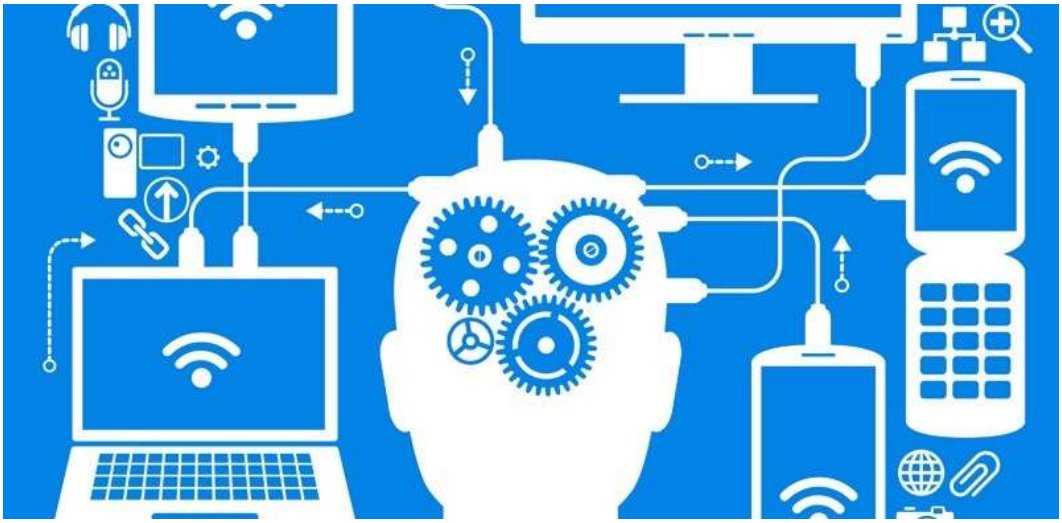
Introduction to Software Engineering

JERUSALEM COLLEGE OF TECHNOLOGY
LEV ACADEMIC CENTER



Recitations

- *Week 0: Home assignment 0 (not graded): Introduction to OOP with Java.*
- *Week 1: Home assignment 1: Implementation of primitives with operations, geometries*
- **Week 2: Home assignment 2**: Implementation of geometries primitives' unit testing for primitives (JUnit). Implementation of normal calculation and their unit testing through geometries.
- *Week 3: Home assignment 3: Implementation of ray-geometry intersections and their unit testing through geometries.*
- *Week 4: Home assignment 4: Implementation of camera class, rays through view plane construction and unit testing of camera*
- *Week 5: Home assignment 5: Implementation of ambient light, scene, render and image-writer classes with their appropriate test units*
- *Week 6-7: Home assignment 6: Adding material support. Implementation of Phong model, with light emission, directional, point and spot lighting, multiple light sources*
- *Week 8: Home assignment 7: Implementation of shadow rays, reflection and refraction*
- *Week 9: Mini-project 1: Implementation of a picture improvement algorithm*
- *Week 10-11: : Mini-project 2: implementation of a performance improvement algorithm*
- *Week 12-13: Mini-projects presentation*

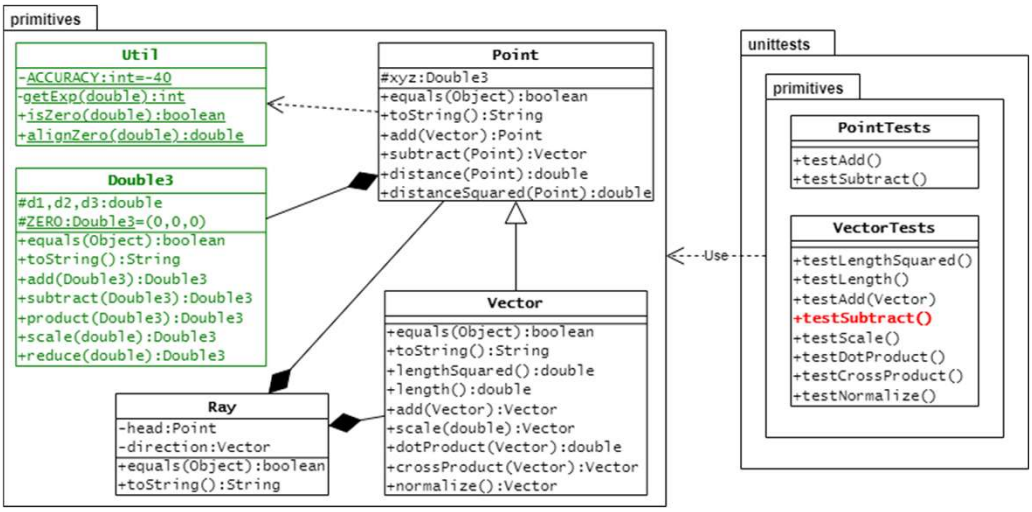
Home assignment 2

Implementing the primitives and geometries (normal vector) testing units

Implementing normal vector calculations for geometry+point

Bonus (1pt): Test and implement correctly normal for Cylinder

Our architectural design + Test Driven Design



קודם כל הוסיפו לפרויקט שלכם בסביבת הפיתוח שלכם את ספריית Junit (גרסה 5 – Jupiter) לפי ההדרכה שקיבלתם על הנושא בכיתה. הדרכה בסיסית מופיע בהמשך המצגת הזו, כמו כן במצגת נפרדת.

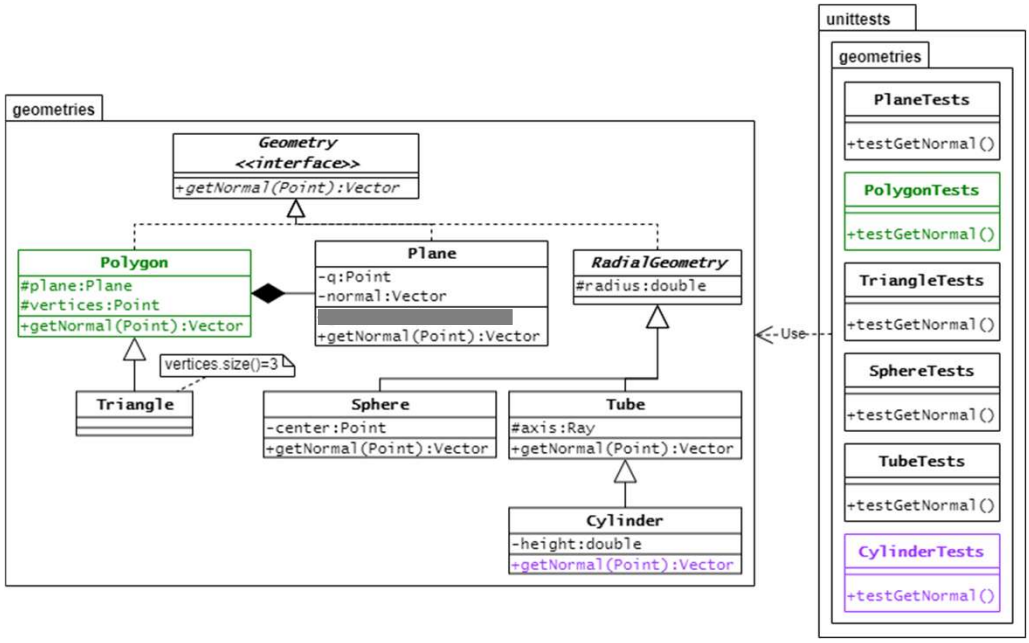
לאחר מכן : ב-eclipse קודם כל תיצרו חבילה עבור בדיקות unit tests בשם unittests כנ"ל. מי שעובד ב-IntelliJ IDEA – תיקיות של בדיקות נוצרות בצורה שונה והן מקבלות וגם שמות החבילות מתנהלות שם קצת אחרת – אנא עקבו אחרי הדרכות של מר אליעזר גנסבורגר איך לעבוד עם Junit בסביבה הזאת.

<u>שימו לב</u> : בשקף הבא תראו שאתם מקבלי מודול PolygonTests.java – אך כבר עכשיו תעבירו אותו לתת-חבילה geometries של החבילה unittests ותעיינו במודול הזה על מנת ללמוד איך לארגן את הבדיקות וגם איך לכתוב את התיעוד – גם חיצוני וגם פנימי.

הפורמט הזה הוא **<u>חובה</u>** בכל המודולים של בדיקות.

לאחר מכן תוסיפו תת-חבילה primitives בחבילה unittests ובה תוסיפו שתי מחלקות לבדיקות מחלקות Point ו-Vector. במחלקות האלה תיצרו פונקציות לבדיקת פעולות שיצרתם בתרגיל 1 במחלקות הנבדקות. **<u>אסור</u>** להעתיק את הבדיקות מהתוכנית הראשית של תרגיל 1 – אלו בדיקות בשיטה מהעידן שלפי Junit. אמנם תראו את רוב הבדיקות הנדרשות בתוכנית הראשית ההיא – אבל עליכם ליצור בדיקות חדשות בעזרת Junit ולוודא שלא שכחתם שום בדיקה שנדרשת ע"פ החלוקה למחלקות שקילות (EP) וניתוח ערכי גבול\קצה (BVA) כפי שלמדתם בקורס התיאורטי.

Our architectural design + Test Driven Design



בשלב השני של התרגיל תיצרו בתת-חבילה geometries של החבילה unittests מחלקות לבדיקת מחלקות גופים גיאומטריים ותיצרו בדיקות של פונקציה getNormal עבור כל מחלקת גוף גיאומטרי. הקפידו על כך שהבדיקות נכתבות **לפני** שאתם ממשים את הפונקציות הנבדקות. תתעלמו מהמימוש העתידי – גם אם הוא כבר קיים לכם בראש. הקפידו על תכנון הבדיקות ע״פ מחלקות שקילות וניתוח ערכי גבול\קצה. הקפידו על הפורמט ועל התיעוד של בניית הבדיקות. שימו לב שגם במשולש אתם תוסיפו בדיקות עבור הפונקציה getNormal, אפילו שבפועל לא תצטרכו לממש אותה (תחשבו **מדוע**!).

השתדלו למקם את הגופים ולבחור נקודה שבה תיצרו את וקטור הנורמל לכל בדיקה – בדרך הכי פשוטה ונוחה. על תחפשו לסבך את עצמכם אלא תחפשו להקל על עצמכם. תזכרו ששלשות פיתגורס – מאד שימושיות למציאת טסטים נוחים לכתביה (תזכורת – שלשת פיתגורס – הינה שלשת מספרים שלמים שסכום ריבועים של שניים מהם שווה לריבוע של המספר השלישי).
רק לאחר השלמת כתיבת הבדיקות (כמובן הבדיקות נכשלות לפני השלב הזה), תעברו למימוש בתוך הפונקציות getNormal.

במישור גם תממשו את הבנאי שמקבל 3 נקודות. על תעשו בבנאי פעולות מיותרות. תחשבו למה לא צריך לזרוק חריגות מתוך הבנאי בור נקודות מתלכדות או עבור מצב ששלושת הנקודות נמצאות על אותו הישר. בגליל יש מקרה קצה – תגלו אותו ותוודאו שאתם בודקים את המקרה הזה.
הוספת בדיקות ומימוש של getNormal עבור גליל סופי (Cylinder) בינו לבונוס, ואתם לא חייבים לעשות את זה. יש כאן מספר מחלקות שקילות ומקרי גבול. שימו לב שנק׳ בונוס עבור בדיקות ומימוש הנורמל בגליל סופי ניתנן רק יחד עם הצגת השלב הזה. אם תעשו את הבונוס הזה מאוחר יותר – לא תקבלו ניקוד עליו.

How to implement tests?

- Adding JUnit library to Java build path
- Using the @Test annotation
- Using ***assertTrue***, ***assertFalse***, ***assertEqual***, ***assertThrows***, ***fail***, etc

בשקפים הבאים נרחיב ונדגים איך להוסיף ספרית Junit לפרויקט ע״י הוספת הסיפריה לנתיב בניית פרויקט ה-JAVA, איך להשתמש באנוטציה @Test ואיך להשתמש בפונקציות בדיקה של הספריה

What is JUnit?

- Lightweight framework
- Used to write and run repeatable tests
- Open source
- Originally written by Erich Gamma and Kent Beck

ספריה Junit הנה ספריה "קלת משקל" (קטנה בגודלה וקלה לשימוש). מטרת הסיפריה – לתת כלים לכתיבה והרצה של בדיקות (טסטים) שניתן לחזור ולבצע אותם כל פעם שנרצה. הסיפריה היא קוד פתוח (איננה מסחרית). במקורה היא נכתבה ע"י שני אנשים ששמעתם את שמותיהם בקורס התיאורטי גם בקשר לתבניות עיצוב (design patterns), וגם בקשר לפיתוח זריז (Agile).

Why Use JUnit?

- Easy and convenient
- Versatile
- Runs in most IDE’s (built in)
- Used to build a relationship between the development and testing process
- Automation
- Saves time
- Less prone to errors

יתרונות של ספריית Junit הם למשל:

- קלות ונוחות שימוש בה
- גיוון אפשרויות בדיקה
- נתמך ברוב סביבות הפיתוח
- מאפשר הבניית קשר בין תהליכי פיתוח לבין תהליכי בדיקות
- אוטומציה של טסטים (הרצה אוטומטית של קבוצת טסטים או של כל הטסטים)
- חיסכון זמן בביצוע בדיקות
- כתיבת בדיקות בעזרת הסיפריה פחות נוטה ליצירת טעויות

JUnit vs. Print statements

- JUnit eliminates need for LMAO (looking manually at output)
- JUnit eliminates need for LMFAO (looking manually for abnormal output)
- Much simpler to write a test to check then to run many output statements and check the output of each statement
  - Eliminates time
  - Eliminates chance of error
- Print statements within a class – do not reflect an object's true interface to the outside world
  - Need to test methods from within locations that do not have access to protected and private members.
- Easier to reuse when checking results of few changes in code.

שימוש בספריית Junit לעומת שימוש הדפסות בדיקה בעזרת תוכנית ראשית לבדיקות:

- מעלים צורך ב-LMAO (סריקה ידנית של פלט של בדיקות)
- מעלים צורך ב-LMFAO (חיפוש פלט של תקלות)
- הרבה יותר פשוט לכתוב טסט מאשר לכתוב סדרה של הדפסות ובדוק תוצאה של כל ביטוי: חוסל זמן וחוסך טעויות
- כתיבת הדפסות דיבג בתוך מחלקות ונקציונליות משבש את התהליך ההנדסי – המחלקות האלה לא מיועדות לתקשר לעולם החיצון דרך הדפסות בקונסול
- מתוכנית ראשית לא ניתן לבדוק פונקציות עם הרשאה private\protected (בעצם בשתי השיטות)
- הרבה יותר קל לשימוש חוזר ונשנה (בתדירות גבוהה) לאחר כל שינוי או כמה שינויים קטנים בקוד
- יותר קל לנתח תוצאות לאחר מספר שינויים בקוד

JUnit vs. Debugging

- Debugging requires manually stepping through many lines of code
- Using JUnit saves time
- Debugging generally checks a few cases
- JUnit can easily check many more cases

שימוש בספריית Junit לעומת דיבג:

- דיבג דורש מעבר פקודה-פקודה צעד-צעד על הרבה שורות קוד, וביצוע מעקב אחרי ערכי משתנים מקומיים וכו'
- לכן שימוש בספריה Junit חוסך המון זמן
- כמו כן דיבג מאפשר בדיקת של כמות מצומצמת של תסריטים, כאשר בעזרת הסיפריה נוכל לתכנן הרבה יותר בדיקות ובכך לכסות הרבה יותר מקרים ולהבטיח איכות גבוהה יותר של הקוד

When Should Test be Written

- Before the code!
- Good programming practice to first design the code and then write it.
- Writing tests first effectively defines the intended use for code.
- Easier to debug code if no bugs are present.

מתי כותבים טסטים?

- **לפני שמממשים את הפונקציה הנבדקת!**
- הגישה הנכונה – קודם כל לתכנן (דיזיין)
- ואז לכתוב בדיקות ע״פ הדרישות וע״פ הדיזיין – כך בדיקות יבדקו את השגת המטרות של הקוד שנכתוב בהמשך
- ואז גם יותר קל לדבג כאשר נדרש

Types of Tests

- Run reasonable tests that test functionality of class methods
- No need to test things that are “too simple to break”
  - Aka get methods etc.

How to Run Tests

- Tests should always run at 100%
- Tests should be run every time there is a change in code

סוגי בדיקות:

- תכתבו טסטים הגיוניים שבודקים את הפונקציונליות של הפונקציות הנבדקות – השתמשו בשיטת מחלקות שקילות וניתוח מקרי גבול\קצה, בחרו נתונים נוחים לחישוב תוצאות צפויות של בדיקות
- אין צורך לבדוק פונקציונליות בסיסית (למשל גטרים וסטרים)

איך מריצים בדיקות?

- הטסטים חייבים להסתיים בכך שהם מצליחים 100%
- מריצים את הטסטים כל פעם שעושים שינוי בקוד
- אם הטסט לא עובד – אל תתאימו את הטסט לתוצאות הפונקציה – **זו אחיזת עיניים**! חפשו את הטעות. הטעות יכולה להיות גם בטסט עצמו או בנתוני הטסט – אך ברוב המקרים היא תהיה בפונקציה הנבדקת!

How to Write a JUnit Test

- Can write a test for class method
- Use the @Test annotation
- Combine the different tests into a JUnit suite

Assert Tests in JUnit

- Assert = הכרזה
- Tries to assert that a certain thing is true
- If it is true then the assert passes otherwise it fails.

איך כותבים טסט:

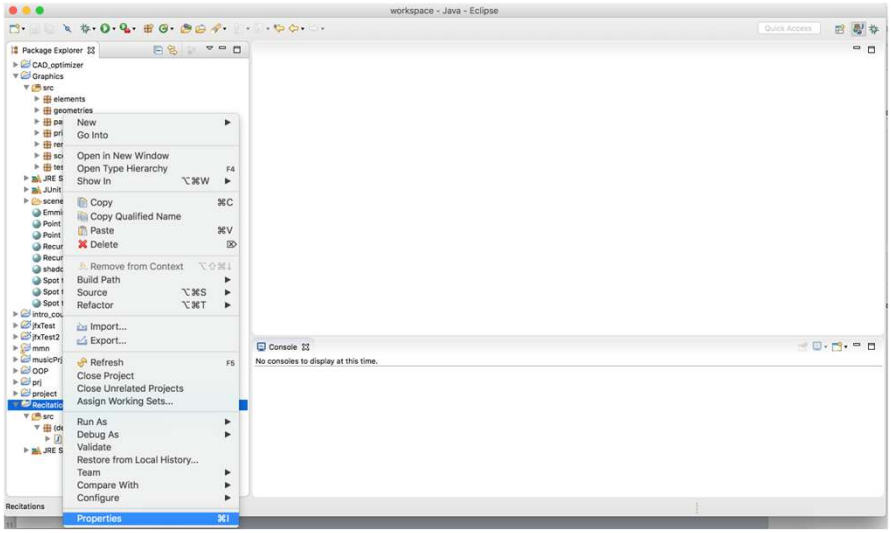
- כותבים מחלקת בדיקות עבור כל מחלקה נבדקת (בדרך כלל)
- כותבים פונקציית בדיקות עבור כל פונקציה נבדקת (בדרך כלל)
- ניתן לכתוב מערך (סוויטת) בדיקות – אז זה לא נדרש בקורס שלנו ולא נתעמק בכלי הזה של Junit כמו גם בעוד כמה כלים שהספריה נותנת לנו ואנחנו לא מתעסקים אתם בקורס שלנו

בדיקות "**טענה\הכרזה**" ב-JUnit

- בתסריט בדיקה אנחנו טוענים (או מכריזים) שהפעלת פונקציה מסוימת עם נתונים מסוימים חייבת להחזיר לנו תוצאה מסוימת
- בעצם נבנה בדיקות ע"י אימות ההכרזות האלה – אם תוצאה היא "**אמת**" אזיי הטסט הצליח, אחרת הטסט נכשל

Adding JUnit

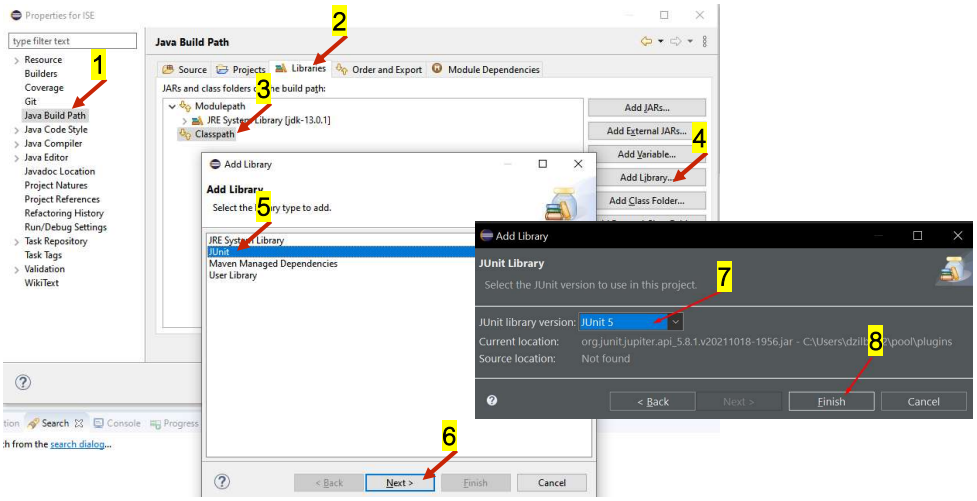
- Go to properties of project



פתיחת מאפייני הפרויקט ב-eclipse

Adding JUnit

1. Go to **Java Build Path**
2. Switch to **Library** tab
3. Choose **Classpath**
4. Press **Add Library…** button
5. Choose **JUnit**
6. Press **Next>** button
7. Choose **Junit 5**
8. Press **Finish** button



הוספת הספרייה במאפייני הפרויקט

Example

- Using the **@Test** annotation
- Using ***assertEqual***, ***assertThrows***

```
class VectorTest {
   ...

   @Test
   void testNormalize() {
      Vector v = new Vector(0, 3, 4);
      Vector n = v.normalize();
      // ============ Equivalence Partitions Tests ==============
      // TC01: Simple test
      assertEquals(1d, n.lengthSquared(), 0.00001, "wrong normalized vector length");
      assertThrows(IllegalArgumentException.class, () -> v.crossProduct(n), //
                   "normalized vector is not in the same direction");
      assertEquals(new Vector(0, 0.6, 0.8), n, "wrong normalized vector");
   }

   ...
}
```

לפני כל פונקציית בדיקה נוסיף אנטציה @Test שמסמנת ל-Junit שזאת פונקצית בדיקות.

בתוך פונקצית בדיקות נשתמש במגוון פונקציות assertXXXX של Junit על מנת לאמת טענות\הכרזות – למשל שוויון תוצאה בפועל לתוצאה צפויה, זריקת חריגה מסוימת במצב מסוים, ועוד...

שימו לב על פורמט בדיקת זריקת חריגה – משתמשים בפונקציית לאמבדה של java.

List of Assert Tests of JUnit

- Based on calling **equals()** function
- In every function a string parameter can be added with a message for failure

המתודה	הסבר
assertEquals(expected, actual)	בודק האם parm1 שווה ל-parm2
assertEquals(expected, actual, delta)	בודק האם par1 שווה ל-par2 עם הפרש לא גדול מ-par3
assertNotEquals(expected, actual)	בודק האם parm1 לא שווה ל-parm2
assertNotEquals(parm1, parm2, parm3)	בודק האם par1 לא שווה ל-par2 עם הפרש לא גדול מ-par3
assertTrue(parameter)	בודק האם לפרמטר יש הערך של אמת
assertFalse(parameter)	בודק האם לפרמטר יש הערך של שקר
assertNull(object)	בודק האם האוביקט שווה ל-null
assertNotNull(object)	בודק האם האוביקט שונה מ-null
assertSame(obj1, obj2)	בודק האם obj1 מצביע לאותו אובייקט ש-obj2 מצביע עליו
assertNotSame(obj1, obj2)	בודק האם obj1 לא מצביע לאותו אובייקט ש-obj2 מצביע עליו
assertArrayEquals(expected, actual)	בודק האם התוכן של מערך הראשון שווה לתוכן של מערך השני
assertNotThrow(exec)	בודק שה-exec לא זורק חריגות
assertThrows(exception, exec)	בודק שה-exec זורק חריגה exception
fail()	מכשיל טסט

לפניכם רשימת פונקציות assert של Junit בחתימותיהם השימושיות. לפונקציות האלה יש הרבה העמסות לפי טיפוסים אפשריים של הנתונים הנבדקים – מספרים, אובייקטים, וכו'. כמו כן לכולן יש העמסות המאפשרות הוספת מחרוזת בפרמטר הראשון – המחרוזת תוצג **במקרה של כישלון** הטסט.

פונקציות assertEquals ו-assertNotEquals :

- על מספרים עם נקודה צפה (לא שלמים) – מאפשרים פרמטר נוסף שמאפשר להכריז שערכים מסוימים שווים כאשר ההפרש ביניהם קטן מ"דלטא" מסוימת (הניתנת בפרמטק נוסף
- על אובייקטים – משתמשת בדריסת הפונקציה equals באובייקטים המתאימים – למשל Point, Vector, וכו'

אם אתם רוצים לבדוק שוויון מספר – אל תעשו assertTrue לביטוי השוואה! תפעילו assertEquals

אם אתם רוצים לבדוק תנאי מסוים (ביטוי בולאני) – רק אז מתאים assertTrue\False

אם אתם רוצים לבדוק האם קיבלתם null או לאו – אל תשתמשו ב-assertEquals/NotEquals עם ערך צפוי של null. במקום זה תשתמשו ב-assertNull/NotNull

אם אתם רוצים לבדוק שבתוצאה יש בדיוק אותו (או לא אותו) האובייקט (לא אובייקט שווה אלא ממש אותו האובייקט – אותה ההפניה\הכתובת) – השתמשו ב-assertSame/NotSame

רק במקרים קיצוניים תשתמשו במתודת fail תחת תנאי – השתדלו למצוא שימוש באחד הפונקציות מתאימות במקום זה