The first law of defensive programming:

When you have to defend, You have already lost هنگامی که شما باید دفاع کنید، قبلا از دست داده اید

* **Consistent objects**
  + Create consistent and complete objects ایجاد اشیاء سازگار و کامل
* Consistent mutations
  + All transitions must lead to complete and consistent state تمام گذرها باید به حالت کامل و سازگار منجر شوند
* **No primitive types**
  + Avoid excessive use of primitive types اجتناب از استفاده بیش از حد از انواع ابتدایی
  + Replace primitive types with custom domain types
  + Consequence : no defense
* **Function domains**
  + Satisfy mandatory conditions before making a function call رعایت شروط اجباری قبل از اجرای توابع

Try{

doSomethings();

}catch(InvalidOperationException){ 🡨 **Defensive code**

…

}

* Defensive design starts after this point
* **Defensive design**
  + Encapsulate state and operations
  + Unify positive and negative execution flows متحد کردن جریان های مثبت و منفی اجرا
* **Objects, not nulls**
  + Do not use null references
  + Without null : no defense
* **Rich domain models**
  + Immutability: Good servant, bad master تغییر ناپذیری : کلاینت خوب ، سرور بد
  + Restricted mutability for the best of both worlds تغییر پذیری محدود برای هر دو جهان
  + Historical modeling for powerful features
* **No exceptions**
  + Do not introduce alternate execution paths with exceptions
  + Do not use exceptions as just another heavyweight if-else
  + Better approach : pass either the result or the error in a discriminated union رویکرد بهتر: نتیجه یا خطای یک یونیون تبعیض آمیز را منتقل کنید

**Defensive code is a matter of choice**

**Cheap features: defined in code**

**Expensive features: defined in design**

**Traditional defensive techniques**

|  |  |
| --- | --- |
| Predictability پیش بینی پذیری | Reveal intention افشای قصد   * Method name * Argument name * Return value |
| **Certainty** یقین | Do not pass switches to a method   * No Boolean arguments * No enum arguments   Switches introduce uncertainty  Method should do one thing with absolute certainty متد باید یک چیز را با اطمینان مطلق انجام دهد |
| **Simplicity** سادگی | * Keep small number of method arguments * Keep the code short * Avoid branching when possible * Avoid code repetition |
| **Completeness** تکمیل | * **If** instruction should have a reasonable **else** branch * **Switch** instruction should have a reasonable **default** case |
| **Failing fast** شکست سریع | Do not let bad data propagate اجازه ندهید اطلاعات بد منتشر شود |
| **Handling errors** رسیدگی به خطاها | * Handle errors when day happen * Assert preconditions, postconditions and invariants * Add global exception handler to keep the application running |

**An experiment**

bool Contains(int[] arr, int srch)

{

for (int i = 0 ; i< arr.length(); i++)

{

if(arr[i] == srch)

{

return true;

}

}

return false;

}

-🡪

bool Contains(int[] arr, int srch)

{

Int end = ar.length-1;

Int last = ar[end];

Arr[end] = srch;

Int index = 0;

While(ar[index] != search)

{

Index = index +1;

}

Ar[end] = last;

Return index < end || last == srch;

}

Proactive caller: if(valid(x)) obj.Dosomethings(x);

Reactive caller : try{

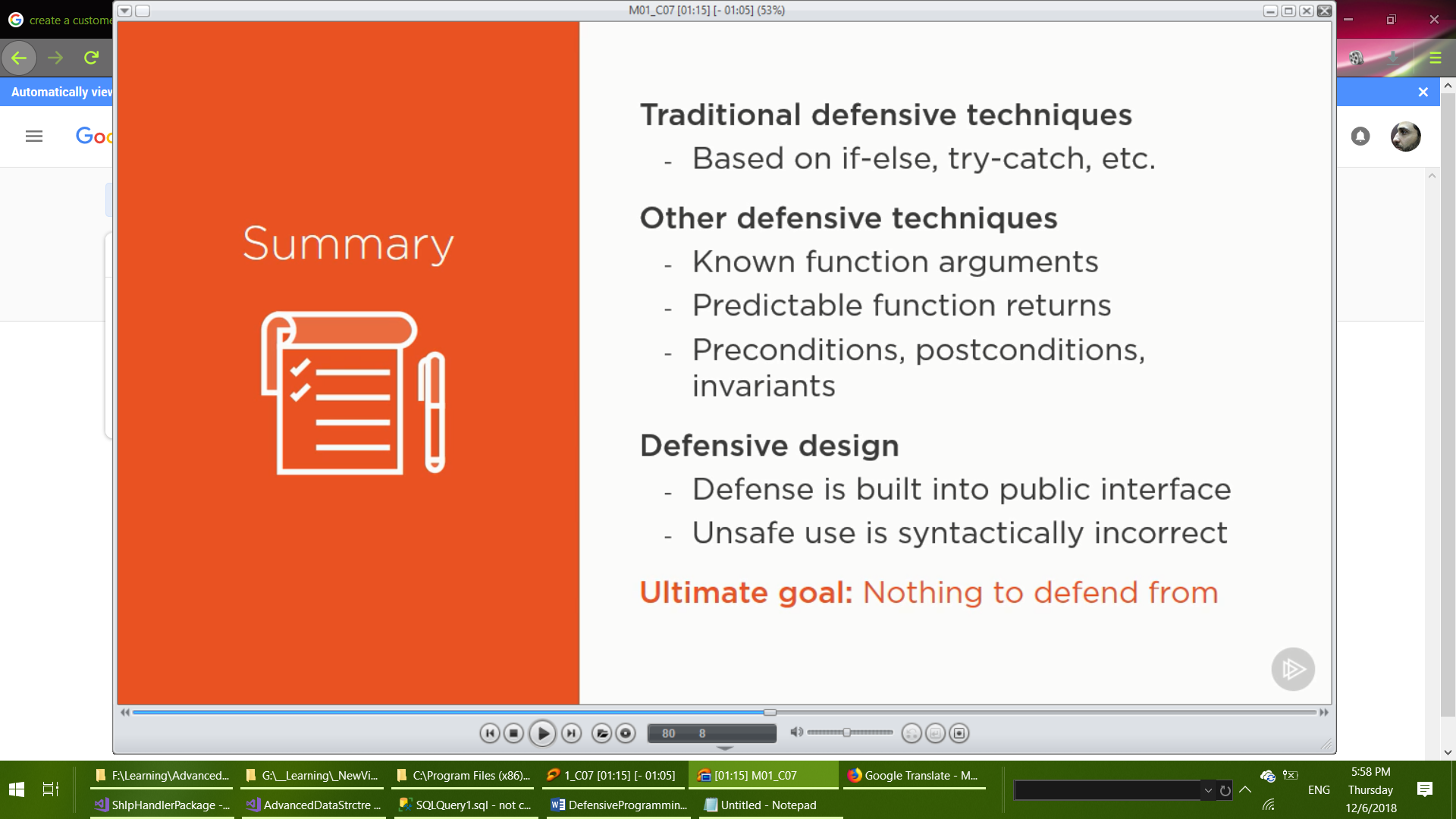
obj.Dosomethings(x);

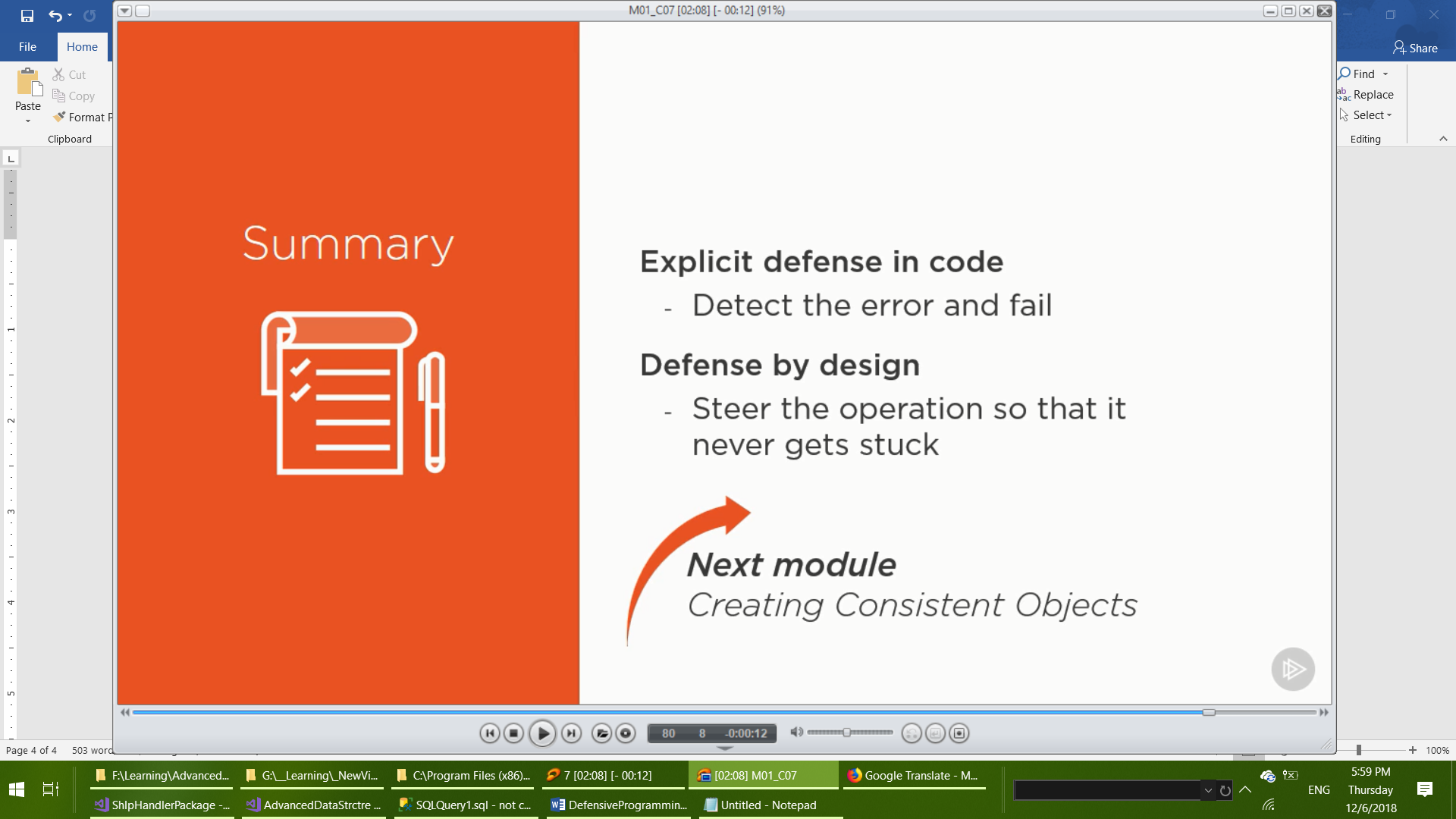
}catch…

Want to avoid error conditions

Produce design in which they can’t appear

**Software which repeatedly transforms one system state into the next system state**





Implicit parameter less constructor is the built-in factory function

Set numeric fields to zero, Boolean to false, references to null

In customized constructor: when invalid data are received new object must **not** constructed.

Constructor must ensure that only valid objects can be created.

The customized constructor caller will never be able to obtain an inconsistent object.

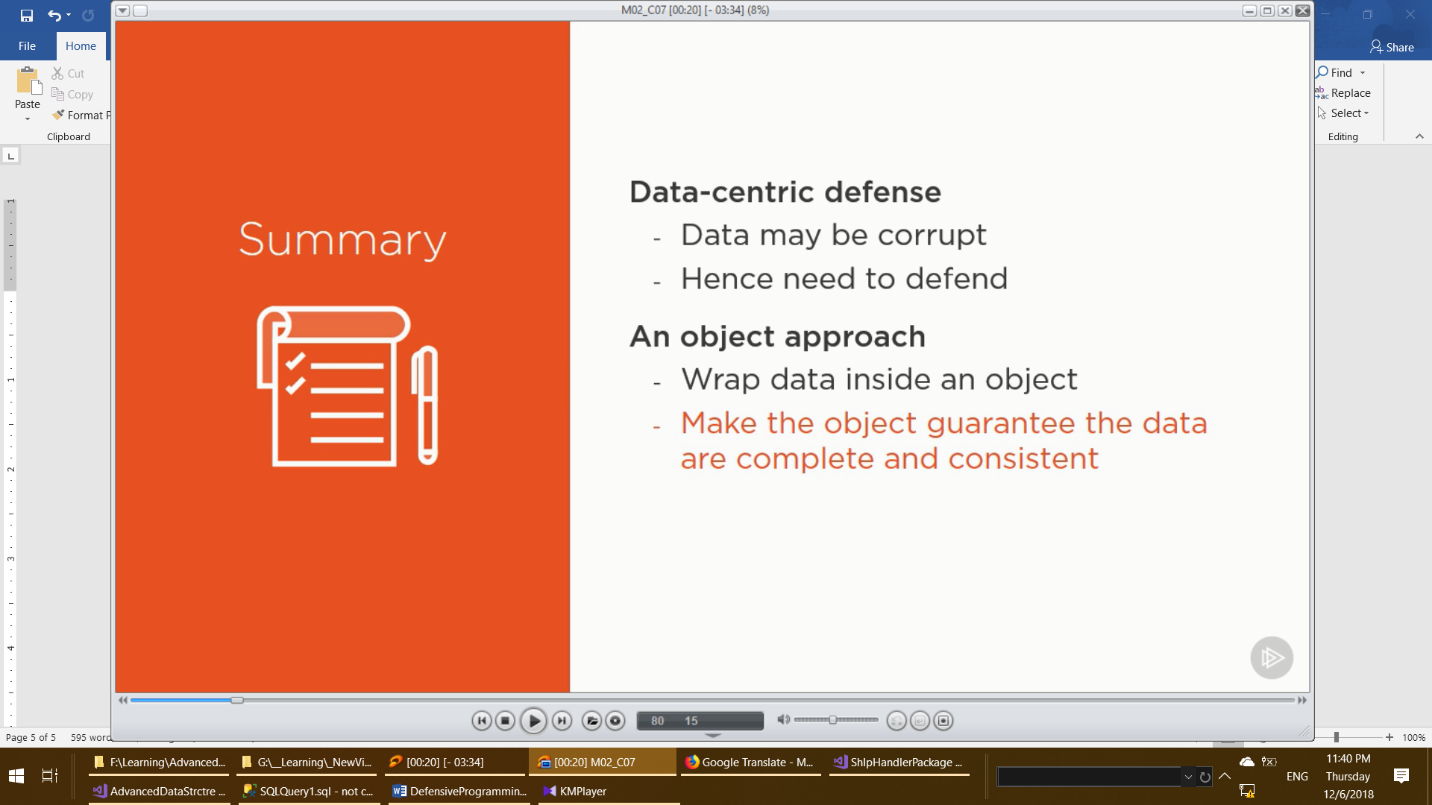
Provide your own factory function for every stateful objects.

**Rules of thumb:**

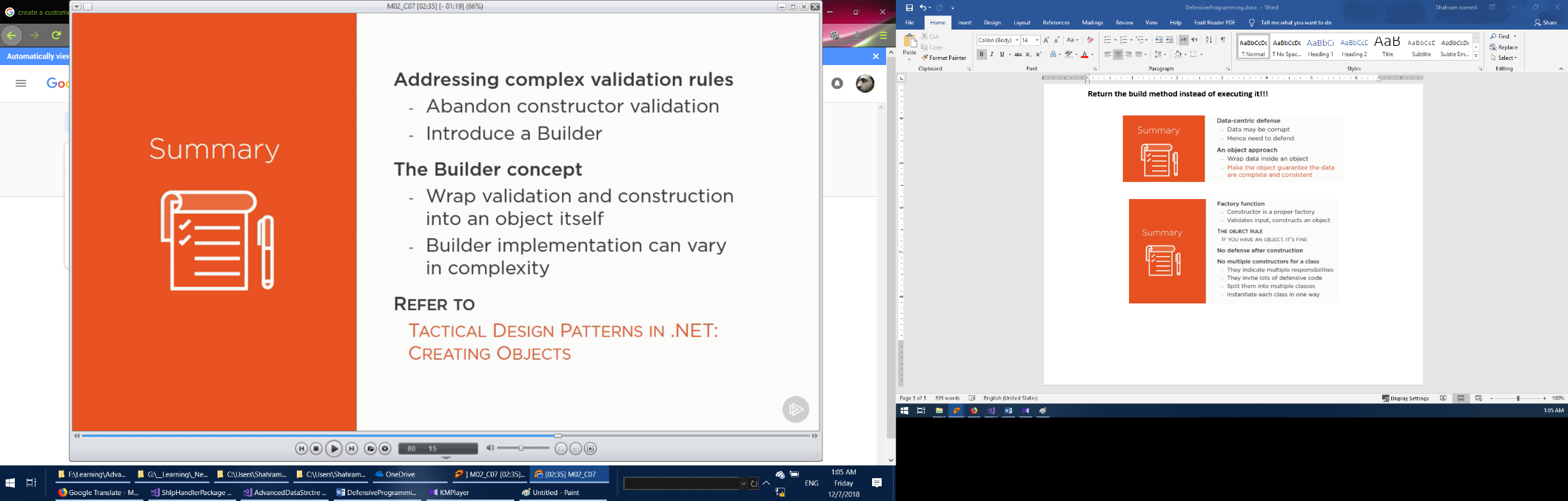
Define one factory function per class

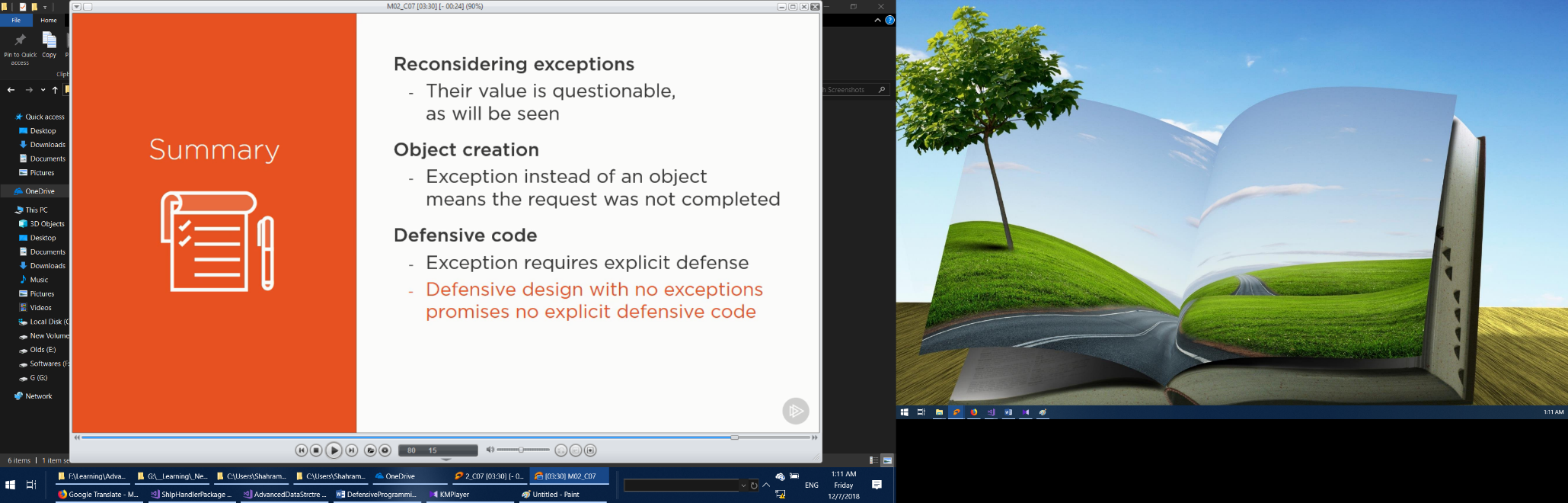
Have no discrete parameters (no enum , Boolean , etc.)

**Return the build method instead of executing it!!!**

****

****

****

****

**For complex construction validation:**

1. Lightweight solution
   1. Use constructor as a factory function
2. Middle solution
   1. Series of small factory function
3. heavyweight solution
   1. use builder for complex validation

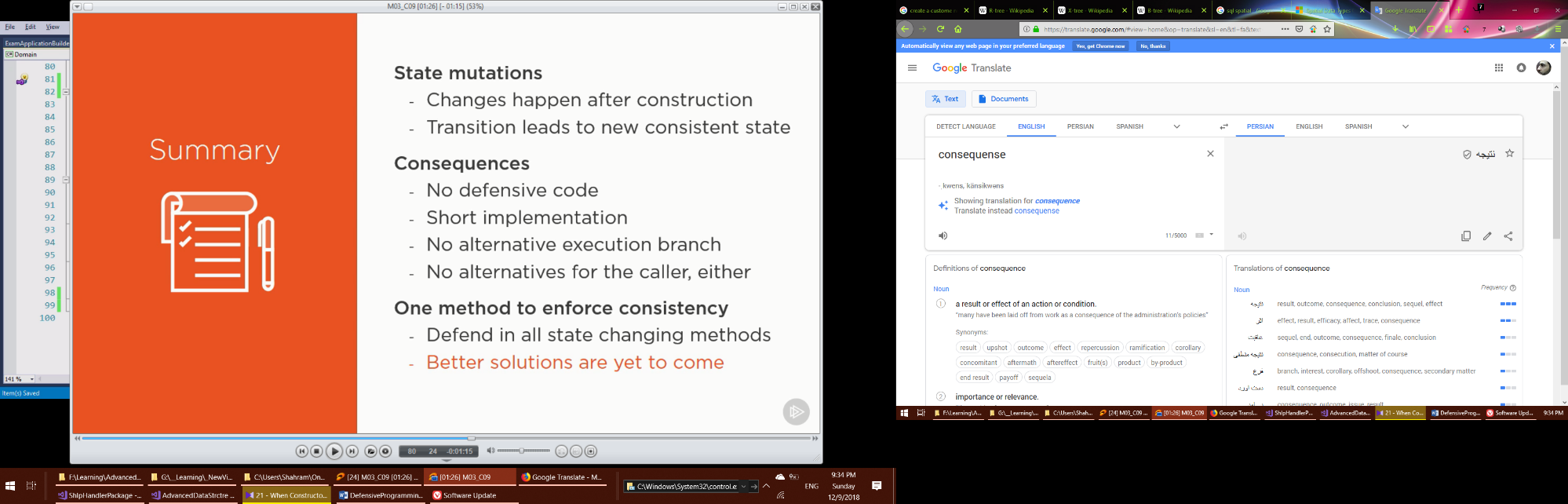
**Domain Model:**

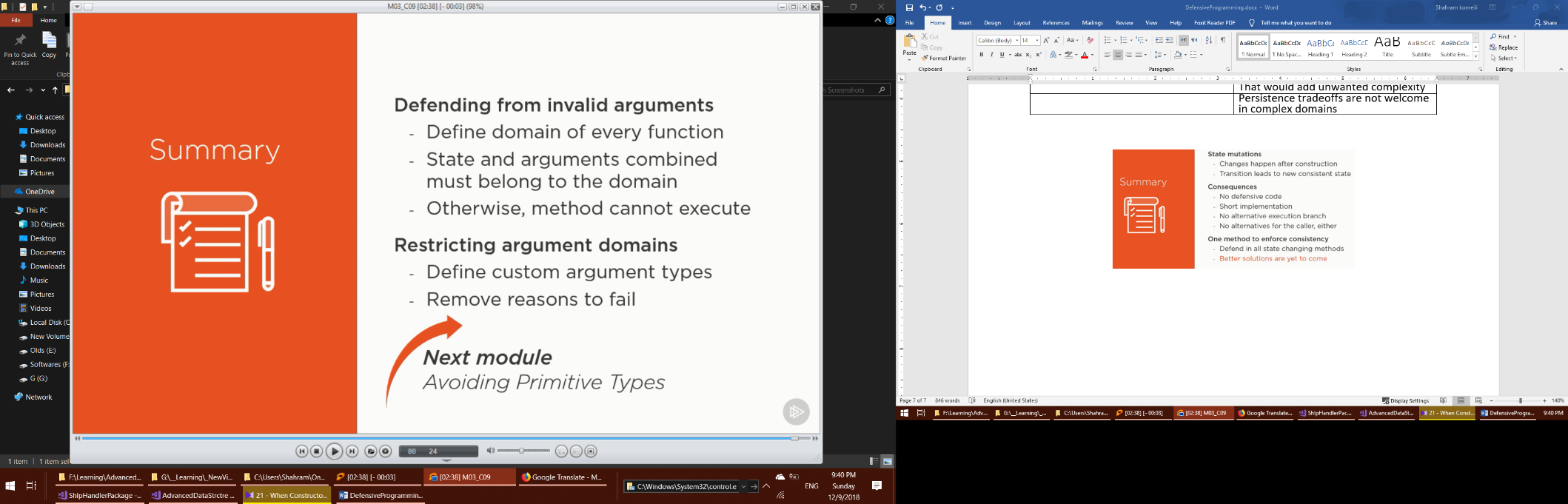
1. Default constructor:
   1. Include parameter less constructor in all model classes
   2. Constructor can be private (ORM will access it via reflection)
2. Property setters:
   1. Setters on all properties defining persist able state
   2. Setters can be private (ORM will access them via reflection)
3. No setter validation:
   1. Property setters must be dumb(بی عقل)
   2. ORM is not aware(مطلع) of any setter rules
   3. Object materialization(تحقق) may fail if setters can throw
4. Database ID field:
   1. Include database identity in every model class

**Separate persistence model:**

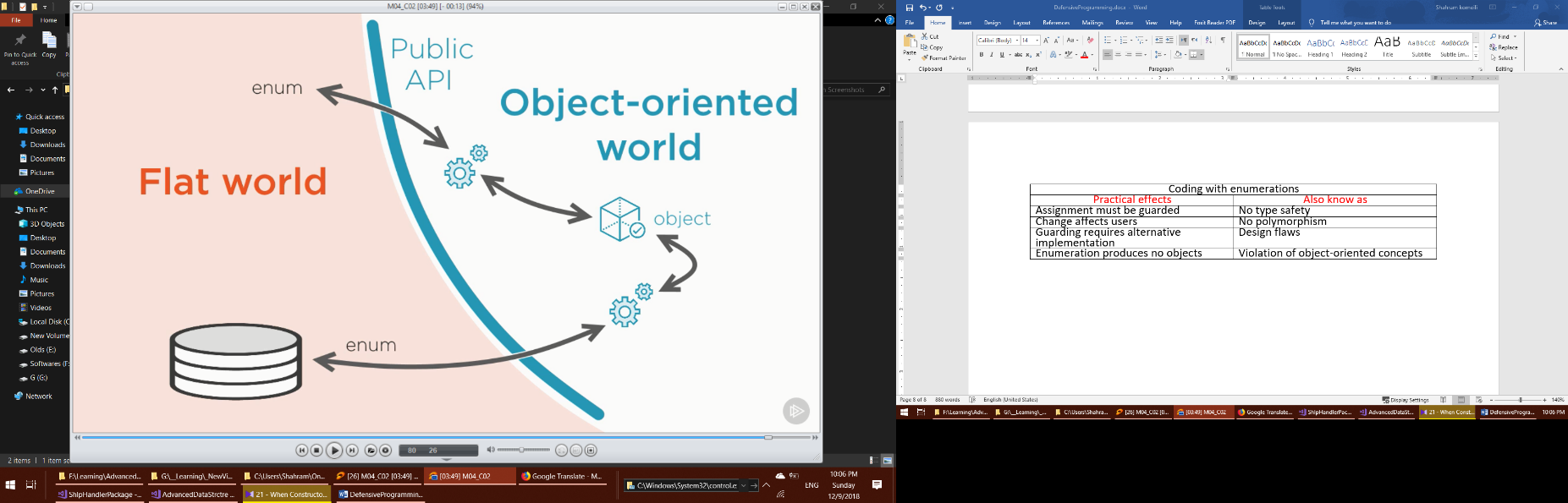
1. New model:
   1. Define new model in the infrastructure layer
   2. No constructor rules
   3. No validation
2. Plain getters/setters:
   1. Plain property getters and setters
   2. Setters with no validation
   3. Not same as the domain model
   4. Designed to support fast and easy persistence
3. Two-way mapping:
   1. There must be a mapping between domain model and persistence model
   2. Map new domain object before persisting it
   3. Map persisted object to domain object before using it

|  |  |
| --- | --- |
| How to decide | |
| Simple model | Complex model |
| Make the domain model persistable | invest in separating persistence from domain |
| That saves a lot of work | Hard to add persistence to an already complex domain model |
|  | That would add unwanted complexity |
|  | Persistence tradeoffs are not welcome in complex domains |





|  |  |
| --- | --- |
| Coding with enumerations | |
| Practical effects | Also know as |
| Assignment must be guarded | No type safety |
| Change affects users | No polymorphism |
| Guarding requires alternative implementation | Design flaws |
| Enumeration produces no objects | Violation of object-oriented concepts |



The Stringification trap

Turning objects into strings = stringification

To send over network, save to databae , etc.

Then, why string inside domain objects?

**Use Class instead of Enum , Use class instead of String**

**USE CLASS INSTEAD OF ALLLLLL.**

**pitfalls of primitive’s types**

Encapsulate primitive types in a class

Do not accept primitive type arguments