# מחלקה Class

## הגדרה

תכנות מונחה עצמים היא תפיסה בעולם התכנות לפיה התוכנה מורכבת מאוסף עצמים (אובייקטים) שונים. כל אובייקט הוא בעל תכונות ושיטות משלו, ומכלול העצמים יוצר את התוכנה המוגמרת. תכנות מונחה עצמים מסייע לבניית קוד מודולרי, פשוט לשינוי, קל לשליטה גם בתוכניות גדולות, ורב שימושי, כלומר קטע קוד אחד יכול להשתלב בתוכניות רבות.

כדי ליצור אובייקט יש לכתוב מחלקה שבה נכתוב את כל השדות של האובייקט, כיצד בונים אותו, ואת כל השיטות (מתודות) שניתן להפעיל עליו. לאחר שכתבנו את המחלקה ניתן ליצור אובייקטים רבים ממחלקה זו שהם בעצם משתנים מהטיפוס של המחלקה. שם המחלקה תמיד מתחיל באות גדולה. כדי להצהיר על מחלקה נכתוב:

public class Class\_name {

//class variables, constructors and methods

}

## משתני המחלקה

משתני המחלקה הם המשתנים אותם מכיל האובייקט בתוכו. הם יכולים להיות מכל סוג שהוא: משתנים פשוטים כמו int או String, מערכים, או אף אובייקטים אחרים. לעיתים קרובות הם מכונים גם "שדות". מצהירים עליהם מיד לאחר שמצהירים על שם המחלקה, ויש להשתדל לאתחל אותם למען בהירות הקוד. לאחר שבנינו אובייקט מטיפוס המחלקה נוכל לפנות אל משתני המחלקה (אלא אם כן המשתנה הוגדר כ-private) על ידי שנרשום אופרטור נקודה לאחר שם האובייקט - object.class\_var. אמנם בדרך כלל משתנים אלו יוגדרו פרטיים, ונכתוב מתודות get ו-set עבור כל אחד ממשתני המחלקה, כדי שאם מישהו ירצו לגשת למשתנה המחלקה הוא יעשה זאת לפי הדרך שאנו רוצים. וכך נמנע נזקים אפשריים בגישה ישירות למשתני המחלקה.

### משתנים סטטיים

משתני מחלקה סטטיים אלו הם משתנים שנוצרים פעם אחת בלבד בתחילת התוכנית, והם אינם שייכים לאף אובייקט שנוצר מהמחלקה. משתמשים במשתנים סטטיים כדי לתאר את המחלקה כולה או משהו שמשותף לכל האובייקטים. כדי להגדיר שמשתנה הוא סטטי, יש להוסיף לתחילת השורה שבה הוא מוצהר את המילה השמורה static.

public static int var = 0;

כאמור, המשתנה הסטטי לא נמצא באף אחד מהאובייקטים שנוצרו, אלא מוקצה לו באופן אוטומטי שטח זיכרון מיד בתחילת הרצת התוכנית ב-static heap, אפילו אם לא נוצר אף אובייקט מהמחלקה. לכן כל פעם שאובייקטים פונים למשתנה סטטי הם בעצם פונים לאותו שטח בזיכרון. מחוץ למחלקה, ניתן לגשת למשתנה סטטי דרך כל אחד מהאובייקטים שנוצרו מהמחלקה או אפילו דרך שם המחלקה עצמה.

Object\_name.static\_var / Class\_name.static\_var

### יצירת קבועים

משתנים קבועים הם משתנים שאם ננסה לשנות את ערכם נקבל שגיאת קומפילציה. מצהירים על משתנים אלו עם המילה השמורה final לפני שם הטיפוס של המשתנה, ונאתחל אותו לערך הקבוע שיקבל. ניתן להוסיף את המילה השמורה static בתחילת שורת ההצהרה על הקבוע, ובכך לקבל קבוע שאיננו שייך לאף אובייקט של המחלקה, ויחד עם זאת כל האובייקטים שנוצרו מהמחלקה יכולים לגשת אליו. בדרך זו אנו חוסכים הרבה זיכרון, ולכן הוספת static היא הגישה המקובלת ביצירת קבוע.

public static final double EPSILON = 0.0000001;

## בנאי - Constructor

בנאי הוא מתודה שנהוג לכותבה לאחר משתני המחלקה, ותפקידה ליצור ולאתחל אובייקט חדש ולתת ערכים לכל משתני המחלקה שאינם סטטיים. שמו של הבנאי זהה לשם של המחלקה. יכולים להיות מספר בנאים שמקבלים ארגומנטים שונים, כך שכל בנאי יוצר אובייקט לפי הארגומנטים שהוא מקבל. פעולה זו נקראת "העמסת בנאים". בתוך הבנאי אסור שתהיה פקודת return ואפילו לא return void.

public Class\_name (type arg, …) {}

הקריאה לבנאי מתבצעת בעזרת המילה השמורה new, וההבדלה בין הבנאים היא לפי סוג ומספר הארגומנטים שהכנסנו בבנאי. אם רק מכריזים על אובייקט אך לא יוצרים אותו בעזרת הבנאי האובייקט לא נוצר. הדבר היחיד שנוצר על ידי הכרזה בלא קריאה לבנאי הוא מצביע לאובייקט מהטיפוס עליו הכרזנו.

Class\_name name = new Class\_name (type arg, …);

### בנאי ברירת מחדל

הוא בנאי שאינו מקבל ארגומנטים כלל. אם אנו לא מגדירים בנאי כזה במחלקה, אזי Java יוסיף באופן אוטומטי בנאי ברירת מחדל (Default Constructor), שבו כל אחד ממשתני המחלקה יקבל ערך ברירת מחדל לפי הטיפוס שלו. אמנם מכמה סיבות הקשורות לירושה חשוב שאנו נגדיר תמיד בנאי ברירת מחדל.

### בנאי מעתיק

בנאי מעתיק הוא בנאי שמקבל אובייקט מסוג המחלקה, ויוצר לו אובייקט חדש במקום אחר בזיכרון שהוא עותק מדויק לאובייקט המקורי. פעולה זו נקראת "העתקה עמוקה", והיא שונה מלהגדיר אובייקט חדש ולהשוות אותו לאובייקט המקורי, שאז שני האובייקטים בסך הכל מצביעים לאותו מקום בזיכרון.

public Class\_name (Class\_name obj) {

this(obj.get\_var(), …);

}

## שיטות - Methods

שיטות או מתודות הן פונקציות שניתן להפעיל על האובייקט שניצור מהמחלקה. המתודות שיש בכמעט כל מחלקה הם get ו-set עבור כל משתנה מחלקה, toString, ו-equal.

### העמסה - Overloading

בתוך אותה מחלקה ניתן לכתוב מספר מתודות בשם זהה, שמה שמבדיל ביניהן הוא מספר הפרמטרים ו/או סוגם. כדי שניתן יהיה לכתוב מספר מתודות בשם זהה חייב להיות שוני או במספר הפרמטרים או בטיפוס הערך שלהם. פעולה זו נקראת "העמסה" (Overloading). ניסיון לכתוב שתי מתודות זהות אך עם טיפוס ערך מוחזר שונה לא יתקמפל.

### שיטות סטטיות

בדומה לקיומם של משתנים סטטיים כך גם קיימות מתודות סטטיות, שאינן שייכות לאף אובייקט אלא למחלקה. מתודות אלו קיימות וניתנות להפעלה עוד בתחילת התוכנית לפני שבכלל נוצר אובייקט. כלומר, מתודות סטטיות אינן צריכות אובייקט כדי להפעיל אותן, אלא אפשר להפעילם ישירות באמצעות אופרטור נקודה על שם המחלקה.

### שיטות קבועות

אלו שיטות שלא ניתן לדרוס אותן (overriding). מצהירים על שיטות קבועות באמצעות המילה השמורה final. אם נירש מחלקה וננסה לדרוס שיטה בה שמוגדרת final נקבל שגיאת קומפילציה.

## הרשאות - Modifiers

ניתן להגדיר למשתני מחלקה, בנאים ומתודות הרשאות גישה, כך שניתן לגשת אליהן לפי ארבעת האפשרויות הבאות:

1. **private** - ניתן לגשת אך ורק מתוך מחלקה.
2. **package friendly** - ניתן לגשת רק ממחלקות ששייכות ל-package שאליו שייכת המחלקה. אם המחלקה אינה שייכת ל-package ספציפי המשמעות שהיא שייכת ל**-**default Package,ורק מחלקות בחבילה זו יוכלו לגשת למחלקה שלנו.
3. **protected** - ניתן לגשת רק מהמחלקה עצמה, או ממחלקות מאותו package, או ממחלקות שיורשות ממנה.
4. **public** - ניתן לגשת מתוך כל מחלקה.

### עקרון הכימוס

ככלל, יש להשתדל להעניק את הרשאת הגישה private לכמה שיותר משתני מחלקה. הסיבה לכך היא כדי שמתכנתים אחרים יוכלו לגשת למשתני המחלקה אך ורק דרך מתודות שאנחנו כתבנו ולא יהיה להם גישה ישירה אליהם. כך אנו בעצם מבטיחים שמשתני מחלקה רגישים מכילים כל העת ערכים חוקיים בלבד.

מתודות לעומת זאת בדרך כלל נגדיר כ-public, אולם לעתים גם אותם נגדיר כ-private. אנו נעשה זאת כאשר מתודה מבצעת פעולות או נותנת שירות חשוב ורגיש ללוגיקה הפנימית של המחלקה, כך שלא נרצה לאפשר לכל מתכנת להפעיל אותה.

## this

המילה השמורה this מכילה מצביע לאובייקט הנוכחי, כלומר האובייקט שעליו המתודה פועלת. נשתמש במילה השמורה this בשני מצבים:

1. בתוך בלוק המחלקה, כאשר רוצים לפנות אל משתני המחלקה מספיק לרשום את שם המשתנה, לדוגמא var. אמנם כאשר כבר יש משתנה בשם זה (כגון שאנחנו הגדרנו אותו או שהמתודה מקבלת ארגומנט בשם זהה), אזי אם נשתמש בשם var הקומפיילר לא יפנה אל משתנה המחלקה. במצב זה נכתוב this.var כדי לפנות אל משתנה המחלקה var.
2. כאשר בהרצה של מתודה רוצים שבתוכה היא תפעיל מתודה נוספת, ולשלוח אל המתודה השנייה את המצביע של האובייקט שעליו המתודה הראשונה פועלת. במקרה כזה נוכל לשלוח אל המתודה השנייה את המילה this בלבד, כך שהקומפיילר יבין שנשלח האובייקט שעליו המתודה הראשונה פועלת.

# תכנון נכון של מחלקה

## תחביר

**השתמש בסכמה קבועה ועקבית של הגדרת מחלקה -** הקפדה על תבנית ברורה בהגדרתה של מחלקה תורמת לבהירות הקוד. כך למשל, אם מחליטים כיהמשתנים של המחלקה יופיעו לפני המתודות שלה, ומקפידים על כך בכל המחלקות שמגדיריםאזי בהירות הקוד גדלה. באופן דומה, הקפדה על סגנון נאה, והערות עקביות ומסודרות באמצעות ה-Javadoc, כל זה גם תורם לבהירות הקוד.

**להשתדל להמעיט במספרם של משתנים מטיפוס בסיסי בהגדרת המחלקה -** מחלקה שיש בה מספר רב מאוד של משתנים מטיפוס בסיסי מעלה את השאלה שמא ניתן היהלהגדיר במקום מחלקה אחת מספר מחלקות, ובכך להקנות לתוכנית רמה גבוהה יותר של מודולאריות.

**להשתדל לאתחל משתנים של המחלקה -** למרות שמשתנים של מחלקה מאותחלים באופן אוטומטי בערכי ברירת המחדל על פי טיפוס הערךשל כל אחד ואחד מהם, אתחול מכוון מקנה בהירות רבה יותר לקוד.

**לתת למחלקות ולמתודות שמוגדרות בתוכן שמות שמעידים על פעולתן -** מתן שמות שניתן להבין מהם את תפקיד המחלקה, תפקיד המתודה וגם תפקידו של המשתנה - חשובביותר. בהירות הקוד עוזרת גם למי שמתכנת וגם למי שרק מסתכל של התכנית הנכתבת להבין טוביותר את פעולתה של התכנית.כעיקרון ניתן למשתנה מחלקה את הערך אותו הם מייצגים, ולמתודות ניתן שם **אקטיבי** שמייצג את הפעולה שעושים, לדוגמא checkPassword.

**להשתדל לקבוע את הרשאת הגישה private לכל משתנה -** במתן הרשאת הגישה private למשתנים רבים ככל האפשר מבטיחים יותר את שלמות ונכונות נתוניהאובייקט.

**לקבוע מתודות שמאפשרות גישה למשתנים הפרטיים של המחלקה רק כאשר יש צורך בכך -** כדי לאפשר גישה למשתני מחלקה עם הרשאת הגישה private, יש לכתוב לפחות שתי מתודותמתאימות: האחת לצורך קבלת ערכו של המשתנה (get), והשנייה לצורך קביעת ערכו של המשתנה (set).כתיבת שתי מתודות אלה לכל משתנה שיש לו את הרשאת הגישה private גם כשאין בכך צורךפוגעת בבהירות של הקוד, ופוגעת במטרה שיש להרשאת הגישה private.

## עיצוב המחלקה - S.O.L.I.D

עקרונות SOLID הם חמישה עקרונות בסיסיים בעיצוב מונחה-עצמים. המחשבה מאחורי העקרונות היא שכאשר הם מיושמים יחדיו בפיתוח של מערכת תוכנה, היא תהיה יותר קלה לתחזוק והרחבה לאורך הזמן.

באופן עקרוני, קוד גרוע הוא קוד שבו:

* כל שינוי משפיע על הרבה חלקים בקוד.
* שינוי בקוד משפיע על אזורים לא קשורים בקוד.
* קוד לא פריק. לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שלשמם נכתב הקוד במקור.

האופי המרכזי של הבעיות האלו הוא יותר מידי תלות בתוך הקוד. עקרונות SOLID באים לתת קווים מנחים שיגרמו לנו להימנע מלכתוב קוד עם הבעיות הנ"ל.

### Single responsibility

לכל מחלקה צריך להיות תחום אחריות אחד עליו היא אחראית בצורה מלאה. טעות מאוד נפוצה היא שכאשר בונים מחלקה שמייצגת אובייקט כלשהו, אזי מוסיפים לה שיטות שלא בהכרח קשורות לאובייקט אך במבט ראשוני נראה שנוח אם האובייקט יוכל לבצע אותן. פעמים רבות שיטות אלו מאוד מגבילות ודורשות שינויים מהותיים כאשר מרחיבים או משנים משהו במחלקה. לדוגמא, עבור אובייקט שמייצג משתמש, לא נרצה שתהיה שיטה שמקבלת אימייל ובודקת אם הוא תקין. במקום זה נבנה מחלקה מיועדת שבודקת מידע.

באופן עקרוני, כדאי לפצל מחלקות למחלקות קטנות יותר כאשר זה אפשרי. מספר רב יותר של מחלקות מקנה לתוכנית רמה גבוהה יותר של מודולאריות. חשוב להשתדל לעבוד בכיוון זה.

### Open for Extension, Closed for Modification

מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים. כלומר, שהוספת שיטות למחלקה לא ידרוש מאיתנו לשנות קוד שכבר כתבנו. גם הוספה של מחלקות אחרות לתוכנית לא אמור לשנות מחלקות שכבר כתבנו. נעשה זאת בעזרת תכנון נכון של המחלקה ושימוש נכון בממשקים.

### Liskov substitution principle

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל. עצה לכך היא שכאשר כותבים מחלקה שיורשת ממחלקה אחרת או מממשת ממשק, אזי תמיד נסתכל על המחלקה שאנו כותבים כאל אובייקט מסוג מחלקת האב או הממשק עם תוספות קלות. באופן כללי, תמיד כדאי להשתמש בממשקים ולא בירושה.

### Interface segregation principle

יש לדאוג לממשקים מצומצמים, כך שיאלצו מחלקה שמממשת את הממשק לממש כמה שפחות שיטות. פתרון לבעיה זו הוא ממשקים שמממשים ממשקים.

### Dependency inversion

מחלקות high level לא צריכות להשתמש באופן ישיר במחלקות low level, כלומר מחלקות המתארות מקרה כללי לא צריכות להשתמש במחלקות המתארות מקרה פרטי. הסיבה לכך היא שאם בשלב כלשהו נרצה להוסיף עוד מחלקה, המייצגת מקרה פרטי לתוכנית שלנו, נצטרך לשנות גם את המחלקה הכללית.

הפתרון לבעיה זו היא ליצור ממשק, המייצג גם כן מקרה כללי, שאותו יממשו המחלקות המתארות מקרה פרטי. כעת במחלקת ה-high level ממנה התחלנו, נוכל להשתמש בממשק זה במקום. כל הוספה של מחלקה המייצגת מקרה פרטי תממש את הממשק.

# ממשקים -Interface

## הגדרה

במקרים רבים בתכנות אנו יוצרים מחלקות רבות עם מכנה משותף, לדוגמא: אם נרצה לכתוב תוכנית שבה נייצג סגל עובדים בחברה, כאשר יש הבדל בין מנהל, עובד, רואה חשבון, עורך דין ומנקה, נצטרך לממש מחלקה עבור כל אחד מאובייקטים אלו. אמנם בוודאי שבתוכנית שלנו יהיה שיטות שנרצה להפעיל על כל אחת מהמחלקות, לכן נצטרך להגדיר שיטה עבור כל מחלקה, וכך נקבל קוד ארוך ומסורבל שיכול לגרום לשגיאות. כדי לפתור בעיה זו ישנו מנגנון ב-Java הנקרא "ממשק". מנגנון זה מאפשר לנו ליצור מחלקה מדומה "סגל" שמוגדרות עליה שיטות מסוימות, וכל מחלקה יכולה לממש ולהרחיב אותה, כך שכעת נוכל להסתכל על המחלקה המממשת בתור מחלקת סגל עם תכונות נוספות. כך נוכל להגדיר שיטה אחת המקבל אובייקט "סגל" ולהתייחס רק לשיטות המשותפות לכל המחלקות המממשות אותה.

ממשק הוא למעשה מחלקה מדומה, שבה יש אוסף שיטות לא ממומשות שמגדירות מכנה משותף לכל המחלקות שיממשו ממשק זה. שימוש בממשקים מאפשר לנו להסתכל על אותה מחלקה כל פעם בכובע שונה. כדי לבנות ממשק במקום המילה class נשתמש במילה השמורה interface.

Public interface Staff{…}

ממשק יכול להרחיב מספר ממשקים אחרים, כך שכעת מי שיממש את הממשק האחרון יהיה חייב לממש את כל השיטות של כל הממשקים בהיררכיה. כדי שממשק ירחיב ממשק נשתמש במילה השמורה extends.

Public interface Staff extends Human{…}

### יתרונות

* מודולאריות -
* קוד קצר -

### תכונות

* כל השיטות בממשק הן ציבוריות (public) ואבסטרקטיות, כלומר שיטות ריקות שהן רק הצהרה ובלי גוף. מ-Java גרסה 8 אפשר שלשיטה יהיה גוף אך יש לציין בתחילת השיטה "default".
* שיטות בממשק לא יכולות להיות private, protected, final, synchronized, native.
* כל המשתנים בממשק static ו-final.

## מימוש ממשק

כאשר מחלקה מממשת ממשק כלשהו, היא חייבת לממש את כל השיטות הריקות שהוכרזו בו, אחרת נקבל שגיאת קומפילציה. מחלקה מממשת ממשק יכולה להוסיף משתנים, בנאים ושיטות כרצונה.

בניגוד לירושה, שמוגבלת לירושה של מחלקה אחת בלבד, מחלקה יכולה לממש ממשקים רבים, אמנם אז היא תצטרך לממש את כל השיטות של כל הממשקים מעליה. הכרזה על מימוש של ממשק נעשית באמצעות המילה השמורה implements. אם נרצה שהמחלקה שלנו תממש את הממשקים inter1 ו-inter2, ההכרזה שלה תיראה כך:

public class MyClass implements inter1, inter2{…}

## אובייקט מסוג הממשק

ניתן ליצור אובייקט מסוג ממשק באמצעות בנאי של מחלקה שמממשת את הממשק. אובייקט זה רואה רק השיטות הכלולות בממשק לפי המימוש של המחלקה שהשתמש בבנאי שלה. באופן דומה ניתן ליצור אובייקט מסוג ממשק ולבצע השמה בין אובייקט זה לאובייקט מסוג המחלקה.

## פולימורפיזם

פולימורפיזם (בעברית: "רב-צורתיות") הוא תכונה של שפות תכנות המאפשרת לטפל בערכים מטיפוסים שונים בעזרת ממשק תוכנה אחיד.

פולימורפיזם מהווה תכונה חשובה במתודולוגית תכנות מונחה עצמים ובתכנות פונקציונלי. שפת תכנות התומכת בפולימורפיזם נותנת למתכנת את היכולת לממש אלגוריתמים ומבני נתונים לשימוש כללי, ולגזור מהם צורות שימוש שונות בהתאם לעצמים ולנסיבות המשתנות, באופן מפורש או מובלע.

## instanceOf

הוא אופרטור בינארי שמקבל אובייקט משמאל ומחלקה/ממשק מימין, ומחזיר אמת אם האובייקט הינו מטיפוס של המחלקה/ממשק.

if(obj instanceOf Class\_name)

### ממשק לצורך סימון

שימוש נוסף של ממשקים הוא סימון של מחלקות. נבנה ממשק ריק שלא מוגדרת בו אף שיטה, וכל המחלקות שנרצה לסמן יממשו ממשק זה. כעת באמצעות אופרטור instanceOf נוכל לדעת אילו מחלקות מסומנות ואילו לא. דוגמא לממשק שמשמש לסימון בלבד הוא Serializable.

## default

ממשק לא חייב להיות רק עם שיטות ריקות, אלא אפשר להגדיר שיטה עם תוכן כאשר בתחילת ההצהרה על השיטה, עוד לפני ההרשאה, נרשום את המילה השמורה default. לאחר שעשינו זאת נוכל למלא את השיטה בתוכן עוד בתוך הממשק. כל מחלקה שתממש ממשק זה אינה חייבת לממש את שיטות default אך הן כן יכולות להשתמש בהן באמצעות המילה השמורה super.

# הורשה

## הגדרה

הורשה הינה דרך להגדרת מחלקות על ידי שימוש בתכונות ושיטות אשר הוגדרו על ידי מחלקות אחרות, כלליות יותר. מחלקה אשר יורשת ממחלקה אחרת, מקבלת ממנה את משתני המחלקה, הבנאים, ואת השיטות שלה. פרט לכך המחלקה עצמה יכולה להגדיר תכונות ושיטות נוספות. הורשה אינה קשר היא מנגנון. מחלקה A שיורשת ממחלקה B מקיימת את הקשר "A הוא B". לדוגמא, מחלקת "עובד" תירש ממחלקת "אדם", מפני שעובד הוא אדם, והיינו רוצים שלעובד יהיה את כל התכונות של אדם עם הרחבות המיוחדת לעובד.

הגדרתה של מחלקה שיורשת ממחלקה אחרת נקרא גם בשם extending. בהמשך שורת הכותרת של המחלקה היורשת יש לרשום את המילה extends ומייד אחריה את שמה של המחלקה המורישה. רק לאחר מכן, בתוך הסוגרים המסולסלות של המחלקה המוגדרת ניתן להוסיף משתנים ומתודות אשר יתווספו לאלה שהגיעו בהורשה.

public class Employee extends Human{…}

כל מחלקה יכולה לרשת רק ממחלקה אחת. אין הורשה מרובה, כי אז יכולים להיות שיטות זהות בשני האבות ולא נדע איזה מהם לבצע. תכנון התכנית תחת המגבלה שלא ניתן לרשת מיותר ממחלקה אחת יותר מבני ויותר מסודר. 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.

כל מחלקה שמגדירים ולא מציינים שהיא יורשת ממחלקה אחרת תירש כברירת מחדל מהמחלקה Object את כל המשתנים והמתודות שמוגדרות בה. אם המחלקה שהגדרנו יורשת ממחלקה אחרת אז התכונות והמתודות שהוגדרו במחלקה Object יגיעו בהורשה מאותה מחלקה אחרת. וכן הלאה. מסיבה זו, ניתן לומר, כי התכונות והמתודות שמוגדרות במחלקה Object קיימות בכל מחלקה.

**כיום, בגלל התפתחות השימוש בממשקים, השימוש היחיד בהורשה הוא למחזור קוד, כדי להרחיב מחלקה קיימת שלא רוצים לגעת בה מחשש לשגיאות. אם לא צריך למחזר קוד עדיף להשתמש בממשקים, מפני שמנגנון הירושה הוא לא יעיל ומסובך.**

## הרשאת גישה

מחלקה יורשת רק את המשתנים והשיטות המוגדרות public או protected. מה שמוגדר כ-private או package friendly (ברירת מחדל), אי אפשר לרשת.

## דריסה - Overriding

המחלקה היורשת יכולה להגדיר מתודות חדשות או לתת משמעות חדשה למתודות שנורשו על ידי הגדרתם מחדש. פעולה זו מכונה "דריסה" (Overriding). לדוגמה: שיטת toString המוגדרת במחלקת Object ונדרסת כמעט בכל מחלקה. כאשר נקראת מתודה על עצם ממחלקה מסוימת, נבחרת הגרסה העדכנית ביותר של המתודה.

### final

שיטות קבועות הן שיטות שהוסיפו להן את המילה השמורה final. שיטות אלו עוברות בהורשה אך הן סופיות ולא ניתן לדרוס אותן. אם נירש מחלקה וננסה לדרוס שיטה בה שמוגדרת final נקבל שגיאת קומפילציה. בהוספת המילה final לשורת הכותרת של מתודה יש שני יתרונות בולטים:

* פחות שגיאות - יש מקרים שבהם שינוי של שיטה יכול להוביל לשגיאות ולכן נרצה ששיטה זו לא תשונה אלא תישאר קבועה.
* יעילות - כאשר השיטה מוגדרת final אין צורך לאתר את הגרסה של השיטה שמתאימה לאובייקט הנוכחי, אלא מיד עם המילה final מגלים כי יש רק גרסה אחת של השיטה. כך משיגים חסכון בזמן.

את המילה final ניתן גם להוסיף לשורת הכותרת של מחלקה. המשמעות היא שלא ניתן יהיה להגדיר מחלקה אחרת אשר תירש ממנה, אך היא עצמה כן יכולה לרשת. את המילה final יש להוסיף לשורת הכותרת של המתודה מייד אחרי המילה שמייצגת את הרשאת הגישה שלה, ולפני המילה class. מחלקה מסוג final לא יכולה להיות abstract.

## שרשרת בנאים

כאשר יוצרים מופע של מחלקה שיורשת ממחלקה אחרת, אזי בפעולה זו גם נוצר מופע של מחלקת האב. ואם מחלקת האב עצמה גם יורשת, אזי קודם ייווצר מופע של המחלקה ממנה ירשה, וכן הלאה. נמצא כי עבור כל אובייקט שאנו יוצרים קודם נוצר מופע של Object. בגלל שרשרת הבנאים, כאשר קוראים לבנאי ברירת מחדל במחלקת הבן, זה ישר קורא לבנאי הברירת מחדל במחלקת האב. אם לא הגדרנו בנאי ברירת מחדל במחלקת האב נקבל שגיאת קומפילציה. לכן תמיד כדאי להגדיר בנאי ברירת מחדל.

ניתן ליצור אובייקט מסוג האב באמצעות הבנאי של הבן. במקרה זה האובייקט ידע רק מה שיש בבן ששייך גם לאב. אמנם אם בבן דרסנו שיטות, אזי נחלק בין שתי אפשרויות: אם השיטה שדרסנו בבן היא לא סטטית, אזי שנקרא לשיטה זו באובייקט המשונה שלנו הוא יפעיל את השיטה שמוגדרת בבן ולא באב. אך אם השיטה שדרסנו היא סטטית, אזי בכל מקרה תופעל השיטה שמוגדרת באב ולא בבן.

ניתן לבצע השמה בין מחלקת האב A למחלקת הבן B, מפני שכל בן הוא גם אב. אך ההיפך לא ניתן מפני שיש איבוד מידע. ניתן לבצע casting, אבל זה יכול להוביל לשגיאה, מפני שיכול להיות של-A יש עוד מחלקות שיורשות ממנו, והאובייקט שעליו עושים casting השתמש בבנאי מסוג הבן. לכן לפני שעושים casting יש לבדוק שאכן אובייקט האב הוא instanceOf של מחלקת האב או הבן.

A = B // good

B = A // bad

B =(B)A // bad

If(A instanceOf B) // good

B =(B)A

## super

כמו שאמרנו בסעיף קודם, כאשר אנו יוצרים מופע של מחלקת הבן אזי גם נוצר באופן אוטומטי מופע של מחלקת האב. נוכל לפנות למופע זה באמצעות המילה השמורה super. ובכך לגשת אל משתני מחלקת האב, הבנאים והשיטות שלו. פעולה זו תהיה מאוד שימושית אם דרסנו אחד מהם במחלקת הבן אך בכל זאת נרצה גם לגשת אל המימוש המקורי. אל המשתנים והשיטות ניגש באמצעות אופרטור נקודה לאחר super, ואל הבנאים ניגש באמצעות פתיחת סוגריים לאחר super והכנסת הפרמטרים המתאימים.

# מחלקה אבסטרקטית

מחלקות אבסטרקטיות הן מחלקות שלא ניתן ליצור מהן אובייקטים, אך כן ניתן לרשת אותם. מחלקה אבסטרקטית יכולה להכיל משתנים ושיטות עם גוף או בלי גוף, אך כמובן לא תכיל בנאים. מחלקה שיורשת ממחלקה אבסטרקטית מקבלת גם את כל הקוד שלה ויכולה לדרוס שיטות. אם במחלקה האבסטרקטית יש שיטות ריקות (הצהרה בלבד), אזי בניגוד לממשק שבו המחלקה שמממשת חייבת לממש את כל השיטות, במחלקה אבסטרקטית המחלקה היורשת לא חייבת לממש את השיטה. אמנם אם נגדיר במחלקה האבסטרקטית גם את השיטה כאבסטרקטית זה יחייב את המחלקה היורשת לממש אותה (אלא אם כן גם המחלקה היורשת היא אבסטרקטית).

נמצא שמחלקה אבסטרקטית היא כעין שילוב בין ממשק לירושה. מצד אחד, כמו בממשקים, אי-אפשר ליצור ממנה אובייקטים, ומשמשת כמעין מבנה בסיס למחלקות היורשות ממנה, ואף ניתן לחייב אותן לממש חלק מהשיטות אם הם אבסטרקטיות. אך מצד שני, כמו בירושה, אפשר למחזר ממנה משתנים ושיטות ולא רק הצהרות.

נשתמש במחלקה אבסטרקטית כאשר יש לנו מבנה בסיס שלא נרצה ליצור ממנו אובייקט, אך כן נרצה שישמש בסיס למחלקות אחרות, כך שבמחלקות אלו לא נהיה חייבים לממש כל תכונה של הבסיס. לדוגמא: לא נרצה לבנות אובייקט מצורה Shape, אך כן נרצה שישמש בסיס למחלקות Square, Circle. אמנם שיטה כמו "מספר צלעות" רלוונטית לריבוע אך לא לעיגול, לכן נגדיר את Shape כמחלקה אבסטרקטית שבה שיטה זו היא לא חובה למימוש.

הדרך המקובלת להגדרתה של המחלקה כמחלקה אבסטרקטית היא הוספת המילה abstract לשורת ההגדרה של המחלקה. מחלקה אבסטרקטית לא יכולה להיות מוגדרת final, כי אז אי אפשר לרשת ממנה והיא תהיה חסרת ערך.

public abstract class Shape{…}

# גנריות

## מחלקה גנרית

כאשר נכריז על המחלקה, נוסיף לשמה סוגריים משולשים עם מילה כלשהי שתייצג את טיפוס המשתנה, באופן הבא:

public class MyGeneric<T>{…}

כעת, בקוד של אותה המחלקה, נוכל לכתוב שם זה (בדוגמה שלנו - T) במקום טיפוס משתנה מסוים. אין חובה, אך נהוג להשתמש באותיות גדולות בודדות למטרה זו. למשל: E, T, K, V.

## פונקציה גנרית

פונקציה גנרית יכולה לקבל כל סוג של אובייקט. כדי לבנות פונקציה כזו יש לרשום בהצהרת הפונקציה לפני הטיפוס החזרה <T>, ואזי בפרמטרים של הפונקציה תקבל טיפוס מסוג T. יש לבנות פונקציות אלו בזהירות רבה.

public static <T> boolean isInT(T x, T[] arr) {…)

## מבנה נתונים גנרי

הוא מבנה נתונים שיכול להחזיק כל סוג של אובייקט בתוכו. רוב מבני הנתונים הבסיסיים ב-java הם גנריים. לאחר שהגדרנו את הטיפוס של המבנה נתונים לא ניתן להכניס אליו אובייקטים אחרים מטיפוס שונה.