עבודה עצמית 8 – השירותים finalize ו clone של Object

חלק א' - finalize:

המחלקה A מממשת את השירות finalize. ה finalize נקרא כאשר ה Garbage Collector (GC) מחליט לפנות את האובייקט וה finalize מאפשר לאובייקט להגדיר ולבצע פעולות "נקיון" אחרונות לפני שהוא מפונה מהזכרון.

```
public class A {
	private int i;

	public A(int i){
		this.i = i;
	}
	@Override
	protected void finalize() throws Throwable{
		super.finalize()
		System.out.println(String.format("{%d} : Bye Bye", this.i));
	}

	public static void main(String[] args) throws InterruptedException{
		A[] aArr = new A[] {new A(1), new A(2)};
		aArr = null;
		System.gc();	//call gc
		Thread.sleep(2000); //sleep for 2 seconds
		System.out.println("Finished running main!");
	}
}
```

בדוגמא זו, אנחנו מייצרים מערך של 2 אובייקטים מטיפוס A ואז מוחקים את כל ההפניות לאובייקטים האלה ע"י ההשמה של aArr ל null בשורה השניה של ה main. לאחר מכן, אנחנו מבצעים קריאה יזומה ל GC בשביל לוודא שהאובייקטים מטיפוס A יפונו.

הריצו את הקוד וודאו ש finalize אכן נקראת (ניתן יהיה לראות את ההדפסות שלה ב console).

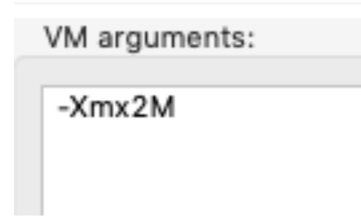
הערה: אנחנו קוראים ל finalize של Object (באמצעות super.finalize) ליתר ביטחון, כיוון שהיא מכילה קוד שאמור לרוץ עבור כל אובייקט שעובר "פינוי" (ונדירות הפעמים שבהן היא נדרסת).

כעת, התבוננו על התוכנית הבאה, המבוססת על תוכנית שניתנה כשאלה בבחינה בקורס באחד הסמסטרים הקודמים.

```
public class A {
	private A next = null;

	public static A singleA = new A();
	public static long instancesCounter;

	@Override
	public void finalize() throws Throwable {
		super.finalize()
		this.next = singleA;
		singleA = this;
	}

	public static void main(String[] args) throws InterruptedException {
		while (true) {
			A a = new A();
			instancesCounter++;
			if (instancesCounter % 10000 == 0){
				System.out.println(String.format("instances: {%d}",
instancesCounter));
				int singleALength = 0;
				for (A start = singleA; start != null; start =
start.next){
					singleALength++;
				}
				System.out.println(String.format("singleALength =
%d", singleALength));
			}
		}
	}
}
```

בתוכנית זו קורה משהו מפתיע – ה finalize נקראת כשהאובייקט אמור להימחק, וזה קורה כשאין שום מצביע לאובייקט הזה, אבל ה finalize בעצם מייצרת מצביע לאובייקט עליו הוא מופעל ע״י קישורו לרשימה המקושרת המוצבעת ע״י singleA שהוא שדה סטטי (שימו לב של A יש שדה מטיפוס A שנקרא next, ולכן נוצר מבנה נתונים מקושר).. האם האובייקט ימחק? התשובה היא שלא! למרות שה finalize נקראת, ההצבעה שהיא מייצרת תמנע מה GC למחוק את האובייקט.

אז מה יקרה כאן? זה ניראה שאנחנו מייצרים אינסוף אובייקטים שלא מתרוקנים. בסופו של דבר נגיע למצב שאנחנו צורכים יותר מדי זכרון על ה heap ונקבל שגיאה.
אם נריץ את הקוד כמו שהוא על ה eclipse, יקח לו הרבה זמן להגיע לשגיאה. כיצד נקבל אותה מהר יותר? נגביל את הזכרון המוקצה לתוכנית.
הפרמטר Xmx מקבע את גודל ה heap המקסימלי המוקצה לתוכנית בעת ההרצה שלה (קיים אתחול דיפולטי, כמובן). נקבע את גודל ה heap המקסימלי להיות מאוד קטן (2 מגהבייט) ע״י האתחול הבא:
ב eclipse, דרך אותו המסך שבו קובעים את הפרמטרים שהתוכנית מקבלת (חלון תחתון)



ב command line מריצים באופן הבא:

java -Xmx2M A

תוך זמן קצר, נקבל שגיאת OutOfMemory.
הכל טוב ויפה, אבל מאיפה אנחנו יודעים שהשגיאה התקבלה באשמת ה finalize? יכול להיות שיצרנו אינסוף אובייקטים, ה GC לא נקרא אף פעם, ובאמת חרגנו מהזכרון. כלומר, אולי התוכנית הזו לא מוכיחה שניתן "להחיות" אובייקט אחרי הקריאה ל finalize.
כאן אנחנו יכולים להעזר בהדפסה (מדפיסים את אורכו של singleALength שמתארך רק כשה finalize נקרא).
ההסבר התיאורטי הוא שה GC מתוזמן גם לפי גודלה של המחסנית למול הגודל המקסימלי – כשמגיעים לגבול העליון של הזכרון המוקצה עבור ה heap, ה GC נקרא בשביל לנסות ולפנות אותו. במקרה שלנו, הוא לא מפנה כלום ולכן בסופו של דבר התוכנית נופלת על OutOfMemory.

<u>חלק ב - clone:</u>
כעת נרצה לממש clone במספר אופנים.
המחלקה בה נעסוק היא Box שעוטפת בתוכה אובייקט מסוג A.

```
public class A {
	private int i;
	public A(int i){
		this.i = i;
	}

	public String toString(){
		return String.format("A({%d}", this.i);
	}
}
```

```
public class Box {
	A a;
	String str;

	public Box(A a, String str){
		this.a = a;
		this.str = str;
	}

	public String toString(){
		return String.format("B(%s, %s)", this.a.toString(), 
this.str);
	}

	public Object clone(){
		return new Box(this.a, this.str);
	}

	public static void main(String[] args) {
		A a1 = new A(1);
		Box b1 = new Box(a1, "abc");
		Box b2 = (Box)b1.clone();
		System.out.println(b2);
	}
}
```

<u>נסיון ראשון:</u>

נממש את ה clone בעצמנו, מבלי להשתמש בשירות שנורש מ Object. המימוש יעשה באמצעות קריאה לבנאי של Box.

המימוש אכן עובד, אבל אנחנו צריכים לשים לב לכך שזהו שכפול רדוד (shallow). מה זאת אומרת? אמנם נוצר אובייקט Box חדש, אבל שני האובייקטים, b1 ו b2 מכילים הצבעה לאותו האובייקט מטיפוס A (השדה a של שניהם).

אם נדפיס את השורה b1.a == b2.a נקבל True (כזכור, השימוש ב == מבצע השוואת כתובות)

בשביל לקבל שכפול עמוק אמיתי, עלינו לשכפל גם את a.

נוסיף ל A שירות clone:

```
	public Object clone(){
		return new A(this.i);
	}
```

ונעדכן את המימוש של clone בתוך Box כך שישכפל גם את a.

```
	public Object clone(){
		return new Box((A)this.a.clone(), this.str);
	}
```

שימו לב לשימוש ב casting. מכיוון שה clone מחזירה Object, אנחנו צריכים לבצע casting לטיפוס המתאים.

לאחר שני השינויים האלה, וודאו ש b1.a == b2.a מדפיס false.
שימו לב שאנחנו מפעילים clone על this.a ולא קוראים לבנאי שלו, כיוון שאנחנו לא יודעים אם שימוש בבנאי יבצע שכפול עמוק (ואם בכלל יש משמעות לשכפול עמוק בתוך האובייקט שלנו, במקרה של A אין משמעות כי הוא מכיל רק int יחיד). מסיבה זו אנחנו סומכים על שירות ה clone שלו שיבצע שכפול עמוק.

נסיון שני:
אז מימשנו clone וראינו את ההבדל בין שכפול עמוק לרדוד. ומה עם ה clone שירשנו מ Object? אולי הוא בעצם יודע כבר לבצע את השכפול הנדרש?

בקוד הבא נוריד את המימוש שלנו של clone ונשתמש ב clone שנורשה מ Object. שימו לב ששימוש ב clone עלול לייצר שגיאה (מופיעה בחתימה של main, ניגע בזה מאוחר יותר).
את ה clone שאנחנו מפעילים על b1 ירשנו מ Object. הניראות שלה הוא protected ולכן ניתן להשתמש בה מתוך הקוד של B (protected נגיש למחלקה היורשת וגם למחלקות שנמצאות באותה החבילה).

```
public class Box {
	A a;
	String str;

	public Box(A a, String str){
		this.a = a;
		this.str = str;
	}

	public String toString(){
		return String.format("B(%s, %s)", this.a.toString(), this.str);
	}


	public static void main(String[] args) throws
CloneNotSupportedException {
		A a1 = new A(1);
		Box b1 = new Box(a1, "abc");
		Box b2 = (Box)b1.clone();
		System.out.println(b2);
	}
}
```

הריצו את הקוד ובדקו את הפלט.
עבדו עלינו! ירשנו clone של Object אבל שימוש בו מייצר שגיאה. האם אפשר לפתור את זה?

כן!
הטריק הוא לשנות את ההגדרה של Box כך שתממש את המנשק Cloneable

```
public class Box implements Cloneable
```

וזהו. הריצו את הקוד ועכשיו הוא יעבוד ויבצע את השכפול.
אז מהוא בעצם המנשק הזה?
כשבוחנים את התיעוד שלו, רואים שזה מנשק שבעצם לא מגדיר שום שירות. אז למה הוא טוב?
כשמחלקה מממשת מנשק כלשהו, זה אומר שהיא מקיימת איתו יחס is-a וממשת את כל השירותים שהוא מגדיר. כשאין שירותים, זה עדין משאיר יחד is-a. כלומר, המנשק Cloneable הוא מנשק הצהרתי – המחלקה Box מצהירה על כך שהיא בת שכפול.
האם השכפול הוא עמוק? בדקו את תוצאת ההשוואה בין b1.a ל b2.a. אנחנו רואים שהשכפול הוא לא עמוק.

בעצם, זה די הגיוני. ה clone של Object לא באמת מכירה את המבנה הפנימי של המחלקה שהיא משכפלת, ולכן היא פשוט מעתיקה את תוכן הזכרון של b1 לתוך מקום חדש שאליו יצביע b2. ומה כתוב בתוך b1? שתי כתובות – אחת של str והשניה של a. כלומר, אין שכפול לא של a ולא של str.

ויש עוד בעיה – ה clone שירשנו מ Object לא נגיש מחוץ לחבילה שבה נמצאת B. הוא נגיד רק למחלקה היורשת (שהיא Box) ורק למחלקות בתוך החבילה שלנו.

נפתור את שתי הבעיות בשני שלבים. נתחיל מבעיית הנראות:
נחזיר ל Box את השירות clone, אבל הפעם הוא ישתמש ב clone שנורש מ Object

```
public Object clone() throws CloneNotSupportedException{
      return super.clone();
}
```

באופן דומה, עדכנו את ה clone של A.

שימו לב, ה clone שהוספנו דורסת את clone של Object ומרחיבה את הניראות שלה מ protected ל public. זה חוקי ב Java ונדבר על זה בהמשך הקורס. כעת, ניתן יהיה להפעיל clone על אובייקט מטיפוס Box מכל מקום. שימוש לב שהוספת השירות clone לא פותרת אותנו מהצורך להכריז על מימוש המנשק Cloneable. זה נדרש מאיתנו בגלל השימוש ב clone של Object.

פתרנו את בעיית הניראות, ועדכן נשאר לוודא ש a משוכפל גם הוא. מכיוון ש clone שלנו לא קוראת לשום בנאי, הדרך לשכפל את a היא קודם כל לשכפל את Box ואז לוודא שהשדות שמצריכים שכפול ישוכפלו.

```
public Object clone() throws CloneNotSupportedException{
      Box clone = (Box)super.clone();
      clone.a = (A)this.a.clone();
      return clone;
}
```

לאחר ביצוע השורה הראשונה אנחנו מקבלים שכפול רדוד של האובייקט שלנו. השורה השניה דואגת לכך ש a ישוכפל, ו clone.a יצביע לשכפול. אם היו לנו עוד שדות שמצריכים שכפול עמוק, היינו צריכים לשכפל גם אותם בנפרד.

אז איזו משתי השיטות טובה יותר? האם שימוש בבנאי או שימוש ב clone של Object?
השיטה העדיפה היא השיטה השניה כיוון ה clone של Object הוא יעיל כיוון שהוא מעתיק שטחי זכרון שלמים, לעומת קריאה לבנאי שגם כותבת לזכרון אבל מבצעת עוד פעולות.