

דפי עבודה ללימוד סביבת Eclipse

סיון טולדו, אוניברסיטת תל־אביב

© כל הזכויות שמורות לסיון טולדו, 2004.

דפי העבודה האלה הוכנו תוך הסתייעות ב-Eclipse Innovation Grant, שהוענק על ידי חברת יבמ.

דפי העבודה הללו מיועדים לסייע לך ללמוד להשתמש באקליפס (Eclipse), סביבה לפיתוח תוכנה. הגרעין של אקליפס תומך בשתי יכולות מרכזיות: ניהול קבצים, ושילוב כלי פיתוח, כגון מהדרים (compilers), מנפים (debuggers), ועוד כלים מסוגים רבים. כלומר, אקליפס הוא מעין מנהל קבצים מתוחכם ביחד שמכיל "נקודות עגינה" שאליהן ניתן לחבר כלי פיתוח נוספים. התצורה הבסיסית של אקליפס כוללת את כלי הפיתוח לשפת ג'אווה, אם כי יש גם תצורות של אקליפס ללא כלי פיתוח לג'אווה ויש תצורות עם כלים רבים נוספים.

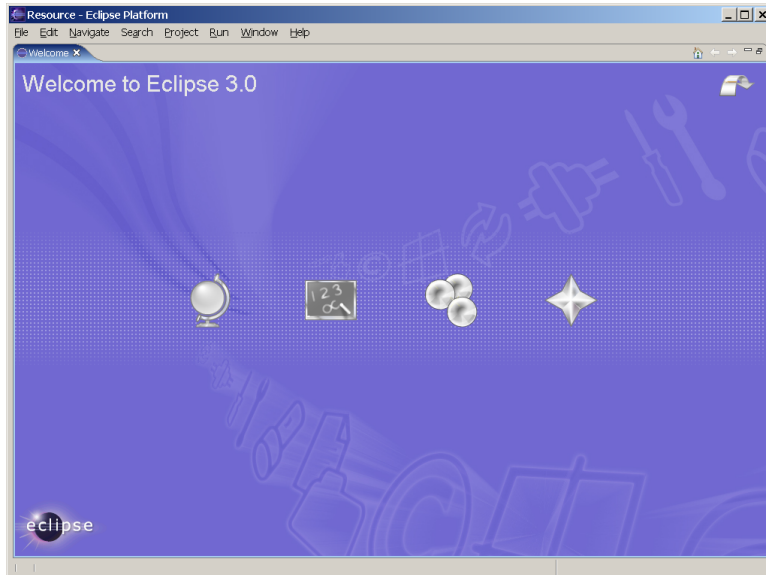
הדפים הללו מחולקים לשיעורים. מומלץ לעבור על השיעורים לפי הסדר, ומומלץ לעקוב אחרי כל שיעור על ידי ביצוע בפועל של המטלה המוצגת באותו שיעור. ישנם שיעורים שניתן לדלג עליהם ללא יצירת חסר שיפריע בשיעורים הבאים. בשיעורים כאלה, העובדה שניתן לדלג עליהם מצוינת בתחילת השיעור.

תמונות המסך שמובאות בדפי העבודה צולמו מתוך גרסה 3.0 של אקליפס. בגרסאות אחרות, המסכים עשויים להיראות שונה מעט. למשל, תפריטים עשויים להכיל מעט יותר או מעט פחות פריטים, או שאשפים עשויים להכיל דיאלוג נוסף, וכדומה. אבל עקרונות השימוש המוצגים בדפים נכונים גם לגרסאות אחרות של אקליפס. סביבת אקליפס רצה על מגוון של מערכות הפעלה: חלונות, לינוקס ויוניקס, ומערכת ההפעלה של המקינטוש. תמונות המסך צולמו במחשב שמריץ חלונות. תחת מערכות הפעלה אחרות, המראה של המסכים עשוי להיראות שונה מעט, וצורת האינטראקציה עם הכלי עשויה להיות שונה מעט. במקינטוש, למשל, יש לעכבר הסטנדרטי רק כפתור אחד, ולכן אקליפס מפרש הקשה על כפתור+עכבר כהקלקה על עכבר ימני. אבל מעבר להבדלים קוסמטיים כאלה, אין הבדל בין גרסאות אקליפס שרצות על מערכות הפעלה שונות.

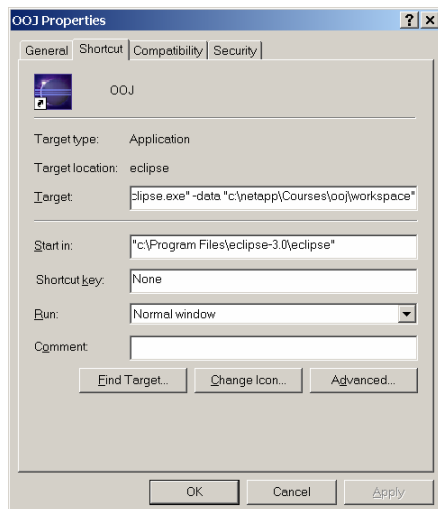
אפשר להשתמש באקליפס גם בלי להבין את כל היכולות שלה, ובלי לדעת מה משמעות כל פרט בממשק הגרפי שלה. אקליפס, כמו תוכנות רבות אחרות, היא סביבת עבודה מורכבת מאוד, שלימוד יסודי שלה דורש השקעה רבה. אולם על מנת להפיק מאקליפס תועלת, אין צורך לדעת להשתמש בכל היכולות שלה. גם אני אני מכיר את כל היכולות של אקליפס, אבל אני מפיק ממנה תועלת רבה. הדרך הטובה ביותר ללמוד להשתמש באקליפס היא ללמוד לבצע את מטלות הפיתוח העיקריות בצורה פשוטה. לאחר רכישת מיומנויות פיתוח בסיסיות, כדאי לחקור את הסביבה באופן יותר יסודי, אם על ידי שימוש בדפי עבודה מתקדמים, אם על ידי קריאת התיעוד המקוון (help), או על ידי קריאת ספרים או מאמרים אודות אקליפס (האתר www.eclipse.org מכיל מאמרים מתקדמים רבים). דרך טובה נוספת ללמוד להשתמש ביכולות מתקדמות היא על ידי ניסוי, כלומר פשוט על ידי בחירת אופציות מתפריטים, סרגלי כלים, ודיאלוגים, והתבוננות בתוצאות. מכל מקום, אין צורך להירתע משימוש בכלי בגלל שלא מבינים מה התפקיד של כל פרט בממשק.

מרחב העבודה (workspace) של אקליפס

על מנת להפעיל את Eclipse, יש להקליק על הצלמית של הכלי, שמופיעה מימין. בלינוקס או יוניקס יתכן שלא תוכלו להפעיל את הכלי על ידי הקלקה על צלמית, אלא יהיה צורך לתת את הפקודה eclipse בתוכנת מעטפת (shell). המסך הראשון שיעלה יראה בערך כמו המסך הבא, והצלמיות שבמרכזו מאפשרות לכם ללמוד אודות הכלי. צלמית החץ בצד הימני העליון תעביר אתכם למסך הראשי של הכלי.



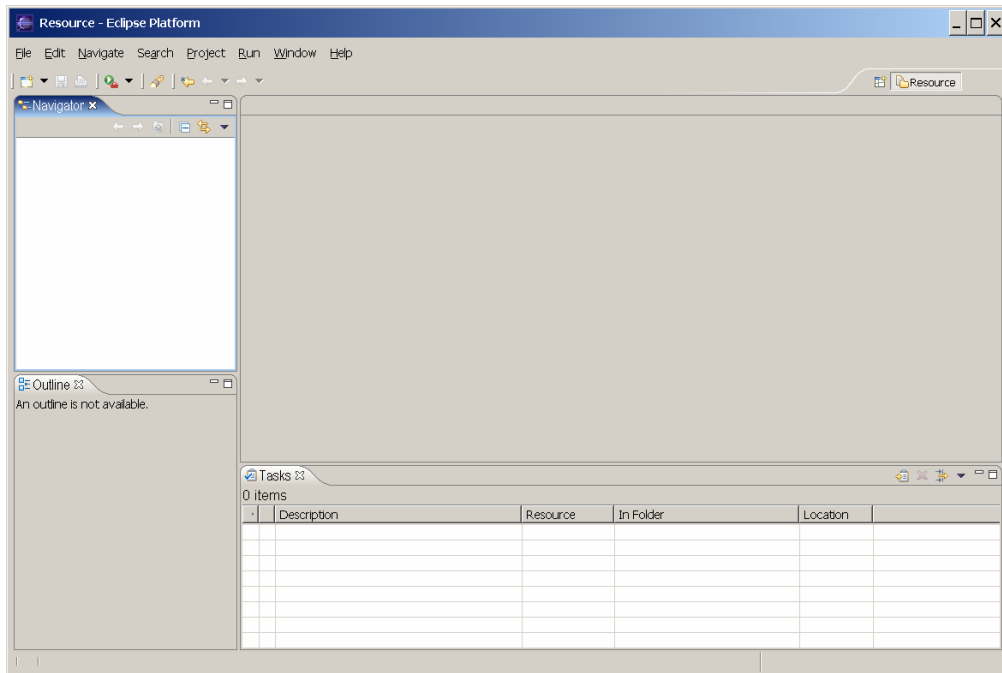
המסך הזה מופיע כאשר משתמשים באקליפס בפעם הראשונה, או ליתר דיוק, כאשר מרחב העבודה (workspace) שרוצים לעבוד איתו עדיין לא קיים.



כאמור, אקליפס הוא במידה רבה כלי לניהול קבצים, קבצים שהם חלק מפרויקטים של פיתוח תוכנה. אקליפס מנהל קבצים שנמצאים תחת ספרייה אחת, שנקראת מרחב העבודה (workspace). כאשר מפעילים את הכלי, בוחרים מרחב עבודה. בהחלט אפשר לעבוד על פרויקטים שונים במרחבי עבודה שונים. הדרך הקלה ביותר לבחור סביבת עבודה היא בעזרת הדגל -data לתוכנית eclipse שמפעילה את הכלי. בדרך כלל, התוכנית מופעלת על ידי קיצור דרך, ובקיצור הדרך ניתן לתת את הדגל הזה, כך שקיצורי דרך שונים יפעילו את אקליפס על מרחבי עבודה שונים. הדיאלוג

משמאל מראה כיצד קיצור הדרך גורם להפעלת הכלי על מרחב עבודה מסוים.

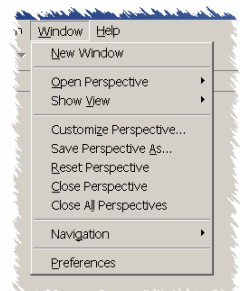
מרחב העבודה הוא של מפתח בודד, גם בפרויקטים שבהם מספר מפתחים עובדים על אותו קוד. שיתוף קבצים בין מפתחים לא מתבצע בדרך כלל על ידי שיתוף קבצים, אלא על ידי סנכרון מרחב העבודה של כל מפתח עם מאגר קבצים מרכזי לפרויקט. אי לכך, מרחב העבודה שלך הוא שלך בלבד, ואת או אתה יכולים לארגן אותו כרצונכם.



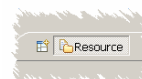
הקליקו על צלמית החץ במסך הפתיחה, ועברו למסך הראשי, שצריך להיראות כמו המסך שבראש העמוד.

גם במצב כזה, כאשר מרחב העבודה ריק, המסך עמוס למדי. החלק העליון של המסך מוקדש לתפריט ראשי, וממש מתחתיו סרגל כלים. החלק שמתחת לסרגל מחולק למלבנים, שבכל אחד מהם מופיע פריט ממשק אחד או יותר. המסך שלמעלה מחולק לארבעה מלבנים. הימני העליון ריק, הימני התחתון מוקדש לפריט בשם Tasks, השמאלי העליון לפריט בשם Navigator, והשמאלי התחתון ל-Outline. החלוקה הזו, ותוכן הפריטים שמופיעים בכל חלק של המסך, נקראת באקליפס פרספקטיבה (perspective).

כאשר מפתחים תוכנה, נוח להשתמש בפרספקטיבות שונות בזמן שמבצעים פעולות שונות. למשל, כאשר מפתחים קוד חדש יש צורך בכלים שונים מאלו הדרושים בזמן ניפוי קוד קיים. היכולת לעבור במהירות, ולפעמים באופן אוטומטי, בין פרספקטיבות מקלה על המפתח. אתם יכולים להשתמש בפרספקטיבות שמוגדרות מראש, להגדיר פרספקטיבות חדשות, או לשנות את התצורה של פרספקטיבות קיימות כך שיתאימו לצרכים והרגלי העבודה שלכם. יצירת ושינוי פרספקטיבות מתבצעות דרך איבר ה-window בתפריט הראשי.

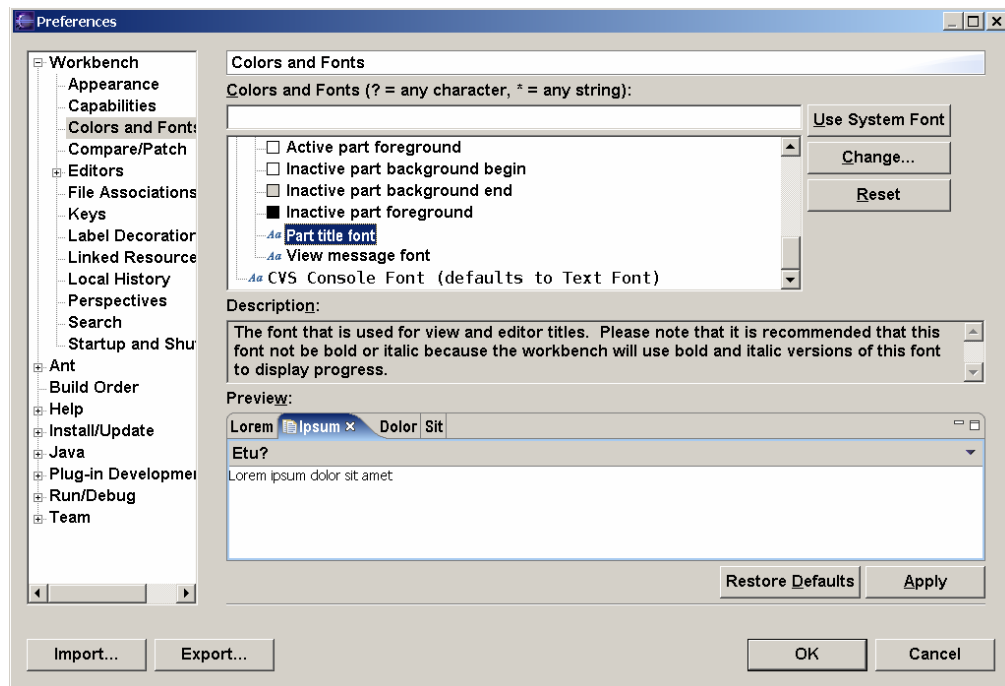


הפרספקטיבה שנפתחה באופן אוטומטי עבור מרחב העבודה החדש נקראת פרספקטיבת resource, והיא הפרספקטיבה היחידה שהיא חלק מאקליפס עצמה; כל שאר הפרספקטיבות מגיעות עם כלי פיתוח שמתממשים לאקליפס. את שם הפרספקטיבה שנמצאת כרגע בשימוש ניתן לראות בצד ימין למעלה של המסך. הקלקה על הצלמית שמשמאל לשם הפרספקטיבה מאפשרת להחליף פרספקטיבה באופן ידני (כאמור, פעולות מסוימות, כמו מעבר מפיתוח לניפוי, יכולות להחליף פרספקטיבה באופן אוטומטי).



לפני שנמשיך, כדאי לציין שניתן לשלוט במראה ובהתנהגות של סביבת העבודה דרך דיאלוג ההעדפות. באקליפס יש דיאלוג אחד כזה, שמוצג בעמוד הבא, ושניתן להגיע אליו מהתפריט הראשי דרך window ולאחר מכן preferences. כאשר אתם

מתחילים לעבוד עם אקליפס, עדיף להשאיר את רוב ההעדפות במצב ברירת המחדל שלהן, פרט אולי לגודל של גופנים, שכדאי להגדיל אם הטקסט קטן מדי עבורכם. אבל בהמשך, כדאי לבקר מדי פעם בדיאלוג הזה ולהתאים את התנהגות הכלי להעדפות האישיות שלכם.



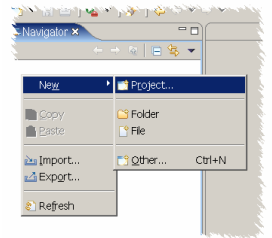
דיאלוג ההעדפות מאפשר לשלוט בהתנהגות של כל מרכיבי אקליפס. בדיאלוג שמוצג המשתמש משנה את התנהגות הממשק הגרפי של אקליפס, שנקרא ה-workbench.

כפי שניתן לראות, ניתן לשמור העדפות לקובץ (Export), ניתן לקרוא העדפות מקובץ (למשל העדפות של משתמש אחר, או העדפות שלנו ממרחב עבודה אחר. ניתן גם לשחזר את ההעדפות המקוריות (Restore Defaults).

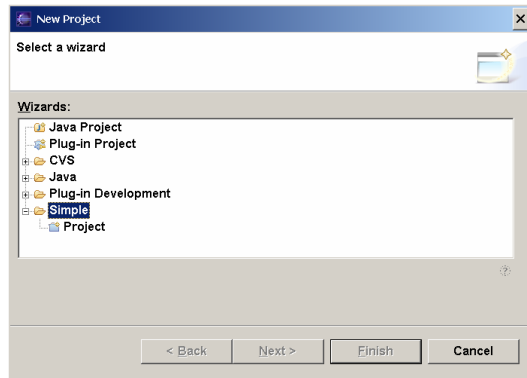
יצירת פרויקט פשוט

בשיעור זה נלמד ליצור פרויקט חדש. מרחב העבודה של אקליפס מחולק לפרויקטים, שכל אחד מהם הוא צביר של קבצים ומדריכים (folders או directories). המרכיבים השונים של אקליפס תומכים בפרויקטים מסוגים שונים, כמו למשל פרויקטים של תוכניות ג'אווה. בשיעור הזה נלמד ליצור פרויקט פשוט שתומך בניהול קבצים, אבל לא בשפת תכנות או בתוסף של אקליפס. השיעור הזה עובר על נושאים בסיסיים רבים בממשק של אקליפס, ולכן לא כדאי לדלג עליו.

על מנת ליצור את הפרויקט, יש להקליק עכבר ימני על רכיב ה-Navigator, שמוצג בדרך כלל בחלק השמאלי העליון של המסך. ההקלקה הזו מציגה תפריט שנקרא תפריט הקשר (context menu). מתוך התפריט יש לבחור New ואחר כך Project. אפשר לבחור בתפריטים הללו על ידי הקלקה, או על ידי הקלדת האות שמודגשת בקו תחתון, למשל w ב-New.



כעת יפתח אשף יצירת פרויקט חדש, שהמסך הראשון שלו מוצג משמאל. במסך הזה

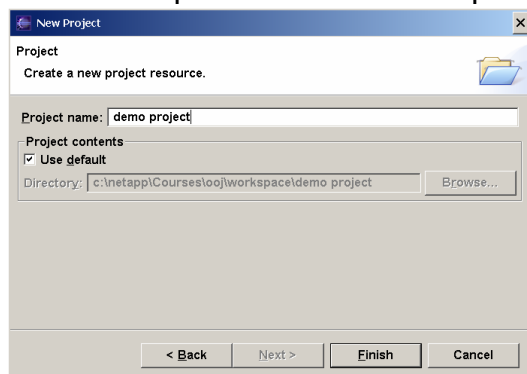


יש לבחור מתוך הקטגוריה Simple את האפשרות היחידה, Project. כאשר האשף נפתח, הקטגוריות סגורות, וסוגי הפרויקטים בכל קטגוריה לא מוצגים. על מנת לפתוח קטגוריה, יש או להקליק על סימן ה-+ שמשמאלה, או לבחור אותה ולהקליד Enter. ניתן לעלות ולרדת ברשימת הקטגוריות וסוגי הפרויקטים בעזרת מקשי החיצים, או על ידי הקלקה בעכבר. את בחירת סוג

הפרויקט יש לבצע על ידי הקלקה בעכבר על הסוג או על Next, או לחיצת Enter כאשר הסוג, Project, בחור (הבחירה מסומנת על ידי רקע כחול, כמו הרקע של הקטגוריה Simple בצילום האשף). הכפתור Next יהיה פעיל אך ורק כאשר הבחירה במסך היא של סוג פרויקט ולא של קטגוריה. כדאי לשחק מעט עם הדיאלוג על מנת להבין את צורת האינטראקציה עם דיאלוגים באקליפס.

היכולת לשלוט בתפריטים ובאשפים גם על ידי המקלדת, ולא רק בעזרת העכבר, משותפת לכל הממשקים של אקליפס. היכולת הזו מאפשרת לאנשים עם מוגבלויות פיזיות שמונעות שימוש בעכבר להשתמש באקליפס. היכולת הזו גם מאפשרת למשתמשים מיומנים להפעיל את אקליפס באופן מהיר ויעיל יותר מכפי שאפשר בעזרת העכבר.

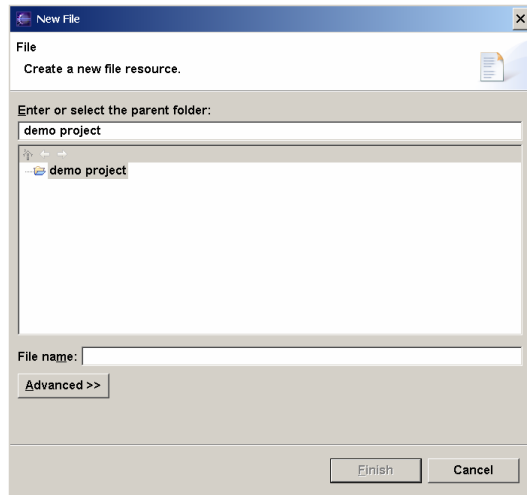
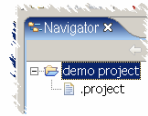
המסך הבא באשף מאפשר לקבוע שם לפרויקט החדש שיוצור. יש להקליד שם בשדה



המיועד לכך, ואז להקליק Finish. כפי שאפשר לראות, המסך מאפשר לקבוע היכן ישמרו הקבצים של הפרויקט. באקליפס, פרויקט הוא מיכל לקבצים. המיכל ממומש כמדריך. ברירת המחדל למיקום היא בתוך המדריך של מרחב העבודה, וברירת המחדל לשם המדריך היא כשם הפרויקט. בפרט, ניתן למקם

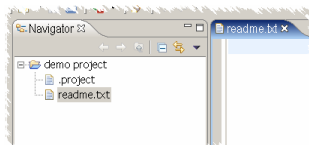
את הקבצים של פרויקט מחוץ למדריך של מרחב העבודה.

לאחר הלחיצה על Finish, סגור, ברכיב עם הכותרת Navigator, שהוא סייר הקבצים של אקליפס. אם נפתח אותו, נראה קובץ בודד, project, שמתאר עבור אקליפס עצמה את המצב הנוכחי של הפרויקט. בדרך כלל, אין צורך לערוך ישירות את הקובץ הזה ועדיף להניח לאקליפס לנהל אותו. ניתן גם להסתיר אותו, אבל לא נטרח בכך עכשיו.



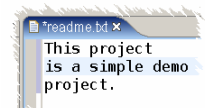
הבה נוסיף קובץ טקסט לפרויקט. נפעיל את תפריט ההקשר של הפרויקט על ידי הקלקה ימנית על שם הפרויקט ב-Navigator, ומהתפריט שיפתח נבחר New ו-File. הבחירה הזו תגרום להופעת אשף להוספת קובץ. במסך של האשף נקליד שם, למשל readme.txt, ונסיים. אקליפס מזהה את סוג הקובץ וצורת הטיפול בו על פי הסיומת שלו, כמו מערכת ההפעלה חלונות, ולכן כדאי לבחור שם עם סיומת שמתאימה לסוג הקובץ. הכפתור Advanced במסך הזה

מאפשר לקשר את הקובץ לקובץ קיים במקום כלשהו במערכת הקבצים. כלומר, למרות שהפרויקט הוא מיכל לקבצים, אין חובה למקם את כל הקבצים של הפרויקט בתוך המדריך של הפרויקט. אבל שמירת כל הקבצים בתוך המדריך של הפרויקט היא פשוטה ונוחה, ומונעת מקבצים ללכת לאיבוד, ולכן כדאי להשתמש כאן בברירת המחדל.



לאחר שהקובץ החדש נוצר, הוא מופיע ב-Navigator, אבל זה לא השינוי היחיד בממשק. בנוסף, אקליפס פותחת את הקובץ בעורך טקסטים (editor), שמופיע בחלק הימני העליון של המסך, מימין ל-Navigator. ניתן להתחיל להקליד טקסט לתוך הקובץ בעורך הזה.

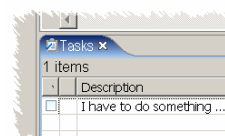
כאשר נתחיל להקליד טקסט בעורך, הכותרת של העורך תשתנה מעט, וכוכבית תופיע משמאל לשם הקובץ, להזכיר לנו שהקובץ שונה אבל לא נשמר עדיין.



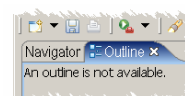
זה הזמן להסביר על המרכיבים השונים שמופיעים בממשק של אקליפס. סוג אחד של רכיבים, שכבר ראינו, הוא **תפריטים**. ראינו את התפריט הראשי ותפריטי הקשר. באקליפס יש עוד סוגי תפריטים שנפגוש בהמשך. תפריטים מיועדים למתן פקודות; חלק מהפקודות המופיעות בתפריט ניתן לתת גם על ידי צירופי מקשים, וחלק ניתן לתת על ידי הקשה על צלמיות שמופיעות ב**סרגלי כלים**. יש לאקליפס סרגל כלים ראשי מתחת לתפריט הראשי, אבל יש גם סרגלי כלים קטנים לרכיבים של אקליפס, כמו סרגל הכלים הקטן של ה-Navigator. כאשר משאירים את הסמן על צלמית למספר שניות, מופיע חלון עזרה קטן שמסביר מה הפעולה שהצלמית מפעילה. שני סוגים נוספים של רכיבי ממשק שכבר פגשנו הם **אשפים**, ליצירת עצמים חדשים, ו**דיאלוגים**, לשינוי תכונות של עצמים קיימים. ראינו את דיאלוג ההעדפות, ולכל פרויקט, מדריך וקובץ יש דיאלוג תכונות, שניתן להציג על ידי בחירת Properties מתוך תפריט ההקשר המתאים.

שני סוגים חשובים של רכיבי ממשק שאנו למעשה רואים כעת בפעם הראשונה הם עורכים ומבטים. **עורכים** משמשים, כפי ששמו מלמד, לעריכת קבצים. העורך הנפוץ ביותר הוא עורך טקסטים, אבל יש באקליפס גם עורכים אחרים. עורך נפתח כאשר מקליקים על קובץ ב-Navigator. כאשר משנים קובץ בעורך, הקובץ לא משתנה באופן אוטומטי במערכת הקבצים; יש לשמור אותו באופן מפורש, על ידי בחירת Save בתפריט הקבצים (File) או הקשה על Control-S. כאשר פותחים מספר קבצים בו זמנית, לכל אחד נפתח עורך משלו, אבל כל העורכים הללו נערים זה על גבי זה באותו חלק של המסך. התוויות של כל העורכים עם שמות הקבצים מציצות בחלק העליון של מלבן העורכים וניתן לבחור עורך על ידי הקלקה על אחת התוויות. ניתן לסגור עורך על ידי הקלקה על סימן ה-x שליד שם הקובץ, או על ידי בחירת Close בתפריט הקבצים.

מבטים (views) הם רכיבים שמאפשרים לראות היבט מסוים של קובץ, פרויקט, או ישות אחרת. באקליפס יש מגוון גדול מאוד של מבטים. בפרספקטיבת ה-Resource שנפתחה עבורנו, מופיעים שלושה, Navigator, Outline, ו-Tasks (מטלות), כל אחד בפנל נפרד. המבט שעבדנו איתו כבר, Navigator, הוא מבט על כל מרחב העבודה. המבט Outline הוא מבט שמסכם קובץ בודד, אבל אינו מציג מאומה עבור קבצי טקסט. המבט Tasks מציג מטלות שיש לבצע בפרויקט מסוים. אחד ההבדלים החשובים בין מבט ועורך, פרט לכך שמבטים בדרך כלל מובנים יותר, הוא ששינויים במבטים נשמרים מיד במערכת הקבצים, ואין צורך להפעיל פעולת Save כדי לשמור את השינוי. נסו, למשל, להוסיף מטלה למבט המטלות על ידי הפעלת תפריט ההקשר שלו (בהקלקה ימנית) ובחירת Add Task, או על ידי לחיצה על הצלמית המסומנת ב-+ בסרגל הכלים של המבט. בדיאלוג שיופיע תוכלו להקליד טקסט לתיאור המטלה ולבחור את הדחיפות שלה. לאחר הקלקה על OK בדיאלוג וחזרה למסך הראשי, המטלה החדשה תופיע במבט. המטלה החדשה כבר שמורה בקובץ. אין צורך לבחור Save ואם נצא מאקליפס ונכנס חזרה, המטלה לא תאבד. ניתן להסיר את המטלה על ידי בחירתה והקלקה על צלמית ה-x האדומה בסרגל הכלים של המבט או על ידי הפעלת תפריט ההקשר של המטלה ובחירת Delete. גם מחיקה היא שינוי שמתבטא מייד במערכת הקבצים.



המיקום של מבטים ועורכים במסך אינו קבוע, וניתן לגרור אותם. גרירה מתבצעת על ידי משיכת התווית של העורך או המבט. גרירה של מבט או עורך לכיוון אחד מקצוות השטח המוקצה להם מאפשר להעביר את הרכיב לאותו קצה של השטח. ניתן גם לערום מבטים זה על גבי זה על ידי גרירתם לאזור שבו יש כבר מבט. נסו, למשל, לגרור את המבט Outline כך שיערם יחד עם Navigator. גררו את התווית שלו כך שסימן הגרירה יהיה ליד התווית של Navigator, ועזבו. שני המבטים יתפסו כעת את אותו מקום במסך. ערימה כזו מציגה פחות מבטים בו זמנית, אבל משאירה יותר מקום לכל מבט, והיא שימושית כאשר יש מספר גדול של מבטים פתוחים, או כאשר עובדים עם מסך קטן.

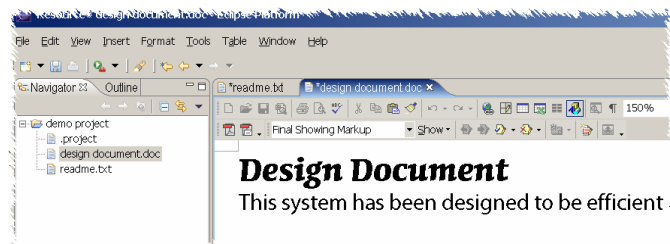


ניתן לסגור מבטים על ידי הקלקה על סימן ה-x שבצד ימין של התווית שלהם, וניתן לפתוח מבטים בעזרת Window->Show View בתפריט הראשי.

בחלק האחרון של השיעור, נלמד כיצד סביבת אקליפס מתמודדת עם ניהול ועריכת קבצים שאינם קבצי טקסט ושאינם באקליפס עצמה עורך מתאים עבורם. על החלק הזה ניתן לדלג בלי לפגוע בהבנת השיעורים הבאים.

ראשית, ניצור קובץ חדש מחוץ לאקליפס, למשל על שולחן העבודה או במדריך כלשהו. אם אתם מתרגלים את השיעור הזה על מחשב חלונות ואם מותקנת במחשב אחת

מתוכנות Microsoft Office, כדאי ליצור קובץ Office Word, למשל קובץ Word. כעת גרו את הקובץ החדש לאקליפס, והטילו אותו לתוך הפרויקט (סימן הגרירה משתנה כאשר הוא מעל לפרויקט או מדריך על מנת לסמן שניתן להטיל את הקובץ). יועתק למרחב העבודה (אם רוצים לקשר קובץ במקומו הנוכחי, יש להשתמש באשף יצירת

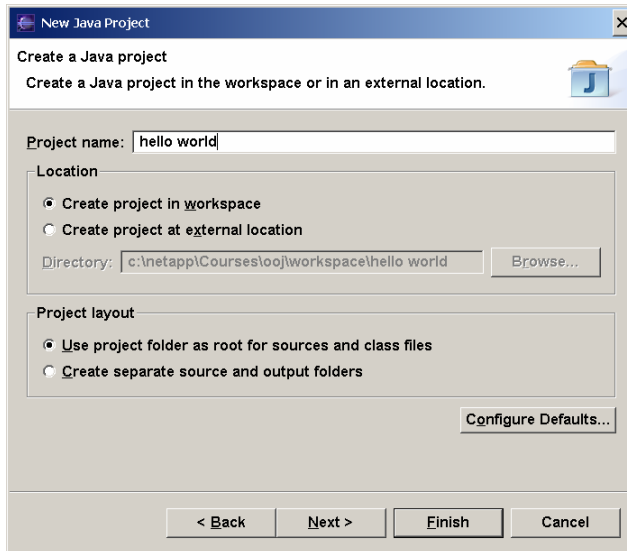


הקבצים) ויופיע בתוך הפרויקט. כעת הקליקו על שם הקובץ ב-Navigator. הקובץ יפתח בעורך בתוך אקליפס, אך התבוננות קצרה בעורך תגלה שהעורך הוא תוכנת Word עצמה,

אלא שהיא מופיע כחלק מהמשק הגרפי של אקליפס. אפילו התפריט הראשי התחלף לתפריט של Word. ברגע שעוברים להשתמש בחלק אחר של המשק של אקליפס ולא בעורך הזה, התפריט הראשי חוזר להיות התפריט של אקליפס. כעת שנו מעט את הקובץ, ושימרו את העותק העדכני. הקובץ ישמר, אבל הכוכבית לא תיעלם מהתווית שלו, משום שאקליפס אינה מודעת לכך שהקובץ נשמר; השמירה התבצעה למעשה על ידי תוכנת Word ולא על ידי אקליפס. כאשר נסגור את העורך, אקליפס תציע לשמור את הקובץ, כאילו שלא נשמר עדיין. לא כל עורך ניתן לשילוב באקליפס בצורה כזו.

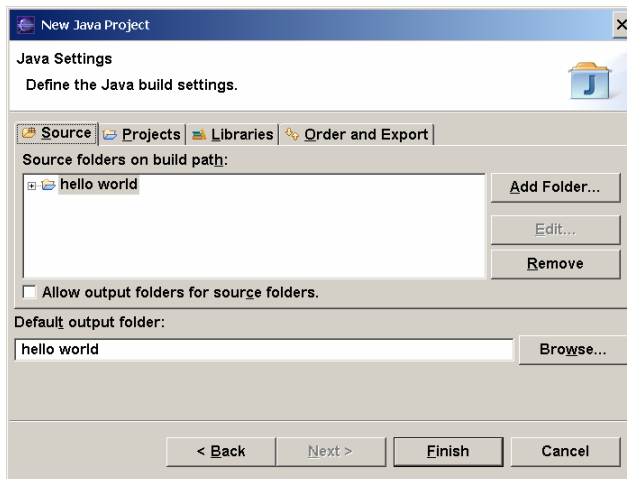
יש עוד שתי דרכים לערוך קובץ כזה מתוך המשק של אקליפס. הפעילו את תפריט ההקשר של הקובץ ובחרו Open With. בתפריט הקטן שיפתח יש ארבע אפשרויות. האפשרות הראשונה עורך טקסט (Text Editor), שיציג את הקובץ בעורך טקסט רגיל. לקבצים בינריים, כגון קובץ Word, זו לא אפשרות מועילה במיוחד, אבל כדאי לנסות אותה. האפשרות השנייה, העורך של מערכת ההפעלה (System Editor), מפעילה את התוכנה שאמורה לטפל בסוג הקובץ, על פי הסיומת של שמו, בתצורה הנוכחית של מערכת ההפעלה. לקבצים שאינם קבצי טקסט, זו בדרך כלל האפשרות הנוחה ביותר. התוכנה נפתחת באופן עצמאי, ללא קשר לאקליפס. האפשרות השלישית, In-Place Editor היא זו שראינו ראשונה: הקובץ נפתח בעורך הרגיל שלו, שאינו חלק מאקליפס, אבל כחלק מהמשק הגרפי של אקליפס. לא כל קובץ ניתן לפתוח כך, וזה לא תמיד נוח, אבל זה מונע את הצורך במעבר בין חלונות רבים של מערכת ההפעלה. האפשרות הרביעית, עורך ברירת המחדל, מפעילה פשוט את סוג העורך האחרון שהפעלנו על הקובץ.

יצירת פרוייקט ג'אווה



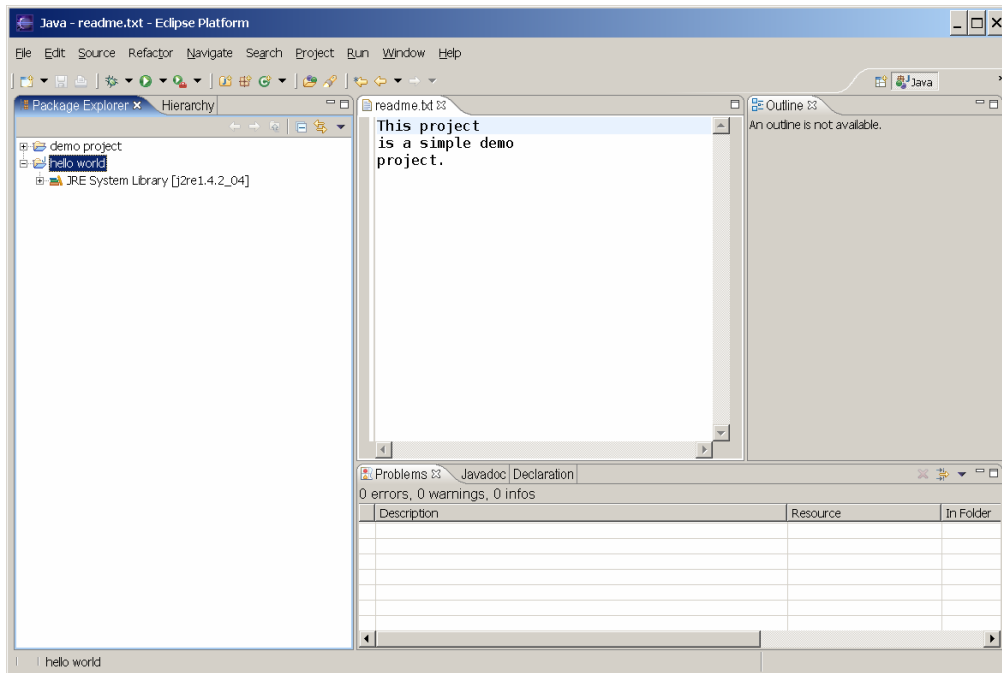
בשיעור זה נלמד ליצור פרויקט ג'אווה. מתפריט ההקשר של Navigator או מהתפריט הקבצים הראשי בחרו New Project, ובמסך הראשון של אשף יצירת הפרויקטים בחרו Java Project מתוך הקטגוריה Java. המסך הבא באשף מאפשר לקבוע את שם הפרויקט, את מיקומו, והאם הקבצים הבינריים שהקומפיילר ייצר ישמרו ביחד עם קבצי קוד המקור, או בנפרד. תנו שם לפרוייקט

והמשיכו למסך הבא על ידי לחיצה על Next. לחיצה על Finish תבחר למעשה ברירת מחדל עבור כל האפשרויות במסך הבא. כדאי לראות מה מציע המסך הבא, ולכן לחצו על Next, לא על Finish.



המסך הבא והאחרון באשף מאפשר לקבוע את תצורת הפרויקט. המסך מחולק לפנלים שניתן לבחור על ידי תוויות. התווית הראשונה קובעת היכן נמצאים קבצי קוד המקור. התווית השנייה קובעת האם הפרוייקט הזה תלוי בפרוייקטים אחרים במרחב העבודה. למשל, במרחב עבודה שבו יש פרויקט של ספריה אלגוריתמית (הכוונה לספרית מחלקות)

ושגרות, לא למדריך במערכת הקבצים) ופרוייקט של תוכנית גרפית שמשתמשת בספריה הזו, הפרוייקט של התוכנית צריך להיות תלוי בפרוייקט הספריה, כדי ששינויים בספריה ישפיעו על התוכנית הגרפית. התווית הבאה מאפשרת לקבוע באיזו ספריות ג'אווה הפרוייקט ישתמש (כלומר, ספריות קיימות שאינן חלק מפרוייקט במרחב העבודה). בתצורת ברירת המחדל, הפרוייקט תלוי אך ורק בספריות הסטנדרטיות של השפה. התווית האחרונה מאפשרת לקבוע את סדר השימוש בספריות חיצוניות, ואת זהות הספריות שהפרוייקט מייצא לפרוייקטים אחרים במרחב העבודה. בנוסף לתוויות הללו, המסך מאפשר לקבוע מיקום לקבצים הבינריים שיוצרו. השאירו את כל הערכים כפי שהם ולחצו על Finish. אם נרצה בהמשך לשנות את תצורת הפרוייקט, נוכל לעשות זאת מתוך דיאלוג התכונות (Properties) של הפרוייקט.

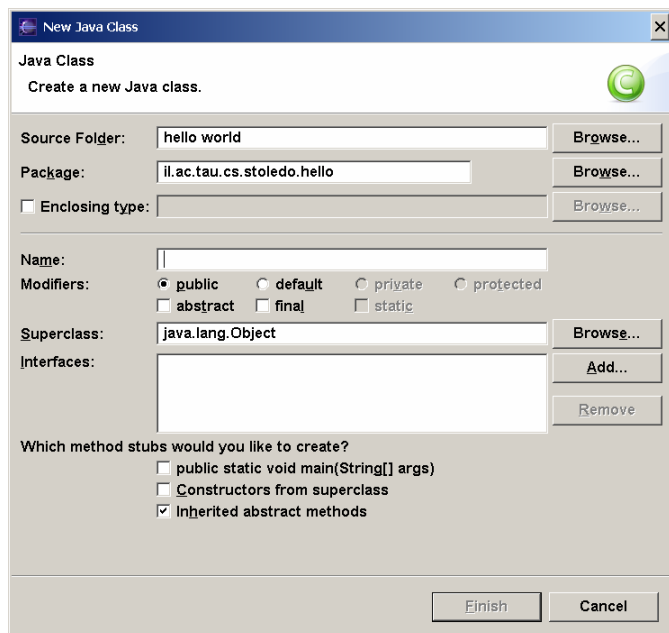


לפני שאקליפס מחזירה אותנו מהאשף למסך הראשי, היא מציע לנו לעבור לפרספקטיבת ג'אווה. כדאי לעבור, וכדאי גם לבחור בדיאלוג באפשרות למעבר אוטומטי לפרספקטיבת ג'אווה. אם תבחרו באפשרות הזו, כל מעבר לפרויקט ג'אווה בעתיד יעביר אתכם אוטומטית לפרספקטיבה הזו, שתראה בערך כמו המסך בראש העמוד.

הפרספקטיבה הזו יותר מורכבת מפרספקטיבת ה-Resource שראינו בשיעור הקודם. בצד שמאל של המסך יש שני מבטים ערומים זה על גבי זה. המבט העליון נקרא Package Explorer, והוא מחליף למעשה את ה-Navigator. המבט הזה מאפשר עבודה פשוטה יותר מה-Navigator עם קבצי קוד ג'אווה, שצריכים להיות מקוננים בדרך כלל בתוך עץ מדריכים עמוק. במבט הזה, בניגוד ל-Navigator, הקבצים מסודרים על פי מרחב השמות של המחלקות בשפת התכנות, ולא על פי עץ המדריכים. מתחתיו מופיע מבט Hierarchy, שהוא כרגע ריק. בצד ימין מופיע מבט Outline, שכבר ראינו, וביניהם אזור לעורכים. הקובץ שפתחנו בשיעור הקיים עדיין פתוח בעורך; המעבר לפרויקט אחר לא סוגר קבצים, ועורכים של קבצים ממספר פרויקטים יכולים להיות פתוחים בו זמנית. כדאי לסגור את העורך הזה כעת. בחלק התחתון של המסך מופיעים עוד שלושה מבטים זה על גבי זה. העליון, Problems, מראה בעיות קומפילציה, ובאחרים נדון בהמשך. כולם כרגע ריקים.

בג'אווה מחלקות שייכות לחבילות (packages) שמהוות בעיקר חלוקה של מרחב השמות. נתחיל את הפרויקט ביצירת חבילה חדשה. שמות של חבילות הם בדרך כלל היררכיים, ולכן אני אבחר בשם `il.ac.tau.cs.stoledo.hello`, שמורכב משם האתר שאני שייך אליו, `cs.tau.ac.il`, משם המשתמש שלי, `stoledo`, ומשם החבילה הזו, `hello`. מתוך תפריט ההקשר של הפרויקט נבחר `New Package`. באשף שיפתח נקליד את שם החבילה, ו-`Finish`. החבילה החדשה תופיע ב-`Package Explorer`.

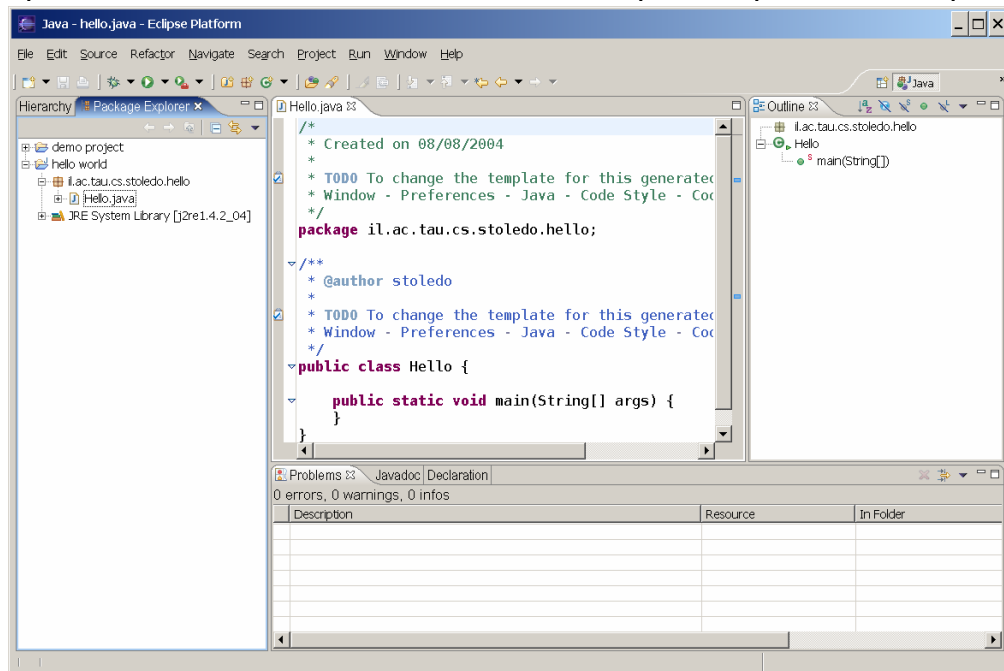
פרויקט ג'אווה יכול להכיל מספר חבילות, אבל כעת לא נשתמש ביכולת הזו.



כעת נוסף מחלקה בתוך החבילה הזו. מתפריט ההקשר של החבילה נבחר New Class ואז האשף שיפתח מציע אפשרויות רבות. אנו רק נקליד את שם המחלקה, Hello, ונבקש ליצור אוטומטית שגרת main (שדה בחירה שלישי מלמטה). בנוסף לשם המחלקה, האשף מאפשר לבחור את החבילה שאליה תשוך המחלקה, את המחלקה שהיא מרחיבה (superclass), ממשקים שהיא מממשת

ותכונות נוספות של המחלקה, וכן אפשר לבקש ליצור אוטומטית שגרות ריקות מסוימות, בתחתית האשף. כאמור, נבחר שם ונבקש ליצור main, ונצא.

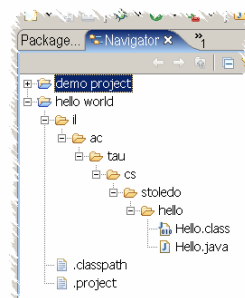
כאשר חוזרים מהאשף, אקליפס יצרה קובץ על פי תבנית מוגדרת, הקובץ החדש פתוח בעורך, והמבטים משקפים את קיומו ומצבו. מבט ה-Outline מציג את מבנה המחלקה



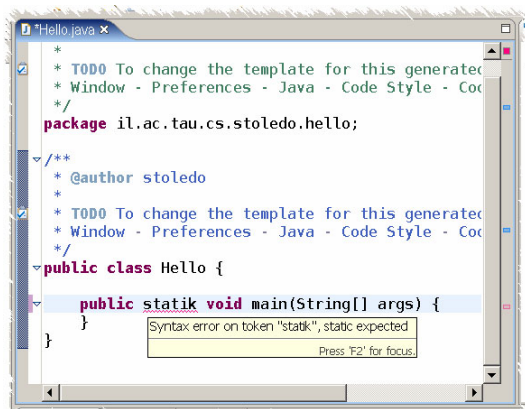
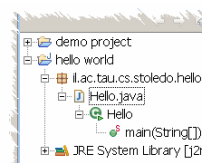
שמוצגת בעורך. המבט הזה מסונכרן עם העורך. אם נשנה את הקובץ בעורך, המבט ישתנה אוטומטית. נסו, למשל, לשנות את שם השגרה מ-main למשהו אחר; שם השגרה במבט משתנה כמעט מייד. נסו לחוקק את התג static מהגדרת השגרה; האות s האדומה נעלמת מצלמית השגרה במבט. ניתן, כמובן, להקליק על סימן המינוס במבט ולסגור את המחלקה הזו. גם המבט Package Explorer מבין את מבנה המחלקה. אם נקליק על סימן ה-+ שליד שם הקובץ במבט הזה, יפתח גם כאן מבנה הקוד, ושינויים של הקוד בעורך ישתקפו גם הם כמעט מיידית במבט. ניתן להשתמש במבטים הללו, בין היתר, כדי לנווט בתוך הקובץ. בקובץ גדול אין צורך לדפדף או לבצע

חיפוש טקסט על מנת להגיע לשגרה מסוימת או להגדרת שדה. מספיק למצוא את השגרה או השדה ב-Outline ולהקליק עליהם. נסו זאת.

כדי להעריך את הנוחות שבעבודה עם ה-Package Explorer לעומת ה-Navigator, כדי לפתוח את ה-Navigator (מהתפריט הראשי Window->Show View). המבט יציג את אותם שני פרויקטים, אבל כדי להגיע לקובץ הג'אווה שלנו, נצטרך לפתוח 6 רמות של מדריכים. כמו כן, במבט הזה יוצג הקובץ project. שאין לנו שימוש ישיר בו, ועוד קובץ דומה, classpath. המבט גם מציג את הקובץ הבינרי Hello.class, שגם בו אין לנו שימוש. בעבודה על קוד Java, ה-Package Explorer נוח הרבה יותר. נסגור את המבט.



הבה נתבונן לרגע בצלמיות שמופיעות במבט Package Explorer, שחלקן מופיע גם ב-Outline. אנו רואים כאן שש צלמיות שונות: מדריך/פרוייקט פתוחים (צלמית בצורת תיקיה פתוחה), חבילת מחלקות ג'אווה (מעין חבילה קשורה), קובץ קוד מקור ג'אווה (נייר ועליו האות J), מחלקה (עיגול ירוק גדול ובתוכו האות C), שגרה (עיגול ירוק קטן), וספרית מחלקות (ערימת ספרים). לשלוש צלמיות יש דקורציות, לציון תכונות נוספות. לפרוייקט הג'אווה יש דקורציה של האות J, לציון העובדה שזהו פרויקט ג'אווה, למחלקה Hello יש דקורציה בצורת משולש קטן, לציון שזו מחלקה שמכילה שגרת main, ולשגרה main יש דקורציה, אות S אדומה, שמציינת שזו שגרה סטטית. הצלמיות והדקורציות שלהן מאפשרות לקבל תמונה כללית על כמות קוד גדולה בלי לקרוא את קוד המקור עצמו.



השלב הבא בשיעור הוא לבחון כיצד אקליפס מגיבה לשגיאה בתוכנית ג'אווה. בעורך, שנו את התג static לתג statik. לאחר פרק זמן קצר, שנייה לערך, יופיע קווקו אדום מתחת לתג השגוי. אם נשאיר את הסמן מעל למילה הזו, לאחר פרק זמן יופיע הסבר בחלון צהוב קטן שנקרא חלון עזרה בריחוף (hover help), חלון שנפתח כאשר הסמן מרחף מעל המילה). החלון הזה יציג לנו את מהות השגיאה, כאן

שגיאת תחביר. בעורך קרו עוד שלושה שינויים כתוצאה משגיאת התחביר. בפניה הימנית העליונה של העורך, הופיע ריבוע אדום, שמסמן שהקובץ אינו תקין ויש בו שגיאות. בשול הימני של העורך, הופיע מלבן ורוד, שמסמן את מיקום השורה שמכילה את השגיאה. הקלקה על המלבן הורוד מביאה את השורה השגויה לחלק של הקובץ שרואים על המסך, ואת הסמן לשגיאה. הסימון הזה מועיל בעיקר כאשר בקובץ גדול יש מעט שגיאות, ורוצים להגיע אליהן במהירות. ריחוף עם הסמן מעל המלבן הוורוד או מעל הקווקו האדום בתוכנית מציג את מהות השגיאה עם הצעה לתיקון. במקרה הזה, התיקון המוצע נכון. המלבן הוורוד בשול הימני מסמן שהשורה שונתה ומאפשר לחזור לטקסט הקודם על ידי הקלקה ימנית.

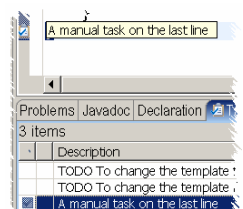
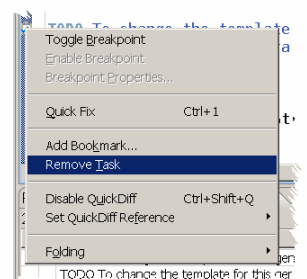
לפני שנתקן את השורה, נשמור את הקובץ עם השגיאה (Save מתפריט הקבצים או Control-S). שמירת הקובץ גורמת למספר שינויים נוספים בממשק. ראשית, ב-Package Explorer וב-Outline כל ההיררכיה שמעל הקובץ השגוי (כולל אותו) קיבלה דקורציה של X לבן על רקע אדום. זה מאפשר לזהות בעיות גם בחלקים של

הפרויקט שהם כעת סגורים. הדקורציה הזו עשויה להופיע לא רק על הקובץ שבו נוצרה השגיאה, אלא גם בקבצים אחרים שאולי כלל אינם פתוחים שנהפכים ללא תקינים בגלל השינוי. אקליפס מזהה את כל הקבצים שהפכו ללא תקינים לאחר שמירת קובץ ומסמנת אותם בהתאם. הבעיה מופיעה כעת גם במבט Problems, עם תיאורה. הקלקה על שורת הבעיה מביאה אותנו אליה, גם אם הקובץ סגור או פתוח אבל חלק אחר שלו מוצג. סגרו את העורך והקליקו על שורת הבעיה: הקובץ נפתח מחדש, והשורה הבעייתית מוצגת. סימון הבעיה, x לבן במשושה אדום, מופיע גם בשול של העורך. כעת תקנו את השגיאה ושימרו את הקובץ. כל סימוני הבעיה ייעלמו.

הטיפול של אקליפס בשגיאות תחביר מדגים חלק חשוב מהגישה של אקליפס לפיתוח בשפת ג'אווה. ראשית, העורך אינו עורך טקסט שמתעלם מהתוכן של הקובץ. העורך מבין את השפה ולכן הוא יכול לסמן שגיאות, ולכן המבטים יכולים להציג את המבנה של הקובץ, וכדומה. אולם בזמן העריכה, אקליפס מזהה רק שגיאות בקובץ הפתוח. אם שינוי בקובץ גורם לבעיה בקובץ אחר בפרויקט, אקליפס לא תזהה את הבעיה מייד. שנית, בזמן שמירת הקובץ אקליפס מקמפלת מחדש את כל הקבצים בפרויקט שהושפעו מהשינויים בקובץ. זה כולל, כמובן, את הקובץ שאנו שומרים, אבל לפעמים גם קבצים אחרים. זה מאפשר לאקליפס לזהות, בזמן שמירת קובץ, את כל הבעיות שהקובץ המעודכן גרם להן, ולסמן אותן במבטים השונים. כלומר, אקליפס עוזרת לנו לשמור על התקינות של קובץ בודד באופן רציף בזמן העריכה, ועל התקינות של הפרויקט כולו כל אימת ששומרים קובץ. שלישית, אקליפס אינה כופה תקינות על קבצי קוד המקור. כאשר מבקשים לשמור קובץ לא תקין, אקליפס שומרת אותו ומסמנת את השגיאות. כמו הרבה אספקטים אחרים של אקליפס, ניתן לשנות את ההתנהגויות הללו (למשל, לבקש מאקליפס שלא לקמפל את הפרויקט בזמן שמירה), אבל לרוב הן נוחות ומועילות ואין סיבה לשנות אותן.

לפני שנמשיך, נסביר את מהות הסימונים בצבע תכלת בשוליים הימניים והשמאליים של העורך (מלבני תכלת מימין וצלמית עם סימן v תכול משמאל). הסימונים הללו דומים במהותם לסימוני השגיאה האדומים, אך הם מציינים מטלות שלדעת אקליפס או לדעת המפתח צריך עדיין לבצע. ריחוף מעליהם יציג את המטלה (השלמת התיעוד). המבט Tasks, שאינו מוצג בדרך כלל בפרספקטיבת Java, מפרט את המטלות הללו. פתחו את המבט. הקלקה על שורה במבט תביא אתכם לשורה המתאימה בקובץ, כמו הקלקה על שגיאה במבט Problems. ניתן להסיר מטלות על ידי הפעלת תפריט ההקשר על הצלמית של המטלה בשול השמאלי של העורך, ובחירת Remove Task. אבל שמירה של הקובץ תחזיר את המטלה, בגלל הטקסט TODO שמופיע בתוך ההערה. תפריט ההקשר שמופעל כשמקליקים על השול השמאלי מול שורה מסוימת מאפשר גם להוסיף מטלה ידנית; נסו זאת. אחר כך סמנו את המטלה כמבוצעת על ידי הקלקה על העמודה הראשונה בשורה המתאימה במבט Tasks. לבסוף, בתפריט ההקשר של המבט בחרו Delete Completed Tasks, והיא תיעלם.

בדומה לשגיאות ולמטלות, אקליפס תומך גם בסימוניות (bookmarks) שניתן להוסיף לשורות בקובץ. הן מופיעות כצלמית של סימניה בשול השמאלי של העורך, כמלבן ירוק בשול הימני, וכרשימה במבט Bookmarks.



```

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}

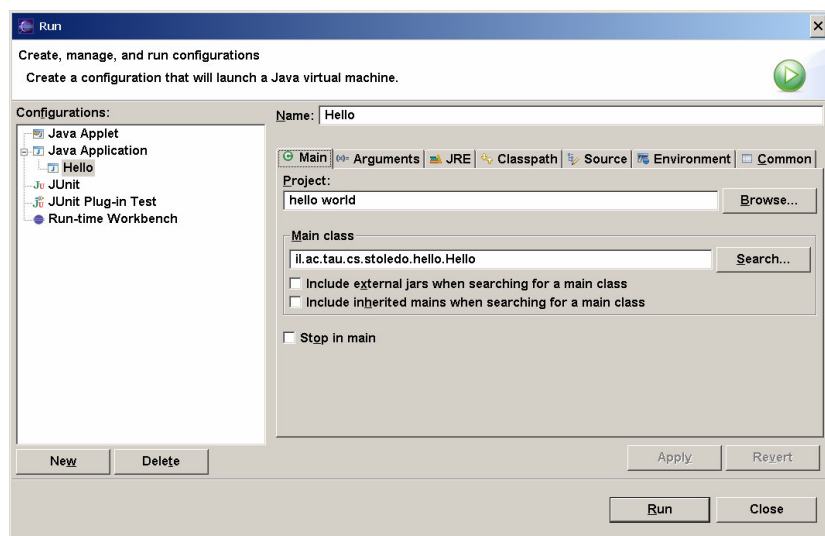
```

Problems | Javadoc | Declaration | Console x

<terminated> Hello [Java Application] C:\Program Files\Java\j2re1\bin\java.exe
Hello, World

הגיע הזמן להריץ את התוכנית. כדי שנוכל לראות פלט כלשהו, נוסיף ל- main שורה שתדפיס "Hello, World" לפלט הסטנדרטי. באחד המבטים (Package Explore או Outline) נפעיל את תפריט ההקשר של השגרה, של המחלקה, או אפילו של הקובץ או החבילה, ונבחר Run, ואז Java Application. ניתן גם רק לבחור את אחד הפריטים הללו במבט, או את Run->Run As->Java Application הראשי בתפריט. הפלט שלה יצטבר במבט בשם Console שיפתח אוטומטית. במבט הזה ניתן גם להקליד קלט, אם התוכנית מצפה לקלט.

לעיתים קרובות צריך לקבוע פרמטרים שונים של ריצת התוכנית, ולכן אי אפשר פשוט לבקש מאקליפס להריץ אותה. את הפרמטרים הללו מזינים בדיאלוג על ידי בחירת Run ושוב בתפריט הראשי, או Run ושוב בתפריט הקשר. הדיאלוג הזה מורכב ממספר פנלים שניתן להרים לחזית בעזרת תוויות. התווית הראשונה, Main, מאפשרת בעיקר לבחור פרויקט ושם מחלקה שרוצים להריץ את



שגרת ה-main שלה. התווית השנייה, Arguments, מאפשרת להעביר ארגומנטים לתוכנית. שאר התוויות מאפשרות לשלוט על אספקטים רבים של ריצת התוכנית, אספקטים שלא נדון בהם עכשיו. הקלידו מילה אחת כארגומנט לתוכנית בתווית Arguments, ולחצו על Run. התוצאה זהה לריצה הקודמת, משום שהתוכנית לא משתמשת כלל בארגומנטים שמועברים לה.

```

public class Hello {
    public static void main(String[] args) {
        for (int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}

```

Problems | Javadoc | Declaration | Console x

<terminated> Hello [Java Application] C:\Program Files\Java\j2re1\bin\java.exe
Sivan

כדי לראות שהתוכנית אכן רצה בתצורה שקבענו בדיאלוג, נשנה אותה כך שתדפיס את הארגומנטים שמועברים לה, כפי שרואים משמאל. כאשר נריץ את התוכנית כעת מתוך דיאלוג ה-Run, היא תדפיס את המילה שקבעתם כארגומנט, Sivan במקרה שבדוגמה. התצורה שקבענו נשמרת, וניתן להשתמש בה שוב ושוב.

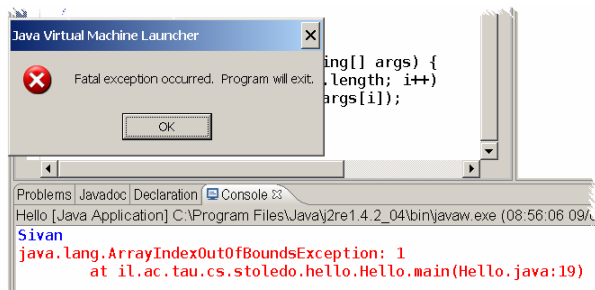
בזה מסתיים השיעור הזה; אתם יודעים כעת ליצור ולהריץ תוכניות ג'אווה באקליפס.

ניפוי תוכניות ג'אווה

אני מקווה שהצלחתם להריץ את שתי התוכניות מהשיעור הקודם ללא בעיות. אבל גם אם הצלחתם, אי אפשר לצפות שכל תוכנית שנפתח תרוץ ללא בעיות. כאשר תוכנית עפה או כאשר היא רצה אך אינה מתנהגת כפי שאנו מצפים, צריך למצוא את הפגם בתוכנית שגורם לתעופה או להתנהגות הלא תקינה. תהליך מציאת הפגמים נקרא ניפוי. בשיעור זה נראה כיצד אקליפס תומכת בניפוי.

על מנת לתרגל ניפוי, נכניס במכוון פגם בתוכנית שיגרום לה לעוף: נשנה את הלולאה שמדפיסה את הארגומנטים כך שתדפיס החל מארגומנט 0 ועד

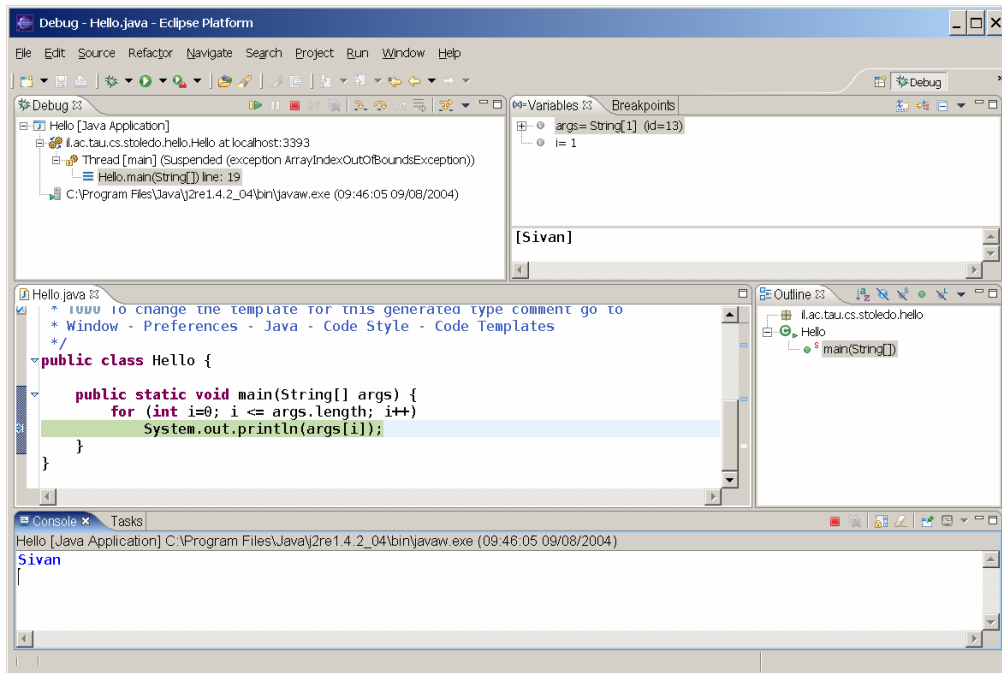
לארגומנט `args.length`. האינדקס של הארגומנט האחרון הוא `args.length-1`, כך שזה אכן פגם. כעת הריצו את התוכנית. אם שכחתם לשמור אותה (ובכך לקמפל אותה), אקליפס תציע לכם לשמור אותה. כאשר התוכנית תרוץ, היא תדפיס את הארגומנטים שהעברנו לה, וכאשר תנסה להדפיס ארגומנט מספר `args.length`, היא תעוף. חלון



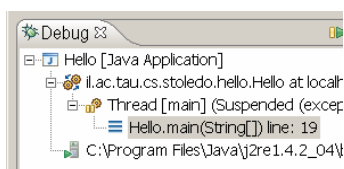
קטן יפתח ויאמר לכם שהתוכנית עפה, ובמבט ה-Console תופיע הודעת שגיאה שתתאר את הבעיה: חריגה מגבולות מערך בשורה מסוימת בתוכנית. בתוכנית כה פשוטה, קל להבין מהודעת השגיאה בלבד מה הבעיה, לחזור לתוכנית ולתקן

אותה. בפרט, הטקסט בהודעה שמתאר את השורה שבה אירעה התקלה הוא קישורית (link) לשורה בקובץ (ריחוף מעל השורה יציג קו תחתון, שמלמד שהטקסט מהווה קישורית). אם נקליק על הטקסט הזה, הקובץ יפתח בעורך כשהסמן בשורה הפגומה.

בתוכניות מורכבות יותר שיטת הניפוי הפשוטה הזו, המבוססת על התבוננות בפלט של התוכנית ובהודעת השגיאה (אם הוצגה הודעה), אינה מספיקה. פגמים בתוכניות מורכבות קל יותר לגלות באמצעות **מנפה** (debugger), כלי שמאפשר לעצור את ריצת התוכנית, לבחון ולשנות ערכים של משתנים, ובאופן כללי, להתערב באופן ידני בריצה. את המנפה מפעילים באקליפס בעזרת Run ואז Debug מהתפריט הראשי או מתפריטי הקשר. הפעולה הזו פותחת דיאלוג זהה כמעט לדיאלוג ה-Run, שמאפשר לשלוט על תצורת הריצה. אין לנו צורך לשנות את התצורה כעת, אז בדיאלוג הזה הקליקו על Debug. אקליפס תציע לעבור לפרסקטיב Debug. הסכימו. כדאי להורות לאקליפס, בדיאלוג שבו יוצע לעבור לפרסקטיב Debug, לעבור באופן לפרסקטיבה הזו בזמן ניפוי.

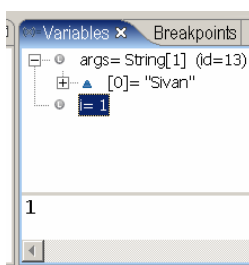


כעת אקליפס עוברת לפרספקטיבת Debug ומתחילה להריץ את התוכנית, שעפה לאחר זמן קצר. המסך הבא מתאר את מצב סביבת העבודה לאחר שהתוכנית עפה. בצד התחתון של המסך, אנו רואים את מבט ה-Console שמציג את הפלט (שהתקבל עד רגע התעופה. מעליו, את העורך, שמציג את הקובץ והשורה שגרמו לתעופה, ולצידו את מבט ה-Outline. בשורת המבטים העליונה מוצגים שלושה מבטים חדשים שלא ראינו עד כה. משמאל, מבט Debug, ומימינו שני מבטים נוספים זה על גבי זה, Variables ו-Breakpoints.



מבט ה-Debug מתאר היכן התוכנית עצרה. המבט בנוי להציג את המצב של תוכניות מרובות חוטים ותהליכים, ולכן הוא מורכב, אבל החלק החשוב עבורנו הוא השורות שמשמאלן צלמית של שלושה פסים כחולים. שורות כאלו מתארות הפעלה של שגרה. כאן הופעלה רק שגרה

אחת, main. במקרים יותר מורכבים, המבט הזה מאפשר לברר איזו שגרה קראה לזו שפועלת כרגע (או לזו שעפה), מי קראה לה, וכדומה, וכן לבחון את המשתנים של כל אחת מהשגרות הללו. זה מאפשר למפתח/ת לטפס במחסנית הקריאות על מנת לנסות למצוא את המקור לשיגאה.

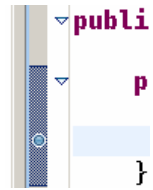


מבט ה-Variables מאפשר לבחון את ערכי המשתנים של שגרה. כרגע הוא מראה שהערך של args הוא מערך בגודל 1 של מחרוזות, ושהערך של i הוא 1. אם נקליק על סימן ה-+- משמאל ל-args, נראה את הערכים שבמערך, במקרה זה שהערך של args[0] הוא "Sivan". בחלק התחתון של המבט רואים ערך אחד, את ערך המשתנה שנבחר בחלק העליון. המבט הזה מאפשר לא רק לבחון את הערכים של המשתנים, אלא גם

לשנות אותם. שינוי ערכי המשתנים לא יסייע לנו לנפות את התוכנית הזו, מכיוון שהיא כבר עפה. אבל בתוכנית שעצרנו לפני שעפה, שינוי ערך משתנה יכול לסייע בניפוי. מייד נראה כיצד עושים זאת, אבל בינתיים, הפסיקו את פעולת התוכנית (היא כבר עפה, אבל

לא טיפלה עדיין בתעופה). לחצו על צלמית הריבוע האדום בסרגל הכלים של המבט Debug או Console, או בחרו Terminate מתפריט Run.

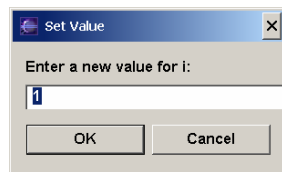
אבל כיצד עוצרים את התוכנית לפני שהיא עפה? הדרך הפשוטה ביותר היא על ידי נקודת עצירה (breakpoint), סימון בתוכנית שצריך לעצור בשורה מסוימת. יוצרים נקודת עצירה על ידי הקלקה ימנית בשול השמאלי של העורך, ובחירת Toggle Breakpoint מתפריט ההקשר שיפתח. אותה פעולה בדיוק גם מסירה נקודת עצירה. נקודת העצירה מסומנת בצלמית של עיגול כחול. סמנו נקודת עצירה בשורה שמדפיסה את הארגומנטים, והריצו שוב את התוכנית על ידי Run ואז Debug. הריצה תעצור בפעם הראשונה שהתוכנית תגיע לשורה הזו.



כאשר התוכנית תעצור בנקודת העצירה, במבט ה-Debug יהיה כתוב שהתוכנית עצרה בנקודת עצירה (ולא בחריג כמו בריצה הקודמת), ובשול השמאלי של העורך השורה תסומן בחץ קטן. המשיכו את ריצת התוכנית על ידי לחיצה על הצלמית עם המשולש הירוק, או על ידי בחירת Resume מתפריט Run, או על ידי F8. התוכנית תעצור שוב באותה שורה, כאשר הלולאה תגיע אליה שוב. התוכנית עוצרת בנקודת העצירה לפני שהשורה מופעלת, כך שהתוכנית עדיין לא ניסתה לגשת ל-args[1] ולכן עדיין לא עפה.



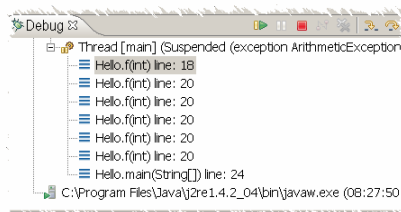
כעת הערך של i כבר 1, ולכן אם נמשיך את הריצה התוכנית תעוף מייד. במבט ה-Variables נפעיל את תפריט ההקשר של המשתנה i, ומהתפריט נבחר Change Value. יפתח דיאלוג שבו ניתן לשנות את ערך המשתנה. נשנה את הערך ל-0. גם הקלקה כפולה על המשתנה במבט פותח את הדיאלוג. כעת נמשיך את ריצת התוכנית בעזרת F8. התוכנית תדפיס שוב את args[0], הארגומנט הראשון שלה. חזרו על כך פעמיים שלוש ועצרו את התוכנית.



```
public class Hello {  
    static int f(int n) {  
        int r = 1/n;  
        if (n==0) return r;  
        return f(n-1) + r;  
    }  
    public static void main(String[] args) {  
        System.out.println( f(5) );  
    }  
}
```

כדי להמשיך ללמוד להשתמש במנפה, כדאי ליצור פגם מעניין יותר בתוכנית. עברו חזרה לפרספקטיבת הג'אווה (על ידי בחירה מהתפריט הקטן שנפתח כאשר מקליקים על הסימן « מימין לשם הפרספקטיבה הנוכחית, או מ-Window בתפריט הראשי. נשנה את התוכנית לתוכנית שמשמאל. התוכנית הזו מנסה

להדפיס את f(5), כאשר f היא פונקציה רקורסיבית. בתוכנית יש פגמים. פגם אחד הוא שכאשר הארגומנט של f הוא 0, היא מנסה לחשב את השלם ההופכי שלו, חישוב שיגרום לניסיון חלוקה ב-0. בג'אווה, חלוקת שלם בשלם 0 היא שגיאה שגורמת לחריג (exception). הריצו את התוכנית ב-Debug.



התוכנית תעוף, כמובן. מבט ה-Debug שיפתח יהיה מעניין יותר מאשר בתוכנית הקודמת, מכיוון שהתעופה קרתה בשגרה רקורסיבית, עמוק במחסנית הקריאות. כעת ניתן לראות, שמסגרות ההפעלה של השגרות מסודרות במבט מלמטה למעלה, כמו במחסנית: השגרה הראשונה שהופעלה (main) נמצאת בתחתית המחסנית, והשגרה שהופעלה אחרונה ועפה בראש

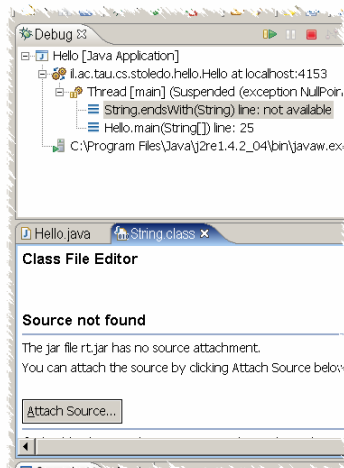
המחסנית. במבט ה-Variables ניתן לראות את ערכו של הארגומנט n, 0. בשל כך עפה השגרה, תוך כדי ניסיון להגדיר את r להיות n/1. אם בוחרים מסגרת הפעלה אחרת במבט ה-Debug, רואים ב-Variables את המשתנים של אותה מסגרת, כלומר של אותה הפעלה של שגרה. למשל, במסגרת שמתחת לזו שעפה הערך של n הוא 1, לפני כן 2, וכן הלאה.

כעת נלמד להריץ את התוכנית צעד אחר צעד, על מנת לגלות את מקור הבעיה. הפסיקו את ריצת התוכנית, וקבעו נקודת עצירה ב-main בשורה שקוראת ל-f ומנסה להדפיס את התוצאה. הריצו את התוכנית מחדש. התוכנית תעצור בנקודת העצירה. כעת יש לנו שלוש אפשרויות: להמשיך את הריצה (F8 או Resume מתפריט Run), להריץ שורה אחת בשגרה הנוכחית, בלי להיכנס לשורות שקוראים להן (F6 או Step Over), או להריץ פקודה אחת, ולהיכנס לשורות שקוראים להן אם הפקודה קוראת לשגרה (F5 או Step Into). בשורות שאין בהן קריאה לשגרה, אין הבדל בין שתי האפשרויות האחרונות. אבל במקרה שלנו, השורה קוראת ל-f ואחר כך ל-System.out.println, ולכן במקרה הזה יש הבדל. נכנס לקריאה לשגרה f בעזרת F5. המשיכו לעבור על הקוד שורה אחרי שורה בעזרת F5. התבוננו כיצד נוספות מסגרות הפעלה למחסנית הקריאות, ועקבו אחרי ערכי המשתנים במבט ה-Variables. בסופו של דבר התוכנית תעוף שוב, אבל עכשיו קל יותר להבין למה היא עפה, ואיך היא הגיעה לנקודה שבה היא עפה.

```
public static void main(String[] args) {
    String s = "bye";
    s.endsWith(null);
    System.out.println( f(5) );
}
```

השלב הבא יהיה לברר אין נראות שגיאות במנפה, כאשר הן מתרחשות בתוך שגרת ספרייה. שנו את השגרה main כך שתיראה כמו השגרה משמאל. כעת יש פגם נוסף

בתוכנית, ניסיון לבדוק האם מחרוזת מסתיימת באחרת, אבל האחרת היא null, כלומר התייחסות לעצם שלא קיים. הריצו את התוכנית. התוכנית תעוף בתוך השגרה endsWith, בגלל ההתייחסות ל-null. כאשר היא תעוף, במחסנית הקריאות יהיה שתי מסגרות הפעלה, של main ושל endsWith, שעפה.



אקליפס פותחת את הקובץ שמכיל את השגרה שעפה בעורך. אבל במקרה זה, השגרה שעפה היא שגרת ספרייה של ג'אווה, ולא חלק מקוד המקור שלנו. לאקליפס אין גישה לקוד המקור, ולכן היא פותחת את הקובץ הבינרי שמכיל את השגרה בעורך, עם ההודעה Source not found. אקליפס מציע לקשר את הקובץ לקוד מקור (על ידי לחיצה על Attach Source), שתפתח דיאלוג שיבקש מכם להצביע על קוד המקור). אבל עדיף שלא לקשר את הקובץ לקוד מקור: בדרך כלל במקרים כאלה, הפגם בתוכנית אינו ב-endsWith, אלא בתוכנית שלכם. העובדה שאין קוד מקור מלמדת בדרך כלל שהשגיאה בתוכנית

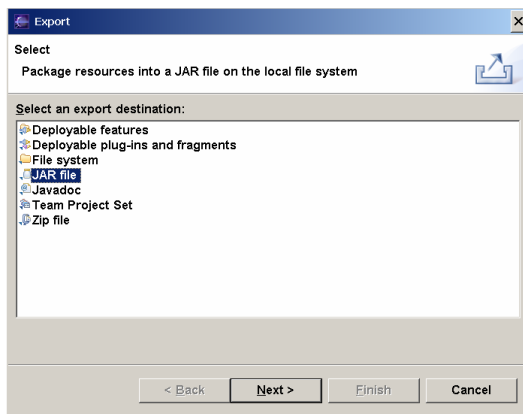
התרחשה עמוק יותר במחסנית הקריאות מהמקום שבו מצוי הפגם בתוכנית. במקרים כאלה, היעדר קוד המקור רומז שצריך לחפש נמוך יותר במחסנית הקריאות את הפגם.

הערה אחרונה לגבי ניפוי באקליפס: אקליפס שומרת את המצב של תוכניות שעפו או שהפסקנו לנפות, וההרצות הישנות הללו ממלאות את מבט ה-Debug. כדאי לנקות אותן מדי פעם בעזרת Terminate and Remove או Remove All Terminated מתפריט ההקשר של המבט.

אריזת תוכנית ג'אווה לקובץ jar

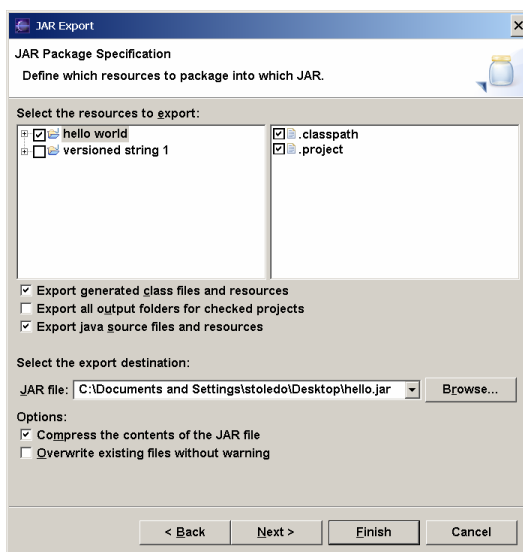
הפרויקטים שלכם מורכבים מאוסף של קבצים ומדריכים. פרויקט ג'אווה כזה אפשר להריץ בתוך אקליפס, אבל לא נוח להפיץ אותו בצורה זו למשתמשים אחרים (ולא נוח להגיש אותו לבדיקה, אם הפרויקט נכתב במסגרת לימודית). תוכניות ג'אווה מפיצים בדרך כלל בקובץ עם סיומת jar. תוכנית ג'אווה פשוטה יכולה להיות מופצת בקובץ אחד, ותוכניות מורכבות יותר מופצות לפעמים במספר קבצים כאלה.

קובץ jar הוא למעשה ארכיון של קבצים ומדריכים. למעשה, זהו קובץ בפורמט zip, וניתן לטפל בו בכלים שמיועדים לקובצי zip. קובץ jar שונה מקובץ zip בשתי דרכים: ראשית, יש לו סיומת שונה, ושנית, הוא כולל קובץ בשם MANIFEST.MF במדריך META-INF, שמתאר את מבנה התוכנית הג'אווה (או ספריית המחלקות). בפרט, הקובץ הזה מציין את שירות ה-main של איזו מחלקה להריץ כאשר המשתמש מבקש ממערכת ההפעלה להריץ את קובץ ה-jar. קובץ jar יכול להכיל לא רק קבצי class, אלא גם קבצי קוד מקור וקבצים נוספים שהתוכנית זקוקה להם, כמו צלמיות, קבצי קול, וכדומה.



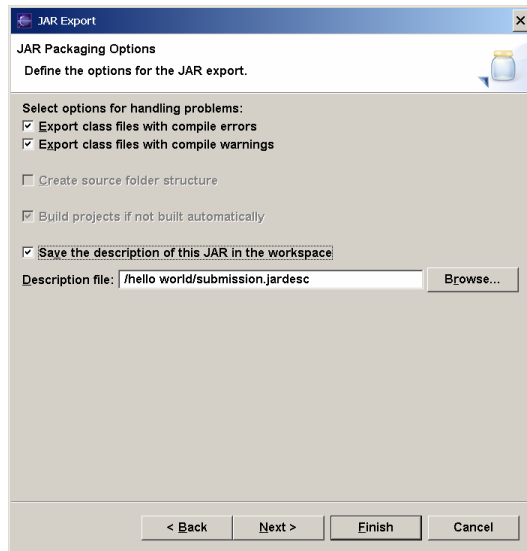
יצירה של ארכיון jar באקליפס היא פשוטה, ואקליפס גם מאפשרת לעדכן בקלות קבצים כאלו לאחר שינויים בפרויקט. כדי ליצור ארכיון jar, יש לסמן פרויקט במבט החבילות, ואז לבחור בתפריט הראשי file ואחר כך export. הפקודה הזו תפתח אשף שמאפשר לייצא את הפרויקט במספר צורות. נבחר את האפשרות JAR file ונמשיך בעזרת הקלקה על next.

לפני שנמשיך לתאר את תהליך היצירה של קובץ jar, נזהיר אתכם ממכשלה. באשף הזה יש מספר מסכים, אך לאחר שממלאים את הראשון, אפשר כבר להקליק על finish. על מנת לקבוע תצורה מדויקת לארכיון שלכם, צריך לעבור בכל המסכים, ולכן יש להקפיד להקליק על next ולא על finish עד שאתם מגיעים למסך האחרון, שבו כפתור next כבר לא עובד.

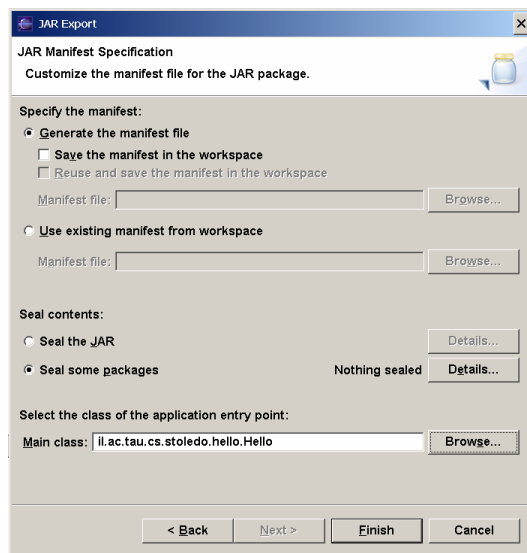


המסך הבא מאפשר לכם לקבוע אילו קבצים יכללו בארכיון, היכן לשמור אותו, והאם לדרוס ארכיון קיים באותו שם ובאותו מקום בלי לבקש קודם לכן רשות לדרוס. החלק השמאלי העליון מאפשר לכם לבחור איזה פרויקטים ואיזה חבילות ג'אווה יכללו בארכיון. החלק הימני העליון מאפשר לכם לכלול בארכיון קבצים נוספים שהם חלק מהפרויקט ושאינם קבצי ג'אווה. שני הקבצים שיופיעו שם תמיד הם .classpath ו-.project. קבצים

שמכילים את התצורה של הפרויקט באקליפס. כדאי לכלול אותם אם מתכוונים שאפשר יהיה להמשיך לפתח את הקבצים שבארכיון בנוחות, ואם מתכוונים להגיש את הפרויקט לבדיקה. אחרת אין סיבה לכלול אותם. החלק שמתחת לפנלים הלבנים מאפשר לקבוע איזה סוגי קבצים יכללו בארכיון. הקטגוריה הראשונה היא של קבצי class. וקבצים נוספים (כגון צלמיות) שכמעט תמיד צריכים אותם. הקטגוריה השנייה מאפשרת לכלול מדריכים שכוללים קבצי פלט (בדרך כלל אין צורך בהם), והשלישית קבצי קוד מקור. אם הארכיון אמור לאפשר המשך פיתוח או בדיקה, יש לכלול אותם. החלק הבא במסך מאפשר לכם לקבוע היכן ישמר הארכיון ותחת איזה שם, והחלק שמתחתיו האם הקבצים יידחסו והאם מותר לדרוס ארכיון קיים בלא אזהרה. מלאו את המסך והמשיכו.



המסך הבא שולט על שני אספקטים נוספים של יצירת הארכיון. שני השדות העליונים מאפשרים לכלול בארכיון גם קבצים עם שגיאות או אזהרות קומפילציה. שני השדות בתחתית המסך מאפשרים לשמור בקובץ את ההוראות לבניית הארכיון. כדאי מאוד לשמור קובץ כזה, מכיוון שהוא יאפשר בעתיד ליצור ארכיון מעודכן בלחיצת כפתור אם הפרויקט ישתנה. הכפתור `browse` מאפשר לכם לבחור היכן ישמר הקובץ הזה. כדאי לשמור אותו בתוך הפרויקט. לחצו `next` למסך הבא.



המסך האחרון שולט על הקובץ `MANIFEST.MF` שייכלל בארכיון. המסך מאפשר לכם ליצור מניפסט חדש באופן אוטומטי, לשמור אותו לשם עריכה ידנית, או להשתמש בקובץ קיים. מומלץ ליצור קובץ באופן אוטומטי ואין בדרך כלל צורך לשמור אותו מחוץ לארכיון. החלק האמצעי של המסך מאפשר לקבוע את כל החבילות בארכיון או חלק מהן כך שאי אפשר יהיה להוסיף מחלקות לחבילות. זו אפשרות מועילה לספריות מחלקות שמופצות באופן מסחרי, אבל בדרך כלל אין צורך לקבוע את החבילות. החלק

התחתון של המסך מאפשר לקבוע איזו שגרת `main` תרוץ כאשר המשתמש מבקש להריץ את קובץ הארכיון. אם נשאיר את השדה הזה ריק, הארכיון יהיה שימושי בתור ספריית מחלקות, אבל לא בתור תוכנית עצמאית. זהו המסך האחרון, ולאחר שממלאים אותו ולוחצים על `finish`, אקליפס יוצרת את הארכיון.

על מנת לייצר גרסה חדשה של ארכיון מתוך הוראות ייצור שמורות בקובץ `jar`, יש לבחור מתוך תפריט ההקשר שלו `create Jar`. על מנת לשנות את ההוראות בעזרת אותו אשף, יש לבחור `open JAR packager` מתפריט ההקשר.

הוראות להגשת תרגילים בבית הספר למדעי המחשב באוניברסיטת תל-אביב: יש להגיש קובץ `jar` אחד עבור כל תרגיל, אלא אם ניתנה הוראה מפורשת אחרת. ארכיון ה-`jar` צריך להכיל את כל הקבצים הרלוונטיים לתרגיל. יש להקפיד על הנקודות הבאות כאשר מכינים את הארכיון:

- יש לכלול בארכיון גם קבצי קוד מקור.
- יש לכלול את הקבצים `project` ו-`classpath`, על מנת שהבודקים יוכלו לייבא בקלות את הקובץ לאקליפס כפרויקט שלם.
- אם התרגיל מהווה תוכנית שלמה שניתן להריץ באופן עצמאי, יש לקבוע איזו שגרת `main` היא נקודת ההתחלה של התוכנית.
- רצוי לבדוק את התכולה לפני ההגשה. ניתן לבצע זאת על ידי הפקודה `jar tf archivename.jar` בתוכנית מעטפת (הפקודה `jar` היא חלק מסביבת הפיתוח הסטנדרטית של ג'אוה). הפקודה הזו מדפיסה את רשימת הקבצים הכלולים בארכיון. שתי דרכים נוספות לבדוק תכולת ארכיון `jar` הן לשנות את הסיימת שלו ל-`zip` ולבחון את תכולתו באמצעות איזשהו כלי לטיפול בארכיוני `zip`, או לפתוח פרויקט ג'אוה חדש באקליפס ולייבא לתוכו את תוכן הארכיון.

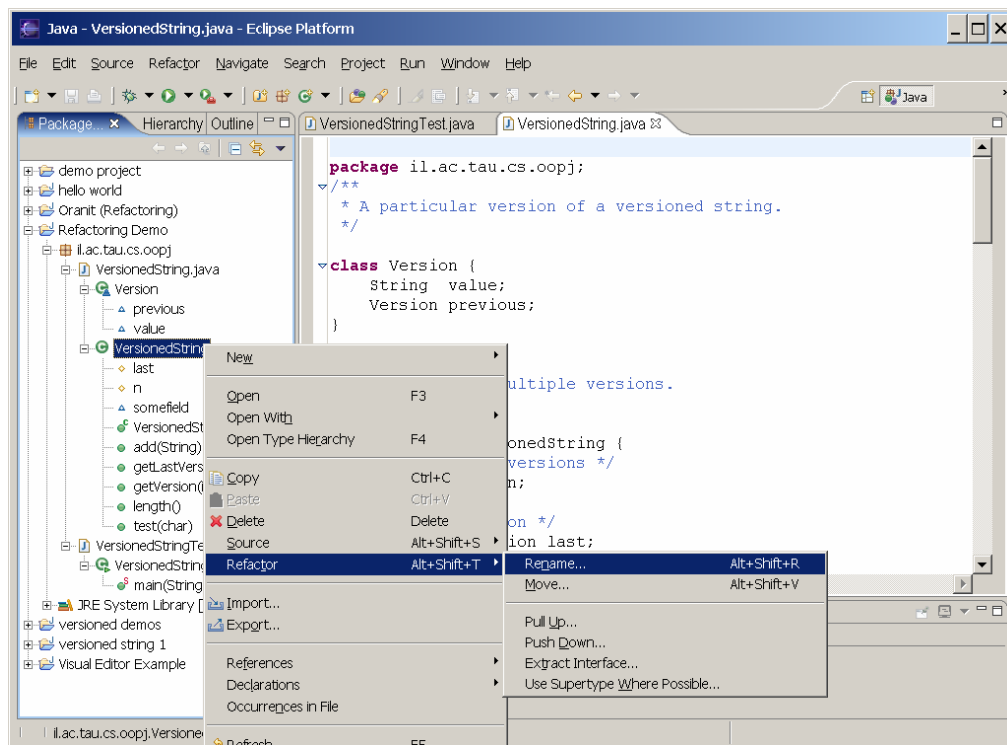
שינויים מובנים בקוד: Refactoring

גם אם מתכננים תוכנית בקפדנות לפני תחילת שלב המימוש, אי אפשר להימנע מביצוע שינויים בקוד. השינויים יכולים להיות תוצאה של תיקון באגים, של הוספת יכולות חדשות לתוכנה, או של ניסיון לשפר את מבנה הקוד. בתוכנית גדולה, גם שינוי קטן לכאורה עשוי לדרוש שינויים במקומות רבים בקוד. למשל, שינוי שם מחלקה דורש לשנות את כל ההתייחסויות אליה (באופרטור new, במחלקות שמרחיבות אותה, ובהצהרה על טיפוסים משתנים ושדות).

סביבת הפיתוח לג'אווה באקליפס כוללת מנגנון לסיוע בביצוע שינויים בקוד. המנגנון נקרא refactoring. השם נובע מהרעיון ששינוי מובנה בקוד דומה לפירוק הקוד למרכיביו והרכבתם מחדש בצורה אחרת. התמיכה של סביבת התכנות בביצוע שינויים בקוד מיועדת לעודד תוכניתנים לבצע שינויים כאלה על מנת לשפר את מבנה הקוד. ללא תמיכה כזו, תוכניתנים נמנעים לפעמים מביצוע שינויים בקוד כדי לא לגרום לפגמים חדשים בקוד. למרות התמיכה ב-refactoring כדאי לנהוג בשמרנות לגבי ביצוע שינויים, בייחוד אם איננו סבורים שהקוד הקיים גרוע או זקוק לתיקונים, אבל לפעמים באמת צריך לשנות קוד, ואז התמיכה הזו מועילה מאוד.

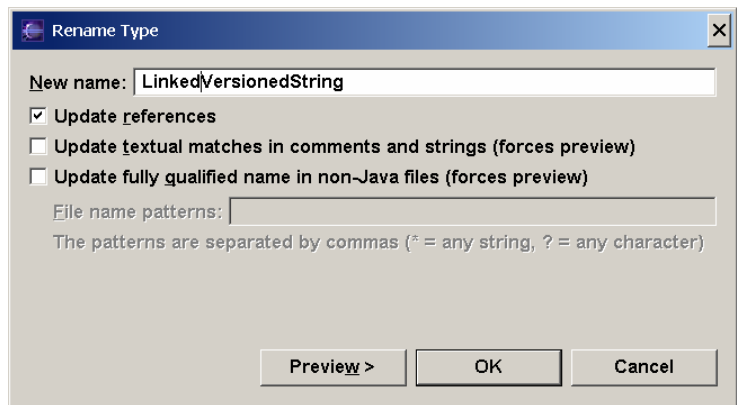
תהליך שינוי קוד בעזרת מנגנון ה-refactoring כולל ארבעה שלבים. בשלב הראשון, המשתמש בוחר את סוג השינוי המבני הרצוי. למשל, השינוי המבני "שינוי שם", בחירת האלמנט ששמו ישונה, ובחירת השם החדש. בשלב השני, אקליפס עצמה מזהה את כל השינויים בקוד שהשינוי המבני גורר. למשל, אם משנים שם מחלקה, צריך בדרך כלל לשנות את שם הקובץ שמכיל אותה, ולשנות את כל ההתייחסויות אליה. בשלב השלישי, אקליפס מציגה למשתמש את כל השינויים הללו ומאפשרת לו לבחון כל אחד מהם (הצגת לפני/אחרי של קטע הקוד הרלוונטי) ולהחליט האם הוא רוצה שאקליפס תבצע את השינוי או לא. בשלב הרביעי והאחרון אקליפס מבצעת את השינויים שהמשתמש אישר לבצע.

בשיעור זה נלמד להשתמש בחלק מיכולות ה-refactoring של אקליפס. אנו נתחיל



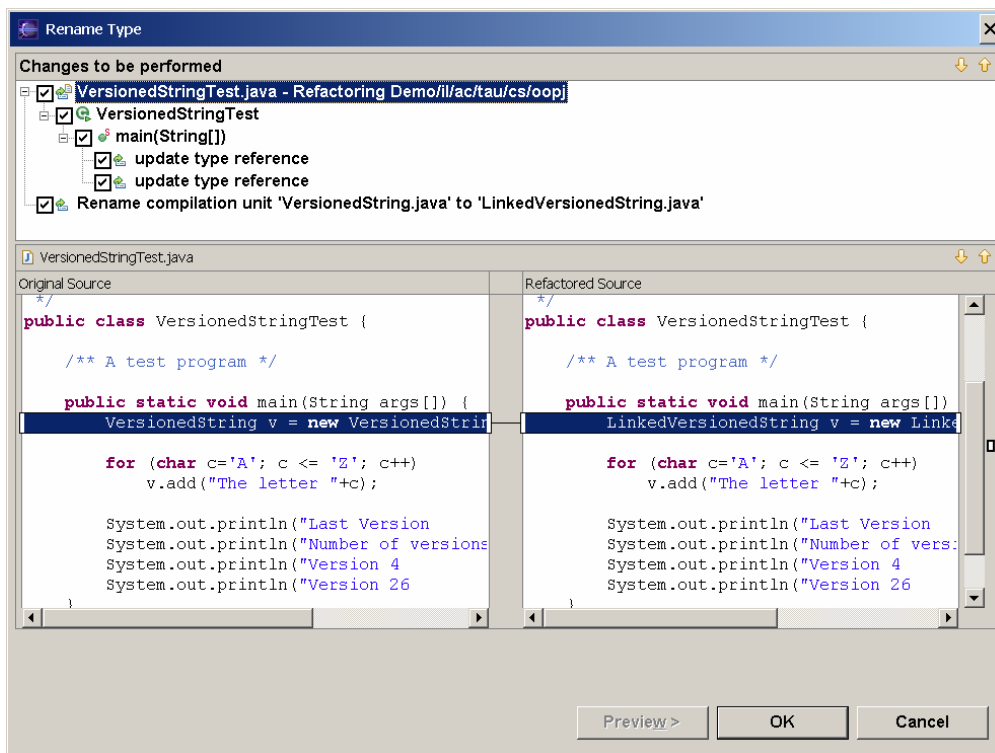
מפרייקט קטן שמכיל שתי מחלקות, `VersionedString`, שמייצגת סדרת מחרוזות, ומחלקה לבדיקת הקוד, `VersionedStringTest`. המחלקה `VersionedString` מייצגת את הסדרה בעזרת רשימה מקושרת, ומציעה ללקוחותיה את השירותים `add(String)`, `int length()`, `String getVersion(int)`, ו-`String getLastVersion()`. המטרה המיידית שלנו היא לשנות את מבנה הקוד כך שהמחלקה הזו תהפוך למחלקה שממשת ממשק (interface). אנו רוצים שהממשק יקרא `VersionedString` והמחלקה הקיימת תקרא `LinkedVersionedString`. אי לכך, השינוי הראשון שנבצע הוא לשנות את שם המחלקה.

נבחר את המחלקה `VersionedString` במבט החבילות או במבט ה-`outline`, או שנציב את הסמן בעורך על ההצהרה על המחלקה, ונפעיל את תפריט ההקשר (הקלקה ימנית). מתפריט ההקשר נבחר `Refactor` ואז `Rename`. אפשר גם לבחור את המחלקה ולבחור ישירות `Rename` מהפריט `Refactor` בתפריט הראשי. הדיאלוג שנפתח מאפשר לבחור שם חדש למחלקה ולבחור את השינויים שאקליפס תבצע בעצמה. השינוי שמוצע בבירור המחזל הוא לשנות התייחסויות לשם המחלקה בקוד, אבל לא בהערות או בקבצים שאינם תוכניות ג'אווה. נשאיר את ברירת המחזל על כנה ונקליק על `Preview`, על מנת לבחון ולאשר (או לא לאשר) את

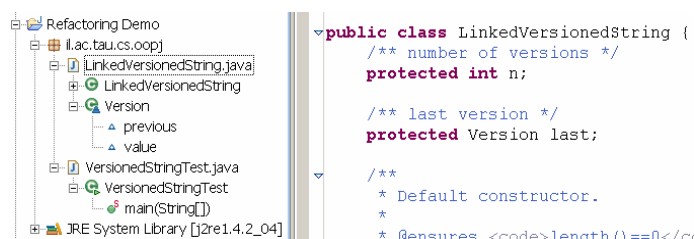


השינויים שאקליפס מבקשת לבצע.

המסך שנפתח מראה את השינויים שאקליפס מציעה לבצע עבורנו. חשוב להבין את המסך הזה על מנת שנוכל באמת לבחון את השינויים שאקליפס מציעה. החלק העליון

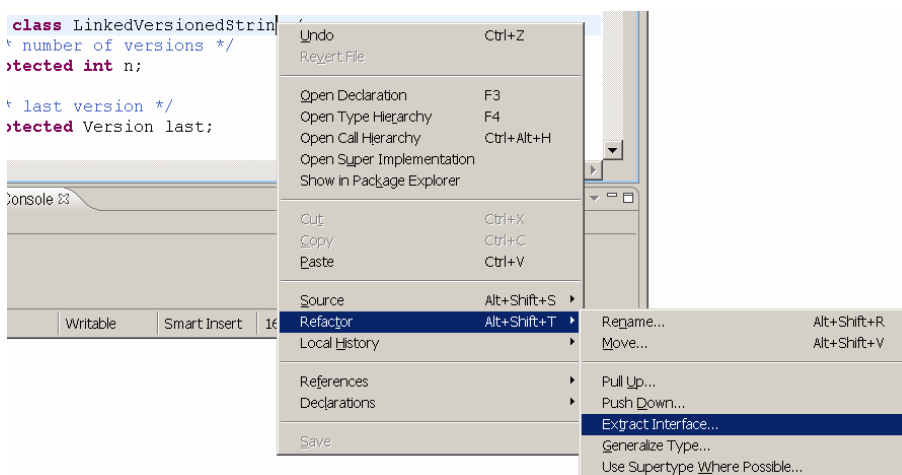


של המסך מציג רשימה של השינויים המוצעים. אפשר לפתוח את הרשימה, וכאן אכן פתחנו אותה. במחלקת הבדיקה, שהיא המחלקה שבחרנו ברשימה ואותה רואים בחלק התחתון של המסך, אקליפס מציעה לעדכן שתי התייחסויות למחלקה, הצהרת טיפוס ובניית עצם. השינוי המוצע השני הוא שינוי שם הקובץ שמכיל את המחלקה, `VersionedString`; בגלל שמחלקות ציבוריות צריכות להיות בקובץ ששמו כשמן, השינוי הזה אכן דרוש. שימו לב שהשינוי המקורי שביקשנו לבצע, שינוי שם המחלקה, לא מופיע במסך הזה. המסך הזה מציג רק שינויים במקומות אחרים שהשינוי המקורי גורר. בחלק התחתון של המסך אקליפס מציגה את מצב הקובץ שישתנה, לפני ואחרי השינוי. כאן השינוי הוא שינוי טקסט של שורה אחת. לפעמים השינוי גדול יותר, ולפעמים הוא כולל יצירת קוד חדש או מחיקת קוד קיים. מעבר בין השינויים המוצעים בחלק העליון של המסך מחליף את התצוגה בחלק התחתון. על ידי בחירה או אי-בחירה של שינויים מוצעים אפשר לראות את הקוד שיווצר בכל תצורה. לאחר שבחרנו את כל



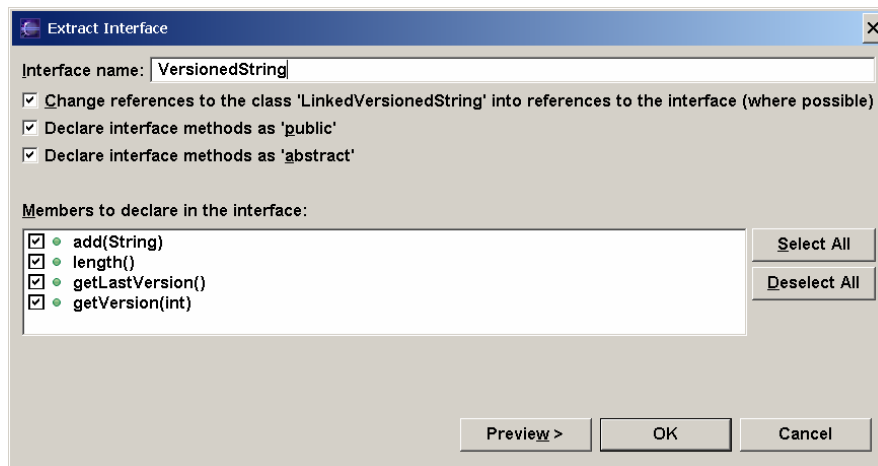
השינויים שאנו מעוניינים בהם (כאן כולם) וביטלנו את הבחירה של שינויים שאיננו מעוניינים לבצע. נקליק OK, והפרויקט ישתנה, כפי שאפשר לראות בתמונת המסך.

קעת נבצע שינוי מורכב יותר. נבחר שוב את המחלקה (שכעת קוראים לה `LinkedVersionedString` ונבחר `Extract Interface` מתפריט ה-`Refactor`. מטרטנו קעת היא לייצר אוטומטית מנשק שיגדיר את השירותים של המחלקה שלנו



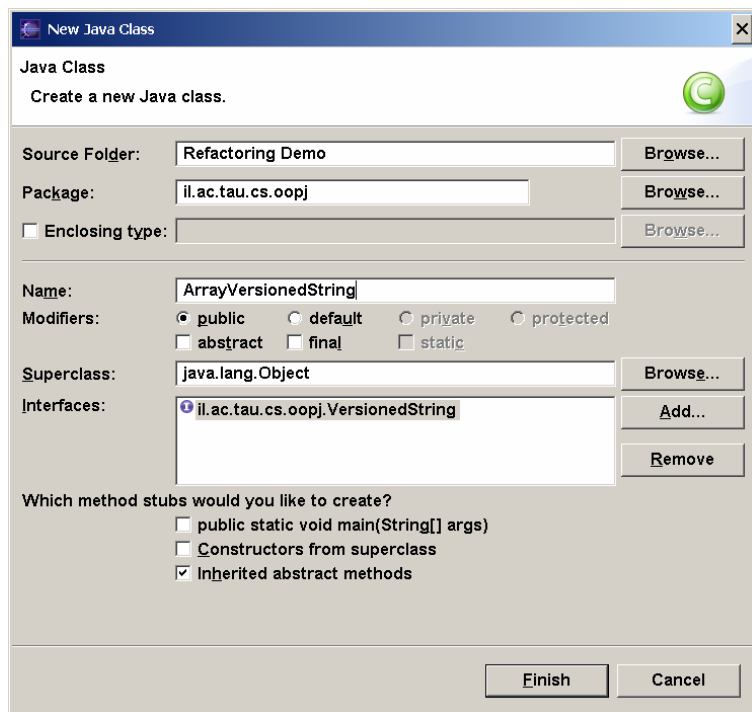
מספקת. זה יאפשר לנו להגדיר עוד מחלקות שמממשות בדיוק את אותו מנשק, כך שלקוחות יוכלו לבחור ביניהן.

המסך הבא מאפשר לבחור שם למנשק החדש שיווצר, לבחור האם לשנות התייחסויות למחלקה המקורית להתייחסויות למנשק (כדאי), כיצד להצהיר על השירותים בממשק, ואיזה שירותים של המחלקה יופיעו בממשק החדש. נקליק על `Preview`,

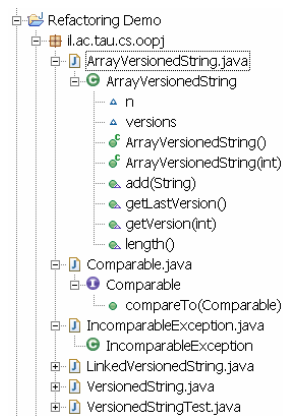


ולאחר מכן נאשר לבצע את השינויים המוצעים וליצור את המנשק. עכשיו נוסיף את הפסוק `implements VersionedString` להצהרת המחלקה `LinkedVersionedString`, ובכך סיימנו את השינוי הזה.

כעת נוסיף עוד מחלקה פשוטה שמממשת את המנשק, הפעם בעזרת מערך של



מחרוזות. ניצור את המחלקה על ידי בחירת החבילה במבט החבילות ובחירת New Class מתפריט ההקשר של החבילה או מ-File בתפריט הראשי. בדיאלוג נציין שהמחלקה החדשה צריכה לממש את המנשק `VersionedString`, ונבקש להגדיר את כל השירותים שהמנשק דורש. כמובן שהם ייווצרו ריקים, אבל זה בכל זאת מקל על הגדרת המחלקה.

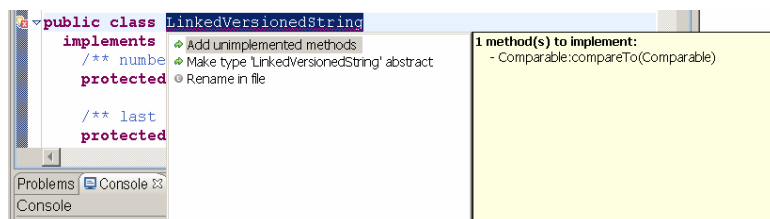


כעת נוסיף לחבילה עוד מנשק, `Comparable`, שמגדירה שירות `compareTo` שמאפשר להשוות עצמים, וחריג, `IncomparableException`. החריג יאפשר לשירות `compareTo` להודיע למי שקורא לו שהעצם שהועבר כארגומנט אינו בר השוואה לעצם `compareTo` שלו הופעל. למשל, `String` אינו בר השוואה לעצם מטיפוס `VersionedString`. בתמונת המסך משמאל אפשר לראות את המבנה החדש של החבילה, שכולל את שתי המחלקות החדשות שהוספנו (`ArrayVersionedString` והחריג) ואת המנשק החדש.

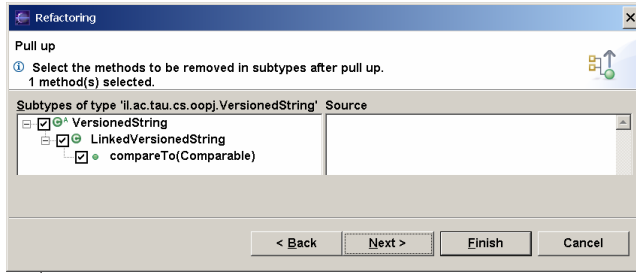
```
public class LinkedVersionedString
    implements VersionedString, Comparable {
```

לאחר שהוספנו את מנשק ההשוואה, נוסיף להצהרה של `LinkedVersionedString`

הבטחה שהמחלקה מממשת את מנשק ההשוואה. מייד מופיע סימון של בעיה בקוד, ואם משאירים את הסמן מעל אחד מסימוני הבעיה, אקליפס מסבירה שהבעיה היא שהמחלקה מבטיחה לממש את `Comparable`, אבל לא מממשת את `compareTo`.



נפעיל את תפריט ההקשר של סימון הבעיה ונבחר Quick Fix. אקליפס מציעה



```

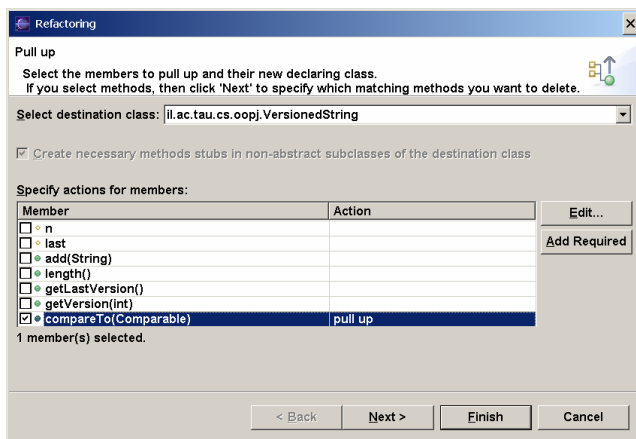
}
if (length() > ovs.length()) return 1;
if (length() < ovs.length()) return -1;
return 0;
}

```

השירות החדש, שמוצג משמאל, ממומש תוך שימוש בשירותים שהמנשק

VersionedString מגדיר, אבל ללא שימוש בשדות המופע של המחלקה. לכן, הגדרת השירות תקפה למעשה לכל מחלקה שמממשת את VersionedString. כדי להשתמש בשירות הזה בכל מחלקה כזו, צריך להעביר אותו למחלקה מופשטת שממנה כל המחלקות הללו יירשו את השירות. אבל VersionedString היא מנשק, לא מחלקה מופשטת. לכן, ראשית נהפוך את המנשק למחלקה מופשטת. מנגנון ה-refactoring לא תומך בשינוי כזה (לפחות בגרסאות 3.0 ומטה), ולכן את השינוי הזה נצטרך לבצע ידנית. נשנה את המנשק כך שיהפך למחלקה מופשטת (abstract class), נצהיר שהמחלקה הזו מממשת את Comparable, ונשנה את המחלקות LinkedVersionedString ו-ArrayVersionedString כך שירחיבו את המחלקה המופשטת, במקום שיממשו מנשק. לאחר סיום סדרת השינויים הזו תיווצר בעיה ב-ArrayVersionedString, משום שזו מחלקה לא מופשטת אבל חסר בה השירות compareTo. נעזוב בינתיים את הבעיה הזו.

העובדה שהתמיכה ב-refactoring באקליפס לא כוללת הסבה של מנשק למחלקה מופשטת מצביעה על עובדה חשובה. תהליכי refactoring הם לא קבוצה סגורה שאפשר לממש את כולה על מנת להגיע ל-"תמיכה מלאה". נכון יותר לחשוב על התהליכים הללו כקטלוג של שינויים בקוד שתוכניתנים עושים לעיתים קרובות יחסית, ושלגביהם ניתן להגדיר את כל השינויים הנגררים מהם. הקטלוג הזה הוא אינסופי, מתחיל בשינויים נפוצים מאוד, כמו שינויי שמות, וממשיך בשינויים יותר מורכבים ופחות שימושיים. מנגנון ה-refactoring תומך בשינויים הנפוצים ביותר, וכמובן רק באלה שניתן לבצע בדרך כלל בשלמות באופן אוטומטי. אבל לא כל סוג שינוי נתמך, גם אם הוא מובנה לחלוטין.



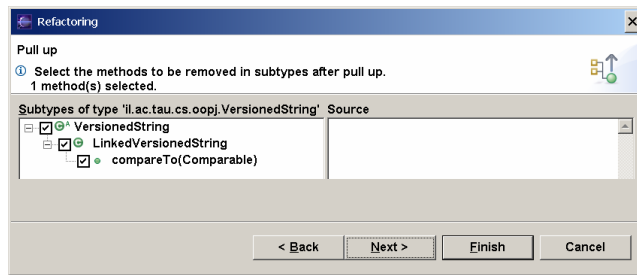
שני תיקונים אפשריים: או לממש את השירות החסר, או לסמן את המחלקה כמופשטת. נבחר את האפשרות הראשונה. אקליפס תגדיר שירות כנדרש, אבל הוא כמובן יהיה ריק. נגדיר את השירות.

השירות החדש, שמוצג משמאל, ממומש תוך שימוש בשירותים שהמנשק

נמשיך בביצוע השינויים. כעת המחלקה VersionedString (מופשטת) שהמחלקה LinkedVersionedString מרחיבה. במצב הזה, אפשר להשתמש במנגנון ה-refactoring על מנת למשוך את השירות compareTo מהמחלקה המרחיבה

למחלקת הבסיס. נבחר את השירות או את המחלקה VersionedString ומתפריט ה-refactoring נבחר Pull Up. המסך שייפתח מאפשר לבחור איזה שירותים ושדות

להעביר מהמחלקה למחלקה, ולאן להעביר אותם. נסמן רק את השירות `compareTo` ונמשיך.



כאשר מושכים שירות למעלה בגרף הירושה, שירותים שקודם היו מוגדרים במחלקות "אחיות" נהפכים לפעמים למיותרים, משום שהמחלקות שלהם יורשות כעת את השירות שנמשך

למעלה. בפרט, השירות שאותו מושכים למעלה כמעט לעולם אינו דרוש יותר במחלקה שבה היה מוגדר. במסך הבא, אקליפס מציעה למחוק הגדרות של שירותים שאולי כעת אין בהם צורך. במקרה שלנו, זה רק השירות שאותו משכנו, ובאמת אין בו יותר צורך. נמשיך.

למעשה סיימנו, אבל כאשר הכנו את דפי העבודה הללו, אקליפס דיווח על בעיה: לדעת אקליפס, השירות `length` שהשירות שמשכנו קורא לו, אינו מוגדר במחלקה שאליה משכנו את השירות `compareTo`. זה לא נכון; אקליפס טועה. זה כנראה פגם במנגנון ה-`refactoring`. האזהרה של אקליפס אינה נכונה, ולכן נורה לה לבצע את השינוי בכל זאת. גם הטענה של אקליפס הייתה נכונה, יכולנו להתעלם מהאזהרה ולבצע את השינוי. גם אם הקוד שהיה נוצר לא היה תקין, יכולנו לתקן אותו ידנית, או לבטל לחלוטין את השינוי המובנה בעזרת `undo` מהפריט `refactor` בתפריט הראשי.

בזה סיימנו למעשה את השינוי שביקשנו לבצע. גם הבעיה שהייתה במחלקה `ArrayVersionedString` נעלמה מאליה, כי השרות החסר נורש כעת מ-`VersionedString`.

לפני שאתם עוזבים את השיעור, כדאי לנסות את מנגנון ה-`undo/redo` בתפריט ה-`refactoring`. המנגנון הזה הוא כלי חזק מאוד, מכיוון שהוא מאפשר לנסות לבצע שינויים מובנים ללא חשש, גם אם הם משנים קבצים רבים. אם השינוי לא מוצלח, או שאקליפס נכשל (או שאנחנו נכשלו בבחירת השינויים שצריך ולא צריך לבצע מתוך הרשימה המוצעת), ניתן פשוט להחזיר את המצב לקדמותו. ואם התחרטנו על ההתחרטות, אפשר לבחור `redo` ולבצע את השינוי שוב.

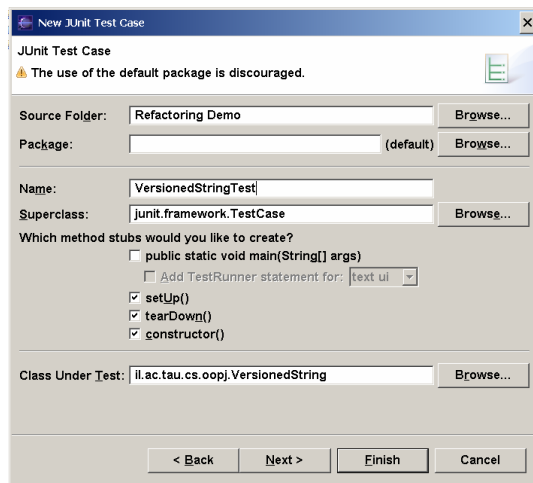
מנגנון ה-`undo/redo` הזה נפרד ממנגנון ה-`undo/redo` של עורכי הקבצים, ולכן הוא גם מופעל מתפריט אחר. המנגנון הזה מאפשר לצעוד קדימה ואחורה בסדרת שינויים מובנים שביצענו, אבל רק כל עוד לא שינינו את הקבצים הרלוונטיים ידנית בעורך. ברגע שמשנים קובץ ידנית, מנגנון ה-`redo/undo` של ה-`refactoring` לא מסוגל להחזיר אותו למצב קודם או מאוחר. אי לכך, כדאי לבצע סדרת שינויים מובנים, לוודא שהמבנה שנוצר אכן רצוי, ורק אז לשוב ולערוך את הקבצים ידנית. עריכה ידנית מוקדמת מדי מבטלת את אפשרות ה-`undo` של השינויים המבניים.

בדיקת תוכנה בעזרת JUnit

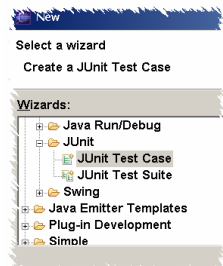
אקליפס כוללת תמיכה מובנית במנגנון בדיקות בשם JUnit. המנגנון מיועד בעיקר לבדיקות יחידה (unit tests). המנגנון מאפשר למפתחים להגדיר מערכת היררכית של בדיקות. הרמה הנמוכה ביותר היא בדרך כלל בדיקות יחידה של מחלקות בודדות; בדיקת יחידה של מחלקה מורכבת בדרך מקבוצה גדולה של בדיקות, שכל אחת מהן בודקת היבט אחר של המחלקה (מצבים מופשטים שונים, שירותים שונים, וכדומה). הרמה הבאה עשויה להיות חבילה (package), והרמה שמעליה עשויה להיות התוכנית כולה.

בשיעור הזה נלמד להשתמש ב-JUnit מתוך אקליפס; נלמד זאת עת ידי יצירת בדיקות למחלקות שיצרנו בשיעור הקודם אודות refactoring. אנו נתמקד בבדיקות קופסה שחורה, שבודקות מחלקה מול החוזה שלה (מול ההגדרה של מה כל שירות צריך לבצע). JUnit מתאים גם למימוש בדיקות כיסוי, שבודקות את הקוד של מחלקה באופן ממצה, אבל בשיעור זה לא נדגים סוג כזה של בדיקות. (ניתן לממש בדיקות כיסוי בעזרת אותם מנגנונים של JUnit שנלמד כאן.)

בדיקה של מחלקה על ידי JUnit מתבצעת בדרך כלל על ידי מחלקת בדיקה, ששמה הוא שם המחלקה הנבדקת עם סיומת Test. נתחיל את השיעור ביצירת מחלקת בדיקה ל-VersionedString הנבדקת, על מנת לוודא שיש גישה לשירותים הציבוריים ללקוחות מחוץ לחבילה. נבחר את הפרויקט שאליו אנו רוצים להוסיף את הבדיקות, נבחר New, ומתוך התפריט

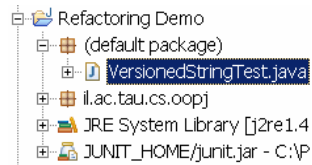


שיפתח נבחר Other. יפתח אשף לבחירת סוג המשאב שברצוננו ליצור. נבחר JUnit Test Case, אפשרות מתחת ל-JUnit שמופיע מתחת ל-Swing. נקיש Next למסך הבא. לפני שיפתח המסך הבא, יוצג דיאלוג שבו אקליפס מציע לנו להוסיף את קובץ ה-jar של JUnit לרשימת קבצי ה-jar שהפרויקט משתמש בהם. מכיוון שקוד הבדיקה שנכתוב תלוי ב-jar הזה, נבחר yes בדיאלוג. המסך הבא של האשף, שמוצג משמאל, יפתח



כעת. האשף הזה מאפשר לנו לבחור את המיקום והחבילה של מחלקת הבדיקה (בשליש העליון של המסך), את שם מחלקת הבדיקה, המחלקה שאותה היא מרחיבה, ואיזה שירותים ייווצרו אוטומטית עבורה (בשליש המרכזי), ואת שם המחלקה שאותה היא בודקת, בתחתית. נשאיר כרגע את שם החבילה ריק, ונמלא שם עבור מחלקת הבדיקה. מכיוון שאנו מבקשים לבדוק את VersionedString, נקרא למחלקת הבדיקה VersionedStringTest. כדאי כעת גם למלא את שם המחלקה הנבדקת. קל לעשות זאת על ידי שימוש בכפתור Browse, מימין. שם המחלקה שנרחיב היה מלא כאשר המסך נפתח בערך junit.framework.TestCase. נשאיר את ברירת המחדל הזו. מתחת לשם המחלקה שנרחיב האשף מציע לנו לבנות ארבעה שירותים: main, שיאפשר להריץ את הבדיקה כתוכנית עצמאית, setup ו-tearDown, שיאפשרו לבצע אתחול וסיום לפני כל בדיקה, ובנאי. נותר על main, מכיוון שאנו מתכוונים להריץ את

הבדיקה מתוך אקליפס, לא כתוכנית עצמאית. את שלושת השירותים האחרים נבקש מאקליפס ליצור. נקיש Next ונעבור למסך האחרון של האשף.



המסך הזה מאפשר ליצור בדיקות בודדות, בדיקה לכל שירות של המחלקה הנבדקת (כולל שירותים שירשה ממחלקות אחרות, במקרה הזה רק מ-`java.lang.Object`). צורת ארגון כזו, של בדיקה לכל שירות, טובה לבדיקות כיסוי, אבל לא לבדיקות קופסה שחורה. בכל זאת נבחר שירות אחד, למשל `add`, רק כדי שיווצר שירות בדיקה עם חתימה מתאימה. לחיצה על `Finish` תיצור את מחלקת הבדיקה. במבט החבילות נראה כעת בפרויקט חבילת ברירת מחדל שמכילה את מחלקת הבדיקה, וגם נראה את ה-`jar` שמכיל את `JUnit`.

```
import junit.framework.TestCase;

public class VersionedStringTest extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public VersionedStringTest(String name) {
        super(name);
    }

    public void testAdd() {
    }

}
```

מחלקת הבדיקה תיפתח בעורך, ופרט להערות היא תיראה כמו המחלקה משמאל. יש בה בנאי שמקבל מחרוזת ומעביר אותה לבנאי של המחלקה שאותה הוא מרחיב, שירותי `setUp` ו-`tearDown` שגם הם רק קוראים למחלקת הבסיס, ושירות `testAdd` שמיועד לבדוק את השירות `add` של המחלקה הנבדקת.

כעת נממש את הבדיקה. נגדיר במחלקה שדה בשם `vs` מטיפוס `VersionedString` (כדאי להוסיף פסוק `import` בתחילת הקובץ), עם הגנת `protected`. בשירות `setUp` נבנה עצם מטיפוס `LinkedVersionedString` ונשמור התייחסות אליו ב-`vs`. בשירות

```
import junit.framework.TestCase;
import il.ac.tau.cs.oopj.*;

public class VersionedStringTest extends TestCase {

    VersionedString vs;

    protected void setUp() throws Exception {
        super.setUp();
        vs = new LinkedVersionedString();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        vs = null;
    }

}
```

`tearDown` נסיר את ההתייחסות על ידי השמת `null` ל-`vs`. השירותים הללו נקראים לפני ואחרי כל בדיקה בודדת. בדיקה בודדת היא שירות ציבור ששמו מתחיל ב-`test`, כמו השירות `testAdd` שהאשף יצר עבורנו. אנו רוצים ליצור עצם לפני כל בדיקה, כדי שכל בדיקה

תתחיל ממצב ידוע, גם אם הבדיקה שלפניה חשפה פגם במחלקה הנבדקת. הסיבה שיצרנו עצם מטיפוס `LinkedVersionedString` ולא מהטיפוס `VersionedString`, שאותו אנו מבקשים בעצם לבדוק, היא שהטיפוס הנבדק הוא מחלקה מופשטת ולכן אי

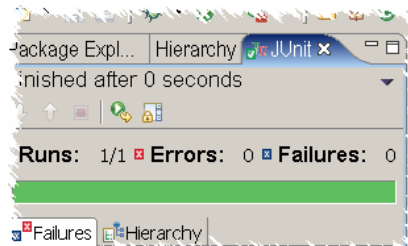
אפשר לייצר ממנו עצמים מוחשיים. בהמשך נראה כיצד להתמודד עם הבעיה הזו בצורה טובה יותר.

כעת נממש את בדיקה. הבדיקה הראשונה תוודא שהמצב של העצם מייד לאחר בנייתו תקין. נשנה את שם הבדיקה מ-testAdd ל-testEmpty, ונוסיף הצהרה לגבי תנאי

```
public void testEmpty() {  
    assertEquals(0, vs.length());  
}
```

האחר של הבנאי: שאורך סדרת הגרסאות היא 0. כעת אפשר כבר להריץ את הבדיקה. נבחר את הקובץ שמכיל את מחלקת הבדיקה במבט

החבילות, ומתפריט ההקשר שלו נבחר Run ואחר כך JUnit Test. אם מימשתם את הבדיקה נכון, לכאורה לא יקרה דבר. למעשה, אקליפס הריצה את הבדיקה, הציגה את התוצאות במבט מיוחד שנפתח (מבט JUnit, שנקרא לו מעתה מבט הבדיקות), אבל מכיוון שכל הבדיקות עברו בהצלחה, המבט הזה נשאר מכוסה מאחורי מבט החבילות.



אם נקליק על שם המבט, נראה את תוצאת הבדיקות. JUnit הריצה בדיקה אחרת (שירות אחד ששמו מתחיל ב-test), ובדיקה אחת הצליחה. JUnit מדווחת על 0 שגיאות (errors) ו-0 כשלונות (failures). אם היו כשלונות או שגיאות, פרטיהם היו מופיעים בהמשך המבט. המלבן הירוק לגמרי מציין שכל הבדיקות עברו

בהצלחה. מה ההבדל בין כשלון ושגיאה? כשלון הוא מצב שבו הצהרה בבדיקה, כמו הצהרת ה-assertEquals שלנו, נכשלת. כלומר מצב העצמים אינו המצב שהבדיקה מצפה לו, או שהשאלות מחזירות מידע לא נכון. שגיאה היא מצב שבו שירות נכשל באופן בלתי צפוי, למשל בגלל NullPointerException. בשני המקרים מדובר בפגם בקוד שצריך לתקן (או בבדיקה פגומה). צלמית בסרגל הכלים של המבט מאפשרת לחזור על הבדיקות בלי לצאת מהמבט. זה מאפשר להריץ שוב את הבדיקות לאחר תיקון פגם בעורך.

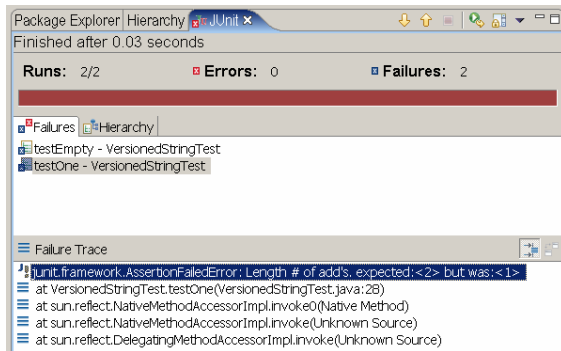
כעת נוסיף עוד שירות בדיקה, testOne, שיבדוק את מצב העצם לאחר הוספת מחרוזת

```
public void testOne() {  
    vs.add("Version 1");  
    assertEquals("Length # of add's.", 1, vs.length());  
    assertEquals("Version 1", vs.getVersion(1));  
    assertEquals("Version 1", vs.getLastVersion());  
}
```

אחת. השירות מוסיף מחרוזת, ואז קורא לשלוש השאלות במחלקה הנבדקת, על מנת לוודא

שהן מחזירות את הערך הנכון. בקריאה הראשונה ל-assertEquals העברנו שלושה ארגומנטים, שהראשון מבניהם הוא מחרוזת. המחרוזת אמורה להסביר למי שמריץ את הבדיקות למה ציפינו לערך מסוים. במקרה הזה אין למחרוזת הזו ערך רב, אבל לפעמים יכול להיות לה ערך תיעודי מועיל. ב-JUnit יש מבחר גדול של הצהרות דומות ל-assertEquals, כמו assertTrue לערכים בוליאניים, assertNotNull, וכדומה, וכולן באות בשתי גרסאות, עם מחרוזת הסבר ובלעדיה. אם נריץ את הבדיקה כעת, נראה ש-JUnit מדווחת על שתי בדיקות מתוך שתיים שהצליחו.

על מנת לראות מה קורה כאשר קוד אינו עובר בדיקה, נשנה את הבדיקה כך שלוש ההצהרות ב-testOne יכשלו. למשל, נצפה לקבל את הערך 2 מהשירות length ונצפה לקבל את המחרוזת "Version 2" משתי השאלות האחרות. כאשר נריץ את הבדיקה, JUnit תדווח על כשלון אחד. משמעות הכישלון היא שאחת ההצהרות בשירות בדיקה נכשלה; JUnit מפסיקה להריץ את שירות הבדיקה כאשר הצהרה כזו נכשלת, ולכן אינה מדווחת על עוד הצהרות לא נכונות, אם יש כאלה. אבל JUnit כן ממשיכה



להריץ שירותי בדיקה אחרים. אם נשנה גם את השירות `testEmpty` כך שהצהרה שלו תהיה לא נכונה, JUnit ידווח על שני כשלונות, אחד בכל שירות בדיקה. הדיווח יראה כמו במסך שמשמאל. בחלק העליון של המבט יש דיווח על שירותי הבדיקה שנכשלו. כאשר בוחרים אחד מהם, החלק התחתון של המסך מראה את

מצב תוכנית הבדיקה בנקודת הכשל. שתי השורות העליונות הן החשובות. בעליונה רואים לאיזה ערך ציפינו מהשאלתה ואיזה ערך (שונה כמובן) קיבלנו, וכן את המחרוזת שהעברנו על מנת להבהיר את הציפיות שלנו, אם העברנו מחרוזת כזו. השורה שמתחת מתארת בדיוק את נקודת הכשל והקלקה עליה תביא אותנו לנקודה הזו בשירות הבדוק בעורך.

הנקודה החשובה להבין היא שמכל שירות בדיקה שאינו מצליח JUnit מדווחת רק על הצהרה אחת שלא מתקיימת. אבל תמיד מריצים את כל שירותי הבדיקה. לכן, אם אנו מסתפקים בהודעה על הבעיה הראשונה שמתגלה, אפשר להכניס הרבה בדיקות לשירות בדיקות אחד. למשל, אפשר לבדוק בשירות אחד עצמים שמייצגים סדרה ריקה של מחרוזות, סדרה באורך 1, סדרה באורך 2, וכדומה. אבל אם רוצים לקבל בבת אחת דיווח על מספר פגמים בתוכנה, צריך לבדוק כל פגם בשירות בדיקה נפרד.

לפני שנמשיך לבדוק עוד מחלקות, נראה כיצד לבדוק חריגים (exceptions). לשירות

```
public void testComparable() {
    class LocalComparable
        implements il.ac.tau.cs.oopj.Comparable {
        public int compareTo(il.ac.tau.cs.oopj.Comparable o) {
            return 0;
        }
    }
    vs.add("Version 1");
    vs.add("Version 2");
    VersionedString vs1 = new LinkedVersionedString();
    vs1.add("Version 1 (1)");
    try {
        assertEquals(1, vs.compareTo(vs1));
    } catch (IncomparableException ie) { fail(); }
    try {
        int c = vs.compareTo(new LocalComparable());
        fail();
    } catch (IncomparableException ie) {
    }
}
}
```

compareTo
Versioned-String
מותר להודיע על חריג מסוג Incomparable-Exception. אנו רוצים לוודא שהשירות מחזיר ערך נכון כאשר העצם המועבר לו הוא בר השוואה לעצם שמפעיל את השירות,

ושהוא מודיע על חריג אחרת. נוסיף שירות `testComparable` בשירות הבדיקה. השירות יגדיר מחלקה שממשת את `Comparable` אבל היא לא ברת השוואה ל-`VersionedString`, ובונה עוד עצם מסוג `VersionedString` שהוא כן בר השוואה לעצם `vs`. כאשר משווים את `vs` לעצם בר השוואה, אנו מצפים שלא נקבל הודעה על חריג, ולכן הצהרנו שאם מגיעים לפסוק ה-`catch`, הבדיקה צריכה להיכשל. השירות `fail` הוא שירות של JUnit ש-"סוגר" שורות בשירות הבדיקה שלא אמורים להגיע אליהם. לעומת זאת, כאשר משווים את `vs` לעצם שאינו בר השוואה ל-`VersionedString`, אנו מצפים שכן נקבל הודעה על חריג, ולכן נקרא ל-`fail` מייד לאחר הקריאה ל-`compareTo`.

כעת הגיע הזמן לבדוק את שתי המחלקות שמרחיבות את `VersionedString`, את `LinkedVersionedString` ואת `ArrayVersionedString`. למעשה, כבר בדקנו את הראשונה, אבל אנו ניצמד למדיניות בדיקה קפדנית שבה יש לבדוק לחוד מחלקה מופשטת ולחוד את המחלקות שמרחיבות אותה.

דרך פשוטה לבדוק את המחלקות הללו היא להעתיק את מחלקת הבדיקה. אבל יש דרך טובה יותר. אנו נשאיר את הבדיקות של השירותים שמוצהרים ב-`VersionedString` במחלקת הבדיקה `VersionedStringTest`, אבל נוציא ממנה את אתחול העצמים. את האתחול נעביר למחלקות שתפקידן לבדוק את המחלקות המוחשיות המרחיבות. ניצור בעזרת האשף מחלקת בדיקה נוספת, `LinkedVersionedStringTest`, אבל נגדיר את מחלקת הבסיס שלה להיות `VersionedStringTest`, לא ברירת המחדל `TestCase`

```
import il.ac.tau.cs.oopj.*;

public class LinkedVersionedStringTest
    extends VersionedStringTest {

    protected void setUp() throws Exception {
        super.setUp();
        vs = new LinkedVersionedString();
    }

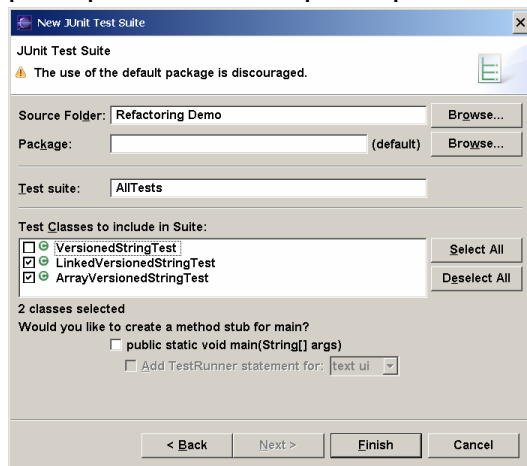
    protected void tearDown() throws Exception {
        super.tearDown();
        vs = null;
    }

    public LinkedVersionedStringTest(String name) {
        super(name);
    }
}
```

ממחלקת הבסיס שלה. נגדיר בצורה דומה גם מחלקת בדיקה עבור `ArrayVersionedString`. כמוכן שגם את המשתנה `vs1` שהשתמשנו בו בשירות הבדיקה `testComparable` כדאי להגדיר במחלקות המרחיבות ולא במחלקת הבסיס, כדי שהבדיקה של מחלקת הבסיס לא תהיה תלויה ב-`LinkedVersionedString`.

מהחבילה `junit.framework`. במחלקה הזו נתחיל את `vs` ונשחרר אותו, ונסיר את השורות המקבילות מ-`VersionedStringTest`. כעת אי אפשר יותר להריץ את `VersionedStringTest` (נסו ותראו מה קורה), אבל אם נריץ את מחלקת הבדיקה החדשה, נראה שהיא מבצעת שלוש בדיקות, למרות שאינה מגדירה אף בדיקה. הבדיקות שהיא מבצעת הן כמוכן הבדיקות שהיא ירשה

כעת יש לנו שתי מחלקות בדיקה מוחשיות ומחלקת בדיקה נוספת שמתפקדת רק



כמחלקת בסיס. אנו יכולים להפעיל את כל אחת ממחלקות הבדיקה, אבל כאשר מצטברות מחלקות בדיקה רבות, הפעלה ידנית שלהן מייגעת. על מנת להפעיל מחלקות בדיקה רבות בבת אחת, מגדירים מחלקה שמייצגת קבוצת בדיקות, `Test Suite`. קבוצה כזו יכולה להכיל מחלקות בדיקה בודדות וגם קבוצות שלמות. אפשר לייצר קבוצה כזו בעזרת אשף, על ידי בחירת `New`, `Other`, ו-`JUnit Test Suite`. המסך השני של האשף מאפשר לבחור

את מחלקות הבדיקה (וקבוצות בדיקות) שהקבוצה החדשה תריץ. איננו רוצים להריץ את `VersionedStringTest`, רק את שתי מחלקות הבדיקה האחרות. נבחר בהתאם ונסיים. כאשר נריץ את הקבוצה, `JUnit` יריץ 6 בדיקות, 3 מכל מחלקת בדיקות.

לפני שנסיים את השיעור, נעביר את מחלקות הבדיקה לחבילה נפרדת עם שם. ניצור בפרויקט חבילה בשם `unittests`, ונגרור אליה את כל מחלקות הבדיקה. הגרירה מבצעת למעשה תהליך של `refactoring`, ואם נבחן את אחת ממחלקות הבדיקה, נראה שנוסף לה פסוק `package` בראש הקובץ. ההפרדה מיועדת להקל על אריזת התוכנה בקובץ `jar` לקראת הפצתו ללקוחות: ללקוחות נרצה להפיץ את התוכנה עצמה, אבל בדרך כלל לא את מחלקות הבדיקה.

לסיום, שלוש הערות על בדיקות יחידה. ראשית, למיקום קוד הבדיקה יש חשיבות לגבי תוצאות הבדיקה. אם קוד הבדיקה שייך לאותה חבילה כמו הקוד הבודק, אזי קוד הבדיקה יכול להפעיל שירותים ומחלקות עם נראות חבילה (`package visibility`) שלקוחות בחבילות אחרות לא יכולים להפעיל. במילים אחרות, מיקום הבדיקה בחבילה של המחלקה הנבדקת מסתיר פגמים של שימוש בנראות חבילה במקום בנראות ציבורית. לכן, אם הקוד הנבדק מיועד לשימוש על ידי חבילות אחרות, גם קוד הבדיקה שלו צריך להיות חלק מחבילה אחרת. הבדיקות היחידות שצריך למקם באותה חבילה יחד עם המחלקות הנבדקות הן בדיקות שבודקות שירותים ומחלקות לא ציבוריים, אלא עם נראות חבילה.

שנית, `JUnit` מספק מנגנון בדיקות, אבל לא מדיניות בדיקות. את המדיניות אתם צריכים לקבוע. המדיניות קובעת איזה מחלקות ושירותים בודקים, כיצד בודקים, היכן ממקמים את קוד הבדיקה, וכדומה. על מנת למזער את כמות הפגמים בתוכנה שלכם, חשוב לקבוע מדיניות בדיקות ולדבוק בה. המדיניות צריכה לאזן בין הרצון למצוא פגמים ובין עלות פיתוח והרצת הבדיקות.

שלישית, מכיוון ש-`JUnit` משתמשת ב-`reflection` על מנת למצוא את שירותי הבדיקה (השירותים ששםם מתחיל ב-`test`), חיוני לשמור על חתימה של `public void` עבור שירותים אלו. הגדרתם בנראות נמוכה יותר תסתיר אותם מ-`JUnit` ותגרום לכך שלא יתבצעו. למרבה המזל, `JUnit` תתלונן אם תגדירו בדיקה בנראות לא ציבורית.

שימת ההצהרות העיקריות המוגדרות ב-`TestCase` (לכולן יש גרסה עם מחרוזת הסבר כארגומנט ראשון):

<code>assertTrue</code>	(condition)
<code>assertFalse</code>	(condition)
<code>assertEquals</code>	(expected, actual)
<code>assertSame</code>	(expected, actual)
<code>assertNotSame</code>	(expected, actual)
<code>assertNull</code>	(reference)
<code>assertNotNull</code>	(reference)
<code>fail</code>	()