מרחב השמות
(namespace)

אוהד ברזילי
אוניברסיטת תל אביב

מה בתוכנית?

- לקוח וספק במערכת תוכנה
- ממשקים
- הכרת מרחב השמות
- מניעת תלות פוטנציאלית בין רכיבים במערכת ע"י עיצוב חלופי של מבנה הממשקים

לקוח וספק במערכת תוכנה

- ספק (supplier) – הוא מי שקוראים לו (לפעמים נקרא גם שרת, server)
- לקוח (client) הוא מי שקרא לספק או מי שמשתמש בו (לפעמים נקרא גם משתמש, user)
- דוגמא:

```
void do_something()
{
  // doing...
}

main()
{
  do_something();
}
```

- בדוגמא זו הפונקציה main היא **לקוחה** של הפונקציה ()do_something
- do_something היא **ספקית** של main

לקוח וספק במערכת תוכנה

- הספק והלקוח עשויים להיכתב בזמנים שונים, במקומות שונים וע"י אנשים שונים ואז כמובן לא יופיעו באותו קובץ

supplier.cpp

```
void do_something()
{
  // doing...
}
```

client.cpp

```
main()
{
  do_something();
}
```

לקוח וספק במערכת תוכנה

- מנקודת מבטו של הלקוח קוד המקור של הספק עשוי לא להיות זמין כלל

supplier.o

```
#@!!!>>??%^#&@
```

client.cpp

```
main()
{
      do_something(???);
}
```


לקוח וספק במערכת תוכנה

- כדי לתקשר בין הספק והלקוח עליהם להגדיר ממשק (interface, מנשק) בינהם

בצד הלקוח

בצד הספק

supplier.h

```
void do_something();
```

```
#include "supplier.h"
void do_something()
{
      // doing...
}
```

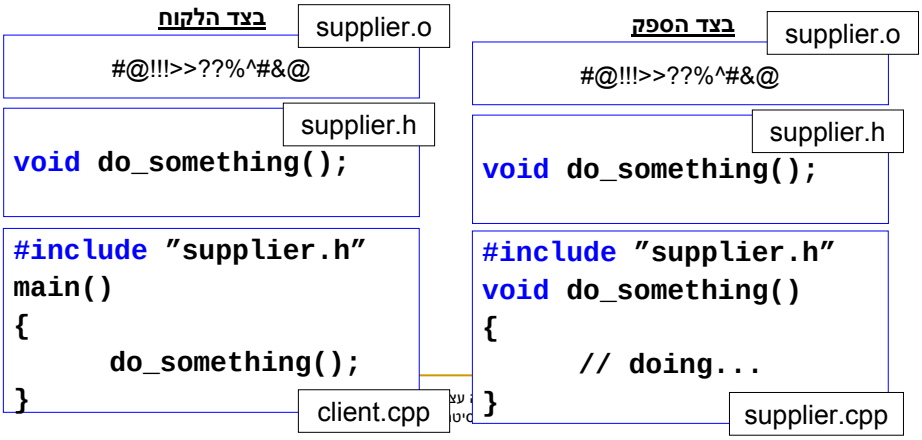
supplier.cpp

```
#include "supplier.h"
main()
{
    do_something(???);
}
```

client.cpp

לקוח וספק במערכת תוכנה

- לאחר תהליך ההידור מקבל הלקוח גם קובץ בינארי (o. או OBJ.) וגם קובץ כותרות (h.)



ממשק תחילה

- בתהליך פיתוח תוכנה תקין, כתיבת הממשק תעשה בתחילת תהליך הפיתוח
- כל מודול מגדיר מהם השרותים שלהם הוא זקוק ממודולים אחרים ע"י ניסוח ממשק רצוי
- ממשק זה מהווה בסיס לכתיבת הקוד הן בצד הספק, שיממש את הפונקציות הדרושות והן בצד הלקוח, שמשתמש בפונקציות (קורא להן) ללא תלות במימוש שלהן

ממשקים

- מטרת הממשק היא למזער את התלות בין חלקים שונים של המערכת
- ממשקים זעירים יוצרים מערכת:
  - קלה יותר להבנה
  - בעלת הסתרת מידע טובה יותר
  - קלה לתחזוקה ושינויים
  - מתקמפלת במהירות

לקוח וספק במערכת תוכנה

בצד הלקוח **בצד הספק**

supplier.h

```
void do_something();
```

משתמש בפונקציות לפי הממשק

ממש ע"פ הממשק את הפונקציות

client.cpp

```
#include "supplier.h"
main()
{
      do_something();
}
```

supplier.cpp

```
#include "supplier.h"
void do_something()
{
      // doing...
}
```

המודול Calendar

cal.h

```
// representation
struct Date {
   int d, m, y;
};

// interface for users
void init_date(Date& d, int, int, int); // initialize d
void add_year(Date& d, int n);    // add n years to d
void add_month(Date& d, int n);   // add n months to d
void add_day(Date& d, int n);     // add n days to d
Date next_weekend();// returns the next weekend Date

// inner use only
bool valid(const Date& d) ; // checks if d is valid

// global variable
extern Date today;
```


מודול כמרחב שמות

- אנו רוצים מבנה תחבירי שיציין את ה"רכיביות" של Calendar
- מיקום כל ההגדרות באותו קובץ אינו מפורש מספיק (למשל אינו נאכף ע"י המהדר)
- נעטוף את ההגדרות במרחב שמות:

```
namespace Calendar {

   // representation
   struct Date {
      int d, m, y;
   };

   // interface for users
   void init_date(Date& d, int, int, int); // initialize d
   ...
}
```

אופרטור השיוך ::

- לקוח של Calendar צריך מעתה לציין את מרחב השמות בכל שימוש (כמעט)

```
main()
{
  Date d;              // Error
  Calendar::Date d;    // OK

  Calendar::init_date(d,7,7,2005);    // OK
  init_date(d,7,7,2005);              // also OK - argument-
                                      // dependent lookup

  d = next_weekend()                  // Error
  d = Calendar::next_weekend()        // OK
}
```


אופרטור השיוך ::

- המממש של Calendar צריך גם הוא לציין את מרחב השמות בכל מימוש

```
// Date.c
void add_year(Date& d, int n)              // Error
{ d.y += n; }

void Calendar::add_year(Date& d, int n)    // OK
{ d.y += n; }
```

- בתוך מרחב השמות אין צורך לחזור על השם המלא

```
Date Calendar::next_weekend()
{
   Date rslt = {7,7,2005}; // no need for Calendar::Date
   return rslt;
}
```

הצהרת משתמש

- כאשר לקוח של Calendar עושה שימוש נרחב *בשרותים מסוימים* של Calendar, הוא יכול לחסוך את הסרבול התחבירי ע"י **יבוא** של שמות מסוימים מתוך המרחב (using declaration)

```
main()
{
   using Calendar::Date;
   using Calendar::init_date;  // using all overloaded versions

   Date d;      // Same as Calendar::Date d;

   init_date(d,7,7,2005);  // Same as Calendar::init_date(d,7,7,2005);

   d = next_weekend()   // Error – not imported
}
```


הנחיית שימוש

- כאשר לקוח של Calendar עושה שימוש נרחב בכל השרותים של Calendar, הוא יכול לחסוך את הסרבול התחבירי ע"י **יבוא כל מרחב השמות** (using directive)

```
main()
{
  using namespace Calendar;

  Date d;
  init_date(d,7,7,2005);
  d = next_weekend()
}
```

הנחיית שימוש

- ייבוא מרחבי שמות רבים בשלמותם מעקר מתוכן את מרחב השמות
- פקודה זו הוספה לשפה בעיקר כדי להקל הטמעה של קוד שנכתב מחוץ למרחב שמות לתוך מערכת התומכת בכך (כדוגמת שילוב ספריות C בתוכנית C++)
- שימוש נרחב בתכונה זו מרמז בדרך כלל על התלויות הפנימיות במערכת

ריבוי ממשקים

- לא כל השרותים המוגדרים ב Calendar אמורים להיות מיוצאים (exported)
- למשל הפונקציה `bool valid(const Date&)` אמורה לשמש את מממשי Calendar בלבד
- לדוגמא כך:

```
void Calendar::add_day(Date& d, int n)
{
  d.d += n;
  if (!valid(d))
      // illegal date – fix it
}
```

ריבוי ממשקים

- מתכנתים ומהדרים נוקטים בגישה הקונסרבטיבית (המוצדקת!) – אם שרות כלשהו זמין ללקוח כלשהו יש להניח כי הלקוח תלוי בקטע קוד זה
- כלומר יש צורך להגדיר ממשקים נפרדים (לא בהכרח זרים) לממשים ולמשתמשים (ניתן אפילו לחשוב על ממשקים שונים למשתמשים שונים)

ריבוי ממשקים

בצד הלקוח

```
namespace Calendar {
// only exported features
}
```

Calendar'

בצד המממש

```
namespace Calendar {
// all features
}
```

Calendar

בצורה גרפית המערכת נראית כך:

החיצים מתארים את היחס "תלוי במימוש המוצג ע"י"

ריבוי ממשקים

- הפתרון שהוצג בשקף הקודם הוא פתרון פשוט וטוב
- החסרון העיקרי הוא שלו הוא ש- Calendar ו- Calendar' מתארים שני מרחבי שמות בעלי אותו שם (ניתן להשיג זאת ע"י שימוש בקובצי h. שונים).
- המהדר (compiler) לא יוכל תמיד לבדוק את עקביות ההגדרות הכפולות
- בעיה זו נפתרת כמעט תמיד ע"י שימוש במהדרים חכמים וע"י המקשר (linker)

עיצוב חלופי

בצד הלקוח

```
namespace Calendar_interface
{
  using Calendar::Date;
  using Calendar::init_date;
  // for all exported operations…
}
```

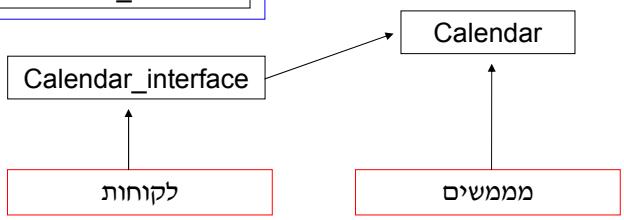
Calendar_interface

בצד המממש

```
namespace Calendar {
// all features
}
```

Calendar

בצורה גרפית המערכת נראית כך:



עיצוב חלופי

- בעיצוב זה ברור שהמשתמשים תלויים רק בתכונות שיוצאו עבורם ע"י Calendar_interface (ולא יושפעו משינויים בפעולות פנימיות)
- ואולם בגישה הקונסרבטיבית שאותה אנו נוקטים, גרף התלויות מראה תלות (טרנזיטיבית) בין המשתמשים ובין Calendar
- נרצה למנוע מראית עין זו, ע"י עיצוב חלופי

עיצוב חלופי (2)

בצד הלקוח

```
namespace Calendar {
// only exported features
}
```

Calendar'

בצד המממש

```
namespace Calendar {
// all features
}
```

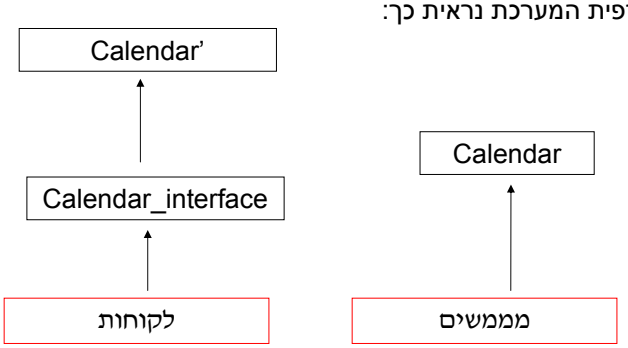
Calendar

```
namespace Calendar_interface
{
  using Calendar::Date;
  using Calendar::init_date;
  // for all exported operations…
}
```

Calendar_interface

עיצוב חלופי (2)

בצורה גרפית המערכת נראית כך:



- ההבדל היחיד מהגרסה הראשונה הוא הוספת ממשק הביניים Calendar_interface
- עדיין אנו סומכים על הקומפיילר שיעזור לנו לפתור בעיות חוסר עקביות בין ההגדרות

עיצוב חלופי (3)

בצד הלקוח

```
namespace Calendar_interface {
  void init_date(Date& d, int, int, int);
  void add_year(Date& d, int n);
  // for all exported operations…
}
```

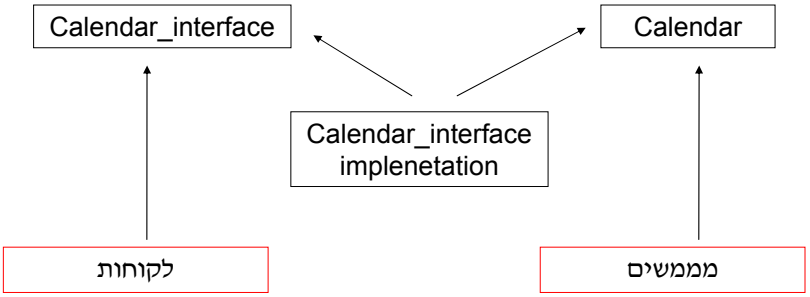
Calendar_interface

```
void Calendar_interface::init_date(Date& d, int dd, int mm, int yy)
{
        Calendar::init_date(d, dd, mm, yy);
}
// for all exported operations…
```

Calendar_interface implementation

עיצוב חלופי (3)

בצורה גרפית המערכת נראית כך:



- פתרון זה ממזער את התלויות במערכת
- זהו כנראה OVERKILL

מניעת התנגשות שמות

- קשה להכליל את שני הקבצים הללו באותה תוכנית:

```
// my.h:
char f (char);
int f(int);
struct String { /* ... */ };


// your.h:
char f(char);
double f(double);
struct String { /* ... */ };
```

מניעת התנגשות שמות

- עכשיו קל:

```
namespace my
{
   char f (char);
   int f(int);
   struct String { /* ... */ };
}

namespace your
{
   char f(char);
   double f(double );
   struct String { /* ... */ };
}
```


שימוש בספריות C

```
// stdio.h:
namespace std {
  // ...
  int printf (const char * ... );
  // ...
}
using namespace std ;
```

שימוש בספריות C – לשומרי סביבה

```
// cstdio:
namespace std {
  // ...
  int printf (const char * ... );
  // ...
}
```

שימוש בספריות C – למתנגדי שכפול הקוד

```
// stdio.h:
#include <cstdio>
using namespace std;
```

תפקידי מרחב השמות

- לבטא קשר לוגי-רעיוני בין אוסף של פעולות
- למנוע ממשתמשים גישה לפעולות לא להם
- לחסוך למשתמש את הטירחה (התחבירית) בהשגת מטרות אלו באמצעים אחרים

מרחב השמות והעולם האמיתי

- הערה:
  - מרחב השמות משמש לניהול רכיבים בקנה מידה גדול בהרבה מאלה שנתאר כאן
  - מחלקות מייתרות למעשה את השימוש במרחב שמות לתאור מבנה נתונים אחד בתור מודול
  - מרחב שמות טיפוסי מכיל עשרות מחלקות ויותר
  - השימוש במרחבי שמות מקובל מאוד בשפות C# ו- Java (שם הוא נקרא package)

הפסקה