



חושבים Java

ברוס אקל



Authorized translation from the English language edition of:
Thinking in Java
By Bruce Eckel, President, MindView, Inc.
Published by Prentice Hall PTR
Copyright © 2000 By Bruce Eckel, President, MindView, Inc.

תרגום:
יהודה בן-מוחה

בדיקה מקצועית:
עמית בר-ניר

עריכה ממוחשבת:
תמי טרכט

עיצוב גרפי ועטיפה:
סילביה לודמר-כהן,
www.silvia.co.il

לוחות:
מ. אביב בע"מ

הדפסה:
טופSפרינט בע"מ

כריכה:
כריכיית אהרון בע"מ

הפצה לחנויות:
ליאור שרף
שיווק והפצה בע"מ
טל': 09-7498844

חושבים Java
דאנאקוד: 239-363
מהדורה ראשונה בעברית - 2002
הדפסה: 1 2 3

כל הזכויות למהדורה העברית שמורות
להוצאת פוקוס-מחשבים בע"מ
©2002

אין להעתיק, לשכפל ולצלם ספר זה ו/או את התקליטור
המצורף לו, או קטעים מהם בשום צורה ובשום אמצעי
אלקטרוני, אופטי או מכני לכל מטרה שהיא, ללא אישור
בכתב מהוצאת פוקוס-מחשבים בע"מ.

הוצאת פוקוס-מחשבים בע"מ, ת"ד 863 ר"ג, 52108
טל': 1-700-700-212 • פקס: 03-5746513
אתר אינטרנט: <http://www.focus.co.il>
דואר אלקטרוני: focus@focus.co.il

על מנת למנוע סרבול, נכתב הטקסט בלשון זכר
אך יש לקרוא אותו בהתאם למינו של הקורא

תוכן מקוצר

19	הקדמה
31	פרק 1 : מבוא לאובייקטים
85	פרק 2 : כל דבר הוא אובייקט
109.....	פרק 3 : שליטה על זרימת התוכנית
159.....	פרק 4 : אתחול וניקיון
203.....	פרק 5 : הסתרת המימוש
225.....	פרק 6 : שימוש חוזר במחלקות
259.....	פרק 7 : רב-צורתיות
291.....	פרק 8 : ממשקים ומחלקות פנימיות
341.....	פרק 9 : החזקת האובייקטים שלך
451.....	פרק 10 : טיפול בשגיאות באמצעות חריגים
485.....	פרק 11 : מערכת הקלט/פלט של ג'אווה
559.....	פרק 12 : זיהוי טיפוסים בזמן ריצה
585.....	פרק 13 : יצירת חלונות ויישמונים
705.....	פרק 14 : ריבוי הליכי משנה
775.....	פרק 15 : מחשוב מבוזר

- 869..... נספח א': העברה והחזרה של אובייקטים
- 913..... נספח ב': הממשק המקומי של ג'אווה (JNI)
- 923..... נספח ג': קווים מנחים לתכנות בג'אווה
- 933..... נספח ד': משאבים
- 435..... מפתח העניינים

תוכן העניינים

19	הקדמה
31	פרק 1: מבוא לאובייקטים
32	התפתחות ההפשטה
34	לכל אובייקט יש ממשק
36	המימוש המוסתר
37	שימוש חוזר במימוש
38	הורשה: שימוש חוזר בממשק
41	יחס של "הוא סוג של" לעומת יחס של "דומה ל"
42	רב-צורתיות ואובייקטים ניתנים להחלפה
46	מחלקות בסיס מופשטות וממשקים
46	שטח מחייה ואורך חיים של אובייקטים
47	אוספים ואיטרטורים
49	היררכיית השורש היחיד
50	ספריות אוספים ותמיכה בשימוש קל באוספים
52	טיפול בחריגים: התמודדות עם שגיאות
53	ריבוי הליכי משנה
54	התמדה
54	ג'אווה והאינטרנט
55	מהי האינטרנט?
56	תכנות בצד הלקוח
62	תכנות בצד השרת
62	זירה אחרת: יישומים
63	ניתוח ותכנון
65	שלב 0: ערוך תוכנית
66	שלב 1: מה אנחנו בונים?
69	שלב 2: כיצד נבנה זאת?
72	שלב 3: בנה את הגרעין
72	שלב 4: מעבר על פעולות המשתמש
73	שלב 5: אבולוציה
75	התכנון משתלם

75	תכנות אתגרי
75	כתיבת הבדיקות לפני הכל
77	תכנות בזוגות
78	סוד ההצלחה של ג'אווה
78	קל יותר לבטא ולהבין מערכות
78	תנופה מרבית עם ספריות
78	טיפול בשגיאות
78	תכנות בגדול
79	אסטרטגיה למעבר
79	הנחיות
81	מכשולים ניהוליים
82	ג'אווה לעומת C++?
83	סיכום

פרק 2: כל דבר הוא אובייקט

85	באובייקטים מטפלים באמצעות הפניות
86	חובה עליך ליצור את כל האובייקטים
87	המקום בו מתבצע האחסון
88	מקרה מיוחד: סוגי נתונים בסיסיים
90	מערכים בג'אווה
90	לעולם לא תצטרך לפרק אובייקטים
90	טווחי הכרה
91	טווח הכרה של אובייקטים
92	יצירת טיפוסים נתונים חדשים: מחלקה
92	שדות ושיטות
94	שיטות, ארגומנטים וערכי החזרה
95	רשימת הארגומנטים
96	בניית תוכנית ג'אווה
96	זמינות שמות
97	שימוש ברכיבים נוספים
98	מילת המפתח static
99	תוכנית ג'אווה הראשונה שלך
101	הידור והרצה
101	הערות ותיעוד מוטבע
102	תיעוד באמצעות הערות
102	תחביר
103	HTML מוטבע
104	@see - הפניה למחלקות אחרות
104	תגי תיעוד מחלקה

105	תגים לתיעוד משתנים
105	תגים לתיעוד שיטות
106	דוגמה לתיעוד
107	סגנון קידוד
107	סיכום
107	תרגילים

פרק 3: שליטה על זרימת התוכנית

109	שימוש באופרטורים של ג'אווה
110	קדימות
110	אופרטור השמה
113	אופרטורים מתמטיים
115	קידום והסגה באופן אוטומטי
116	אופרטורים יחסיים
118	אופרטורים לוגיים
120	אופרטורים לסיביות
121	אופרטורים להסטה
125	האופרטור הטרנרי (if-else)
126	אופרטור הפסיק
126	אופרטור המחרוזות +
127	מלכודות שכיחות בשימוש באופרטורים
127	אופרטורים להמרה
130	בשפת ג'אווה אין sizeof
130	עוד מילה על קדימות
131	סקירת אופרטורים
142	שליטה על הביצוע
142	אמת ושקר
142	if-else
144	לולאות
145	do-while
145	for
147	continue ו break
153	switch
157	סיכום
157	תרגילים

פרק 4: אתחול וניקיון

159	אתחול מובטח באמצעות הבנאי
161	העמסת שיטות

163	הבחנה בין שיטות מועמסות
164	העמסה עם נתונים בסיסיים
168	העמסת ערכי החזרה
168	בנאי ברירת המחדל
169	מילת המפתח this
173	ניקיון: סגירה ואיסוף אשפה
174	למה נועדה השיטה finalize()?
174	חובה עליך לבצע ניקיון
178	תנאי הפטירה
179	כיצד פועל מאסף אשפה
182	אתחול חברים
184	ציון ערכי אתחול
185	אתחול על ידי הבנאי
192	אתחול מערכים
196	מערכים רב-ממדיים
199	סיכום
200	תרגילים

פרק 5: הסתרת המימוש 203

204	חבילה: יחידת הספרייה
206	יצירת שמות חבילה ייחודיים
209	ספריית כלים אישית
210	שימוש ביבוא כדי לשנות התנהגות
212	אזהרות חבילה
213	מצייני הגישה של ג'אווה
213	ידידותי
214	public: גישת ממשק
215	private: נא לא לגעת!
217	protected: ידידותי שכזה
218	ממשק ומימוש
219	גישת מחלקות
222	סיכום
223	תרגילים

פרק 6: שימוש חוזר במחלקות 225

226	הרכבה: תחביר
229	הורשה: תחביר
231	אתחול מחלקת הבסיס

234	שילוב של הרכבה והורשה
235	הבטחת ניקיון תקין
239	הסתרת שמות
240	בחירה בין הרכבה להורשה
241	protected
242	פיתוח מצטבר
243	המרה למעלה
244	מדוע "המרה למעלה"
245	מילת המפתח final
245	נתונים סופיים
249	שיטות סופיות
251	מחלקות סופיות
252	final: בזהירות
253	אתחול וטעינה של מחלקות
253	אתחול עם הורשה
255	סיכום
256	תרגילים

פרק 7: רב-צורתיות

260	בחינה מחודשת של המרה למעלה
261	לשכוח את טיפוס האובייקט
263	העוקץ
263	קישור בין קריאה לשיטה
264	הפקת ההתנהגות הנכונה
267	יכולת הרחבה
269	דריסה לעומת העמסה
271	מחלקות ושיטות מופשטות
274	בנאים ורב-צורתיות
274	סדר הקריאות לבנאים
277	הורשה ושיטת finalize()
280	התנהגות של שיטות רב-צורתיות בתוך בנאים
282	תכנון עם הורשה
284	הורשה טהורה לעומת הרחבה
285	המרה למטה וזיהוי טיפוסים בזמן ריצה
287	סיכום
288	תרגילים

פרק 8 : ממשקים ומחלקות פנימיות 291.....

292 ממשקים

295 "הורשה מרובה" בג'אווה

298 הרחבת ממשק באמצעות הורשה

299 קיבוץ קבועים

301 אתחול שדות בממשקים

302 קיבון ממשקים

305 מחלקות פנימיות

307 מחלקות פנימיות והמרה למעלה

309 מחלקות פנימיות בתוך שיטות וטווחים

311 מחלקות פנימיות אנונימיות

314 הקישור למחלקה החיצונית

316 מחלקות פנימיות סטטיות

319 הפניה לאובייקט המחלקה החיצונית

320 הפניה החוצה ממחלקה המקוננת במספר רמות

321 ירושה ממחלקות פנימיות

321 הניתן לדרוס מחלקות פנימיות?

324 מזהי מחלקות פנימיות

324 מדוע להשתמש במחלקות פנימיות?

329 מחלקות פנימיות ומסגרות בקרה

336 סיכום

337 תרגילים

פרק 9 : החזקת האובייקטים שלך 341.....

342 מערכים

343 מערכים הם אובייקטים ממעלה ראשונה

346 החזרת מערך

348 המחלקה Arrays

360 מילוי מערך

361 העתקת מערך

362 השוואה בין מערכים

363 השוואה בין איברי מערכים

366 מיון מערך

368 חיפוש במערך ממוין

370 מערכים - סיכום

370 מבוא למכולות

371 הדפסת מכולות

373 מילוי מכולות

379 חסרון של מכולות: טיפוס לא ידוע

381 לעתים זה עובד בכל זאת

383	יצירה של רשימת מערך מודעת לטיפוסים
384	איטרטורים
388	מיפוי של מכולות
390	Collection - תפקודיות אוסף
394	List - תפקודיות רשימה
398	יצירת מחסנית מתוך רשימה מקושרת
399	יצירת תור מתוך רשימה מקושרת
399	Set - תפקודיות קבוצה
402	SortedSet
403	Map - תפקודיות מיפוי
407	SortedMap
408	גיבוב וקודי גיבוב
416	דריסת השיטה hashCode()
419	החזקת הפניות
422	מיפוי-גיבוב חלש
423	בחינה מחודשת של איטרטורים
424	בחירת מימוש
425	בחירה בין רשימות
428	בחירה בין קבוצות
431	בחירה בין מיפויים
433	מיון וחיפוש ברשימות
435	עזרים
435	הפיכת אוסף או מיפוי לבלתי ניתנים לשינוי
437	סנכרון של אוסף או מיפוי
438	פעולות לא נתמכות
441	מכולות ג'אווה 1.0/1.1
441	Enumeration ו-Vector
442	Hashtable
442	Stack
443	BitSet
445	סיכום
446	תרגילים
451	פרק 10 : טיפול בשגיאות באמצעות חריגים
452	חריגים בסיסיים
453	ארגומנטים של חריגה
454	לכידת חריגה
454	מקטע try

454	פונקציות לטיפול בחריגים
456	יצירת חריגים משלך
460	מפרט חריגים
461	לכידת כל החריגים
463	הטלה חוזרת של חריגה
466	חריגים תקינים של ג'אווה
467	המקרה המיוחד של RuntimeException
468	ביצוע ניקיון באמצעות finally
470	למה נועדה ? finally
472	בעיה: החריגה האבודה
473	מגבלות של חריגים
477	בנאים
480	התאמת חריגים
481	הנחיות לשימוש בחריגים
482	סיכום
482	תרגילים
485	פרק 11 : מערכת הקלט/פלט של ג'אווה
486	המחלקה File
486	קבלת פירוט תיקיה
490	חיפוש ויצירת תיקיות
492	קלט ופלט
492	טיפוסים של InputStream
494	טיפוסים של OutputStream
495	הוספת מאפיינים וממשקים שימושיים
496	קריאה מ-InputStream באמצעות FilterInputStream
497	כתיבה ל-OutputStream באמצעות FilterOutputStream
499	המחלקות Reader ו-Writer
499	מקורות ויעדים של נתונים
500	שינוי התנהגות של "זרם"
501	מחלקות שלא השתנו
502	לבד במערכה: RandomAccessFile
502	שימושים אופייניים ב"זרמי" קלט/פלט
505	זרמי קלט
507	זרמי פלט
509	"זרמים ערוציים"
509	קלט/פלט תקיני
509	קריאה מקלט תקיני

510	הפיכת System.out ל-PrintWriter
511	ניתוב מחדש של קלט/פלט תקני
512	דחיסה
513	דחיסה פשוטה עם GZIP
514	אחסון רשימת קבצים עם Zip
516	ארכיבי ג'אווה (JAR)
518	הסדרת אובייקטים
522	מציאת המחלקה
524	בקרה על הסדרה
533	שימוש בהתמדה
540	חלוקה של קלט לאסימונים
541	StreamTokenizer
544	StringTokenizer
546	בדיקת סגנון השימוש ברישיות
555	סיכום
556	תרגילים

פרק 12 : זיהוי טיפוסים בזמן ריצה.....559

559	הצורך ב-RTTI
562	אובייקט Class
564	בדיקה לפני ביצוע המרה
573	תחביר RTTI
575	שיקוף: מידע מחלקות בזמן ריצה
577	חילוץ שיטות מחלקה
582	סיכום
583	תרגילים

פרק 13 : יצירת חלונות ויישומונים.....585

587	יישומון בסיסי
587	מגבלות של יישומונים
588	יתרונות של יישומונים
589	מסגרת היישום
590	הרצת יישומונים בתוך דפדפן אינטרנט
592	שימוש ב-Appletviewer
593	בדיקת יישומונים
594	הרצת יישומונים מתוך שורת פקודה
595	מסגרת תצוגה
598	שימוש ב- Windows Explorer
599	יצירת לחצן

600	לכידת אירוע
603	אזורי טקסט
605	שליטה בפריסה
605	BorderLayout
606	FlowLayout
607	GridLayout
608	GridBagLayout
608	הצבה מוחלטת
608	BoxLayout
613	והגישה המומלצת?
613	מודל האירועים של Swing
614	טיפוסים של אירועים וקשבים
621	מעקב אחר ריבוי אירועים
624	קטלוג של רכיבי Swing
624	לחצנים
627	Icon (סמל)
629	תיאורי-כלי
630	שדות טקסט
632	גבולות
633	חלוניות נגללות
636	עורך טקסט מיניאטורי
637	תיבות סימון
639	לחצני אפשרויות
640	תיבות משולבות (רשימות נפתחות)
641	תיבות רשימה
644	תיבות כרטיסיות
645	תיבות הודעה
647	תפריטים
653	תפריטים מוקפצים
655	שרטוט
658	תיבות דו-שיח
663	תיבות דו-שיח לקבצים
665	HTML ברכיבי Swing
666	מחוננים ופסי התקדמות
667	עצים
670	טבלאות
672	בחירת מראה ותחושה
675	לוח הגזירים
677	אריזת יישומון בקובץ JAR
678	טכניקות של תכנות

678	קישור דינמי של אירועים
680	הפרדת הלוגיקה העסקית מהלוגיקה של ממשק המשתמש
683	צורה כללית (canonical form)
684	תכנות ויזואלי ורכיבי JavaBeans
685	מהו Bean?
687	חילוץ מידע רכיב באמצעות Introspector
694	רכיב JavaBeans מתוחכם יותר
698	אריזת רכיב JavaBeans
699	תמיכה ברכיבים מורכבים יותר
700	עוד על רכיבי JavaBeans
700	סיכום
701	תרגילים

705..... פרק 14 : ריבוי הליכי משנה

706	ממשק משתמש מהיר-תגובה
708	ירושה ממחלקת Thread
710	שימוש בהליכי משנה ליצירת ממשק מהיר-תגובה
713	שילוב הליך המשנה עם המחלקה הראשית
715	יצירת הליכי משנה רבים
718	הליכי רקע
720	שיתוף משאבים מוגבלים
720	גישה לא נכונה למשאבים
725	כיצד ג'אווה משתפת משאבים
730	בחינה מחודשת של רכיבי JavaBeans
735	חסימה
736	כניסה למצב חסום
747	קיפאון
751	עדיפויות
752	קריאה והגדרה של עדיפויות
756	קבוצות הליכי משנה
764	בחינה מחודשת של Runnable
767	יותר מדי הליכי משנה
771	סיכום
772	תרגילים

775..... פרק 15 : מחשוב מבוזר

776	תכנות לרשת
776	זיהוי מחשב
780	שקעים

786 בשירות מספר לקוחות

792 צורות נתונים

792 שימוש בכתובות URL מתוך יישומון

795 עוד על עבודה ברשת

795 קישוריות מסדי נתונים של ג'אווה (JDBC)

799 מה לעשות כדי שהדוגמה תפעל

802 גרסת GUI לתוכנית החיפוש

805 מדוע נראה ממשק התכנות JDBC מסובך כל-כך

806 דוגמה מתוחכמת יותר

814 שירותונים

815 שירותון בסיסי

819 שירותונים וריבוי הליכי משנה

820 טיפול בפעילויות באמצעות שירותונים

824 הרצת דוגמאות השירותונים

824 דפי השרת של ג'אווה (JSP)

826 אובייקטים משתמעים

827 הנחיות JSP

828 רכיבי סקריפטים של JSP

830 חילוף שדות וערכים

831 מאפיינים וטווחים של דף שרת

832 תפעול פעילויות בדפי שרת של ג'אווה

834 יצירה ושינוי של "עוגיות"

835 JSP - סיכום

836 קריאה לשיטות מרוחקות (RMI)

836 ממשקים מרוחקים

837 מימוש של ממשק מרוחק

840 יצירת "ספחים" ו"שלדים"

840 שימוש באובייקט המרוחק

841 CORBA

842 יסודות CORBA

844 דוגמה

848 יישומונים של ג'אווה ו-CORBA

849 CORBA לעומת RMI

849 רכיבים ארגוניים (EJB)

850 רכיבי JavaBeans לעומת רכיבים ארגוניים

851 מפרט EJB

852 רכיבים ארגוניים

853 החלקים של רכיב ארגוני

854 פעולתו של רכיב ארגוני

854 סוגים של רכיבים ארגוניים

855 רכיבי ישות

856	פיתוח רכיב ארגוני
860	רכיבים ארגוניים - סיכום
860	Jini: שירותים מבוזרים
861	Jini בהקשר הנכון
861	מהי Jini?
862	כיצד פועלת Jini
862	תהליך הגילוי
863	תהליך הצירוף
864	תהליך האיתור
864	הפרדה בין ממשק ומימוש
865	הפשטה של מערכות מבוזרות
866	סיכום
866	תרגילים
869	נספח א': העברה והחזרה של אובייקטים
913	נספח ב': הממשק המקומי של ג'אווה (JNI)
923	נספח ג': קווים מנחים לתכנות בג'אווה
933	נספח ד': משאבים
435	מפתח העניינים

הקדמה

בדומה לכל שפת אנוש, מספקת ג'אווה דרך לביטוי רעיונות. הצלחתו של כלי ביטוי זה, תהפוך אותו קל וגמיש בהרבה מהחלופות, ככל שהבעיות תהיינה גדולות ומורכבות יותר

אינך יכול לראות את שפת ג'אווה כאוסף של כלים בלבד - לחלק מכלים אלה אין כל משמעות בנפרד. תוכל להשתמש במכלול החלקים רק אם אתה חושב על **תכנון**, ולא רק על קידוד. וכדי להבין את ג'אווה באופן כזה, עליך להבין את הבעיות של השפה ושל תכנות בכלל. ספר זה דן בבעיות תכנות, בשאלה מדוע הן מהוות בעיות, ובדרך שנקטה שפת ג'אווה כדי לפתור אותן. כתוצאה מכך, קבוצת הכלים שנתאר בכל פרק מבוססת על הדרך בה מאפשרת השפה לפתור סוג מסוים של בעיות. באופן כזה אנו מקווים להניע אותך, צעד אחר צעד, לעבר הנקודה בה צורת החשיבה של ג'אווה תהפוך לשפת האם שלך.

לאורך הספר, נקטנו בגישה הגורסת שעליך לבנות מודל בראש, המאפשר לך לפתח הבנה מעמיקה של השפה. אם תיתקל בבעיה, תוכל להזין אותה למודל שלך ולהסיק את התשובה.

תנאים מוקדמים

ספר זה מניח שיש לך היכרות מסוימת עם עולם התכנות: אתה מבין שתוכנית היא אוסף של משפטים, אתה מבין את הרעיון של שגרת-משנה/פונקציה/מאקרו, אתה מכיר משפטי בקרה כדוגמת "if", ומבני לולאה כדוגמת "while", וכדומה. עם זאת, ייתכן שלמדת נושאים אלה במקומות שונים ומגוונים, כגון תכנות באמצעות שפת מאקרו, או עבודה עם כלי כדוגמת Perl. כל עוד התנסית בתכנות ברמה כזאת שחשת בנוח עם הרעיונות הבסיסיים של התכנות, תהיה מסוגל לעבוד עם ספר זה. כמובן, ספר זה יהיה **קל יותר** למתכנתי C, ועוד יותר למתכנתי ++C, אך אל תוציא את עצמך מהעסק אם אינך בעל ניסיון בשפות אלה (עם זאת, היה מוכן לעבוד קשה. כמו כן, תקליטור המולטימדיה באנגלית המצורף לספר יעדכן אותך באשר לתחביר הבסיסי של שפת C הדרוש ללמידת ג'אווה). ספר זה יציג בפניך את המושגים הבסיסיים של התכנות מונחה-האובייקטים (Object-Oriented Programming - OOP) ואת מנגנוני הבקרה הבסיסיים של ג'אווה, כך שתוכל לעשות אתם הכרה, והתרגילים הראשונים יכללו את משפטי זרימת-הבקרה הבסיסיים.

למרות שתופענה התייחסויות רבות לתכונות שפה של C ו-C++, אלה לא נועדו רק ליודעי דבר, אלא לסייע לכל המתכנתים לראות את ג'אווה מנקודת המבט של אותן שפות שאחרי הכל, ג'אווה היא צאצא שלהן. בכל מקרה, נשתדל שהתייחסויות אלה תהיינה פשוטות, ונסביר כל דבר שלא יהיה מוכר למתכנתים חסרי רקע ב-C ו-C++.

יעדים

בדומה לספרו הקודם של מחבר ספר זה, Thinking in C++, גם ספר זה נבנה סביב התהליך של הוראת השפה. באופן מיוחד, הרעיון היה ליצור כלי שיספק למרצה דרך להוראת השפה בסמינרים, והחלוקה לפרקים של הספר התבססה על השאלה מה יהווה שיעור מוצלח בסמינר. המטרה הייתה להגיע ליחידות עצמאיות שניתן ללמד כל אחת מהן בזמן סביר, בצירוף תרגילים שהם ברי-ביצוע במסגרת אותה הרצאה.

היעדים של ספר זה הם:

- 1 להציג את החומר בפשטות צעד אחר צעד, כך שתוכל לעכל כל רעיון בקלות לפני שתמשיך הלאה.
- 2 להשתמש בדוגמאות קצרות ופשטות ככל שניתן. הדבר מונע לעתים התמודדות עם בעיות "מהעולם האמיתי", אך הניסיון מלמד שתלמידים מאושרים הרבה יותר כשהם יכולים להבין כל פרט בדוגמה, מאשר כשמנסים להרשים אותם בהיקף הבעיה שהיא מסוגלת לפתור. כמו כן, יש גבול ברור לכמות הקוד שניתן לספוג במשך שיעור אחד. מסיבות אלה, הספר יקבל בוודאי ביקורות על השימוש ב"דוגמאות צעצוע", אך עדיף לקבל ביקורות כאלה ולהפיק משהו שיש בו תועלת חינוכית אמיתית.
- 3 לקבוע בזהירות את סדר הצגת הכלים כך שלא תיתקל במשהו שלא הוצג בפניך קודם לכן. כמובן, לעתים הדבר אינו אפשרי, אך במקרים כאלה, יינתן תיאור מבוא קצר.
- 4 להעביר לך מה שחשוב לך להבין על אודות השפה, ולא כל מה שידוע לנו. ניתן לומר שיש היררכיה של חשיבות המידע, ושקיימות עובדות מסוימות ש-95 אחוזים מהמתכנתים לא יידרשו לדעת לעולם, והן רק מבלבלות אנשים ומוסיפות לתחושת הסיבוכיות שיש להם בקשר לשפה. קח למשל דוגמה משפת C: אם תלמד בעל-פה את טבלת קדימות האופרטורים (כותב הספר מעיד על עצמו שלא עשה זאת מעולם), תוכל לכתוב קוד חכם למדי. אך אם עליך לחשוב על כך, הדבר יבלבל גם את הקורא/המתחזק של אותו קוד. לכן תוכל לשכוח מנושא הקדימויות, ולהשתמש בסוגריים בכל פעם שהמצב לא ברור.
- 5 לדאוג לכך שכל סעיף יהיה ממוקד מספיק, כדי שזמן ההרצאה, וזמן השיעור שבין הפסקות התרגול, יהיו קצרים. לא רק שהדבר שומר על ערנות ומעורבות גבוהה של הקהל בזמן סמינר, הדבר גם מעניק לקורא תחושה גדולה יותר של הישגיות.
- 6 לספק לך יסודות מוצקים, כדי שתוכל להבין את הנושאים במידה כזאת שתוכל לעבור בהמשך ללימודים ולספרים מתקדמים וקשים יותר.

תיעוד מקוון

לשפת ג'אווה ולספריות של ג'אווה שניתן להוריד באופן חופשי מאתר חברת Sun Microsystems מצורף תיעוד בפורמט HTML שניתן לקרוא תוך שימוש בדפדפן אינטרנט, וכמעט כל מימוש ג'אווה של צד שלישי כולל מערכת תיעוד זו או מערכת דומה לה. רוב הספרים שפורסמו על שפת ג'אווה משכפלים למעשה תיעוד זה, כך שהוא בוודאי נמצא כבר ברשותך, או שתוכל להוריד אותו בקלות. לכן, מלבד במקרים מיוחדים, ספר זה לא יחזור על התיעוד, שכן קל ומהיר הרבה יותר לאתר תיאור של מחלקה בדפדפן האינטרנט שלך מאשר לחפש אותו בספר (והתיעוד המקוון בדרך כלל גם מעודכן יותר). ספר זה מספק תיאור נוסף של מחלקות רק כשהדבר נחוץ כהשלמה לתיעוד המקוון כדי שתוכל להבין דוגמה מסוימת.

פרקי הספר

ספר זה תוכנן במחשבה אחת בלבד: הדרך בה אנשים לומדים את שפת ג'אווה. ניסיונו של מחבר הספר בהעברת סמינרים סייע לזהות את הנושאים הקשים הדורשים הבהרה נוספת. כמו כן, במקרים בהם הייתה נטייה להעביר כמות גדולה של תכונות בבת אחת, הסתבר תוך כדי הצגת החומר, שאם תכלול תכונות חדשות רבות, עליך גם להסביר את כולן, והדבר רק מגביר את הבלבול של התלמיד. כתוצאה מכך, הושקע מאמץ רב בהצגת הכלים במנות קטנות ככל שניתן.

המטרה אם כן, היא שכל פרק ילמד תכונה יחידה, או קבוצה קטנה של תכונות קשורות, וזאת בלי להסתמך על תכונות נוספות. באופן כזה תוכל לעכל כל חלק בהקשר של הידע הנוכחי שלך לפני שתמשיך הלאה.

להלן תיאור קצר של הפרקים הנכללים בספר זה:

פרק 1: מבוא לאובייקטים

פרק זה סוקר את יסודות התכנות מונחה-האובייקטים, כולל מתן תשובה לשאלה הבסיסית "מהו אובייקט?", ממשק לעומת מימוש, הפשטה וכימוס, הודעות ופונקציות, הורשה והרכבה, הנושא החשוב במיוחד, רב-צורתיות. בפרק זה תקבל גם סקירה של נושאי יצירת אובייקטים כגון בנאים, היכן חיים האובייקטים, היכן יש להציב אותם לאחר שהם נוצרו, ואותו **מאסף אשפה** מופלא המנקה אובייקטים שאינם דרושים יותר. כמו כן יוצגו נושאים נוספים, כגון טיפול בשגיאות עם חריגים, ריבוי הליכי משנה ליצירת ממשק משתמש מהיר-תגובה, ועבודה ברשת ובאינטרנט. תלמד מה מייחד את שפת ג'אווה, מדוע היא הצליחה כל-כך, וכן תלמד על ניתוח ופיתוח מונחי-אובייקטים.

פרק 2:**כל דבר הוא אובייקט**

פרק זה מביא אותך לנקודה בה תוכל לקדד את תוכנית ג'אווה הראשונה שלך, כך שעליו לסקור את הרעיונות העיקריים, כולל הרעיון של הפניה לאובייקט, כיצד ליצור אובייקט, מבוא לסוגי נתונים בסיסיים ולמערכים, טווחי שמות והאופן בו מושמדים אובייקטים על ידי מאסף האשפה, כיצד כל דבר בג'אווה הוא טיפוס נתונים חדש (מחלקה) וכיצד תיצור מחלקות משלך, זמינות של שמות ושימוש ברכיבים מספריות אחרות, מילת המפתח static, וכן הוספת הערות ותיעוד מוטבע.

פרק 3:**שליטה על זרימת התוכנית**

פרק זה פותח בכל האופרטורים שהגיעו לשפת ג'אווה מהשפות C ו- C++. בנוסף לכך, תגלה מלכודות שכיחות בשימוש באופרטורים, וכן תלמד על המרה, קידום, וקדימות של אופרטורים. לאחר מכן תבוא סקירה של זרימת הבקרה והאופרטורים לבחירה הקיימים בכל שפת תכנות כמעט: בחירה באמצעות if-else, לולאות for ו-while, יציאה מלולאה באמצעות break ו-continue, וכן באמצעות משפטי break ו-continue עם תוויות (המסבירים את היעדרותה של מילת המפתח goto משפת ג'אווה), ובחירה באמצעות switch. למרות שלחלק גדול מחומר זה יש קווים משותפים עם הקוד של C ו- C++, הרי שקיימים הבדלים. בנוסף לכך, כל הדוגמאות תהיינה דוגמאות מלאות בשפת ג'אווה, כך שתוכל להתחיל לחוש בנוח עם המראה של קוד ג'אווה.

פרק 4:**אתחול וניקיון**

פרק זה פותח בהצגת הבנאי, המבטיח אתחול כהלכה. הגדרת הבנאי מובילה לרעיון של העמסת פונקציות (מאחר שאתה עשוי להזדקק למספר בנאים). לאחר מכן יבוא דיון בתהליך הניקיון, שאינו תמיד פשוט כפי שזה נדמה. בדרך כלל, אתה פשוט שומט אובייקט לאחר שסיימת לעבוד אתו, ומאסף האשפה יגיע בסופו של דבר וישחרר את הזיכרון. חלק זה של הפרק בוחן את מאסף האשפה וחלק מתכונותיו הייחודיות. הפרק מסתיים במבט קרוב יותר על תהליכי אתחול: אתחול חבר אוטומטי, אתחול חבר מפורש, סדר האתחול, אתחול סטטי, ואתחול מערכים.

פרק 5:**הסתרת המימוש**

פרק זה סוקר את אופן האריזה של קוד, ומדוע נחשפים חלקים מסוימים של ספרייה בשעה שחלקים אחרים מוסתרים. הפרק נפתח בבחינת מילות המפתח package ו- import, המבצעות אריזה ברמת הקובץ, ומאפשרות לך לבנות ספריות של מחלקות. לאחר מכן הוא בוחן נושאים של נתיבי תיקיות ושמות קבצים. יתרת

הפרק סוקרת את מילות המפתח `public`, `private` ו-`protected`, את הרעיון של גישה "ידידותית", ומה פירושו של הרמות השונות של בקרת גישה בעת השימוש בהן בהקשרים שונים.

שימוש חוזר במחלקות

פרק 6:

רעיון ההורשה הוא חלק בלתי נפרד מכל שפות התכנות מונחות-האובייקטים כמעט. זאת דרך לקחת מחלקה קיימת ולהוסיף לתפקודיות שלה (כמו גם לשנות אותה, וזהו נושא פרק 7). הורשה היא פעמים רבות דרך לשימוש חוזר בקוד על ידי שימוש באותה מחלקת בסיס, תוך ביצוע שינויים פה ושם כדי להשיג את ההתנהגות המבוקשת. עם זאת, הורשה אינה הדרך היחידה ליצור מחלקות חדשות ממחלקות קיימות. תוכל גם להטביע אובייקט בתוך המחלקה החדשה שלך באמצעות הרכבה. בפרק זה תלמד על שתי דרכים אלה לשימוש חוזר בקוד בג'אווה, וכיצד ליישם אותן.

רב-צורתיות

פרק 7:

לו היית נדרש לגלות ולהבין לבד את הרב-צורתיות (Polymorphism), אחת מאבני היסוד של התכנות מונחה-האובייקטים, הדבר היה גוזל ממך חודשים ארוכים. באמצעות דוגמאות קטנות ופשוטות, תלמד כיצד ליצור משפחה של טיפוסים באמצעות הורשה, ולעבוד עם אובייקטים מאותה משפחה באמצעות מחלקת הבסיס המשותפת שלהם. הרב-צורתיות בג'אווה מאפשרת לך להתייחס לכל האובייקטים ממשפחה זו באופן כללי, מה שאומר שעיקר הקוד שלך אינו מסתמך על מידע טיפוסים ייחודי. הדבר מעניק לתוכניות שלך יכולת הרחבה, כך שבניית התוכניות ותחזוקת הקוד הופכות קלות יותר וזולות יותר.

משקים ומחלקות פנימיות

פרק 8:

ג'אווה מספקת דרך שלישית להגדרת יחסי גומלין של שימוש חוזר, באמצעות הממשק (interface), שהוא הפשטה טהורה של ממשק אובייקט. ממשק הוא יותר ממחלקה מופשטת באופן קיצוני, שכן הוא מאפשר לך לבצע מעין "הורשה מרובה" כפי שניתן לעשות בשפת ++C, על ידי יצירת מחלקה שניתן להמיר אותה למעלה ליותר ממחלקת בסיס אחת.

בתחילה, נראות המחלקות הפנימיות כמנגנון פשוט להסתרת קוד: אתה מציב מחלקות בתוך מחלקות אחרות. עם זאת, תלמד שהמחלקה הפנימית עושה יותר מזה - היא יודעת על המחלקה המקיפה אותה ויכולה לתקשר אתה - ושסוג הקוד שתוכל לקדד עם מחלקות פנימיות הוא אלגנטי ובהיר יותר, למרות שזאת תפיסה חדשה לרוב הקוראים, ונדרש זמן מה כדי להרגיש בנוח עם תכנון העושה שימוש במחלקות פנימיות.

פרק 9 :**החזקת האובייקטים שלך**

תוכנית המכילה מספר קבוע של אובייקטים שהם בעלי משך חיים ידוע היא פשוטה למדי. בדרך כלל, התוכניות שלך תיצורנה תמיד אובייקטים חדשים בהתבסס על קריטריון כלשהו שיהיה ידוע רק בעת ריצת התוכנית. לפני זמן הריצה, לא תוכל לדעת את כמות האובייקטים שתזדקק להם, או אפילו את הטיפוס המדויק שלהם. כדי לפתור בעיית תכנות כללית זו, עליך להיות מסוגל ליצור כל מספר של אובייקטים, בכל עת ובכל מקום. פרק זה בוחן באופן מעמיק את ספריית המכולות שמספקת ג'אווה 2 כדי להחזיק אובייקטים כשאתה עובד אתם: המערכים הפשוטים, והמכולות המורכבות יותר (מבני נתונים) כדוגמת **רשימת מערך** (ArrayList) ו**מיפוי-גיבוב** (HashMap).

פרק 10 :**טיפול בשגיאות באמצעות חריגים**

הפילוסופיה הבסיסית של ג'אווה היא ש"אין להריץ קוד שנבנה לא נכון". ככל שהדבר ניתן, המהדר לוכד בעיות, אך לעתים אותן בעיות - אם אלה שגיאות תכנות או מצבי שגיאה טבעיים המתרחשים כחלק מביצוע התוכנית הרגיל - ניתנות לזיהוי ולטיפול בזמן ריצה בלבד. ג'אווה כוללת מנגנון **טיפול בחריגים** (exception handling) לטיפול בכל בעיה המתעוררת בזמן ריצת התוכנית. פרק זה מתאר כיצד פועלות בג'אווה מילות המפתח try, catch, throw, throws, ו-finally, מתי עליך להטיל חריגים ומה עליך לעשות כשאתה לוכד אותם. בנוסף לכך, תכיר את החריגים התקניים של ג'אווה, ותלמד כיצד ליצור חריגים משלך, מה קורה עם חריגים בתוך בנאים, וכיצד ממוקמות הפונקציות לטיפול בחריגים.

פרק 11 :**מערכת הקלט/פלט של ג'אווה**

להלכה, ניתן לחלק כל תוכנית לשלושה חלקים: קלט, עיבוד ופלט. הדבר מעיד על כך שקלט/פלט (I/O) הוא חלק חשוב במשוואה. בפרק זה תלמד על המחלקות השונות שמספקת ג'אווה לקריאה וכתובה אל קבצים, בלוקים של זיכרון, ואל המסוף. כמו כן, תתואר ההבחנה בין קלט/פלט "ישן" ו"חדש" בג'אווה. בנוסף לכך, בוחן פרק זה את התהליך של לקיחת אובייקט, "הזרמתו" (כדי שניתן יהיה לשמור אותו בדיסק או לשלוח אותו ברשת) ושחזורו מחדש, תהליך המטופל על ידי **הסדרת האובייקטים** (object serialization) של ג'אווה. כמו כן, מתוארות ספריות הדחיסה של ג'אווה, העובדות עם תבנית JAR.

פרק 12 :**זיהוי טיפוסים בזמן ריצה**

זיהוי הטיפוסים בזמן ריצה (RTTI) של ג'אווה מאפשר לך לזהות את הטיפוס המדויק של אובייקט כשיש בידך רק הפניה לטיפוס הבסיס. בדרך כלל, תעדיף להתעלם במכוון מהטיפוס המדויק של אובייקט ולאפשר למנגנון הקישור הדינמי של ג'אווה (רב-צורתיות) לממש את ההתנהגות הנכונה לאותו טיפוס. אך לעתים מועיל מאוד לדעת את הטיפוס המדויק של אובייקט שעבורו יש לך רק הפניית בסיס. לעתים קרובות מאפשר לך מידע זה לבצע פעולה על בסיס מקרה מיוחד באופן יעיל יותר. פרק זה מסביר למה נועד זיהוי הטיפוסים בזמן ריצה, כיצד תשתמש בו, וכיצד לסלק אותו כשאינו מועיל לך. בנוסף לכך, פרק זה מציג בפניך את מנגנון השיקוף של ג'אווה.

פרק 13 :**יצירת חלונות ויישומונים**

לג'אווה מצורפת ספריית הממשק הגרפי Swing, שהיא ערכה של מחלקות המטפלות בחלונות באופן נייד. יישומי חלונות אלה יכולים להיות הן יישומונים והן יישומים עצמאיים. פרק זה מהווה מבוא ל-Swing וליצירת יישומונים לאינטרנט. הטכנולוגיה החשובה של רכיבי JavaBeans מוצגת בפניך לראשונה. טכנולוגיה זו חיונית ליצירה של כלים לבניית תוכניות בשיטת פיתוח היישומים המהיר (RAD).

פרק 14 :**ריבוי הליכי משנה**

ג'אווה מספקת יכולת מובנית לתמיכה במספר משימות-משנה בו-זמנית, המכונות **הליכי משנה**, הרצות בתוך תוכנית יחידה. (למעשה, אלא אם יש לך מספר מעבדים במחשב שלך, מדובר רק באשליה של ריבוי משימות-משנה.) למרות שניתן להשתמש בהליכי המשנה בכל מקום, הם משמעותיים יותר כשאתה מנסה ליצור ממשק משתמש מהיר-תגובה, כך שלמשל, לא יימנע מהמשתמש ללחוץ על לחצן או להכניס נתונים בעת שמתבצע עיבוד אחר כלשהו. פרק זה בוחן את התחביר והסמנטיקה של ריבוי הליכי המשנה בג'אווה.

פרק 15 :**מחשוב מבוזר**

כל הכלים והספריות של ג'אווה מגיעים למיצוי מלא כשאתה מתחיל לכתוב תוכניות לעבודה ברשתות. פרק זה בוחן את התקשורת ברשתות ובאינטרנט, ואת המחלקות שמספקת ג'אווה כדי להקל על כך. הפרק מציג בפניך את הרעיונות החשובים במיוחד של **שירותונים** (servlets) ו**דפי שרת** (JSP) (לתכנות עבור צד השרת), יחד עם **קישוריות מסדי הנתונים של ג'אווה** (JDBC), וה**קריאה לשיטות מרוחקות** (RMI). לבסוף, הפרק כולל גם מבוא לטכנולוגיות החדשות של Jini, ורכיבי JavaBeans הארגוניים (EJB).

נספח א:**העברה והחזרה של אובייקטים**

מאחר שהדרך היחידה לתקשר עם אובייקטים בג'אווה היא באמצעות הפניות, יש השלכות מעניינות לרעיונות של העברת אובייקט לתוך פונקציה והחזרת אובייקט מפונקציה. נספח זה מסביר מה שעליך לדעת כדי לטפל באובייקטים כשאתה מעביר אותם לתוך פונקציות והחוצה מהן, וכן מציג בפניך את מחלקת String, העושה שימוש בגישה שונה לאותה בעיה.

נספח ב:**הממשק המקומי של ג'אווה (JNI)**

לתוכנית ג'אווה שהיא ניידת לגמרי יש חסרונות רציניים: איטיות, ואי-יכולת לגשת לשירותים הייחודיים לסביבת ההפעלה. כשידועה לך המערכת שהתוכנית רצה עליה, ניתן להאיץ פעולות מסוימות באופן דרמטי על ידי הפיכתן לשיטות מקומיות, שהן פונקציות הכתובות בשפת תכנות אחרת (לעת עתה, נתמכות רק השפות C ו-C++). נספח זה מעניק לך מבוא לתכונה זו, שבאמצעותו תוכל ליצור דוגמאות פשוטות המתחברות עם קוד שאינו ג'אווה.

נספח ג:**קווים מנחים לתכנון בג'אווה**

נספח זה מכיל הצעות שיסייעו להנחות אותך בעת תכנון התוכנית ברמה הבסיסית ובעת כתיבת הקוד.

נספח ד:**משאבים**

נספח זה מרכז בתוכו את עיקר משאבי התוכנה והתיעוד שהוזכרו לאורך הספר.

תרגילים

ניסיונו של מחבר ספר זה מלמד שתרגילים פשוטים יכולים לסייע רבות בהשלמה של הבנת התלמיד בעת סמינר, ולכן תמצא תרגילים כאלה בסופו של כל פרק. רוב התרגילים נועדו להיות קלים מספיק כדי שניתן יהיה לסיים אותם תוך זמן סביר בכיתת לימוד, וכדי לוודא שכל התלמידים קלטו את החומר. חלק מהתרגילים מתקדמים יותר, כדי למנוע שעמום אצל תלמידים מנוסים. הרוב נועדו לפתרון תוך זמן קצר, וכן לליטוש ולהשלמה של הידע שלך. חלקם קשים יותר, אך אין ביניהם תרגילים קשים מדי. (יש להניח שתמצא תרגילים אלה בכוחות עצמך - או למען הדיוק, סביר להניח שהם ימצאו אותך).

פתרונות לתרגילים נבחרים תוכל למצוא במסמך The Thinking in Java Annotated Solution Guide, שניתן לקבל תמורת תשלום סמלי מהכתובת www.BruceEckel.com.

קוד המקור

כל קוד המקור של ספר זה זמין כתוכנה חופשית המוגנת בזכויות יוצרים ומופצת בחבילה אחת, על ידי ביקור באתר האינטרנט www.BruceEckel.com. כדי לוודא שתקבל את הגרסה העדכנית ביותר, זהו האתר הרשמי להפצת הקוד והגרסה האלקטרונית של הספר המקורי באנגלית. תוכל למצוא גרסאות של הספר והקוד גם במספר אתרי "ראי" (תוכל למצוא הפניות לחלקם באתר www.BruceEckel.com), אך ראשית עליך לבדוק באתר הרשמי כדי לוודא שעותקי הראי הם אכן העתקים של המהדורה העדכנית ביותר. תוכל להפיץ את הקוד בכיתות לימוד ובסביבות חינוך אחרות. מטרתה העיקרית של הגנת זכויות היוצרים היא לוודא ציון מדויק של מקור הקוד, ולמנוע ממך להוציא לאור את הקוד מחדש ללא אישור. (כל עוד מקור הקוד ציון, השימוש בדוגמאות מהספר ברוב סוגי המדיה אינו מהווה בעיה בדרך כלל).
בכל קובץ של קוד מקור תמצא התייחסות להודעת הזכויות הבאה:

```

//:!: Copyright.txt
Copyright ©2000 Bruce Eckel
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the

```

```
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///  
~
```

תוכל להשתמש בקוד בפרויקטים ובכיתת לימוד (כולל בחומר היכרות ומבוא), כל עוד תישמר הודעת הזכויות המופיעה בכל קובץ מקור.

מוסכמות קידוד

בדוגמאות הקוד שבספר זה, עשינו שימוש בסגנון קידוד מוגדר. סגנון זה תואם לסגנון בו משתמשת חברת Sun עצמה כמעט בכל הקוד שתמצא באתר שלה (ראה הכתובת java.sun.com/docs/codeconv/index.html), ונתמך על ידי הרוב הגדול של סביבות הפיתוח בג'אווה. הנושא של סגנון עיצוב ראוי לשעות של דיון וויכוח, ועלינו רק לציין שספר זה אינו מנסה להכתיב סגנון נכון דרך הדוגמאות. מאחר ששפת ג'אווה היא שפת תכנות בעיצוב חופשי, תוכל להמשיך ולהשתמש בכל סגנון שנוח לך לעבוד אתו.

התוכניות בספר זה הם קבצים שהוכללו בטקסט על ידי מעבד התמלילים, ישירות מתוך הקבצים שעברו הידור. לכן, קובצי הקוד המודפסים בספר אמורים לפעול כולם ללא שגיאות הידור. השגיאות שאמורות לגרור הודעות שגיאה בזמן הידור סומנו כהערות מסוג '///
' כך שניתן לאתר אותן בקלות ולבחון אותן באופן אוטומטי. שגיאות שיתגלו וידווחו למחבר יופיעו ראשית בקוד המופץ, ומאוחר יותר בגרסאות העדכון של הספר (שגם הן תופענה באתר המחבר www.BruceEckel.com).

גרסאות ג'אווה

בדרך כלל ניתן להשתמש במימוש של חברת Sun כסימוכין לקביעה אם התנהגות מסוימת היא נכונה.

עם הזמן, הוציאה חברת Sun שלוש גרסאות עיקריות של ג'אווה: 1.0, 1.1, ו-2 (המכונה גרסה 2 למרות שהמהדורות של JDK מחברת Sun ממשיכות את שיטת המספור של 1.2, 1.3, 1.4 וכו'). גרסה 2 מביאה סופסוף את ג'אווה לבשלות, במיוחד כשמדובר בכלים לממשק המשתמש. ספר זה מתמקד בג'אווה 2, ונבחן עם ג'אווה 2, למרות שהוא משתדל לעתים לתמוך גם בתכונות מוקדמות יותר של ג'אווה, כדי שהקוד יעבור הידור תחת Linux (באמצעות ערכת הפיתוח האחרונה ל-Linux שהייתה זמינה בעת כתיבת הספר).

אם אתה רוצה ללמוד על גרסאות מוקדמות יותר של השפה שאינן נסקרות בספר זה, הרי שהמהדורה הראשונה המקורית של ספר זה מופיעה בתקליטור המצורף לספר, וניתנת גם להורדה חינם מאתר המחבר www.BruceEckel.com.

אחד הדברים שתוכל להבחין בהם, הוא שבעת אזכור גרסאות קודמות של השפה לאורך הספר, לא השתמשנו במספרי תת-מהדורה, אלא התייחסנו לג'אווה 1.0, 1.1 או 2 בלבד, וזאת כדי להימנע משגיאות דפוס כתוצאה מהופעת גרסאות משנה נוספות של מוצרים אלה.

התקליטור המצורף

התקליטור המצורף מכיל את קוד המקור של הספר, אך הוא מכיל גם את הספר המקורי באנגלית בשלמותו, במספר פורמטים אלקטרוניים. בין אלה, הנוח ביותר לשימוש הוא פורמט HTML, שכן הוא מהיר ומאונדקס בשלמותו. עליך רק ללחוץ על ערך באינדקס או בתוכן העניינים כדי לדלג מיד לאותו חלק בספר.

עם זאת, עיקר הנפח הפיזי של התקליטור מוקדש לקורס מולטימדיה שלם (באנגלית) בשם Thinking in C: Foundations for C++ & Java מאת Chuck Allison, שצורף לספר כתוצאה מכך שאנשים רבים ניגשים ללמוד את השפות C++ או ג'אווה ללא רקע מתאים בשפת C. החשיבה הולכת כך: "אני מתכנת מנוסה, ואני לא רוצה ללמוד את שפת C, אלא רק את C++ או ג'אווה, לכן אדלג על שפת C, ואגש ישר ל-C++/ג'אווה". לאחר התחלת הלימוד, אנשים קולטים שהדרישה המוקדמת להכרה בסיסית של שפת C נבעה מסיבה טובה מאוד. צירוף התקליטור לספר יאפשר לך לקבל את הרקע המתאים. כמובן, תוכל להסתייע גם באחד מספרי C/C++ בעברית, כדוגמת **המדריך לשפת C או המדריך לשפת C++**, בהוצאת פוקוס-מחשבים.

כדי לחסוך בהורדת עשרות MB של תוכנה, צרפנו לתקליטור את כלי הפיתוח של חברת Sun, Java 2 SDK (ראה קובץ התקנה בתיקיה Java2SDK).

מבוא לאובייקטים

ראשיתה של מהפכת המחשוב הייתה במכונה. ראשיתן של שפות התכנות שלנו נוטה לכן להיראות כמו אותה מכונה

אך מחשבים אינם מכונות כפי שהם כלים להגברת החשיבה ("אופניים לשכל" כפי שאוהב לומר Steve Jobs) וסוג שונה של אמצעי ביטוי. כתוצאה מכך, הכלים מתחילים להיראות פחות ופחות כמו מכונות, ויותר כמו חלקים מהתודעה שלנו, וכן להידמות יותר לאמצעי ביטוי אחרים כדוגמת כתיבה, ציור, פיסול, אנימציה, והפקת סרטים. **התכנות מונחה-האובייקטים** (Object-Oriented Programming - OOP) הוא חלק מתנועה זו לעבר השימוש במחשב כאמצעי ביטוי.

פרק זה יציג בפניך את הרעיונות הבסיסיים של התכנות מונחה-האובייקטים, כולל סקירה של שיטות פיתוח. פרק זה, וספר זה, מניחים שיש לך ניסיון מסוים בשפת **תכנות מבני** (procedural programming), אם כי לאו דווקא בשפת C. אם אתה סבור שאתה זקוק להכנה נוספת בתכנות ובתחביר של שפת C לפני שתוכל להתמודד עם ספר זה, תוכל להסתייע בקורס המולטימדיה & Thinking in C: Foundations for C++ מאת Chuck Allison, הנכלל בתקליטור המצורף לספר (וניתן להורדה מאתר המחבר www.BruceEckel.com), או באחד מספרי C שניתן להשיג בעברית, כדוגמת **המדריך לשפת C בהוצאת פוקוס מחשבים**.

פרק זה מכיל חומר רקע וחומר משלים. אנשים רבים לא יחושו בנוח עם קפיצה למי התכנות מונחה-האובייקטים בלי להבין קודם לכן את התמונה הכוללת. לכן, יוצגו בפניך כאן רעיונות רבים כדי להעניק לך סקירה יסודית של התכנות מונחה-האובייקטים. לעומת זאת, רבים יתקשו להבין את רעיונות התמונה הכוללת לפני שראו חלק מהכלים בפעולה. אנשים כאלה עלולים להרגיש תקועים אם לא יוכלו להתנסות באופן אישי במספר דוגמאות קוד. אם אתה נכלל בקבוצה השנייה ומשתוקק להגיע לפרטים הטכניים של השפה, הרגש חופשי לדלג על פרק זה - הדבר לא ימנע ממך

לכתוב תוכניות או ללמוד את השפה. עם זאת, מומלץ שתחזור אליו בהמשך, כדי להעמיק את הידע שלך ולהיות מסוגל להבין מדוע אובייקטים חשובים כל-כך וכיצד לתכנן בהתבסס עליהם.

התפתחות הפשטה

כל שפות התכנות מספקות **הפשטה** (abstraction). ניתן לטעון שמורכבות הבעיות שתוכל לפתור נמצאת ביחס ישר לסוג ההפשטה ולאיכותה. הכוונה במילה "סוג" היא "מהו הדבר שאתה מפשט"? **שפת-סף** (assembly language) היא הפשטה קלה של מערכת המחשב העומדת ביסודה. שפות רבות המכונות "שפות ציווי" (imperative languages - שפות המבוססות על הצבת נתונים במשתנים) שהופיעו לאחר מכן (כגון Fortran, BASIC ו-C) היו הפשטות של שפת-סף. שפות אלו מהוות שיפור ניכר לעומת שפת-הסף, אבל ההפשטה העיקרית שלהן עדיין מחייבת אותך לחשוב במונחים של מבנה המחשב, במקום במונחים של מבנה הבעיה שאתה מנסה לפתור. המתכנת חייב ליצור את הקישור בין מודל המכונה (ב"מרחב הפתרון", שהוא המקום שבו אתה בונה מודל של הבעיה, כמו למשל, במחשב) ובין מודל הבעיה הנפתרת בפועל (ב"מרחב הבעיה", שהוא המקום בו מתקיימת הבעיה עצמה). המאמץ שנדרש למיפוי זה, והעובדה שהוא חיצוני לשפת התכנות, הוליד תוכניות שהיו קשות לכתובה ויקרות לתחזוקה, ויצר כתופעת לוואי את כל תעשיית "שיטות התכנות".

החלופה ליצירת מודל של המכונה, היא יצירת מודל של הבעיה שאתה רוצה לפתור. שפות מוקדמות כמו LISP ו-APL בחרו לראות את העולם מזוויות מוגדרות ("כל הבעיות הן רשימות בסופו של דבר" או "כל הבעיות הן אלגוריתמיות", בהתאמה). שפת PROLOG יוצקת את כל הבעיות לתוך שרשראות של החלטות. כך נוצרו שפות לתכנות מבוסס-אילוצים ולתכנות אך ורק על ידי עבודה עם סמלים גרפיים (הסוג האחרון הסתבר כמגביל מדי). כל אחת מהגישות הללו היא פתרון טוב לסוג הבעיות המוגדר שהיא תוכננה לפתור, אך כשאתה יוצא מהתחום הצר שלהן, הן הופכות מגושמות.

הגישה מונחית-האובייקטים הולכת צעד אחד קדימה בכך שהיא מספקת למתכנת כלים לייצוג רכיבים במרחב הבעיה. ייצוג כזה הוא כללי במידה כזו שהמתכנת אינו מוגבל לסוג מסוים כלשהו של בעיה. אנו מתייחסים לפריטים שבמרחב הבעיה ולייצוגים שלהם במרחב הפתרון, כאל **אובייקטים** (objects). (כמובן, תזדקק גם לאובייקטים אחרים שאינן להם מקבילות במרחב הבעיה). הרעיון הוא שהתוכנית מסוגלת להתאים את עצמה לעולם המונחים של הבעיה, על ידי הוספת סוגים חדשים של אובייקטים. בדרך זו, כשאתה קורא את הקוד שמתאר את הפתרון, אתה קורא מילים המבטאות גם את הבעיה עצמה. זאת הפשטת שפה גמישה וחזקה הרבה יותר מכל מה שהכרנו בעבר. באופן כזה, התכנות מונחה-האובייקטים מאפשר לך לתאר את הבעיה במונחים של הבעיה, ולא במונחים של מערכת המחשב שעליה ירוץ הפתרון. אך בכל זאת קיים קשר עם המחשב. כל אובייקט דומה מאוד למחשב קטן. יש לו מצב (state), ויש לו פעולות שתוכל לבקש ממנו לבצע. עם זאת, הדבר אינו שונה בהרבה מאובייקטים בעולם האמיתי - לכל אחד מהם יש מאפיינים והתנהגויות.

מספר מתכנני שפות החליטו שהתכנות מונחה-האובייקטים כשלעצמו אינו מספיק כדי לפתור בקלות את כל בעיות התכנות, והם מטיפים לשילוב של גישות שונות ליצירת שפות תכנות מרובות מודלים (Multiparadigm programming languages).¹

Alan Kay סיכם חמישה מאפיינים בסיסיים של Smalltalk, שפת התכנות המוצלחת הראשונה שהייתה מונחית-אובייקטים, ואחת השפות שעליהן מבוססת שפת ג'אווה. מאפיינים אלה מייצגים גישה טהורה לתכנות מונחה-אובייקטים:

- 1 כל דבר הוא אובייקט. חשוב על אובייקט כעל משתנה משוכלל. הוא מאחסן נתונים, אבל אתה יכול "להגיש בקשות" לאובייקט זה, כדי שיבצע פעולות על עצמו. להלכה, תוכל לקחת כל רכיב רעיוני של הבעיה שאתה מנסה לפתור (כלבים, מבנים, שירותים, וכו') ולייצג אותו כאובייקט בתוכנית שלך.
- 2 תוכנית היא חבורה של אובייקטים האומרים זה לזה מה לעשות על ידי משלוח הודעות. כדי להגיש בקשה לאובייקט, אתה "שולח הודעה" לאותו אובייקט. תוכל לחשוב על הודעה כזו באופן מוחשי יותר כעל בקשה לקרוא לפונקציה ששייכת לאובייקט מסוים.
- 3 לכל אובייקט יש זיכרון משלו הבנוי מאובייקטים אחרים. במילים אחרות, אתה יוצר סוג חדש של אובייקט על ידי יצירת חבילה המכילה אובייקטים קיימים. באופן כזה, תוכל לבנות מורכבות בתוכנית שלך, תוך הסתרתה מאחורי פשטותם של האובייקטים.
- 4 לכל אובייקט יש טיפוס. במונחי השפה, כל אובייקט הוא מופע (instance) של מחלקה (class), כאשר "מחלקה" היא סוג, או "טיפוס". המאפיין המזהה החשוב ביותר של מחלקה הוא "מהן ההודעות שניתן לשלוח אליה?"
- 5 כל האובייקטים מטיפוס מוגדר יכולים לקבל את אותן הודעות. הצהרה זו אינה מדויקת, כפי שתיווכח בהמשך. מאחר שאובייקט מסוג עיגול הוא גם אובייקט מסוג צורה, מובטח שעייגול יקבל הודעות של צורה. פירושו של דבר שתוכל לכתוב קוד שמדבר עם צורות ולטפל באמצעותו באופן אוטומטי בכל דבר שמתאים לתיאור של צורה. חילופיות (substitutability) זו היא אחד הרעיונות רבי העצמה בתכנות מונחה-האובייקטים.

Booch מציע תיאור תמציתי אפילו יותר של האובייקט:

לאובייקט יש מצב (state), התנהגות (behavior) וזהות (identity).

פירושו של דבר שאובייקט יכול להכיל מידע פנימי (להגדרת מצב) ושיטות (להפקת התנהגות), ושכל אובייקט ניתן להבחנה באופן ייחודי מכל האובייקטים האחרים, ואם לומר זאת באופן מוחשי יותר, לכל אובייקט יש כתובת ייחודית בזיכרון.²

¹ ראה הספר Multiparadigm Programming in Leda מאת Timothy Budd בהוצאת Addison Wesley.

² הגדרה זו מגבילה במידת מה, שכן אובייקטים יכולים בהחלט לשכון במחשבים שונים ובמרחבי שמות שונים, וניתן גם לאחסן אותם בדיסק. במקרים כאלה, זהותו של האובייקט חייבת להיקבע על ידי אמצעי אחר מאשר כתובת זיכרון.

לכל אובייקט יש ממשק

אריסטו (Aristotle) היה כנראה הראשון שהתחיל לחקור באופן מעמיק את רעיון ה**טיפוס** (type). הוא דיבר על "מחלקת דגים ומחלקת ציפורים". הרעיון שכל האובייקטים, בנוסף להיותם ייחודיים, מהווים גם חלק ממחלקה של אובייקטים שיש להם מאפיינים והתנהגויות משותפים, הופיע באופן ישיר בשפת התכנות הראשונה שהייתה מונחית-אובייקטים, Simula-67, במילת המפתח הבסיסית שלה, class (מחלקה), ששימשה להצגת טיפוס חדש בתוכנית.

שפת Simula, כפי שמשמע משמה, נוצרה כדי לפתח סימולציות כמו הדוגמה הקלאסית "בעיית הכספר בבנק". בדוגמה זו קיימת חבורה של כספרים, לקוחות, חשבונות, עסקאות ויחידות של כסף - הרבה "אובייקטים". אובייקטים אלה הם זהים זה לזה, פרט למצבם במהלך ביצוע התוכנית, מקובצים יחד לתוך "מחלקות של אובייקטים", ומכאן באה מילת המפתח class. יצירת סוגי נתונים מופשטים (מחלקות) היא רעיון בסיסי בתכנות מונחה-אובייקטים. סוגי נתונים מופשטים פועלים כמעט בדיוק כמו טיפוסים מובנים: ניתן ליצור משתנים של טיפוס מוגדר, הנקראים **אובייקטים** (objects) או **מופעים** (instances) במונחי התכנות מונחה-אובייקטים, ולעבוד עם משתנים אלו, מה שנקרא **משלוח הודעות** (messages) או **בקשות** (requests). אתה שולח הודעה, והאובייקט מחליט מה לעשות אתה. המשתנים **החברים** (members) בכל מחלקה חולקים מספר תכונות משותפות: לכל חשבון יש יתרה, כל כספר יכול לבצע הפקדה, וכו'. יחד עם זאת, לכל חבר יש מצב משלו: לכל חשבון יש יתרה שונה, לכל כספר יש שם. כתוצאה מכך, ניתן לייצג כל אחד מן הכספרים, הלקוחות, החשבונות, העסקאות, וכו', על ידי ישות ייחודית בתוכנית המחשב. ישות זו היא האובייקט, וכל אובייקט משתייך למחלקה נתונה, המגדירה את המאפיינים וההתנהגויות שלו.

וכך, למרות שמה שאנו עושים באמת בתכנות מונחה-אובייקטים, הוא ליצור סוגי נתונים חדשים, כל שפות התכנות מונחות-אובייקטים כמעט משתמשות במילת המפתח class. כשאתה רואה את המילה type (טיפוס), חשוב על class, ולהפך.³

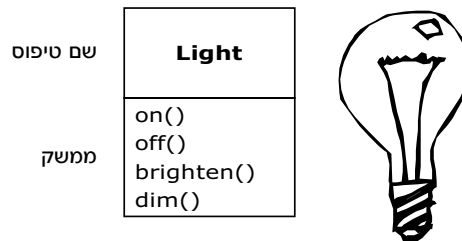
מאחר שמחלקה מתארת קבוצה של אובייקטים שיש להם מאפיינים זהים (רכיבי נתונים) והתנהגויות זהות (תפקודיות), מחלקה היא למעשה **סוג נתונים** (data type), שכן גם לערך נקודה צפה, למשל, יש קבוצה של מאפיינים והתנהגויות. ההבדל הוא שמתכנת מגדיר מחלקה כך שתתאים לבעיה, במקום שייאלץ להשתמש בסוג נתונים קיים שתוכנן לייצוג של יחידת אחסון במחשב. אתה מרחיב את שפת התכנות על ידי הוספת סוגי נתונים חדשים המתאימים לצרכיך. מערכת התכנות מקבלת בברכה את המחלקות החדשות ומעניקה להן את כל הטיפול ו**בדיקת הטיפוסים** (type checking) שהיא מספקת לטיפוסים מובנים.

³ יש המבחינים בין השניים, וטוענים שטיפוס מגדיר ממשק, לעומת מחלקה המגדירה מימוש מוגדר של אותו ממשק.

הגישה מונחית-האובייקטים אינה מוגבלת לבניית סימולציות. אם אתה מסכים לקביעה שכל תוכנית היא סימולציה של המערכת שאתה מתכנן ואם לא, השימוש בטכניקות מונחות-אובייקטים יכול לצמצם בקלות קבוצה גדולה של בעיות לפתרון פשוט.

מרגע שהוגדרה מחלקה, תוכל ליצור כל מספר של אובייקטים ממחלקה זו שתרצה, ואז לעבוד עם אובייקטים אלה כאילו היו רכיבים של הבעיה עצמה שאתה מנסה לפתור. ואכן, אחד האתגרים של התכנות מונחה-האובייקטים הוא ליצור מיפוי של אחד-לאחד בין הרכיבים של מרחב הבעיה לבין האובייקטים שבמרחב הפתרון.

אבל כיצד תגרום לאובייקט לעשות עבורך עבודה שימושית? חייבת להיות דרך להגיש בקשה לאובייקט כדי שיעשה משהו, כגון השלמת עסקה, שרטוט דבר מה על המסך או הפעלת מתג. וכל אובייקט יכול למלא אחר בקשות מסוימות בלבד. הבקשות שבאפשרותך להגיש לאובייקט מוגדרות על ידי הממשק (interface) שלו, והטיפוס הוא שקובע את הממשק. דוגמה פשוטה לכך עשוי להיות ייצוג של נורה:



```
Light lt = new Light();
lt.on();
```

הממשק קובע אילו בקשות ניתן לבקש מאובייקט מסוים. עם זאת, חייב להיות קוד במקום כלשהו שימלא אחר אותה בקשה. קוד זה, יחד עם הנתונים המוסתרים, מהווה את המימוש (implementation). מנקודת המבט של התכנות המבני, הדבר אינו מסובך כל-כך. לטיפוס יש פונקציה מתאימה לכל בקשה אפשרית, וכשאתה מגיש בקשה מסוימת לאובייקט, הפונקציה המתאימה נקראת. תהליך זה מסתכם בדרך כלל בכך שאתה שולח הודעה (מגיש בקשה) אל האובייקט, והאובייקט מחליט מה לעשות עם ההודעה (הוא מבצע קוד).

בדוגמה שלנו, שם הטיפוס/המחלקה הוא Light ("אור"), שם אובייקט המסוים שבדוגמה הוא lt, והבקשות שבאפשרותך להגיש לאובייקט מטיפוס Light הן להדליק אותו, לכבות אותו, להגביר אותו או לעמעם אותו. אתה יוצר אובייקט Light על ידי הגדרת הפניה (reference) לאובייקט זה (lt) וקריאה למילת המפתח new (חדש) כדי לבקש אובייקט חדש מסוג זה. כדי לשלוח הודעה אל האובייקט, אתה מציין את שם האובייקט, ומחבר אותו עם בקשת ההודעה באמצעות נקודה. מנקודת המבט של משתמש במחלקה מוגדרת מראש, בזה מסתכם התכנות עם אובייקטים.

התרשים שהובא לעיל עשוי בתבנית UML (Unified Modeling Language) - שפת רישום מאוחדת). כל תיבה מייצגת מחלקה, כאשר שם הטיפוס מופיע בחלקה העליון של התיבה, נתונים חברים שאתה רוצה לתאר מופיעים בחלקה האמצעי, ופונקציות חברות (member functions) - פונקציות השייכות לאובייקט זה ומטפלות בכל הודעה שאתה שולח אל האובייקט) מופיעות בחלקה התחתון של התיבה. לעתים קרובות, מוצגים בתרשימי UML רק שמות המחלקה והפונקציות החברות הציבוריות, ולכן לא מופיע החלק האמצעי. אם אתה מתעניין רק בשם המחלקה, אין חובה להציג גם את החלק התחתון.

המימוש המוסתר

ניתן לחלק את מגרש המשחקים שלנו ליוצרי מחלקות (class creators) - אלה שיוצרים סוגי נתונים חדשים) ולמתכנתי לקוח⁴ (client programmers) - צרכני המחלקות שמשתמשים באותם סוגי נתונים ביישומים שלהם). מטרתו של מתכנת הלקוח היא לצבור ארגז כלים מלא במחלקות לפיתוח מהיר של יישומים. מטרתו של יוצר המחלקה היא לבנות מחלקה שחושפת רק מה שנחוץ למתכנת הלקוח, ושומרת על כל השאר כשהוא מוסתר. מדוע? כיוון שאם זה מוסתר, מתכנת הלקוח אינו יכול להשתמש בזה, ויוצר המחלקה יוכל לשנות את החלק המוסתר ככל שירצה, בלי לדאוג להשפעות שתהיינה לכך על אחרים. החלק המוסתר מייצג בדרך כלל את הקרביים העדינים של האובייקט, בהם יכול מתכנת לקוח פזיז או לא מיודע, לפגוע בקלות. לכן, הסתרת המימוש מצמצמת את מספר הבאגים בתוכנית. רעיון הסתרת המימוש הוא רעיון חשוב במיוחד.

בכל מערכת יחסים, חשוב לקבוע גבולות שיכובדו על ידי כל הצדדים המעורבים. כשאתה יוצר ספרייה, אתה מבסס מערכת יחסים עם מתכנת הלקוח, שגם הוא מתכנת, אלא שהוא בונה יישום תוך שימוש בספרייה שלך, ואולי גם משתמש בה כדי לבנות ספרייה גדולה יותר.

אם כל חברי המחלקה זמינים לכולם, מתכנת הלקוח יוכל לעשות כל מה שירצה עם המחלקה, ואין שום דרך לאכוף עליו כללים. למרות שאולי היית מעדיף שמתכנת הלקוח לא יעבוד ישירות עם חברי המחלקה שלך, אין כל דרך למנוע זאת ללא בקרת גישה (access control). הכל חשוף אל העולם.

אם כן, הסיבה הראשונה להחלת בקרת גישה, היא כדי למנוע ממתכנתי לקוח לגעת בחלקים של התוכנית שבהם הם לא אמורים לגעת - חלקים הדרושים למנגנון הפנימי של סוג הנתונים אך אינם חלק מן הממשק שהמשתמשים זקוקים לו כדי לפתור את הבעיות המסוימות שלהם. זהו למעשה שירות למשתמשים, שכן בזכותו הם יכולים להבדיל בקלות בין מה שחשוב להם ובין מה שהם יכולים להתעלם ממנו.

הסיבה השנייה להפעלת בקרת גישה, היא כדי לאפשר למתכנן הספרייה לשנות את המנגנון הפנימי של המחלקה בלי לדאוג כיצד ישפיע הדבר על מתכנת הלקוח. לדוגמה, ייתכן שתממש מחלקה מסוימת באופן פשוט כדי להקל על הפיתוח, ולאחר מכן תגלה

⁴ תודות ל- Scott Meyers על מונח זה.

שעליך לשכתב אותה כדי לגרום לה לרוץ מהר יותר. אם הממשק והמימוש מופרדים בבירור ומוגנים, תוכל להשיג זאת בקלות.

שפת ג'אווה עושה שימוש בשלוש מילות מפתח מפורשות כדי להציב את הגבולות במחלקה: `public`, `private` ו-`protected`. השימוש בהן ומשמעותן פשוטים להבנה. מצייני גישה (access specifiers) אלה קובעים מי יכול להשתמש בהגדרות הבאות אחריהם. המילה `public` (ציבורי) פירושה שההגדרות שמופיעות אחריה זמינות לכולם. לעומתה, מילת המפתח `private` (פרטי) פירושה שאף-אחד לא יכול לגשת להגדרות אלו זולתך, יוצר הטיפוס, מתוך פונקציות חברות של אותו טיפוס.

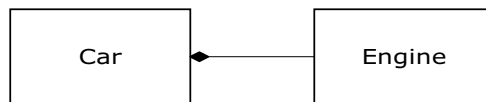
מילת המפתח `private` דומה לחומת לבנים בינך לבין מתכנת הלקוח. אם מישהו ינסה לגשת לחבר פרטי, הוא יקבל שגיאה בזמן הידור. המילה `protected` (מוגן) מתפקדת כמו `private`, פרט לכך שמחלקה יורשת יכולה לגשת לחברים מוגנים, אך לא לחברים פרטיים. נושא ההורשה יוסבר בהמשך.

בשפת ג'אווה קיימת גם גישת ברירת מחדל, שנכנסת לתוקף אם אינך משתמש באף אחד מן המציינים שהוזכרו לעיל. גישה זו מכונה לעתים גישה "ידידותית" (`friendly`), שכן מחלקות יכולות לגשת לחברים הידידותיים של מחלקות אחרות באותה חבילה (`package`), אך מחוץ לחבילה אותם חברים ידידותיים יופיעו כפרטיים.

שימוש חוזר במימוש

מרגע שמחלקה נוצרה ונבדקה, היא אמורה (באופן אידיאלי) לייצג יחידה שימושית של קוד. מסתבר שהיכולת לשימוש חוזר (`reusability`) אינה פשוטה כל-כך להשגה כפי שיכולת לקוות. דרושים ניסיון ותובנה כדי להפיק תכנון מוצלח. אך ברגע שיש בידך תכנון כזה, הוא פשוט מזמין שימוש חוזר. שימוש חוזר בקוד הוא אחד היתרונות הגדולים ביותר שמספקת שפת תכנות מונחית-אובייקטים.

הדרך הפשוטה ביותר לשימוש חוזר במחלקה, היא להשתמש ישירות באובייקט של אותה מחלקה, אך תוכל גם להציב אובייקט של מחלקה זו בתוך מחלקה חדשה. אנו מכנים זאת "יצירת אובייקט חבר". המחלקה החדשה שלך יכולה להיות מורכבת מכל מספר וטיפוס של אובייקטים אחרים, ובכל שילוב שתזדקק לו כדי להשיג את התפקודיות הרצויה במחלקה החדשה שלך. מאחר שאתה מרכיב מחלקה חדשה מתוך מחלקות קיימות, רעיון זה מכונה הרכבה (`composition`), או באופן כללי יותר, צבירה (`aggregation`). לעתים קרובות מתייחסים להרכבה כאל מערכת יחסים מסוג "has-a" (יש לו), כמו בדוגמה "למכונית (Car) יש מנוע (Engine)".



(תרשים UML לעיל מזהה הרכבה באמצעות סימן המעוין המלא, המציין שקיימת מכונת יחידה. בדרך כלל נשתמש בצורה פשוטה יותר: קו בלבד, ללא מעוין, המצביע על השתייכות.⁵)

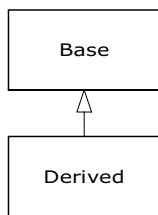
הרכבה מתאפיינת בגמישות רבה. האובייקטים החברים במחלקה החדשה שלך הם פרטיים בדרך כלל, מה שהופך אותם לבלתי-נגישים למתכנתי הקוח המשתמשים במחלקה. הדבר מאפשר לך לשנות את אותם חברים בלי לפגוע בקוד לקוח קיים. תוכל גם לשנות את האובייקטים החברים בזמן ריצה, כדי לשנות באופן דינמי את התנהגות התוכנית. **הורשה** (inheritance), המתוארת בהמשך, אינה מתאפיינת בגמישות כזו, שכן **המהדר** (compiler) חייב להציב הגבלות בזמן הידור על מחלקות שנוצרו באמצעות הורשה.

מאחר שההורשה חשובה כל-כך בתכנות מונחה-האובייקטים, קיימת נטייה להדגיש אותה מאוד לעתים קרובות, ומתכנת חדש עלול לחשוב שעליו להשתמש בהורשה בכל מקום. הדבר עלול להוביל לתכנון מגושם ומסובך מדי. במקום זאת, נסה קודם להשתמש בהרכבה כשתיצור מחלקות חדשות, שכן שיטה זו פשוטה וגמישה יותר. אם תנקוט בגישה זו, התכנון שלך יהיה נקי יותר. לאחר שתצבור קצת ניסיון, יהיה לך ברור למדי מתי עליך להשתמש בהורשה.

הורשה: שימוש חוזר בממשק

הרעיון של אובייקט כשלעצמו הוא כלי נוח. הוא מאפשר לארוז נתונים ותפקודיות יחדיו על בסיס של רעיון, כך שתוכל לייצג רעיון מבוקש ממרחב הבעיה במקום שתיאלץ להשתמש בביטויים השאולים משפת המחשב. רעיונות אלה מבוטאים כיחידות בסיסיות בשפת התכנות על ידי שימוש במילת המפתח class (מחלקה).

עם זאת, נראה שחבל לטרוח על יצירת מחלקה, ולאחר מכן להיות נאלץ ליצור שוב מחלקה חדשה, שעשויה להכיל תפקודיות דומה. היה עדיף לו יכולנו לקחת את המחלקה הקיימת, לשכפל אותה, ואז לבצע רק הוספות והתאמות בהעתק המשוכפל.



זהו בערך מה שמעניקה לך **הורשה** (inheritance), פרט לכך שאם ייערך שינוי במחלקה המקורית (המכונה **מחלקת הבסיס** - base class - או **מחלקת-על** - superclass, או **מחלקת אב** - parent class), הרי שההעתק המשופר (המכונה **מחלקה נגזרת** - derived class - או **מחלקה יורשת** - inherited class, או **תת-מחלקה** - subclass, או **מחלקת צאצא** - child class) ישקף גם הוא את השינויים הללו.

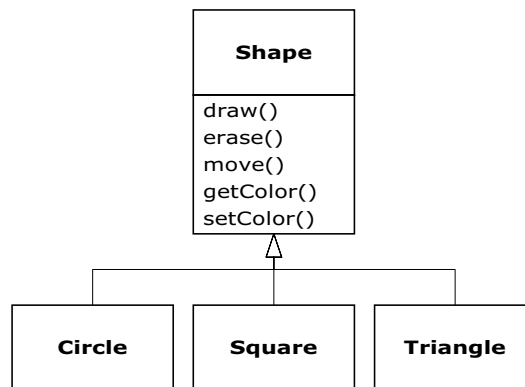
(החץ בתרשים UML לעיל מצביע מן המחלקה הנגזרת אל מחלקת הבסיס. כפי שתראה בהמשך, יכולה להיות יותר ממחלקה נגזרת אחת.)

⁵ בדרך כלל, זהו פירוט מספיק ברוב התרשימים, כשאינך זקוק לפרטים מדויקים עד כדי מענה לשאלה אם נעשה שימוש בצבירה או בהרכבה.

טיפוס עושה יותר מאשר לתאר את ההגבלות על קבוצה של אובייקטים. יש לו גם יחסי גומלין עם טיפוסים אחרים. לשני טיפוסים יכולים להיות מאפיינים והתנהגויות משותפים, אך טיפוס אחד עשוי להכיל יותר מאפיינים מן השני, ועשוי גם לטפל במספר גדול יותר של הודעות (או לטפל בהן אחרת). הורשה מבטאת את הדמיון הזה בין טיפוסים באמצעות הרעיון של **טיפוסי בסיס** (base type) ו**טיפוסים נגזרים** (derived type). טיפוס הבסיס מכיל את כל המאפיינים וההתנהגויות המשותפים לטיפוסים שנגזרים ממנו. אתה יוצר טיפוס בסיס כדי לייצג את לב הרעיון שלך באשר לאובייקטים מסוימים במערכת שלך. מטיפוס הבסיס, אתה גוזר לאחר מכן טיפוסים אחרים, כדי להביע את הדרכים השונות בהן ניתן לממש רעיון זה.

לדוגמה, מכונה למחזור זבל ממיינת גושי אשפה. טיפוס הבסיס הוא "אשפה", ולכל גוש אשפה יש משקל, ערך, וכו', והוא ניתן לגריסה, להתכה או לפירוק כימי. מכאן נגזרים טיפוסים מוגדרים יותר של אשפה, שיכולים להיות להם מאפיינים נוספים (לבקבוק יש צבע), או התנהגויות נוספות (פחית אלומיניום ניתן למעוך, פחית פלדה היא מגנטית). בנוסף לכך, התנהגויות מסוימות עשויות להיות שונות (ערכו של נייר תלוי בסוג הנייר ובמצבו). בעזרת הורשה, תוכל לבנות היררכיה של טיפוסים שמבטאת את הבעיה שאתה מנסה לפתור במונחים של הטיפוסים שלה.

דוגמה שנייה היא דוגמת ה"צורות" הקלאסית, המשמשת בדרך כלל במערכת לתכנון בעזרת מחשב או בסימולציות משחק. טיפוס הבסיס הוא "צורה" (Shape) ולכל צורה יש גודל, צבע, מיקום, וכו'. כל צורה ניתן לשרטט (draw()), למחוק (erase()), להזיז (move()), לצבוע (setColor()), וכו'. מכאן נגזרים (בהורשה) טיפוסים מוגדרים של צורות: **עיגול** (Circle), **ריבוע** (Square), **משולש** (Triangle), וכן הלאה, כאשר לכל אחד יכולים להיות התנהגויות ומאפיינים נוספים, כמו לדוגמה, צורות מסוימות שניתן גם להפוך. התנהגויות מסוימות עשויות להשתנות מצורה לצורה, כמו למשל אופן החישוב של שטח הצורה. היררכית הטיפוסים מגלמת בתוכה הן את הדמיון והן את ההבדלים בין הצורות:



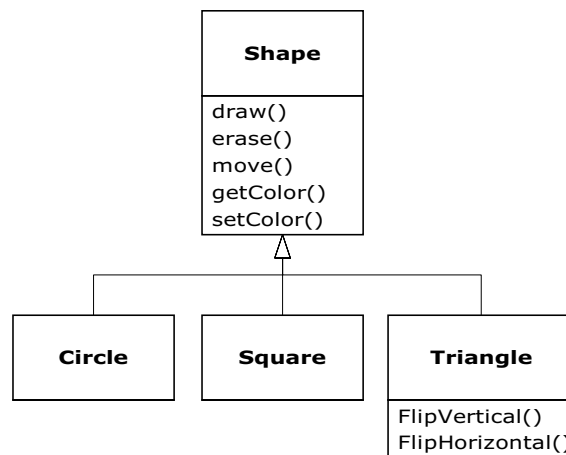
גיבוש הפתרון במונחי הבעיה מועיל במיוחד כיוון שאינך זקוק למודל ביניים כלשהו כדי להגיע מתיאור הבעיה לתיאור הפתרון. בעבודה עם אובייקטים, היררכית הטיפוסים היא המודל העיקרי, כך שאתה עובר ישירות מתיאור המערכת בעולם

האמיתי לתיאור המערכת בקוד. ואכן, אחת הבעיות שיש לאנשים עם התכנון מונחה-האובייקטים היא העובדה שפשוט מדי להגיע מההתחלה אל הסוף. מוח המורגל לחיפוש אחר פתרונות מורכבים מתבלבל בתחילה לעתים קרובות למראה פשטות זו.

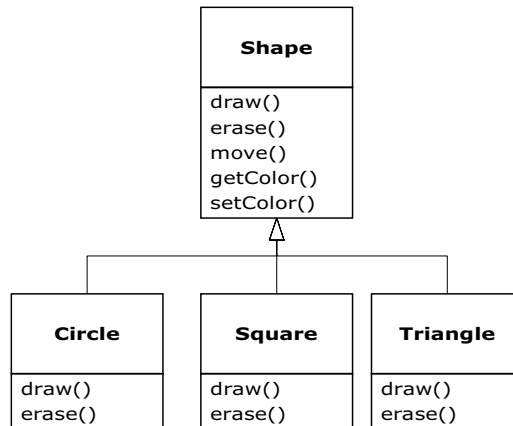
כשאתה יורש מטיפוס קיים, אתה יוצר טיפוס חדש. טיפוס חדש זה מכיל את כל החברים של הטיפוס הקיים (למרות שהחברים הפרטיים מוסתרים ולא ניתן לגשת אליהם), אך חשוב מזה, הוא משכפל את הממשק של מחלקת הבסיס. כלומר, את כל ההודעות שתוכל לשלוח לאובייקטים של מחלקת הבסיס, תוכל לשלוח גם לאובייקטים של המחלקה הנגזרת. כיוון שאנו יודעים את טיפוס המחלקה לפי ההודעות שאנו יכולים לשלוח אליה, פירושו של דבר הוא שמחלקה נגזרת היא תמיד **מאותו הטיפוס של מחלקת הבסיס**. למשל, בדוגמה הקודמת, "עיגול הוא צורה". שקילות זו בין הטיפוסים באמצעות ההורשה, היא אחד השערים הראשיים להבנת משמעותו של התכנות מונחה-האובייקטים.

כיוון שגם מחלקת הבסיס וגם המחלקה הנגזרת הן בעלות אותו ממשק, חייב להיות מימוש כלשהו שיתאים לממשק זה. כלומר, צריך להיות קוד שיתבצע כשאובייקט יקבל הודעה מסוימת. אם רק תירש מחלקה ולא תעשה דבר בנוסף לכך, שיטות הממשק של מחלקת הבסיס יעברו ישירות למחלקה הנגזרת. פירושו של דבר, שאובייקטים של המחלקה הנגזרת לא יהיו רק מאותו טיפוס, אלא שתהיה להם גם אותה התנהגות בדיוק, וזה כבר לא מעניין במיוחד.

יש לך שתי דרכים להבדיל בין המחלקה הנגזרת החדשה לבין מחלקת הבסיס המקורית. הראשונה פשוטה למדי: אתה פשוט מוסיף פונקציות חדשות למחלקה הנגזרת. פונקציות חדשות אלו אינן חלק מן הממשק של מחלקת הבסיס. פירושו של דבר שמחלקת הבסיס פשוט לא עשתה כל מה שרצית שהיא תעשה, ולכן הוספת פונקציות נוספות. שימוש פשוט ובסיסי זה בהורשה, הוא לעתים הפתרון המושלם לבעיה שלך. עם זאת, עליך לבדוק היטב אם מחלקת הבסיס לא עשויה גם היא להזדקק לפונקציות נוספות אלו. תהליך זה של בחינה ומעבר מחודש על התכנון שלך מתרחש בקביעות בתכנות מונחה-אובייקטים.



למרות שהורשה עשויה לרמוז לעתים על כך שאתה מתכוון להוסיף לממשק פונקציות חדשות (במיוחד בג'אווה, שבה מילת המפתח שמציינת הורשה היא extends - "מרחיב את"), הדבר אינו נכון תמיד. הדרך השנייה והחשובה יותר לייחוד המחלקה החדשה שלך, היא לשנות את ההתנהגות של פונקציה קיימת של מחלקת הבסיס. פעולה כזאת מכונה **דריסה** (overriding) של אותה פונקציה.



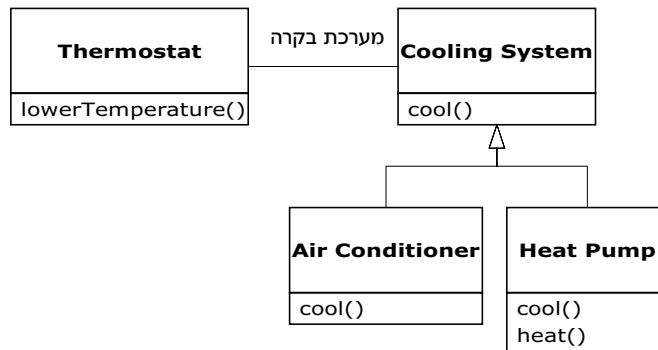
כדי לדרוס פונקציה, אתה פשוט יוצר הגדרה חדשה לאותה פונקציה במחלקה הנגזרת. אתה אומר, "אני משתמש כאן באותה פונקציית ממשק, אך אני רוצה שהיא תעשה משהו שונה עבור הטיפוס החדש שלי".

יחס של "הוא סוג של" לעומת יחס של "דומה ל"

יש ויכוח המתעורר לעתים בקשר להורשה: האם הורשה צריכה רק לדרוס פונקציות של מחלקת הבסיס (ולא להוסיף פונקציות חברות חדשות שאינן קיימות במחלקת הבסיס)? פירושו של דבר, שהטיפוס הנגזר יהיה **בדיוק** מאותו טיפוס כמו מחלקת הבסיס, שכן יהיה לו אותו ממשק בדיוק. כתוצאה מכך, תוכל להחליף אובייקט של מחלקת הבסיס באובייקט של המחלקה הנגזרת. ניתן להגדיר זאת כ**החלפה טהורה** והדבר מכונה לעתים **עקרון ההחלפה** (substitution principle). מבחינה מסוימת, זוהי הדרך האידיאלית לטיפול בהורשה. יחסי הגומלין שבין מחלקת הבסיס לבין מחלקות נגזרות מכונים בדרך כלל יחסי "is-a" (הוא סוג של), כיוון שניתן לומר ש"עייגול הוא סוג של צורה" ("a circle is a shape"). הורשה מתקיימת כאשר ניתן לקבוע שמתקיימים יחסי "הוא סוג של" בין המחלקות, ושיש להם משמעות.

במקרים מסוימים, עליך להוסיף רכיבי ממשק חדשים לטיפוס נגזר, כשאתה מרחיב בכך את הממשק ויוצר טיפוס חדש. עדיין ניתן להחליף בין הטיפוס החדש וטיפוס הבסיס, אך ההחלפה אינה מושלמת, שכן הפונקציות החדשות שלך אינן זמינות בטיפוס הבסיס. ניתן לתאר יחסי גומלין כאלה כיחסי "is-like-a" (דומה ל): לטיפוס החדש יש הממשק של הטיפוס הישן, אך הוא מכיל גם פונקציות אחרות, ולכן הם אינם זהים ממש. קח לדוגמה, מזגן. נניח שביתך מצויד בכל התשתית הנחוצה לשליטה בקירור. כלומר, יש לו

ממשק שמאפשר לך לשלוט בקירור. תאר לעצמך שהמזגן מתקלקל, ואתה מחליף אותו במשאבת חום המסוגלת לחמם וגם לקרר. משאבת החום **דומה** למזגן, אך היא יכולה לעשות יותר. מאחר שמערכת הבקרה בביתך מיועדת אך ורק לשליטה בקירור, היא מוגבלת לתקשורת עם חלק הקירור של האובייקט החדש. הממשק של האובייקט החדש הורחב, אך המערכת הקיימת אינה יודעת דבר מעבר לממשק המקורי.



כמובן, כשתבחן את התכנון הנ"ל, יתברר לך שמחלקת הבסיס המוגדרת כ"מערכת קירור" (Cooling System) אינה כללית מספיק, ויש להגדיר אותה מחדש כ"מערכת בקרת טמפרטורה", כך שתוכל לכלול גם חימום, ומרגע זה ואילך, עקרון ההחלפה יוכל לפעול. עם זאת, תרשים זה הוא דוגמה למה שעשוי להתרחש בתכנון ובעולם האמיתי.

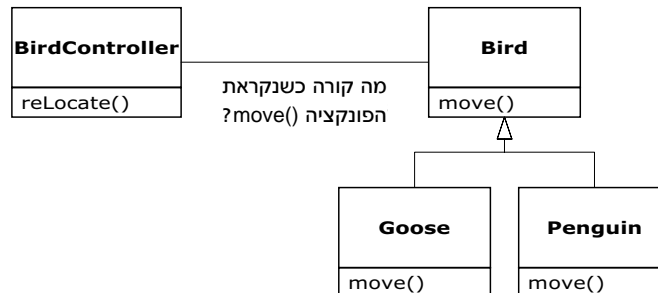
כשתבין את עקרון ההחלפה, קל לחשוב שגישה זו (החלפה טהורה) היא הדרך היחידה לבצע דברים, וללא ספק יהיה נחמד אם התכנון שלך יתפקד באופן כזה. אולם תגלה שישנם מקרים בהם ברור לא פחות שעליך להוסיף פונקציות חדשות לממשק של מחלקה נגזרת. אם תבחן זאת, שני המקרים צריכים להיות מובנים מאליהם.

רב-צורתיות ואובייקטים ניתנים להחלפה

כשמדובר בהיררכיות של טיפוסים, מעת לעת תרצה להתייחס לאובייקט לא כאובייקט מהטיפוס המוגדר שלו, אלא כטיפוס הבסיס שלו. הדבר מאפשר לך לכתוב קוד שאינו תלוי בטיפוסים מוגדרים. בדוגמת הצורות, פונקציות עובדות עם צורות כלליות בלי להתחשב בכך שהן עיגולים, ריבועים, משולשים או צורה אחרת כלשהי שאפילו לא הוגדרה עדיין. את כל הצורות ניתן לשרטט, למחוק ולהזיז, כך שפונקציות אלה פשוט שולחות הודעה לאובייקט צורה. הן אינן מוטרדות מאופן ההתמודדות של האובייקט עם ההודעה.

קוד כזה אינו מושפע מהוספת טיפוסים חדשים, והוספת טיפוסים חדשים היא הדרך הנפוצה ביותר להרחבה של תוכנית מונחית-אובייקטים כך שתהיה מסוגלת להתמודד עם מצבים חדשים. לדוגמה, תוכל לגזור תת-טיפוס חדש של צורה המכונה "מחומש" בלי לשנות את הפונקציות העובדות רק עם צורות כלליות. יכולת זו להרחיב תוכנית בקלות על ידי גזירת תת-טיפוסים חדשים חשובה, שכן שהיא משפרת את התכנון במידה ניכרת תוך צמצום העלויות של תחזוקת התוכנה.

עם זאת, מתעוררת בעיה כשמנסים לעבוד עם אובייקטים מטיפוסים נגזרים כטיפוסי הבסיס הכלליים שלהם (עיגולים כצורות, אופניים ככלי רכב, קורמורנים כציפורים וכו'). אם פונקציה מסוימת אומרת לצורה כללית לשרטט את עצמה, או לכלי רכב כללי לבצע פנייה, או לציפור כללית לזוז, המהדר אינו יכול לדעת בזמן-ההידור איזה קטע קוד בדיוק עליו לבצע. וזהו לב העניין: כשההודעה נשלחת, המתכנת אינו רוצה לדעת איזה קטע קוד בדיוק יתבצע. ניתן להחיל את פונקציית השרטוט על עיגול, ריבוע או משולש באותה המידה, והאובייקט יבצע את הקוד המתאים בהתאם לטיפוס המוגדר שלו. אך אם אינך חייב לדעת איזה קטע קוד יתבצע, כשתוסיף תת-טיפוס חדש, הקוד שהוא יבצע יכול להשתנות בלי שהדבר ידרוש שינוי בקריאה לפונקציה. כתוצאה מכך, המהדר אינו יכול לדעת איזה קטע קוד בדיוק עליו לבצע, ולכן מה הוא עושה? לדוגמה, בתרשים הבא, האובייקט BirdController ("בקר ציפורים") עובד עם אובייקטים מהטיפוס הכללי Bird ("ציפור"), כשהוא אינו יודע מה הטיפוס המוגדר שלהם. הדבר נוח מנקודת מבטו של בקר הציפורים, כיוון שהוא אינו צריך לכתוב קוד מיוחד כדי לזהות את הטיפוס המוגדר של הציפור שעמה הוא עובד, או את ההתנהגות הייחודית של אותה ציפור. אם כך, כיצד קורה שכשנקראת הפונקציה move("זוז"), ללא התחשבות בטיפוס המוגדר של הציפור, מתרחשת ההתנהגות הנכונה (אוזר רץ, עף, או שוחה, לעומת פינגווין שרק רץ או שוחה)?



התשובה לכך היא העוקץ העיקרי בתכנות מונחה-האובייקטים: המהדר אינו יכול לבצע קריאה לפונקציה במובן המסורתי. הקריאה לפונקציה המופקת על ידי מהדר שאינו מונחה-אובייקטים גורמת למה שמכונה **קישור מוקדם** (early binding), מושג שייתכן שלא שמעת בעבר כיוון שמעולם לא חשבת על אפשרות אחרת. פירושו של דבר שהמהדר (compiler) מחולל קריאה לשם פונקציה מוגדר, והמקשר (linker) מפענח את הקריאה הזו לכתובת המוחלטת של הקוד שיש להריץ. בתכנות מונחה-האובייקטים, התוכנית אינה יכולה לקבוע את כתובת הקוד עד לזמן הריצה, ולכן נחוץ כאן מנגנון אחר כשנשלחת הודעה לאובייקט כללי.

כדי לפתור את הבעיה, השפות מונחות-האובייקטים עושות שימוש ברעיון של **קישור מאוחר** (late binding). כשאתה שולח הודעה לאובייקט, הקוד שנקרא נקבע באופן דינמי בזמן הריצה. המהדר מוודא עדיין שהפונקציה קיימת ומבצע **בדיקת טיפוסים** (type checking) על הארגומנטים ועל הערך המוחזר (שפה שבה המהדר אינו עושה זאת מוגדרת כחלשה מבחינת זיהוי הטיפוסים - weakly typed), אך הוא אינו יודע איזה קטע קוד בדיוק עליו לבצע.

כדי לבצע קישור מאוחר (המכונה גם **קישור דינמי** - dynamic binding), ג'אווה עושה שימוש בקטע קוד מיוחד במקום בקריאה מוחלטת. קוד זה מחשב את הכתובת של גוף הפונקציה, תוך שימוש במידע המאוחסן באובייקט (תהליך זה נידון בהרחבה בפרק 7). כתוצאה מכך, כל אובייקט יכול להתנהג באופן שונה לפי התוכן של קטע קוד מיוחד זה. כשאתה שולח הודעה לאובייקט, האובייקט הוא המפענח מה עליו לעשות עם ההודעה. בשפות מסוימות (ובמיוחד בשפת ++C), עליך להצהיר במפורש שאתה רוצה להעניק לפונקציה את הגמישות המאפיינת את הקישור המאוחר. בשפות אלו, פונקציות חברות אינן מקושרות באופן דינמי כברירת מחדל. הדבר גרם לבעיות, ולכן בשפת ג'אווה, נקבע שהקישור הדינמי הוא ברירת המחדל, ואינך צריך להוסיף מילות מפתח כלשהן כדי לקבל **רב-צורתיות** (polymorphism).

חשוב על דוגמת הצורה. משפחת המחלקות (המבוססות כולן על אותו ממשק אחיד) תוארה בתרשים מוקדם יותר בפרק זה. כדי להדגים את הרב-צורתיות, נכתוב קטע קוד יחיד שיתעלם מפרטי הטיפוס המוגדר וידבר עם מחלקת הבסיס בלבד. קוד זה מופרד (decoupled) מהמידע הייחודי לטיפוס, ולכן הוא פשוט יותר לכתיבה וקל יותר להבנה. ואם יתווסף טיפוס חדש (כמו לדוגמה, משושה) על ידי הורשה, הקוד שכתבת יפעל היטב גם עבור טיפוס הצורה החדש, כפי שפעל עבור הטיפוסים הקיימים. כתוצאה מכך, התוכנית **ניתנת להרחבה** (extensible).

אם תכתוב שיטה בג'אווה (כפי שתלמד בקרוב):

```
void doStuff(Shape s) {
    s.erase();
    // ...
    s.draw();
}
```

פונקציה זו מדברת עם כל צורה, ולכן היא אינה תלויה בטיפוס המוגדר של האובייקט שהיא משרטטת ומוחקת. ואם נשתמש במקום אחר כלשהו בתוכנית בפונקציה doStuff("עשה משהו"):

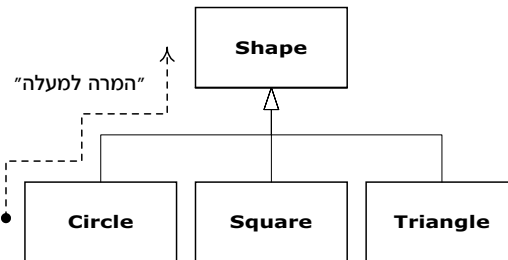
```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

הקריאות ל-doStuff() תפעלנה כהלכה באופן אוטומטי, ללא קשר לטיפוס המוגדר של האובייקט.

למעשה, זאת תחבולה מדהימה למדי. קח לדוגמה את השורה:

```
doStuff(c);
```

מה שקורה כאן הוא שעיגול מועבר לפונקציה המצפה לצורה. כיוון שעיגול הוא סוג של צורה, הפונקציה `doStuff()` יכולה להתייחס אליו כאל צורה. כלומר, עיגול יכול לקבל כל הודעה ש-`doStuff()` יכולה לשלוח לצורה. לכן, פעולה זו הגיונית ובטוחה לחלוטין. אנו קוראים לתהליך זה של התייחסות לטיפוס נגזר כאילו היה טיפוס הבסיס שלו, **המרה למעלה** (upcasting). המילה **המרה** (casting) משמשת במובן של "יציקה לתוך תבנית", והמילה **למעלה** (up) באה מאופן העימוד הנהוג של תרשים ההורשה, כאשר מחלקת הבסיס נמצאת למעלה והמחלקות הנגזרות נפרסות מתחתיה. לכן, המרה למחלקת בסיס היא טיפוס **במעלה** תרשים ההורשה, ומכאן, **המרה למעלה**.



תוכנית מונחית-אובייקטים מכילה המרה כלפי-מעלה במקום כלשהו, כיוון שזו הדרך שבה אתה "מפריד" את עצמך מידיעה על הטיפוס המוגדר שאתה עובד אתו. הבט בקוד של `doStuff()`:

```

s.erase();
// ...
s.draw();

```

שים לב לכך שלא נאמר פה "אם אתה עיגול, עשה כך, אם אתה ריבוע, עשה כך, וכן הלאה". אם תכתוב סוג כזה של קוד, שבודק את כל הטיפוסים האפשריים שצורה יכולה להיות, קוד כזה יהיה מסובך ויהיה עליך לשנות אותו בכל פעם שתוסיף סוג חדש של צורה. בקוד לעיל, אתה פשוט אומר "אתה צורה, אני יודע שאתה יכול למחוק ולשרטט את עצמך, לכן עשה זאת, וטפל בפרטים כפי שנדרש".

מה שמרשים בקוד של `doStuff()`, הוא שאיכשהו, הדבר הנכון אכן קורה. קריאה לפונקציה `draw()` ("שרטט") עבור עיגול גורמת לקוד שונה לרוץ מאשר קריאה לאותה פונקציה עבור ריבוע או קו, אך גם כשהודעת `draw()` נשלחת לצורה אנונימית, ההתנהגות הנכונה מתרחשת בהתבסס על הטיפוס המוגדר של הצורה. הדבר מדהים, שכן כפי שהוזכר קודם לכן, כשמהדר ג'אווה מהדר את הקוד עבור `doStuff()`, הוא אינו יכול לדעת באילו טיפוסים בדיוק מדובר. בדרך כלל, היית מצפה שהוא יקרא בסופו של דבר לשיטות `erase()` ו-`draw()` בגרסאות של מחלקת הבסיס `Shape`, ולא לגרסאות של המחלקות הייחודיות `Circle`, `Square` או `Line`. הדבר הנכון מתרחש בכל זאת הודות לרב-צורתיות. המהדר ומערכת זמן הריצה מטפלים בפרטים. כל שעליך לדעת הוא שזה קורה, וחשוב מזה, כיצד לתכנן בהתחשב בכך. כשאתה שולח הודעה לאובייקט, האובייקט יעשה את הדבר הנכון, גם כשמתבצעת המרה למעלה.

מחלקות בסיס מופשטות וממשקים

לעתים קרובות כחלק מהתכנון שלך, אתה רוצה שמחלקת הבסיס תציג רק ממשק עבור המחלקות הנגזרות שלה. כלומר, אינך רוצה שמישהו ייצור ממש אובייקט של מחלקת הבסיס, אלא רק ימיר אליה למעלה כדי להשתמש בממשק שלה. הדבר נעשה על ידי הפיכת המחלקה למופשטת (abstract) תוך שימוש במילת המפתח abstract. אם מישהו ינסה ליצור אובייקט של מחלקה מופשטת, המהדר ימנע ממנו לעשות זאת. זהו כלי המאפשר לאכוף תכנון מסוים.

תוכל גם להשתמש במילת המפתח abstract כדי לתאר שיטה שלא מומשה עדיין - כשלב של שיטה האומר "זאת פונקציה ממשק עבור כל הטיפוסים הנגזרים ממחלקה זו, אך בשלב זה אין עדיין מימוש עבורה". תוכל ליצור שיטה מופשטת רק בתוך מחלקה מופשטת. כשתירש ממחלקה זו, יהיה עליך לממש את השיטה הזאת, או שגם המחלקה היורשת תהפוך למופשטת. יצירת שיטה מופשטת מאפשרת לך להציב שיטה בממשק בלי שתאלץ לספק לה גוף קוד חסר משמעות.

מילת המפתח interface (ממשק) לוקחת את הרעיון של מחלקה מופשטת צעד אחד קדימה, על ידי מניעה מוחלטת של האפשרות להגדיר פונקציות. הממשק הוא כלי שימושי ונפוץ מאוד בשימוש, כיוון שהוא מספק את ההפרדה המושלמת בין ממשק למימוש. בנוסף לכך, תוכל לשלב מספר ממשקים יחד, אם תרצה, כאשר הורשה ממספר מחלקות רגילות או מופשטות אינה אפשרית.

שטח מחייה ואורך חיים של אובייקטים

טכנית, התכנות מונחה-האובייקטים עוסק בהגדרת טיפוסים מופשטים, בהורשה, וברב-צורתיות, אך קיימים נושאים נוספים החשובים לא פחות. שארית סעיף זה תסקור סוגיות אלו.

אחד הגורמים החשובים ביותר, הוא האופן בו אובייקטים נוצרים ומפורקים. היכן נמצאים הנתונים של האובייקט וכיצד ניתן לשלוט במשך החיים שלו? מספר פילוסופיות שונות מעורבות בכך. שפת ++C גורסת ששליטה על היעילות היא הסוגיה החשובה ביותר, ולכן היא מציעה למתכנת ברירה. למהירות מקסימלית בזמן-ריצה, ניתן לקבוע את מקום האחסון ומשך החיים בזמן כתיבת התוכנית, על ידי הצבת האובייקטים במחסנית (stack) (אלה מכונים לעתים משתנים אוטומטיים - automatic, או משתנים בעלי-טווח - scoped), או בשטח האחסון הסטטי. הדבר מעניק עדיפות למהירות הקצאת האחסון ושחרורו, והשליטה באלה יכולה להיות רבת ערך במקרים מסוימים. עם זאת, אתה מקריב בתמורה מידה של גמישות, כיוון שעליך לדעת את הכמות, משך החיים, והטיפוס המדויקים של האובייקטים כבר בזמן כתיבת התוכנית. אם אתה מנסה לפתור בעיה כללית יותר, כגון תכנון בעזרת מחשב, ניהול מחסן, או בקרת תעבורה אווירית, הדבר מגביל מדי.

הגישה השנייה היא ליצור אובייקטים באופן דינמי במאגר הזיכרון המכונה **ערימה** (heap). בגישה זו, אינך יודע עד לזמן הריצה לכמה אובייקטים תזדקק, מה יהיה משך החיים שלהם, או מהו טיפוס המדויק שלהם. דברים אלה נקבעים לפי צורכי הרגע במהלך ריצת התוכנית. אם אתה זקוק לאובייקט חדש, אתה פשוט יוצר אותו בערימה כשאתה צריך אותו. מאחר שהאחסון מנוהל באופן דינמי בזמן הריצה, משך הזמן הדרוש להקצאת שטח אחסון בערימה ארוך באופן משמעותי ממשך הזמן הדרוש לאחסון במחסנית. (יצירת אחסון במחסנית דורשת בדרך כלל רק הוראה אחת בשפת-סף להזזת מצביע המחסנית למטה, והוראה נוספת להזזתו בחזרה למעלה.) הגישה הדינמית יוצאת מההנחה ההגיונית בדרך כלל שאובייקטים נוטים להיות מסובכים, כך שהעלויות הנוספות של הקצאה ושחרור שטח האחסון לא תהיינה בעלות השפעה מהותית על יצירת אובייקט. בנוסף לכך, הגמישות הרבה יותר חיונית לפתרון בעיית התכנות הכללית.

שפת ג'אווה עושה שימוש בגישה השנייה באופן בלעדי⁶. בכל פעם שתרצה ליצור אובייקט, תשתמש במילת המפתח new (חדש) כדי לבנות מופע דינמי של אותו אובייקט.

סוגיה נוספת היא אורך חייו של אובייקט. בשפות המאפשרות ליצור אובייקטים במחסנית, המהדר הוא הקובע את משך חיי האובייקט, והוא יכול לפרק (להשמיד) אותו באופן אוטומטי. עם זאת, אם תיצור אותו בערימה, לא יהיה למהדר מושג באשר למשך החיים שלו. בשפה כדוגמת ++C, עליך לקבוע באופן תכנותי מתי לפרק את האובייקט, דבר העלול להוביל לדליפות זיכרון אם לא תעשה זאת כהלכה (וזאת בעיה נפוצה בתוכניות ++C). שפת ג'אווה מספקת כלי המכונה **מאסף אשפה** (garbage collector), המגלה באופן אוטומטי מתי אובייקט לא נמצא יותר בשימוש ומפרק אותו. מאסף האשפה נוח בהרבה, כיוון שהוא מצמצם את מספר הפרמטרים שעליך לעקוב אחריהם ואת כמות הקוד שעליך לכתוב. חשוב מזה, מאסף האשפה מספק ביטוח טוב בהרבה נגד הבעיה החמקנית של דליפות זיכרון (בעיה שהורידה פרויקטים רבים ב- ++C על הברכיים).

שארית סעיף זה בוחנת גורמים נוספים הקשורים לשטח המחייה ולמשך החיים של אובייקטים.

אוספים ואיטרטורים

אם אינך יודע לכמה אובייקטים תזדקק כדי לפתור בעיה מוגדרת, או למשך כמה זמן הם יצטרכו להתקיים, לא תוכל לדעת גם כיצד לאחסן את האובייקטים הללו. כיצד תוכל לדעת כמה מקום ליצור עבור אותם אובייקטים? לא תוכל לעשות זאת, שכן מידע זה אינו ידוע עד לזמן הריצה.

הפתרון לרוב הבעיות בתכנון מונחה-האובייקטים נשמע פזרני: אתה יוצר טיפוס נוסף של אובייקט. טיפוס האובייקט החדש הפותר בעיה מסוימת זו, מחזיק הפניות לאובייקטים אחרים. כמובן, תוכל לעשות דבר דומה עם מערך, שהוא טיפוס הקיים ברוב שפות התכנות. אבל זה לא הכל. אובייקט חדש זה, הקרוי בשם הכללי **מְכֻלָּה** (container - הוא נקרא גם **אוסף** - collection - אך ספריית ג'אווה משתמשת במונח זה במובן אחר ולכן

⁶ סוגי הנתונים הבסיסיים (primitive types), עליהם תלמד בהמשך, הם מקרה מיוחד.

נשתמש בספר זה במונח "מכולה", ירחיב את עצמו בכל פעם שיהיה צורך בכך כדי לקבל כל דבר שתשים בתוכו. באופן כזה, אינך נדרש לדעת כמה אובייקטים אתה עתיד להחזיק במכולה. פשוט צור אובייקט מכולה והנח לו לטפל בפרטים.

למרבה המזל, שפת תכנות מונחית-אובייקטים טובה מגיעה עם אוסף של מכולות כחלק מהחבילה. בשפת ++C, אוסף זה הוא חלק מהספרייה התקנית של ++C, והוא מכונה לעתים "ספריית התבניות הבסיסיות" (STL - Standard Template Library). שפת Object Pascal כוללת מכולות בספריית הרכיבים הגרפיים שלה (Visual Component Library - VCL). שפת Smalltalk כוללת אוסף שלם של מכולות. וגם שפת ג'אווה כוללת מכולות בספרייה התקנית שלה. בחלק מהספריות הנ"ל, מכולה כללית נחשבת לטובה מספיק עבור כל הצרכים, ובאחרות (בשפת ג'אווה, למשל) הספרייה מכילה סוגים שונים של מכולות לצרכים שונים: וקטור (המכונה בג'אווה ArrayList - רשימת מערך) לגישה עקבית לכל הפריטים, ורשימה מקושרת (linked list) להכנסה עקבית בכל הפריטים, כדי שתוכל, לדוגמה, לבחור את הטיפוס המסוים המתאים לצרכיך. ספריות של מכולות עשויות להכיל גם קבוצות (sets), תורים (queues), טבלאות גיבוב (hash tables), עצים (trees), מחסניות (stacks) וכדומה.

לכל המכולות יש דרך כלשהי להכנסה ולהוצאה של פריטים מתוכן. בדרך כלל, קיימות פונקציות להוספת פריטים למכולה, ואחרות לאחזור של אותם פריטים. אך אחזור פריטים יכול להיות בעייתי יותר, שכן פונקציה לאחזור ערך יחיד היא מגבילה. מה אם תרצה לעבוד עם קבוצה של פריטים או להשוות בין קבוצות של פריטים במקום לעבוד עם פריט יחיד?

הפתרון הוא האיטרטור (iterator), שהוא אובייקט שתפקידו לבחור את הפריטים בתוך מכולה ולהביא אותם אל המשתמש באיטרטור. כמחלקה, הוא מספק גם רמה של הפשטה. הפשטה זו יכולה לשמש להפרדה בין פרטי המכולה ובין הקוד הניגש אליה. המכולה, באמצעות האיטרטור, מטופלת כסדרה פשוטה. האיטרטור מאפשר לך לעבור על אותה סדרה בלי לדאוג לגבי המבנה הפנימי - כלומר, אם זאת רשימת מערך, רשימה מקושרת, מחסנית, או משהו אחר. הדבר מעניק לך את הגמישות לשנות בקלות את מבנה הנתונים הפנימי בלי לשנות דבר בקוד שבתוכנית שלך. ג'אווה יצאה לדרך (בגרסאות 1.0 ו-1.1) עם איטרטור תקני בשם Enumeration (ספרור) עבור כל מחלקות המכולה שלה. ג'אווה 2 הוסיפה ספריית מכולות שלמה בהרבה, המכילה איטרטור בשם Iterator, שהוא בעל יכולות רבות יותר משל ה-Enumeration הישן.

מנקודת המבט של התכנון, כל מה שדרוש לך באמת הוא סדרה שניתן לעבוד אתה כדי לפתור את הבעיה שלך. אם סוג יחיד של סדרה היה מספק את כל צרכיך, לא הייתה לך כל סיבה ליצור סוגים שונים. קיימות שתי סיבות לכך שאתה זקוק למספר סוגים של מכולות. ראשית, מכולות מספקות סוגים שונים של ממשקים והתנהגות חיצונית. למחסנית יש ממשק והתנהגות שונים משל תור, ואלה שונים מהממשק וההתנהגות של קבוצה או של רשימה. אחד מאלה עשוי לספק פתרון גמיש יותר לבעיה שלך מאשר האחרים. שנית, מכולות שונות מבצעים פעולות מסוימות ברמות יעילות שונות. הדוגמה הטובה ביותר היא רשימת המערך (ArrayList) והרשימה המקושרת (LinkedList). שתיהן סדרות פשוטות שיכולות להיות בעלות ממשק והתנהגות חיצונית זהים. אך לפעולות מסוימות עשויות להיות בשתייהן עלויות שונות מאוד. גישה אקראית

לפריטים ברשימת מערך היא פעולה שצורכת משך זמן קבוע. היא נמשכת אותו פרק זמן, ולא משנה באיזה פריט בדיוק בחרת. לעומת זאת, ברשימה מקושרת, המעבר על הרשימה כדי לבחור בפריט באופן אקראי הוא פעולה יקרה, ונדרש זמן רב יותר למציאת פריט הממוקם לקראת סוף הרשימה. מצד שני, אם אתה רוצה להכניס פריט באמצע הסדרה, זול בהרבה לעשות זאת ברשימה מקושרת מאשר ברשימת מערך. פעולות אלו ואחרות מתבצעות ביעילות שונה בהתאם למבנה הפנימי של הסדרה. בשלב התכנון, אתה עשוי להתחיל עם רשימה מקושרת, כאשר בשלב מאוחר יותר, כשתתמקד בשיפור ביצועים, תעבור לשימוש ברשימת מערך. הודות להפשטה המושגת על ידי שימוש באיטרטורים, תוכל לבצע את המעבר מהאחת לשנייה תוך שינויים מזערניים בקוד שלך.

בסופו של דבר, זכור שמכולה היא רק תיבת אחסון שבה אתה מניח אובייקטים. אם התיבה עונה על כל צרכיך, אין זה משנה כיצד בדיוק היא ממומשת (רעיון בסיסי הנכון לרוב טיפוסים האובייקטים). אם אתה עובד בסביבת תכנות הכרוכה בעלויות מובנות במשאבים עקב גורמים אחרים, ייתכן שהפרש העלויות בין רשימת המערך והרשימה המקושרת לא יהיה משמעותי במיוחד. ייתכן שתזדקק רק לסוג אחד של סדרה. ניתן אף לחשוב על הפשטת מכולה "מושלמת", שתדע לשנות את המימוש הפנימי שלה באופן אוטומטי בהתאם לאופן השימוש בה.

היררכיית השורש היחיד

אחת הסוגיות בתכנות מונחה-האובייקטים שבלטה במיוחד מאז הופעתה של שפת ++C, היא השאלה אם כל המחלקות אמורות להיגזר בסופו של דבר ממחלקת בסיס יחידה. בג'אווה (כמו ברוב השפות האחרות שהן מונחות-אובייקטים) התשובה היא "כן", ושמה של מחלקת בסיס מוחלטת זו הוא פשוט Object (אובייקט). מסתבר שיש יתרונות רבים להיררכיית השורש היחיד.

לכל האובייקטים בהיררכיית שורש יחיד יש ממשק משותף, כך שכולם נגזרים בסופו של דבר מאותו טיפוס. החלופה (אותה מספקת שפת ++C), היא שאינך יודע שכל דבר הוא מאותו טיפוס בסיסי. מנקודת מבט של תאימות לאחור, הדבר מתיישב טוב יותר עם המודל של שפת C, וניתן לטעון שהוא פחות מגביל, אך אם תרצה לבצע תכנות מונחה-אובייקטים במלוא מובן המילה, יהיה עליך לבנות היררכיה משלך כדי לספק את אותה הנוחות שהיא מובנית בשפות מונחות-אובייקטים אחרות. ואז, בכל ספריית מחלקה חדשה שתרכוש, ייעשה שימוש בממשק אחר כלשהו שלא יהיה תואם לשלך, ויידרש מאמץ (ואולי גם הורשה מרובה) כדי לשלב את הממשק החדש בעיצוב שלך. האם ה"גמישות" הנוספת של ++C שווה את זה? אם אתה זקוק לכך - אם חלק גדול מהתוכנית כתוב כבר בשפת C - יש לכך ערך. אם אתה מתחיל מבראשית, סביר להניח שחלופות אחרות כדוגמת ג'אווה תהיינה פוריות יותר.

ניתן להבטיח שלכל האובייקטים בהיררכיית שורש יחיד (כפי שמספקת ג'אווה) תהיה תפקודיות כלשהי. אתה יודע שתוכל לבצע פעולות בסיסיות כלשהן על כל אובייקט במערכת שלך. היררכיית שורש יחיד, יחד עם יצירת כל האובייקטים בערימה, מפשטת מאוד את העברת הארגומנטים (אחד הנושאים המורכבים יותר ב- ++C).

היררכיית השורש היחיד מקלה מאוד על מימוש מאסף האשפה (המובנה לנוחותך בג'אווה). ניתן להתקין את התמיכה הנחוצה במחלקת הבסיס, ומאסף האשפה יוכל לאחר מכן לשלוח את ההודעות המתאימות לכל אובייקט במערכת. ללא היררכיית שורש יחיד ומנגנון לעבודה עם אובייקט באמצעות הפניה, קשה לממש מאסף אשפה.

כיוון שמובטח שמידע טיפוסים בזמן-ריצה יימצא בכל האובייקטים, לעולם לא תמצא את עצמך עם אובייקט שאינך יכול לקבוע את הטיפוס שלו. הדבר חשוב במיוחד בפעולות ברמת המערכת, כגון טיפול בחריגים (exception handling), וכדי לאפשר גמישות רבה יותר בתכנות.

ספריות אוספים ותמיכה בשימוש קל באוספים

מאחר שמכולה היא כלי שבו תשתמש לעתים תכופות, הגיוני שתהיה ספרייה של מכולות הבנויות באופן המאפשר שימוש חוזר, כך שתוכל פשוט לקחת מכולה מן המדף ולהכניס אותה לתוכנית שלך. ג'אווה מספקת ספרייה כזו, האמורה לענות על רוב הצרכים שלך.

המרה למטה לעומת תבניות מחלקה

כדי שמכולות אלה תתאמנה לשימוש חוזר, הן מחזיקות בטיפוס אוניברסלי אחד של ג'אווה שהוזכר קודם לכן: Object. בהיררכיית השורש היחיד, כל דבר הוא Object. לכן, מכולה המחזיקה אובייקטים מטיפוס Object יכולה להכיל כל דבר. תכונה זו מקלה מאוד על השימוש החוזר במכולות.

כדי להשתמש במכולה כזו, אתה פשוט מוסיף אליה הפניות ל-Object, ומאוחר יותר, מבקש אותן בחזרה. אך מאחר שהמכולה מכילה רק אובייקטי Object, כשאתה מוסיף את הפניית האובייקט שלך למכולה, היא מומרת למעלה ל-Object, ובכך מאבדת את זהותה. כשתבקש אותה בחזרה, תקבל הפניה ל-Object, ולא הפניה לטיפוס שהכנסת פנימה. אם כן, כיצד תהפוך אותה בחזרה למשהו שיש לו הממשק השימושי של האובייקט שהכנסת למכולה?

כאן תשתמש שוב בהמרה, אלא שהפעם לא תמיר במעלה היררכיית ההורשה לסוג כללי יותר. הפעם תמיר במורד ההיררכיה אל סוג ייחודי יותר. המרה כזאת מכונה **המרה למטה** (downcasting). בהמרה למעלה, אתה יודע, למשל, שעיגול הוא סוג של צורה, ולכן בטוח להמיר למעלה, אך אינך יכול לדעת ש-Object הוא בהכרח עיגול, או אפילו צורה, ולכן אין זה בטוח כלל להמיר למטה, אלא אם כן ידוע לך עם מה בדיוק יש לך עסק.

עם זאת, אין זה מסוכן לחלוטין, כיוון שאם תמיר למטה לדבר הלא נכון, תקבל שגיאת זמן ריצה המכונה **חריגה** (exception), שתתואר בהמשך. למרות זאת, כשאתה שולף הפניות לאובייקטים ממכולה, צריכה להיות לך דרך כלשהי לזכור מהן בדיוק, כדי שתוכל לבצע את ההמרה המתאימה למטה.

המרה למטה ובדיקות בזמן ריצה דורשות זמן נוסף להרצת התוכנית ומאמץ נוסף מצד המתכנת. האם לא היה כדאי ליצור את המכולה כך שתדע בדרך כלשהי באילו סוגים היא מחזיקה, ובכך לבטל את הצורך בהמרה למטה ואת הסיכון שתתרחש טעות? הפתרון נמצא בטיפוסים **מבוססי פרמטרים** (parameterized types), שהם מחלקות

שהמהדר יכול להתאים אוטומטית לעבודה עם טיפוסים מוגדרים. לדוגמה, במקרה של מכולה מבוססת-פרמטרים, המהדר יוכל להתאים את המכולה כך שתקבל צורות בלבד ותחזיר צורות בלבד.

טיפוסים מבוססי פרמטרים הם חלק חשוב של שפת C++, בין היתר מכיוון של C++ אין היררכיית שורש יחיד. בשפת C++, מילת המפתח המממשת טיפוסים מבוססי פרמטרים היא template (תבנית מחלקה). בג'אווה אין עדיין טיפוסים מבוססי פרמטרים, כיוון ששפה זו מסוגלת להסתדר - אם גם בצורה מגושמת למדי - תוך שימוש בהיררכיית השורש היחיד. עם זאת, קיימת הצעה לטיפוסים מבוססי פרמטרים המתבססת על תחביר הדומה מאוד לתבניות המחלקה של C++.

בעיית משק הבית: מי צריך לנקות?

כל אובייקט זקוק למשאבים כדי להתקיים, ובעיקר לזיכרון. כשאובייקט אינו נחוץ יותר יש "לנקות" אותו כדי שמשאבים אלה ישוחררו לשימוש חוזר. במצבי תכנות פשוטים השאלה כיצד ינוקו אובייקטים אינה נשמעת תובענית במיוחד: אתה יוצר את האובייקט, משתמש בו למשך הזמן הדרוש לך, ואז עליך לפרק (להשמיד) אותו. עם זאת, לא קשה להיתקל במקרים בהם המצב מסובך יותר.

לדוגמה, נניח שאתה מתכנן מערכת לניהול תעבורה אווירית עבור נמל תעופה. (אותו מודל עשוי לעבוד גם עבור ניהול תיבות במחסן סחורות, או עבור מערכת להשכרת סרטי וידאו, או עבור אכסניה לחיות מחמד.) בתחילה זה נראה פשוט: צור מכולה להחזקת מטוסים, ולאחר מכן צור "מטוס" חדש עבור כל מטוס שנכנס למרחב בקרת הטיסה ומקם אותו בתוך המכולה. לניקוי, פשוט מחק את אובייקט המטוס המתאים כשהוא עוזב את המרחב האווירי.

אבל ייתכן שיש לך מערכת נוספת לרישום נתונים על המטוסים: אולי אלו נתונים שאינם דורשים תשומת לב מידית כמו פונקציית הבקר הראשית. אולי זהו רישום תוכניות הטיסה של כל המטוסים הקטנים שעוזבים את נמל התעופה. אם כך, יש לך מכולה שנייה של מטוסים קטנים, ובכל פעם שאתה יוצר אובייקט מטוס, אתה מוסיף אותו גם למכולה שנייה זו אם זהו אכן מטוס קטן. לאחר מכן, יש תהליך רקע כלשהו המבצע פעולות על האובייקטים במכולה זו ברגעי הפנאי של המערכת.

כעת הבעיה קשה יותר: כיצד תוכל לדעת בכלל מתי עליך לפרק את האובייקטים? כשסיימת לעבוד עם האובייקט, חלק אחר של המערכת אולי לא סיים עדיין. בעיה דומה עלולה להתעורר במספר רב של מצבים אחרים, ובמערכות תכנות שבהן עליך למחוק אובייקט במפורש ברגע שסיימת אתו (כמו בשפת C++), הדבר עלול להסתבך במידה ניכרת.

בשפת ג'אווה, מאסוף האשפה תוכנן לטיפול בבעיה של שחרור הזיכרון (אם כי הדבר אינו כולל היבטים אחרים של ניקוי אובייקט). מאסוף האשפה "יודע" מתי אובייקט אינו נמצא יותר בשימוש, ואז הוא משחרר את הזיכרון של אותו אובייקט באופן אוטומטי. תכונה זו (יחד עם העובדה שכל האובייקטים נגזרים ממחלקת השורש היחידה Object, ושניתן ליצור אובייקטים רק בדרך אחת - בערימה), הופכת את התהליך של תכנות

בג'אווה לפשוט בהרבה מהתכנות ב- C++. יש לך מספר קטן בהרבה של החלטות להחליט ומשוכות לעבור.

מאספי אשפה לעומת יעילות וגמישות

אם הרעיון הזה כל-כך מוצלח, מדוע לא נעשה אותו הדבר ב- C++? ובכן, כמובן שעליך לשלם מחיר עבור כל הנוחות הזו בתכנות, והמחיר הוא עלויות בזמן-ריצה. כפי שהוזכר קודם לכן, בשפת C++ אתה יכול ליצור אובייקטים במחסנית (stack), ובמקרה זה הם מנוקים באופן אוטומטי (אך נשללת ממך הגמישות ליצור כמה מהם שתמצה בזמן-ריצה). יצירת אובייקטים במחסנית היא הדרך היעילה ביותר להקצות שטח אחסון לאובייקטים ולשחרר שטח אחסון זה. יצירת אובייקטים בערימה (heap) עלולה להיות יקרה בהרבה. גם האילוץ לגזור תמיד ממחלקת בסיס כלשהי והפיכת כל הקריאות לפונקציות לרב-צורתיות תובעים מחיר מסוים. אך מאסף האשפה הוא בעיה מיוחדת, כיוון שלעולם אינך יודע מתי בדיוק הוא יופעל וכמה זמן יידרש לכך. פירושו של דבר שקיים חוסר עקביות במהירות הביצוע של תוכנית ג'אווה, ולכן לא תוכל להשתמש בה במצבים מסוימים, כמו למשל כשמהירות הביצוע של התוכנית היא קריטית באופן רציף. (תוכניות כאלו נקראות בדרך כלל תוכניות זמן אמת - real time - למרות שלא כל בעיות התכנות בזמן אמת חמורות כל-כך).

מתכנני שפת C++, בניסיון לחזר אחרי מתכנני C (ובהצלחה מרובה, יש לציין), לא רצו להוסיף לשפה מאפיינים שיגבילו את המהירות, או את השימוש ב- C++ בכל מצב שבו המתכנתים היו יכולים לבחור במקומה בשפת C. מטרה זו הושגה, אך במחיר של מורכבות רבה יותר בעת התכנות ב- C++. ג'אווה פשוטה יותר מ- C++, אך זאת בתמורה ליעילות ולעתים גם לאפשרויות היישום. עם זאת, עבור החלק המשמעותי של בעיות התכנות, ג'אווה היא הבחירה העדיפה.