# עקרונות כלליים

## כתיבת קוד נכון

מאוד חשוב לכתוב קוד בצורה נכונה לפי כל העקרונות שנפרט בהמשך. יש לכך מספר סיבות:

* קוד יהיה יעיל ואיכותי יותר.
* יותר קל לפתור בעיות.
* קוד גמיש - שינויים או הוספות יתבצעו בצורה קלה ונוחה. יותר קל לתחזוקה.
* הקוד יהיה קריא וברור למתכנתים אחרים ולא תלוי רק במי שכתב אותו.

## לשמור על פשטות

מתכנתים פעמים רבות נוטים לסבך את הקוד ולא לשמור על כל הכללים והעקרונות בכתיבת קוד נכון. שיטה זו יוצרת קוד מבולגן ולא יעיל. יש לבצע תכנון איכותי לפני שכותבים קוד ולא לכתוב את הקוד תוך כדי מחשבה. הרעיון המרכזי בכתיבת קוד הוא לשמור על פשטות כמה שיותר (עקרון ה-K.I.S.S).

קוד צריך לספר סיפור, לכן קוד שקשה להסביר אותו הוא כנראה לא קוד טוב. לפני שכותבים קוד מומלץ לחשוב ולספר לעצמך אותו בראש בצורה הכי קלה ומובנת, ולאחר מכן לכתוב אותו בצורה זו.

## תיעוד

מטרת ההערות להעביר מידע לקורא שאינו ברור מאליו מקריאת הקוד, אין צורך לרשום בהערות מידע המובן מאליו. עקרונית הקוד אמור להסביר את עצמו, הערות נכתוב בשביל להסביר את ה-'למה'. לא להשאיר קוד ישן בגרסה הסופית של התוכנית.

## עקרון S.O.C

לפי עקרון S.O.C (Separation Of Concerns) אין לכתוב סקריפט אחד כל יכול, אלא יש להפריד לחלקים (מודולים) את התוכנית כך שלכל חלק יש תפקיד אחד מוגדר על נושא יחיד. בין מודולים צריך שיהיה צמידות נמוכה ולכידות גבוהה.

**צמידות נמוכה** – כל מודול משמש כיחידה עצמאית ותלוי כמה שפחות במודולים אחרים במערכת. כלומר, אם משנים אחד אין צורך לשנות את השני. כאשר יש תלות בין מודולים יש לנהל תלות זו באמצעות ממשקים מצומצמים ואירועים.

**לכידות גבוהה** – בתוך מודול צריך להיות קשר חזק בין הפעולות השונות שנמצאות בו. במידה ואין קשר יש לפצל אותו למספר מודולים. נדע שיש לנו מודול מפוקס אם נוכל לתת לו הגדרה אחת.

## ארגון הקוד - תבנית MVC:

Controller – מקבל את כל הבקשות. הוא מפנה לקוד שנקרא model כדי לקבל ממנו דאטה. את מה שקיבל מהמודל מפנה ל-view שיחזיר לו קוד בצורת HTML שניתן להציג למשתמש. את התשובה הסופית מחזיר חזרה ללקוח.

Data Model – מתפעל את הדאטה. מחזיר את הדאטה הנדרש ממנו.

View – מקבל מידע להציג וממיר אותו ל-HTML מעוצב יפה.

בפרויקט גדול לכל משימה יש תבנית MVC נפרדת. כדי לשלוט בכל המשימות יוצרים front controller שיקבל את כל הבקשות ויפנה ל-controller המתאים.

## תהליך כתיבת קוד

1. בשלב ראשון המטרה היא ליצור קוד שעובד, כלומר מיישם את המטרה. בשלב זה פחות חשוב שהקוד יהיה יעיל ומהיר.
2. לכתוב טסטים איכותיים ולוודא שהקוד עובר אותם בהצלחה לאחר כל שלב בהמשך.
3. פישוט הקוד כדי שיהיה כמה שיותר מובן וקל לתחזוקה. תיקון באגים קטנים.
4. לבחון כמה זמן הרצה לוקח כל חלק בקוד ולנסות לקצר את זמן הריצה כמה שניתן.

# משתנים

## שמות משתנים

שמות משתנים צריכים להיות קצרים ובעלי משמעות, כך שפעולות השמה על המשתנים יהיו חלק "מהסיפור" של הקוד ולא פעולה הדורשת פיענוח.

יש שתי שיטות נפוצות בנתינת שמות עם כמה מילים למשתנים, אחת עם אות גדולה בתחילת מילה numPending (נקראת camelCase), והשנייה עם מקף תחתון num\_pending. לא משנה באיזו שיטה נבחר העיקר שנהיה עקביים בשיטה אחת למשך הקוד.

## סוגי משתנים

משתנים לוקליים: משתנים שהוגדרו בתוך בלוק מסוים. נראים רק בתוך בלוק זה, כולל בתוך בלוק פנימי. ניתן להצהיר בתוך בלוק על משתנה עם אותו שם כמו משתנה שהוצהר מחוץ לבלוק. במקרה זה ההצהרה החדשה "מחביאה" את ההצהרה הקודמת. לאחר הבלוק ערך המשתנה יחזור להיות כמו לפני הבלוק.

int x=1;

{   
   int x=2; //x is 2   
} //x is 1 again!

משתנים סטטיים: משתנים החיים מרגע תחילת התוכנית ועד סופה. תמיד מוקצים בתחילת ריצת התוכנית, לא משנה היכן הם הוגדרו. נגישים בבלוק בו הם הוגדרו ובכל בלוק פנימי (כמו משתנים מקומיים). מוגדרים ע"י כתיבת המילה השמורה static לפני שם הטיפוס בהגדרת המשתנה.

static int count;

מכיוון שמשתנים אלו חיים לכל אורך התוכנית, בפונקציה שמוגדר בה משתנה static, אם נפעילה מספר פעמים, בכל פעם אנו בעצם חוזרים לאותו משתנה, ואם נבצע השמה על משתנה זה השמה זו תישמר בכל פעם שנפעיל את הפונקציה.

אתחול משתנים סטטיים ניתן ע"י ערכים ידועים בזמן קומפילציה בלבד. אם לא צוין ערך אתחול, משתנה סטטי יאותחל ל-0, בניגוד למשתנה מקומי שאם לא יאותחל לא ניתן לצפות את ערכו.

משתנים גלובליים: משתנים שהוגדרו מחוץ לכל הבלוקים, כולל main. נגישים מכל נקודה בתוכנית **החל מרגע הגדרתם**. גם הם חיים מרגע תחילת התוכנית ועד סופה. מאותחלים ל-0 אם לא צוין ערך אתחול אחר. ניתן לאתחל רק לערכים קבועים הידועים בזמן קומפילציה.

## סוגי טיפוסים

**שלמים** - char, short, int, long, long long.

**חיוביים שלמים** - unsigned char, unsigned short, etc..

**שברים** - float, double, long double.

## אופרטורים

**חילוק** (/) - כשמבצעים חלוקה על מספרים שלמים התוצאה תהיה מספר שלם וזריקת השארית, לכן חשוב לעשות casting או לשנות את אחד השלמים.

float f=1/3; //0   
float f=(float)1/3; //0.3333..

float f=1/3.0; //0.3333..

**מודולו** (%) - פעולת השארית מוגדרת על שלמים בלבד, ומחזירה את השארית של פעולת החילוק בין שני הפרמטרים שלה.

11%-5 = 1

-11%-5 = -1

**קיצורים לאופרטורים אונאריים** - האופרטור a++ וכן ++a מבצעים את הפעולה a=a+1, אלא שההבדל ביניהם הוא כאשר יש עוד אופרטור במשוואה. a++ יבצע את הפעולה ואז יקדם את a, ואילו ++a יקדם קודם את a ואז יבצע את הפעולה הנוספת. אותם אופרטורים יש גם במינוס: a-- וגם --a.

Int x=3, y=0;

y = ++x + 5; // y=9, x=4

y = x++ + 5; // y=8, x=4

**קיצורים לאופרטורים בינאריים** - האופרטור a+=b מבצע את הפעולה a=a+b. אופרטורים דומים יש בין כל ארבעת הפעולות האריתמטיות: +, -, \*, /.

**אופרטורי השוואה** – הם אופרטורים בינאריים כמו: ==, =!, >, <, =>, =<, שמחזירים 0 או 1 בהתאם לנכונות ההשוואה.

**אופרטורים לוגיים** – מתייחסים לאופרנדים עליהם מופעלים כערכי אמת, כאשר אופרנד 0 משמעו false, וכל אופרנד השונה מ-0 משמעו true. מחזירים 0 או 1 לפי הכללים הבאים:

!A – הופך את ערך האמת של A. אם A שונה מ-0 אז ביטוי זה שווה 0, ואם שווה ל-0 ביטוי זה שווה 1.

B&&A – מחזיר 1 רק אם גם A וגם B אמיתיים.

B| |A – מחזיר 1 רק אם לפחות אחד מ-A או B אמיתיים.

**אופרטורים נוספים** – ניתן להפעיל על כל הטיפוסים מסוג מספרים שלמים.

& - כאשר מפעילים בין שני אופרנדים את הסימון &, פעולה זו לוקחת את שני המספרים הבינאריים המייצגים שני אופרנדים אלו ויוצרת מספר בינארי חדש, שבו בכל ספרה יהיה 1 רק אם יש בשני האופרנדים 1 במיקום זה, אחרת יהיה 0.

| - כמו ב-&, רק שבמספר הבינארי החדש יהיה 1 רק אם לפחות באחד מהאופרנדים יש 1 במיקום זה.

^ - כאשר מפעילים בין שני אופרנדים את הסימון ^, פעולה זו לוקחת את שני המספרים הבינאריים המייצגים שני אופרנדים אלו ויוצרת מספר בינארי חדש, שבו בכל ספרה יהיה 1 רק אם בשני האופרנדים הספרה שונה במיקום זה, אם זהה יהיה 0.

~ - כאשר מפעילים אופרטור זה על אופרנד, הוא מחליף את כל הספרות הבינאריות, 1 ב-0, ו-0 ב-1.

Int a = 5, b = 9; // a = 00000101, b = 00001001

c = a&b ; // c = 00000001

c = a | b ; // c = 00001101

c = a ^ b ; // c = 00001100

c = ~a ; // c = 11111010

<< - הסימון a<< פירושו הכפלת a ב-2. הסימון a<<n משמעו .

<< - הסימון a>> פירושו חילוק a ב-2. הסימון a>>n משמעו .

נרשום 0 לפני בהשמה של מספר כדי לציין שהוא בסיס אוקטלי, ו-0x כדי לציין בסיס הקסדצימלי.

# פונקציות

## פונקציות

כל פונקציה היא בעצם כתובת בזיכרון שלאחריה יש רצף מסוים של פקודות. כל שורת פקודה שהמחשב מבצע היא בעצם חלק מפונקציה. הפונקציה הראשית של התוכנה היא פונקציית main, וממנה אפשר לקרוא לפונקציות נוספות, כלומר לקפוץ לכתובת של הפונקציה הרצויה ולבצע את רצף הפקודות שלה. מתוך פונקציה אפשר לקפוץ לפונקציה אחרת ואפילו לפונקציה שאנו נמצאים בה (רקורסיה). בקריאה לפונקציה, הפונקציה שממנה קראנו נקראת caller, והפונקציה שקראנו לה נקראת callee. לאחר סיום כל הפקודות של הפונקציה נחזור לשורת הפקודה האחרונה שממנה הפעלנו את הפונקציה.

**משתני פונקציה:**

כל פונקציה צריכה להחזיק את המידע הבא:

* **כתובת החזרה בסיום הפונקציה** - כלומר כתובת שורת הפקודה שאחרי השורה שקראה לפונקציה.
* **פרמטרים לפונקציה** - הם האוגרים שנשלחים לפונקציה בתור פרמטרים, ועליהם הפונקציה מבצעת פעולות.
* **ערך חזרה מהפונקציה** - הוא הערך שאנו רוצים להחזיר מהפונקציה הנוכחית אל הקודמת.

**שימוש במחסנית:**

צריך לא לדרוס אוגרים קיימים במהלך הפונקציה. ולכן בתחילת הפונקציה נקצה מקום במחסנית לכל האוגרים שאנחנו רוצים לשמור בצד. בסוף הפונקציה נחזיר חזרה את הערכים ששמרנו במחסנית לאוגרים שדרסנו, ונשחרר את הזיכרון שהקצנו.

## כללי פונקציות

פונקציות מאוד שימושיות בארגון ושיפור הקוד שלנו. יש מספר מקרים בהן נרצה להשתמש בפונקציות:

1. מניעת שכפול קוד - כאשר רואים שאנו משתמשים בקוד זהה לפחות שלוש פעמים, יש לכתוב פונקציה אחת עבור קטע קוד זה ולקרוא לה במקום.
2. שימוש עתידי - כאשר אנו כותבים ממשק שישתמשו בו משתמשים אחרים, יש להגדיר פונקציות בעלות משמעות, מובנות, ומתועדות היטב, כדי שיהיה נוח לאחרים להשתמש בקוד שכתבנו.

השמות לפונקציות צריכות לתאר את הפעולה שהן מבצעות, לדוגמה get\_top\_k\_items, copy\_object.

## רקורסיה

פונקציה רקורסיבית היא פונקציה אשר על מנת לפתור בעיה מסוימת קוראת לעצמה (או לפונקציה שקוראת לה, במקרה של "רקורסיה הדדית"). השימוש ברקורסיה בעצם מאפשר לנו במקום להתמודד עם הבעיה הגדולה, לחלק את הבעיה לתתי בעיות שעליהם נפעיל שוב ושוב את הפונקציה הרקורסיבית.

כל פונקציה רקורסיבית צריכה להיות מחולקת לשני מקרים: "מקרה בסיס" (הנקרא גם "בסיס הרקורסיה") ו"מקרה רקורסיבי". במקרה הבסיס הפונקציה לא קוראת לעצמה אלא מחזירה ערך, שאם לא כן תהיה זו לולאה אינסופית.

בזמן הריצה, בכל קריאה לפונקציה נשמר כתובת החזרה ומוקצה מקום במחסנית עבור אוסף המשתנים המקומיים של הפונקציה. לאחר ביצוע הפונקציה ערכים אלו נשלפים מהמחסנית כך שכל הזיכרון שנדרש לחישובה מתפנה. בפונקציה רקורסיבית דורשת הרקורסיה זיכרון בגודל פרופורציונלי לעומקה. אם יהיו יותר מדי קריאות רקורסיביות תתקבל שגיאת "Internal Stack Overflow". לכן יש לוודא שקיים בסיס רקורסיה איכותי שאינו דורש המון קריאות רקורסיביות.

## העמסת פונקציות

ב-C++ יש מנגנון שנקרא "העמסת פונקציות" (Function Overloading). מנגנון זה מאפשר לנו להגדיר כמה פונקציות עם אותו שם אך עם ארגומנטים שונים. התועלת מכך היא שנוכל לקרוא לפונקציות בשמות קצרים וברורים כך שלא נצטרך להמציא שם חדש לכל מקרי קצה של פעולה מסוימת. כאשר מוגדרות לנו שתי פונקציות כאלו.

בקריאה לפונקציה הקומפיילר בוחר איזו מהפונקציות נכונה. כיצד הוא עושה זאת? בכמה שלבים:

1. מציאת כל הפונקציות עם השם המתאים.

2. מתוך קבוצה 1, מציאת כל הפונקציות עם מספר הפרמטרים המתאים.

3. מתוך קבוצה 2, מציאת הפונקציות עם סוג הפרמטרים **המתאים ביותר**.

בשלב השלישי, אם הקומפיילר לא מוצא בדיוק את הפונקציה שמקבלת ארגומנטים מסוג הפרמטרים שהכנסנו, הוא יעשה עליהם casting. לכן יכול להיות מצב שבו יתקבלו שתי פונקציות מתאימות. במצב זה הקומפיילר יחזיר שגיאת ambiguous (רב משמעות).

לדוגמה: בפונקציות הבאות אם נקרא לפונקציה כך power(2, 3.5) נקבל שגיאת קומפילציה, כי בשני המקרים צריך לבצע casting למספר אחד לכן רמת ההתאמה היא שווה.

int power (int a, unsigned int b) {

cout << "power of ints" << endl;

return b==0? 1: a\*power(a,b-1);

}

double power(double a, double b) {

cout << "power of reals" << endl;

return exp(b\*log(a));

}

המסקנה היא שצריך מאד להיזהר בהעמסת פונקציות, במיוחד כשמדובר בפרמטרים מספריים.

## ביטויי למדא

ביטוי למדא הוא ביטוי היוצר אובייקט שיש לו סוגריים עגולים (). השימוש העיקרי בביטויי למדא הוא כאשר יש פונקציה f שמקבלת כפרמטר פונקציה אחרת g. במקרה זה נוכל להעביר כפרמטר ל-f או שם של פונקציה כמו שאנחנו מכירים, או אובייקט מחלקה שיש לה אופרטור סוגריים עגולים, או ביטוי למדא. האפשרות השלישית היא במקרים רבים הנוחה ביותר, שכן מאפשרת לנו ליצור כמעין פונקציה זמנית, תוך כדי הקוד, שאינה כתובה במקום מיוחד.

כדי ליצור ביטוי למדא נרשום סוגריים מרובעים ריקים, לאחר מכן סוגריים עגולים שבתוכן הפרמטרים לביטוי למדא, ולאחר מכן סוגרים מסולסלים המכילים את הפקודות והערך חזרה מהביטוי למדא. לדוגמה, ביטוי למדא המקביל לפונקציה add.

[](int x, int y){return x+y;}

# מונחה עצמים

## הגדרה

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

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

הקפדה על תבנית ברורה בהגדרתה של מחלקה תורמת לבהירות הקוד. למשל, אם מחליטים כי המשתנים של המחלקה יופיעו לפני המתודות שלה ומקפידים על כך בכל המחלקות שמגדירים, בהירות הקוד גדלה. באופן דומה, הקפדה על סגנון נאה, והערות עקביות ומסודרות, כל זה גם תורם לבהירות הקוד.

## משתני המחלקה

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

**משתנים סטטיים**

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

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

**יצירת קבועים**

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

### כללי משתני מחלקה

* **להמעיט במספרם של משתנים מטיפוס בסיסי בהגדרת המחלקה** - מחלקה שיש בה מספר רב מאוד של משתנים מטיפוס בסיסי מעלה את השאלה שמא ניתן היה להגדיר במקום מחלקה אחת מספר מחלקות, ובכך להקנות לתוכנית רמה גבוהה יותר של מודולאריות.
* **לאתחל משתנים של המחלקה** - גם אם מאותחלים אוטומטית. אתחול מכוון מקנה בהירות רבה יותר לקוד.
* **נתינת שמות** - לתת למחלקות ולמתודות שמוגדרות בתוכן שמות שמעידים על פעולתן.

## בנאים - Constructors

בנאי הוא שיטה שתפקידה ליצור ולאתחל אובייקט חדש ולתת ערכים לכל משתני המחלקה שאינם סטטיים. שמו של הבנאי זהה לשם של המחלקה. יכולים להיות מספר בנאים שמקבלים פרמטרים שונים, כך שכל בנאי יוצר אובייקט לפי הפרמטרים שמקבל. פעולה זו נקראת "העמסת בנאים". בתוך הבנאי אסור שתהיה פקודת return ואפילו לא return void.

ClassName(type arg, …) {}

### בנאי ברירת מחדל

הוא בנאי שאינו מקבל פרמטרים כלל או בנאי שלכל פרמטר בו יש ערך ברירת מחדל. אם אנו לא מגדירים אף בנאי במחלקה, אזי הקומפיילר יוסיף באופן אוטומטי בנאי ברירת מחדל (Default Constructor) שקורא לבנאי ברירת המחדל של כל שדה באובייקט. אמנם אם הגדרנו בנאי עם פרמטרים ולא בנאי ברירת מחדל, אם ננסה לקרוא לבנאי ברירת מחדל נקבל שגיאת קומפילציה. לכן חשוב שאנו נגדיר תמיד בנאי ברירת מחדל.

ClassName() {…}

ClassName(int a=1, double b=2.0, …) {…}

## שיטות - Methods

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

יש להשתדל לקבוע מתודות שמאפשרות גישה למשתנים הפרטיים של המחלקה רק כאשר יש צורך בכך - כדי לאפשר גישה למשתני מחלקה עם הרשאת הגישה private, יש לכתוב לפחות שתי מתודות מתאימות: get ו-set. כתיבת שתי מתודות אלה לכל משתנה private גם כשאין בכך צורךפוגעת בבהירות של הקוד, ופוגעת במטרה שיש להרשאת הגישה private.

**העמסה -** **Overloading**

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

**שיטות סטטיות**

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

## הרשאות - Modifiers

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

1. **private** - ניתן לגשת אך ורק מתוך מחלקה.
2. **protected** - ניתן לגשת רק מהמחלקה עצמה, או ממחלקות מאותו package, או ממחלקות שיורשות ממנה.
3. **public** - ניתן לגשת מתוך כל מחלקה.

### עקרון הכימוס

ככלל, עקרון חשוב בתכנות מונחה עצמים הוא עקרון הכימוס. העיקרון אומר שיש להשתדל שכמה שיותר מידע יהיה פרטי כדי למנוע שימוש שגוי בו. תמיד נרצה להציג את הממשק למשתמש אך נחביא את המימוש.

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

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

## S.O.L.I.D

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

באופן עקרוני, קוד גרוע הוא קוד שבו:

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

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

**Single responsibility**

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

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

**Open for Extension, Closed for Modification**

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

**Liskov substitution principle**

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

**Interface segregation principle**

אין להכריח לקוח להיות תלוי בממשק שהוא אינו משתמש בו באופן מלא. יש לדאוג לממשקים מצומצמים, כך שיאלצו מחלקה שמממשת את הממשק לממש רק מה שרלוונטי לה. פתרון לבעיה זו הוא ממשקים שמממשים ממשקים.

**Dependency inversion**

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

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

## פולימורפיזם

פולימורפיזם (בעברית: "רב-צורתיות") הוא אחד מאבני היסוד של תכנות מונחה עצמים (יחד עם כימוס, הורשה והפשטה). פולימורפיזם מתרחש כאשר ישנה היררכיה של מחלקות הקשורות על ידי הורשה, ממשקים וכדומה. באופן כללי, פולימורפיזם אומר שבכל פעם שאנו מפעילים שיטה על אובייקט של מחלקה שהיא חלק מהיררכיה של מחלקות, תופעל השיטה הרלוונטית שמומשה עבור האובייקט שהפעיל שיטה זו.

שפת תכנות התומכת בפולימורפיזם נותנת למתכנת את היכולת לממש אלגוריתמים ומבני נתונים לשימוש כללי, ולגזור מהם צורות שימוש שונות בהתאם לעצמים ולנסיבות המשתנות, באופן מפורש או מובלע, ובכך להעלות את רמת ההפשטה של התוכנה.

# תהליכים ותהליכונים

## תהליך (Process)

תוכנית מחשב היא בעצמה רק אוסף פקודות, בעוד שתהליך הוא ההפעלה של אותן פקודות, כלומר תוכנית בהרצה. הפעלה של מספר מופעים של אותה תוכנה יגרום לעיתים קרובות למספר תהליכים נפרדים של התוכנה להיפתח בו זמנית. לכל תהליך יש מספר ייחודי הנקרא process ID או בקיצור PID.

מספר התהליכים שניתן להריץ במקביל הוא כמספר המעבדים במחשב, כל מעבד יכול להריץ תהליך אחד בלבד. אולם אפשר לקבל הדמיה של הרצת מספר תהליכים במקביל במעבד אחד על ידי מעבר מהיר בין תהליכים שונים. המעבר הזה נקרא גם שיתוף זמן (time sharing). המעבר בין הרצת משימות שונות נקרא החלפת הקשר (Context Switch), מכיוון שבו עוברים מהסביבה (ההקשר) של משימה אחת לזו של משימה אחרת. תהליך החלפת ההקשר נחשב לתהליך בזבזני מבחינת משאבי מערכת, ועל כן מערכות הפעלה מנסות לבצע אופטימיזציה בשימוש בהן. כדי לאפשר למשתמש להפעיל מספר תהליכים במקביל, מתבצעת על ידי סדרן התוכניות (Scheduler) חלוקת זמן מעבד בין התהליכים. חלוקת הזמן לא בהכרח שווה ויכולה להתבסס על היוריסטיקות שונות. לדוגמה, לתת העדפה (יותר זמן עיבוד) לתהליכים גדולים יותר.

בסיום ריצתו, התהליך מחזיר ערך יציאה (Exit status) שמאפשר לתהליך האב שיצר אותו לקבל מידע על סיום מוצלח של התהליך או לחלופין ערך שגיאה המעיד על סוג הכישלון שמנע מהתהליך להסתיים בהצלחה. תהליך שסיים את פעולתו אך עדיין קיים בטבלת התהליכים כדי שיתאפשר לתהליך האב לקרוא את ערך היציאה שלו, מכונה "תהליך זומבי".

מערכת ההפעלה שומרת על הפרדה בין התהליכים ומקצה להם את המשאבים בצורה שתבטיח עד כמה שניתן שתהליכים לא יפריעו אחד לשני. לכל תהליך יש מרחב כתובות מיוחד לו וכן משאבים נפרדים. עם זאת, מערכת ההפעלה מספקת כל מיני מנגנונים כדי לאפשר לתהליכים לתקשר אחד עם השני באופן בטוח.

### ייצוג בזיכרון

בזיכרון תהליך מורכב מארבעה חלקים:

1. Text Section - מכיל את הפקודה הנוכחית שמתבצעת שנמצאת בערך של ה-Program Counter.
2. Stack - מכיל את כל המידע הזמני, כמו משתנים מקומיים, פרמטרים לפונקציות וכתובות חזרה.
3. Data Section - מכיל את המשתנים הגלובליים.
4. Heap - מכיל את הזיכרון הדינאמי שהוקצה בזמן ריצה.

### PCB

ה-kernel שומר במרחב הזיכרון שלו (kernel space) טבלה של כל התהליכים הקיימים במערכת, הנקראת process table, כדי שיוכל לעקוב אחריהם. בטבלה זו לכל תהליך יש כניסה משלו המכילה מצביע אל מבנה נתונים הנקרא PCB (Process Control Block), המכילה את כל המידע הנחוץ כדי לנהל את התהליך. התוכנית היחידה שרצה ללא PCB היא מערכת ההפעלה עצמה ולכן אינה מוגדרת כתהליך. המידע ב-PCB כולל:

* מצב התהליך.
* PID - מספר הזהות הייחודי של התהליך.
* תזמון - עדיפות, הזמן שחלף מאז שנכנס למעבד, מצביע לתור התזמונים, ופרמטרים נוספים.
* זהות והרשאות - מידע אודות הבעלים של התהליך והרשאות ריצה.
* Program Counter - כתובת הפעולה הבאה שצריכה להתבצע על ידי התהליך.
* CPU registers - מידע זה מכיל את ערך האוגרים הנדרשים כדי לשחזר את פעולת התהליך לאחר החלפת הקשר.
* ניהול זיכרון - טבלת הדפים וגבולות הזיכרון המוקצה לתהליך.
* טבלה של כל הקבצים וההתקנים שנפתחו עבור התהליך, הנקראת file descriptors table.

## תהליכון (Thread)

תהליכון (Thread - בעברית חוט) הוא רצף של פעולות שמתבצעות באופן עצמאי במטרה לבצע משימה מסוימת. כל תהליכון הוא חלק מתהליך וכל תהליך מכיל לפחות תהליכון אחד. תהליך אחד יכול להכיל מספר תהליכונים שונים שרצים במקביל, מצב זה נקרא ריבוי תהליכונים (Multi-Threading). המטרה בריבוי תהליכונים היא להשיג מקביליות באמצעות חלוקת המשימה הכללית של התהליך למספר משימות קטנות המתבצעות במקביל באמצעות תהליכונים, בלי שנצטרך לחכות שאחד המשימות יגמרו כדי להתחיל את המשימה הבאה. לדוגמה, בתהליך המריץ תוכנת דפדפן, כל כרטיסייה יכולה להיות תהליכון נפרד. בנוסף, פעולת המעבר בין תהליכונים קצרה יותר מדפדוף בין תהליכים, מפני שאינו דורש החלפה של מרחב הכתובות. בסיום התהליך כל הזיכרון משתחרר וכל התהליכונים מתים.

גישה של מספר תהליכונים לאותו מקום בזיכרון עלולה להוביל לדריסת מידע ושגיאות (race condition). לפתרון בעיות אלו משתמשים בכלים המאפשרים לסנכרן ולתקשר בין תהליכונים. אמנם גם בסנכרון יכולים להיווצר בעיות כמו קיפאון (Deadlock), שבו שני תהליכונים נועלים כל אחד משאבים שהשני ממתין להם וכך לא מתקדמים לעולם, או הרעבה (Starving), שבו אחד התהליכונים הרצים לעולם לא מקבל את המשאבים הנחוצים לו.

## השוואה בין תהליכים לתהליכונים

ההבדל המרכזי בין תהליכים לתהליכונים הוא שתהליכונים אינם נפרדים אחד מהשני כמו תהליכים. תהליכים אינם חולקים זיכרון ומשאבים משותפים, אך לעומת זאת לתהליכונים יש זיכרון משותף, הם חולקים אותו code section ו-data section (משתנים גלובליים), וכן משאבים ממערכת ההפעלה כמו קבצים ו-signals. אמנם בדומה לתהליכים יש להם program counter, מחסנית ורגיסטרים שונים. נרצה להשתמש בתהליכונים כאשר נרצה לעשות פעולות מהירות על אותו מרחב זיכרון.

היתרונות של תהליכונים על תהליכים הם:

1. **היענות** - אם תהליך מחולק למספר תהליכונים, אם תהליך אחד משלים את ביצועו, ניתן להחזיר את הפלט שלו מייד.
2. **החלפת הקשר מהירה יותר** - החלפת הקשר בתהליך דורשת תקורה רבה מהמעבד, אולם מעבר בין תהליכונים מהיר יותר. בנוסף, גם יצירת תהליך לוקחת יותר זמן מאשר יצירת תהליכון, שכן צריך לשכפל את כל ה-PCB במלואו.
3. **שיתוף משאבים** - ניתן לחלוק משאבים כמו קוד, נתונים וקבצים בין כל התהליכונים בתהליך. אמנם לא ניתן לשתף מחסנית ורגיסטרים בין תהליכונים, מפני שלכל תהליכון מחסנית ורגיסטרים משלו.
4. **תקשורת** - התקשורת בין תהליכונים קלה יותר מכיוון שהם חולקים מרחב כתובות משותף. בין תהליכים לעומת זאת, צריך להשתמש בטכניקות מיוחדות לתקשר.
5. **תפוקה** - אם תהליך מחולק למספר תהליכונים, וכל תהליכון מבצע משימה אחת, אזי גדל מספר המשימות שהושלמו ליחידת זמן, וכך מגדיל את תפוקת המערכת.

## גישה למעבד

במחשבים בעלי מעבד בודד אין למעשה ריצה של קוד במקביל, אלא עיבוד "כמו מקבילי". תחושת המקביליות מתקבלת מכך שהמעבד מתחיל להריץ תהליכון אחד ולאחר זמן מסוים עוצר אותו, ואז מריץ תהליכון אחר וגם אותו עוצר לאחר זמן מסוים, ושוב מפעיל תהליכון אחר. מכיוון שהמעבד עושה פעולות אלו מאוד מהר וההחלפות מתבצעות בהפרשי זמן קצרים מאוד, המשתמש מקבל תחושה של מקביליות. קפיצה זו, המכונה החלפת הקשר (Context switch), מבצעת מעין צילום מצב של המעבד (אוגרים וכדומה), לפני נטישה של תהליכון מסוים, כדי לאפשר מעבר לתהליכון אחר וחזרה לתהליכון הקודם בהמשך. החלק במעבד שאחראי לפעולה זו נקרא "מתזמן" (thread scheduler).

המתזמן נותן זכות קדימה לתהליכונים שהגדרנו עם עדיפות גבוהה, כך שהם יופעלו יותר פעמים וגם יקבלו פרק זמן ארוך יותר לרוץ. אמנם במקרה של תהליכונים עם עדיפויות שוות, ההחלטה כמה פעמים וכמה זמן ירוץ כל תהליכון נתונה בידי מערכת ההפעלה. בכל מקרה, לנו כמתכנתים אין אפשרות לשלוט בסדר התהליכונים שהמעבד יריץ ובזמן המדויק שיפעל כל תהליכון, אלא הכל תלוי במערכת ההפעלה.

במחשבים עם מספר מעבדים, כמו רוב המחשבים כיום, העיבוד הוא מקבילי באמת, כאשר כל תהליכון רץ על מעבד אחר. במחשבים אלו היתרון של שימוש בתהליכונים משמעותית מהיר יותר.

1. **Race Condition**

כאשר מספר תהליכים עם משאבים משותפים רצים במקביל, ישנה אפשרות שהם ייגשו למשאב משותף וישנו את הנתונים במקביל, כך שתוצאת התהליך תלויה בסדר שבו התהליכים ניגשו למשאב זה. מצב זה יכול להוביל לאי עקביות בנתונים וכתוצאה מכך לתוצאות שגויות. מצב זה נקרא Race Condition.

ניתן למנוע מצב זה באמצעות טכניקה הנקראת סנכרון תהליכים, בה אנו מגדירים קטע קוד שיכול להוביל לבעיה זו "קטע קריטי", כך שבכל זמן נתון מאפשרים רק לתהליך אחד להיכנס לקטע קוד זה ולתמרן את הנתונים המשותפים.

## קטע קריטי

הוא רצף של פקודות והצהרות שצריך להתבצע באופן אטומי (כיחידה אחת) על ידי תהליך אחד בכל פעם. כדי להגדיר קטע קריטי יש להגדיר בתהליך ארבעה חלקים:

1. Entry Section - מחליט איזה תהליך בודד להכניס לקטע הקריטי.
2. Critical Section - זהו החלק האטומי שבו מתבצע השינוי של המשאב המשותף.
3. Exit Section - מאפשר לתהליך האחר הממתין בקטע הכניסה להיכנס לחלק הקריטי. הוא בודק שהתהליך האחרון שנכנס סיים את ביצועו בקטע קריטי דרך חלק יציאה זה.
4. Remainder Section - החלקים האחרים בקוד שמבוצעים לאחר הקטע הקריטי.

ישנם מספר גורמים שיש להקפיד עליהם כאשר ניגשים לפתרון בעיית הקטע הקריטי כדי לפתור את הבעיה ולא ליצור בעיות אחרות.

1. מניעה הדדית (Mutual exclusion) - כאשר הקטע הקריטי רץ, יש למנוע שימוש במשאב המשותף על ידי גורם אחר שיכול להפריע לפעולה התקינה של הקטע הקריטי. כיוון שכך, כדאי להגדיר קטע קריטי רק קטע קוד הכרחי כדי שהשימוש במשאב לא יהיה חסום ללא צורך.
2. התקדמות (Progress) - כאשר מונעים שימוש של גורמים נוספים במשאב, יש לוודא שלאחר שהקטע הקריטי סיים לרוץ, לא תישאר המניעה על השימוש במשאב.
3. המתנה מוגבלת (Bounded Waiting) - יש לוודא שכל גורם שמבקש לרוץ, יעשה את זה תוך מספר מוגבל של קטעים קריטיים שירוצו לפניו.

יש מספר מודלים של סינכרון:

1. Mutex
2. מנעול
3. Semaphore
4. משתני מצב (לדוגמה, פטרסון).

## אלגוריתם פטרסון

אלגוריתם בגישת התוכנה לפתרון בעיית הקטע הקריטי עבור 2 תהליכים. האלגוריתם משתמש בשני משתנים משותפים, flag ו-turn. ערך חיובי עבור flag[n] מראה על כך שתהליך n מעוניין להיכנס לקטע הקריטי שלו. הכניסה לקטע הקריטי ניתנת לתהליך P0 אם תהליך P1 לא מעוניין להיכנס לקטע הקריטי שלו או אם P1 נתן קדימות לתהליך P0 על ידי השמת הערך 0 במשתנה turn.

bool flag[0] = false;

bool flag[1] = false;

int turn;

P0: flag[0] = true;

p1\_gate: turn = 1;

**while** (flag[1]==true && turn == 1) {

*// busy wait*

}

*// critical section*

...

*// end of critical section*

flag[0] = false;

P1: flag[1] = true;

P1\_gate: turn = 0;

**while** (flag[0]==true && turn == 0) {

*// busy wait*

}

*// critical section*

...

*// end of critical section*

flag[1] = false

האלגוריתם עובד בשיטה של המתנה פעילה (busy waiting) כלומר, כאשר הקטע הקריטי לא סיים לרוץ, כל זמן שלא הגיע תור התהליך השני להריץ את הקטע הקריטי הוא יהיה תקוע בלולאת ה-while וידרוש זמן מעבד. ניתן ליישם רעיון דומה עבור N תהליכים.

## מנעול

מתחזקים משתנה משותף שמשמש כמעין מנעול. משתנה זה יכול לקבל את הערכים 0 או 1, כאשר 0 מציין לא נעול ו-1 מציין נעול. בעת הגעה לקטע קריטי, מבקש התהליך לבצע נעילה. במידה והמנעול פנוי, הנעילה תצליח והתהליך יוכל להתקדם. בשלב זה, כל תהליך אחר שיגיע לקטע הקריטי ויבקש לבצע נעילה - יאלץ להמתין עד לשחרור המנעול על ידי התהליך הנועל. עם יציאת התהליך הנועל מהקטע הקריטי, הוא יבצע שחרור של המנעול, ותהליך אחר שביקש לבצע נעילה יצליח בנעילה ויכנס לקטע הקריטי.

ישנם שני סוגי מימוש למנעול, מנעול הוגן בו כל מי שמבקש לבצע נעילה נכנס לתור המתנה ומתקדם לפי מיקומו בתור, ומנעול לא הוגן שבו באופן לא צפוי מתקבלת ההחלטה מי ישתחרר, דבר שעלול לגרום להרעבה אם מישהו מחכה אך כל פעם לפני שמישהו יוצא מהקטע הקריטי מישהו חדש מנסה לשחרר את המנעול ומקבל קדימות על פני האחר

## סמפור (Semaphore)

מטרתו היא לבצע סנכרון ולפתור את בעיית ההמתנה העסוקה. בנוסף, קיימת סכנה כי בעת ביצוע קטע הקוד לכניסה לקטע הקריטי, אשר מורכב ממספר פקודות, "ייחטף" המעבד מהתהליך וכך עלול לקרות מצב בו המשך ביצוע התוכנית מסתמך על נתונים לא מעודכנים.

סמפור: מעין דגל (flag) אשר כדי לשנות את ערכו התהליך נדרש לבקש זאת ממערכת ההפעלה. במידה והדגל כבר "נתפס" על ידי תהליך אחר, מערכת ההפעלה תעביר את התהליך ל"מצב שינה", ו"תעיר" אותו על ידי משלוח פסיקה ברגע שהדגל "משוחרר" על ידי התהליך שתפס אותו. בצורה זו מתאפשר סנכרון ללא בזבוז זמן מעבד, וכן מובטח לנו כי מערכת ההפעלה מפקחת על מצב הדגל ושינוי של ערכו מבוצע בצורה אטומית, כלומר לא ניתן לחטוף לתהליך את המעבד בזמן שינוי ערך הדגל. אמנם אם הסמפור איננו "הוגן", משחרר לפי הסדר בו ניסו לתפוס אותו, קיימת בעיה של הרעבה.

הסמפור הוא משתנה היכול לקבל ערכים שלמים לא שליליים. על הסמפור מוגדרות שתי פעולות אטומיות:

* אם ערך הסמפור גדול מ-0 הקטן אותו ב-1 והמשך, אחרת המתן עד שיהיה גדול מ-0, ואז הקטן אותו ב-1 והמשך.
* הגדל את ערך הסמפור ב-1.

## הרעבה (Starving)

זהו מצב שבו thread אינו מקבל זמן ריצה שהוא צריך מהמעבד, או שאינו מקבל את המוניטור של האובייקט לו הוא מחכה. במצב זה ל-thread ייקח זמן רב עד שיבצע את הפעולה אותה הוא אמור לבצע. הפתרון לכך לחלק את המשאבים של המעבד בצורה חכמה יותר, ניתן לעשות זאת לדוגמא על ידי שינוי סדר העדיפות של ה-thread ובכך להעלות את חשיבותו לתוכנית, אפשרות נוספת היא להגדיל את משאבי המעבד. בנוסף, לשים לב שמוניטור של האובייקט לו הוא מחכה אינו נתקע ב-thread אחר.

## קיפאון (Deadlock)

זהו מצב שבו שני תהליכונים תקועים מפני שכל אחד ממתין לשני שיעיר אותו, או שכל אחד נועל מוניטור שהשני מחכה לו. במצב זה הרצת התוכנית תיתקע והדרך היחידה לסיים אותה הוא מבחוץ. הפתרון לבעיה זו הוא פשוט תכנון נכון של ה-thread, ולשים לב שבכל רצף הפעולות שלו אין שום thread אחר שמונע ממנו לסיים פעולתו.

להשלים: livelock, mutex, dekker, semaphore, לפתור deadlock.

# חריגות

## הגדרה

ב-Java קיים מנגנון של חריגות. מנגנון זה מאפשר לנו להחליט מה לעשות כאשר מישהו משתמש בתוכנית שלנו בצורה לא תקינה. נוכל לעצור את התוכנית ולזרוק למשתמש הודעה שמסבירה לו מדוע התוכנית לא יכולה לבצע מה שהוא מנסה לעשות, או שנוכל להבין מה הטעות שלו ולטפל בה בלי שהוא ישים לב. באותו אופן כאשר אנו משתמשים בקוד המובנה ב-Java או בספריות חיצוניות אנו עלולים ליצור שגיאות/חריגות. במקרה זה מישהו הגדיר מה לעשות עבור כל מצב כזה.

## סוגי חריגות

כאשר מתרחשת שגיאה/חריגה נוצר אובייקט מתאים המייצג שגיאה/חריגה זו. ישנה היררכיה של ממשקים ומחלקות שמטרתם ליצור אובייקטים אלו. בראש ההיררכיה נמצא ממשק Throwable המייצג אובייקט שניתן "לזרוק" אותו למשתמש ולסיים את התוכנית. הממשקים והמחלקות היורשות מ-Throwable מטרתם לייצג שגיאות וחריגות ספציפיות כדי שהמשתמש יוכל להבין מהי הבעיה. סוגי הבעיות הן:

1. **Errors** - אלו הן שגיאות שאינן תלויות במתכנת או במשתמש, כמו לדוגמא חוסר מקום בזיכרון או הצפה שלו. שגיאות אלו מתגלות בזמן ריצה. המתכנת אינו צריך לטפל בשגיאות אלו כי בדרך כלל אין לו מה לעשות נגדן.
2. **Checked Exceptions** - אלו הן חריגות שנבדקות בזמן הקומפילציה עוד לפני שהתוכנית מתחילה לרוץ. דוגמא לפקודות שיכולות לזרוק חריגות מסוג זה הן פקודות קלט ופלט על קבצים. כאשר הקומפיילר מזהה שפקודות מסוג זה קיימות בתוכנית, הוא בודק שהמתכנת אכן טיפל במקרים בהם יכולים להיזרק חריגות מסוג זה. לא ניתן להתעלם מהם, במידה והמתכנת לא טיפל בחריגות אלו התוכנית לא תקומפל ותיזרק חריגה מתאימה.
3. **Unchecked Exception** - אלו חריגות שאינן נבדקות בזמן קומפילציה אלא מתגלות בזמן ריצה, ולכן נקראות "חריגות זמן ריצה". חריגות אלו מתחלקות לשני סוגים: חריגות מתכנת וחריגות משתמש. חריגות מתכנת אלו הן חריגות שהמתכנת לא שם לב אליהן כמו: חריגות חישוביות, גישה לערך null, או גישה לתא לא מוגדר במערך. חריגות משתמש הן בדרך כלל חריגות קלט לא תקין.

ניתן גם ליצור חריגה על ידי הגדרת מחלקה שמממשת את ממשק Exception. לכל אובייקט מסוג Throwable יש שיטות שמוגדרות עליו, כמו הדפסה של מסלול החריגה, מחרוזת מייצגת את השגיאה וכו'. ניתן לקרוא עוד בקישור: <https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>.

## עקרון הכרז או טפל

כאשר תופסים חריגה ניתן לטפל בה בשני דרכים:

1. **להכריז** - לעצור את התוכנית ולזרוק את אובייקט החריגה המציין מה הייתה הבעיה, באיזה קובץ, ובאיזה שורה. דרך זו בעצם אומרת - זיהיתי שיש כאן בעיה אך זה לא האחריות שלי לטפל בה, אלא אני רק מודיע עליה, ומי שמקבל את ההודעה יחליט מה לעשות עם מידע זה.
2. **לטפל** - לא לעצור את התוכנית, אלא לטפל בחריגה. הטיפול בחריגה יכול להיות לנסות לבצע אותה פעולה שוב, או לבצע פעולות אחרות שמונעות את הבעיה. דרך זו אומרת - זיהיתי את הבעיה ואני אחראי לטפל בה כך שלא תפגע בתקינות של התכנית.

## זריקת חריגה

### throw

כאשר רוצים לזרוק חריגה נשתמש בפקודה throw Exception(), המשתמשת במילה השמורה throw ולאחריה מצביע לאובייקט מסוג Exception או מחלקה שיורשת מ-Exception. ניתן להוסיף כפרמטר גם מחרוזת String שמסבירה בפירוט על החריגה.

throw new ArithmeticException("can't divide by zero");

ביצועה של פקודה זו נעשה בדרך כלל כחלק ממשפט if, ובעצם הפעלתה נזרקת למעשה החריגה.

### throws

כאמור לעיל ישנם חריגות שאם הן נזרקות מחייבות אותנו לטפל בהם. אם מתוך שיטה כלשהי נזרקה חריגה מסוג זה, בין אם מתוך פקודה throw או שהופעלה שיטה אחרת שזורקת חריגה, אזי נוכל לטפל בחריגה בשתי דרכים. דרך אחת לשפל בחריגה כמו שנלמד בהמשך באמצעות בלוקים של try ו-catch. דרך שנייה להגדיר שחריגות מסוימות שיתפסו בשיטה זו מיד ייזרקו הלאה למי שהפעיל את השיטה. נוכל לעשות זאת על ידי הוספה בסוף הכותרת של השיטה (לאחר הפרמטרים) את המילה השמורה throws ולאחריה מצביע לאובייקט מסוג Exception או מחלקה שיורשת מ-Exception. לאחר פעולה זו אם תיתפס בשיטה כל חריגה מהסוג שהגדרנו בכותרת, או חריגה שיורשת מהסוג שהגדרנו בכותרת, אזי חריגה זו תיזרק אוטומטית הלאה. ניתן להגדיר מספר חריגות שהשיטה זורקת על ידי הוספת פסיק בין שמות החריגות.

public void f(int a) throws ArithmeticException, NullPointerException{…)

היתרונות של דרך זו הוא שכך הקוד קריא וברור יותר, וגם ניתן לתעד שהשיטה זורקת חריגה ב-javadoc כך שזה יהיה ברור יותר למשתמשים אחרים במחלקה. שיטה שבכותרתה יצוין שזורקת חריגה מסוג האב Exception תהיה מסוגלת לזרוק כל חריגה אפשרית

אם מבצעים overriding לשיטה שמגיעה בהורשה, לא ניתן לקבוע throws בשיטה החדשה שתזרוק שגיאות שלא הוגדרו בשיטה המקורית, אך ניתן להגדיר שזורקת חלק מהחריגות שהוגדרו בשיטה המקורית.

## טיפול בחריגה

טיפול בחריגה יעשה באמצעות המילים השמורות try, catch, ו-finally.

### try

זוהי מילה שמורה שלאחריה פותחים בלוק שבתוכו נמקם את שורות הקוד שיכולות לעורר ולזרוק חריגה. החריגה יכולה להיזרק או מתוך פונקציה שהקריאה להפעלתה ממוקמת בתוך אותו בלוק, או על ידי הפקודה throw שמופעלת בתוך אותו בלוק. המשמעות של פעולה זו היא שהקוד בתוך בלוק זה הוא מוגן. מערכת ההפעלה תפעיל קוד זה, וכאשר תזהה כי יש חריגה, מיד תעבור לבלוק של catch המתאים שיתמודד עם חריגה זו. לאחר שנזרקת חריגה וטיפלנו בה לא ניתן לחזור למקום שממנו נזרקה החריגה בבלוק ה-try.

try{

// Critical code

}

### catch

לאחר בלוק ה-try נכתוב בלוק catch שתופס חריגה מסוימת שנזרקה מבלוק ה-try. ניתן לכתוב כמה בלוקים של catch עבור כל סוג חריגה שייזרק מבלוק ה-try, כך שכל בלוק תופס חריגה שונה.

catch זוהי מילה שמורה שמקבלת פרמטר מסוג Exception או מחלקה שיורשת מ-Exception, בדומה לפרמטר שמכניסים לפונקציה. סוג החריגה בפרמטר הוא סוג החריגה שיתפוס בלוק ה-catch הנוכחי. לאחר כתיבת הפרמטר נפתח בלוק ובו נכתוב מה אנחנו רוצים שהתוכנית תעשה כדי להתמודד עם החריגה שנזרקה מבלוק ה-try. ניתן לבצע פעולות על אובייקט החריגה באמצעות השם שנתנו לפרמטר.

catch(FileNotFoundException ex) {…)

ניתן גם להגדיר שבלוק אחד יתפוס ויטפל בכמה סוגי חריגות באמצעות סימן ' | '.

catch(IOException|FileNotFoundException ex) {…)

אחרי שבלוק catch מתבצע, התכנית ממשיכה להתבצע מהנקודה שאחרי בלוק ה-catch (ואם יש בלוק finally אז אחריו), לא ניתן לחזור אל המקום שבו נוצרה החריגה בבלוק ה-try ולהמשיך משם. כאשר בתוך בלוק ה-catch מופיעה אחת מהפקודות: return, throw, break, continue, התכנית תבצע תחילה את בלוק ה-finally (אם קיים) אך לא תמשיך מהנקודה שאחריו.

אם חריגה שנזרקה לא מטופלת באף בלוק catch, אזי ב-default handler שקיים ב-Java יטפל בה, בדרך כלל התוכנית תסתיים עם הודעת שגיאה על המסך.

### finally

מילה שמורה אשר תופיע לאחר כל בלוקי ה-catch, ומיד אחריה בלוק של פקודות אשר יתבצעו בכל מקרה, בין אם נזרקה חריגה מבלוק ה-try או לא. השימוש העיקרי של בלוק זה הוא לבצע פקודות שמטרתן סגירת המשאבים של התכנית.

public static void main(String args[]) {

int a[] = new int[2];

**try** {

System.out.println("Access element three :" + a[3]);

} **catch** (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception thrown :" + e);

} **finally** {

a[0] = 6;

System.out.println("First element value: " + a[0]);

System.out.println("The finally statement is executed");

}

}

# בדיקות

## הגדרה

בדיקות תוכנה הן מכלול תהליכים, שיטות, פעילויות וכלים המלווים את פיתוח המערכות והמוצרים לאורך מחזור החיים, מתוך כוונה להעריך את איכות התוכנה והוכחת תקפות להתאמת התוצרים לדרישות הלקוח, למפרטים הטכניים, לתקנים ולנהלים מחייבים. בדיקות תוכנה מהוות חלק אינטגרלי מתהליכי הנדסת תוכנה, ומוסיפות נדבך בעל חשיבות מכרעת להבטיח את איכות המערכת.

הבדיקות מבוצעות בראשונה על ידי צוות הפיתוח עצמו בסביבת הפיתוח. לאחר מכן צוות הבדיקות מוסיף את הבדיקות שלו ומודיע לצוות הפיתוח על התוצאות. לבסוף, בשלבי הטמעת המערכת גם הלקוחות בודקים את המערכת. נעמוד על ההבדלים בין שלושת סוגי הבדיקות בהמשך. בכדי לייעל את תהליך הבדיקות, ניתן להשתמש בכלים שונים לניהול ולביצוע הבדיקות.

מטרות שלב הבדיקות הם:

* Validation - האם המערכת אכן עובדת כמו שרצינו?
* Verification - האם המערכת אכן תואמת לדרישות ולמפרטים שהציבו לנו? כולל בדיקות פונקציונאליות ובדיקות לא פונקציונאליות.
* לבדוק האם המערכת איכותית - מינימום שגיאות ועמידה בדרישות ביצועים ועומסים.
* מטרה נוספת של תהליך הבדיקות היא לצמצם ככל הניתן את עלויות אי האיכות. לפי העיקרון של בוהם, ככל שנאתר את השגיאות בשלבים מאוחרים יותר, תגדל עלות התיקון.

1. **מה בודקים**

**בדיקות מסירה**

אלו הן מכלול הבדיקות הנעשות בתהליך הפיתוח טרם מסירת המוצר לבדיקות הקבלה של הלקוח. נמצאות באחריות צוות הפיתוח והבדיקות. אפשר לחלק לארבעה שלבים:

* בדיקות יחידה: בודקים כל פונקציה בקוד בנפרד.
* בדיקות אינטגרציה (שילוב): כיצד מחלקות או פונקציות עובדות ביחד.
* בדיקות ממשקים: בדיקות בין ממשקים במערכת או מחוצה לה.
* בדיקות מערכת: סוגים שונים של בדיקות למערכת כולה: פונקציונאליות, יעילות, עומסים, עיצוב וכו'.

**בדיקות קבלה**

אלו הן בדיקות כוללות של המערכת לפני הפעלה, במטרה לוודא עמידה בדרישות והתאמת המערכת לצרכי הלקוח. נמצאות באחריות הלקוח/משתמש ומבוצעות בסביבת הלקוח או בסביבת pre-production.

1. **שיטות לבדיקה**

כל הבדיקות מבוססות על הכנסת קלט, קבלת פלט ובדיקה על הפלט.

**בדיקות קופסה לבנה**

סוג בדיקות מעמיק המתמקדות בזרימת המידע הפנימית בתוך המערכת בין רכיבי התוכנה/חומרה ובדיקת זרימת הנתונים בתוך התהליכים. בבדיקות אלו נדרש ידע אודות המימוש הפנימי של המערכת ולכן בדרך כלל מבוצעות על ידי המפתחים בלבד. לא צריך הסתכלות כוללת על המערכת אלא ספציפית לקטע קוד שאנו בודקים. דוגמה לבדיקות מסוג זה הן בדיקות יחידה ובדיקות אינטגרציה.

**בדיקות קופסה שחורה**

שיטת הבדיקות הרלוונטית לרמות הבדיקה שלאחר בדיקות השילוב (בדיקות מערכת ומעלה). מוודאים שהמערכת מממשת את הדרישות שהציבו לה. אין עניין במבנה הפנימי של הקוד עצמו. בבדיקות קופסה שחורה יש חשיבות אך ורק ל-"מה" ולא ל-"איך", כלומר חשובה רק התוצאה ולא הדרך שבה המערכת הגיעה אליה. מבוצעים על ידי צוות הבדיקות והלקוחות.

**בדיקות קופסה אפורה**

בדיקות ביניים בין קופסה לבנה לשחורה. הן כן ניגשות לקוד אך לא בצרה מעמיקה. מטרתן היא בעיקר לבדוק תהליכים גדולים ומקרי קצה. מבוצעים על ידי צוות הבדיקות. דוגמה לבדיקות מסוג זה הן בדיקות ממשקים.

1. **עקרונות הבדיקות**

* מטרת הבדיקה היא להכשיל את המערכת.
* לא לבדוק רק מה שבמפרט אלא גם מקרי קצה.
* בדיקות רגרסיה - לוודא שהבדיקה לא פוגעת בדברים קודמים שעשינו.
* תוצאות הבדיקות צריכות להיות מושוות עם כלים מתאימים. לדוגמה, אם בדיקה צריכה להחזיר ערך מספרי נבדוק זאת עם כלי חישוב ולא ידנית.
* יש לשלב בין בדיקות ידניות ואוטומטיות.
* צריך לוודא בצורה אמפירית שאסטרטגיית הבדיקה היא נכונה לקוד שאנו בודקים.
* להגדיר קריטריונים כיצד להעריך את תוצאות הבדיקות. הקריטריון המרכזי הוא כמה שגיאות מתגלות כפונקציה של זמן.

1. **רמת חומרה של באגים**

מאוד חשוב להגדיר רמות חומרה של שגיאות בבדיקות. דירוג השגיאות עוזר לכולם להבין מתי יש שגיאה קלה ומתי יש שגיאה חמורה, וכן עוזר למפתחים לתעדף שגיאות חמורות על פני קלות.

|  |  |  |
| --- | --- | --- |
| **תיאור** | **שם** | **חומרה** |
| - מונעת ביצוע של רכיבים מרכזיים במערכת.  - מסכנת את הבטיחות או הביטחון.  - תקלה מהותית בנושא שהוגדר בעל חשיבות גבוהה. | Highly Critical | 5 |
| - משפיעה לרעה על ביצוע של רכיבים מרכזיים במערכת.  - בעיית משאבים או ביצועים המשפיעה על תפקוד המערכת.  - כל תקלה שברור שהמשתמש הסביר לא יסכים לקבל. | Critical | 4 |
| - תקלה בולטת במיוחד לעיני הלקוח (גם אם בפועל ההשפעה שלה פחותה).  - בעיית משאבים או ביצועים חמורה אך עדיין מאפשרת שימוש. | Major | 3 |
| - תפקוד של המערכת אינו פועל אבל יש דרך סבירה לעקוף זאת.  - סדר פעולות הנדרש מהמשתמש אינו טוב: לא אינטואיטיבי, מסורבל, מבלבל וכו'.  - תקלות בממשק הגראפי לא אינטואיטיבי, טקסט או נתונים קשים לקריאה או חתוכים, שגיאות כתיב וכו'. | Medium | 2 |
| - תקלות קלות אחרות בממשק הגראפי או בהיבטים אחרים. | Minor | 1 |

1. **סוגי בדיקות מערכת**

* **בדיקות פונקציונליות** - לאימות פעילות המערכת. בדיקות אלו מבוססות על מסמך הדרישות ומסמך האפיון ומטרתן לבדוק כי המערכת עושה את מה שהיא צריכה ולא עושה את מה שאינה צריכה לעשות. יש לעשות בדיקות נקודתיות לפונקציות מקומיות ובדיקות כלליות של המערכת. הקלטים לבדיקות יהיו חוקיים ולא חוקיים לוודא שאכן המערכת מגיבה כצפוי בכל תרחיש.
* **בדיקות לא פונקציונליות** - בדיקות אלו בודקות איך פועלת המערכת וכוללות בדיקות עומסים, ביצועים, שימושיות וסוגי בדיקות רבים נוספים (שחלקם מפורטים למטה).
* **בדיקות תצוגה** - נוודא שממשק המשתמש עומד בדרישות שהוגדרו לו מבחינת פונקציונאליות, מבחינה ויזואלית ומבחינת סטנדרטים של עיצוב. בסיום הבדיקה נוכל לומר האם ממשק המשתמש מכיל את כל המרכיבים שהוא צריך להכיל והאם הם מתפקדים כיאות. יש למפות את כל מסכי המערכת ובכל מסך ממפים את כל השדות, פקדים, טקסטים ועוד. לאחר המיפוי ייבדק על רכיב בצורה פרטנית מול ההגדרות באפיון.
* **בדיקות שימושיות** - להעריך האם ממשק המשתמש יעיל, אסוציאטיבי, קל ונוח לשימוש, כך שיוכל להשתלב בקלות בתהליך עבודת המשתמשים ולא יכביד עליהם בשל בעיות עיצוב של ממשק לא נוח ולא ברור. בדיקת סובלנות של המערכת לטעויות המשתמש, הצגת הודעה במקרה של שגיאה ואפשרויות לתיקון. יש להתאים בדיקות מסוג זה לסביבת הלקוח כמה שניתן.
* **בדיקת ביצועים ועומסים** - עמידת המערכת בעומסים ובזמני תגובה הנדרשים. בדיקות אלו מאבחנות את יכולת המערכת להמשיך ולתפקד בצורה צפויה תחת עומסי עבודה שונים. יש להפעיל את המערכת תחת עומס ממוצע ועומס שיא ובחינת זמני הביצוע של תהליכים מרכזיים במערכת.
* **בדיקת רגרסיה** - בדיקות רגרסיה הן בדיקות רוחביות המבוצעות בכל סבב בדיקות, לקראת שחרור גרסת מערכת חדשה ועם סיום התקנת גרסה באתר הייצור, בכדי לאמת אי פגיעה בתקינות המערכת בעקבות תיקון / שינוי קוד. יש להריץ תרחישי בדיקות שנכתבו עבור תהליכים שפותחו ונבדקו בעבר.
* **בדיקות מסירה** (Factory Acceptance Test - FAT) - בדיקות מסירה של מערכת המפותחת ע"י גורם חיצוני. זוהי הבדיקה הסופית ואחרונה לפני בדיקות הקבלה אשר מבוצעות ע"י הלקוח.
* **בדיקות קבלה** (Acceptance Test Procedure - ATP) - בדיקות קבלה מבוצעות על ידי הלקוח ונציגיו, לאחר סיום מוצלח של בדיקות המערכת. מטרתן היא לוודא התאמת המערכת לצרכים ולדרישות שהוגדרו במשימת הפרויקט.