

תכנות מונחה עצמים – תרניל 4

PEPSE: Precise Environmental
Procedural Simulator Extraordinaire

תאריך הגשה: 04.01.26 בשעה 23:55

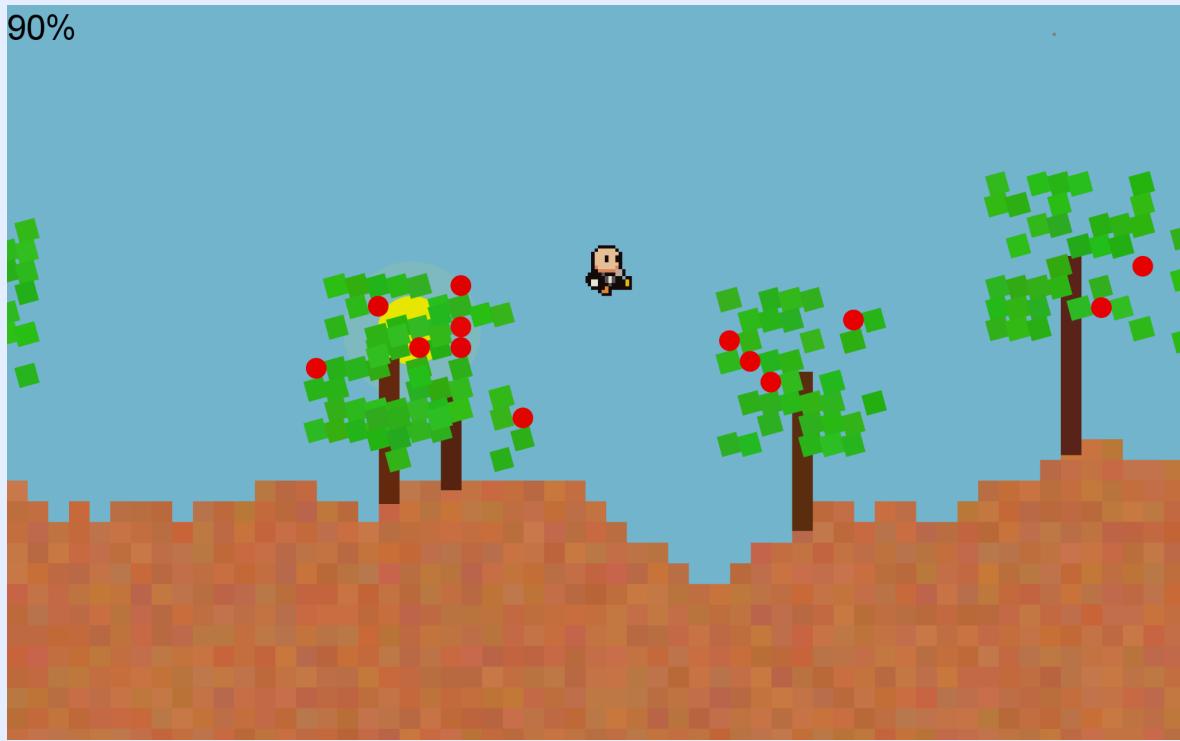
תאריך הגשת המוקדם ביותר הוא 28.12.2025 בשעה 23:55
(ראו הסבר בפרק 9.3)

תרניל זה מומלץ להגיש בחוגות

0. הקדמה

היום אנחנו מכינים סימולציה, אוCLI (ממשק משתמש בו אבל שלא מכיל תנאי ניצחון או הפסד, בשלב זה). הכירנו את הסימולטור הפרטורי הנאמן לחוטין למציאות של... המציאות. הנה היא:

90%



הסימולציה כוללת מחזור יום-לילה, לרבות החשכה של המסר (כמו במציאות!), דמות אוטומטית שיכולה לרחוץ ולקפוץ, עצים עם עליים הנעים ברוח, ופירות שנוטנים אונרנית לדמות. סימולציה כזו אפשר יהיה לפתח בעתיד לאחד מכמה כיוונים מתבקשים: הוספה אלמנט של מים וניהם, פיתוח משחק פלטפורמה, משחק בניה, או משחק הישרדות. פונקציונלית, הסימולציה מאוד מודולרית. ניתן לראות סרטון דמו [כאן](#).

0.1 מטרות התרגיל

- .0.1.3 פיתוח הסימולציה הוא תירגול מצוין לתוכנות באופי מעט אחר, שבו ריבוי האסטרטגיות הופך את הייצורה של עצם למורכבת, עד שלא יוכל לדוחף עוד את הכל למחלקה אחת – נרצה לחלק את מלאכת ייצורת העצמים עצמה לקבצים שונים ושיטות שונות. למעשה, הנדרת האסטרטגיות ויצירת העצמים תהווה חלק נכבד מהקוד שלנו, ובפנוי עצמה לא תיכתב בצורה מונחית עצמים. התרכנו עד כה בתוכנות מונחה עצמים כי זה הנושא שלנו, אבל בהמשך תכינו פרדינומות תכונות אחרות וכזו יקרה תיזכרו בנוסטלגיה שאמרנו לכם שאין הכרח שקוד יהיה 100% בפרדינמה אחת; כל פרדינמה היא כל' בארגז. התרגיל יהיה הוצאה לך.
- עד סוף המדריך, אתם אמורים להרגיש בנח עם נושאים כמו הנדרת למבוזת, שימוש ב-method references, callbacks, ושליחת .callbacks.
- .0.1.4 .0.1.5 أولי החשוב מכלום: מהקנת הסימולציה תמשכו ללמידה להנות מתוכנות.

0.2 עיצוב ייצורת לצד הפרויקט

```
src/
└── assets/
    └── avatar
        └── images
pepse/
    ├── PepseGameManager.java
    └── pepse.utils/
        ├── ColorSupplier.java
        └── NoiseGenerator.java
    └── pepse.world/
        ├── Sky.java
        ├── Block.java
        ├── Terrain.java
        └── pepse.world.daynight/
            ├── Night.java
            └── Sun.java
            └── SunHalo.java
        └── pepse.world.avatar/
            └── Avatar.java
    └── pepse.world.trees/
        └── Flora.java
```

README

על הפרויקט שלכם לכלול את החבילות הבאות:

• pepse: החבילה הראשית עם מנהל המשחק (PepseGameManager).

<ul style="list-style-type: none"> • utils.pepse: מחלקות עוזר. <p>כרנו מילה מחלוקת אחת שמייצרת וריאציות של צבעים, ומחלוקת נוספת נספחת שתעזר לכם לייצר אקראיות למשחק.</p> <ul style="list-style-type: none"> • pepse.world: אחראית על יצירת העולם. • pepse.world.daynight: מכילה פונקציונליות הנוגעת למחרור יום-לילה. • pepse.world.avatar: מכילה את הקוד הנוגע לדמות. • pepse.world.trees: מכילה את הקוד הנוגע ליצירת עצים. <p>שימו לב: כדי לאפשר לכם להתאמן גם בהחלטות עיצוב, סייפקנו לכם תרשימים חלקית בלבד. כדי לכסות את כל דרישות התרגnil, נראה שתוכנאו את עצמכם מוסיפים חבילות ו/או מחלוקות נוספות.</p> <p>דגש חשוב: כל מה שלא מוגדר באופן מפורש בהנדרות התרגnil (מייקום/גולד/התנהנות וכו') נתון להחלהתכם.</p>

0.3. תחילת עבודה

<p>ועכשו, ניגש למלאכה.</p> <p>שים לב! יש לקרוא את כל חלקו התרגnil לפני שמתחילה בשימוש.</p> <p>במיוחד חלקים 6 והלאה, חשוב לתכנן ייחד את העיצוב שלהם לפני המימוש.</p> <p>בתוור התחילה, הגדרו פרויקט בשם Pepse עם תלוות ב-DanoGameLab. אפשר להיעזר בפרק 6 של מדריך ההתקנות. לאחר מכן צרו את החבילות ומחלוקתם לעיל; השאירו את המחלוקות ריקות. אנחנו נמלא אותן בהמשך. יצאת הדוף היא המחלוקת ColorSupplier שנשתף אתכם. לבסוף, הגדרו את PepseGameManager כמחלקה בת של GameManager, הוסיףו את ה-main הבא:</p> <pre>public static void main(String[] args) { new PepseGameManager().run(); }</pre>

ודرسו באופן ריק את השיטה [initializeGame](#). הריצו כדי לוודא שנפתח חלון ריק במסך מלא, ממנה אתם יכולים לצאת עם Esc.

cutת נתחיל להגדיר את הדרישות על פי הסדר הבא :

- .1. [শ্মাই](#)
 - .2. [কৰকু](#)
 - .3. [আৰ ও চোশৰ](#)
 - .4. [শমশ](#)
 - .5. [হইলত শশমশ](#)
 - .6. [হোসপত দমোত](#)
 - .7. [জমাহা](#)
- .7.3. [তনোদত উলিম বৰো](#)
 - .7.4. [হোসপত ফিরোত](#)

.8. [עולם אינסופי](#) לאחר שנסממש את הפונקציונליות הבסיסית, נחוור אחריה ונעדקן את המחלקות שמשמשו אותנו ליצירת הקרקע והעצים, כך שייתמכו בעולם אינסופי שנבנה עם התנועה של הדמות.

cutת נתיחה בהנדרת המחלקות:

1. שמות

להלן חלקו קוד והצעות שיעזרו לכם למש את הממשק. שימוש לב שאנו נותרנו נזננים לכם את הקוד כדי שלא רק תעתקו ותדביקו אלא שגמ תיזכרו איך לעשות את זה בעצמכם.

כדי ליצור מלבן תכל ברקע, נתיחיל ונגידיר במחלקה `Sky.pepse.world` שיטה סטטית בשם `:create`

```
public static GameObject create(Vector2 windowDimensions)
```

השיטה מקבלת את גודל החלון, ומוחירה את השמיים שנוצרו.
צבע השמיים בו השתמשנו הוא:

```
private static final Color BASIC_SKY_COLOR = Color.decode("#80C6E5");
```

הנדירו ב-`create` את ה-`GameObject` שמייצג מלבן בצבע השמיים:

```
GameObject sky = new GameObject(  
    Vector2.ZERO, windowDimensions,  
    new RectangleRenderable(BASIC_SKY_COLOR));
```

ליתר ביטחון הנגידו שהמלבן זו עם המצלמה, כך שלא ישאר מאחור אם וכאש המצלמה תזוז:

```
sky.setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES);
```

במהלך הדיבוגים, מכיוון שהרבה מהעצמים הולכים להיות מופיעים ישירים של `GameObject`, עשוי לעזור לכם להבדיל ביניהם באמצעות השדה `tag`:

```
sky.setTag("sky");
```

התנית חסרת משמעות מבחןת המנווע – היא רק לשימושכם.
לבסוף החזירו את `sky` המשיטה.

از הנדרנו פונקציה שאחריות לייצר השמיים. קראו לה מ-`initializeGame` של `PepseGameManager` ודאננו להוסיף את האובייקט שחזור מהפונקציה לשכבה מתאימה ע"י קראיה ל:

```
gameObjects().addGameObject(sky, skyLayer);
```

(שים לב שגםם משתמשים ב- `windowController` כדי לקבל את ה`dimensions` של המשחק). הריצו; קיבלתם שמיים בצלע שמיימי קרע לחלון הריק? (בשלב זהה אמרו להיות לכם מסך עם רקע כחול בלבד). אם תרצו, בשלב זה תוכלו להוסיף לשם גמ עכנים, ציפורים, צלחות מעופפות וכל דבר שעולה בראשכם.

2. קרייז

2.1 המחלקה Block

עכשו ננדיר את המחלקה `Block` בקובץ `java.Block`, שתיהה הבסיס לכל המלבנים הללו שראויים על המסך:

```
public class Block extends GameObject{
    public static final int SIZE = 30;

    public Block(Vector2 topLeftCorner, Renderable renderable) {
        super(topLeftCorner, Vector2.ONES.mult(SIZE), renderable);
        physics().preventIntersectionsFromDirection(Vector2.ZERO);
        physics().setMass(GameObjectPhysics.IMMOVABLE_MASS);
    }
}
```

כפי שניתן לראות, אובייקט `Block` כמעט זהה ל-`GameObject` רגיל, רק שמנדרים עבורה כמה מאפיינים ייחודיים:

- 2.1.1. הגודל שלו קבוע (ר' הקריאה ל-`super`).
בשורה הבאה מונדר שעצם אחר לא יכול לחלוף על בני העצם הזה, אף כיון (פרמטר `after`-`Vector2.ZERO`) היה מנדיר שהבלוק הזה חוסם תנועה ורק מכיוון אחד). שימו לב שתנועה זו תיחסם רק בין שני עצמים שמעוניינים בכך (קרי, שהטיטה זו נקראה עבור כל אחד מהם בנפרד). בנוסף, לשורה זו אין אפקט קר או קר עבור עצמים שבלאו היכי לא מונדרת התנגשות בין עצם זה, למשל עצמים שבשכבה (`layer`) שלא מונדרת התנגשות בין לבין השכבה של עצם זה.
- 2.1.2. השורה الأخيرة אומרת שם וכאשר המנווען חוסם התנגשות בין עצם זה לאחר, העצם הזה לא אמרו להידחף או להזז בשל ההتانגשות, כי אם רק העצם השני.

2.1.4. שתי השורות האחרונות יחד נועדו לנורם לכך שהשחקן, מים, עלים, או כל דבר אחר שאמור שלא ליפול דרך הקרקע אכן יתנגש בה, ושהקרקע לא תידחף מטה בשל כך.

2.1.5. הגדרו את המחלקה **Block**.

שאלה למחשבה: `Block` לא דורסת שום שיטה של `GameObject`, וגם לא מוסיפה שיטות או שדות משלها. בנסיבות ניתן להגדיר שיטה סטטית בשם `create` שמייצרת `GameObject` בעל אותם מאפיינים ומוחירה אותו. מה היתרון של הגדרת מחלקה בת במקרה זה? רמז: איך בגישה החלופית זו תגדירו בהמשך "בלוק" עם התנהנות ייחודית עבור `onCollisionEnter` למשל?

2.2. המחלקה **Terrain**

אחריות המחלקה:

- לייצר את כל בלוקי הקרקע הדרושים.
- לבנוסף, המחלקה מאפשר עצמים אחרים לדעת מהו גובה הקרקע בקואורדינטת X נתונה. שימוש לב שמוספע של `Terrain` אינו `GameObject` בעצמו; הוא רק אחראי לייצור `GameObjects` אחרים (בלוקי הקרקע).

2.2.1. ראשית, נגדר בנאי:

```
public Terrain(Vector2 windowDimensions, int seed)
```

ושדה בשם `float groundHeightAtX0` שמכיל את גובה הקרקע הרצוי ב-`x=0`. למשל, אתם יכולים לאתחל אותו עם גובה החלון כפолов $\frac{1}{2}$, ואז ב-`x=0` האדמה תכסה בערך את השלישי התיכון של המseed (`seed` תוכל למשמש כדי לאותל מחולל מספרים אקרים כפויים שנסbir בבחולק של עולם אינסופי).

2.2.2. בואו ניגש דוקא לתפקיד השני של המחלקה – להגדיר לעצמה ולעצמים אחרים מה גובה הקרקע הרצוי. נגדר במחלקה `Terrain` את השיטה (זו דוגמה, אתם תצטרכו לחשב על משהו מורכב יותר אח"כ):

```
public float groundHeightAt(float x) { return groundHeightAtX0; }
```

סביר: השיטה הנ"ל מקבלת מקום על ציר `x` של החלון שלנו, היא מחפשת עבורנו מה הגובה של הרצפה. בדוגמה הנ"ל, לכל מקום בחalon המשחק קיבל גובה אחד.

לקראת ההגשה, על תוויאי הקרקע להיות מעניין יותר. הנה דרך שלא עובוד: גובה האדמה בכל קואורדינטה יהיה אקרים. נכון, הקרקע תהיה אחרת בכל רצפה, אבל היא לא תהיה רציפה. דרך אחרת להפוך את תוויאי הקרקע לפחות צפוי היא פשוט להשתמש בהרכבה של פונקציותSinCos עם גודלים, זמני מחזור, ופאות שונות. כדי לקבל בכל פעם קרקע אחרת, אפשר להפוך את הפאות לאקרים.

הצורך בפונקציה שהיא אקראיות למראה מחד, ורציפה מאידך, הוא צורך נפוץ. פונקציה כזו נקראת "פונקציית רעש חלקה" (smooth noise-function), ורבים וטובות מימשו סוגים רבים של פונקציות כאלה. **Perlin Noise** הוא אלגוריתם פופולרי לפונקציה כזו.

בקבצים המסופקים לנו תוכלנו למצוא את המחלקה [NoiseGenerator](#) שבזרתה תוכלנו לייצר Perlin Noise לפי התיעוד במחלקה.

2.2.3. השיטה השנייה של Terrain מייצרת רשימה של כל בלוקי הקרקע בתחום א-ים מסוים, על פי הנבאים שמנדרה `groundHeightAt`. שמה `createInRange`. עדין לא ננדיר אותה, נתחיל מלבדן מלבד:

```
public List<Block> createInRange(int minX, int maxX) { ... }
```

(שים לב ליבא את `List`, `java.util.List` ולא להשתמש ב-`List`)

כדי לייצר בלוקים, כדאי להגדיר את הצבע הבסיסי של אדמה:

```
private static final Color BASE_GROUND_COLOR = new Color(212, 123, 74);
```

ואז אפשר להגדיר את ה-`Renderable`:

```
new RectangleRenderable(ColorSupplier.approximateColor() (BASE_GROUND_COLOR))
```

אתם עושים לרצות גם להגדיר תנית לבлок האדמה (`setTag`), עם מחרוזת מאפיינת כמו `."ground"`.

לפנינו שנמשח את השיטה כפי שהיא אמורה להיות, ממשו אותה כרגע כך שתיצור רק בלוק אדמה ייחד במקום שהוא נראה לעין כמו ראשית הצירים (פינה שמאלית עליונה) או מרכז המסך. ב-`PepseGameManager.initializeGame` והוסיפו את הבלוקים שחזרו מהרשימה ל-`Terrain.createInRange`. קראו ל-`gameObjects` שימו לב שאתה צריך לבחור את השכבה המתאימה לאדמה כך שעצמים אחרים שנופלים עליה יתגנשו בבלוקים של האדמה (`Layer.STATIC_OBJECTS`).

2.2.5. רק עכשיו נממש את `createInRange` כך שתיצור שטח אדמה שלם על פי הנבאים שמנדרה `groundHeightAt`, ותחזר את רשימת הבלוקים שיצרה.

טיפ: **תמיד** מקטנו בלוקים בקואורדינטות שמתחלקות ב-`SIZE`!
הבלוקים אם כן לא צריכים להיות מוגדרים **בדיוק** בקואורדינטות `X min -X max` ו-`Y` שקיבלה השיטה `createInRange`, ונום לא **בדיוק** בנבאים שמחזירה `groundHeightAt`. אין כדי לוודא **שלפחות** כל תחום האיקסים המבוקש יcosה באדמה. למשל, עבור `minX=-95` ו-`maxX=40`, ואם `SIZE=30` הוא 30, הרי שהבלוק הראשון יוכל להתחיל ב-120-, והבלוק האחרון יוכל להתחיל ב-30.

מבחן קואורדינט ה- z : אם אנחנו מיצרים את עמודת האדמה של $60=x$, הרי שהבלוק **העליון** בעמודה יוכל להתחיל למשל בקואורדינט $z=0$:

```
Math.floor(groundHeightAt(60) / Block.SIZE * Block.SIZE)
```

כלומר בערך בגובה הנכוון, אבל מעגל לנודל שמתחלק בנודל בלוק.

מתוך הבלוק הזה יונחו כמובן עוד בלוקי-אדמה. כמה? נניח שכל עמודה תהיה בגובה 20 בלוקים:

```
private static final int TERRAIN_DEPTH = 20;
```

ممשו את השיטה. לאחר מכן הגדרו את הפרמטרים בקריאה לה (`initializeGame`) כך שכל רוחב המסך יcosa באדמה.
הריצו את המשחק - קיבלתם נתח אדמה יפה?

בהמשך, אחרי שנדרז דמות שיכולה לטיל בעולם הוירטואלי שלנו, נרצה לתמוך גם בעולם אינסופי שהולך ונפרס לנגד עיניה של הדמות שלנו. אתם יכולים לחשב בין היתר איך הייתם רואים לנכון להוסיף פונקציונליות זו, אך אנו ממליצים לעשות זאת רק אחרי שכבר יצרתם דמות שיכולה להסתובב בעולם.

שאלה למחשבה: חישבו איך אתם יכולים לעשות שימוש מנגנון השכבות על מנת ליעיל את החישובים במשחק, בין איזה שכבות לאיזה שכבות צריכים לחשב התנשויות ובאיזה לא, האם צריך לחשב התנשויות בתוך השכבה או לא, האם צריך לחשב התנשויות בין כל בלוקי הקרקע (לדוגמא עם `Avatar` שנוצר בהמשך) או רק עם חלק מהם וכו'. פירוט על איך לעבוד עם מנגנון השכבות תוכלו למצוא [במדריך על DanoGL](#).

3. אור וחושך

3.1. אור וחושך יומשו באמצעות טריק פשוט: גדר מלבן שחור בנודל החלון, שמצויב ממש לפני המצלמה ומסתר את כל יתר העצמים. בהירות העולם תיעשה על ידי שינוי האטימות שלו. בצהרי היום הוא יהיה באטימות 0 (שחור לחЛОטי), ובבחצות באטימות של נניח $f0.5$.
3.2. במחלקה `Night` הגדרו את השיטה הסטטית `create`:

```
public static GameObject create(Vector2 windowDimensions, float cycleLength)
```

תפקיד השיטה לייצר את המלבן לעיל על פי `windowDimensions`, ולגרום לאטימות שלו להשתנות בהתאם עם זמן מחזור של `cycleLength` (מספר השניות שלוקחת "יממה"). השיטה מוחזירה את עצם האובייקט שיצרה. החלק המעניין כאן הוא השיקיפות, אבל קודם נתחל מכל היתר.

- 3.3. צרו בשיטה עצם משחק (מופע יישיר של `GameObject`) בשם `night` עם התכונות הבאות:
- ה-`Renderable` הוא מלבן שחור (עם אטיומות ברירת מחדל של 1).
 - מרחב הקואורדינטות שלו הוא `zo` של המצלמה:

```
setCoordinateSpace(CoordinateSpace.CAMERA_COORDINATES)
```

- הוא מלא לחלוטין את כל החלון.
- הנדרו תנית מתאימה (`setTag`) שתבדיל אותו ממופעים אחרים של המחלקה לצורכי דיבוב.

- 3.4. קראו לשיטה `initializeGame` של `Night.create`. בתוכה אורר המחוור של יממה השתמשו ב-30 שניות (אם כי כמובן שכרגע עוד אין לפרמטר זהה ביטוי), והוסיפו את האובייקט שנוצר לרשימה `gameObjects`. חשבו לאיזו שכבה הוא שייר כדי ליצור את האפקט המתאים.

- 3.5. הריצו; קיבלتم מסך שחור?
- 3.6. שינוו האטיותם כל מהשייתם חשובם. הכירו את המחלקה `Transition` של `DanoGameLab`.

```
public Transition(
    GameObject gameObjectToUpdateThrough,
    Consumer<T> setValueCallback,
    T initialValue,
    T finalValue,
    Interpolator<T> interpolator,
    float transitionTime,
    TransitionType transitionType,
    Runnable onReachingFinalValue)
```

בגדרול, יצירת מופע של המחלקה מנעה שינוי כלשהו במשחק (למשל הזהה של עצם, סיבוב שלו וכו'). השינוי נעשה על ידי הרצת `callback` נתון על תחום ערכים מסוים, כשהערכים הם מטיפוס גנרי `T`.

נכיר את המחלקה והפרמטרים של הבניוי על ידי יצירת `Transition` שישנה את האטיות של `night` שלנו בצורה מעגלית. הפרמטרים, לפי הסדר:

`GameObject gameObjectToUpdateThrough` .3.6.1

על מנת של-`Transition` יהיה אפקט, הוא צריך להיות מוקשור לעצם-משחק כלשהו, בדרך כלל נזין כאן את עצם המשחק שה-`Transition` עושה בו שינוי – במקרה שלנו, נשלח את מלבן הלילה `night`.

`Consumer<T> setValueCallback` .3.6.2

תפקיד ה-`Transition` הוא להביא לשינוי בערך מסוים; זו הפונקציה שמשנה את אותו ערך. במקרה שלנו, אנחנו רוצים לשנות את האטימות של `night`. ניתן לעשות זאת עם השיטה:

```
night.renderer().setOpacity(float opacity)
```

כמו בכל `callback`, אנחנו לא מעוניינים לקרוא בפועל לשיטה `setOpacity`, אלא לידע את `Transition` באיזו שיטה הוא יכול להשתמש לצורך להביא לשינוי. לכן בתור הפורטר השני נשלח את ה-`method reference`:

```
night.renderer() :: setOpacity
```

`T initialValue` .3.6.3

מה צריך להיות הערך הראשון, בתחילת ה-`Transition`. אם נניח שהמשחק מתחילה בצהרי היום, הרי שה-`initialValue` של `night` צריך להיות 0 (שאינו לחוטין).

`T finalValue` .3.6.4

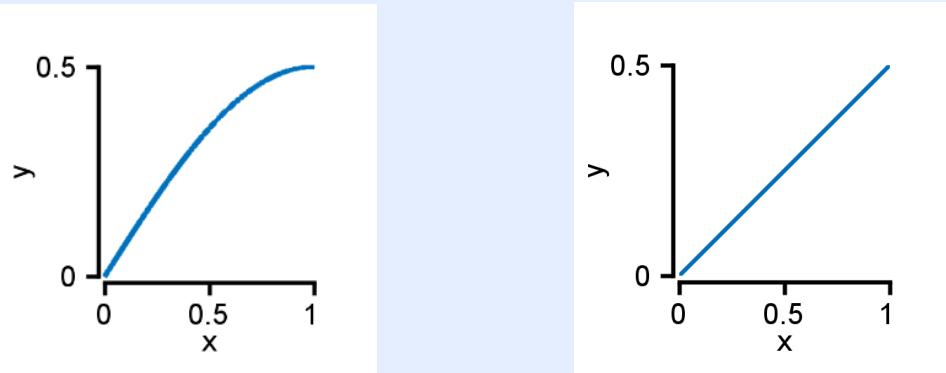
ערך הקצה השני של האטימות הוא חצי:

```
private static final float MIDNIGHT_OPACITY = 0.5f;
```

3.6.5 `Interpolator<T> interpolator`

יש הרבה (אינסוף!) דרכם לעבור מ-0 ל-0.5f תוך זמן נתון. דמיינו פונקציה שערכה ב- $x=0$ הוא 0, וב- $x=1$ שערכה 0.5. מה צורת הפונקציה הרצוי? האם היא:

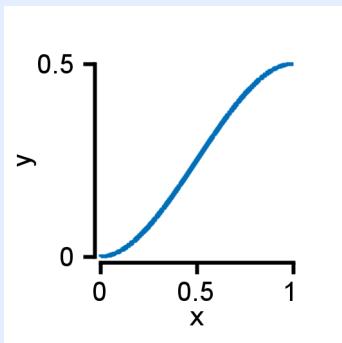
קו לינארי: אולי פולינום מדרגה גבוהה יותר? אולי סינוסידיאלית:



טיפוס הparameter, הממשק `Interpolator`, הוא ממשק פונקציוני שמאפשר לשולוט על צורת הפונקציה זו. לא ניכנס למשק זהה לעומק כי לנוחיות המשתמשים המחלקה `Transition` כבר כוללת מספר קבועים מסוג `Interpolator` שונים על הרבה מהצרכים. במקורה שלנו השתמש בקבוע:

`Transition.CUBIC_INTERPOLATOR_FLOAT`

שישנה את האטיימות מ-0 ל-0.5f על פי הפולינום מהמעלה השלישי הבא:



כפי שראתם רואים, שיפוע הפונקציה הוא קרוב ל-0 גם ב- $x=0$ וגם ב- $x=1$. אם כן משמעות השימוש באינטראפולטור הזה על פניו אינטראפולטור לינארי היה שהבהירות תהיה כמעט 1 גם מעט אחרי הצהרים, והחושך יהיה כמעט במלואו גם מעט לפני חצות, קצר כמו במציאות.

3.6.6 `float transitionTime`

זמן שלוקח לעבור מהערך `initialValue` (כלומר 0) ל-`finalValue` (כלומר 0.5f). חשבו מה הערך המתאים כאן.

TransitionType transitionType .3.6.7

הפרמטר הוא enum בעל הערכים האפשריים הבאים:

```
enum TransitionType {  
    TRANSITION_ONCE,  
    TRANSITION_LOOP,  
    TRANSITION_BACK_AND_FORTH  
}
```

כלומר הוא מאפשר לנו לבחור את אופן המעבר: האם יש לבצע את המעבר מערך הראשון לסוף רק פעם אחת (TRANSITION_ONCE), האם אחרי שמנועים לערך הסופי צריך להתחילה שוב מערך התחילה, כלומר אחרי שמנועים ל-0.5 לחזור מיד ל-0 וחזור חילתה (LOOP_TRANSITION), או האם אחרי שמנועים לערך הסופי צריך לחזור ברוחם אל ערך הראשון וחזור חילתה (TRANSITION_BACK_AND_FORTH). חשבו אילו מערכיהם לעיל מתאימים עבורנו.

Runnable onReachingFinalValue .3.6.8

יש שיש להריץ כאשר מניעים לערך הסופי. לא רלוונטי במקרה הספציפי שלנו, אז נשלח null.

.3.6.9. בסך הכל, על מנת לנורות עצם *night* לשנות את אטימותו כפי שרצינו ניצר את המעבר הבא (החליפו את סימני השאלה בערכים המתאימים לפי ההוראות לעיל):

```
new Transition<Float>(  
    night, // the game object being changed  
    night.renderer()::setOpaqueness, // the method to call  
    0f, // initial transition value  
    MIDNIGHT_OPACITY, // final transition value  
    Transition.CUBIC_INTERPOLATOR_FLOAT, // use a cubic interpolator  
    ???, // transition fully over half a day  
    Transition.TransitionType.???, // Choose appropriate ENUM value  
    null  
) // nothing further to execute upon reaching final value
```

שים לב שייצור העצם הנ"ל הוא מסוייך: אין צורך "לעשות" עם העצם הנוצר שם דבר נוסף (אין אפילו הכרח לשמור את העצם המוחזר ברפנס), כי בתוך הבנייה המעבר כבר "מתעלק" על העצם *night* לצרכי, דרך מנגן פנימי של הספרייה.
הבדיקו את השורות לעיל לסוף השיטה *Night.create* (*לפניהם* ה-*חזרה return* כਮובן), והפלאו, אם הגדרתם את הקבוע *MIDNIGHT_OPACITY* לחצי, יתכן שקיבלתם את מחזור האור והחשש הרצוי.

לשם הכרת המחלקה **Transition** הנדרנו כאן ביחד את הקריאה המדויקת, אבל עברה עליה שוב וודאו שגםם מבנים את תפקיזו של כל פרמטר, כי בפעם הבאה אתם תיצרו את המעבר בעצמכם!

משמעותי לדעת: כשאנחנו מגדירים **option**, מאחרו הקיימים אנחנו מייצרים "תוספות" **GameObject** שנשלח בformatter הראשון של הבנאי. יצירת **the-Transition** אם כן היא למשה רק עוד חלק בהנדרתאות **GameObject** (במקורה זהה, חלק מהאתחול של night). אפשר להתחיל לראות איך הרבה מהקוד שלנו בפרויקט הזה יהווה "רק" אתחול של מופעים של **GameObject** בדרכים שונות ומשונות.

במידה רבה, הנגישות הזו מתאפשרת בזכות למבזות -**method references**. ה-**API** של **Transition** היה כל כך פחות יידוטי בלבד, שמלכתחילה המחלקה לא הייתה מוצעת כפוי שהיא. איתן, מחלקה כמו **Transition** שמקבלת בformatter השני שלה, **callback strategy**, היא פשוטה להכנה מצד ספק השירות, ולשימוש מצד הלקוח.

4. שות

4.1. השמש תיווצר במחלקה **Sun**, עם השיטה **:create**:

```
public static GameObject create(Vector2 windowDimensions, float cycleLength)
```

שמחזירה את השימוש, אחרי שייצרה אותה.

4.2. גם השימוש יוכל להיות מופיע וישר של **GameObject**.

בתוך התחלת, צרו עיגול צהוב סטטי במרכז השמיים (במרכז ציר ה-X ואמצע גובה השמיים) (**by-עזרת המחלקה OvalRenderable**). עברו צבע השימוש אפשר להשתמש פשוט **Color.YELLOW**. קואורדינטות השימוש הן במרחב המצלמה כמובן, כמו השמיים **setCoordinateSpace** (setTag(**sun**), ורכזו להגדיר לה תכנית ייחודית).

4.3. קיראו לשיטה **משיטת האתחול** של המשחק, הוסיפו את המופיע של השימוש **gameObject**, וודאו שיש לכם שם שמש (סטטית). חשבו לאיזו שכבה להוסיף את השימוש.

4.4. כדי לאפשר לשימוש לנوع בשמיים במסלול מעגלי, נרצה שהשימוש תסתובב בעיגול סביב נקודה固定ה **center** במרקם קו האופק (**grounHeightAtX0**), ככלומר ציר ה-x יהיה לפני אמצע המסך, וציר ה-y יהיה לפני גובה האדמה ההתחלת. המעבר יהיה לפני 360 מעלות מסביב לנקודה זו, בלולאה. הולאה תתווזן בהתאם לזמן היממה שהנדרכנו - 30 שניות. השימוש תבצע סיבוב מלא ותחזור לנקודת ההתחלת.

בסוף השיטה **:create** (**Sun**), צריך ליצור מופיע חדש של **Transition** שיזן את השימוש לפני זווית נתונה (להסביר לגבי הפרמטרים של הבנאי חזרו לחלק של אור-וחושך).

.4.4.1. בטור `gameObjectToUpdateThrough` כਮובן שנשלח את המופיע של שמש עצמה.

.4.4.2. עברו ה-`callback` של `callback` נשלח את הלמבדה:

```
(Float angle) → sun.setCenter  
(initialSunCenter.subtract(cycleCenter)  
    .rotated(angle)  
    .add(cycleCenter))
```

המקבלת זווית, וקובעת את מיקום מרכז השימוש לפי סיבוב של השם `angle` מעלה מהמקום המקורי שלו. שמו לב שכדי שהלמבדה תפעל יש להנידר לפני המעבר את המשתנה `initialSunCenter`.

.4.4.3. בשבי טווח הערכים `initialValue-finalValue` נשלח את הערכים 0-360. שאלון הזווית שהמעבר עבר בינהן.

.4.4.4. עברו `interpolator` נשתמש באינטראפולציה לינארית פשוטה, עם:

`Transition.LINEAR_INTERPOLATOR_FLOAT.`

.4.4.5. ובעור `onReachingFinalValue` נשלח `null`.

5. הילת השימוש

.5.1. את הילת השימוש ניציר כ-`GameObject` נפרד במחלקה `SunHalo`, עם השיטה הסטטית `:create`

```
public static GameObject create(GameObject sun)
```

שדומה בהתנהגנותה לכל יתר שיטות ה-`create` שלנו.

.5.2. בטור צבע ההילה, נרצה שהוא עם שקיפות. אטימות מוצנת בערזץ `alpha`, או בקיצור ערזץ `a`, של צבע. למשל, תוכלנו להשתמש בצבע צבע צהוב עם אטימות של 20 מתוך 255:

```
new Color(255, 255, 0, 20)
```

.5.3. כמו במקרה של השימוש, נתחל מליצר הילה סטטית. גירמו לשיטה ליצר עיגול פשוט, בנודל כרצונכם, בצבע שהנדכנו, שתקייף את השימוש, והוסיפו אותו ל-`gameObjects` בשכבה המתאימה (לפניהם). ה-`CoordinateSpace` שלו הוא כמובן זה של המצלמה (בדומה לשמיים, לשמש, ולמלבן החושך). הנדרו להילה תנית שתיעיד אותה.

.5.4. בשבייל לקרוא לשיטה `PepseGameManager.initializeGame` `SunHalo.create` נצטרך לשלוח את המופיע של השימוש. הריצו; קיבלתם הילה סטטית תקועה בשםים?

מכיוון שהשיטה מקבלת את עצם המשחק `hus`, ההייה לא צריכה ליצור `Transition`, (`sunHalo.setCenter`), חדש שיזעזב אותה מפורשות: די שההילה תעתק את המרכז שלו (`sunHalo.getCenter`).
 בכל פרימיט, להיות כמו זה של השימוש (`GameObject.update`). איך? אירן אפשרות אחת היא ליצור תת-מחלקה של `GameObject`, בה נדרס את השיטה `update`.
 העניין יצרן מספר שורות ספורות, כ-10, ויהפוך את הקוד למורכב יותר רק בקצת, בקר שמתווסףת מחלוקת קטנה מאוד.
 ישנה גם אפשרות אחרת, מבוססת אסטרטגיה. אם נניח שלעצמם שיצרתם קוראים `sunHalo`,
 תוכלן לקרוא לשיטה `sunHalo.addComponent`. השיטה (הנמצאת כבר במחלוקת) מצפה לפרמטר מסוג `Component`, (סוג של `Observer`) שהוא המשחק
 הפונקציונלי הבא:

```
@FunctionalInterface
public interface Component {
    void update(float deltaTime);
}
```

השיטה `addComponent` מצפה ל-`callback` שמקבלת `float` ומ回馈ת דבר.
 השיטה מבטיחה לקרוא ל-`callback` שהתקבל בסוף כל עדכון (`update`) של העצם.
 כמובן, שלו ל-`sunHalo.addComponent` ביטוי لماذا שמקבל `deltaTime` ומעדק את מרכז ההייה להיות מרכז השימוש. נוף הלמדא לא צריך לעשות שימוש בפרמטר שלו `deltaTime` במקרה הזה. שמו לב, אפשר לנורם לעצם `sunHalo` לעקב אחריו `hus` גם בשורה אחת, ובמבלאי טיפוסים נוספים.

6. הוספת דמות (Avatar).

בחלק זהה, נוסיף משחק שלנו דמות שיכולה לזרז בעולם.
 הדמות צוברת אנרגניה כאשר היא במנוחה, ומשתמשת באנרגניה כדי לבצע פעולות שונות.
 הדמות צריכה להיות מסוגלת לבצע את הפעולות הבאות:

- לנוע לצדדים באמצעות החיצים.
- ל קופץ באמצעות מושך הרווח.

6.2. אתחול ראשוני

בשלב זה תוכלן לחתה השראה מהדמות הפשוטה של פלטפורמר שספקנו לכם בקובץ [Platformer.java](#)
שימוש לב: אין להניש את `java.Platformer`.
 בתור התחלה, הגדרו ב-`Avatar`, `Avatar` את המחלוקת `Avatar`, עם הבניי הבא:

```
public Avatar(Vector2 topLeftCorner,
              UserInputListener inputListener,
```

ImageReader imageReader)

כasher `topLeftCorner` הוא המיקום במסך עליו אמורה הדמות לעמוד (פינה שמאלית עליונה). כשאתם מאתחלים את הדמות, עליה להיות בגובה הkrkע. בטור `Renderable` כרגע השתמשו בתמונה "idle_0.png" המספקת לכם בתיקיה `assets`. הורידו את התקייה ומתקמו אותה `שיירות תחת תקית src` של הפרויקט שלכם (כלומר מחוץ לחבילה `pepse`), וקראו את התמונה באמצעות ה-`imageReader`:

```
imageReader.readImage("assets\\idle_0.png", true);
```

cutת תוכלו להעביר את ה-`Renderable` שנוצר בקריאה `super` (כמו `Platformer.java`) או למשל באמצעות `setRenderable(renderer, () ->`. שימו לב: כאשר אתם משתמשים בתמונות שב-`assets` בקוד שלכם, עשו זאת בעזרת נתיב יחסית (רלטיבי), כמו בדוגמה לעיל.

הוסיפו לדמות לוגיקה בסיסית של תזוזה ימינה ושמאליה עם החיצים, וקפיצה באמצעות מושך הרוחות, בדוגמה `Platformer.java`.

הוסףו מושע של `Avatar` ב-`PepseGameManager`, בדקו שהדמות נוצרה במקום הנכון (חישבו איפה כדאי לאתחל את הדמות כך שהיא תיווצר בבדיקה על הkrkע), והיא זהה ימינה שמאלה וקופצת בהתאם להוראות.

6.3. מצבי הדמות

לדמות יהיו 3 מצבים תזוזה שיופיעו על התגובה שלה לפעולות שונות:

- **idle (מנוחה)**: הדמות עומדת במקום על הkrkע.
- **run (ריצה)**: הדמות נעה ימינה או שמאלה על הkrkע.
- **jump (קפיצה/אוויר)**: הדמות נמצאת באוויר, במלר עלייה או נפילה (הדמות קופצת למעלה או נופלת חזרה לkrkע).

מצב הדמות ישפייע על האנימציה שתוצג עבור הדמות (חלק 6.7), צבירת ובזבוז האנרגיה שלה (חלק 6.4) וכן על התגובה שלה לקלטים מהמשתמש (חלק 6.5).

6.4. אנרגניה

cutת נוסיף למשחק את המרכיב של צבירת ובזבוז אנרגניה. לדמות יש משבב אנרגניה שחיוני לביצוע פעולות.

טווח האנרגניה נמדד באחוים, וכל אורך המשחק כמוות האנרגניה של הדמות תനוע בין 0-100. המשחק מתחילה כך שיש לדמות אנרגניה מקסימלית של **100%**.

עלויות פעולות:

- **ריצה**: דרושת **2** יחידות אנרגניה, **בכל עדכון**.
- **קפיצה**: דרושת **20** יחידות אנרגניה, לביצוע **жд-פעמי**.
- **קפיצה כפולה (Double Jump)**: דרושת **50** יחידות אנרגניה לביצוע **жд-פעמי**.

6.5. לוגיקת עדכון ההתנהנות (מתודת update)

6.5.1. כללי תנועה וקלט

1. **תנועה ימינה/שמאלת** - לחיצה על מקשי החיצים ימינה/שמאלת:

- אם הדמות נמצאת **על הקרקע** ויש **פחות 2 ארגניהם** לחיצה על מקשי החיצים ימינה/שמאלת גורמת לדמות לנוע בכיוון החץ. אם הדמות לא סוף הארגניהם הנדרש, לא יקרה כלום.

- אם הדמות נמצאת **באוויר** לחיצה על מקשי החיצים ימינה/שמאלת גורמת לדמות לנוע בכיוון החץ **בלי לשבוב ארגניהם**. אם הדמות זהה ימינה/שמאלת באוויר זה יהיה בנוסף למשואה למעלה/למטה ולא למקום.

2. **קפיצה** - לחיצה על מקש רווח גורמת לקפיצה במקרים הבאים:

- אם הדמות **על הקרקע** ויש **פחות 20 ארגניהם** לחיצה על מקש הרווח גורמת לקפיצה רגילה.

- אם הדמות **באוויר** **במהלך נפילה** (כלומר מצב jump עם מהירות חיובית בziej ע) **ושפחות 50 ארגניהם** לחיצה על מקש הרווח גורמת **לקפיצה נוספת מהמיוקם הנוכחי**.

- אם הדמות לא סוף הארגניהם הנדרש, לא יקרה כלום.

6.5.2. לוגיקת צבירה וابידן ארגניהם (בכל update)

מצב	שינוי ארגניהם
Idle (מנוחה על הקרקע)	נצח 1 יחידות ארגניהם
Run (ריצה על הקרקע)	מתבצעות 2 יחידות ארגניהם
jump (באוויר)	אין שינוי (לא ננצח ולא מתבצע)
קפיצה חדשה (רגילה)	מתבצעות 20 יחידות ארגניהם באופן חד-פעמי מיידי
קפיצה חדשה מהאוויר (כפולה)	מתבצעות 50 יחידות ארגניהם באופן חד-פעמי מיידי

כמה הבהרות:

- אם הדמות עומדת ולא זהה, היא מקבלת נקודת ארגניהם אחת **רק כאשר היא על הקרקע** במקורה של לחיצה על שני החיצים, ימינה ושמאלת, יחד באותו update - הדמות לא אמרה לזרז ולא אמרה לרדת ארגניהם.
- לא ניתן לקפוץ קפיצה רגילה מהאוויר. היינו, מלבד קפיצה כפולה, כשהדמויות באוויר ונלחץ על קפיצה (רווח), היא לא אמרה לעלות כלפי מעלה, גם אם יש לה מספיק ארגניהם לקפיצה רגילה, ועל כן היא לא אמרה לאבד ארגניהם. כלומר, הדמות תוכל לקפוץ קפיצה

רנילה שוב ורק כאשר היא נחתה על הקרקע. הדמות תקוףז באוויר רק אם יש לה מספיק אנרגיה לקפיצה כפולה.

- אם ערכו ההורדה וההעלאה של האנרגיה יוצרים אצלם בעיות, כמו ריצוד של הדמות בזמן ריצה על הקרקע, אתם מוזמנים לשנות את הערכים ולציין זאת ב-README.
- **שימוש לבן!** אם לדמות אין מספיק אנרגיה אז היא לא תוכל לבצע פעולות כלל וצטרך לעמוד במקום/להישאר במצב הנוכחי. למשל אם לדמות יש 5 נקי אנרגיה והוא תנסה לקפוץ (שדורש 10 נקי אנרגיה) אז היא לא תצליח לקפוץ (וכמובן לא תרד האנרגיה). וכן, אם הדמות במצב ריצה ונגמרת לה האנרגיה היא תעבור למצב מנוחה נס אם אחד החיצים נלחץ.

6.6. ממשק תצוגת האנרגיה

6.6.1. כתעת נוספת למסך שלכם חיוי שמראה בכל זמן נתון את כמות האנרגיה הנוכחית של הדמות.

כדי להזכיר איך עושים זאת, תוכלו לחזור לתרגיל 2 ולהסתכל בקוד שהכתבתם בעבר הצנת כמות החיים שנשarra במשחק. העיצוב של החלק הזה נתון לבחירתכם.

6.6.2. איך מי שטפל בהצנת כמות האנרגיה יהיה מעודכן באנרגיה של הדמות שלנו? אם בעבר כדי להעביר מידע בין מחלקות שהינו צריכים ליצור קשר של הכלה בין מחלקות (או לשבור אנקופולציה לא علينا...), עכשו אתם כבר יודעים להשתמש גם בתכנות פונקציונלי, ואפשר פשוט לשלוח למחלקה callback שיספק לה נתונים ממחלקה אחרת בלי שהיא תכיר אותה בכלל.

אבל... האם ממשק תצוגת האנרגיה צריך להיות מעודכן בכל update, או שמא בכלל אפשר לעדכן אותו רק כאשר נעשה שינוי באנרגיה? חשבו על מנגנון בקוד שיעול לאפשר עדכונים של תצוגת האנרגיה רק כאשר נעשה שינוי באנרגיה.

6.7. אнимציה

6.7.1. כתעת נרצה להציג אнимציה של הדמות במקום תמונה סטטית, וכן נרצה שהanimציה תתחלף בהתאם לתזוזה של הדמות.

אפשר ליצור אнимציה של הדמות ע"י החלפה בלולאה של סדרת תמונות של התזוזה בה הדמות נמצאת במקום להציג את הדמות בצורה סטטית ע"י תמונה בודדת. לכל תזוזה, יש סדרת תמונות מתאימה בקבצים שישיפנו לכם בתיקיה assets:

- עבור מצב `idle` בו הדמות לא זהה יש 4 תמונות עם השמות:

.idle_0.png, idle_1.png, idle_2.png, idle_3.png

עבור מצב `jump` בו הדמות עולה או יורדת (בלי תזוזה לצדדים) נשתמש בתמונות:

.jump_0.png, jump_1.png, jump_2.png, jump_3.png

עבור מצב `run` בו הדמות זהה לצדים (בון על הקרקע ובון באוויר) נשתמש בתמונות:

.run_0.png, run_1.png, run_2.png, run_3.png, run_4.png, run_5.png

כדי להציג אнимציה בעזרת סדרת תמונות השתמשו במחלקה `AnimationRenderable`.
המקבלת במבנה שלה את סדרת התמונות וכן את אורך הזמן בין החלפת התמונות.
(אפשר גם לחת `AnimationRenderable` תיקיה במבנה ולא רק רץ תמונות).
שימו לב:

- תוכלו להחליף את ה-`Renderable` של העצם בזמן ריצה כאשר הוא עובר בין מצבים באמצעות המתודה `setRenderable(())`.
 - כדי לשקף את ה-`Renderable` אופקית השתמשו במתודה: `setIsFlippedHorizontally(())`.
 - ניתן גם להתאים לכל סט תמונות מידדים (dimensions) מסוג `Vector2` כדי לשומר על גודל אחיד, ולעדכן את הממדים באמצעות `setDimensions(())`.
 - בקבצים שיש במודול סייפקנו לכמם תמונות עבור דמות שיש לה 3 מצבים תזוזה (אפשר להנדר יותר, לדוגמה אнимציה של נפילה מול קפיצה, וכמוון גם מצבים חדשים). תוכלו כמובן גם להשתמש בכל דמות אחרת שאתם רצים, ואפשר למצוא ריעונות [כאן](#).
- שימו לב** במידה שאתם נתקלים במצב בו הדמות נכנסת לתוך האדמה תוכלו להוסיף במתודה `onCollisionEnter` של Avatar את קטע הקוד הבא (ודאו שהנתן משן את התג "block" לתג המתאים בתכנית שלכם):

```
if(other.getTag().equals("block") && isFalling()){
    System.out.println();
    this.transform().setVelocity(0);
}
```

כאשר `()isFalling()` היא מתודה שאתם יכולים להנדר המתארת נפילה של הדמות. אפשר גם לנסות להשתמש במקומה בבדיקה: `0>(<)y.()collision.getNormal`, רק וודאו שהנתן שאתם בזדקים עובד. בנוספ אם נתקעים באובייקטים מהצדדים (ציר x) תוכלו לטפל בצורה דומה.

7. צמחיה - עיצוב לבחירתכם

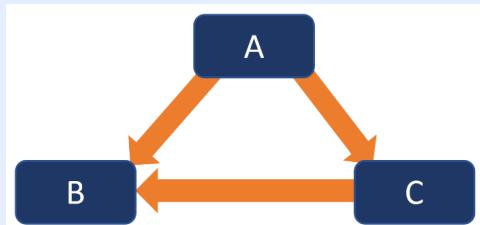
- 7.1. בחלק זהה הנדר בשלבים את תוכנות העצים כאשר בהתחלה ננדיר איך נראה עץ סטטי, שלא זו, ועוד נוסיף לעצ תוכנה של תזוזה של העלים. מומלץ לעבור על כל הפרק לפני שמתחילה בעיצוב כדי לקבל את התמונה השלמה של העצים.
- 7.2. את רוב העיצוב של החבילה ההו נשאיר לכם, למעט הדרישה למחלקה בשם Flora ובה שיטה `createInRange(int minX, int maxX) { ... }` עם החתימה:

```
public ??? createInRange(int minX, int maxX) { ... }
```

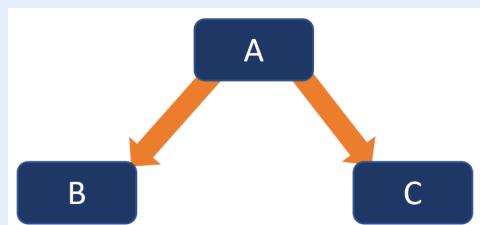
עיצוב עצים (במקומות אקרים/קבועים מראש לבחירתכם) בתחום X-ים מסוימים ותחזיר מבנה נתונים הכלול את האובייקטים שנוצרו. סוג מבנה הנתונים וטיפוס האובייקטים תלוי בעיצוב שלכם.

הכוונה עיצובית: החופש העיצובי בתכנון הספרייה הזו זה לא אומר שצורך להתפרק עם איזה עצוב גראנדיזי, כי הפונקציונליות לא ענקית. כן ניתן עתה אחת כללית בונגוע לקשר בין העצים לבין Terrain, היהות והעצים יctrarco לדעת איפה בעולם להופיע.

עדיף על נרף התלויות הבא:



нерף התלויות הבא:



למולנו, את הנרף השני אפשר לעיתים להפוך לראשון. אם התלות של C ב-B נובעת מהצורך לקרוא לשיטה של המחלקה B, דרך טובה וקלה לחסוך את התלות היא ש-A תשלח ל-C ישירות את השיטה הדרושה, באמצעות callback. כך C תכיר את ה-A callback שהוא מקבל, והוא הינה בעל טיפוס כללי של functional interface, ולא את המחלקה הקונקרטית B. זהו מקרה פרטני של העקרון "תכונות למשחק, לא למימוש" (program to interface, not implementation).

אם ההכוונה העיצובית זו לא מובנת לכם בשלב זהה, נסו לקרוא אותה שוב לאחר שעברתם על כל הפרק של יצור העצים.

7.3. עצים סטטיים

7.3.1. הגדרו ב-peps.world.trees מחלקות לשם שתילת עצים **סטטיים** (כשלב ראשון): הם כוללים גזע בנובה אקראי וצמרת ריבועית מסביב לקצה הגזע, עם עלים שעוד לא חיים. בנוסף לעיצוב, נשאיר בידיכם גם את כל פרטי המימוש, אבל כן ניתן כמה הכוונות.

המע יכול להיות פשוט מלבן עם צבע שהוא וריאנט של:

`new Color(100, 50, 20)`

ואילו העלים יכולים להיות מורכבים מבולוקים בעלי צבע שהוא וריאנט של:

`new Color(50, 200, 30)`

7.3.2. בחירת מקום העצים פשוטה: לכל עמודה אפשר להטיל מטבע מוטה (נניח עם הסתברות של 0.1), ואם הוא יצאאמת, "שותלים" בעמודה הזו עץ (קרי: יוצרים בלוקים עבורי).

"תלת מטבע מוטה" נעשית על ידי בחירת מספר אקראי בין 0 ל-1 ובדיקה האם הערך גדול או קטן מסף רצוי. כדי ליצור עלווה לא צפופה מיידי מומלץ גם להטיל מטבע מוטה גם כדי להחליט כמה עלים ליצור בראש כל עץ. כמו כן, את העץ צריך לשימוש בנובה הנק� של הקרקע, ר' השיטה `Terrain.groundHeightAt`.

שםו לב! הדמות במשחק שלנו צריכה להתנש בזע העץ, אך לא בעלים, וכן העלים לא אמרום להתנש אחד בשני. (ראו סרטון הדגמה). חשבו היטב באיזו שכבה לשימוש כל אובייקט שיצרתם כדי לקבל את התנהנות המשחק הרצואה.

7.4. תנודות העלים ברוח

7.4.1. כדי לנענע את העלים ברוח, די להוסיף שני מעברים (Transitions) מסוג BACK_AND_FORTH: אחד על זווית העלים, ואחד על רוחב העלים (גם מעברים שיונדרו על תוכנות אחרות של העלים יעשו את העבודה בצורה דומה). הזווית נשלטת על ידי:

```
leaf.renderer().setRenderableAngle(angle)
```

ורוחב העלים נשלט על ידי:

```
leaf.setDimensions(dimensionsAsVector2)
```

הנדירו Transitions כאלו לבולוקים של העלים צפוי בתוצאה.

אולי שמתם לב שאם כל העלים נעים יחד בדיק באותו קצב וזמן זה נראה רע – אם תסתכלו החוצה בחלון, תראו שלא קר הם נעים בעולם האמיתי.

אם כן נניח שברצוננו להתחיל ב-Transition מסויים, אבל אנחנו לא רוצים שהוא יתחל בבדיקה ברגע זה, אלא בעוד זמן כלשהו. בנוסף נניח שלכל העלים יש למשה בדיק את אותו Transition, אבל כל אחד מתחילה אותו בדילוי מעט אחר – למשל אחד כעבור 0.1 שניות מצירתו, ואחר לאחר 0.5 שניות.

על מנת להרייך קוד נתון **בדילוי** (לגרום לו לוחץ בעוד זמן נתון), ישנה ב-DemoGameLab מחלוקת שימושית נוספת: ScheduledTask. הבניי שלו הוא בעל החתימה הבאה:

```
ScheduledTask(  
    GameObject gameObjectToUpdateThrough,  
    float waitTime,  
    boolean repeat,  
    Runnable onElapsed)
```

המבנה מקבל:

- עצם משחק שאליו המשימה רלוונטיות (בדומה ל-Transition),
- את מסגר השניות שיש להמתן לפני הרצת המשימה,
- דגל שואמר האם המשימה אמורה לחזור על עצמה כל `waitTime` שניות או שמא יש להרייך את המשימה רק פעם אחת ולשכו ממנה,
- ומופיע מסוג `Runnable` שמייצג את המשימה המذובורת.

מכיון ש-`Runnable` הוא ממשק פונקציוני (של פונקציה שלא מקבלת כלום ולא מחזירה כלום), המחלוקת עובדת היטב עם למדות ומצביים לשיטות.

בדומה ל-Transition, אחרי שיצרתם מופיע של `ScheduledTask` אין צורך לעשות אותו עוד שום דבר נוסף – עצם יצירת ה-`ScheduledTask` מספקת להרצת המשימה

המתווזמת, בזכות עצם המשחק שנשלח בפורמט הראשון ושהמשימה המתווזמת "מתעלקת"
על ה-update שלו.

במקרה שלנו, נרצה לדאוג לכך שה-transition של העלה יוצר בדיל' קטן וקצת שונה
לכל עלה, ואפשר לעשות זאת ע"י שיחת callback שיצרת את ה-transition
ל-ScheduledTask.

נשאר לנו לכמם את הפרטים המדויקים של איך לגרום לעליים להתנווע בצורה "מציאותית". אבל הנה טיפ
אחרון: החכמה כאן היא לחלק את הקוד לשיטות קצחות ומחלקות מוגדרות היטב. אמן הסנון התכונתי
שלנו בתרגיל זהה השתנה מעט, אבל שמירה על שיטות קומפקטיות, ועל קבצים עם אחריות מוצמצמת
היא עקרון תכונתי אוניברסלי.

7.5. הוספת פירות

עתה נרצה להוסיף משחק שלנו פירות שיופיעו על העצים והדמאות שלנו תוכל "לאכול" אותם כדי לצבור
עוד אנרגיה. העיצוב של חלק זה כמו כל העיצוב של העצים נתן לבחירתכם.
כמו הנקודות להתנהגות הפירות:

- 7.5.1. הפירות יהיו פשוט אובייקטים בצורה עיגול.
- 7.5.2. צבע הפירות נתון לבחירתכם.
- 7.5.3. גודל הפרי יהיה לכל היותר כנודל העליים.
- 7.5.4. הפירות יופיעו על צמרות העצים בין או על העליים. שימוש לב שהעלים והפירות לא
אמורים "להתנגש".
- 7.5.5. בכל פעם שדמאות תתגש בפרי, פרי יעלם, והדמאות תקבל 10 נקודות אנרגיה.
- 7.5.6. פרי ש"נאכל" יופיע מחדש במסחר ב��ום מהוחר שלם מרגע הוא נעלם (מכור, מהוחר
של המשחק הוא 30 שניות).
- 7.5.7. אם פרי מופיע בתוך גע אחר מכיוון שהעלווה של העץ שלו חופפת לגע, זה תקין,
פשוט יותר והפרי לא יהיה נגיש לאכילה.

7.6. סיכום

בשלב הזה של התרגיל ננסה לעשות סדר בעניין השכבות. מכור, הדמאות שלנו צריכה להתנש באדמה,
בגעמי העצים, אך לא בעליים שלהם (כਮון שלא בשימוש או בהילת השימוש). אם זה לא המצב אצלכם
בתכנית, בדקו באיזו שכבה כל אובייקט נמצא, ונסו לסדר אותם מחדש כדי שתקבלו את ההתנהגות
הרצויה.

8. עולם אינסוי

8.1. כתעת, הנחו את המצלמה לעקב אחרי הדמאות שיצרתם. אם שם המשטנה שבו נשמרה הדמות
הוא `avatar`, תצטרכו להכניס שורה לקוד שלכם שורה שורא שורא שורא שורא שורא שורא שורא שורא:

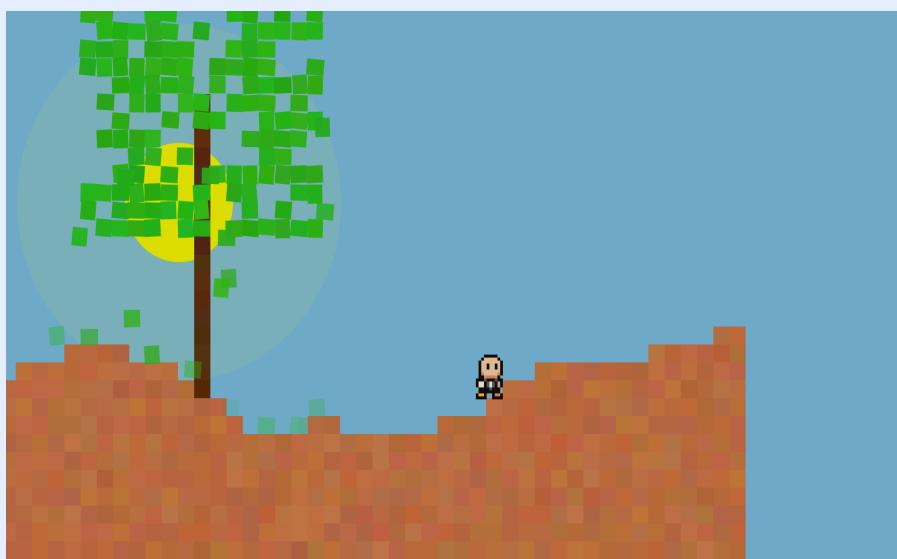
```
setCamera(new Camera(avatar, Vector2.ZERO,  
                    windowController.getWindowDimensions(),
```

```
        windowController.getWindowDimensions());
```

כאשר הפורטט השני במבנה של Camera הוא המרחק של האובייקט הנעקב ממרכז המסך. לכן, אם נרצה שבמצב ההתחלתי הקואורדינטה (0,0) על המסך תתאים לקואורדינטה (0,0) בעולם (כמו שהוא המצב לפני שהוספנו את הדמות), נצטרך להחליף את Vector2.ZERO במשהו כמו:

```
windowController.getWindowDimensions().mult(0.5f) - initialAvatarLocation
```

8.2. בהנחה שבקריה ל- Flora.createInRange ו-Terrain.createInRange לא הגדרתם $x_{\min} = -\infty$ ו- $x_{\max} = \infty$, כנראה שדי מהר תנלו שהגעתם לסוף העולם:



כדי להימנע ממצב זה, علينا להגדיר עולם אינסופי.

כਮון שלא נוכל ליצור מראש בלוקי אדמה ועצים אשר יכסו את כל ערכי-x האפשריים, ולכן אין לנו ברירה אלא להמשיך ליצור אותם בזמן ריצה. יש לכם יד חופשית לחלווטין בבחירה האופן שבו תבצעו זאת, אבל כדאי לזכור שצדדי עדכו וчисוב התגשויות בין אובייקטים שנמצאים הרחוק מחוץ לאזור הנראתה של המסך סתם יגלו לנו משאבי חישוב ויהפכו את הריצה לאייתת בלי שתהייה לך איזושהי תרומה. לכן, אנחנו ממליצים לכם למצוא דרך לצמצם כמה שאפשר את החישובים שהקשרים לאובייקטים שנמצאים מחוץ לאזור הנראתה.

בנוסף, העולם שלנו צריך להיות עקי - כלומר, אם החלטתם למחוק את האובייקטים שנמצאים מחוץ לאזור הנראתה וליצור אותם מחדש רק כאשר מתקרבים אליהם, שימו לב שהם צריכים להיות זהים (לפחות למראית עין) לאובייקטים שהיו שם בפעם הקודמת שעברנו באותו נקודה.

במילים אחרות, אם גובה האדמה ב- $x=0$ היה 100, גם אם נלך קילומטרים ואז נחזור שוב לאאותה נקודה, גובה האדמה בה עדין יהיה 100, ובאופן דומה, אם ב- $x=0$ היה עץ בגובה מסוים עם מבנה עליים מסוימים, גם בפעם הבאה שנבקר באותה נקודה יהיה שם עץ באותו

נובה ועם אותו מבנה עליים (וכמובן שם לא היה בנקודה מסוימת עז, נम בפעם הבאה שנבנה בה לא יהיה בה עז). הצורך הזה בעקבות מוביל אותנו לסעיף הבא.

8.3. אקרואיות ניתנת לשחזור

המפתח לאקרואיות משוחזרת הוא היכרות עם האופן בו מספרים פסודו-אקרים מיוצרים ברוב המימושים. כאשר אתם מייצרים עצם חדש מסוג `Random`, הוא מאותחל עם זרע (`seed`). את הזרע ניתן להעביר לבנייה, או לחילופין אם השתמשתם לבניית הריך, יונדר עבר העצם ורעד שmboss על פרמטרים משתנים כמו הזמן הנוכחי. הזרע עצמו יכול להשתנות מעצם, אבל שני עצמי `Random` שאותחלו באמצעות זרע, יצרו בדיקות את אותה סדרה של מספרים "אקרים". כמובן, עבור לולה שקרוואת `nextInt` מ-`Int` נקבע עבור שני העצמים את אותם מספרים בדיקות.

אם כן, אם העולם כולו מיוצר על ידי עצם ייחיד של `Random` (שMOVED בין המחלקות), בפעם הבאה שנರיץ את הסימולציה עם אותו `seed`, יוצרו אותו עולם!
אבל זה לא לנMRI מדויק. זה נכון עבור עולם מיוצר כלו מיד בקריאה ל-`initializeGame`, אבל אם שטחים נוספים בעולם מיוצרים בזמן ריצה שנגilibים את המסר, סדר הקריאה `Terrain.createInRange` תלוי בקלט המשתמשים.

נניח שאנו מיצרים עצים בעלי צורה ייחודית שתלויה במספרים אקרים. דרך קלה להבטיח שעץ שנוצר בקואורדינטה `x=60` תמיד יהיה אותו עץ, עבור `seed` נתון של הסימולציה, היא ליצר את העץ באמצעות עצם `Random` שהזרע שלו הוא פונקציה של הזרע הכללי של הסימולציה, והקואורדינטה `60`. למשל, אם הזרע הכללי הוא `mySeed`, אפשר ליצור את העץ הבא:

```
new Random(Objects.hash(60, mySeed))
```

עוז שיעזר בעצם הנ"ל לאקרואיות שלו מובטח להיות בעל אותה צורה עבור אותו מקום ואותו זרע ראשוני, גם בהרצאות הבאות.
עדכנו את הקוד שלכם כך שהעולם שנוצר יהיה עקי - כמובן, אם החלטתם למחוק את האובייקטים שנמצאים מחוץ לאזור הנראה וליצור אותם מחדש רק כאשר מתקרבים אליהם, שימושם לב שהם צריכים להיות זהים (פחות למראות עין) לאובייקטים שהיו שם בפעם הקודמת שעברנו באותה נקודת.

9. הוראות הנשנה

9.1. הנישואת התרנגול כקובץ zip/tar/jar בשם ex4 (והסימנת בהתאם, למשל zip ex4.zip).

בתוך קובץ זה, ימצאו החבילות והקבצים כמו שמתוואר בסעיף [0.2](#). שימושם לב שמותר לכם **להוסיף** חבילות ומחלקות כרצונכם.

9.2. קובץ README.

.9.2.1 בשורה הראשונה בקובץ יופיע שם המשתמש *CSE* שלכם.
אם אתם מנישים בהוגן, יש להפריד בין שמות המשתמש עם פסיק (שניהם באותה שורה).

.9.2.2 בשורה השנייה מספר תעודת זהות.
אם אתם מנישים בהוגן, יש להפריד בין מספרי הזהות עם פסיק (שניהם באותה שורה).

.9.2.3 השורה השלישית ריקה.

.9.2.4 **החל מהשורה הרביעית ענו בקובץ על השאלות הבאות:**

• הסבירו על הדרכם בהחרתם למש את הדמות (*Avatar*), התייחסו בתשובתכם
לנקודות הבאות:

- פרטו על המחלקות/חבילות השונות שיצרתם בחבילה ועל הקשרים ביניהן.

- איך בחרתם לעצב את שינוי המוצבים בדמות?

- איך בחרתם לעצב את שינוי האנרגיה בדמות? איך בחרתם לעדכן אותה, ואייר
בחחרתם לעצב ולעדכן את התצוגה שלה?

• הסבירו על הדרכם שבהחרתם למש את החבילה *trees*, פרטו על
המחלקות/חבילות השונות שיצרתם בחבילה ועל הקשרים ביניהן.

• אם הוספتم חבילות/מחלקות נוספת על מה שמצוור בשאלות 2-1, פרטו
והסבירו עליהן.

• אם שינויתם את ה-API (בין אם חבילות/מחלקות, ובין אם חתימות של מתודות או
כל שינוי אחר) נא לציין זאת כאן ולהוסיף הסבר.

.9.3 **תכנון קוד ובונוס מגן**

.9.3.1anno ממליצים תמיד לתקן את הקוד לפני שניגשים לכתיבת הקוד עצמו.
כדי לעודד אתכם לעשות זאת, בתרגnil זה תהיה אפשרות להניש תכנון מוקדם של הקוד
שבוע לפני תאריך ההגשה ולקבל על כך ציון מנגן לתרגnil.

.9.3.2קראו את כל ההוראות לפני שאתם מתחילה לנשת למלאכה. את החלקים המודרכים
(בهم אטם נדרשים בהוראות לכתוב קוד יחד עם הקריאה) ניתן לפני תכנון
הפרויקט אך ממש לא חייב. במיוחד כדי לתקן ברכיניות את הקוד של חלק 6 (דמות)
והלאה (כלומר לשיטים לקרוא ותיכנן ורק אז לנשת לכתיבת הקוד). את תכנון הקוד כדאי
לעשוו בדרכים שלימדנו לאורך הפרויקט.

.9.3.3בונוס מגן - בתרגnil זה אנחנו נאפשר בונוס מגן, כולל גרפים ותרשיimi תכנון של
הפרויקט. אלו יכולים להיות באיזה סגנון שאתם רוצים, ולא מחייבים את הפרויקט הסופי.
המטרה היא לתקן מראש, אבל אם נדרשים לשינוי לפחות מחרוג מהתכנון המוקורי.

.9.3.4גרפים יכולים לכלול גרף אחד גדול מכל הפרויקט, יכולים להיות גם גרפים קטנים
לנושאים שונים. הם יכולים להיות מסוג UML, חלוקה לחברות ופירוט המחלקות, גרף
תלוויות, גרפים המתארים מבנים עיצוב שונות וכו' - יש לכם חופש יצירתית לתוכנו
סוג שnoch لكم.

.9.3.5 הבונוס הוא בנדיר 10% מגן, וניתן לקבל 100 נם בלבד.

9.3.6 ציון התרגיל (כולל בונוס ציון המגן) יהיה:

$$final_ex4_grade = \max\{ex4_grade, (0.9 * ex4_grade + 0.1 * magen)\}$$

תאריך הנשחת הבונוס הוא **28.12.2025**, ואין עליו הארוכות.

בהצלחה!

10. פרולוג

דיון על שילוב פרדייניות תוכנות

אלו שמתם לב שלא השתמשנו הפעם הרבה במשחקים. למעשה, הרבה מהקוד יוצר מופעים שימושתיים בשודות וบทישות של `GameObject`, והם נבדלו "רך" באסטרטגיות שלהם ולא בימוש שונה של שיטות. אז מה עושים שם `Sun`, `Sky`, `Night`, `Cloud` וכו' – האם תפקידן של מחלקות אלו להגדר עצמים מסווג חדש?

ראינו שאשר למחלקה כמו `GameObject` יש תמייה נרחבת באסטרטגיות, יצירתיות של מופע בודד, כמו אלה, יכולה להימשך בפני עצמה عشرות או מאות שורות קוד עם לוגיקה מורכבת: הגדרנו לכל עצם `Renderable` אחר, ושלחנו לו Components שונים כמו `Transition`, `ScheduledTask` ואחרים, `Object` אחד מthem דרש נתח קוד לא מבוטל.

את כל הקוד של ייצור עצם בודד שמננו במחלקה ייעודית שמחולקת לשיטות, אבל קוד הייצור הזה הוא לא שיטה של עצם אחר. ייצור המופעים של `GameObject` נעשה במסגרת שיטות סטטיות, או שיטות מופע של עצם "סמליל" – שהוא המופע היחיד של המחלקה שלו, ושאין לו מצב (שודות) מעניין. זה היה המקורה עם המחלקה `Terrain` ונום במקורה של `PepseGameManager`.

או בזמן שהקוד שלנו יוצר עצמים, הוא עצמו לא היה מאורגן בעצמים, אלא בפונקציות הקשורות זו לזו. תוכנות מבוססות קריאה לפונקציות נקרא **תכנות פרוצדורלי** – לא נרחב עליו כי רובכם כבר תכנתתם באופן פרוצדורלי (בין אם קראתם לוה כן או לא). הסימולטור שלנו, אם כן, תוכנתה במודל "היברידי": הליבה הייתה פרוצדורלית, והיא מצידה נשענה על עצמים ועל אסטרטגיות פולימורפיות שהן **Object Oriented Design** בהתגלמותו.

גם מי מכמ שיעיחספו בהמשך לתוכנות **פונקציונלי**, יזכירו בلمודות שכתבו כאן וימצאו בקוד גניעות קלות גם של הפרדינמה הזאת.

כמו פטיש הבית המצרי, תוכנות מונחה עצמים הוא כלי חשוב בארגון הכלים שלנו. ישנן משימות או חלקיים בתוכנה שאין הכרח לעשות עם פטיש, כמו הברנת ברנים, ניתוח ניירות, וכו'. בדומה לפטיש, אם משתמשים בו למקורה הנכון אבל בצורה לא נכונה, נוצרות בעיות. בדומה לפטיש, אם עושים בו שימוש זמני ומוצלח בשבייל להבריג בורג זה עדין יהיה יותר טרחה מאשר אחרת. אף על פי כן, תוכנות מונחה עצמים הוא כלי חזק ונוח, שבנינגד לפטיש מתאים לספקטרום רחב מאוד של משימות, ובנוסף משחק יפה עם כמות מפתיעה של כלים שונים ומשלים – למשל תוכנות פרוצדורלי כפי שראינו כאן. אז תשמרו את הראש פתוח בהמשך לשילוב של כלים מעולמות שונות, כמו מונחה עצמים, פרוצדורלי, פונקציונלי, מונחה אירופיים (event-driven), מונחה מידע (data oriented) ואחרים.