**העמסת אופרטורים (operator overloading)**

 **מאפשר למתכנת לכתוב קוד שבו ביטוי אריתמטי יקרא לקוד מתכנת.**

**למשל**

**Matrix a,b,c;**

**c = a\*b;**

**יקרא לקוד מתכנת של כפל מטריצות.**

**היתרון המעשי ביותר של האמצעי הזה:**

**קל יותר לתכנת נוסחאות מורכבות.**

**העמסת אופרטורים אפשרית בתנאים מסוימים:**

* **האופרטור חייב להיות קיים בשפה.**
* **זה צריך להיות מתוך רשימה שך אופרטורים נתונים להעמסה.**
* **לפחות אחד האופרנדים הוא מסוג user defined כלומר מוגדר ע"י המתכנת.**
* **ב-C# העמסת "[]", "()" ו-"=" נופלים לקטגוריות מיוחדות (indexers, explicit, implicit).**
* **ב-C# העמסת אופרטורים בעזרת מילת המפתח operator תמיד static.**
* **לא ניתן לתכנת את אופרטורי ההסבה לקבל אופרנד מאותו סוג. השמה של משתנה מחלקה למשתנה מחלקה מאותו סוג יהיה תמיד השמה של כתובת.**
* **כמעט תמיד העמסת אופרטור מקצה מופע חדש של המחלקה ומחזיר אותו ולא מכניס שינוי באופרנדים (פרמטרים).**

 **נציגים Delegates**

* **מבחינה מעשית מקביל למושג ב-C של פוינטר לפונקציה.**
* **ב-Java עם מחיקת הפוינטרים נעלם גם הפוינטר לפונקציה. זה עושה בעיות.**
* **ב-C# החזירו את הפוינטר לפונקציה בקטגוריה של delegate.**
* **delegate הוא מעין typedef של מתודות.**
* **ב-C# (כמו ב-C) שם של מתודה (ללא סוגריים) היא הכתובת שלה.**
* **הפקודה**

 **public delegate double func(double x);**

**מגדירה מעין typedef בשם func של פוינטר לפונקציה שמקבלת double ומחזירה double.**

**מרחבי שמות namespaces:**

**מניעת התנגשות שמות גלובליים, בעיה רצינית בפיתוח תוכנות ע"י קבוצות פיתוח שונות.**

**שימוש במרחבי שמות מחייב יותר הקלדה, אפשר לקצר ע"י using.**

**זה עשוי מחדש לגרום להתנגשות שמות (תוכנית דו משמעית).**

**הורשה ב-C#**

**הורשה ב-C# נראה ככה:**

 **class NewClass: A, I1, I2**

**המקביל ב-java :**

**class NewClass extends A implements I1, I2**

**כמו ב-Java שפת C# אינו תומך בהורשה של יותר ממחלקה אחת.**

**ב-C# המילה base מחליפה את super של Java.**

**קריאה לבנאי של האב מתוך הבנאי של הבן – בכותרת.**

**הפקודה**

 **public point2(double xv, double yv):base(xv, yv)**

**ב-Java נכתבת:**

 **public point2(double xv, double yv)**

**{**

 **super(xv, yv);**

**סוגי דריסה בהורשה:**

**ב-C# כמו ב-C++ אבל בניגוד ל-Java יש שני סוגי דריסה:**

1. **דריסה סטטית**
2. **דריסה וירטואלית**

**ב-Java יש אך ורק דריסה וירטואלית.**

**סוג הדריסה הסטטית בדרך כלל רק יכולה לעשות צרות.**

**היתרון היחיד של דריסה סטטית היא יעילות (חסכון בגישה לזיכרון).**

**אם כותבים מתודה שעשויה להידרס בהורשה תמיד תגדירו אותה וירטואלית.**

**אחד המקרים של עקרון הפולימורפיזם:**

**קוד שנכתב עבור מחלקת האב, יכול לרוץ על מופע של הבן, ומתאים את עצמו למופע של הבן.**

**ב-Java זה תמיד נכון.**

**ב-C# ו-C++ זה נכון רק בדריסה וירטואלית.**

**מה קורה כאשר:**

* **משתנה מסוג האב מצביע בפועל על מופע של הבן, ו-**
* **קורא למתודה קיימת באב ונדרסת בבן?**

**ב-Java יקרא תמיד המתודה של הבן.**

**ב-C# יקרא:**

* **אם מדובר בדריסה סטטית המתודה של האב**
* **אם מדובר בדריסה וירטואלית המתודה של הבן**

**ב-C# מתודה היא וירטואלית אם בהכרזה שלה (במחלקת האב) מופיעה מילת המפתח virtual.**

**הדריסה תהיה וירטואלית אם בבן מופיעה מילת המפתח override.**

**המניע של דריסה סטטית היא כנראה אבטחה: לא לאפשר דריסה של מתודות בקוד ספריה.**

**קוד שנכתב עבור מחלקת האב ומכיל קריאת למתודות לא וירטואליות הנדרסות סטטית בבן, לא מתאימות את עצמן לריצה על מופע של מחלקת הבן.**

**במקרה כזה עקרון הפולימורפיזם לא תופס.**

**עקרון הפולימורפיזם הוא "קוד מתפרש בכמה צורות אפשריות בהתאם להקשר שבו הוא נקרא".**