סיכום לקורס

תכנות מונחה עצמים

(67125)

סמסטר א', תש"פ

תוכן העניינים

להערות ותיקונים: עמרי מרום - omri.marom@mail.huji.ac.il

הקדמה

מבנה הציון

- 50% - שישה תרגילים.
- 50% - מבחן. במידה ולא יהיה ניתן לקיים את המבחן באוניברסיטה המשקל של התרגילים עשוי לעלות וציוני המבחן ינורמלו.

מאפיינים של תוכנה טובה

עבור המשתמש:

- התוכנה צריכה לעבוד (לבצע את המשימה, ולעמוד בדרישות)
- להיות קלה לתיפעול
- להיות מהירה ויעילה
- להיות Fail-Safe: להתמודד עם שגיאות ולמנוע קריסה
- להיות Fool-Safe: למנוע שגיאות של משתמש בתום לב. לדוגמה: מעבד תמלילים שמתריע בפני המשתמש על סגירת התוכנה אם הקובץ לא נשמר.
- להתמודד עם ניסיונות זדוניים לפגוע בתוכנה
- התוכנה מתאימה לסביבות שונות

עבור המתכנת:

- תוכנה מהירה לכתיבה
- קל לבדוק ולדבג את התוכנה
- התוכנה קריאה - קל להבין את התוכנה
- אפשר לעשות שימוש חוזר בתוכנה
- קל לעדכן ולשנות את התוכנה

סינטקס בשפת Java

Java היא אחת משפות התכנות הפופולרית ביותר.

טיפוסי נתונים

כל משתנה בשפת Java יכול להיות אחד מבין שני הדברים הבאים: **משתנה פרימיטיבי** או **רפרנס** (מצביע לאובייקט).

משתנה פרימיטיבי

משתנה פרימיטיבי מחזיק טיפוס נתונים בסיסי; כל המשתנים הפרימיטיביים מאותו הסוג (לדוגמה, כל המספרים השלמים) דורשים את אותה כמות של זיכרון, כמפורט להלן:

`int`	**מספר שלם**	5 , 7 , -1 , 0	4 בייטים
`long`	**מספר שלם ארוך**		8 בייטים
`float`	**שבר עשרוני (קצר)**		4 בייטים
`double`	**שבר עשרוני**	5.0 , -2.6 , 0.0	8 בייטים
`char`	**תו בודד**	'a' , 'b' , '#' , '?'	2 בייטים
`boolean`	**משתנה בוליאני**	true או false	גודל לא מוגדר

אין כאן למשל משתנה שמייצג מחרוזת, המיוצגת בשפה על-ידי מחלקה מובנית.

הצהרה על משתנים תיעשה לצד טיפוס הנתונים שלהם, וניתן להגדיר מספר משתנים מאותו טיפוס באותה שורה:

```
int x = 3;
boolean iHadCake = true, iHadDinner = false;
```

בדומה למשתנים, צריך להכריז גם על טיפוס הנתונים שפונקציה מקבלת כארגומנטים.

ניתן להמיר בין סוגי משתנים בצורה מפורשת או מרומזת. במקרה של המרה שתגרור אובדן מידע (למשל ממספר עשרוני למספר שלם) נעדיף לבצע אותה בצורה מפורשת. נשים לב במיוחד את להמרה בין טיפוסים אחרים לבין מחרוזות:

```
double res = 5/2;      // res is 2.0
int res2 = (int) Math.round(5.7);


int a = Integer.parseInt("13"); // a = 13
double d = Double.parseDouble("-0.7"); // d = -0.7
float f = Float.parseFloat("inigo"); // exception!


String s = "" + 5.4; // s = "5.4"
```

מחלקות עוטפות

לכל טיפוס פרימיטיבי יש class wrapper, כדוגמת המחלקה `Integer` עבור `int`. קיימות בהן מתודות שימושיות עבור אותו הטיפוס - כגון פונקציות המרות, קבלת ערכים מקסימליים ועוד. ניתן להשתמש במחלקות העוטפות גם למשל כשנעבוד עם מבני נתונים שמקבלים אובייקטים ולא טיפוסים פרימיטיביים. קיימת המרה אוטומטית בין `int` לבין `Integer` שנקראת Autoboxing או Unboxing, כמתואר להלן:

```
Integer intObj = 5;  // int to Integer - Autoboxing
int five = intObj; // Integer to int - Unboxing
```

רפרנס - מצביע לאובייקט

רפרנס אינו אובייקט בפני עצמו אלא רק מצביע לאובייקט קונקרטי. לכל רפרנס יש טיפוס (שהוא השם של המחלקה). נתבונן לדוגמה על השורה הבאה:

```
Bicycle Bike1 = new Bicycle(1);
```

באגף שמאל אנו מגדירים רפרנס חדש לאובייקט מסוג `Bicycle`, ובאגף ימין אנו יוצרים אובייקט חדש (content) על-ידי קריאה לבנאי של המחלקה. ניתן להגדיר מספר רפרנסים לאותו האובייקט:

```
Bicycle bike1 = new Bicycle(1);
Bicycle bike2 = bike1;
Bicycle bike3 = bike1;
```

אם נשנה משהו ב-`bike1`, זה ישפיע גם על `bike2` ועל `bike3`.
מבחינת זיכרון, למרות שהגדרנו שלושה משתנים, רפרנסים הם יחסית זולים בזיכרון, בניגוד ליצירה של אובייקטים חדשים (על-ידי קריאה לבנאים) שמבזבזת הרבה זיכרון. הזיכרון של אובייקט "מיותם" ללא רפרנס שמצביע עליו ישוחרר באופן אוטומטי על ידי מנגנון של Garbage Collector. אנחנו לא יודעים בדיוק מתי זה יקרה, ובכל מקרה כדאי לנסות לצמצם את כמות האובייקטים שאנו מגדירים כדי לחסוך בזיכרון.

המחלקה String

String היא המחלקה הנפוצה ביותר בשפה שמתנהגת במובנים מסוימים כמו טיפוס פרימיטיבי, אך בפועל היא מחלקה לכל דבר. בניגוד לתו, מחרוזת תיכתב עם גרשיים (כפולות). בין היתר, למרות שלא מדובר בטיפוס פרימיטיבי, ניתן להגדיר מחרוזת באמצעות השמה ולא עם קריאה לבנאי:

```
String myString = "hello";
```

קיימות מתודות מובנית שימושיות שונות שמוגדרות על מחרוזות; לדוגמה – המתודה `length()` שמחזירה אורך של מחרוזת נתונה. מחרוזת היא אובייקט שהוא immutable - כלומר שאינו ניתן לשינוי; ברגע שיצרנו מחרוזת מסוימת לא ניתן לשנות את התוכן שלה.

משתנים קבועים

שפות תכנות רבות מאפשרות ליצור משתנים קבועים, כלומר שלא ניתן לשנות את הערך שלהם במהלך ריצת התכנית לאחר שהאובייקט נוצר. בשפת Java נשתמש במילה השמורה `final` בעת ההצהרה על משתנה כדי להגדיר משתנה קבוע:

```
final int myInt = 5;
final String str = "hello";
final Bicycle myBike = new Bicycle(1);
```

ניתן לבצע השמה של ערך לשדה של מחלקה שמוגדר כקבוע בזמן ההצהרה עליו או בתוך קונסטרקטור של מחלקה; אם מדובר במשתנה מקומי, אפשר לתת לו ערך גם אחרי ההכרזה.

אם מדובר במשתנה שהוא רפרנס, ההכרזה עליו כקבוע תגרור שלא נוכל להשים לתוכו משתנה אחר (לגרום לו להצביע לאובייקט אחר בזיכרון), אבל כן נוכל לבצע שינויים באובייקט עצמו.

המוטיבציה להשתמש בקבועים היא תכונות של אובייקטים שלא נרצה לשנות, ולכן נרצה להגן על התוכנה ולמנוע באגים.

נדגיש את ההבדל בין immutability של מחרוזת לבין הגדרת משתנה בתור `final`; לא ניתן לשנות את התוכן של מחרוזת, אבל כן ניתן להשים לתוכה מחרוזת חדשה. לעומת זאת, אם נגדיר מחרוזת בתור קבועה עם `final` לא נוכל לעשות גם את זה:

```
String s = "hello";
s.charAt(0) = 'y'; // not possible (immutable)
s = "goodble"; // possible!

final String s = "hello";
s.charAt(0) = 'y'; // not possible (immutable)
s = "goodble"; // not possible (final)
```

אפשר גם להגדיר מתודות בתור `final` כדי למנוע ממחלקות יורשות לדרוס את המתודה, ואף להגדיר מחלקות בתור `final` כדי למנוע את האפשרות לרשת מהן.

מערכים

מערך הוא רצף באורך קבוע של משתנים מטיפוס מסוים:

```
int [] intsArray = new int[2];
```

ניתן לגשת ישירות לכל תא במערך (תאים של מערך מאונדקסים החל מ-0) ולקבל את גודלו של מערך באמצעות `arr.length`.

אם מגדירים מערך של מחרוזות הערכים בתוכו מאותחלים לערך התחלתי `null`, כיוון שמחרוזות הן אובייקטים.

ניתן לאתחל מערך עם ערכים בצורה ישירה, בתנאי שהדבר נעשה בשורת ההצהרה עליהם:

```
int [] intsArrya = new int []{32,64};
String [] strArray = new String[]{"Dana", "Jack"};
```

ניתן לבצע איטרציה על מערכים גם בלולאת for-each. לדוגמה, פונקציה שסוכמת את איבריו של מערך:

```
int [] intArray = new int [5];
   …
int result = 0;

for(int i : intArray){

   result += i;
}
re
turn result;
```

בשפת Java אנו מגדירים **מערך דו-מימדי** בתור מערך של מערכים, ואז הוא יכול להיות גם מערך לא מלבני:

```
int [][] intsArray = {{1,2}, {4,8}, {1,2,3}};
int [][] intsArray = new int[2][3];
```

פונקציות

כיוון שכל הקוד בשפת Java נכתב במסגרת מחלקות, כל הפונקציות הן **מתודות** או פונקציות סטטיות. המבנה של מתודה הוא כדלהלן:

```
returnValue name(type arg1) {
    ...
    return;
}
```

אם רוצים להחזיר ממתודה אובייקט של מחלקה כלשהי, ערך ההחזרה יהיה השם של המחלקה. הטיפוס של פונקציה שלא מחזירה כלום הוא `void`.

סקופים

סקופ הוא קטע קוד שתחום בתוך סוגריים מסולסלים; לדוגמה: סקופ של מחלקה, סקופ של מתודה, סקופ של לולאה או תנאי. כל משתנה לוקאלי מוכר רק בתוך הסקופ שבו הוגדר ובסקופים פנימיים יותר, ולא ניתן לגשת אליו מסקופ חיצוני.

ככלל, עיקרון טוב הוא להגדיר משתנים לוקאליים בסקופ הפנימי ביותר שניתן. הגדרה של משתנה בסקופ חיצוני מידי שלא לצורך מיותרת ומקשה על ההבנה. יוצא דופן לכך הוא מקרה של הכרזה על משתנים (על אחת כמה וכמה משתנה "מורכב" שכולל שדות רבים) בתוך סקופ של לולאה שתחזור על הקוד פעמים רבות, דבר שיפגע בביצועים ובזיכרון.

אופרטורים

אופרטורים אריתמטיים: `+` `-` `*` `/` `%`

כתיבה מקוצרת: `a += 5` במקום `a = a + 5`

קידום: `n++` או `++n` ששקולים ל-`n = n+1`; ההבדל ביניהם הוא האם קודם מתבצעת דגימה של הערך או קודם קידום שלו.

אופרטורים בוליאניים: `==` `!=` `<` `<=` `>` `>=`

אופרטורים לוגיים: `&&` `||` `!`

תנאים

אם כותבים רק פקודה אחת בלבד אחרי תנאי אפשר לכתוב בלי הסוגריים המסולסלים, אבל זה פחות קריא.

```
if (a == b) {
   ...
}

else if (a == c) {
   ...
}

else {
   ...
}
```

קיים גם מבנה של `switch` שהוא מחליף `else` – `if` עבור אחד ספציפי. צריך להוסיף `break` בסיום של כל מקרה, אחרת כל המקרים שאחריו ירוצו גם. אפשר לרוץ במבנה של על משתנים מטיפוס מחרוזת, תו, מספר שלם או enum.
לדוגמה, שאילתות על הערך של המשתנה button:

```
switch (button) {

   case 1:
      makeTea();
      break;

   case 2:
      makeHotChocolate();
      break;

   default:
      makeCoffee();
      break;
}
```

תיעוד

```
// one-line command


/*
   multi-line command
 */


/**
 * java-doc
 */
```

לולאות

```
while (a<0) {
    ...
}

for (int i = 0; i < 10; i++) {
    ...
}

for (int i: intArray) {
    ...
}

for (String str: str Array) {
    ...
}
```

אפשר להשתמש גם בפקודות `break` ו-`continue` כמו ב-Python כדי להפסיק את הריצה או לדלג על האיטרציה הנוכחית ולדלג לאיטרציה הבאה.

קלט ופלט

הדפסה למסך

הדפסה למסך תיעשה בעזרת הפונקציה הבאה:

```
System.out.println("A String!");
```

לחילופין, אפשר לקצר את הפקודה על-ידי הגדרת פונקציית עזר בשם print:

```
void print(String s) {
   System.out.println(s);
}

print("A String!");
```

קבלת פרמטרים משורת הפקודה

מקבלים את הארגומנטים משורת הפקודה כפרמטר של הפונקציה הראשית בתכנית:

```
public static void main(String [] args) {

   int firstNumber = Integer.parseInt(args[0]);
}
```

עבודה עם קבצים

לכל קובץ יש נתיב, ויש שוני במבנה של נתיבים ב-windows וב-linux. יש שתי דרכים לתת נתיב לקובץ – נתיב אבסולוטי (מתיקיית הבית) או נתיב יחסי (ביחס למקום שבו אנחנו נמצאים כרגע).

מחלקת File בשפה מאפשרת עבודה עם קבצים (למשל לבדוק האם קובץ קיים, האם מוסתר, מה גודלו, מתי השתנה) אבל לא לכתוב או לקרוא מקובץ. בעזרת המתודה listFiles() ניתן לקבל מערך של כל הקבצים בנתיב מסוים.

כדי לקרוא ולכתוב מקובץ אפשר להשתמש בספריות java.io.Reader ו-java.io.Writer. קיימת גם גרסה מתקדמת יותר java.nio. לדוגמה:

```
File f =  new File("test.text");
f.delete(); // delete the file

try{

   // Write to file
   FileWriter writer = new FileWriter(f);
   writer.write("Hello World!");
   writer.close();
}

catch(IOException ioException){
   ...
}
```

Streams

קיימת ספריה מובנית של השפה שמאפשרת לעבוד עם Streams, כלומר לקבל או להעביר מידע (מול קובץ או כל מקור אחר). בעבודתנו עם מידע נבחין בין מידע טקסטואלי (שמכיל רצפים של אותיות, שבני אדם יכולים לקרוא – כמו קבצי טקסט וקבצי קוד) לבין מידע בינארי (קבצים שמורכבים מרצפים של בייטים שאינם בהכרח ברי משמעות לאדם הפשוט – כמו קבצי תמונה, מוזיקה, קובץ מקומפל ועוד). צריך להיות תיאום בין שני הצדדים שבשני צידי ה-Stream.

קיימת בשפה המחלקה `java.io` שמאפשרת לעבוד עם Streams. היא כוללת את המחלקות האבסטרקטיות `Reader` ו-`Writer` לעבודה עם קבצי טקסט, והמחלקות האבסטרקטיות `InputStream` ו-`OutputStream` לעבודה עם מידע בינארי. מתוך כל אחת מהן יורשות מחלקות שעובדות עם מקורות מידע שונים.

לדוגמה, כתיבה לתוך הקובץ `mail.txt`:

```
Writer writer = new FileWriter("mail.txt");
writer.write('a');
```

דוגמה נוספת לקריאה וכתיבה מקבצים: אנחנו פותחים את הקבצים בתוך הסוגריים של ה-`try` וכך הקבצים ייסגרו לבד במקרה של בעיה. אם היינו כותבים אותם בתוך הבלוג הם היו נשארים פתוחים במקרה של שגיאה.

```
try (OutputStream output = new FileOutputStream(args[1]);
   InputStream input = new FileInputStream(args[0]);) {

   int result;

   // reading the file
   while ((result = input.read() != -1)){
      output.write(result);
   }
}

catch (IOException ioErrorHandler){
   System.out.println("Couldn't copy file")
}
```

טסטים

קיימים שלושה מדרגים של טסטים שניתן להריץ על הקוד:

- **Unit Test**: בדיקה של חלקים קטנים מאוד של הקוד (2-3 שורות), במנותק משאר הקוד.
- **Integration Tests**: בדיקה שחלקים שונים בקוד פועלים ביחד כיחידה אחת.
- **End-to-End Tests**: בדיקה של ההתנהגות של כל הקוד במלואו.

אנחנו נכתוב טסטים בעזרת הספרייה החיצונית `org.junit.Test`.

כדי להגדיר מתודה בתור טסט חייבים לסמן אותה ככזו בעזרת האנוטציה `@Test`. לדוגמה:

```
import org.junit.Test;

public class Tester{

   @Test
   public void test1(){
      ...
   }
}
```

את הבדיקות עצמן אפשר לכתוב בעזרת פקודות `assert`, אשר גם אותן צריך לייבא בנפרד. לדוגמה:

```
import static org.junit.Assert.*;

@Test
public void test1(){
   assertEquals(stud.getId(), id);

}
```

קיימות פונקציות סטטיות רבות שניתן להשתמש בהם לצרכים דומים; לדוגמה: `assertNotNull`.

בעזרת האנוטציה `@Before` אפשר לכתוב מתודה שתרוץ לפני הריצה של כל אחד מהטסטים; נשתמש בכך למשל כדי ליצור אובייקט שעליו הטסטים ירוצו. לדוגמה, לפני כל טסט על המחלקה "סטודנט" נאתחל מופע של סטודנט עם תעודת זהות כלשהי וציון בקורס אינטרו:

```
public class Tester{

   Student stud;
   int id = 12345;

   @Before
   public void before(){
      stud = new Student(id);
      stud.setGrade("Intro", 95);
   }
}
```

בדומה, בעזרת האנטוציה `@BeforeAll` אפשר לכתוב מתודה שתרוץ פעם אחת בלבד, לפני שהטסטים של המחלקה מתחילים לרוץ.

כדי להריץ מספר מחלקות של טסטים בבת אחת אפשר לכתוב מחלקת TestRunner:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
      Tester.class,
      Tester2.class,
      Tester3.class
})

public class TestRunner{

}
```

גנריות

העיקרון של **גנריות** מאפשר לנו לכתוב קוד מופשט כך שיוכל לעבוד עם טיפוסי נתונים לא פרימיטיביים שונים (מחלקות או מערכים, ובכלל זאת גם מערכים של טיפוסים פרימיטיביים). הטיפוס האמיתי של המחלקה ייקבע בעת יצירת מופע שלה.

ניתן להוסיף פרמטר אחד או יותר למחלקות, ממשקים או מתודות:

```
LinkedList<String> myList = new LinkedList<String>();
HashMap<String,Double> map = new HashMap<String,Double>();
```

בדוגמה לעיל הפרמטר הגנרי מגדיר מה סוג האיברים במבנה הנתונים ומאפשר לעבוד איתו.
אם ננסה לחרוג מהפרמטר שהגדרנו נקבל שגיאת קומפילציה:

```
LinkedList<String> myList = new LinkedList<String>();
myList.add("hello");
myList.add(new(Double(3,14))); // compilation error
```

עבודה עם קוד גנרי מגדירה type safety והופכת את הקוד לקריא יותר.

שגיאות טיפוס

שגיאת טיפוס היא הכנסה של ערך מטיפוס אחד לתוך טיפוס אחר שאינו מתאים לו; לדוגמה:

```
String s1 = new Integer();
```

שגיאה כזו תזוהה בזמן הקומפילציה.
לעומת זאת, שגיאת הטיפוס הבאה היא שגיאה שתזוהה רק בזמן הריצה של התכנית:

```
Object o = new Integer(5);
String s2 = (String) o;
```

כיצד נוכל להימנע משגיאות טיפוס? הפיתרון האופטימלי לכך הוא שימוש בגנריות: תכנות גנרי הוא מנגנון שמבטיח לנו type safety, כלומר שנוכל לזהות שגיאות טיפוס כבר בזמן הקומפילציה של התכנית ולא רק בזמן הריצה.

כתיבת מחלקה גנרית

נוסיף לחתימה של המחלקה סוגריים משולשים עם פרמטר, ונכתוב אותו בכל מקום במחלקה שנרצה שיהיה גנרי:

```
public class Node<T> {
   private T elem;

   public Node(T elem){
      this.elem = elem;
   }

   public T data(){
      return this.elem;
   }
}
```

אפשר ליצור מופע של מחלקה גנרית גם בלי פרמטר (מה שנקרא raw type), אבל זה נחשב שימוש רע ולא יספק לנו type-safety. נהוג להשתמש באות `E` עבור פרמטר גנרי שמייצג אלמנט מסויים, `K` מפתח, `N` מספר, `T` טיפוס ו-`V` ערך.

לא ניתן להכניס טיפוס פרימיטיבי בתור פרמטר גנרי (לדוגמה, במקום `int` נצטרך להשתמש במחלקה העוטפת `Integer`).

דוגמאות לשימוש במחלקה גנרית:

```
Node<String> n = new Node<String>("hello");

LinkedList<String> list = new LinkedList<String>();
list.add("hello");

String s = list.get(0);
list.add(new Integer(5)); // compilation error!
```

נדגיש כי כאשר אנו משתמשים בתוך גוף המחלקה בפרמטר שהגדרנו בחתימה שלה, אנחנו משתמשים באותו הפרמטר שהגדרנו (ולא מגדירים אותו מחדש). לדוגמה, בקוד הבא, גם השדה `next` הוא חוליה עם אותו טיפוס גנרי `T`.

```
public class Node<T>{

   private T elem;
   private Node<T> next = null;

   public Node(T elem){
      this.elem = elem
   }
}
```

אינווריאנטיות

מחלקות גנריות הן אינווריאנטיות: לא משנה מה הקשר (והאם קיים קשר היררכי) בין שני טיפוסים `Type1` ו-`Type2`, לא יהיה קשר היררכי בין מחלקות `MyClass<Type1>` ו-`MyClass<Type2>`; לדוגמה, למרות שהמחלקה `String` יורשת מהמחלקה `Object` לא ניתן לבצע המרה בין `LinkedList<String>` לבין `LinkedList<Object>` והפעולה הזו תגרור שגיאת קומפילציה, למרות שהפעולה הבאה כן חוקית:

```
Object o = new string(...);
```

במילים אחרות, לא ניתן לבצע up-cast דרך הפרמטר הגנרי ולא ניתן ליצור היררכיה בין רשימה של מחלקת בן לרשימה של מחלקת אב. עם זאת, זה **לא** אומר שאי אפשר להכניס לרשימה מקושרת של מחרוזות אובייקט מטיפוס Object, למשל; ההיררכיה נשמרת בטיפוס עצמו אבל לא במחלקה הגנרית.

קלף ג'וקר – Wildcards

כאשר אנו לא יודעים או כשלא משנה לנו מה הסוג האמיתי של הפרמטר ניתן לכתוב במקום הפרמטר סימן שאלה; לדוגמה: `List<?>`.

אם משתמשים בזה לא ניתן להניח דבר על הטיפוס האמיתי של האיברים ברשימה: הדבר היחיד שניתן לדעת הוא שהפרמטר יורש מהמחלקה `Object`, ולכן ניתן לשלוף פריטים מהרשימה רק כטיפוס `Object`. בנוסף, בזמן בניית המחלקה לא ניתן להכניס אליה שום דבר מלבד `null`; הטיפוס האמיתי של הרשימה יכול להיות כל מחלקה שהיא ולכן null הוא האיבר היחיד שיכול להיכנס לכל סוג של רשימה.

רק רפרנסים יכולים להשתמש בקלף ג'וקר, בעוד שאובייקט קונקרטי צריך להכיל פרמטר ממשי:

```
LinkedList<?> c1 = new LinkedList<String>(); // legal!
LinkedList<?> c2 = new LinkedList<?>(); // illegal! Compilation error
```

דוגמאות לשימוש:

```
public boolean isEqualSizes(List<?> c1, List<?> c2) {
   return c1.size() == c2.size();
}

void printList(List<?> c) {
   for (Object e : c){
      System.out.println(e);
   }
}

public void removeAll(List<?> list){
   Iterator<?> e = list.iterator();
   while (e.hasNext()){
      e.remove();
   }
}
```

שימוש בקלף ג'וקר מאפשר לנו גם להשתמש ב"פסודו-פולימורפיזם": אפשר להשתמש ברפרנס של `List<?>` כדי להחזיק רשימות עם פרמטרים שונים. לדוגמה:

```
private List<?> generateListFromUserInput(String str){

   switch(str){

      case "string":
         return new LinkedList<String>();

      case "Integer":
         return new LinkedList<Integer>();

      default:
         return new LinkedList<Object>();
   }
}

List<?> c3 = new LinkedList<String>();
```

מדובר בפיתרון טוב יותר מאפשר להשתמש ברשימה של `Object` מכיוון שאז בגלל האינווריאנטיות לא היינו יכולים להשתמש בפולימורפיזם ולגשת לרשימות עם פרמטרים שונים בתור `List<Object>`.

זה גם פיתרון טוב יותר מאשר רשימה לא גנרית מכיוון שהיא לא type-safe, בניגוד ל- `List<?>` שהיא כן type-safe (מאחורי הקלעים יש לה פרמטר כלשהו אבל אנחנו לא יודעים אותו בזמן כתיבת הקוד).

קלף ג'וקר מצומצם – Wildcard extends

אפשר גם להשתמש ב-wildcard בתור "חסם עליון" ולציין ממי הפרמטר יורש:

```
LinkedList<Amimal> myList = new LinkedList<Dog>(); // Compila. error!

// Works!
LinkedList<? extend Animal> myList = new LinkedList<Dog>();
LinkedList<? extend Animal> myList = new LinkedList<Animal>();
```

כלומר שאפשר להחזיר רשימה עם פרמטר שהוא "לפחות" Animal. לדוגמה:

```
void printAnimals(List<? extends Animal> c) {

   for (Animal e : c){
      e.printEars();
      e.printNose();
   }
}
```

מאותה סיבה כמו קודם, גם במקרה הזה אפשר להוסיף לרשימה רק איברים שהם null! שהרי גם כאן אנחנו לא יודעים מה הפרמטר האמיתי (לדוגמה, יכול להיות שהפרמטר יהיה לברדור וננסה להוסיף כריש).

```
myList.add(new Animal()); // compilation error
myList.add(new Dong()); // compilation error
myList.add(null)); // works
```

אבל כן ניתן לשלוף מהרשימה אובייקטים מטיפוס Animal או כל מה שנמצא מעליה בהיררכיה:

```
Animal a = myList.get(0); // OK
Creature c = myList.get(0); // OK
```

בניגוד לשימוש בקלף ג'וקר רגיל:

```
Dog d = list.get(0); // Compilation error
```

גם כאן, היתרון האפשרות ליהנות מפולימורפיזם ולהחזיק רשימות של חיות ספציפיות, מה שלא היינו יכולים לעשות עם רשימה של Animal בגלל האינוריאנטיות.

אפשר לתת הגבלה גם עם ממשקים ולא רק עם מחלקות, אבל עדיין נשתמש בתוך הסוגריים המשולשות במילה extends ולא במילה implements.

קיימת גם וריאציה הפוכה:

```
List<? super Shape> list
```

במקרה זה נקבל רק מחלקות שנמצאות מעל המחלקה Shape בהיררכיה – כלומר משהו שהמחלקה Shape יורשת ממנו. במקרה הזה כן אפשר להוסיף לרשימה כל אובייקט מטיפוס Shape או ממחלקות שיורשות ממנו (לדוגמה Square, כי כל מחלקת אב של Shape יכולה לייצג גם את Square), אבל ניתן לשלוף רק null.

פונקציות גנריות

פעולות ומשתנים סטטיים משותפים לכול האובייקטים של המחלקה, ובפרט גם לכל סוגי הפרמטרים שמתאפשרים במחלקה; לפיכך, פעולות סטטיות לא יכולות לתמוך בפעולות גנריות (למשל פעולות שמקבלות פרמטר מטיפוס גנרי T). כן אפשר להגדיר פעולות סטטיות וגם מתודות כפעולות גנריות עם פרמטר גנרי ששונה מהפרמטר הגנרי של המחלקה, ובמקרה זה נעדיף כמובן לקרוא לו בשם אחר כדי להימנע מבלבול. לדוגמה:

```
public class Car<T>{
   T wheelShape;


   public<S> void doSomething(S param){
      // S is NOT the same as the class' <T>
   }


   public static<T> void doSomething2(T param){
      // Static function -  T is NOT the same as the class' <T>!
   }
}
```

באופן דומה, מחלקה מקוננת סטטית אינה מקושרת למופע ספציפי של המחלקה החיצונית ולכן לא ניתן להשתמש בפרמטר הגנריות החיצוני – אבל כמו קודם אפשר להגדיר אותה עם פרמטר גנרי אחר.

Erasure

בפועל, מאחורי הקלעים בתהליך הקומפילציה, קוד גנרי מומר לקוד לא-גנרי עם המרות:

```
Node<String> n = new Node<String>("hello");
String s = n.data();

      ||
      \/

Node n = new Node("hello");
String s = (String) n.data();
```

לכן, נקבל את התוצאה המפתיעה:

```
ArrayList<String> l1 = new ArrayList<String>();
ArrayList<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass()); // True!
```

בגלל שבתהליך הקומפילציה הקוד הגנרי שלנו מוחלף בשתי מחלקות מטיפוס `ArrayList`. דהיינו, בשונה ממה שהכרנו ב-++C, כאן לא נוצר עותק נפרד של המחלקה לכל פרמטר גנרי.

מסיבה זו, אין טעם להשתמש ב-`instanceof` כדי לשאול האם אובייקט כלשהו הוא מסוג של אובייקט גנרי עם פרמטר ספציפי, כי גם כאן בתהליך ה-erasure ימחקו את זה; בנוסף, מאותה הסיבה אין טעם לעשות down-casting לרשימה עם פרמטר ספציפי ולא ניתן ליצור מופע של פרמטר גנרי (כי בזמן הריצה איננו יודעים מה הוא) או להגדיר מערך מהטיפוס של הפרמטר הגנרי:

```
E e = new E(); // error!
T [] arr = new T[2]; // error!
```

למרות זאת, מבחינתנו יש יתרון להשתמש בגנריות כיוון שזה נותן לנו מנגנון של type-safety.

הערה על מערכים

בשונה ממחלקות גנריות, מערכים הם קו-אינווריאנטיים! לדוגמה, מערך מטיפוס `Child` כן נמצא בהיררכיה תחת מערך מטיפוס `Parent`.
יש לכך השלכות שליליות וזה עלול לגרום לכל מיני בעיות; לדוגמה:

```
String [] strArray = new String[10];

// Allowed due to covariance (String extends object)
Object [] objArray = strArray;

// Allowed - ObjArray is an array of Objects, Integers are objects
objArray[0] = new Integer(5);
```

מה שיגרור כמובן לשגיאת זמן ריצה.

לפיכך, מכיוון שהשימוש בגנריות נועד למנוע type errors, לא ניתן להגדיר מערך של אובייקטים גנריים! לדוגמה:

```
List<Integer>[] list = new LinkedList<Integer>[2]; // Not allowed
```

מבני נתונים (אוספים)

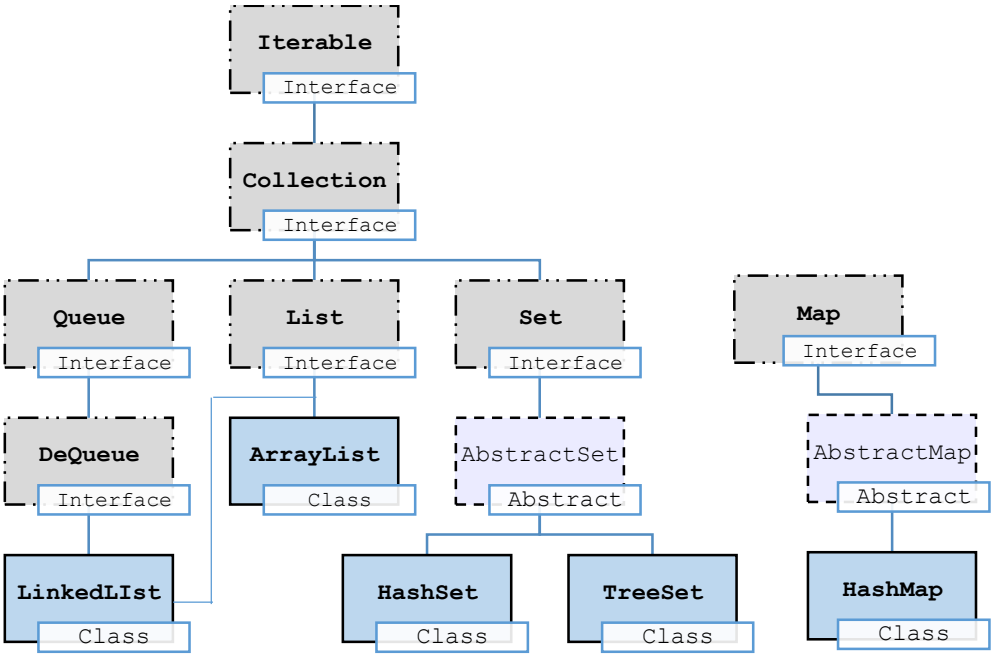
אוסף הוא אובייקט תכנותי שמחזיק אובייקטים אחרים ומאפשר לאחסן מידע ולבצע עליו מניפולציות (חיפוש, מיון). בשפת Java יש רכיב שנקרא Collections Framework, ספריה של מבני נתונים, שניתן להשתמש בה כדי להשתמש ולעבוד עם מבני נתונים; היא כוללת **ממשקים**, **מימושים** ו**אלגוריתמים** למבני נתונים. כדי להשתמש בהם יש צורך לייבא אותם:

```
import java.util.*;
```

השימוש באוספים שימושי מאוד וחוסך עבודה וגם משפר את התפקוד של הקוד וזמן הריצה שלו. כל הרכיבים ב-Collections Framework הם גנריים.

מערך הוא מבנה הנתונים הפרימיטיבי ביותר בשפה אבל הוא לוקה בחסרונות רבים: לא ניתן לשנות את גודלו או לשנות את ההתנהגות שלו, ולכן הוא לא יספיק לפיתרון של בעיות מורכבות.

להלן תיאור חלקי של האוספים שמוצעים במסגרת הספרייה המובנית. כפי שאנו רואים בתרשים הספרייה כוללת ממשקים ומימושים שלהם.



ממשקים

ממשקים של מבני נתונים עונים על שאלות כלליות לגבי מבני הנתונים, שאינן תלויות מימוש; לדוגמה: האם מבנה הנתונים מאפשר להכניס אותו איבר פעמיים; האם מבנה הנתונים ממויין; האם דורשים לממש מתודות מסויימות; Collections Framework מגדיר שבעה ממשקים בסיסיים שאותם כמעט כל המימושים מממשים.

כחלק מהמימוש, הוגדר כי כל מימוש צריך לכלול שני קונסטרקטורים: קונסטרקטור ריק שלא מקבל פרמטרים ויוצר מבנה נתונים ריק, וקופי-קונסטרקטור שמקבל אוסף אחר ומכניס את כל האיברים שבתוכו לאוסף הנוכחי. חשוב לזכור שאין דרך לאכוף את הקונבנציה ולא ניתן "לכפות" על מחלקה לממש אותם.

מבין הפעולות שנתמכות בממשק `Collection` ניתן למנות את המתודה `size()` שמחזירה את מספר האיברים באוסף, `isEmpty()` שבודקת האם האוסף ריק ו-`contains()`שמקבלת אלמנט ובודקת האם הוא נמצא באוסף.

מימושים

נציג כעת את המימושים של מבני נתונים מרכזיים:

ממשק	מימושים	
List<E>	**ArrayList<E>**	**LinkedList<E>**
מייצג רשימה סדורה (לאו דווקא ממויינת) ומאפשרת גישה לאיברים לפי אינדקס. אותו איבר יכול להופיע ברשימה מספר פעמים.	מימוש של **מערך בעל גודל משתנה**; תומך בפעולות `get()` ו-`set()` בזמן קבוע, אבל פעולות שדורשות חיפוש כמו `contains()`, `remove()` בזמן לינארי. פעולת ההוספה ב-O(1) בזמן ממוצע (תלוי אם צריך להגדיל את המערך)	**רשימה מקושרת**; מימוש של רשימה מקושרת דו-כיוונית. תומכת בהוספה בזמן קבוע O(1) אבל כל הפעולות האחרות (חיפוש, מחיקה) בזמן לינארי O(n). תופסת פחות מקום במחשב מאשר המימוש של מערך כיוון שמחזיקה בדיוק את כמות האיברים שצריך.

ממשק	מימושים	
Set<E>	**TreeSet<E>**	**HashSet<E>**
ממדל את הקו נספט של קבוצה מתמטית: קבוצה של איברים ללא סדר שבה כל איבר יכול להופיע פעם אחת בלבד. קיים גם ממשק של סט ממוין: `SortedSet<E>`	מימוש של `SortedSet<E>` שמבוסס על עץ ומסדר את האיברים בסדר ממוין. הוא תומך בפעולות add, remove ו-contains בזמן $O(\log n)$. המימוש הפנימי הוא של עץ AVL.	מימוש שמבוסס על גיבוב. הוא תומך בפעולות הוספה, הסרה וחיפוש בזמן O(1) בממוצע, אבל ללא הבטחה על סדר האיברים.

ממשק	מימושים	
Map<K,V>	**TreeMap<K,V>**	**HashMap<E>**
מאפשר מיפוי של מפתחות לערכים. המפתחות צריכים להיות ייחודיים, אבל הערכים יכולים להופיע כמה פעמים (יכולים להיות שני מפתחות שממופים לאותו הערך). קיים גם ממשק של מפה ממוינת: `SortedMap<K,V>`	מימוש שמבוסס על עץ בינארי, כאשר לכל איבר יש גם מצביע למפתח שלו.	מימוש שמבוסס על גיבוב.

מבנה נתונים נוסף שלא פירטנו לגביו הוא `Queue<E>` המייצג תור ותומך בפעולות ניהול התור (הכנסה לראש התור, שליפת ראש התור והצצה לראש התור), לרוב במנגנון FIFO, למרות שקיימים גם תורי קדימויות עם חוקיות אחרת.

נסכם את המימושים של מבני הנתונים בטבלה נוספת שמציגה את זמני הריצה של הפעולות הבסיסיות:

	הוספה	מחיקה	גישה לפי אינדקס	חיפוש	איטרציה
ArrayList	O(1) בממוצע	O(n)	O(1)	O(n)	O(n)
LinkedList	O(1)	O(n)	O(n)	O(n)	O(n)
HashSet	O(1) בממוצע	O(1) בממוצע	-	O(1) בממוצע	O(T) טבלת הגיבוב
TreeSet	$O(\log n)$	$O(\log n)$	-	$O(\log n)$	O(n)

אלגוריתמים

המחלקה `Collections` מספקת לנו פונקציות סטטיות שימושיות שמאפשרות מניפולציות ופעולות שונות על מבני הנתונים: חיפוש בינארי, מנייה של כמות הופעות של איבר, מיון, עירבול, ועוד. בכלל זאת, כשאנו רוצים לבצע פעולה על מבנה נתונים, ייתכן מאוד שקיים מימוש מובנה שלה ולא צריך לממש אותו בעצמנו.

נציג לדוגמה מיון של רשימה:

```
List<String> strs = new ArrayList<>();
   ...
Collections.sort(strs);
```

איטרטור

כפי שראינו, רוב מבני הנתונים מממשים את הממשק Iterable, כלומר שניתן לעבור עליהם באיטרציה. איטרטור הוא אובייקט שלוקח מבנה נתונים ועובר על האיברים שלו. גם איטרטור הוא אובייקט גנרי, והפרמטר שלו צריך להתאים לפרמטר של מבנה הנתונים. איטרטור מאפשר לעבור על מבנה נתונים גם בלי לדעת מה המימוש שלו; אותו איטרטור יכול לשמש גם עבור מספר מבני נתונים.

שימוש באיטרטורים שונים מאפשר לעבור על מבני הנתונים באופנים שונים (בסדר ישר, בסדר הפוך, בסדר אקראי ועוד); במקרים מסויימים כלל לא מוגדר סדר טבעי לעבור על מבנה נתונים (כמו סט שלא מוגדר עליו סדר).

איטרטורים תומכים בשתי פעולות עיקריות:

- `hasNext()` - האם יש איברים נוספים באוסף.
- `next()` - מקדמת את האיטרטור ומחזירה את האיבר הבא באוסף.

בנוסף, איטרטור תומך במתודה `remove()` שמאפשרת למחוק איבר ממבנה הנתונים תוך כדי האיטרציה.

לדוגמה – איטרציה על רשימה בעזרת איטרטור והדפסת איבריה:

```
List<String> myList =...;
Iterator<String> myIterator = myList.iterator();

while(myIterator.hasNext()) {

   String next = myIterator.next();
   System.out.println(next);
}
```

השוואה בין אובייקטים

כדי שנוכל למיין ולסדר איברים שנמצאים בתוך אוסף צריך להיות מוגדר עליהם יחס סדר כלשהו. אם מדובר באובייקטים מורכבים אנחנו יכולים להגדיר אותו בעצמנו ואף להגדיר מספר יחסי סדר אפשריים בעזרת מימוש הממשק `Comparable` או יצירת אובייקט שמממש את הממשק `Comparator`.

האפשרות הראשונה היא לממש במחלקה את הממשק הגנרי `Comparable` שדורש מאיתנו לממש במחלקה את המתודה `compareTo` שמקבלת אובייקט נוסף ומחזירה את תוצאת ההשוואה ביניהם: המתודה תחזיר את הערך 1 אם המופע הנוכחי גדול יותר מהפרמטר, 1- אם המופע הנוכחי קטן יותר, או 0 אם האובייקטים שווים לפי הקריטריון שנבחר. כדוגמה, נממש פעולת השוואה בין שני מספרים מרוכבים לפי החלק הממשי שלהם:

```
public class ComplexNumber implements Comparable<ComplexNumber>{

   @Override
   public int compareTo(ComplexNumber o) {
      return Double.compare(this.getReal(), other.getReal());
   }
}
```

כעת נוכל לממש מיון על רשימה של מספרים מרוכבים:

```
LinkedList<ComplexNumber> compNums = new LinkedList<ComplexNumber>();
Collections.sort(strs, new ComplexNumber());
```

האפשרות השנייה היא ליצור `Comparator` – אובייקט של מחלקה חיצונית שמממשת את הממשק `Comparator` עם המתודה `compare` שמקבלת שני אובייקטים ומחזיקה מספר חיובי, שלילי או אפס בהתאם לשאלה האם הפרמטר הראשון גדול, קטן או שווה לפרמטר השני.

לדוגמה, `Comparator` שמשווה בין אובייקטים של כלבים לפי ערך ההאש שלהם:

```
public class DogComparator implements Comparator<Dog>{

   @Override
   public int compare(Dog o1, Dog o2){
      return o1.hashCode() - o2.hadhCode();
   }
}
```

כעת נוכל לממש מיון על רשימה של כלבים:

```
LinkedList<Dog> dogs = new LinkedList<Dog>();
Collections.sort(strs, new DogComparator());
```

היתרון במימוש של `Comparator` מאפשר להשתמש בו עבור מספר מחלקות ובכך תורם למחזור של קוד, ובנוסף מאפשר להגדיר מספר אסטרטגיות שונות למיון ולבחור ביניהם בזמן ריצה; מצד שני, אין לו גישה לשדות פרטיים של המחלקה שעשויים להיות רלוונטיים למיון.

טיפול בשגיאות: חריגות

קיימים מספר סוגים של שגיאות בתוכנה: **שגיאות קומפילציה** (שעשויות לקרות למשל בגלל טעויות סינטקס או קריאה למתודות בשמות לא נכונים) שמזוהות על-ידי הקומפיילר, ו**שגיאות זמן ריצה** שאינן מזוהות על-ידי הקומפיילר ונגרמות, בין היתר, בגלל באגים בתוכנה או קלטים לא נכונים מהמשתמש. שגיאות אלו דורשת מהתכנית שלנו להתמודד איתן ולטפל בשגיאות. תכנית טובה תתמודד עם שגיאות במקום הנכון ובצורה הנכונה.

חריגות

עד כה כשרצינו לכתוב מתודה שיכולה להצליח או להיכשל השתמשנו בערכי החזרה כדי להתריע על שגיאה. המנגנון של חריגות מהווה אלטרנטיבה לכך: כאשר מתרחשת בעיה ומתודה לא יכולה להמשיך לבצע את ריצתה, המתודה תזרוק **חריגה** למי שקרא לה. המימוש של חריגות נעשה באמצעות מחלקות בשפה שיכולות לכלול שדות – ובכלל זאת מחרוזת שמתארת את השגיאה.

נציג דוגמה פשוטה לכך: מחלקה של רשימה שמקבלת אינדקס ומחזירה את האיבר במקום שקיבלנו – אבל צריכה להתמודד עם מצבים שבהם הרשימה ריקה. אם המתודה נקראה עם רשימה ריקה תיזרק חריגה: ריצת הקוד מפסיקה, שאר הקוד במתודה לא ירוץ ולא יהיה שום ערך החזרה.

```
public int get(int index) throws ListException{

   if (list.isEmpty()) {
      throw new ListException();
   }
}
```

החריגה שנזרקת מהמתודה מופיעה גם בחתימה שלה עם המילה השמורה `throws` ומהווה חלק אינטגרלי מה-`API` של המתודה כדי שנדע שאנו עשויים לקבל חריגה, ולכן חשוב לתעד את החריגה.

try/catch

קריאה למתודה שעלולה לזרוק חריגה תיעשה בצורה "מוגנת" בתוך בלוק `try` ולאחריו בלוק `catch` שתופס את החריגה שנזרקה, אם הייתה כזו, ומטפל בה. הקוד בתוך בלוק `catch` ירוץ אם אחת המתודות בתוך בבלוק `try` זרקה חריגה. כפי שאנו רואים בדומה לעיל, ניתן לתפוס בנפרד חריגות מסוגים שונים ולטפל בכל סוג בנפרד, בהתאם לסוג הבעיה.

```
try{

   int element = list.get(0);

} catch (ListException e) {
   // hanled list errors

}

 catch (OtherException e) {
   // handle other errors
}
```

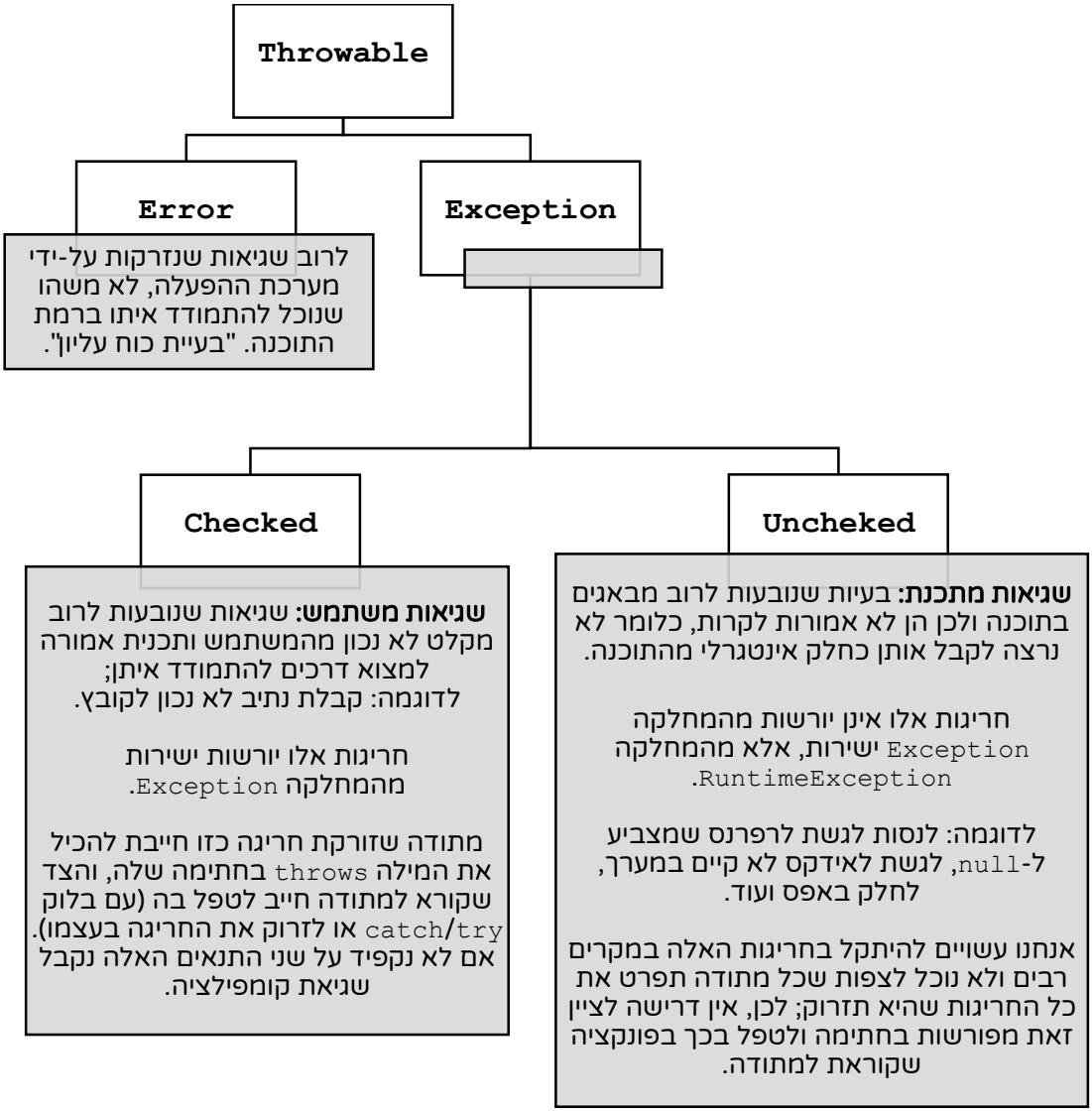
אפשר גם לתפוס מספר סוגים של חריגות באותו בלוק `catch`:

```
catch (SugarException | MilkException e)
```

דרך אחרת לטפל בחריגה היא להעביר את החריגה אחורה בהיררכיה. אם נזרקת חריגה מהפונקציה הראשית אין מי שיתפוס אותה, ולכן היא תחזור למשתמש שהריץ את התוכנה.

סוגי חריגות

קיימת היררכיה של מחלקות של חריגות:



סיבות להשתמש בחריגות

- ליצור הפרדה בין החלק הלוגי בקוד לבין החלק בקוד שמטפל בשגיאות.
- שימוש בחריגות כדרך לפעפע שגיאות במעלה מחסנית הקריאות.
- היכולת לאחד ביחד סוגים של שגיאות (בעזרת פולימורפיזם, בהסתמך על ההיררכיה של חריגות) ולהפריד בין סוגים אחרים.
- חזרה מהירה מפונקציה עמוקה בשרשרת הקריאות לפונקציה המקורית, גם שלא במקרה של שגיאה.
- הקצבת זמן לריצה של קוד (מנגנן של Timeout).
- מימושים של מחלקות עם וריאציות - זריקת `UnsupportedOperationException`; לדוגמה: רשימה שלא מאפשרת להוסיף איברים חדשים.

לא נשתמש בחריגות כדי להחליף פעולות סטנדרטיות ולוגיות של תכנות; לדוגמה: לכאורה במקום להשתמש בלולאת for אפשר לכתוב לולאת while(true) ולעצור אותה כשנצרה עם זריקת חריגה של `IndexOutOfBoundersException`. זה קוד פחות יעיל, לא אלגנטי, לא צפוי ועלול להחביא באגים.

יצירת חריגה חדשה

מחלקה שיורשת ישירות מהמחלקה `exception` תהיה `CheckedException`.

כל מחלקת חריגה צריכה להכיל כשדה את השורה הבאה, אשר נסביר עליה בהמשך בפרק "שיכפול של אובייקטים":

```
private static final long serialVersionUID = 1L;
```

נקפיד על כך שמחלקות של חריגות יהיו באותה חבילה יחד עם המחלקות שזורקות אותן, ולא בחבילה נפרדת של חריגות, כיוון שהחריגות קשורות למחלקות הזורקות אותן יותר משהן קשורות אחת לשנייה.

חבילות

השימוש ב**חבילות** נועד לחלק את הקוד לחלקים בצורה הגיונית, לשמור על סדר פנימי (קצת בדומה לתיקיות במחשב) ולפתור התנגשות בין שמות של קבצים.

נהוג לקרוא לחבילות באותיות קטנות. חבילות של השפה עצמה יתחילו עם `java.` או `javax`.
כדי לשייך קובץ לחבילה צריך להוסיף בשורה הראשונה שלו הכרזה על החבילה:

```
package package.subpackage;
```

כדי לייבא חבילה צריך להשתמש במילה `import`. אפשר לייבא גם משתנים סטטיים בעזרת הפקודה `import static` אבל זה יותר מסובך מבחינת הקומפילציה ולכן נעדיף לייבא את כל החבילה.

כל מחלקה מקבלת שם ייחודי (fully qualified name) בהתאם לחבילות שבהן היא נמצאת שמאפשר להשתמש במחלקות בעלות אותו השם מחבילות שונות; לדוגמה: `vehicles.Driver` לעומת `computers.Driver`.

נהוג לשים חריגות שמחלקות זורקות באותה החבילה ולא במחלקה נפרדת של חריגות, מכיוון שחריגות קשורות למחלקות הזורקות אותן בקשר לוגי יותר מאשר שהן קשורות אחת לשנייה.

חבילות והרשאות גישה

הגדרת הגישה הדיפולטיבית בשפה היא שתכנים (שדות ומתודות) חשופים בפני כל המחלקות באותה החבילה, בעוד שמחלקות אחרות מחוץ לחבילה (וגם בתת-חבילות של החבילה המקורית) לא יכולות לגשת או להשתמש בהן.

נתבונן על הקוד הבא שמציג עבודה עם חבילות:

```
package pack1;

public class A{
   int packageInt;
}
```

```
package pack2;
import pack1.A;

class B {
   A a = new A();

System.out.println(a.packageInt);
// compilation error
}
```

מימין יש לנו את המחלקה `A` בחבילה `pack1` ומשמאל את המחלקה `B` בחבילה `pack2`.
מכיוון שהמחלקה `B` נמצאת בחבילה אחרת, עליה לייבא את המחלקה `A` כדי שתוכל להשתמש בה; אבל ניתן לגשת לשדה `packageInt` של המחלקה `A` רק מתוך אותה החבילה, ולכן נקבל שגיאת קומפילציה.

לעומת זאת, אם המחלקה `B` נמצאת באותה חבילה יחד המחלקה `A` אין צריך לייבא אותה ואפשר לגשת לשדות של `A`:

```
package pack1;

public class A{
   int packageInt;
}
```

```
package pack1;

class B {
   A a = new A();

System.out.println(a.packageInt);
// will work!
}
```

נסכם זאת בטבלה הבאה:

Modifier	Class	Package באותה החבילה, לא בתת-חבילות	Subclass באותה החבילה	Subclass מחוץ לחבילה	World
`public`	✓	✓	✓	✓	✓
`protected`	✓	✓	✓	✓	✗
`default = package`	✓	✓	✓	✗	✗
`private`	✓	✗	✗	✗	✗

כפי שאנו רואים, שדות ומתודות שמוגדרים בתור `protected` חשופים גם בתוך החבילה, ולא רק למחלקות היורשות!

תכנות פונקציונלי

נציג בקצרה כמה כלים לתכנות פונקציונלי בשפת Java, כלומר סגנון תכנותי שבו אפשר להתייחס לתוצאה של חישוב או שאילתא מסוימת כאל פלט של פונקציה.

ממשק פונקציונלי

ממשק פונקציונלי הוא ממשק שמייצג פונקציונליות מסוימת (במקום לייצג נתונים) ומכיל פונקציה אבסטרקטית אחת בלבד (או יחד עם מתודות שמוגדרות בתור `default` ומתודות סטטיות שממומשות בתוך הממשק). כדי להגדיר ממשק פונקציונלי אפשר, אך לא חובה, להוסיף מעל הכותרת של הממשק הפונקציונלי את האנוטציה `@FunctionalInterface`. לדוגמה:

```
@FunctionalInterface
public interface BinaryOperator<T> {

   public T operator(T input1, T input2);
}
```

אחת הדרכים להשתמש בממשק פונקציונלי ולממש אותו היא בעזרת מחלקות אנונימיות.

קיימים מספר ממשקים פונקציונליים שימושיים מובנים בשפה:

- `BinaryOperator<T>` עם המתודה `apply` שמקבלת שני ארגומנטים מטיפוס `T`, מבצעת עליהם פעולה כלשהי ומחזירה ערך מטיפוס `T`.
- `Consumer<T>` עם המתודה `accept` שמקבלת ארגומנט מטיפוס `T`, מבצעת עליו פעולה כלשהי (למשל מדפיסה פלט כלשהו, קוראת למתודה של האובייקט) ומחזירה `void`.
- `Predicate<T>` עם המתודה `test` שמקבלת ארגומנט מטיפוס `T` ומחזירה ערך בוליאני – בודקת האם האובייקט `T` ממלא תנאי כלשהו או לא.
- `Function<T,R>` עם המתודה `apply` שמקבלת ארגומנט מטיפוס `T` ומחזירה ארגומנט מטיפוס `R`.

ביטויי lambda

פונקציית lambda היא מתודה אנונימית שמממשת ממשק פונקציונלי; הטיפוס של המתודה הוא מהטיפוס של הממשק הפונקציונלי שאותו היא מממשת, וניתן להשתמש בה בכל מקום שבו אנו מצפים לקבל ממשק פונקציונלי. הסינטקס של ביטויי lambda כולל את הפרמטרים ואת ערכי ההחזרה (ניתן גם להשמיט את הטיפוסים של הפרמטרים.) לדוגמה:

```
(int x, int y) -> {return x+y;}
```

בתור דוגמה לפונקציית lambda ושימוש במשק פונקציונלי נציג שני מימושים לממשק הפונקציונלי `Comparable` שמממש פונקציה אבסטרקטית אחת בשם `compare` שצריכה למיין שתי רשימות לפי האורך שלהן. מימוש ראשון הוא בעזרת מחלקה אנונימית:

```
Comparator<List> comparator = new Comparator<List>() {
   @Override
   public int compare(List s1, List s2){
      return Integer.compare(s1.length(), s2.length());
   }
};
```

ומימוש שני, קצר ואלגנטי יותר, בעזרת פונקציית lambda:

```
Comparator<List> comparator = (s1, s2) ->
Integer.compare(s1.length(), s2.length()));
```

במקום פונקציות lambda, אפשר להגדיר ממשק פונקציונלי גם עם רפרנס לפונקציה, למשל:

```
Function<String, Double> strToDouble = Double::parseDouble;
```

enum

הסינטקס של enum מאפשר להגדיר מחלקה מסוימת שיכולה לקבל מספר קבוע וידוע מראש של ערכים, ובכך לסייע מאוד לכתוב קוד קריא ואלגנטי יותר, בפרט במקרים שבהם נרצה לאפשר פונקציונליות נפרדת לכל מקרה. לדוגמה:

```
public enum season {SPRING, SUMMER, WINTER, AUTUMN};
```

כל ערך אפשרי מוגדר בתור מעיין מחלקת Singleton. לא ניתן לשנות או להוסיף ערכים בזמן ריצה, ולכן נשתמש במבנה של enum רק במקרים שבהם כל הערכים האפשריים ידועים בזמן הקומפילציה.

יתרון משמעותי נוסף לשימוש במבנה של enum-ים הוא שהם מגדירים type safety – כלומר שאם מתודה שמצפה לקבל פרמטר מסוג `season` תקבל משתנה לא מתאים, הדבר יגרור שגיאת קומפילציה.

אפשר לבצע איטרציה על enum-ים בלולאת for-each בעזרת המתודה `values()`:

```
for (Season season : Season.values()){

   System.out.println(season); // "WINTER", etc
}
```

מדובר במחלקה לכל דבר ועניין ואפשר להגדיר בה קונסטרקטורים, שדות ומתודות. הקונסטרקטור יכול להיות רק פרטי (ברירת המחדל) או בהרשאה דיפולטיבית בלבד – לא `public` או `protected` ולא ניתן ליצור מופעים של enum מחוץ למחלקה עצמה.

לדוגמה: נכתוב enum שמכיר שני ערכים – חיבור וחיסור, ומבצע את פעולות החיבור והחיסור בהתאמה על שני מספרים:

```
public enum Operator {
   ADD {
      @Override
      public int operate(int x, int y) {
         return x+y;
      }
   },

   SUBTRACTION {
      @Override
      public int operate(int x, int y) {
         return x-y;
      }
   };

   public abstract int operate(int x, int y);
}
```

ונקרא למתודת החישוב שהגדרנו בצורה הבאה:

```
Operator.ADD.operate(5,6); // 11
```

שיפור אפשרי נוסף הוא להשתמש במתודת lambda אנונימית:

```
public enum Operator {

      ADD((x,y) -> x+y),
      SUBTRACTION((x,y) -> x-y);

      BinaryOperator<Integer> operator;

      Operator(BinaryOperator<Integer> op){
         this.operator = op;
      }

      public int operate(int x, int y){
         return this.operator.apply(x,y);
      }
   }
```

דוגמה נוספת דומה מאוד – enum של להקות שמקבל זמרים לפי עוצמת הקול שלהם וסופר כמה זמרים יש בכל מחלקה:

```
public enum Band {

   BEATLES((singer) -> singer.getVolume() > 5),
   DOORS((singer) -> singer.getVolume() < 3);

   int memberCount;
   Predicate<Singer> acceptor;

   Band(Predicate<Singer> acceptor){
      this.acceptor = acceptor;
      this.memberCount = 0;
   }

   public boolean accept(Singer singer){
      boolean gotIn = this.acceptor.test(singer);

      if (gotIn){
         this.memberCount++;
      }

      return gotIn;
   }
}
```

כאשר השתמשנו פה בממשק הפונקציונלי המובנה `Predicate` שמממש את המתודה `test` ומחזירה ערך בוליאני.

ביטויים רגולריים

ביטוי רגולרי הוא תבנית שניתן להשוות אותה מול טקסט (מחרוזת) ולבדוק האם הטקסט תואם לביטוי.

סינטקס בסיסי

סינטקס	שימוש	דוגמה
abc	חיפוש ליטרלי של הטקסט המדויק	abc מתאים ל-abc
.	מחליף כל תו בודד	at. מתאים ל-cat, bat, rat, 1at...
[...]	כל אחד מהתווים הבודדים בקבוצה הנתונה	[cbr]at מתאים ל-cat, bat, rat...
[...^]	שלילה של הקודם – כל תו בודד שאינו ברשימת התווים הנתונה	[^bc]at מתאים ל-rat, sat... אבל לא ל-bat, cat
[a-z] [A-Z] [0-9]	מתאים לכל תו בטווח הנתון (אותיות גדולות, אותיות קטנות, מספרים)	[f-l]aaa מתאים ל-faaa, gaaa עד laaa [^a-f]aaa מתאים ל- gaaa, 5aaa וכן הלאה, אבל לא ל-caaa וכו'
*	אפס או יותר הופעות של התו שמופיע קודם	at*. מתאים לכל מה שמסתיים ב-at, למשל hat, at%%_123 ועוד. <*[<^]> יתאים לכל ביטוי מהצורה <...>
+	הופעה אחת או יותר של התו שמופיע קודם	0+123 מתאים לכל הביטויים מהצורה 00123, 000000123 וכו'.
|	איווי - מספר אפשרויות (החיפוש יתבצע לפי הסדר בו הן מופיעות)	abc|xyz מחפש את abc או xyz
()	תחימה של ביטויים	(a|b)c יחפש את a או b ואחריהם c

לדוגמה: הביטוי [A-Za-Z]+[0-9] יתאים לאות אחת (גדולה או קטנה) או יותר ולאחריה ספרה אחת.

קיצורים

סינטקס	שימוש	משמעות
\d	ספרה	[0-9]
\D	כל תו שאינו ספרה	[^0-9]
\s	תו רווח (רווח לבן, טאב, ירידת שורה...)	[/t/n/v/f/r]
\S	כל מה שאינו תו רווח	[^\s]
\w	אות: אות גדולה, אות קטנה, ספרה או קו תחתון	[a-zA-Z_0-9]
\W	כל מה שאינו אות	[^\w]

מסמני גבולות

אפשר גם להגביל את החיפוש למקום מסוים:

סינטקס	משמעות
^	חיפוש בתחילת שורה
$	חיפוש בסוף שורה
\b	חיפוש במסגרת של מילה – מיקום שכולל אות ולפניה לא-אות, או להפך
\B	כל מה שאינו תו רווח

לדוגמה: \brabbit\b ימצא את המילה rabbit במשפטים כמו I saw a rabbit, rabbit rabbit ועוד.

כמתים

סינטקס	משמעות
X{n}	חיפוש של X בדיוק n פעמים
X{n,}	חיפוש של X המופיע לפחות n פעמים
X{,n}	חיפוש של X המופיע לכל היותר n פעמים
X{n,m}	חיפוש של X לפחות n ולא יותר מ-m פעמים
X?	X אופציונלי – יכול להופיע פעם אחת או בכלל לא
X*	X מופיע אפס או יותר פעמים
X+	X מופיע פעם אחת או יותר

סוגי חיפושים

קיימות מספר טכניקות לחיפוש:

- **חיפוש חמדני (greedy):** לחפש כמה שיותר; חיפוש שימשיך להתקדם עד שלא ימצא כבר את מה שצריך, ואז יחזור אחורה. זו ברירת המחדל.
- **חיפוש חסכן (reluctant):** לחפש כמה שפחות. כדי להפעיל חיפוש כזה צריך להוסיף כמת של סימן שאלה ? אחרי כל אחד מהכמתים.
- **חיפוש רכושני (possessive):** לחפש כמה שיותר אבל בלי לחזור אחורה. כדי להפעיל חיפוש כזה צריך להוסיף כמת של פלוס + אחרי כל אחד מהכמתים. שימושי במקרים שבהם אין טעם לחזור אחורה (למשל כשמחפשים תווים משתי קבוצות זרות שאין ביניהן חפיפה).

נבהיר את ההבדל בין סוגי החיפושים עבור הקלט xgaxxxxga:

1. חיפוש חמדני בעזרת הביטוי הרגולרי ga*.
בגלל שהכמת *. תופס את כל התווים החיפוש יתקדם עד הסוף, ועכשיו כדי לחפש ga הוא יתחיל לחזור אחורה ויתפוס את ההופעה של ga בסוף הקלט, כך שהתוצאה תהיה הקלט כולו xgaxxxxga.
2. חיפוש מינימליסטי בעזרת הביטוי הרגולרי ga?*.
נחפש כמה שפחות; בחיפוש הראשון של *. נמצא את המחרוזת הריקה ואז נחפש את ga, אבל לא נמצא; נתקדם צעד אחד ונחפש שוב ga והפעם כן נמצא, ולכן התוצאה תהיה xga. אם נפעיל את זה בצורה סדרתית שוב נקבל את xxxxga.
3. חיפוש רכושני בעזרת הביטוי הרגולרי ga+*
כמו בחיפוש החמדני, בגלל שהכמת *. תופס את כל התווים החיפוש יתקדם עד הסוף, והפעם לא נחזור אחורה ובגלל שלא מצאנו שם ga התוצאה תהיה שלילית.

Capturing Groups

אפשר להשתמש בסוגריים כדי "לזכור" תו שמצאנו ואחר כך למצוא את אותו תו שוב. לדוגמה: הביטוי rabbit\1(\d+) יחפש את המילה rabbit בין שני מספרים זהים. המספר שמופיע אחרי \ בסוף הביטוי ממספר את סדר פתיחת הסוגריים, למקרה שנרצה "לרפרנס" למספר דברים – לנסות לחפש שוב את מה שמצאנו.

עם זאת, נציין שמדובר במנגנון יקר ונעדיף להימנע ממנו במידת האפשר. אפשר גם לציין במפורש שלא נרצה לשמור תוצאת סוגריים לשימוש עתידי בעזרת האופרטור :? למשל: (?:ab|cd)

- רווח מייצג תו של רווח.
- לצמצם כמה שיותר את מרחב החיפוש
- באופרטור | לכתוב את האפשרויות לפי הצפי להופעתן (כדי לחסוך חיפושים מיותרים) וגם להוציא דברים מהסוגריים אם אפשר
- להשתמש בכמת הרכושני איפה שאפשר, זה חוסך הרבה

ביטויים רגולריים – הלכה למעשה

בשפת Java משתמשים בחבילה `java.util.regex.*`.
כדי ליצור פטרן צריך לקרוא למתודה הסטטית `compile` (ולא לקרוא לבנאי):

```
import java.util.regex.*;

Pattern patt = Pattern.compile("[a-z]+");
```

לאחר מכן צריך ליצור אובייקט של המחקה `Matcher` שמחזיק את הטקסט:

```
Matcher matcher = patt.macher("Now is the time");
```

כך, האובייקט `Matcher` מכיל מידע שכולל גם את הפטרן וגם את הטקסט שעליו אנחנו מריצים את הפטרן. מאובייקט פטרן אחד אפשר ליצור הרבה אובייקטים של `Matcher`.

כדי לדעת האם הטקסט מתאים לפטרן קיימות מספר מתודות שניתן להפעיל על `Matcher`:

- המתודה הבוליאנית `matcher.matches()` שבודקת האם הפטרן תואם **לכל** המחרוזת.
- המתודה הבוליאנית `matcher.lookingAt()` שבודקת האם הפטרן תואם **לחלק מהמחרוזת**, **החל מההתחלה**.
- המתודה הבוליאנית `matcher.find()` שבודקת האם הפטרן מתאים **לחלק** מהמחרוזת. בכל פעם שנקרא לה היא תחפש עוד מופע תואם עד שלא יהיו עוד.
- המתודה `matcher.start()` תחזיר את האינדקס של תחילת המופע, והמתדוה `matcher.end()` תחזיר את האינדקס של סוף המופע. כך אפשר לקבל את המוצא שמצאנו בדרך הבאה:

  ```
  str.substring(m.start(), m.end())
  ```

  אם ננסה לקרוא ל-`start` או `end` מבלי שהפטרן נמצא, תיזרק חריגה של זמן ריצה (`IllegalStateException`).

כדי לרפרנס לטקסט שחיפשנו בסוגריים (במסגרת capturing groups) אפשר להשתמש במתודה `group`, כאשר `m.group(0)` יחזיר לנו את כל ההתאמה, `m.group(1)` יחזיר לנו את מה שנמצא בקבוצה הראשונה שחיפשנו, וכן הלאה. המיספור של הקבוצות הוא לפי סדר פתיחת הסוגריים.

לדוגמה:

```
String patternString = "[a-z]+";
String text = "Now is the time";

Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(text);

while (matcher.find()){
   System.out.print(text.substring(matcher.start(), matcher.end()) +
"_");
}
```

תוצאת ההדפסה תהיה: `ow_is_the_time`.

סימנים מיוחדים

ישנם תווים מיוחדים שיש להם משמעות בסינטקס של שפת Java שמתנגשת עם הסינטקס של ביטויים רגולרים שהגדרנו עד כה; למשל, ראינו שהביטוי הרגולרי \b מסמל חיפוש בגבול של מילה, אבל בסינטקס של Java הוא מסמל תו מחיקה (backspace). הכלל שנקבע הוא שהסינטקס של Java קודם לסינטקס של הביטויים הרגולריים; כלומר, הביטוי הרגולרי"\b[a-z]+\b" יחפש מילה עם תווי מחיקה, ואם נרצה להשתמש בביטוי הרגולרי שמסמל גבולות של מילה נוסף לוכסן נוסף: "\\b[a-z]+\\b".

אותו דבר תקף גם לגבי תווים אחרים שיש להם משמעות כביטוי רגולרי: הסימן + משמש בתור כמת בביטויים רגולריים, ולכן אם נרצה לחפש אותו בתור סימן במחרוזת טקסטואלית נוסיף לפניו שני לוכסנים: +\\. ככלל, אם רוצים לכתוב ברג'קס סלש אחד צריך לכתוב בג'אוה שני סלשים.

מתודות של המחלקה String

קיימות מספר מתודות שימושיות במחלקה `String` שניתן להשתמש בהן בעבודה עם ביטויים רגולריים:

- `Matches(String toMatch)` בודקת האם ביטוי רגולרי מסוים מתאים למחרוזת שהתקבלה כפרמטר `toMatch`.
- `split(String rexex)` מקבלת ביטוי רגולרי ומחלקת את המחרוזת לפי הביטוי; היא מחזירה מערך של תתי-מחרוזות שמופרדות לפי הביטוי הרגולרי.
- `replaceAll(String rexex, String replacement)` מקבלת ביטוי רגולרי ומחליפה כל מופע של הביטוי הרגולרי במחרוזת השנייה.

בשביל להשתמש במתודות הנ"ל של המחלקה `String` לא צריך ליצור `Pattern` ו-`Matcher`, אבל מבחינת יעילות יותר טוב לקרוא פעם אחת בלבד ל-`Pattern.compile()` ולהשתמש בזה כמה פעמים, במקום לקרוא כל פעם ל-`String.Matches` שחוזר על התהליך בכל פעם מחדש.

תכנות מונחה עצמים

תכנות מונחה עצמים הוא פרדיגמה תכנותית שבה התוכנה מוגדרת בתור מכלול של אינטרקציות ופעולות בין אובייקטים. פרדיגמה תכנותית זו מאפשרת לבנות מערכות תוכנה גדולות המכילות רכיבים רבים שתלויים אחד בשני ונתונים לשינוי תכוף. תכנות מונחה עצמים מאפשר לכתוב קוד שקל להבין, קל לעשות בו שימוש חוזר וקל לעדכן ולשנות אותו.

עצמים (אובייקטים) ומחלקות

עצם (אובייקט) בתוכנה הוא הוא ישות שמאפשרת לשמור מידע (data members – **שדות/ממברים**) ולבצע פעולות - **מתודות** (methods).

מחלקות הן יחידות תוכנה שמאפשרות לנו להגדיר קבוצות של אובייקטים שלכולם אותם מצבים פנימיים (אותם data members) ואותה התנהגות (אותן מתודות).

אובייקטים של מחלקה נקראים **מופעים** (instances); כל מופע נותן ערכים ספציפיים לאותם שדות של המחלקה, ולכל מופע של אותה המחלקה יש את אותו הסט של מתודות.

אנו שואפים שלכל מחלקה יהיה תפקיד מוגדר אחד ויחיד (Single Responsibility Principle). דרך אחת לממש זאת היא בעזרת מחלקת "על" שתנהל מחלקות אחרות, שלכל אחת מהן יהיה תפקיד מוגדר יחיד.

מבנה של מחלקה

נתבונן על המימוש של המחלקה `Bicycle`. ניתן לראות את השדות של המחלקה ואת המתודות שלה:

```
public class Bicycle {

   /* Data members */
   int speed = 0;
   int gear = 1;
   String brand;

   /* Methods */
   public void changeGear(int newValue) {
      gear = newValue;
      return;
   }

}
```

כפי שראנו רואים, כדי להגדיר שדה של מחלקה עלינו לציין את הטיפוס שלו. בנוסף, בשורת ההגדה של מתודה צריך לציין את טיפוס ההחזרה שלה, את השם שלה ואת הארגומנטים שהיא מקבלת. במקרה דנן מדובר במתודה שטיפוס ההחזרה שלה הוא `void`, כלומר שאינה מחזירה דבר.

בנאי (קונסטרקטור)

כדי ליצור מופעים של מחלקה מסוימת עלינו להשתמש במתודה מיוחדת שנקראת בנאי (קונסטרקטור). בנאי מאפשר גם לתת ערכים התחלתיים לשדות של המופע או לקבל ערכים כארגומנטים ולאתחל איתם את השדות. לקונסטרקטור אין ערך החזרה והוא לא מחזיר דבר. בנאי נקרא בשם של המחלקה.

ניתן להגדיר למחקה מספר קונסטרקטורים לפי הנהלים שנציין בהמשך.

לדוגמה, בנאי למחלקת אופניים:

```
public class Bicycle {

   /* Constructor */
   public Bicycle(String myBrand, int newGear) {
      this.brand = myBrand;
      this.gear = newGear;
   }
}
```

במקרה דנן הבנאי מכניס את הפרמטרים שהוא קיבל לשדות של המחלקה.

כדי לקרוא לקונסטרקטור וליצור מופע חדש של המחלקה צריך להשתמש במילה השמורה new, ולאחר מכן אפשר להשתמש במתודות של המופעים שהגדרנו, לדוגמה:

```
public static void main(String [] args) {

      Bicycle bike1 = new Bicycle("BMX", 1);
      Bicycle bike2 = new Bicycle("newBike", 2);

      bike1.speedUp(10);
      bike1.changeGear(2);
      System.out.println(bike1.gear + "," + bike1.speed);
}
```

אם לא נכתוב מימוש מפורש של בנאי במחלקה יוגדר באופן אוטומטי קונסטרקטור דיפולטיבי שיאתחל את כל השדות לערכים הדיפולטיביים שלהם (אפס, מחרוזת ריקה, וכד'); אבל אם כן נכתוב מימוש לקונסטרקטור כלשהו לא נוכל להסתמך על קונסטרקטור דיפולטיבי מבלי לממש אותו בעצמנו.

משתנים ומתודות סטטיים

המילה static שניתן להוסיף לפני משתנה או מתודה מגדירה שהמשתנה או המתודה משתייכים למחלקה כולה ולא למופע ספציפי שלה .משתנים סטטיים נקראים גם "משתני מחלקה". לדוגמה:

```
public static int nDogs = 0;
```

בדוגמה הזו יצרנו משתנה סטטי שסופר את כמות הכלבים שיש; בכל פעם שיוצרים כלב חדש, מונה הכלבים יעלה ב-1. אפשר לגשת למשתנה סטטי דרך כל מופע של המחלקה, אבל הדבר יגרור אזהרת קומפילציית; במקום זאת, ניגש למשתנה סטטי דרך שם המחלקה:

```
Dog.nDogs += 1;
```

גם מתודות סטטיות לא פועלות על מופע ספציפי אלא על כלל המחלקה; לפיכך, הן לא יכולות לגשת לשדות של מופע אלא רק למשתנים סטטיים. תפקידן של מתודות סטטיות הוא לבצע פעולה על כלל המחלקה ולא על מופע ספציפי.

איתחול של משתנה סטטי לא נעשה עם כל מופע חדש שנוצר דרך פונקציית בנאי אלא קורה פעם אחת בלבד, באחד המקרים הבאים:

- כשמופע של המחלקה נוצר
- כשמתודה סטטית של המחלקה נקראת
- כשמתבצעת השמה לממבר הסטטי
- כשנעשה שימוש בממבר סטטי

קיים גם קונספט של **מחלקות של מתודות סטטיות**: קיימות מחלקות שמחזיקות אוסף של מתודות סטטיות בלי האפשרות או הצורך ליצור מופעים של המחלקה. לדוגמה- המחלקה המובנית Math של

אופרציות מתמטיות. מתודות אלו אינן קשורות למופע ספציפי אלא מבצעות חישוב על מספרים.

דריסת מתודות (Method Overloading)

ניתן להגדיר במחלקה מספר מתודות בעל אותו שם ("לדרוס מתודות") בתנאי שיש הבדל מובהק במימוש של המתודות. בכלל זאת **לא מספיק** שיהיה הבדל רק בטיפוס ההחזרה של הפונקציה (כיוון שאפשר לקרוא למתודות גם בלי לשמור את ערך ההחזרה, ואז הקומפיילר לא ידע לאיזה מתודה אנחנו קוראים), מימוש שונה או שמות שונים של ארגומנטים. הבדלים מספיקים בין מתודות שמאפשרים דריסה הם הבאים:

- מספר ארגומנטים
- טיפוסי הנתונים של הארגומנטים
- סדר שונה אל ארגומנטים (יחד עם זאת, דריסה של מתודות על-ידי שינוי סדר של ארגומנטים הוא לא הרגל טוב מכיוון שזה עלול לבלבל).

המילה this

מילה שמורה שמייצגת את המופע הנוכחי של המחלקה ודרכה ניתן לגשת לשדות של המחלקה בצורה קריאה וברורה יותר, לדוגמה:

```
this.real = num;
```

בשונה משפת Python, כאן מתודות לא צריכות לקבל את `this` כפרמטר בצורה מפורשת.
בעזרת המילה `this` אפשר לקרוא לקונסטרקטור אחר של המחלקה, לדוגמה:

```
public class ComplexNumber{

   private double real;
   private double img;

   public ComplexNumber(double real, double img){
      this.real = real;
      this.img = img;
   }

   public ComplexNumber(ComplexNumber other){
      this(other.real, other.img);
   }
}
```

בנוסף, בערת המילה `this` אפשר לשלוח את המופע הנוכחי בתור פרמטר למתודה כלשהי. נדגים מימוש של קופי-קונסטרקטור בעזרת המילה `this`:

```
public ComplexNumber (ComplexNumber other) {

   this(other.real, other.img);
}
```

המתודה `toString()`

מתודה שמחזירה ייצוג טקסטואלי של המופע הנוכחי:

```
public String toString() {

   return real + "+" + ims + "i";
}

ComplexNumber compNum = new ComplexNumber(5,2);
System.out.println(compNum.toString()); // 5+2i
System.out.println(compNum); // 5+2i
```

החבאת מידע

אנקפסולציה –בעברית: **כימוס**; לקחת קבוצה של רעיונות ולאגד אותם לכדי יחידה אחת.

API מינימלי

API (Application programming interface) הוא "שער הכניסה" לקוד שלנו וכולל את פרטי השימוש בקוד שלנו: המחלקות, המתודות, והקשרים ביניהן. אפילו תוכנות פשוטות עלולות להגיע למימדים גדולים ומסובכים, ולכן כשאנו נותנים קוד אנו שואפים לספק איתו כמה שפחות פרטים - **לספק API מינימלי**; כפועל יוצא מכך, רוב פרטי המימוש שלנו צריכים להיות חבויים, בעוד שנספק למשתמש רק את המידע המינימלי שהוא זקוק לו על-מנת להפעיל את התוכנה שלנו בצורה טובה. כל פרט מידע שאנו חושפים ב-API עלול ליצור תלות של המשתמש בו ולהקשות עלינו לשנות את הקוד. בכלל זאת, API מינימלי יכלול את החתימות של המתודות שהמשתמש צריך להכיר אבל לא את המימוש שלהן.

Modifiers

בדומה לשפות תכנות מונחה עצמים אחרות, גם ב-Java אפשר להגדיר כל שדה או מתודה בתור `public` או `private`:

ההגדרה של שדה או מתודה בתור `public` משמעה שהשדה או המתודה חשופים לכולם ואובייקטים מכל מחלקה יכולים לגשת אליהם. כשאנחנו בונים מחלקה נכניס ל-API רק את הרכיבים שמוגדרים בתור `public`.

ההגדרה של שדה או מתודה בתור `private` משמעה שהשדה או המחלקה חשופים רק לאובייקטים של המחלקה הנוכחית; אובייקטים מחוץ למחלקה לא יכולים להשתמש בהם, וניסיון לעשות זאת יגרור שגיאה. ככלל אצבע, שדות של מחלקה יוגדרו בתור `private` כיוון שהם חלק מהמימוש הפנימי של המחלקה ואיננו רוצים לאפשר גישה חיצונית אליהם. חריגים לכלל זה יכולים להיות משתנים סטטיים קבועים (לדוגמה - הקבוע π במחלקה Math).

במקום זאת, נאפשר גישה לשדות רק בצורה מסודרת דרך מתודות פומביות, getters ו-setters, המאפשרות גישה לשדות מבלי לחשוף את המימוש שלהם (או רק getter אם לא נרצה לאפשר שינוי). האפשרות לבצע גישה רק דרך getters ו-setters מאפשרת לנו לבצע בדיקה של הקלטים ושליטה על מה שנכנס או יוצא מהמחלקה, וגם פחות מגבילה אותנו לשינויים במימוש הפנימי.

יודגש כי משמעותו של `private` היא **לא** סודי, ולא ניתן לשמור מידע כמו סיסמאות בבטחה; `private` לא משמש כדי להחביא דברים ולשמור אותם בסוד, אלא כדי לכתוב תוכנה שמעוצבת בצורה טובה יותר.

היררכיה בין מחלקות

קיימים מספר סוגים של מערכות יחסים בין מחלקות. נציג בקצרה כמה יחסים מרכזיים:

- **הכלה** (Composition) – מייצגת יחס של "Has-a"; לדוגמה: לכל אדם יש שם, לאופניים יש גלגלים - לאובייקט מסוים יש משהו מאובייקט אחר. נממש יחס כזה באמצעות שימוש בשדות של מחלקה – כלומר שלמחלקה יהיה שדה מטיפוס של מחלקה אחרת.
- **ירושה** (Inheritance) – מייצגת יחס של יחס "Is-a". לדוגמה: סטודנט הוא בן אדם. לסטודנטים יש הרבה מן המשותף עם בני אדם והם חולקים את אותם התכונות והמאפיינים; מאידך, לסטודנט יש מאפיינים נוספים ולכן אפשר להגדיר עבורו תכונות ושדות משל עצמו. נממש יחס כזה באמצעות הקונספט של ירושה. אם מחלקה A יורשת ממחלקה B, נאמר כי A הוא sub-class של A, וכי B הוא super-class של A.
- **היחס "instance-of"**: נשים לב להבדל בין היחס "is-a" ליחס "instance-of"; לדוגמה, פלוטו הוא כלב ספציפי ולכן נממש את פלוטו כמופע של המחלקה כלב, ולא כמחלקה נפרדת שיורשת ממנה.

ירושה

נדגים מימוש של ירושה עם המחלקה `Student` שיורשת מהמחלקה `Person`:

```
public class Person {

   private String name;
   private Person mother;

   public String getName () {
      ...
   }
}

public class Student extends Person {

   private int id;

   public void takeExam() {
      ...
   }
}
```

כפי שניתן לראות, הירושה מתבצעת באמצעות המילה `extends` שנותנת למחלקה `Student` באופן אוטומטי את השדות והמתודות של `Person` ומוסיפה עליהם שדות ומתודות נוספים. מחלקה יורשת מקבלת את כל השדות של מחלקת האב, אבל לא יכולה לגשת לשדות הפרטיים של מחלקת האב. ניסיון לעשות כן יגרור שגיאת קומפילציה.

ירושה היא תכונה רקורסיבית וטרנזיטיבית: כל מחלקה יורשת את כל המחלקות שמחלקת האב שלה יורשת באופן אוטומטי וניתן להגדיר "שרשרת" של ירושות.

כל מחלקה יכולה להיות מחלקת אב של מספר בלתי-מוגבל של מחלקות, אבל כל מחלקה יכולה לרשת ממחלקה אחת בלבד.

ירושה מאפשרת להשתמש בפולימורפיזם ומסייעת במחזור קוד, אם כי זו בפני עצמה אינה סיבה להשתמש בירושה.

מחלקת העל Object

אם אנו לא משתמשים במילה extends ומציינים מפורשות מחלקת אב ממנה אנו יורשים, המחלקה שלנו יורשת אוטומטית מהמחלקה java.lang.Object; המחלקה היחידה שאינה יורשת ממנה היא מחלקת Object עצמה, שאין לה מחלקת אב. למחלקת העל Object תכונות שימושיות עבור מחלקות רבות בשפה, וכל אובייקט יורש ממנה את המתודות הבאות (אם כי המימוש הדיפולטיבי שלהן לא מועיל וכדאי לדרוס אותן עם מימוש שמתאים למחלקה):

- toString() שמחזירה ייצוג טקסטואלי של המופע.
- equals(Object obj) שמשווה בין המופע הנוכחי של המחלקה לבין האובייקט שמתקבל כפרמטר. אנחנו יכולים לקבוע את הקריטריונים להשוואה, אבל פעולת השוואה צריכה להיות רפלקסיבית (כל אובייקט שווה לעצמו), סימטרית וטרנזיטיבית, וגם לקיים שאף אובייקט אינו שווה ל-null. זה אולי המקום היחיד שבו כן נאפשר להשתמש ב-instanceof() , כדי לבדוק האם האובייקט שאנחנו מקבלים הוא בכלל מהטיפוס של המחלקה שלנו.
- hashCode() שמחזירה ערך גיבוב עבור המופע הנוכחי, שהוא מספר שלם (רנדומי או לא). אם יש שני איברים שווים (לפי equals), ערך ההאש שיוחזר עבורם צריך להיות זהה. החישוב של ערך ההאש שמוחזר צריך להתבסס על שדות קבועים (final) בלבד, אחרת נוכל להגיע למצב שערך ההאש של אובייקט משתנה במהלך הריצה שלו.

protected

בדומה ל-public או private, ניתן להגדיר מתודות, שדות וקונסטרקטורים גם באמצעות המודיפייר protected שמאפשר למחלקת יורשות לקבל גישה ישירה למשתנים שאותם הם יורשים, בעוד שכל שאר העולם ימשיך להיות ממודר מהם; לדוגמה:

```
public class Person {

   protected String name;
   protected Person mother;

   public String getName () {
      ...
   }
}
```

למעשה, יש למודיפייר protected חסרונות ואנו נימנע מלהשתמש בו; הסיבה לכך היא ששדה או מתודה שמוגדרים בתור protected יופיעו ב-API ויהיו חשופים כלפי חוץ. ככלל, תמיד נעדיף להגדיר שדות ומתודות בתור private היכן שנוכל ולהשתמש במתודות setters ו-getters במקום לתת גישה ישירה.

Overriding

הקונספט של Overriding משמעותו לרשת ממחלקה ולשנות את ההתנהגות שלה – כלומר לקחת מתודה שקיים מימוש שלה אצל מחלקת האב ולכתוב מימוש חדש שלה במחלקה היורשת. ברגע שאנו קוראים למתודה הזו ממופע של המחלקה היורשת נקבל קריאה לקוד החדש במקום לקוד הישן.

לדוגמה, נראה את המחלקה Child שיורדת מהמחלקה Parent ודורסת את המתודה foo עם מימוש אחר מהמימוש הקיים במחלקת האב:

```
public class Parent {

   public void foo() {
      System.out.println("P");
   }
}
```

```
public class Child extends Parent {

   public void foo() {
      System.out.println("C");
   }
}

Parent p = new Parent();
p.foo(); // P

Child c = new Child();
c.foo(); // C
```

אם נבחר לדרוס מתודה, ה-modifier שלה לא יכול להיות פרטי יותר מאשר ה-modifier של המימוש אותו אנו דורסים; בכלל זאת, לא ניתן (וגם אין סיבה הגיונית) לדרוס מתודה פרטית.

ניתן לגשת ממתודה שאנו דורסים למתודה של מחלקת האב בעזרת המילה `super` שמאפשרת לנו לעשות שימוש במתודה הקודמת – למשל במקרים שבהם נרצה להוסיף על הפונקציונליות של המימוש במחלקת האב ולא לדרוס אותה לחלוטין.

אפשר להשתמש במילה `super` גם כדי לגשת מהבנאי של המחלקה היורשת לבנאי של מחלקת האב. בהקשר זה נציין כי כדי ליצור תת-מחלקה חייבת להיות קריאה לבנאי של מחלקת האב בשורה הראשונה של הבנאי במחלקה היורשת; אם לא נבצע קריאה מפורשת ל-`super` תתבצע קריאה אוטומטית לבנאי הדיפולטיבי (ללא פרמטרים) - ואם לא קיים כזה תהיה שגיאת קומפילציה.

כך לדוגמה, בהמשך לדוגמת הקוד הקודמת נתאר מימוש של מתודות במחלקה היורשת `Child` בעזרת המימוש הקיים במחלקה המקורית `Parent`:

```
public class Child extends Parent {

   public Student (String name, int id) {
      super(name);
      this.id = id;
   }

   public String getNanme() {
      return super.getName() + "" +super.getName();
   }

}
```

פולימורפיזם

פולימורפיזם הוא עיקרון בסיסי וחשוב בתכנות מונחה עצמים שמאפשר רב-צורתיות וגמישות בקוד – ובפרט לטפל בערכים מטיפוסים שונים בעזרת ממשק אחיד. לדוגמה, נסתכל על ההמחשה הבאה של פולימורפיזם בגן החיות על המחלקות `Cow` ו-`Dog` היורשות מהמחלקה `Animal`:

```
Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = myCow;

myAnimal.speak(); // moo
myCow.speak(); // moo
myAnimal.getMilk(); // compilation error
```

פרה היא חיה ולכן ניתן לייצג אותה על-ידי רפרנס מסוג `Animal` ולקרוא דרך הרפרנס למתודות שמוגדרות על-ידי הרפרנס, אבל המתודה שתיקרא היא המתודה שמוגדרת על-ידי ה-content, כלומר המימוש של הפרה. לצד זאת, רפרנס של חיה לא יכול לקרוא למתודה שמוגדרת רק אצל

פרה. כלומר: מותר לקרוא רק למתודה שמוכרת בטיפוס של הרפרנס, אבל המתודה שתיקרא היא המתודה המוגדרת על-ידי הסוג האמיתי של האובייקט; ובמילים אחרות: reference-type קובע מה מותר להריץ, ו-object-type קובע מה ירוץ.

עם זאת, במקרה של מתודות ושדות סטטיים, רק הרפרנס קובע איזו מתודה תרוץ (כיוון שמתודות ושדות סטטיים אינם מקושרים לאובייקט ספציפי אלא לכלל המחלקה). ככלל, אנחנו לא אוהבים לקרוא למתודות וממברים סטטיים דרך אובייקט ספציפי אלא דרך שם המחלקה. לכן נימנע מדריסה של שדות או מתודות שאינם פרטיים (מה שנקרא Shadowing) כיוון שזה עלול לבלבל.

שימוש בפולימורפיזם מאפשר להרחיב את התוכנה בקלות רבה יותר ומבלי לשנות את הקוד הקיים, לאפשר שינוי של התנהגות בזמן ריצה, וגם מחזק את האנקפסולציה ומאפשר לחשוף פחות פרטים על המימוש.

מחלקה אבסטרקטית

לעיתים לא נרצה לאפשר ליצור מופעים של מחלקת האב, אלא רק של המחלקות היורשות ממנה; לדוגמה - המחלקה Animal שלא נרצה לממש "סתם חיה" כללית אלא רק כלבים, לוויתנים וכדומה. במקרה זה נכנסת לתמונה מחלקה אבסטרקטית (Abstract Class) שמהווה בסיס שממנו יצאו מחלקות יורשות אחרות.

במחלקות אבסטרקטיות ניתן, אבל לא חובה, להגדיר גם מתודות אבסטרקטיות, כלומר מתודות שאין להן מימוש ואנו מגדירים את החתימה שלהן בלי לתת להן תוכן. לדוגמה:

```
public abstract class Animal{

   public abstract void speak();
}
```

מחלקה אבסטרקטית "כופה" API על כל מי שיורש ממנה; אם מחלקה שאינה אבסטרקטית תירש מהמחלקה Animal ולא תממש את המתודה האבסטרקטית, נקבל שגיאת קומפילציה. מחלקה אבסטרקטית יכולה לרשת ממחלקה אבסטרקטית אחרת, ובמקרה כזה היא לא נדרשת לממש את המתודות האבסטרקטיות של מחלקת האב (למרות שהיא יכולה); לדוגמה: `Mammal` יורש מ-`Animal`.

מחלקות אבסטרקטיות יכולות להחזיק שדות, מתודות רגילות, בנאים - וכל מה שמחלקה רגילה יכולה להחזיק; ההבדל היחיד הוא שלא ניתן ליצור מופעים שלה ושניתן להגדיר גם מתודות אבסטרקטיות, ללא מימושן. מתודות סטטיות לא יכולות להיות אבסטרקטיות. ניתן להגדיר רפרנס מהטיפוס של מחלקה אבסטרקטית אבל לא ליצור מופע שלה.

אם ננסה לקרוא למתודות אבסטרקטיות נקבל שגיאת קומפילציה. לא ניתן להגדיר מתודות אבסטרקטיות כמתודות פרטיות, אלא רק `public` או `protected`.

ירושה לעומת קומפוזיציה

עד כה ראינו שני מנגנונים שמאפשרים להשתמש מחדש בקוד ולמנוע כפל קוד; אחד מהם הוא ירושה והשני הוא הכלה (קומפוזיציה). ככלל, אם המטרה היחידה שלנו היא למחזר קוד, נשתמש בהכלה; ואם גם קיימת היררכיה בין המחלקות, נשקול להשתמש בירושה. ירושה נקראת לעיתים "קופסה לבנה" (על משקל "קופסה שחורה") כיוון שאנו יכולים להיות חשופים למבנים הפנימיים של מחלקה ממנה אנו יורשים, בניגוד לקומפוזיציה שמזכירה יותר "קופסה שחורה" במובן שאנו מקבלים את הפונקציונליות על-ידי החזקת אובייקט מסוים ויכולים לקרוא למתודות שלו רק לפי ה-API, מבלי לדעת את המבנה הפנימי, השדות והמתודות הפרטיות שלו.

נסכם ביתרונות וחסרונות של מנגנון הירושה:

ירושה	
שימוש	כדאי להשתמש בירושה כאשר A הוא סוג של B.

יתרונות	חסרונות
חלק מהמנגנון של השפה - ברורה לשימוש.	שוברת את האנקפסולציה - אנחנו נחשפים לחלק מהמבנה הפנימי של המחלקה שאותה אנו יורשים (לכל המידע שמוגדר כ-`protected`)
מאפשרת פולימורפיזם - אפשר לקרוא למחלקה היורשת גם בשם של המחלקה המורישה.	המחלקה שיורשת מחויבת למימוש של מחלקת האב - ומקבלת את כל המתודות שלו
מוגדרת באופן סטטי בזמן הקומפילציה - לא יכול להשתנות בזמן ריצה.	מוגדרת באופן סטטי בזמן הקומפילציה - אי אפשר לשנות בזמן ריצה.
קל לשנות את המימוש של המחלקה על-ידי דריסה (overriding) של מתודות ו/או להשתמש ב-super.	אפשר לרשת רק ממחלקה אחת

לעומת יתרונות וחסרונות של שימוש בקומפוזיציה:

הכלה (קומפוזיציה)	
שימוש	נשתמש בקומפוזציה כשהמטרה העיקרית שלנו תהיה למחזר קוד; או כשיש שני מועמדים לירושה, במקום אחד המקרים אפשר להשתמש בהכלה.

יתרונות	חסרונות
לא שוברים את האנקפסולציה - סוג של "קופסה שחורה", לא חושף את המימוש.	אין תמיכה בפולימורפיזם.
הרבה פחות תלויות בין האובייקטים השונים – האובייקט שאנו מכילים לא כופה עלינו דבר.	עיצוב שמבוסס על הכלה יהיה מרובה מחלקות.
מוגדר באופן דינאמי בזמן ריצה, ניתן לשנות מימוש בזמן ריצה בהתאם לנסיבות המשתנות.	בניגוד לירושה שהיא סטטית, מחלקה דינאמית תלויה בהחלטות שיכולות להיות שגויות - פוטנציאל לטעויות.
מאוד מתאים למחלקה שממוקדת לבצע תת-משימה – שומר את העיקרון שלכל מחלקה יש משימה אחת.	
מחלקה יכולה להחזיק כמה אובייקטים שהיא רוצה - אין מגבלה כמו בירושה למחלקה אחת.	

דרך ביניים שנציג כעת היא להשתמש בממשקים (התומכים בפולימורפיזם).

ממשקים

ממשק (Interface) מתאר יחס של "can" או "able-" בין מחלקות; באופן כללי ניתן להגיד שממשק מדבר על "מה עושים" במחלקה ולא על "איך עושים". ממשק יכול להכיל רק שני סוגים של נתונים: משתנים קבועים (`final static`) ומתודות אבסטרקטיות. בנוסף, ניתן להגדיר מימוש דיפולטיבי של מתודות בעזרת המילה `default`. בדומה למחלקות אבסטרקטיות לא ניתן ליצור מופע מטיפוס

של ממשק - אבל כן ניתן להחזיק רפרנס מטיפוס של ממשק, לממש ממשק או לרשת מממשק ע"י ממשקים אחרים. לדוגמה:

```
public interface Printable {

   public void print();

   public default void doSomething(){
          ...
   }

}
```

מה שהופך את המתודה לאבסטרקטית הוא העובדה שאין מימוש של המתודה, ואין צורך לכתוב את המילה `abstract`.

אפשר להסתכל על ממשק כאל סוג של "חוזה"; מחלקה שיורשת מממשק יכולה להרחיב אותו אבל חייבת לממש את כל מה שהוא דורש. ממשקים לא יכולים להגדיר מתודות שהן `private` או `protected` אלא רק `public`, שהרי הם מגדירים סוג של API ולכן המתודות צריכות להיות פומביות; הם גם לא יכולים להגדיר שדות, כי API כמעט תמיד לא יכלול את השדות. מימוש של ממשק ייעשה עם המילה `implements`.

דוגמאות לממשקים שניתן לממש הם `Printable` (אובייקט ניתן להדפסה), `Comparable` (אוביקט ניתן להשוואה), `Clonbale` (אובייקט שניתן לשכפול); כלומר, במקרה זה מחלקות שמממשות ממשק "מקבלות על עצמן אחריות" להדפיס את עצמן, לשכפל את עצמן וכדומה.

בשונה מירושה, כל מחלקה יכולה לממש מספר ממשקים, ובדומה לירושה ניתן גם להגדיר רפרנסים מהטיפוס של הממשק:

```
public class MyClass implements MyInterface1, MyInterface2, ... {
   ...
}


MyClass myObj = new MyClass();
Printable myPrintableObj = myObj;
Clonable myClonableObj = myObj;
```

אבל במקרה כזה נוכל לקרוא רק למתודות שמוכרות על-ידי אותו הרפרנס, ולא מתודות שמוגדרות ברמה של המחלקה שמממשת את הממשק.

ככלל, אם מחלקה היא סוג של מחלקה אחרת, כדאי להשתמש בירושה; ואם מה שמשותף לסוגים הוא שסוג א' הוא מעיין הסכם או API של סוג ב', נעדיף להשתמש ביחס של ממשק. היתרון בממשקים הוא שמחלקות יכולות לממש מספר ממשקים אבל לרשת רק ממחלקה אחת.

Casting

הפעולה שנמצאת בלב הקונספט של פולימורפיזם: להתייחס למחלקה מסוג אחד כאל אובייקט מסוג אחד. נשים לב לשני סוגים של casting שניתן לבצע:

```
Animal a = new Cow();
```

פעולה זו נקראת **up-casting** כי אנו עולים במעלה ההיררכיה המחלקתית: הסוג של הרפרנס (הצד השמאלי) הוא מחלקת אב או ממשק של האובייקט (הצד הימני). במקרה הזה עשינו המרה מבלי לציין זאת מפורשת, אבל אפשר לעשות את זה גם בצורה יותר מפורשת; לרוב הקומפיילר יוכל לזהות לבד מה צריך להיות הסוג של הרפרנס:

```
Animal a = (Animal) new Cow();
```

הסוג השני נקרא **down-casting**. המרה זו חייבת להיעשות בצורה מפורשת (אחרת תתקבל שגיאת קומפילציה) ועלולה לגרום גם לשגיאות בזמן ריצה, ולכן נחשבת פעולה בעייתית; בנוסף, המרה כלפי מטה פוגעת בגמישות מאחר שהיא לוקחת קוד שהיה כללי והופכת אותו לספציפי יותר.

המרה מסוג down-casting יכולה להצליח רק אם עשינו קודם לכן up-casting, אבל זה עדיין לא מבטיח שהיא תצליח. במידה והמחלקות בהיררכיה המתאימה לא נקבל שגיאת קומפילציה, אבל אם הטיפוס לא מתאים נקבל שגיאת זמן ריצה. לדוגמה:

```
Animal animal = new Cow();
Cow c = (Cow) animal;
Cow c = animal;

Animal animal = new Dog();
Cow c = (Cow) animal; // illegal - error
```

Instance-of

המתודה `instanceof()` מאפשרת לבדוק בזמן ריצה האם אובייקט הוא מטיפוס מסוים; לכאורה ניתן להשתמש בו כשעושים down-casting, אבל בפועל לא מדובר בפתרון טוב כיוון שישנן הרבה נקודות עדינות שאפשר ליפול בהן בקלות והוא לא גמיש.

Design Patterns

פטרנים עיצוביים באים לתת פיתרון להתמודד עם בעיות עיצוביות שחוזרות על עצמן בהקשרים שונים אך דומים אחד לשני. לפטרנים מספר מאפיינים:

- לכל פטרן עיצובי צריך להיות שם תמציתי, בעל משמעות - ולשמש כלי בשיח בין מתכנתים.
- צריך להגדיר מה הבעיה שאיתה הפטרן בא להתמודד - מה הצורך שעולה ומה התנאים שצריכים להתקיים כדי שנוכל להשתמש בפטרן העיצובי.
- הפיתרון עצמו - מה הרכיבים השונים של הפיתרון ומה הקשרים ביניהם; הפיתרון שיובא לא יהיה פיתרון קונקרטי (מימוש) אלא סכימה או תיאור מופשט של הפיתרון שאפשר לקחת ולממש אותו בהרבה הקשרים שונים.
- מה ההשלכות של הפטרן - לאיזה דברים צריך לשים לב, איך זה ישפיע על שאר המערכת ועל דברים כמו זמן ריצה.

קיימים שלושה סוגים של פטרנים:

1. Creational Patterns - עוסקים ביצירה של אובייקטים;
   לדוגמה: Factory, Singleton.
2. Structural Patterns - עוסקים בהרכבה של אובייקטים;
   לדוגמה: Delegation, façade, Decorator.
3. Behavioral Patterns - עוסקים בהתנהגות של אובייקטים ובקשר בין אובייקטים;
   לדוגמה: Iterator, Strategy.

Delegation

האצלה- הרכבה של מחלקות. לדוגמה, בדוגמת הקוד הבא, בכל פעם שמישהו קורא ל- `B.foo()`, כל מה שקורה בפועל הוא קריאה למתודה `A.foo()`. זוהי דרך לקחת את היתרונות של ירושה (לקבל קוד ממחלקה אחרת עם אותו API) ולהעביר אותם למבנה של קומפוזיציה.

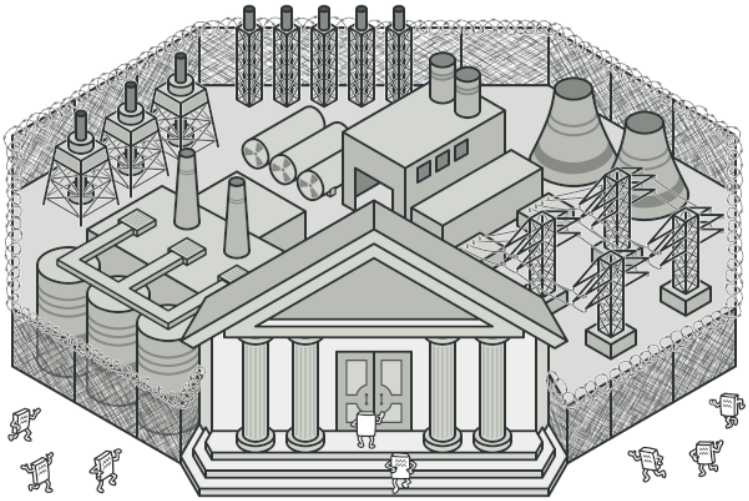
```
public class A{
   public void foo() {...}
}

public class B{
   private A a;

   public B(A a) {
      this.a = a;
   }

   public void foo() {
      this.a.foo();
   }
}
```

בזמן הריצה אפשר להחליט איזה אובייקט אנחנו מרכיבים ולקבל התנהגות קצת שונה.

סיבה נוספת להשתמש בהאצלה ולא בירושה היא שהיא מאפשרת למחלקה `B` לרשת ממחלקה אחרת, שאיננה המחלקה `A`; ובנוסף - זו תבנית כללית שטובה להרבה מקרים והקשרים שונים.

Façade

פטרן של מבנה, כלומר שעוסק בקומפוזיציה. פירוש המילה מצרפתית הוא "חזית" ומקורו בעולם הארכיטקטורה. המשמעות היא לקחת מערכת מורכבת ולהסתכל עליה דרך "חלון" צר ומופשט.



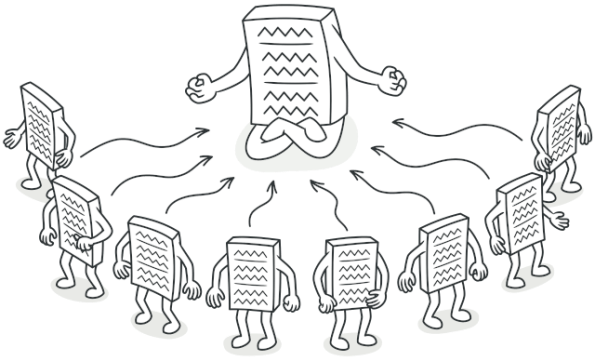
- **הבעיה:** מערכת גדולה עם הרבה מאוד מחלקות ותלויות בין מחלקות שונות שיוצר API מורכב שקשה לעבוד איתו. בהרבה מקרים לקוחות לא צריכים לדעת את כל המורכבות של המחלקה אלא רק חלק קטן ממנה.
- **הפיתרון:** לקחת את המערכת המורכבת ולבנות מחלקה חדשה בעלת API פשוט יותר שתספק דרך פשוטה לעבוד עם ה-API המורכב יותר - מישהו אחר יעשה את הפעולה המורכבת של להתמודד עם ה-API ה"קשה".
- **מימוש:** נתחיל מלבנות מחלקה חדשה, Façade, ונגדיר API פשוט מזה של המערכת הגדולה. נממש אותה על-ידי שימוש ב-API המורכב. מבנה זה לוקח רכיבים קיימים ומשתמש בהם (מנהל את הדברים) אבל לא מוסיף שום יכולת חדשה בזכות עצמו.

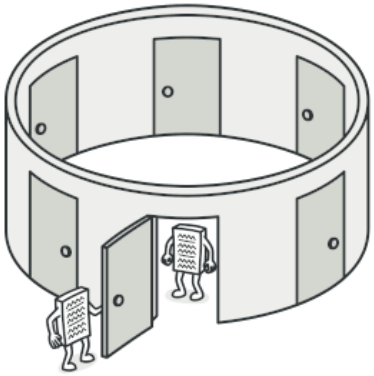
יתרונות:

- הלקוחות חשופים רק לממשק של ה-Façade ולא לממשק המלא, וזה מקל על העבודה שלהם, כי נדרש מהם לעבוד עם API מצומצם יותר.
- לקוחות לא צריכים להיות חשופים או מודעים לשינויים שנעשים במערכת המורכבת - לא צריכים ללמוד API חדש או לדעת שהיו שינויים.
- לא מאבדים פה שום דבר - אפשר להשתמש גם ב-API המקורי וגם במורכב.

Singleton

פטרן של יצירה שעוסק ביצירה של אובייקטים. עיצוב זה מתאים למקרים בהם אנחנו רוצים שיהיה לנו בדיוק מופע אחד ויחיד של המחלקה ולאפשר גישה נוחה לאותו האובייקט מבלי לאפשר ליצור מופעים נוספים.





- **הפיתרון:** להחזיק מופע סטטי פרטי של המחלקה. הקונסטרקטור היחיד של המחלקה יהיה פרטי ולכן לא יהיה ניתן ליצור אובייקטים חדשים מבחוץ; במקום זאת, תהיה מתודה סטטית שמחזירה את השדה הסטטי שהוגדר ויהווה נקודת הגישה לאובייקט היחיד שלנו. מי שעובד עם הקוד לעיתים אפילו לא יהיה מודע לכך שהוא עובד עם מחלקה שהיא Singleton.
- **השלכות:** ההגדרה של הבנאי בתור `private` לא מאפשרת לרשת מהמחלקה.

לדוגמה:

```
public class Singleton{
   private static final Singleton single = new Singleton();

   private Singleton() {}

   public static Singleton instance() {
      return single;
   }
}
```

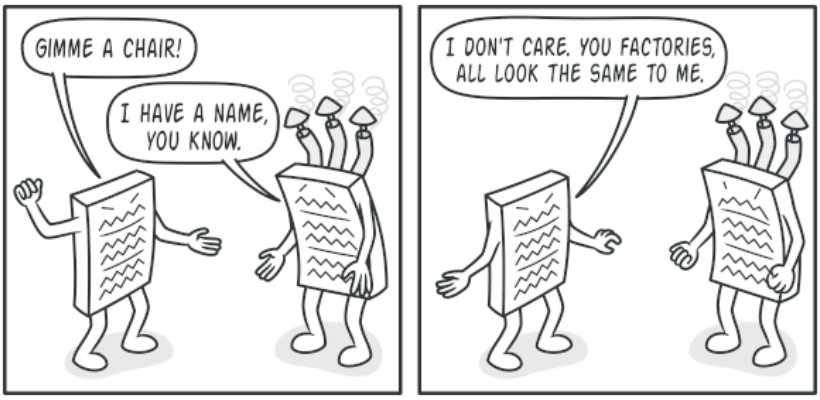
מימוש נוסף:

```
public static Singleton instance() {
   if (Singleton.single == null){
      Singleton.single = new Singleton();
   }
   return Singleton.single;
}


// in other class:
Singleton s = new Singleton(); // illegal - constructor is private
Singleton s2 = Singleton.instance();
Singleton s3 = Singleton.instance(); // s2 == s3
```

אלטרנטיבה למימוש המתואר של Singleton יכולה להיות מחלקה שכל השדות והמתודות שלה הן סטטיות וקונסטרקטור פרטי; החיסרון באלטרנטיבה המוצעת הוא שהמתודות של Singleton אינן סטטיות ולכן הם יכולים לממש ממשקים, להשתמש בפולימורפיזם וגם לתמוך יותר בהחבאת מידע.

Factory

פטרן של יצירה שעוסק ביצירה של אובייקטים. "מפעל" הוא מתודה או מחלקה שמשמשת ליצירת אובייקט אחרים (למשל לפי פרמטרים שהיא מקבלת). הרעיון המרכזי שעומד מאחורי השימוש בעיצוב זה הוא ליצור הפרדה בין הקוד שמייצר את האובייקטים לקוד שמשתמש בהם – ובכך ליצור הפרדה בין החלקים השונים של הקוד, לחזק את המודולריות ולאפשר להוסיף בקלות יותר סוגי מחלקות נוספים שנרצה ליצור (מה שמחזק את עיקרון ה-Open/Closed). מפעלים יכולים לייצר את האובייקטים בצורה דינאמית או לייצר מראש קבוצה של אובייקטים.



Strategy

פטרן של התנהגות.

- **הבעיה:** למערכת יש משפחה של אלגוריתמים או התנהגויות והיינו רוצים להפריד אותם מהמערכת עצמה כך שיהיה אפשר להחליף בין ההתנהגויות השונות.
- **הפיתרון:** להגדיר API (מחלקה אבסטרקטית או ממשק) שמגדיר מה האלגוריתם או ההתנהגות עושה; לכל התנהגות (אסטרטגיה) שונה תהיה מחלקה שתממש את אותו API. מחלקה אחרת תבחר את ההתנהגות הרצויה לכל מצב. בכך ניצור הפרדה בין ההתנהגויות השונות לבין הקוד שמריץ אותן ונאפשר להוסיף אסטרטגיות חדשות או לערוך שינויים באסטרטגיות הקיימות מבלי לשנות את כל הקוד. דרך אפשרית לבחור בין ההתנהגויות היא בעזרת מחלקת Factory.

לדוגמה – מחלקה שצריכה למיין מערך. תחילה נגדיר שני מימושים של מיון – `QuickSort` ו-`MergeSort` הממשים את הממשק `SortStrategy`:

```
public interface SortStrategy {
   void sort (Comparable[] data);
}

public class QuickSort implement SortStrategy{
   public void sort(Comparable[] data);
}

public class MergeSort implement SortStrategy{
   public void sort(Comparable[] data);
}
```

נגדיר מחלקת Factory שתבחר את האסטרטגיה המתאימה:

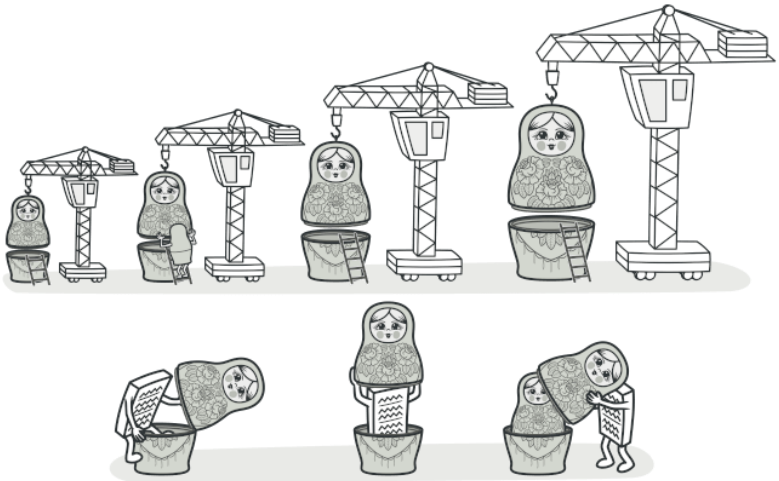
```
public class SortStrategyFactory{
   public static SortStrategy select(...) {...};
}
```

לבסוף, נוכל לקבל את אסטרטגיית המיון המתאימה ולמיין את המערך שלנו:

```
public class SomeCollection{

   private Comparable[] contents;
   private SortStrategy sorter;

   public SomeCollection() {
      this.sorter = SortStrategyFactory.select(...);
   }

   public void sortContents() {
      this.sorter.sort(this.contents);
   }
}
```

Decorator

פטרן מבני שמאפשר להוסיף יכולות נוספות למחלקות קיימות בזמן ריצה באמצעות מחלקה שעוטפת אותם, מבלי לשנות את הקוד הבסיסי.



- **מוטיבציה:** אוסף של הרבה סוגים של Streams ורוצים להוסיף להם יכולות נוספות (למשל – קריאה וכתיבה בצורה מכווצת ויעילה, קידוד והצפנה) בלי לעשות את זה לכל אחד מהם בנפרד באופן שיגרור כפל קוד נרחב.
- **הפיתרון:** לקחת מחלקה A שרוצים להרחיב וליצור מחלקה B שיורשת את המחלקה A (ולכן עובדת עם אותו API) וגם מחזיקה אובייקט של המחלקה A בתור שדה ומאצילה אליו את הפעולות, תוך שהיא עשויה להוסיף פעולות נוספות לפני או אחרי ההאצלה.

למחלקות העוטפות אין שימוש משל עצמן – הן מייצגות פונקציונליות ולא מקור מידע. אפשר גם להגדיר Decorators בצורה רקורסיבית –לקחת Decorator קיים ולעטוף אותו שוב עם עוד יכולות.

סיכום ביניים

- **קומפוזיציה**: A מחזיק מופע של B בתור שדה או משתנה מקומי
- **האצלה**: A מחזיק מופע של B ומעביר אליו בקשות.
- **דקורציה**: A יורש מ-B, מאציל פעולות ל-B ומוסיף פונקציונליות.

מחלקות מקוננות

לעיתים נרצה להכיל מחלקה אחת בתוך מחלקה אחרת; עד כה אמרנו שכל מחלקה נמצאת בקובץ נפרד java משלה, וכעת נראה שאפשר להגדיר מחלקה בתוך מחלקה אחרת. במקרה כזה המחלקה החיצונית תיקרא enclosing class או wrapping class. מחלקה פנימית צריכה להיות קטנה ובעלת מספר קטן של מתודות, כדי לא לפגום בקריאות של הקוד.

סיבות לשימוש במחלקות מקוננות

- **קיבוץ לוגי של מחלקות** – מחלקה שרלוונטית רק בהקשר של מחלקה אחרת ואין כל כך מקום להשתמש בה שלא בהקשר של המחלקה החיצונית.
- **להגדיל את האנקפסולציה** – אנחנו מאפשרים למחלקות מקוננות גישה למשתנים הפנימיים ששאר העולם לא יכול לגשת אליהם. בנוסף, בהרבה מקרים המחלקה הפנימית תהיה בעצמה פרטית, כך שאפילו עצם הקיום שלה לא יהיה חשוף לשאר העולם מלבד למחלקה החיצונית שלה. על כן, ברוב המוחלט של המקרים נעדיף להגדיר מחלקות פנימיות כפרטיות. חרף זאת, נציין כי מחלקה פנימית יכולה להיות מוגדרת בתור כל אחד מארבעת המצבים, בניגוד למחלקות חיצוניות שיכולות להיות רק `public` או דיפולטיביות (הרשאות חבילה).
- **קוד שמסביר את עצמו** – מי שרואה את הקוד מבין שהקיום של המחלקה הפנימית הוא אך ורק בהקשר של המחלקה החיצונית.

למחלקה פנימית יש גישה לכל המשתנים והמתודות של המחלקה העוטפת וגם להפך, אפילו אם הם מוגדרים בתור `private`.

פעמים רבות נרצה לא לשים חריגות במחלקות מקוננות כיוון שחריגות צריכות להופיע ב-API של מחלקה, בעוד שנרצה להשאיר לרוב מחלקות מקוננות כפרטיות.

מחלקה פנימית סטטית

מחלקה מקוננת סטטית היא מחלקה שקיימת ללא קשר למופע של המחלקה העוטפת, כאילו שהייתה מוגדרת בקובץ משלה; ניתן ליצור מופע של המחלקה הפנימית הסטטית גם מבלי ליצור מופע של המחלקה העוטפת שלה, וההיררכיה נועדה בעיקר לצרכי סידור ואירגון של הקוד.

מחלקה פנימית סטטית יכולה לגשת לשדות פרטיים של המחלקה העוטפת, אבל מכיוון שהיא סטטית היא לא יכולה לגשת לשדה שאינו משויך למופע ספציפי.

בדוגמת הקוד הבא המחלקה העוטפת `EnclosingClass` יוצרת מופע של המחלקה הפנימית הסטטית `NestetClass` וניגשת לשדה פרטי שלה, וגם המחלקה הפנימית הסטטית יוצרת מופע של המחלקה העוטפת וניגשת לשדה פרטי:

```
public class EnclosingClass {

   private int dataMember = 7;

   public void createAndIncrease() {
      NestedClass in = new NestedClass();
      in.nestedDataMember++; // a private field of the nested class
   }

   private static class NestetClass{
      private int nestedDataMember = 8;

      private void nestedCreatedAndIncrease() {
         EnclosingClass en = new EnclosingClass();
         en.dataMember++; // a private field of the enclosing class
      }
   }
```

מחלקה פנימית לא-סטטית – Inner Class

מחלקה מקוננת לא-סטטית (Inner-class) היא מחלקה שמקושרת למופע ספציפי של המחלקה העוטפת ולכן יכולה לגשת לממברים של אותו מופע ספציפי. כיוון שמחלקה כזו מקושרת למופע ספציפי היא לא יכולה להגדיר משתנים סטטיים ולא יכולה להתקיים בלי מופע של המחלקה העוטפת. בניגוד למחלקה פנימית סטטית, מחלקה פנימית (שאינה סטטית) יכולה להתקיים רק בהקשר של מופע של המחלקה החיצונית.

הסוג הנפוץ ביותר של מחלקה כזו נקרא Member Class. לדוגמה:

```
public class EnclosingClass {
   private int dataMember = 7;

   public void createAndIncrease() {
      MemberClass mem = new MemberClass();
      mem.memberClassField++;
   }

   private class MemberClass{
      private int nestedDataMember = 8;
      private void memberClassIncrease() {
        dataMember++;
   }
}
```

כפי שאנו רואים, המחלקה הפנימית `MemberClass` יכולה לגשת לשדה של המחלקה החיצונית שלא דרך מופע ספציפי מכיוון שהמחלקה הפנימית כולה מקושרת למופע ספציפי של המחלקה החיצונית; בנוסף, שוב אנחנו רואים שלשתי המחלקות יש גישה לשדות הפרטיים אחת של השנייה.

מחלקה לוקאלית

מחלקה לוקאלית היא סוג של אינר-קלאס - מחלקה מקוננת (לא סטטית) שמוגדרת בתוך מתודה ותהיה נגישה רק בתוך אותה המתודה. נשתמש במחלקה לוקאלית במקרים בו תהיה רלוונטית רק בקונטקסט של אותה המתודה. לפיכך, מטבע הדברים אין משמעות ל-modifier שבו נגדיר אותה. מחלקה לוקאלית איננה סטטית ולכן מקושרת למופע ספציפי של המחלקה העוטפת ויכולה לגשת לממברים פרטיים שלו, ובנוסף לגשת למשתנים לוקאליים של המתודה שעוטפת אותם – בתנאי שהם מוגדרים בתור `final` (גישה למשתנה לוקאלי שאיננו `final` תגרור שגיאת קומפילציה). מצד שני, בדומה למחלקות לא-סטטיות אחרות, מחלקה לוקאלית לא יכולה להכיל משתנים סטטיים. לא ניתן להגדיר ממשקים כלוקאליים.

נציג דוגמה מסקרנת למחלקה לוקאלית:

```
public class StringLinkedList{
   private Node head;

   public Iterator<String> elements() {

      class ListIterator implements Iterator<String> {
         Node current;
         ListIterator() {...}
         public boolean hasNext() {...}
         public String next() {...}
      } // end of local class ListIterator

      return new ListIterator();
   }
}
```

בדוגמה לעיל בתוך המתודה `elements()` קיימת הגדרה של מחלקת `ListIterator` שמממשת את הממשק של `Iterator`. המושג `ListIterator` מוכר רק בתוך המתודה `elements()` ורק היא יכולה להשתמש בו; למרות זאת, המתודה יכולה להחזיר את האובייקט של המחלקה הלוקאלית דרך ה-`API` של המחלקה `Iterator<String>` מכיוון שכלפי חוץ היא מתנהגת כמו `Iterator<String>` והמימוש האמיתי לא חשוף כלפי העולם.

מחלקה אנונימית

מחלקה אנונימית היא מחלקה חדשה הנוצרת באופן ספציפי למקרה מאוד מסוים בקוד – ולכן יהיה בדיוק מופע אחד ויחיד שלה שיווצר. ניתן להגדיר מחלקה אנונימית בתוך כל מקום שבו מגדירים אובייקט, לרבות בקריאה למתודה. זה שימושי למשל כשנרצה לדרוס מתודה אחת בלבד מבלי ליצור מחלקה שלמה חדשה.

בעזרת הסינטקס של מחלקה אנונימית ניתן גם לממש ממשק או לרשת ממחלקה אבסטרקטית. לדוגמה:

```
String[] filelist = dir.list(
      new FilenameFilter() {
       // creating an instance while implementing the class
         public boolean accept(File dir, String s) {
            return s.endsWith(".java");
         }
      } // end of class declaration
      ); // end of statement of calling dir.list
```

בדוגמה לעיל כותבים מימוש של הממשק `FilenameFile`; אנחנו לא יוצרים מופע של הממשק, אלא מחלקה אנונימית חדשה שמממשת אותו.

מודולריות

מודולריות: לקחת את התכונה ולחלק אותה ליחידות קטנות ובלתי תלויות אחת בשנייה – מודולים. קל יותר לתחזק ולהרחיב איפיון מודולרי וגם לחלק את התכנית למספר צוותי עבודה שונים.

עקרונות של תכנון מודולרי

- **פריקות** (Decomposability): לקחת בעיה ולפרק אותה לתתי-בעיות יותר ויותר קטנות; באמצעות מבנה פשוט ניתן לחבר את תתי-הבעיות בחזרה לבעיה המקורית (וכך ניתן לחלק את העבודה על הבעיה בין צוותים שונים).
- **הרכבה** (Composability): יצירת רכיבים שניתן לקחת ולשלב אותם לכדי יחידות מורכבות יותר. כל רכיב בפני עצמו צריך להיות אוטונומי ובלתי-תלוי ביחידות האחרות. למרות שיש ניגוד מסוים בין פריקות לבין הרכבה, לעיתים ניתן לשלב בין שני הפתרונות האלה.
- **מובנות** (Understandability): קורא אנושי יוכל להבין את העיצוב והמבנה של כל מודול בנפרד מבלי להבין את המודולים האחרים; כלומר - כל מודול יהיה אוטונומי מספיק ולא יצריך היכרות כללית עם שאר הסביבה – ובפרט יהיה אפשר לתאר כל מודול בכמה מילים.
- **רציפות** (Continuity): כל אחד מהמודולים יהיה יחסית לוקאלי כך ששינוי בדרישות על מודול אחד לא ישליך יותר מידי על המודולים האחרים: "שינוי קטן משפיע רק בסביבה קטנה".
- **עיקרון פתוח-סגור** (Open-Closed principle): רכיבי תוכנה צריכים להיות סגורים לשינויים אבל פתוחים להרחבה; שינוי בתוכנה הרבה פעמים גורם לשינוי בחלקים רבים של התוכנה, וזה לא טוב. בכלל זאת, שינוי בדרישות לא יגרור שינוי של הקוד הקיים אלא הוספה של קוד חדש.
- **עיקרון** (Choice principle Single): אם התכנית כוללת רשימה של אפשרויות או אלטרנטיביות, רק מקום אחד בתכנית ידע אותה. מכאן, אם אין ברירה אלא לשנות את הקוד הקיים, נעשה את זה במקום אחד ויחיד שנוכל לבודד אותו ולבדוק אותו, בעוד ששאר המודולים לא יצטרכו להשתנות ויישארו סגורים לשינויים.

לדוגמה: אנו כותבים אלגוריתם שינסה למצוא חוקיות בכותרים של סרטים מצליחים. נחלק את הבעיה לחלקים קטנים: להשיג את המידע, לעבד את המידע ולשמור את התוצאות; כל אחד מהנ"ל צריך להוות יחידה נפרדת ובלתי-תלויה בחלקים האחרים.

שיכפול של אובייקטים

לעיתים נרצה לשמור או להעתיק לזיכרון מצב פנימי של אובייקט לצורך שימושים שונים, בין היתר כדי לגבות או להעביר אותו בין תכנות.

Serialization (סִדְרוּת)

התהליך שבו אנו לוקחים אובייקט ושומרים את כל השדות שלו נקרא סריאליזציה (ובעברית: סִדְרוּת). מדובר בתהליך רקורסיבי שעובר על השדות של אובייקט (ובמידת הצורך אם מדובר בשדה שהוא אובייקט מורכב בפני עצמו - עובר גם על השדות שלו, וכך הלאה). נעשה זאת באמצעות שימוש ב-stream. בתהליך ההפוך, Deserialization, אנחנו לוקחים אובייקט שנכתב ל-stream ומשחזרים ממנו את המצב שלו.

כברירת מחדל בשפה ניתן לבצע סריאליזציה לשדות שהם טיפוסים פרימיטיביים, אבל מחלקות בשדה לא מאפשרות סריאליזציה; כדי לאפשר סריאליזציה של מחלקה היא צריכה לממש את הממשק Serializable שהוא ממשק ריק ללא מתודות[1]. מכיוון שמדובר בתהליך רקורסיבי, כל אחד מהשדות של המחלקה צריך להיות שדה פרימיטיבי או לממש גם הוא את Serializable.

תהליך הכתיבה (הסריאליזציה) של אובייקט נעשה באמצעות המחלקה `ObjectOutputStream`, ותהליך הקריאה (דה-סריאליזציה) נעשה באמצעות המחלקה `ObjectInputStream`. שתי המחלקות הנ"ל הינן מחלקות Decorators של stream.

לדוגמה: נניח שאנחנו רוצים לכתוב לתוך קובץ בשם save.ser. תחילה נגדיר `FileOutputStream` ואז נגדיר `ObjectOutputStream` שמקבל אותו כארגומנט; נבצע סריאליזציה בעזרת המתודה `writeObject` שמקבלת אובייקט ושומרת אותו. התהליך ההפוך נעשה בצורה דומה עם המתודה `readObject` שערך ההחזרה שלה הוא Object ולכן יש צורך לבצע המרה (down-casting).

```
// Write an object
try(OutputStream out = new FileOutputStream("save.ser");
   ObjectOutputStream oos = new ObjectOutputStream(out);) {
   oos.writeObject(new Date());

} catch (IOException exception){
   ...
}


// Read an object
try (InputStream in = new FileInputStream("save.ser");
    ObjectInputStream ois = new ObjectInputStream(in);){
   Date d = (Date) ois.readObject();

} catch (IOException | ClassNotFound exception){
   ...
}
```

בדרך הזו לא ניתן לבצע סריאליזציה של אובייקט מטיפוס פרימיטיבי בפני עצמו (למשל `out.writeObject(5)`), אבל נוכל לעשות זאת בעזרת מתודות של `ObjectOutputStream` כגון `writeInt`. יודגש כי לא מדובר כאן על סריאליזציה של שדה פרימיטיבי של אובייקט מורכב שנעשית בדרך שתיארנו קודם לכן.

אפשר גם לדרוס את המתודות `writeObject` ו-`readObject` במחלקה שלנו כדי לממש בעצמנו כתיבה וקריאה של אובייקטים.

[1] כהערת אגב, ממשקים ריקים כאלה שאינם מכילים מתודות ורק מסמלים דבר מה נקראים Marker interfaces.

כפילויות

ככלל, כל אובייקט יישמר פעם אחת בלבד, גם אם מצב פנימי שלו השתנה מאז הפעם הקודמת ששמרנו אותו; אם ננסה לשמור אובייקט שכבר שמרנו מה שיישמר הוא רק מצביע לאותו האובייקט שנשמר, כך ששינוי במצב פנימי של אובייקט לא יתועד. אפשר להתגבר על זה בעזרת סגירה של הסטרים בין כתיבה לכתיבה (בעזרת `out.close()`) ופתיחתו מחדש או קריאה למתודה `out.reset()` ששוכחת את מה שנשמר עד כה) – אבל כמובן שיש לזה מחיר בהיבט של זמן ריצה וזיכרון.

transient

אפשר להגדיר שדות של אובייקט בתור `transient` כדי לסמן שאיננו רוצים לשמור אותם; תהליך הסריאליזציה ידלג עליהם ובתהליך הקריאה הם יאותחלו לערך הדיפולטיבי שלהם. נשתמש בכך עבור שדות שאינם רלוונטיים לצורך איחזור האובייקט. גם שדות סטטיים שאינם שייכים לאובייקט מסויים (אלא למחלקה כולה) אינם נכללים בתהליך הסריאליזציה.

שינויים במחלקה

מה יקרה אם יתבצעו שינויים במחלקה בין שמירה של אובייקט לבין טעינה שלו? אם נשנה טיפוסים של שדות או את ההיררכיה של המחלקה לא נוכל לטעון את האובייקט השמור – ומאידך אם רק נמחוק שדות אפשר להתעלם ממנו ולא לפגוע בתהליך הטעינה. פיתרון נוסף לכך הוא לתעד את הגרסה הנוכחית של המחלקה: נגדיר משתנה סטטי שנקרא `SerialVersionUID` ששומר את הגרסה של המחלקה ונשנה אותו כשנבצע שינויים קריטיים במחלקה.

```
private static final long serialVersionUID = 1L;
```

למרות שמדובר בשדה סטטי הוא כן יישמר יחד עם האובייקט והוא כן נשמר בסריאליזציה. הוא חייב להיות מטיפוס `long`, וכדאי לשמור אותו בתור `private static`. אם ננסה לקרוא אובייקט מגרסה ישנה יותר במהלך תהליך ה- `deserialization` תיזרק חריגה מסוג `InvalidClassException`. אם לא מגדירים את השדה הוא מוגדר באופן דיפולטיבי על-ידי השפה וישתנה עם כל שינוי שנבצע במחלקה. לכן מומלץ להגדיר את השדה בעצמנו כשעובדים עם אובייקט שהוא serializable ולהחליט באופן מודע מתי שינויים דורשים עדכון של הגרסה ומתי לא.

שיכפול בתוך התוכנה – Cloning

קיימים שני סוגים של העתקה של אובייקט:

- העתקה רדודה (Shallow Copy) – העתקה של "השכבה החיצונית" בלבד, כך ששדות שאינם פרימיטיביים לא יועתקו by-value אלא רק המצביע אליהם יועתק; דהיינו, השדה באובייקט המקורי והשדה המקביל באובייקט המשוכפל יצביעו לאותו אובייקט.
- העתקה עמוקה (Deep Copy) – כל שדה שאינו פרימיטיבי יועתק כעותק נפרד לאובייקט השני.

כדי שיהיה אפשר לשכפל מחלקה היא צריכה לממש את הממשק הריק `Cloneable` ולדרוס את המתודה `clone()` של המחלקה Object. כברירת מחדל (אם לא נדרוס את המתודה) היא תחזיר העתקה רדודה של האובייקט. במתודה `clone()` נעשים שני דברים:

- אם האובייקט אינו מממש את הממשק `Cloneable`, כלומר אינו תומך בשכפול של עצמו, תיזרק החריגה `CloneNotSupportedException`
- העתקה רדודה של האובייקט.

גם מערכים מממשים את `Cloneable` והמתודה `clone()` שלהם מחזיקה העתקה רדודה.

לדוגמה, נתבונן על המימוש של `clone()` במחלקה שמייצגת חיית מחמד:

```
public class Pet implements Cloneable {

   private Data birthDate;

   public Object clone() throws CloneNotSupportedException{

      // creating a shallow copy
      Pet pet = (Pet) super.clone();

      // Cloning date for deep copy
      pet.birthDate = (Date) birthDate.clone();

      return pet;
   }

}
```

אלטרנטיבה לכך היא לממש copy-constructor – קונסטרקטור שמקבל אובייקט של אותה המחלקה ויוצר העתק שלו:

```
public Pet(Pet other){

   this(); // First - calling default constructor

   // If field class doesn't have a copy constructor, we can clone
   this.birthDate = other.birthDate.clone();
}
```

Reflections

מנגנון שמאפשר לתוכנה "להסתכל על עצמה" או לשנות חלקים בתוכה.

המחלקה Class

נזכיר כי כל אובייקט בשפה יכול להיות טיפוס פרימיטיבי או רפרנס (מחלקה, מערך או ממשק). לכל אובייקט שהוא רפרנס מאותחל אובייקט מסוג `java.lang.Class` שאפשר לגשת אליו בשתי דרכים:

אפשרות אחת היא לקרוא למתודה הסטטית `Class.forName()` עם שם של מחלקה:

```
Class cls = Class.forName("MyClass");
```

והתוצאה היא שהמשתנה cls מכיל כעת אובייקט מסוג `Class` של המחלקה `MyClass`. אם לא קיימת מחלקה בשם `"MyClass"` תיזרק חריגה `ClassNotFountException`.

דרך נוספת היא לגשת דרך אובייקט של מחלקה מסוימת ולקרוא למתודה `getClass()`:

```
Class cls = myObj.getClass();
```

כעת, לכשיש בידינו אובייקט cls של המחלקה `Class`, אפשר לבצע כל מיני דברים.

אפשר ליצור רשימה של הקונסטרקטורים של המחלקה וליצור מופע חדש של המחקה:

```
Constructor [] ctorlist = cls.getDeclaredConstructors();
Object retboj = ctorlist[i].newInstance(arglist);
```

כלומר לכתוב מחלקה שמקבלת מחרוזת עם שם של מחלקה ויוצרת מופע חדש של המחלקה, בלי לדעת באיזו מחלקה מדובר.

אפשר לקבל רשימה של המתודות של המחלקה, כולל מתודות פרטיות:

```
Method [] methlist = cls.getDeclaredMethods();
```

ואף להריץ את המתודות:

```
methlist[j].invoke(obj, arglist);
```

אפשר לקבל גם שדות של המחלקה ופרטים עליהם:

```
Field [] fieldList = cls.getDeclaredFiled();
```

וכך לקרוא למתודות `set` ו-`get` כדי לקבל או לשנות את הערך של שדה. באופן דיפולטיבי לא ניתן לגשת באופן המתואר לשדות פרטיים, אבל כן קיימת דרך לגשת אליהם! לכן כפי שהזכרנו בתחילת הקורס השימוש ב-`private` הוא אלמנט עיצובי ולא נועד להצפין מידע.

```
field.setAccessible(true);
```

שימושים

יש בכך פוטנציאל לכתוב קוד נאה וקריא (למשל להימנע מבלוקים ארוכים של switch) ולכתוב קוד כללי לגמרי – פשוט צריך לקבל סטרינג שמכיל שם של מחלקה כדי לבנות אובייקט של אותה המחלקה. זה יכול להיות שימושי למשל במחלקות Factory.

בין היתר, בסריאליזציה של מחלקות מקבלים גישה לשדות פרטיים של אובייקט בעזרת המנגנון הזה.

מאידך, המנגנון פוגע בעקרונות של כימוס והחבאה של מידע. בנוסף, שימוש ברפלקציה עלול ליצור בעיות בקוד ובהרצה שלו, וגם פוגע בזמן הריצה של התכנית.

דוגמאות

בקוד הבא נדפיס את כל השדות של המחלקה FieldExample ולכל שדה נדפיס את השם שלו, הטיפוס שלו וה-modifiers שאיתו הוא מוגדר:

```
public class FieldExample{

   private double d;
   public static final int i = 37;
   String s = "testing";

   public static void main(String [] args) throws ClassNotFoundExcep{

      Class cls = Class.forName("FieldExample");
      Field[] fieldList = cls.getDeclaredFields();

      for (Field curField : fieldList) {

         System.out.println("name = " + curField.getName());
         System.out.println("type = " + curField.getType());
         System.out.println("modifiers = " +
Modifier.toString(curField.getModifiers () ));
      }
```

כך שהפלט שלנו יהיה:

```
name = d
type = double
modifiers = private

name = i
type = int
modifiers = public static final

name = s
type = class java.lang.String
modifiers =
```

בקוד הבא נקרא למתודה add של המחלקה MyClass:

```
class MyClass{

   public int add(int a, int b){
      return a + b;
   }

   public static void main(String [] args) throws … {
      Class cls = MyClass.class;
      Class [] parametersType = {int.class, int.class};
      Method method = cls.getMethod("add", parametersType);

      MyClass newClassOjbect = new MyClass();
      Object [] argList = new Object[2];
      argList[0] = new Integer(27);
      argList[1] = new Integer(3);
      Integer returnInt =
                  (Integer) method.invoke(newClassOjbect, argList);
      System.out.println(returnInt); // 30
   }
}
```