הורשה (Inheritance)

הקדמה

אופן הגדרתה של מחלקה שיורשת מאחרת - Derivation Syntax

משמעות ההורשה - Inheritance Effects

הרשאות הגישה – Access Modifiers

הגדרתן מחדש של מתודות שהגיעו בהורשה - Overriding Methods

רב צורתיות – Polymorphism

המילה השמורה super

מתודה מסוג final

מחלקה מסוג final

אופן הפעולה של הבנאים – Constructors Chaining

המתודה finalize

מחלקות אבסטרקטיות - Abstract Classes

ממשק – Interface

שינוי טיפוס – Casting

הפיכת ערך מטיפוס בסיסי לאובייקט ולהפך - Converting Primitive Types to Objects & Vice Versa

מתודות שתם זמנן – Deprecation

מחלקות מוכנות ליצירת מבני נתונים

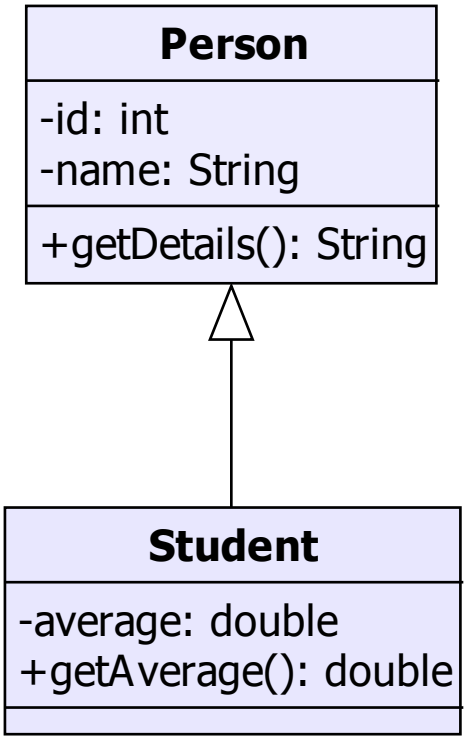
© 2009 Haim Michael. All Rights Reserved.

הקדמה

כאשר בוחנים את העולם סביבנו, ומבחינים באובייקטים שניתן לסווג לקבוצות (מחלקות) ניתן גם להבחין בקווי דמיון שקיימים בין הקבוצות השונות. לדוגמא, בין קבוצת כל הסטודנטים בעולם וקבוצת כל האנשים בעולם ניתן למצוא את המאפיינים המשותפים הבאים: בשתי הקבוצות יש לכל אחד מהאובייקטים ששייך אליהן שם, מספר תעודת זהות ועוד... בחינה מדוקדקת יותר של התכונות שמשותפות לשתי קבוצות שונות תראה כי ניתן, למעשה, לראות בקבוצת כל האנשים בעולם קבוצה כללית יותר אשר כוללת בתוכה את קבוצת הסטודנטים. התכונות שיש לכל אחד מהאובייקטים ששייכים לקבוצת האנשים בעולם, קיימות גם בכל אחד מהאובייקטים ששייכים לקבוצת הסטודנטים. קבוצת הסטודנטים היא קבוצה חלקית לקבוצת האנשים. באופן דומה, נוכל גם לומר שכל הפעולות שאנו מזהים כפעולות אפשריות לביצוע בכל אחד מהאנשים זמינות לביצוע גם בכל אחד מהסטודנטים. בהמשך לניתוח האמור, נוכל לומר שכל אחד מהסטודנטים הוא גם איש. במילים אחרות נוכל לומר שכל אחד מהסטודנטים הוא איש שהתווספו לו מספר תכונות נוספות וכמו כן התאפשרו גם ביצוען של מספר פעולות נוספות. אם נדמיין שכל אחד מהאובייקטים שמייצגים סטודנטים נוצר מתבנית (Template) בדומה ל-cookie cutter שמשמש ליצירת עוגיות, נוכל לומר שההגדרה של אותה תבנית מבוססת על ההגדרה של התבנית (Template) שממנה נוצרו כל אחד מהאובייקטים ששייכים לקבוצת האנשים, ובמלים אחרות נוכל לומר שההגדרה של התבנית שמשמשת ליצירת אובייקטים שמייצגים סטודנטים היא הרחבה של ההגדרה של התבנית שמשמשת ליצירת אובייקטים שמייצגים אנשים. כפועל יוצא, נוכל לפשט ולומר שהמחלקה Student יורשת מהמחלקה Person.

© 2009 Haim Michael. All Rights Reserved.

את ההיררכיה שזיהינו נוכל גם לתאר בתרשים הבא:



המחלקה הכללית יותר תיקרא בשם מחלקת האב, ובאנגלית נהוג לקרוא לה בשם base class או super class.

המחלקה הספציפית יותר תיקרא בשם מחלקת הבן, ובאנגלית נהוג לקרוא לה בשם derived class או sub class.

© 2009 Haim Michael. All Rights Reserved.

את ההגדרה של המאפיינים המשותפים למחלקת האב ולמחלקת הבן נוכל למצוא במחלקת האב. במחלקת הבן, אנו נוסיף להגדרה שמגיעה בהורשה מאפיינים נוספים.

מעתה נוכל לומר שכל משתנה שמוגדר במחלקת האב יהיה קיים בתוך כל אובייקט שיווצר ממחלקת האב וגם בתוך כל אובייקט שיווצר ממחלקת הבן. בנוסף, נוכל גם לומר שכל מתודה שמוגדרת במחלקת האב ניתן יהיה להפעילה גם על כל אובייקט שנוצר ממחלקת האב וגם על כל אובייקט שנוצר ממחלקת הבן. במלים פשוטות נוכל לומר שההגדרה של מחלקת הבן יורשת מההגדרה של המחלקה אחרת (מחלקת האב) את כל התכונות והפעולות שמוגדרות בה, כלומר, מקבלת את כל המשתנים ואת כל המתודות שהוגדרו בה.

משמעות הדבר היא שבהגדרה של מחלקת הבן אין צורך לחזור ולהגדיר את המשתנים והמתודות שכבר מוגדרים במחלקת האב. בהגדרה של מחלקת הבן נוסיף את הגדרתם של תכונות ופעולות נוספות (מתודות ומשתנים נוספים) אשר יתווספו למשתנים והמתודות שהוגדרו במחלקת האב. אם נרצה, נוכל גם להגדיר מחדש במחלקת הבן מתודות שהגיעו בהורשה ממחלקת האב, ובכך לשנות את מה שהגיע בירושה בהורשה (יוסבר בהמשך, overriding).

השימוש באפשרות להגדיר מחלקה שיורשת מאחרת (מרחיבה את הגדרתה) חוסך בכתיבה של קוד מקור הן בעצם הגדרתה של מחלקה המתבסס על הגדרתה של מחלקת האב והן בשינויים שנרצה לעשות באופן פעולת המתודות השונות שהגדרתן הגיעה בהורשה באמצעות הגדרתן מחדש (overriding).

© 2009 Haim Michael. All Rights Reserved.

אופן הגדרתה של מחלקה שיורשת מאחרת - **Derivative Syntax**

הגדרתה של מחלקה שיורשת ממחלקה אחרת (באנגלית: extending) נעשה בכך שבהמשך שורת הכותרת של המחלקה שמגדירים יש לרשום את המילה extends ומייד אחריה את שמה של המחלקה המורישה. לאחר מכן, בתוך הסוגרים המסולסלות של המחלקה המוגדרת ניתן להוסיף משתנים ומתודות אשר יתווספו לאלה שהוגדרו במחלקה המורישה.

לדוגמא :

```
public class Person

{

      int id;

            String name;

             public String getDetails()

             {

                        . . .

            }

}
```

© 2009 Haim Michael. All Rights Reserved.

```
public class Student extends Person
{
      double average;
      double getAverage()
      {
            return average;
      }
            . . .
}
```

בדוגמא זו, המחלקה Student יורשת מהמחלקה Person. כל אחד משני המשתנים שהוגדרו במחלקה Person קיימים בכל אובייקט שיווצר מהמחלקה Person וגם בכל אובייקט שיווצר מהמחלקה Student. בנוסף, בכל אובייקט שיווצר מהמחלקה Student יהיה גם המשתנה שהוגדר במחלקה Student. כמו כן, המתודה getDetails() שהוגדרה במחלקה Person ניתן יהיה להפעיל אותה גם על כל אובייקט שיווצר מהמחלקה Person וגם על כל אובייקט שיווצר מהמחלקה Student. בנוסף, על כל אובייקט שיווצר מהמחלקה Student ניתן גם יהיה להפעיל את המתודה getAverate() שהוגדרה במחלקה Student.

ב-JAVA (להבדיל מ- C++) כל מחלקה יכולה לרשת רק ממחלקה אחת. ב-JAVA אין הורשה מרובה (מצב שבו מחלקה נתונה יורשת ממספר מחלקות). תכנון התכנית תחת המגבלה שלא ניתן לרשת מיותר ממחלקה אחת יותר מבני ומסודר. כמו כן, כאשר אין את האפשרות לרשת מיותר ממחלקה אחת אין מצבים בעייתיים שבהם מקבלים למחלקה בהורשה את אותה מתודה בדיוק משתי מחלקות שונות עם מימוש שונה בכל אחת מהן (אלו מצבים בעיתיים שעדיין קיימים ב-C++). כאשר תכנון המחלקות נכון ומסודר, אין צורך בהורשה מרובה. עם זאת, אין כל מגבלה על מחלקה מלהוריש ליותר ממחלקה אחת. מחלקה יכולה להוריש למספר בלתי מוגבל של מחלקות.

© 2009 Haim Michael. All Rights Reserved.

כל מחלקה שמגדירים ולא מציינים שהיא יורשת ממחלקה אחרת תירש (כברירת מחדל) מהמחלקה Object. אם המחלקה שהגדרנו יורשת ממחלקה אחרת אז התכונות והמתודות שהוגדרו במחלקה Object יגיעו בהורשה מאותה מחלקה אחרת. אם גם המחלקה האחרת ירשה ממחלקה אז בדיקה של היררכית המחלקות תגלה מהר מאד כי במעלה ההיררכיה מגיעים לבסוף למחלקה שלא יורשת מאף אחת. מחלקה זו היא שתירש את התכונות ואת המתודות שהוגדרו במחלקה Object ותעבירן הלאה במורד ההיררכיה עד למחלקה הנתונה. מסיבה זו, ניתן לומר, כי התכונות שהוגדרו במחלקה Object קיימות בכל אובייקט... מכל סוג. כמו כן, ניתן גם לומר כי המתודות שהוגדרו במחלקה Object ניתן להפעילן על כל אובייקט.

התכנית הבאה מתארת הורשה פשוטה בין שתי מחלקות.

```
class Person
{
        protected String name;
        protected int age;
        public Person()
        {
        }
        public Person(String nameOfPerson, int ageOfPerson)
        {
                name = nameOfPerson;
                age = ageOfPerson;
```

© 2009 Haim Michael. All Rights Reserved.

```
        }

        public void printDetails()

        {

                System.out.println("name : " + name + "\nage : " + age);

        }

}




class Student extends Person

{

           protected double average;

      Student(String nameVal, int ageVal, double averageVal)

      {

           name = nameVal;

           age = ageVal;

           average = averageVal;

      }




      double getAverage()
```

© 2009 Haim Michael. All Rights Reserved.

```
    {
        return average;
    }
}


class SimpleInheritanceDemo
{
    public static void main(String args[])
    {
        Student stud = new Student("Moshe",123123,98);
        stud.printDetails();
    }
}
```

בתכנית לעיל יווצר אובייקט מהמחלקה Student. ה-reference שלו מאוחסן בתוך המשתנה stud. באמצעות המשתנה stud תהיה קריאה להפעלת המתודה printDetails אשר הוגדרה במחלקה Person. אנו יכולים לקרוא להפעלתה על האובייקט שנוצר מהמחלקה Student כיוון שהמחלקה Student יורשת מהמחלקה Person.

© 2009 Haim Michael. All Rights Reserved.

משמעות ההורשה - **Effects of Inheritance**

כאשר מחלקה יורשת ממחלקה אחרת אז כל משתנה שהוגדר במחלקה האחרת יהיה קיים בכל אחד מהאובייקטים שנוצרים מהמחלקה היורשת. באופן דומה, כל מתודה שהוגדרה במחלקה האחרת ניתן יהיה להפעיל אותה על כל אובייקט שנוצר מהמחלקה היורשת.

להבדיל מ-++C, ב-Java אין סוגים שונים של הורשה. ב-Java יש הורשה מסוג אחד, ובמסגרתה כל המשתנים וכל המתודות עוברים בהורשה. עם זאת, בהחלט ייתכן מצב שעקב הרשאת הגישה של המשתנה/מתודה שעברו בהורשה הפניה למשתנה ו/או הקריאה להפעלת המתודה לא יתאפשרו באופן ישיר. כך לדוגמא, משתנה שהוגדר עם הרשאת הגישה private במחלקה המורישה, יהיה קיים בתוך כל אובייקט אשר יווצר מהמחלקה היורשת. עם זאת, הגישה הישירה אליו לא תתאפשר מתוך גבולות המחלקה היורשת. הגישה היחידה שתתאפשר אליו היא באופן עקיף באמצעות קריאה להפעלתה של מתודה שהוגדרה במחלקה היורשת ושבגבולותיה אין כל בעיה למקם פניה למשתנה האמור.

כל אובייקט שנוצר מהמחלקה היורשת מכיל לפיכך את כל המשתנים שהוגדרו במחלקה המורישה ואת כל המשתנים שהוגדרו במחלקה היורשת. על כל אחד מהאובייקטים שיווצרו מהמחלקה היורשת ניתן יהיה להפעיל את כל אחת מהמתודות שהוגדרו במחלקה המורישה וגם את כל אחת מהמתודות שהוגדרו במחלקה היורשת.

את ה-reference לאובייקט מטיפוס מחלקה נתונה ניתן לאחסן במשתנה מטיפוס המחלקה הנתונה (זאת כבר וודאי ידוע), אך ניתן גם לאחסן אותו במשתנה מטיפוס מחלקה שהורישה למחלקה הנדונה (בין אם הורישה באופן ישיר ובין אם הורישה באופן עקיף דרך מחלקה אחרת).

© 2009 Haim Michael. All Rights Reserved.

לדוגמא (המשך הדוגמא של המחלקות Person ו-Student):

```
Person stud = new Student("Moshe",123123,98);
```

בתוך המשתנה stud שהוגדר כמשתנה מטיפוס Person מוכנס ה-reference לאובייקט שנוצר מהמחלקה Student.

כעת ניתן לומר כי בעוד שהטיפוס של האובייקט נותר Student ללא שינוי ה-reference אליו אשר מוחזק בתוך המשתנה stud הוא reference מטיפוס Person.

הסיבה לכך שזה אפשרי (ההיפך לא אפשרי: לא ניתן לאחסן reference לאובייקט מטיפוס Person בתוך משתנה מטיפוס Student) היא שכל המשתנים שנצפה שיהיו באובייקט מטיפוס Person קיימים גם באובייקט מטיפוס Student (הוא יורש מ-Person) ובאופן דומה, כל המתודות שהוגדרו במחלקה Person ניתן להפעילן גם על כל אובייקט מטיפוס Student.

את הדוגמא האחרונה נוכל לשכתב ולכתוב מחדש את המחלקה Simple InheritanceDemo.

```
class SimpleInheritanceDemo
{
        public static void main(String args[])
        {
           Person stud = new Student("Moshe",123123,98);
           stud.printDetails();
        }
}
```

© 2009 Haim Michael. All Rights Reserved.

הרשאות הגישה

כפי שכבר הוסבר בפרק מספר 3, ניתן להוסיף בתחילת שורת ההצהרה על משתנה/מתודה את אחת המלים הבאות:

`private`

`public`

`protected`

בהוספת כל אחת מהמלים הנ"ל יש השפעה על אפשרות הגישה/השימוש במשתנה/מתודה. למרות שנושא זה כבר הוסבר (למעט protected), בחרתי לחזור ולהסבירו שוב, והפעם גם להתייחס להרשאת הגישה protected.

הרשאת הגישה private

משתנה/מתודה שמגדירים עם הרשאת הגישה `private` ניתן יהיה להשתמש בהם באופן ישיר אך ורק כשזה נעשה בגבולות של מתודות ששייכות למחלקה שבה הוצהר על המשתנה/המתודה.

הרשאת הגישה public

משתנה/מתודה שמגדירים עם הרשאת הגישה `public` ניתן יהיה להשתמש בהם באופן ישיר בכל מקום.

הרשאת הגישה package friendly

משתנה/מתודה שלא מציינים בשורת ההגדרה שלהם את הרשאת הגישה שלהם יהנו מהרשאת הגישה package friendly. זוהי ברירת המחדל. משמעות הרשאת הגישה package friendly היא שניתן להשתמש בהם באופן ישיר אך ורק כאשר נמצאים בגבולות ה-package שאליו שייכת המחלקה שבתוכה מוגדר/ת המשתנה/המתודה האמור/ה.

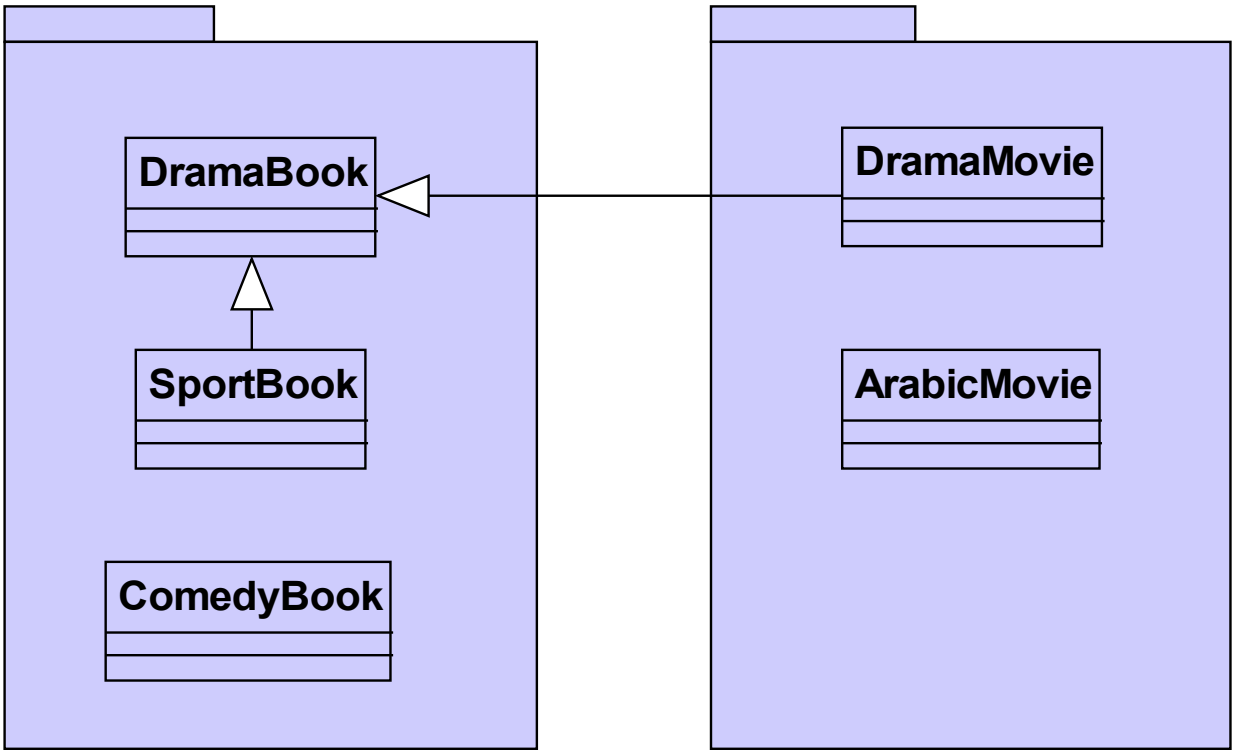
© 2009 Haim Michael. All Rights Reserved.

הרשאת הגישה protected

משתנה/מתודה שמוגדרים עם הרשאת הגישה protected יהנו מהרשאת גישה שזהה להרשאת הגישה package friendly בתוספת האפשרות לגשת אליהם באופן ישיר גם כאשר זה נעשה מחוץ לגבולות ה-package שבו המשתנה/מתודה מוגדרים ובלבד שהמחלקה שבה שורת הקוד נכתבת יורשת מהמחלקה שבה המשתנה/מתודה האמורים מוגדרים.

כדי להבין את שהוסבר בנוגע לכל אחת מהרשאות הגישה האפשריות, ניעזר בדוגמא הבאה. הנח כי במחלקה DramaBook מוגדרים מספר משתנים ומתודות עם הרשאות גישה שונות. הנח גם שהמחלקה DramaBook שייכת ל-package ששמו books וכי יחד עמה שייכות ל-package הזה גם המחלקות הבאות: ComedyBook, ו- SportBook. הנח, בנוסף, את קיומו של package אחר בשם movies אשר כולל בתוכו את המחלקות: DramaMovie ו-ArabicMovie. הנח גם כי המחלקות DramaMovie ו-SportBook יורשות מהמחלקה DramaBook. בתרשים הבא, ניתן לראות ייצוג גראפי לתיאור קשרים אלה.

© 2009 Haim Michael. All Rights Reserved.



כעת, נסכם את שמות המחלקות שמתוכן ניתן לגשת לכל אחד מהמשתנים/מתודות במחלקה DramaBook בהתאם להרשאת הגישה.

הרשאת הגישה	שמות המחלקות שממתודות שמוגדרות בהן ניתן לגשת אל הרכיב
public	ComedyBook, SportBook, DramaBook, DramaMovie, ArabicMovie
protected	ComedyBook, SportBook, DramaBook, DramaMovie
package-friendly	ComedyBook, SportBook, DramaBook
private	DramaBook

© 2009 Haim Michael. All Rights Reserved.

הגדרתן של מתודות שהגיעו בהורשה מחדש - Overriding Methods

כאשר מגדירים במחלקה היורשת מתודה שזהה בשמה, בטיפוס הערך המוחזר, במספר הפרמטרים שלה ובטיפוס של כל אחד מהם למתודה שהועברה בהורשה ממחלקה אחרת, אז המתודה החדשה תתפוס את מקומה של המתודה שהגיעה בהורשה. במצב כזה נוכל לומר שהמתודה החדשה מבצעת overriding למתודה בגרסתה המקורית.

המתודה בגרסתה החדשה באה במקומה של המתודה שהועברה בהורשה, וכך כאשר על אובייקט, שיווצר מהמחלקה שמגדירים, תופעל מתודה זו, המתודה שתפעל היא המתודה החדשה שהוגדרה.

כאשר יש קריאה להפעלת מתודה שיש לה מספר גרסאות שונות (מספר גרסאות שונות שנוצרו באמצעות overriding), הגרסה שתפעל היא הגרסה המתאימה ביותר לטיפוס של האובייקט, ללא כל קשר לטיפוס של ה-reference. כלומר, ללא כל קשר לטיפוס של המשתנה אשר מחזיק ב-reference לאותו אובייקט שעליו מנסים להפעיל את המתודה. בזמן ריצה ה-JVM בודק את הטיפוס של האובייקט שעליו יש ניסיון להפעיל מתודה ומחפש אותה תחילה במחלקה שממנה האובייקט נוצר. אם המתודה לא מופיעה בהגדרה של המחלקה הזו ה-JVM מחפש את המתודה בהגדרה של המחלקה שמורישה ואם גם שם הוא לא מוצא אותה ה-JVM ממשיך בחיפוש במעלה ההיררכייה של המחלקות עד אשר הוא מוצא אותה.

יש לשים לב לכך שאם מספר/סוג הפרמטרים במתודה החדשה שמגדירים מעט שונה מזה שבמתודה שהועברה בהורשה אז זו כבר תהיה מתודה אחרת, ולא יהיה פה מצב של overriding.

המתודה החדשה שמגדירים כדי שתתפוס את מקומה של מתודה אחרת שהגיעה בהורשה יכולה לזרוק את אותם

© 2009 Haim Michael. All Rights Reserved.

exceptions (מהסוג שמחייב התייחסות באמצעות בלוק try & catch או באמצעות הוספת throws לכותרת של המתודה) שהמתודה המקורית יכולה לזרוק או רק חלק מהם או exceptions מסוג שיורש מסוג ה-exceptions שהמתודה המקורית מסוגלת לזרוק. לא ניתן להגדיר את המתודה בגרסתה החדשה כמתודה שעלולה לזרוק exceptions (מהסוג שמחייב התייחסות בדמות בלוק try & catch או באמצעות הוספת throws לכותרת של המתודה) שהמתודה המקורית לא הוגדרה כמי שמסוגלת לזרוק אותם.

ניתן לקבוע כי מתודה שתופסת את מקומה של מתודה אחרת תזרוק exceptions אשר המתודה המקורית לא זרקה (כאשר מדובר ב-exceptions מסוג שאיננו מחייב התייחסות בדמות בלוק try & catch או באמצעות הוספת throws לכותרת של המתודה). אם נושא הטיפול ב-exceptions ב-JAVA עדיין לא מוכר לך (וסביר שכך) אז אל דאגה. בהמשך הספר מובא פרק שלם שמטפל בנושא ה-Exceptions, ונושא זה יוסבר שוב.

הרשאת הגישה של המתודה החדשה, אשר באה לתפוס את מקומה של מתודה אחרת אשר הועברה בהורשה, יכולה להיות זהה לזו של המתודה המקורית או מרחיבה (משמע, מרחיבה את אפשרויות הגישה אליה). לדוגמא: מתודה שהוגדרה במחלקת האב עם הרשאת הגישה protected – למתודה שתתפוס את מקומה במחלקה היורשת (overriding) ניתן לתת את הרשאת הגישה protected או public אך לא ניתן לתת לה את הרשאת הגישה private. הסיבה לכך היא כדי שלא יקרה מצב שבזמן ריצה הגירסה של המתודה שתצטרך לפעול לא תוכל לפעול מסיבות של הרשאת גישה.

התכנית הבאה מדגימה את האפשרות לעשות overriding למתודות שמגיעות בהורשה ממחלקה אחרת.

© 2009 Haim Michael. All Rights Reserved.

```
class Person
{
        protected String name;
        protected int age;
        public Person()
        {
        }
        public Person(String nameOfPerson, int ageOfPerson)
        {
                name = nameOfPerson;
                age = ageOfPerson;
        }
        public void printDetails()
        {
                System.out.println("name : " + name + "\nage : " + age);
        }
}
```

© 2009 Haim Michael. All Rights Reserved.

```
class Student extends Person
{
      double average;
      public void printDetails()
      {
            System.out.println("name="+name+" id="+id+" average="+average);
      }
}


class OverriddingDemo
{
      public static void main(String args[])
      {
            Person per = new Student("Moshe",1233232,90);
            per.printDetails();
      }
}
```

בתכנית הנ"ל ניתן לראות כיצד המתודה printDetails מוגדרת מחדש במחלקה Student ובכך מתבצע overriding למתודה printDetails שהוגדרה במחלקה Person. כאשר יש קריאה להפעלת המתודה printDetails תופעל הגרסה

© 2009 Haim Michael. All Rights Reserved.

שהוגדרה במחלקה Student כיוון שהאובייקט שעליו יש קריאה להפעלת המתודה הזו הוא אובייקט שנוצר מהמחלקה Student. לעובדה שה-reference אליו מוחזק בתוך משתנה מטיפוס Person אין כל השפעה על ה-JVM כאשר הוא בוחר בגרסה של המתודה printDetails שעליו להפעיל. כאשר יש overriding למתודה הגרסה שתופעל תהיה תמיד על פי הטיפוס של האובייקט שעליו מנסים להפעיל אותה.

באמצעות overriding ניתן, למעשה, לכתוב במחלקות היורשות מתודות שמותאמות יותר למחלקות החדשות שיורשות מה-base-class, ואז, כאשר מאחסנים references לאובייקטים שנוצרו מהמחלקות היורשות בתוך משתנים מטיפוס המחלקות המורישות (אם אינך זוכר שזה אפשרי כדאי שתחזור/תחזרי לקרוא פרק זה מתחילתו), וקוראים באמצעותם להפעלתה של מתודה (חייבת להיות מוגדרת במחלקה המורישה) והמתודה הוגדרה מחדש במחלקתו של האובייקט, במקרים אלה תפעל המתודה שהוגדרה במחלקה שממנה האובייקט נוצר. אם את/ה מכיר את שפת התכנות C++ אז אופן פעולה זה וודאי מוכר לך. ב- C++ קוראים למתודות שפועלות באופן זה: מתודות (פונקציות) וירטואליות. ב-JAVA כל המתודות וירטואליות (ברירת מחדל) ! למציאת הגרסה של המתודה שיש להפעיל בזמן ריצה קוראים בשם Dynamic Method Dispatching. כאשר בזמן ריצה של תכנית מופעלת מתודה על אובייקט שה-reference שלו מאוחסן במשתנה מסוים, המתודה המופעלת היא המתודה שהוגדרה במחלקה שאליה אותו אובייקט שייך, וזאת גם אם ה-reference של האובייקט מאוחסן במשתנה מטיפוס מחלקת הבסיס(מחלקת האב, המחלקה המורישה). קישור זה לשם הפעלת המתודה המתאימה נעשה בזמן ריצה, ובדרך זו מנוצלת האפשרות לביצוע overriding ביעילות.

לדוגמא:

```
Person p = new Student();

p.getName();
```

בהנחה שהמחלקה Student יורשת מהמחלקה Person, ובהנחה שבשתי המחלקות הוגדרה המתודה getName, אז המתודה getName שתופעל היא זו שהוגדרה במחלקה Student, למרות שה-reference לאובייקט ה-Student מאוחסן במשתנה מטיפוס המחלקה Person.

© 2009 Haim Michael. All Rights Reserved.

פולימורפיזם – Polymorphism

פולימורפיזם polymorpism (בעברית: רב צורתיות) היא היכולת לכתוב שורת/שורות קוד כך שבכל פעם שהיא/הן מופעלת/ות הפקודות שבפועל מתבצעות הן פקודות אחרות.

להלן תיאור אחת הדוגמאות השכיחות לפולימורפיזם:

כפי שכבר נוכחת לדעת, ניתן לכתוב תכנית שיש בה אובייקטים שונים מטיפוס מחלקות שונות אשר כולן יורשות ממחלקה מסוימת, ובכל אחת מהן מתודה מסוימת, שהגיעה בירושה מאותה מחלקה, מוגדרת מחדש (overriding). את ה-references לאובייקטים אלה, כפי שגם בזאת נוכחת לדעת, ניתן לאחסן במשתנים מטיפוס אותה מחלקה שהורישה לכל המחלקות שמהן נוצרו האובייקטים. הפעלת המתודה האמורה על כל אחד מהאובייקטים יכולה להיכתב באופן די דומה. באמצעות המשתנה שמכיל reference לאובייקט תיכתב קריאה להפעלת המתודה. כיוון שה-reference לאובייקט יכול להיות מאוחסן במשתנה מטיפוס המחלקה המורישה ניתן למשל, לכתוב לולאה אשר תפעיל את אותה מתודה על כל אחד מהאובייקטים שה-references אליהם מאוחסנים במערך של references מטיפוס המחלקה המורישה. שורת הקוד להפעלת המתודה תחזור על עצמה שוב ושוב בכל איטרציה נוספת של הלולאה, ובכל פעם תופעל מתודה אחרת (בכל פעם תופעל המתודה המתאימה בהתאם לטיפוס האמיתי של האובייקט).

```
Person peoples[] = }new Teacher(), new Student(), new Manager()};

int totalSalary=0;

for (int index=0; index<peoples.length; index++)

{

            totalSalary = totalSalary + peoples[index].getSalary();

}
```

© 2009 Haim Michael. All Rights Reserved.

הקריאה להפעלת המתודה על כל אחד מהאובייקטים זהה מבחינת ה-syntax אך בכל פעם מופעלת מתודה אחרת. על כל אובייקט מופעלת המתודה אשר הוגדרה במחלקה שלו. ה-method הספציפי שיופעל ייקבע בזמן הריצה בהתאם לסוג האובייקט עצמו (ולא בהתאם לטיפוס המשתנה שמכיל את ה-reference אליו). ניתן גם לפתח מחלקות חדשות שיירשו ממחלקות אלה, וקוד הגישה להפעלת מתודה זו על כל אחד מהאובייקטים שיווצרו ממחלקות אלה עדיין יהיה זהה.

פיתוחן של מחלקות חדשות ושימוש באובייקטים שנוצרים מהן קל יותר הודות לקיומו של הפולימורפיזם.

© 2009 Haim Michael. All Rights Reserved.

המילה השמורה super

באמצעות מילה שמורה זו ניתן (במהלך פעולתה של מתודה שהוגדרה מחדש במחלקה באמצעות overriding) להפעיל את המתודה בגרסתה במחלקה שמעל (במחלקה שהורישה למחלקה הנתונה). אופן השימוש במילה שמורה זו הוא כדלקמן:

יש לרשום אותה, ומייד אחריה למקם נקודה מפרידה, ואחריה למקם קריאה להפעלת המתודה.

דוגמא:

```
class Person
{
      String name;
      int id;
      Person()
      {
      }
      Person(String nameVal, int idVal)
      {
            name = nameVal;
            id = idVal;
      }
```

© 2009 Haim Michael. All Rights Reserved.

```
      void printDetails()
      {
            System.out.println("name="+name);
            System.out.println("id="+id);
      }
}
class Student extends Person
{
      double average;
      Student(String nameVal, int idVal, double averageVal)
      {
            name = nameVal;
            id = idVal;
            average = averageVal;
      }
      public void printDetails()
      {
            super.printDetails();
            System.out.println("average="+average);
      }
}
```

© 2009 Haim Michael. All Rights Reserved.

```
public class StudentPersonDemo

{

      public static void main(String args[])

      {

            Person per = new Student("David",123123,88);

            per.printDetails();

      }

}
```

בדוגמא זו, תוך כדי פעולתה של המתודה printDetails על אובייקט מטיפוס Student יש קריאה להפעלת הגירסה של printDetails שהוגדרה במחלקה Person. בדרך זו, כאשר אנו עושים overriding למתודה אנו יכולים לשלב בהגדרתה של המתודה החדשה את הפעלת המתודה שאנו דורסים. קריאה להפעלת המתודה printDetails על אובייקט שנוצר מהמחלקה Student תגרום להדפסה של השם ותעודת הזהות וכמו כן להדפסת הממוצע.

יש לשים לב לכך, שלא ניתן להפעיל את המילה השמורה super פעמיים, כלומר, לא ניתן, למשל, לכתוב:

```
super.super.speed();
```

© 2009 Haim Michael. All Rights Reserved.

מתודות מסוג final

לשורת הכותרת של מתודה ניתן להוסיף את המילה השמורה final, ובכך למנוע את האפשרות שייעשה לה overriding. הודות להוספת מילה זו אל תחילת שורת הכותרת של המתודה (מייד אחרי הרשאת הגישה) לא ניתן יהיה להגדיר במחלקה היורשת את המתודה הזו מחדש. מתודה שהיא final מועברת בהורשה, מבלי שניתן יהיה להגדירה מחדש (לעשות לה overriding).

להלן דוגמא להגדרתה של מתודה כמתודת final:

```
public final void eraseName()
{
        .
        .
}
```

בהוספת המילה final לשורת הכותרת של מתודה יש שני יתרונות בולטים:

שיפור בביצועים

כאשר המתודה מוגדרת כמתודת final אין צורך בזמן ריצה לאתר את הגרסה המתאימה להפעלה (אין dynamically dispatching). למתודה שמוגדרת כמתודה מסוג final לא ניתן לעשות overriding, ולכן כשהמחשב נתקל בה הוא לא צריך לחפש את המתודה המסוימת שעליו להפעיל. בדרך זו מושג שיפור בביצועים.

הבטחה כי המתודה לא תשונה במחלקות שיורשות

כאשר מתודה מוגדרת כמתודת final מובטח שהיא לא תשונה במחלקות היורשות. לעתים, בהגדרתן של מחלקות זוהי תכונה נדרשת.

© 2009 Haim Michael. All Rights Reserved.

מחלקה מסוג **final**

את המילה final ניתן גם להוסיף לשורת הכותרת של מחלקה. המשמעות היא שלא ניתן יהיה להגדיר מחלקה אחרת אשר תירש ממנה. מחלקה שמוגדרת כמחלקה מסוג final תוכל לרשת אך לא להוריש.

את המילה final יש להוסיף לשורת הכותרת של המתודה מייד אחרי המילה שמייצגת את הרשאת הגישה שלה, ולפני המילה class.

דוגמא:

```
public final class DeadMachine
{
        .
        .
        .
}
```

בדוגמא זו, המחלקה DeadMachine לא יכולה להוריש. היא יכולה רק לרשת.

אחת הדוגמאות למחלקה שמהווה חלק מהמחלקות שקיימות כבר לצד ה-JVM (חלק מהשפה), ושלא ניתן להגדיר מחלקה אשר תירש ממנה היא המחלקה String. מתכנני השפה רצו למנוע את האפשרות להגדיר מחלקה שתתאר מחרוזת תווים ותספק מתודות אשר לא קיימות במחלקה String.

© 2009 Haim Michael. All Rights Reserved.

אופן הפעולה של הבנאים – Constructors Chaining

כאשר נוצר אובייקט ממחלקה שיורשת ממחלקה אחרת, בפעולת יצירתו של האובייקט החדש תהיה גם הפעלה של ה-constructor ששייך למחלקה המורישה בנוסף להפעלת ה-constructor ששייך למחלקה שממנה נוצר האובייקט. ה-constructor שיופעל במחלקה המורישה יופעל רגע לפני שיפעל ה-constructor במחלקה היורשת שממנה אנו מנסים לייצור אובייקט. ההפעלה הזו נעשית באופן אוטומאטי מאחורי הקלעים.

אם המחלקה שממנה נוצר האובייקט יורשת ממחלקה אחרת (הראשונה), והמחלקה האחרת הראשונה יורשת ממחלקה אחרת (השניה) וכך הלאה. . . בפעולת יצירתו של האובייקט החדש יופעלו ה-constructors של כל אחת מהמחלקות. הפעלתם תתחיל ב-constructor שמוגדר במחלקה Object ותמשיך בכל אחת משאר המחלקות לפי סדר ההורשה החל מהמחלקה שנמצאת בראש ההיראכייה (המחלקה Object) כלפי מטה.

הפעלת כל אחד מה-constructors נעשית באופן אוטומטי גם מבלי שב-constructors השונים תיכתב קריאה להפעלתו של constructor מסוים במחלקה המורישה. ה-constructor שיופעל בכל אחת מהמחלקות כאשר אין קריאה ספציפית להפעלת constructor מסוים הוא ה-default constructor (ה-constructor ללא פרמטרים). אם הוא לא קיים (ה-default constructor , שהוא ה-constructor אשר לא מקבל ערכים בעת הפעלתו) אז תתרחש שגיאת קומפילציה.

כדי לקרוא להפעלתו של constructor מסוים (במקום ה-default constructor) מבין ה-constructors שקיימים במחלקה המורישה יש לכתוב בשורה הראשונה של ה-constructor את המילה השמורה super ומייד אחריה בסוגריים את הערכים הנשלחים. על פי הערכים שנשלחים יופעל ה-constructor המתאים במחלקה המורישה.

התכנית הבאה מדגימה את אופן הקריאה להפעלתו של constructor מסוים באמצעות המילה super.

© 2009 Haim Michael. All Rights Reserved.

```
class Person
{
        String name;
        int id;
        public Person()
        {
                System.out.println("The Person() constructor");
        }
        public Person(String nameOfPerson, int ageOfPerson)
        {
                System.out.println("The Person(String,int) constructor");
                name = nameOfPerson;
                age = ageOfPerson;
        }
        public void printDetails()
        {
                System.out.println("name="+name);
                System.out.println("id="+id);
        }
}
```

© 2009 Haim Michael. All Rights Reserved.

```
class Student extends Person
{
      double average;
      Student()
      {
            System.out.println("Student() constructor");
      }
      Student(String nameVal, int idVal, double averageVal)
      {
            super(nameVal,idVal);
            System.out.println("Student(String,int,double) constructor");
      }
}
public class ConstructorsChainingDemo
{
      public static void main(String args[])
      {
            System.out.println("calling Student()...");
            Person perA = new Student();
            System.out.println("calling Student(“Moshe”,123123,98)...");
            Person perB = new Student("Moshe",123123,98);
      }
}
```

© 2009 Haim Michael. All Rights Reserved.

ניתן באמצעות תכנית זו גם להבחין בכך שסדר הפעלת ה-constructors מתחיל במחלקה הבסיסית ביותר, המחלקה שמורישה לאחרות, המחלקה Person, ואחר כך ממשיך לפי ההיררכיה במחלקות האחרות.

© 2009 Haim Michael. All Rights Reserved.

המתודה finalize

אם המחלקה המורישה כוללת בתוכה את המתודה finalize אז היא חייבת להיקרא באופן מפורש מתוך המתודה finalize שמוגדרת במחלקה היורשת. המתודה finalize ב-JAVA אינה דומה באופן פעולתה ל-destructors שקיימים ב C++. המתודה finalize אשר מוגדרת במחלקה המורישה לא תופעל באופן אוטומטי עם תום חייו של האובייקט שנוצר מהמחלקה היורשת. הקריאה להפעלת המתודה finalize תיעשה באמצעות המילה השמורה super:

```
super.finalize()
```

שורה זו תיכתב בתוך המתודה finalize אשר תוגדר במחלקה היורשת.

המתודה finalize מופעלת כאשר אובייקט מסיים את חייו בעקבות פעולת ה-garbage collector אשר מנקה את הזיכרון שמשמש אותו. כיוון שאין כל ביטחון שה-garbage collector אכן ינקה את הזיכרון שמשמש אובייקט מסויים בעקבות הכנסת null לכל אחד מהמשתנים שמחזיקים ב-reference שלו יש להימנע מלהסתמך על המתודה finalize. פעולות שברצוננו להבטיח את ביצוען עם תום חייו של אובייקט יש לבצע מבלי להסתמך על הפעלתה של המתודה finalize.

© 2009 Haim Michael. All Rights Reserved.

מחלקות אבסטרקטיות - **Abstract Classes**

מחלקות אבסטרקטיות הן מחלקות מבניות (שלדיות) שלא ניתן לייצור מהן אובייקטים. הדרך המקובלת להגדרתה של המחלקה כמחלקה אבסטרקטית היא באמצעות הוספת המילה abstract לשורת הכותרת של המחלקה.

לדוגמא:

```
public abstract class Car
{
        .
        .
}
```

המחלקה Car היא דוגמא לאופן הגדרתה של מחלקה אבסטרקטית. את המילה abstract יש להוסיף אחרי המילה שמציינת את הרשאת הגישה ולפני המילה class.

המחלקה האבסטרקטית משמשת כמחלקת בסיס, אשר ממנה מחלקות אחרות יירשו. הגדרת המחלקה האבסטרקטית יכולה לכלול בתוכה הגדרות של משתנים ומתודות. המתודות יכולות להיות מתודות אבסטרקטיות, כלומר, מתודות שלדיות שבתחילת שורת הכותרת שלה מופיעה המילה abstract, ובסופה במקום בלוק מופיע ; . המשמעות שיש למתודה אבסטרקטית דומה למשמעות שיש לפונקציה וירטואלית טהורה ב- ++C.

לדוגמא:

© 2009 Haim Michael. All Rights Reserved.

```
public abstract class Shape

{

            public abstract double area();

            . . .

}
```

בעצם הגדרתה של מחלקה כמחלקה אבסטרקטית ניתן יהיה לחייב כי בכל אחת מהמחלקות שיירשו ממנה (מחלקות שיווצרו מהן אובייקטים . . .) יידרסו המתודות האבסטרקטיות באמצעות מתודות חדשות שאינן אבסטרקטיות (שאם לא כן, אז גם מהמחלקה החדשה לא ניתן יהיה ליצור אובייקטים).

המחלקה האבסטרקטית לא חייבת לכלול בתוכה רק מתודות אבסטרקטיות. היא בהחלט יכולה לכלול בתוכה גם מתודות שאינן אבסטרקטיות (מתודות עם גוף), והיא בהחלט יכולה לכלול בתוכה גם הגדרות של משתנים והגדרות של construcros.

מהמחלקה האבסטרקטית לא ניתן לייצור אובייקטים. יש צורך להגדיר מחלקה אחרת אשר תירש מהמחלקה האבסטרקטית, ובה יש להגדיר באופן מסודר את המתודות שהוגדרו כמתודות אבסטרקטיות במחלקה האבסטרקטית, שאם לא כן, אז גם המחלקה היורשת תיחשב למחלקה אבסטרקטית, וגם ממנה לא ניתן יהיה לייצור אובייקטים.

© 2009 Haim Michael. All Rights Reserved.

ממשק – Interface

למרות ש-JAVA לא תומכת בהורשה מרובה, קיימת ב-JAVA האפשרות להגיע לתוצאות, שדומות לאלה שמושגות באמצעות הורשה מרובה, באמצעות השימוש ב-interfaces.

Interface היא מחלקה מדומה שיכולה להכיל אך ורק מתודות אבסטרקטיות (abstract) בעלות הרשאת הגישה public, ומשתנים שהם static ו- final. ניסיון לייצור משתנה שאיננו עומד לכאורה בקריטריונים אלה יתברר כניסיון שתוצאתו בכל זאת משתנה מסוג final ו- static. המתודות שמוגדרות ב-interface לא יכולות להיות: native, static, synchronized, final, private או protected.

הגדרת ה-interface נעשית באופן הבא:

```
interface  InterfaceName
{
                    כל המתודות שמוגדרות חייבות להיות abstract
                                    ועם הרשאת הגישה public !

                    כל המשתנים חייבים להיות static ו- final !
}
```

ניתן גם להוסיף בתחילת שורת ההגדרה את הרשאת הגישה.

© 2009 Haim Michael. All Rights Reserved.

ה-interface יכלול מתודות ומשתנים שנרצה שיתממשו במחלקות שיישמו אותו.

כדי שמחלקה תממש interface מסוים יש לעשות שימוש במילה השמורה implements. יש להוסיף אותה אל סוף שורת הכותרת של המחלקה שאותה אנו מגדירים ואחריה את שמו של ה-interface המיושם.

דוגמא:

```
public interface Flyable
{
            . . .
}


public class Aircraft implements Flyable
{
            . . .
}
```

במידה שהמחלקה שאנו מגדירים יורשת ממחלקה אחרת וגם מיישמת `interface` אז אנו נציין בשורת הכותרת שלה את העובדה שהיא מיישמת `interface` בהמשך לאחר שנציין את המחלקה שממנה היא יורשת.

```
public class Bird extends Animal implements Flyable
{
            . . .
}
```

© 2009 Haim Michael. All Rights Reserved.

בדוגמא האחרונה המחלקה Bird יורשת מהמחלקה Animal ובנוסף מיישמת את Flyable. המשמעות היא שבמחלקה Bird יש לממש את כל אחת מהמתודות האבסטרקטיות שהגיעו בהורשה מה-interface ששמו Flyable. כמו כן, במידה שגם המחלקה Animal היא מחלקה אבסטרקטית אז יש לממש גם את כל אחת מהמתודות האבסטרקטיות שהגיעו ממנה.

על כל אובייקט שיווצר מהמחלקה Bird ניתן יהיה להפעיל את כל אחת מהמתודות שהוגדרו במחלקה Animal (הן מגיעות בירושה מהמחלקה Animal), את כל אחת מהמתודות שהוגדרו במחלקה Bird וגם את כל אחת מהמתודות שמוגדרות ב flyable ושבמחלקה Bird נעשה להן overriding.

בהגדרתו של interface יש משום יצירתו של טיפוס (type) חדש. באופן דומה לשימוש במחלקות ניתן להגדיר משתנים שהטיפוס שלהם הוא שמו של interface, מתודות שיש להן פרמטר/ים מטיפוס שהוא interface ומתודות שהערך המוחזר שלהן הוא מטיפוס interface. בהמשך לדוגמא האחרונה, נוכל לרשום:

```
Flyable fly = new Bird();
fly.startFlying();
```

בהנחה כמובן שהמתודה startFlying מוגדרת ב-Fly. כדאי לשים גם לב לעובדה שבאמצעות המשתנה fly ניתן יהיה להפעיל אך ורק את המתודות שהוגדרו ב-interface ששמו Flyable. המשתנה fly הוא מטיפוס ה-interface שהגדרנו (Flyable) ולכן המהדר (ה-compiler) מסוגל לראות דרכו רק את המתודות שמוגדרות באותו interface שהגדרנו (Flyable)..

השימוש ב-interfaces מקנה את היכולת להגדיר מחלקות שיונקות את תכונותיהן ופעולותיהן מיותר ממקור אחד, מבלי להסתבך בהיררכית מחלקות מסובכת, מבלבלת, בעייתית ובעלת סתירות רבות שנובעות מקבלה בהורשה מרובה ממחלקות שונות מתודות זהות עם מימוש שונה (כפי שקורה ב ++C).

© 2009 Haim Michael. All Rights Reserved.

ניתן להגדיר מחלקה שתיישם יותר מ-interface אחד. אם המחלקה מיישמת יותר מ-interface אחד אז יש לרשום את שמות ה-interfaces בצירוף פסיקים מפרידים.

דוגמא:

```
class Bird extends Animal implements Flyable, Moveable
{
        …
}
```

ניתן להגדיר Interface שלא מוגדרת בו אף מתודה ואף קבוע, ושמטרתו היחידה לסמן את המחלקה שמיישמת אותו. סימון מחלקה באמצעות Interface שמיושם בהגדרתה נעשה אודות למילה השמורה instanceof. באמצעות instanceof ניתן לדעת אם אובייקט נתון נוצר ממחלקה שמיישמת Interface מסוים... או לא... ולטפל בו בהתאם.

משפט התנאי:

if (*referenceToAnObject* instanceof *InterfaceName*)

. . .

יהיה true אם האובייקט נוצר ממחלקה שבהגדרתה יושם ה-Interface ששמו צוין. אופן השימוש המלא במילה השמורה instanceof יוסבר בהמשכו של הפרק. אחת הדוגמאות ל-interface שמשמש לסימון מחלקות הוא ה-interface ששמו: Cloneable.

הדוגמא הבאה מציגה את ה-interface ששמו Printagle אשר מיושם בשלוש מחלקות שונות: Person, Car ו-Rectangle. שלוש מחלקות מתחומים שונים ואולי אף מהיררכיות שונות. הדוגמא גם מציגה כיצד באמצעות ה-interface נוצר מכנה משותף בין אובייקטים שנוצרים מהמחלקות השונות אשר מיישמות את אותו interface _ה-interface ששמו Printable).

© 2009 Haim Michael. All Rights Reserved.

```
public interface Printable

{
     public abstract void printDetails();
}


class Person implements Printable

{
     private String name;
     private int id;
     public Person()
     {
     }
     public Person(String nameVal, int idVal)
     {
          name = nameVal;
          id = idVal;
     }
     public void printDetails()
     {
          System.out.println("\nPerson Object");
          System.out.println("name="+name);
          System.out.println("id="+id);
     }
     public String getName()
    {
     return name;
    }
     public int getId()
    {
     return id;
    }
}


public class Car implements Printable

{
     private String brand;
     private long id;
     public Car(String brandVal, long idVal)
     {
          brand = brandVal;
          id = idVal;
     }
```

© 2009 Haim Michael. All Rights Reserved.

```
      public void printDetails()
      {
            System.out.println("\nCar Object");
            System.out.println("brand="+brand);
            System.out.println("id="+id);
      }
}


public class Rectangle implements Printable

{
      private double height;
      private double width;
      public Rectangle(double wVal, double hVal)
      {
            width = wVal;
            height = hVal;
      }
      public void printDetails()
      {
            System.out.println("\nRectangle Object");
            System.out.println("width="+width);
            System.out.println("height="+height);
      }
}


public class PrintableDemo

{

      public static void main(String[] args)
      {
            Printable vec[] = new Printable[3];
            vec[0] = new Car("Toyota",78788);
            vec[1] = new Rectangle(30,20);
            vec[2] = new Person("David",232323);
            for(int i=0; i<vec.length; i++)
            {
                  vec[i].printDetails();
            }
      }

}
```

© 2009 Haim Michael. All Rights Reserved.

שינוי טיפוס - Casting

כפי שכבר ראינו, אין כל בעיה לאחסן reference לאובייקט במשתנה מטיפוס המחלקה שהורישה למחלקה שממנה האובייקט נוצר.

לדוגמא:

```
Person per = new Student();
```

באופן דומה, ראינו גם כי אין כל בעיה לשלוח למתודה שמצפה לקבל reference מטיפוס מסוים reference לאובייקט מטיפוס מחלקה שיורשת מאותו טיפוס מסוים.

באופן כללי ניתן לומר כי ביצוע הפעולות שהפוכות לפעולות שתוארו לעיל איננו אפשרי. לא ניתן לאחסן בתוך משתנה מטיפוס מסוים reference מטיפוס שהוריש לטיפוס של אותו משתנה (לא ניתן לאחסן בתוך משתנה מטיפוס Student את ה-reference לאובייקט מטיפוס Person כיוון שה-reference לאובייקט מטיפוס Person הוא גם מטיפוס Person). באופן דומה, לא ניתן לשלוח למתודה reference מטיפוס שהוריש לטיפוס הפרמטר של אותה מתודה (לא ניתן לשלוח למתודה שיש לה פרמטר מטיפוס Student את תוכנו של משתנה מטיפוס Person כיוון שהטיפוס של ה-reference שנמצא בתוך אותו משתנה Person הוא גם כן מטיפוס Person ו-reference מטיפוס Person לא יוכל להיקלט בתוך משתנה כדוגמת פרמטר של מתודה שהוא מטיפוס Student).

עם זאת, לעתים הפעולות שתוארו לעיל אפשריות בכל זאת. המקרים שבהם הן מתאפשרות הם אותם מקרים שבהם ה-reference הוא מטיפוס ששונה מטיפוס האובייקט. לדוגמה, reference מטיפוס Person לאובייקט מטיפוס Student ייחשב מטיפוס Person כיוון שהוא מאוחסן במשתנה מטיפוס Person. אם את אותו reference מנסים להכניס למשתנה מטיפוס Student, והמחלקה Student יורשת מ-Person אז השמת ה-reference אל תוך המשתנה מטיפוס Student יכולה להתאפשר, ובלבד שיבוצע ל-reference שינוי טיפוס (casting) כך שיהיה מטיפוס Student במקום Person. ה-

© 2009 Haim Michael. All Rights Reserved.

casting מתאפשר כיוון שהאובייקט שאליו ה-reference מכוון נוצר מהמחלקה Student (מחלקה שיורשת מ-Person).

ביצוע casting ל-reference מתאפשר אך ורק אם תוצאות הבדיקה הבאה true:

referenceToObject instanceof *class/InterfaceName*

ערכו של הביטוי הלוגי הנ"ל true, אך ורק אם ה-reference ממלא את אחד התנאים הבאים:

1. ה-reference הוא לאובייקט מטיפוס המחלקה ששמה מצוין
2. ה-reference הוא לאובייקט מטיפוס מחלקה שיורשת מהמחלקה ששמה מצוין
3. ה-reference הוא לאובייקט מטיפוס מחלקה שמיישמת את ה-interface ששמו מצוין

בכל אחד מהמקרים הנ"ל ערכו של הביטוי יהיה true, ולכן, ניתן יהיה לבצע ל-reference האמור casting כך שישנה את טיפוסו ויהיה מטיפוס המחלקה או ה-interface ששמם צוין בפקודת ה-instanceof.

ביצוע בדיקה של casting באמצעות שימוש באופרטור instanceof לפני שמבצעים את ה-casting בפועל מאפשר לנו להבטיח שלא ייזרק exception בזמן ריצת התכנית כתוצאה מניסיון לבצע casting שאיננו חוקי.

דוגמא:

נניח כי קיימות המחלקות Animal ו-Bird וכי המחלקה Bird יורשת מהמחלקה Animal. כמו כן, נניח שהמחלקה Bird מיישמת את Flyable. כמו כן, נניח כי קיימת המחלקה ArtificialBird אשר יורשת מהמחלקה Bird.

נניח כי יצרנו אובייקט מהמחלקה Bird באופן הבא:

```
Fly fly = new Bird();
```

תחת הנחות אלה ניתן לסכם ולומר:

© 2009 Haim Michael. All Rights Reserved.

ערכו של הביטוי fly instanceof Bird הוא true.

ערכו של הביטוי fly instanceof ArtificialBird הוא false.

ערכו של הביטוי fly instanceof Flyable הוא true.

יש לזכור, כי ה-casting נעשה רק לטיפוס של ה-reference. האובייקט לא משתנה ולא משנה את טיפוסו. רק ה-reference אליו משנה את טיפוסו.

© 2009 Haim Michael. All Rights Reserved.

הפיכת ערך מטיפוס בסיסי לאובייקט ולהפך

Converting Primitive Types to Objects & Vice Versa

לא ניתן לבצע casing לערך מטיפוס בסיסי כך שיהפוך להיות אובייקט או reference לאובייקט. עם זאת, בחבילת המחלקות (package) ששמו java.lang קיימות מספר מחלקות שמשמשות ליצירת אובייקטים שיכולים לייצג את כל אחד מהטיפוסים הבסיסיים. שמן של מחלקות אלה מעיד על ייעודן. כך לדוגמא, קיימות המחלקות Integer, Float, Boolean וכו'.

באמצעות המתודות שמוגדרות בתוך כל אחת מהמחלקות אלה ניתן לייצור מכל אחת מהן אובייקט אשר יקביל לאחד הטיפוסים הבסיסיים שקיימים ויתאר ערך בסיסי מסויים. כל מחלקה מבין קבוצת מחלקות אלה משמשת ליצירת אובייקט שיכול לתפקד בדומה לערך מטיפוס בסיסי מסוים כלשהו.

דוגמא:

```
public class PrimitiveDemo
{
        public static void main(String args[])
        {
                Integer intVar = new Integer(70);
                int number = intVar.intValue();
                System.out.println("number="+number);
        }
}
```

© 2009 Haim Michael. All Rights Reserved.

המתודות שתם זמנן - Deprecation

מתודות שנחשבות למיושנות כיוון שבעקבות התפתחותה של השפה וקביעתם של סטנדרטים חדשים פעולותיהן ותפקידיהן ראוי שיבוצעו באמצעות מתודות אחרות. ב-JAVA מתודות אלה נחשבות ל-deprecated. כאשר מהדרים תכנית ב-JAVA ובתכנית נעשה שימוש ב-מתודות אשר נחשבות ל-deprecated מקבלים על כך הודעת הזהרה מתאימה (warning). אם רוצים, ניתן להדר את קובץ קוד המקור בתוספת התגית deprecation- ובדרך זו לקבל מידע יותר מפורט על המתודות שנחשבות למיושנות. כדאי להימנע משימוש במתודות שנחשבות ל-deprecated כיוון שעם התפתחות השפה הן התמיכה בהן עשוייה להיפסק. הדוגמא הבאה מציגה שימוש ב-deprecated method.

כתיבת שורה זו בביצוע ההידור לקובץ קוד המקור הבא תגרום להצגת הסברים מפורטים על המתודות המיושנות שנעשה בהם שימוש. `javac –deprecation DeprecatedDemo.java`

```
import java.util.*;
public class DeprecatedDemo
{
        public static void main(String args[])
        {
                        Date birthday = new Date(1970,1,12);
                        System.out.println(birthday);
        }
}
```

© 2009 Haim Michael. All Rights Reserved.

מחלקות מוכנות ליצירת מבני נתונים

ב-JAVA קיימות מחלקות מוכנות, שאובייקט מטיפוסן מייצג מבנה נתונים (משמע, אובייקט מטיפוס כל אחת מהן יכול לשמש בתור אובייקט שמחזיק באובייקטים אחרים במבנה מסוים).

המחלקות שבולטות בקבוצה זו הן:

Vector

Stack

Hashtable

LinkedList

וזוהי רק חלק מהרשימה.

מידע מלא ומפורט על מחלקות אלה ניתן למצוא ב-API Documentation.

© 2009 Haim Michael. All Rights Reserved.