

A close-up, slightly blurred image of a red pencil with a sharpened lead tip, resting on a piece of graph paper. The pencil is angled diagonally across the frame. The graph paper has a grid pattern, and some faint, handwritten numbers are visible in the background.

INTRO TO S/W ENG

QUICK INTRO TO JAVA

מה הלאה? JAVA!



- JRE – סביבת הריצה
- JDK – כלים לסביבת הפיתוח: Java Development Kit
- סביבות פיתוח שונות ומרובות
- **Eclipse – קוד פתוח**
- IntelliJ IDEA – JetBrains הצ'כית
- NetBeans
- VSCODE
- Xcode (for Mac fans)
- BlueJ, Jdeveloper, MyEclipse, DrJava, Jcreator, ...

מה ב-JRE וב-JDK?

Java Language

Java Language

java	javac	javadoc	jar	javap	jdeps	Scripting
Security	Monitoring	JConsole	VisualVM	JMC	JFR	
JPDA	JVM TI	IDL	RMI	Java DB	Deployment	
Internationalization		Web Services		Troubleshooting		

Tools & Tool APIs

Deployment

Java Web Start

Applet / Java Plug-in

JavaFX

User Interface Toolkits

Swing

Java 2D

AWT

Accessibility

Drag and Drop

Input Methods

Image I/O

Print Service

Sound

Integration Libraries

IDL

JDBC

JNDI

RMI

RMI-IIOP

Scripting

Beans

Security

Serialization

Extension Mechanism

JMX

XML JAXP

Networking

Override Mechanism

JNI

Date and Time

Input/Output

Internationalization

lang and util

lang and util Base Libraries

Math

Collections

Ref Objects

Regular Expressions

Logging

Management

Instrumentation

Concurrency Utilities

Reflection

Versioning

Preferences API

JAR

Zip

Java Virtual Machine

Java HotSpot Client and Server VM

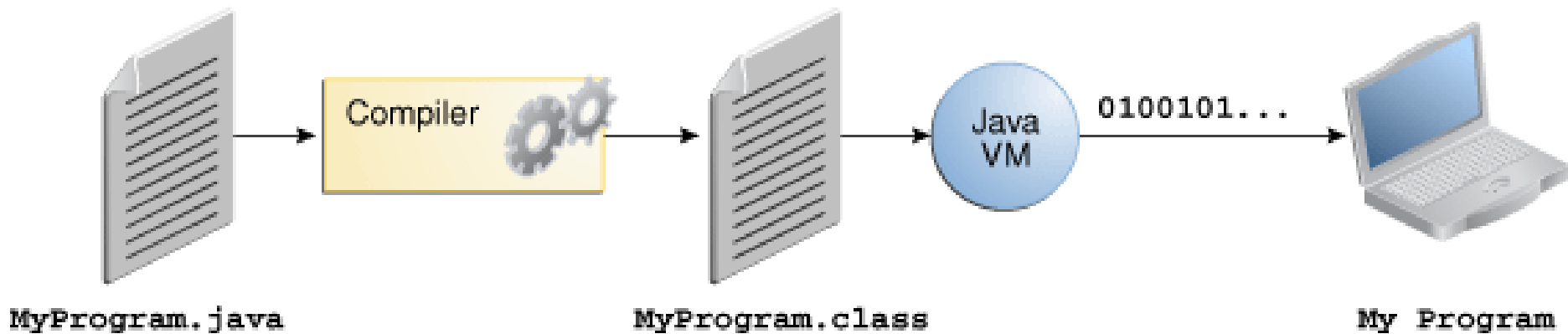
Java SE
API

Compact
Profiles

JDK

JRE

תוכנית ראשונה



➤ עורך טקסטים

➤ `javac` – מהדר (קומפיילר)

➤ `Java` – כלי להפעלת `JVM` והרצת תוכנית

➤ נשתדל ששם הקובץ == שם המחלקה

חלון הקונסול

```
> dir
04/26/2016 10:35 PM <DIR> .
04/26/2016 10:35 PM <DIR> ..
04/26/2016 10:35 PM 3,625 HelloWorld.java
> javac HelloWorld.java
> dir
04/26/2016 10:35 PM <DIR> .
04/26/2016 10:35 PM <DIR> ..
04/26/2016 10:35 PM 3,625 HelloWorld.java
04/26/2016 10:35 PM 625 HelloWorldApp.class
```

שימו לב - נוצר קובץ עבור כל מחלקה במודול

```
> java HelloWorldApp
```

הרצת מכונת ג'אווה עם המחלקה הראשית

שימו לב שלהבא נשתדל לתת שם קובץ == שם המחלקה העיקרית

תוכנית ראשונה

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Print to Console
    }
}
```

4 סוגי משתנים בשפה של ג'אווה

➤ **משתני מופע = שדות לא סטטיים** במחלקות

➤ Instance variables ,Object variables

➤ **משתני מחלקה = שדות סטטיים**

➤ Class variables

➤ עם מילת מפתח **static**

➤ משתנים מקומיים – כרגיל

➤ בתוך פונקציות (מתודות)

➤ פרמטרים של פונקציות – כרגיל



שמות משתנים



➤ כרגיל

➤ מוסכמות:

➤ לא משתמשים בסימנים \$ או _

➤ רצף מילים בד"כ ללא קיצורים

➤ מילה ראשונה מתחילה באות קטנה והיתר מתחילות באות גדולה, מילה עיקרית – שם עצם ביחיד או ברבים (במקרה של שם מערך או אוסף)

```
int veryBigDream;
```


קבועים



➤ משתנים בתוספת מילת מפתח **final** וערך
אתחול (לא חובה בשדה קבוע)

➤ במקרה של שדה קבוע ניתן לא לאתחל אלא
לתת ערך (פעם אחת) בבנאי (פונקציה בונה)

➤ מוסכמה של שמות: רצף מילים עם כל אותיות
גדולות כאשר סימן _ מפריד בין המילים

```
final int MAX_NUM_STUDENTS = 24;
```

כמה הבדלים עקרוניים עם C#

- אין אובייקטים מסוג Value Type – אך יש נתונים פשוטים
- אין העברת פרמטרים בהפניה (by reference)
- אין "properties" – יש ליישם באופן מפורש פונקציות set ו-get לשדות אם וכאשר נדרש
- אין אפשרות לבצע חפיפת אופרטורים בכלל
- אין העמסת אופרטורים
- ועוד

סוגים



➤ סוגים פשוטים (primitive data types)

➤ מערך (array)

➤ סוגים מוגדרים ע"י מתכנת (custom data types)

➤ מחלקה (class)

➤ ממשק (interface)

➤ אנומרציה (enum)

➤ מוסכמה של שמות: כמו במשתנים – רק מתחילין באות גדולה

```
class MyFirstClass {...}
```

```
enum WeekDays { SUN, MON, ... }
```

```
class Students extends ArrayList<Student> { ... }
```

סוגים ומודולים



ניתן להגדיר כמה סוגים במודול (קובץ `java`) ➤

עבור כל סוג נוצר קובץ `class` נפרד ➤

רק סוג אחד במודול יכול להיות `public` ➤

שם המודול חייב להיות זהה לשם הסוג הזה ➤

אין סיבה אמיתית לדרישה הזו וחלק ממהדרי ➤

`java` לא מתעקשים על זה

יש טוענים שהדרישה – חיסכון בזמן קומפילציה ➤

סוגים פשוטים

כמו סוג ערך ב-C# אך לא יורש מכלום ואין מתודות ושדות...



byte - 8 ביטים - מספר בין 128- עד 127 ➤

short - 16 ביטים - מספר בין 32,768- עד 32,767 ➤

int - 32 ביטים - מספר בין 2^{31} - עד $2^{31}-1$ ➤

long - 64 ביטים - מספר בין 2^{63} - עד $2^{63}-1$ ➤

float - 32 ביטים - מספר דצימלי עם נקודה צפה ➤

double - 64 ביטים - מספר דצימלי עם נקודה צפה ➤

boolean - מייצג ערך לוגי - אמת\שקר (true\false) ➤

char - 16 ביטים - תו בודד בקידוד Unicode ➤

void - אין מקום - כלום ➤

ערכי ברירת מחדל



Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
<i>Any kind of object</i>	null

נ.ב. מחרוזות נתמכות בעזרת מחלקה מובנת String

סוגים פשוטים ומחלקות עוטפות (wrappers)

- ב-java סוגים פרימיטיביים אינם אובייקטים
- אבל לכל סוג פרימיטיבי יש מחלקה עוטפת
Boolean, Double, Float, Long, Integer, Short, Byte
Void, Character
- המחלקות העוטפות מספקות מתודות סטטיות ולא סטטיות
- ישנה המרה מרומזת בין נתון מסוג פשוט לבין אובייקט מסוג עוטף ובחזרה

מה יש במחלקות עוטפות?



Number – פונקציות מופע

`byteValue()/doubleValue()/...`

Integer\Long – פונקציות סטטיות

`getInteger(String nm)\parseInt(String s)\max(int,int)\...`

Array – פונקציות סטטיות

`getInt(Object array, int index)\...`

String

`join/valueOf/format`: סטטיות

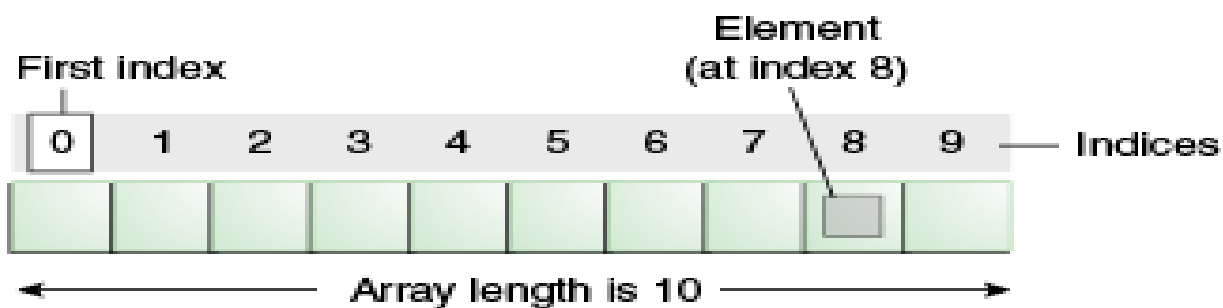
`concat/contains/endsWith/...`: מופע

Enum

`valueOf/values`: סטטיות

`name\ordinal`: מופע

מערכים



מערך ב-Java הנו אובייקט



```
int[] anArray;
```

הצהרה

```
anArray = new int[10];
```

הקצאת זיכרון

```
anArray[2] = 8;
```

אתחול אלמנט

הצהרה ואתחול מקוצר:

```
int[] otherArray = { 1, 2, 3, 4, 5};
```

אין מערך רב מימדי: אבל יש מערך של מערכים:

```
int[][][] arrayOfArraysOfArrays;
```

דורש יצירת אובייקטים מרובים...

הוראות (statements) – כרגיל




פעולות (operators)

פעולות כמו ב-C# ➤

Operator Precedence	
Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~expr !expr
multiplicative	% / *
additive	- +
shift	<<< << >>
relational	< > <= >= instanceof
equality	=! ==
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	: ?
assignment	=<<< =<< =>> = ^= & =% =/ =* =- =+ =

הוראת תנאי – כרגיל

```
if (condition) {  
    ...  
} else {  
    ...  
}
```



```
if (condition1) {  
    ...  
} else if (condition2) {  
    ...  
} else if {  
    ...  
} ...
```

ואם יש רצף? ➤

הוראת מיתוג

```
switch (expression) {  
  case value1:  
    ...  
    break;  
  case value2:  
  case value3:  
    ...  
  case value4:  
    ...  
    break;  
  default:  
    ...  
    break;  
}
```

שימו לב שאין חובה ב-**break**, אבל... יש סכנה... ➤

לולאות



```
while (condiion) {  
    ...  
    ... break;  
    ...  
    ... continue;  
    ...  
}
```

```
do {  
    ...  
    ... break;  
    ...  
    ... continue;  
    ...  
} while (condiion);
```

```
label1:  
while (condition1) {  
    ...  
    while (condition2) {  
        ... break label1;  
        ...  
    }  
    ...  
}
```

C#-1 כמו ב-`break` ו-`continue` ➤

ובלולאות מקוננות: ➤

לולאת איטרצייה



```
for (initialization; termination; increment) {  
    ...  
    break;  
    ...  
    continue;  
    ...  
}
```

חוקי **break** ו-**continue** כמו בלולאות לעיל ➤
דוגמה ללולאה "יורדת" ➤

```
for (int i = maxSize; i > 0; --i) {  
    ...  
}
```

לולאה על אלמנטים של אוסף\מערך



```
for (int i : collection) {  
    ...  
    break;  
    ...  
    continue;  
    ...  
}
```

➤ חוקי **break** ו-**continue** כמו בלולאות לעיל

➤ לולאה על אוסף "דומה" ל-**foreach** ב-**C#**

```
int[] anArray = {1, 2, 3, 4}  
for (int i : anArray) {  
    System.out.println(i);  
}
```


חבילות – packages

דומה ל-namespace ב-C# (אבל לא ממש אותו דבר) ➤

שם יכול להיות מורכב מכמה חלקים – כמו שם דומיין בכתובת אינטרנט ➤

בקורס שלנו – מילה בודדת ➤

מוגדר בתחילת מודול בעזרת מילה **package** ➤

package primitives;

ב-FRAMEWORKS של java הדומיין הראשי – ➤

java



חבילות – packages



- לאחר קומפילציה נוצר עץ תת-תיקיות ע"פ פירוק השם של החבילה
- בקורס שלנו – שם חבילה = שם תת-תיקיה
- JVM מחפש את המחלקות
- בעץ התיקיות לפי שם החבילה
- ולפי קובץ בשם המחלקה עם סיומת `class`

הכרת חבילות אחרות

קצת דומה ל-`using` ב-`C#` ➤

בעזרת מילת מפתח **`import`** ובתוספת שם הסוג שעושים הכרה אתו ➤

`import java.util.ArrayList;`

ואם רוצים להכיר את כל הסוגים מהחבילה? ➤

`import java.util.*;`

כל הסוגים המוגדרים בחבילה **`java.lang`** – מוכרים תמיד ללא צורך ב-**`import`**, למשל **`String`** ➤

וזהו בעצם ה-FRAMEWORK הבסיסי של השפה ➤

ייבוא של איברים סטטיים כאילו "גלובליים": ➤

`import static primitives.Util.*;`

מחלקה

דומה ל-C# ➤



```
class Vehicle { ... }
```

ירושה רק מאבא אחד כמו ב-C# ➤

```
class Truck extends Vehicle { ... }
```

הרשאה: מחלקה יכולה להיות **public** או ללא
הרשאה (package friendly) – מוכרת בתוך כל
החבילה

מחלקה ללא אבא יורשת מ-**Object** ➤

גישה לאיברי האבא בעזרת **super** ➤

כמו base ב-C# ➤

איברים במחלקה

➤ אתחול השדות – אפשרי בהוראת ההגדרה או בבנאי (נדבר על בנאים בהמשך)



➤ ניתן להעמיס פונקציות עם חתימות שונות

➤ כל הפונקציות וירטואליות תמיד!!! בג'אווה פולימורפיזם הוא חלק בלתי נפרד מהמשחק

➤ ניתן לחסום המשך פולימורפיזם ע"י **final**

איברים סטטיים



- כללי למחלקה ומשותף לכל האובייקטים
- גישה דרך שם מחלקה או דרך אובייקט
- שדות ופונקציות (מתודות)
- הרשאות כרגיל
- אתחול השדות – אפשרי בהוראת ההגדרה או בלוק אתחול סטטי (תלמדו מה זה)
- אתחול – בשימוש ראשון במחלקה או ביצירת אובייקט ראשון ממנה

מחלקה אבסטרקטית



- מחלקה שלא ניתן ליצור ממנה אובייקטים
- בד"כ עם פונקציות לא ממומשות
- מילת מפתח **abstract** למחלקה ולפונקציות לא ממומשות

```
abstract class Vehicle {  
    ...  
    abstract void func();  
    ...  
}
```

- מחלקות יורשות המיועדות ליצירת אובייקטים חייבות לממש את כל הפונקציות האבסטרקטיות
- מחלקה אבסטרקטית שמממשת אינטרפייס (נדבר בשקף הבא) לא חייבת לממש את כל הפונקציות של האינטרפייס

ממשקים (אינטרפייסים)

מילת מפתח `interface` ➤

`interface MyInterface extends OtherInterf1, OtherInterf2 { ... }`

מכיל רק חתימות פונקציות ללא הרשאה – ➤

הרשאה הפונקציות בעצם תמיד `public`

הממשק יכול להיות `public` או ללא הרשאה ➤

(`package friendly`) – מוכר בתוך כל החבילה

מחלקות יכולות לממש מספר ממשקים ➤

`class Truck extends Vehicle implements MyInterf1, MyInterf2 { ... }`



איברים והרשאות גישה



- שדות ופונקציות (מתודות)
- הרשאות כרגיל: **public**, **private**, **protected**
- אם אין הרשאה – הכרה בתוך החבילה (כנ"ל)
(package friendly)

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

עוד על מחלקות ואינטרפייסים



➤ מה אין באינטרפייסים ב-Java?

➤ אין שדות

➤ אין הרשאות (הכול `public`)

➤ לא ניתן ליצור אובייקט


➤ ירושה ב-Java:

➤ שדה עם שם זהה – מסתיר

➤ פונקציה עם שם זהה – מעמיסה (פולימורפיזם)

➤ פונקציה סטטית עם שם זהה – מסתירה

יצירת אובייקטים

- 
- משתנים מסוג הפניה (שדות, מקומיים וכו') הם בעצם מחזיקים הפניה לאובייקט
 - אובייקטים תמיד דינמיים – נוצרים `new` ע"י
 - אם אין יצירה – אין אובייקט (`null`)

```
MyClass myObject = new MyClass(1, 2, 3);
```

- משתנה מסוג אבא יכול להחזיק הפניה לבן

```
Father daddy = new Son("Yoni");
```

- משתנה מסוג ממשק יכול להחזיק הפניה לאובייקט מסוג שמממש את הממשק

שימוש באובייקטים

- המרות למעלה\למטה כמו ב-C#
- בדיקת התאמת סוג (כמו is ב-C#)



```
if (daddy instanceof Son) { ... }
```

- הצבה בין משתנים מעתיקה את ההפניה
- השוואת משתנים בודקת שיויון ההפניות
- לצורך השוואת ערכים נשתמש בפונקציה **equals** שירשנו מ-**Object** (וגם נחפוף אותה!)

העברת פרמטרים לפונקציות



➤ **אין** OUT\REF כמו ב-C#

➤ ארגומנטים מועברים לפרמטרים רק ע"י ערך

➤ בסוגים פשוטים – הכול פשוט

➤ בסוגי הפניה (אובייקטים) – הערך שמועבר הוא בעצם הפניה לאובייקט

➤ פרמטר אחרון יכול להיות "אליפסה" – שלוש


נקודות – שזה אומר כמות משתנה של

פרמטרים (דומה ל-params type[] ש C#):

`public PrintStream printf(String format, Object... args)`

➤ במימוש הפרמטר הזה – מערך

בתוך הפונקציות

- 
- שדות ומתודות של האובייקט (במקרה של **this**)
 - התנגשות עם שמות הפרמטרים) – **this**
 - גישה לשדות\מתודות של אבא – **super**
 - החזרת ערך – כרגיל
 - בבנאי, בשורה הראשונה מופעל בנאי ברירת מחדל של מחלקת אב
 - אלא אם כתבנו בשורה הראשונה:

this(parameters of another class constructor);

או: ➤

super(parameters of base class constructor);

עוד על יצירת אובייקטים



➤ בשביל ליצור אובייקט אנחנו:

➤ מגדירים משתנה

➤ יוצרים את האובייקט ע"י **new** ושם המחלקה

➤ מאתחלים את האובייקט ע"י **new** בנאי ע"פ ארגומנטים
שהצמדנו לשם מחלקה עם **new**

➤ אם לא הגדרנו אף בנאי – הקומפיילר נותן לנו
בנאי ללא פרמטרים עם הרשאה כללית – הכול
מאותחל ל"אפסים" – ע"פ הטבלה שראינו קודם

הגדרת בנאי



➤ לא מגדירים לבנאי סוג ערך מוחזר
➤ ניתן להעמיס מספר בנאים – עם פרמטרים שונים

➤ הרשאות כרגיל

➤ לא רוצים לשכתב אותו קוד – ניתן להפעיל בנאי מוכן
ע"י `this(parameters)` – אבל רק כהוראה ראשונה בבנאי

```
public MyClass(int id, String name, char gender) {  
    this(id, name);  
    this.gender = gender;  
}
```

➤ בצורה דומה – הפעלת בנאי מסויים של אבא
בעזרת `super(...)` (אם לאו – יופעל בנאי ברירת
מחדל של אבא) – גם רק בהוראה ראשונה

מחזור חיים של אובייקט



- **JVM מחזיק מידע לגבי כל אובייקט – כתובת שלו, כמות הפניות אליו ועוד**
- **למשל כאשר מציבים הפניה לאובייקט למשתנה – כמות הפניות גדלה**
- **כאשר דורסים את ההפניה במתשנה – כמות הפניות קטנה**
- **כאשר כמות הפניות יורדת לאפס – האובייקט נהיה למטרה של **Garbage Collector****
- **הוא יימחק וזכרוננו ישוחרר מתישהו...**

סוגי Enum



➤ בהתחלה – דומה ל-C#

```
enum Weekdays {SUN, MON, TUE, WED, THU, FRI, SAT}  
Weekdays day = Weekdays.MON;  
switch (day) {  
    case SUN: ... break;  
    ...  
}
```


➤ הערכים אינם קבועיים שלמים, חל מ-0 וגדל

ב-1 – כמו שהיה ב-C#

➤ לא ניתן לקבע ערך בצורה ישירה כמו ב-C#

```
enum MenuOptions {CHOOSE = 2, NEXT, MORE, EXIT = 0, MINE = MORE}
```

החזרת ערך מפונקציה ופולימורפיזם

- 
- סוג ערך מוחזר יכול להיות גם מחלקה או ממשק
 - ערך שמוחזר בפועל יכול להיות מסוג שיורש או מממש את הסוג הנ"ל
 - בפולימורפיזם ניתן להעמיס (override) פונקציה שסוג הערך המוחזר הנו יורש או מממש את הסוג של הפונקציה הוירטואלית המקורית

חריגות

ניתן לצאת מפונקציה ע"י זריקת חריגה – דומה ל-C#

`throw new SomeException(parameters);`

רוב החריגות יורשות מ-**Exception**

כל החריגות מממשות ממשק **Throwable**

Error ו-**Exception** ממשות אותו

לא מקובל להשתמש ב-**Exception** – יש אוסף עשיר של מחלקות (עץ שלם) וניתן גם להגדיר סוגי חריגה משלכם – תוך ירושה מחריגה קיימת מתאימה

יש "בן" מיוחד – **RuntimeException**

חריגות

ניתן לצאת מפונקציה ע"י זריקת חריגה –
דומה ל-C#

`throw new SomeException(parameters);`

רוב החריגות יורשות מ-**Exception**



פונקציה שזורקת חריגה

➤ חריגות שהם **checked** דורשות רישום בהגדרת פונקציה שזורקת אותם (או ממשיכה לזרוק)

`void myFunc() throws IOException { ... throw new IOException(); ... }`

➤ אחרת טעות קומפילציה

➤ ניתן לתת רשימה של חריגות


➤ ניתן להכריז על קבוצת חריגות – ע"פ ירושה

➤ על חריגות שהם **unchecked** לא צריך להכריז

➤ בקורס שלנו נשתמש רק בחריגות **unchecked**
ולכן לא נכתוב **throws**

תפיסת חריגות

דומה ל-C# ➤



```
try {  
    ... /* קוד מוגן */  
} catch (Type10fException e) {  
    ... /* קוד בתפיסה */  
} catch (Type20fException e) {  
    ... /* קוד בתפיסה */  
} ...  
} finally {  
    ... /* קוד לביצוע בכל מקרה */  
}
```

אוספים – Collections Framework

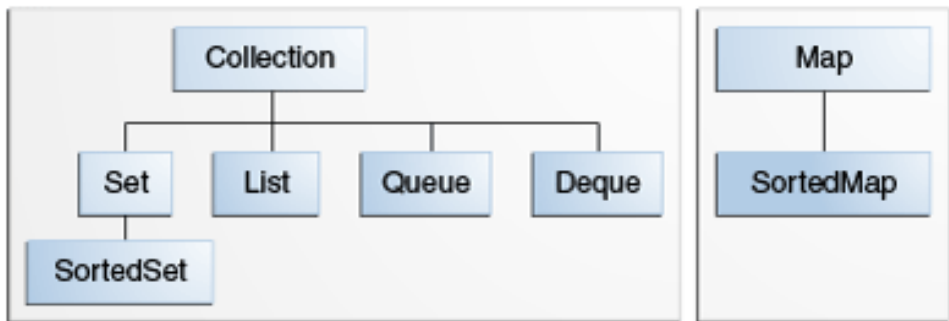


➤ נמצאים ב-`java.util`

➤ אינטרפייס בסיס – `Collection`

➤ מממשים ממשק `Iterable` + מחלקה

➤ עבור `{...}` *(type it : collection)* `for`



➤ ממשקים (גנריים)

➤ `Set` – אוסף ללא אלמנטים כפולים

➤ `List` – אוסף מסודר – עם אינדקסים

➤ `Map` – אוסף זוגות מפתח-ערך

על אוספים



מימוש List ➤

➤ אוסף שימושי – ArrayList (גנרי) – מערך דינמי

➤ LinkedList – רשימה מקושרת (גנרית)

פונקציות אוספים ➤

➤ פונקציות מופע של אוסף

➤ פונקציות סטטיות של מחלקה Collections:

`sort\shuffle\reverse\fill\copy\swap\binarySearch\...`

JAVADOC



➤ סטנדרטיזציה של תיעוד בקוד

➤ לפני מחלקה או פונקציה וכו'

➤ מקלידים **/**** / ולוחצים ENTER

➤ נוצרת הערה ייחודית בפורמט מוגדר

➤ כולל אלמנטים עם תגיות: `@author`, `@param`, ...

➤ יצירת תעוד ע"פ ההערות הנ"ל כ"דפי web"

➤ בתפריט כלים מפעילים יצירת JavaDoc

➤ כלי javadoc של JDK

תודה רבה ולהתראות

