9">34</fibonacci>
      <fibonacci index="10">55</fibonacci>
    </Fibonacci_Numbers>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

אופי העבודה זהה לשימוש בפרוטוקול XML-RPC להלן הקוד המראה כיצד להתחבר לשרת תוך שימוש ב-SOAP יחד עם URLConnection:

```
import java.net.*;
import java.io.*;


public class FibonacciSOAPClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/SOAP";
  public final static String SOAP_ACTION
   = "http://www.example.com/fibonacci";

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println(
       "Usage: java FibonacciSOAPClient index URL");
      return;
    }
```

```
    String input = args[0];
    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;

      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      connection.setRequestProperty("SOAPAction", SOAP_ACTION);

      OutputStream out = connection.getOutputStream();
      Writer wout = new OutputStreamWriter(out);

      wout.write("<?xml version='1.0'?>\r\n");
      wout.write("<SOAP-ENV:Envelope ");
      wout.write("xmlns:SOAP-ENV=");
      wout.write(
        "'http://schemas.xmlsoap.org/soap/envelope/' "
      );
      wout.write("xmlns:xsi=");
      wout.write(
        "'http://www.w3.org/2001/XMLSchema-instance'>\r\n");
      wout.write("  <SOAP-ENV:Body>\r\n");
      wout.write("    <calculateFibonacci ");
      wout.write(
    "xmlns='http://namespaces.cafeconleche.org/xmljava/ch3/'\r\n"
      );
      wout.write("    type='xsi:positiveInteger'>" + input
       + "</calculateFibonacci>\r\n");
      wout.write("  </SOAP-ENV:Body>\r\n");
      wout.write("</SOAP-ENV:Envelope>\r\n");

      wout.flush();
      wout.close();

      InputStream in = connection.getInputStream();
      int c;
      while ((c = in.read()) != -1) System.out.write(c);
      in.close();

    }
    catch (IOException e) {
      System.err.println(e);
    }

  } // end main

} // end FibonacciSOAPClient
```

25. קריאת קובץ XML

קריאת קבצי XML הינה פעולה ישירה. ניתן לבצע שימוש ב- XML Parser על מנת לקרוא את המסמך. בנוסף ה- parser בודק את תוכן המסמך תוך שימוש ב- API.

25.1. InputStreams and Readers

הבא נבחן את תהליך קריאת הנתונים מתוך קובץ XML. נסתכל על קוד התגובה שהגיע מהשרת כאשר ביצענו שימוש ב- XML-RPC

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>28657</double></value>
    </param>
  </params>
</methodResponse>
```

המסמך קורא ומציג את כל התוכן. כעת נרצה לקרוא אך ורק את התוצאה שהינה 34.
מתוצאת השרת אנו יודעים כי נתיב האלמנט הינו methodResponse. אנו גם יודעים שהוא כולל פרמטר בודד והפרמטר כולל אלמנט value.
לדוגמא, פונקציה זו קוראת את הקלט מקובץ XML ומדפיסה אותו:

```
public printXML(InputStream xml) {

  int c;
  while ((c = xml.read()) != -1) System.out.write(c);

}
```

על מנת לבנות את המידע יש צורך לבצע עבודה נוספת. עלינו להחליט איזה מידע אנו רוצים.
בקוד הבא פונקצית readFibonacciXMLRPCResponse() מבצעת את אופי העבודה. ראשית ע"י קריאה למסמך XML לתוך StringBuffer לאחר מכן המרה של ה-buffer לטובת string ואז שימוש בפונקציות indexOf() ו-substring() על מנת לבנות את המידע.
הפונקציה הראשית מתחברת לשרת תוך שימוש במחלקות URLConnection ושולחת בקשת מסמך לשרת תוך שימוש במחלקות OutputStream ו-OutputStreamWriter ומעביר את ה-InputStream המכיל את תגובת מסמך ה-XML לפונקצית readFibonacciXMLRPCResponse().

```
import java.net.*;
import java.io.*;
import java.math.BigInteger;


public class FibonacciClient {

  static String defaultServer
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciClient number url"
      );
      return;
    }
```

```
    String server = defaultServer;
    if (args.length >= 2) server = args[1];

    try {
      // Connect to the server
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();
      Writer wout = new OutputStreamWriter(out);

      // Write the request
      wout.write("<?xml version=\"1.0\"?>\r\n");
      wout.write("<methodCall>\r\n");
      wout.write(
       "  <methodName>calculateFibonacci</methodName>\r\n");
      wout.write("  <params>\r\n");
      wout.write("    <param>\r\n");
      wout.write("      <value><int>" + args[0]
       + "</int></value>\r\n");
      wout.write("    </param>\r\n");
      wout.write("  </params>\r\n");
      wout.write("</methodCall>\r\n");

      wout.flush();
      wout.close();

      // Read the response
      InputStream in = connection.getInputStream();
      BigInteger result = readFibonacciXMLRPCResponse(in);
      System.out.println(result);

      in.close();
      connection.disconnect();
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
  private static BigInteger readFibonacciXMLRPCResponse(
   InputStream in) throws IOException, NumberFormatException,
   StringIndexOutOfBoundsException {

    StringBuffer sb = new StringBuffer();
    Reader reader = new InputStreamReader(in, "UTF-8");
    int c;
    while ((c = in.read()) != -1) sb.append((char) c);

    String document = sb.toString();
    String startTag = "<value><double>";
    String endTag = "</double></value>";
    int start = document.indexOf(startTag) + startTag.length();
    int end = document.indexOf(endTag);
    String result = document.substring(start, end);
    return new BigInteger(result);

  }
}
```

25.2. XML Parsers

ה-Parser הינו ספריית תכנות (בשפצ Java הוא מחלקה) אשר קורא את מסמך ה-XML ובודק האם המסמך בהתאם לסטנדרטים. בנוסף ה-Parser מבצע מספר פעולות:

1. מתרגם את המסמך ל-UNICODE
2. מרכיב את החלקים השונים לתוך יישויות.
3. פותר את בעיית התווים מבחינת התייחסות.
4. מאמת את המסמך כנגד סכימת ה-DTD

25.2.1. בחירת API לטובת שימוש ב-XML

ההחלטה החשובה ביותר שעלינו לקבל היא באיזה API עלינו להשתמש. לטובת השימוש בשפת Java ישנם 3 סוגים API עיקריים:

1. SAX – Simple API for XML
   a. הינו ה-API הטוב ביותר מבחינת סטנדרטים. ניתן לבצע עם ה-API כמעט כל פעולה אפשרית כולל אימות. SAX הינו API המונע ע"י Event Driven. המחלקות והממשקים ב-SAX ממדלים את הקלט להעברה וכך נוצר תהליך של קריאה. SAX מאפשר לנו לבצע קריאה בלבד.
2. DOM – Document Object Model
   a. זהו API די מורכב מבחינת מימוש אשר ממדל את ה-XML במבנה של עץ. שלא כמו SAX ה-DOM הינו API שניתן לבצע בו read-write.
3. JAXP - Java API for XML Processing
   a. סוג זה של API משלב את ה-DOM יחד עם SAX. זהו API סטנדרטי החל מגרסה 1.4.

25.3. מימוש SAX

שימוש ב-SAX מאפשר לנו לממש את XMLReader Interface. ה-Parser קורא את המסמך מהתחלה ועד הסוף. במודל SAX ה-Parser אומר לאפליקצית הלקוח מה הוא רואה מאחר והוא מבצע שימוש באובייקט ContentHandler.
אובייקט זה הינו interface לאפליקצית הלקוח המקבל את ההערות על תכולת המסמך. אפליקצית הלקוח תבצע אתחול למופע של ה-ContentHandler ויירשום אותו יחד עם ה- XMLReader.

בדוגמא הבאה SAX מבצע התקשרות יחד עם שירות XML-RPC. מסמך בקשה נשלח תוך שימוש בטכניקות פלט רגילות ואז התגובה מתבצעת ב-SAX.

```
import java.net.*;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;


public class FibonacciSAXClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciSAXClient number url"
      );
      return;
    }
```

```
    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      // Connect to the server
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();
      Writer wout = new OutputStreamWriter(out);

      // Transmit the request XML document
      wout.write("<?xml version=\"1.0\"?>\r\n");
      wout.write("<methodCall>\r\n");
      wout.write(
       "  <methodName>calculateFibonacci</methodName>\r\n");
      wout.write("  <params>\r\n");
      wout.write("    <param>\r\n");
      wout.write("      <value><int>" + args[0]
       + "</int></value>\r\n");
      wout.write("    </param>\r\n");
      wout.write("  </params>\r\n");
      wout.write("</methodCall>\r\n");

      wout.flush();
      wout.close();

      // Read the response XML document
      XMLReader parser = XMLReaderFactory.createXMLReader(
        "org.apache.xerces.parsers.SAXParser"
      );
      // There's a name conflict with java.net.ContentHandler
      // so we have to use the fully package qualified name.
      org.xml.sax.ContentHandler handler
       = new FibonacciHandler();
      parser.setContentHandler(handler);

      InputStream in = connection.getInputStream();
      InputSource source = new InputSource(in);
      parser.parse(source);
      System.out.println();

      in.close();
      connection.disconnect();
    }
    catch (Exception e) {
      System.err.println(e);
    }

  }

}
```

שימוש באובייקט ContentHandler עבור SAX:

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;


public class FibonacciHandler extends DefaultHandler {

  private boolean inDouble = false;

  public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) throws SAXException {

    if (localName.equals("double")) inDouble = true;

  }

  public void endElement(String namespaceURI, String localName,
   String qualifiedName) throws SAXException {

    if (localName.equals("double")) inDouble = false;

  }

  public void characters(char[] ch, int start, int length)
  throws SAXException {

    if (inDouble) {
      for (int i = start; i < start+length; i++) {
        System.out.print(ch[i]);
      }
    }

  }

}
```

25.4. שימוש ב-DOM

שימוש ב-DOM מאפשר לנו פעולות של כתיבה-קריאה ממסך XML תוך שימוש באובייקט Document.

בדוגמא הקוד להלן נבצע שימוש ב-DOM המתחבר ל- XML-RPC servlet. הבקשה נוצרת בצורה של מסמך DOM והתגובה מועברת כמסמך DOM.

```
import java.net.*;
import java.io.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.*;
import org.apache.xerces.parsers.*;
import org.apache.xml.serialize.*;
import org.xml.sax.InputSource;


public class FibonacciDOMClient {

  public final static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciDOMClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {

      // Build the request document
      DOMImplementation impl
       = DOMImplementationImpl.getDOMImplementation();

      Document request
       = impl.createDocument(null, "methodCall", null);

      Element methodCall = request.getDocumentElement();

      Element methodName = request.createElement("methodName");
      Text text = request.createTextNode("calculateFibonacci");
      methodName.appendChild(text);
      methodCall.appendChild(methodName);

      Element params = request.createElement("params");
      methodCall.appendChild(params);

      Element param = request.createElement("param");
      params.appendChild(param);

      Element value = request.createElement("value");
      param.appendChild(value);

      // Had to break the naming convention here because of a
      // conflict with the Java keyword int
      Element intElement = request.createElement("int");
      Text index = request.createTextNode(args[0]);
```

```
      intElement.appendChild(index);
      value.appendChild(intElement);

      // Transmit the request document
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();

      OutputFormat fmt = new OutputFormat(request);
      XMLSerializer serializer = new XMLSerializer(out, fmt);
      serializer.serialize(request);

      out.flush();
      out.close();

      // Read the response
      DOMParser parser = new DOMParser();
      InputStream in = connection.getInputStream();
      InputSource source = new InputSource(in);
      parser.parse(source);
      in.close();
      connection.disconnect();

      Document doc = parser.getDocument();
      NodeList doubles = doc.getElementsByTagName("double");
      Node datum = doubles.item(0);
      Text result = (Text) datum.getFirstChild();
      System.out.println(result.getNodeValue());
    }
    catch (Exception e) {
      System.err.println(e);
    }

  }

}
```

25.5. שימוש ב-JAXP

שימוש ב-API זה הינו שילוב של SAX יחד עם DOM והוא חלק אינטגרלי מגרסאות 1.4 ומעלה.

להלן קוד:

```
import java.net.*;
import java.io.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;


public class FibonacciJAXPClient {

  private static String DEFAULT_SERVER
   = "http://www.elharo.com/fibonacci/XML-RPC";

  public static void main(String[] args) {

    if (args.length <= 0) {
      System.out.println(
       "Usage: java FibonacciJAXPClient number url"
      );
      return;
    }

    String server = DEFAULT_SERVER;
    if (args.length >= 2) server = args[1];

    try {
      // Build the request document
      DocumentBuilderFactory builderFactory
       = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder
       = builderFactory.newDocumentBuilder();
      Document request = builder.newDocument();

      Element methodCall = request.createElement("methodCall");
      request.appendChild(methodCall);

      Element methodName = request.createElement("methodName");
      Text text = request.createTextNode("calculateFibonacci");
      methodName.appendChild(text);
      methodCall.appendChild(methodName);

      Element params = request.createElement("params");
      methodCall.appendChild(params);

      Element param = request.createElement("param");
      params.appendChild(param);

      Element value = request.createElement("value");
      param.appendChild(value);

      // Had to break the naming convention here because of a
      // conflict with the Java keyword int
      Element intElement = request.createElement("int");
      Text index = request.createTextNode(args[0]);
```

```
      intElement.appendChild(index);
      value.appendChild(intElement);

      // Transmit the request document
      URL u = new URL(server);
      URLConnection uc = u.openConnection();
      HttpURLConnection connection = (HttpURLConnection) uc;
      connection.setDoOutput(true);
      connection.setDoInput(true);
      connection.setRequestMethod("POST");
      OutputStream out = connection.getOutputStream();

      TransformerFactory xformFactory
       = TransformerFactory.newInstance();
      Transformer idTransform = xformFactory.newTransformer();
      Source input = new DOMSource(request);
      Result output = new StreamResult(out);
      idTransform.transform(input, output);

      out.flush();
      out.close();

      // Read the response
      InputStream in = connection.getInputStream();
      Document response = builder.parse(in);
      in.close();
      connection.disconnect();

      NodeList doubles = response.getElementsByTagName("double");
      Node datum = doubles.item(0);
      Text result = (Text) datum.getFirstChild();
      System.out.println(result.getNodeValue());
    }
    catch (Exception e) {
      System.err.println(e);
    }

  }

}
```

26. שימוש ב- XSLT

Extensible Stylesheet Language Transformations עוצב כשפת תבנית ומכאן שניתן להתייחס אל שפה זו כשפת טרנספורמציה. XSLT מתאר כיצד מסמך בפורמט נתון מומר לפורמט אחר. גם הקלט וגם הפלט מיוצגים ע"י מודל XPath.

XSLT מכיל דוגמאות כיצד ייראה קובץ הפלט, בנוסף הוא מכיל הוראות כיצד לבצע את ההמרה לכל node. מעבד ה-XSLT מבצע שימוש בדוגמאות ובהוראות על מנת לייצור את הקובץ הסופי.

Examples and instructions נכתבים עפ"י template rules כאשר לכל תבנית חוק ישנה תבנית pattern ו- template.

החוק של ה- template מיוצג ע"י אלמנט xsl:template.

ה- pattern מייצג גרסה מוגבלת של XPath מאוכלס בתוך מאפיין Match.

תוכן האלמנט xsl:template מציג את התבנית עצמה.

לדוגמא, להלן template rule שמתאים לאלמנטים methodCall ומגיב לתבנית המכילה אלמנט בודד מסוג methodResponse.

```
<xsl:template match="methodCall">
  <methodResponse>
    <params>
      <param>
        <value><string>Hello</string></value>
      </param>
    </params>
  </methodResponse>
</xsl:template>
```

להלן דוגמת קוד מלאה – ניתן לראות כי אלמנט הנתיב של המסמך הנו xsl:stylesheet אשר קיים לו מאפיין מסוג version עם ערך 1.0

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="methodCall">
    <methodResponse>
      <params>
        <param>
          <value><string>Hello</string></value>
        </param>
      </params>
    </methodResponse>
  </xsl:template>

</xsl:stylesheet>
```

מימוש של הסטייל לכל מסמך בקשה XML-RPC ייתן את התוצר הבא:

```
<?xml version="1.0" encoding="utf-8"?><methodResponse><params>
<param><value><string>Hello</string></value></param></params>
</methodResponse>
```

.26.1.1 שימוש בערך הקיים ב-Node

אחד השימושים הנפוצים ביותר של XSLT הנו שימוש ב- `xsl:value-of` אשר מחזיר את סט התווים של Xpath. לדוגמא:

```
<xsl:value-of select="/methodCall" />
```

האלמנט xsl:value-of לוקח את הערך int לאורך העץ:

```
<xsl:value-of select="/methodCall/params/value/int" />
```

להלן דוגמאת קוד:

XSLT stylesheet that echoes XML-RPC requests

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="methodCall" xml:space="preserve">
<methodResponse>
  <params>
    <param>
      <value>
        <string>
          <xsl:value-of select="params/param/value" />
        </string>
      </value>
    </param>
  </params>
</methodResponse>
</xsl:template>

</xsl:stylesheet>
```

An XML-RPC request document

```
<?xml version="1.0"?>
<methodCall>
  <methodName>calculateFibonacci</methodName>
  <params>
    <param>
      <value><int>10</int></value>
    </param>
  </params>
</methodCall>
```

XML-RPC response document

```
<?xml version="1.0" encoding="utf-8"?>
<methodResponse>
  <params>
    <param>
      <value>
        <string>
          10
        </string>
      </value>
    </param>
  </params>
</methodResponse>
```

דוגמאת קוד XSLT הקורא קלט מבקשת XML-RPC וממיר אותה לתגובת XML-RPC.

XSLT stylesheet that calculates Fibonacci numbers

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <xsl:choose>
      <!-- Basic sanity check on the input -->
      <xsl:when
        test="count(methodCall/params/param/value/int) = 1">
        <xsl:apply-templates select="child::methodCall"/>
      </xsl:when>
      <xsl:otherwise>
        <!-- Sanity check failed -->
        <xsl:call-template name="faultResponse"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="methodCall">
    <methodResponse>
      <params>
        <param>
          <value>
            <xsl:apply-templates
              select="params/param/value/int"/>
          </value>
        </param>
      </params>
    </methodResponse>
  </xsl:template>

  <xsl:template match="int">
    <int>
      <xsl:call-template name="calculateFibonacci">
        <xsl:with-param name="index" select="number(.)"/>
      </xsl:call-template>
    </int>
  </xsl:template>

  <xsl:template name="calculateFibonacci">
    <xsl:param name="index"/>
    <xsl:param name="low"  select="1"/>
    <xsl:param name="high" select="1"/>
    <xsl:choose>
      <xsl:when test="$index &lt;= 1">
        <xsl:value-of select="$low"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="calculateFibonacci">
          <xsl:with-param name="index" select="$index - 1"/>
          <xsl:with-param name="low"   select="$high"/>
          <xsl:with-param name="high"  select="$high + $low"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
```

```
  <xsl:template name="faultResponse">
    <xsl:param name="err_code"    select="0" />
    <xsl:param name="err_message" select="'Unspecified Error'"/>
    <methodResponse>
      <fault>
        <value>
          <struct>
            <member>
              <name>faultCode</name>
              <value>
                <int><xsl:value-of select="$err_code"/></int>
              </value>
            </member>
            <member>
              <name>faultString</name>
              <value>
                <string>
                  <xsl:value-of select="$err_message"/>
                </string>
              </value>
            </member>
          </struct>
        </value>
      </fault>
    </methodResponse>
  </xsl:template>

</xsl:stylesheet>
```