Refactoring

- Restructuring existing code by revising its internal structure without changing its external behavior.
  - http://en.wikipedia.org/wiki/Refactoring
  - http://www.refactoring.com/
  - http://sourcemaking.com/refactoring
  - *Refactoring: Improving the Design of Existing Code*
    - by Martin Fowler, Addison-Wesley

Example Refactorings

- Encapsulate Field
  - http://sourcemaking.com/refactoring/encapsulate-field
- Replace Magic Number with Symbolic Constant
  - http://sourcemaking.com/refactoring/replace-magic-number-with-symbolic-constant
- Replace Type Code with Class (incl. enumeration)
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with State/Strategy
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
- Replace Conditional with Polymorphism
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

What is NOT Refactoring? What is it for?

- Refactoring is not about
  - Finding and fixing bugs.
  - Adding new features/functionalities.

- However, refactoring makes it easier to
  - Maintain and debug.
  - Add new features/functionalities.
  - Review/understand code.

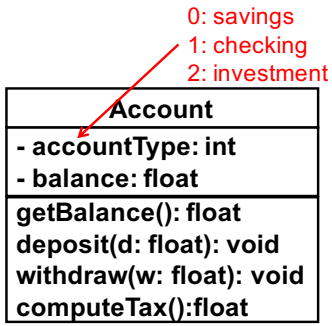
Where/When to Refactor?

- 22 bad smells in code
  - Typical places in code that require refactoring.
  - *Refactoring: Improving the Design of Existing Code*
    - by Martin Fowler, Addison-Wesley
  - http://sourcemaking.com/refactoring/bad-smells-in-code
  - http://en.wikipedia.org/wiki/Code_smell

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class
- Refused bequest
- Comments

Example Bad Smell: Primitive Obsession

- Avoid built-in primitive types. Favor more structured types (e.g. class and enum) and class inheritance.
  - http://sourcemaking.com/refactoring/primitive-obsession



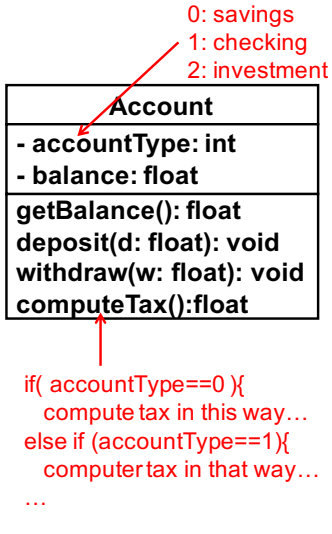
- Replace Magic Number with Symbolic Constant
  - http://sourcemaking.com/refactoring/replace-magic-number-with-symbolic-constant
- Replace Type Code with Class (incl. enumeration)
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with State/Strategy
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
- Replace Conditional with Polymorphism
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

Example Bad Smell: Switch Statements

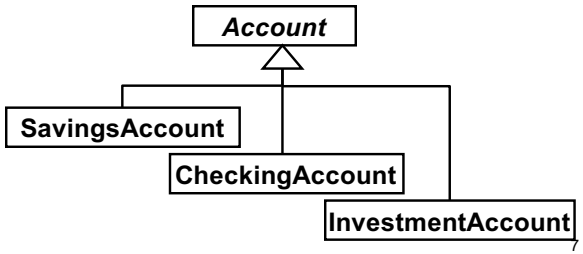
- Minimize the usage of conditional statements and simply them.
  - http://sourcemaking.com/refactoring/switch-statements
  - http://sourcemaking.com/refactoring/simplifying-conditional-expressions
  - Replace Type Code with State/Strategy
    - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
  - Replace Type Code with Subclasses
    - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
  - Replace Conditional with Polymorphism
    - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

What's Wrong with Conditionals?

- They often scatter in code and get harder to maintain.



- The same conditional statement is duplicated in different places in code.
- When you make a change, you have to find all the conditional statements and change them.



Simplifying Conditional Expressions

- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Decompose conditional
- Introduce assertion
- Introduce null object
- Remove control flag
- Replace conditional with polymorphism
- Replace nested conditional with guard clauses

HW 9-1

- Understand code smells
  - http://sourcemaking.com/refactoring/bad-smells-in-code
- Understand several refactorings listed in Slides #2 and #8

Refactoring and Unit Testing

- Refactoring does not change the external behavior of a piece of code
  - e.g. a method, a set of methods, a class or a set of classes,
- Need to make sure that no changes exist in the external behavior.
  - Do unit testing *before* and *after* a refactoring

Design Patterns

Design Patterns

- Tested, proven and documented solutions for recurring design problems in given contexts.
- Each design pattern is structured as
  - Pattern name
  - Intent
  - Motivation
  - Applicability
  - Class structure
  - Participants
  - …etc.

Resources

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  - By Eric Gamma et al.
  - Addison-Wesley
- *Head Start Design Patterns*
  - by Elizabeth Freeman et al.
  - O'Reilly
- Web
  - http://en.wikipedia.org/wiki/Design_patterns_(computer_science)
  - http://sourcemaking.com/design_patterns

Benefits of Design Patterns

- Useful information source to learn and practice good object-oriented designs
- Useful as a communication tool among developers
  - e.g., Recursion, collection (array, stack, queue, etc.), sorting, buffers, infinite loops

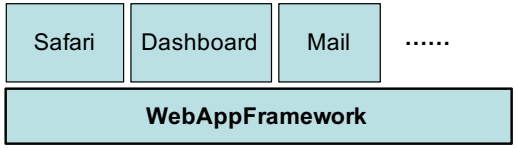
Factory Method

Factory Method

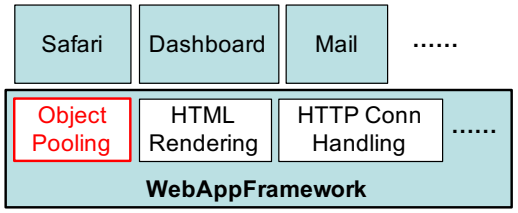
- A method to instantiate a class and initializes a class instance without using its constructors
  - Uses a regular (i.e., non-constructor) method.
  - Lets a class *defer* instantiation to subclasses.
    - Define an abstract class (or an interface) for creating an instance.
    - Let its subclasses (or implementation classes) decide *which class to instantiate*.

An Example: Web App Dev Framework

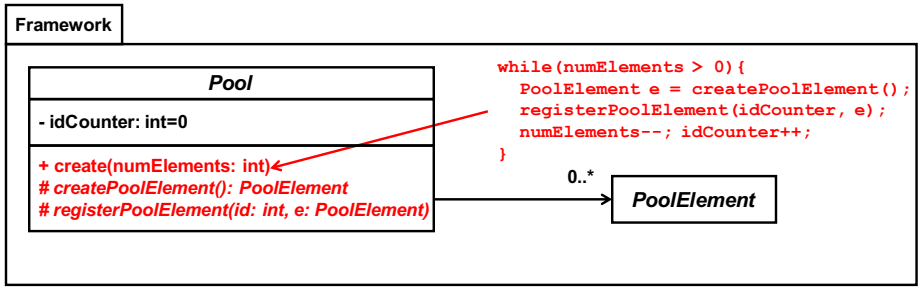
- Assume you are implementing a reusable development framework for various types of web apps
  - Implements common functionalities and make them reusable/available for individual web apps.
  - c.f. WebKit (http://www.webkit.org/)
    - Web browsers (incl. Safari), Dashboard, Mail and many other Mac OS X apps.



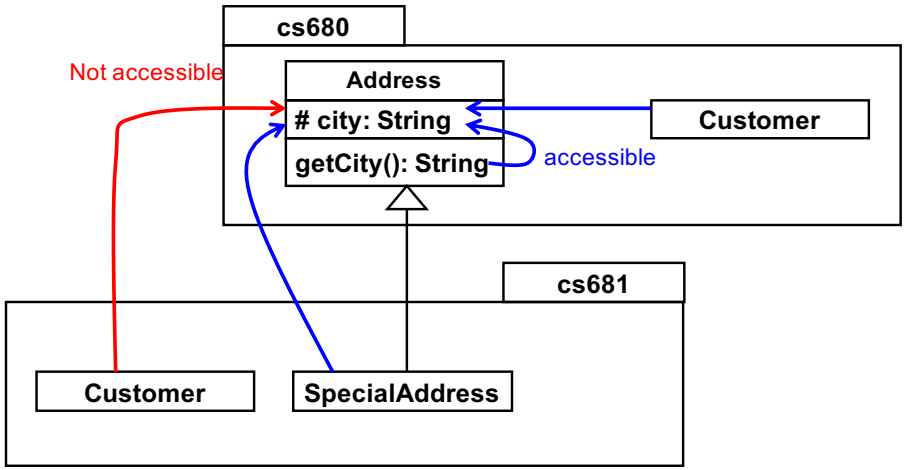
- Assume you are implementing an object pooling API in this framework.
  - Creating and using a pool of (the same kind/type of) objects
    - e.g., a pool of browser windows, a pool of tabs in each browser window, a pool (cache) of HTML files, a pool of HTTP connections, a pool of threads, etc.
- Here, we focus on the *creation* of pools.

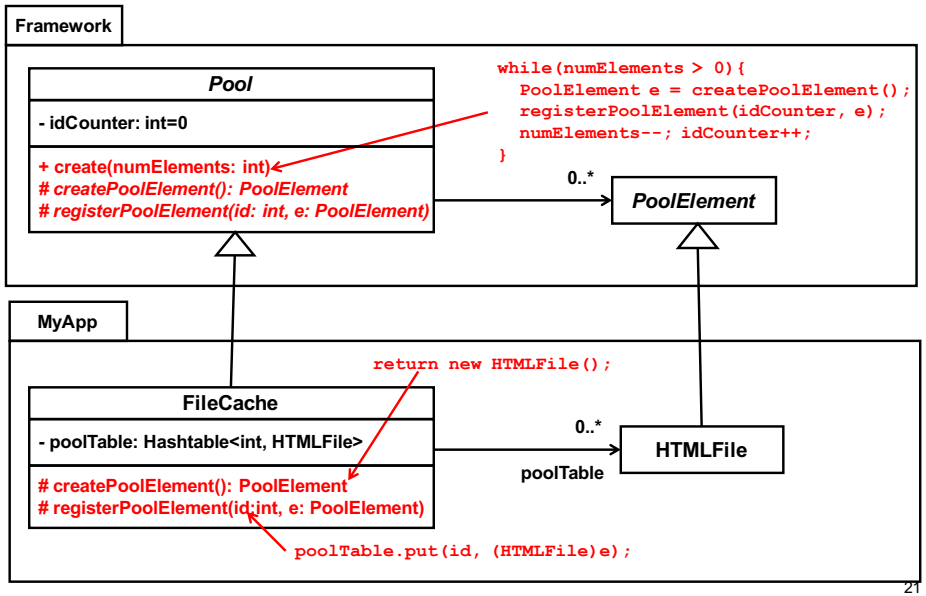


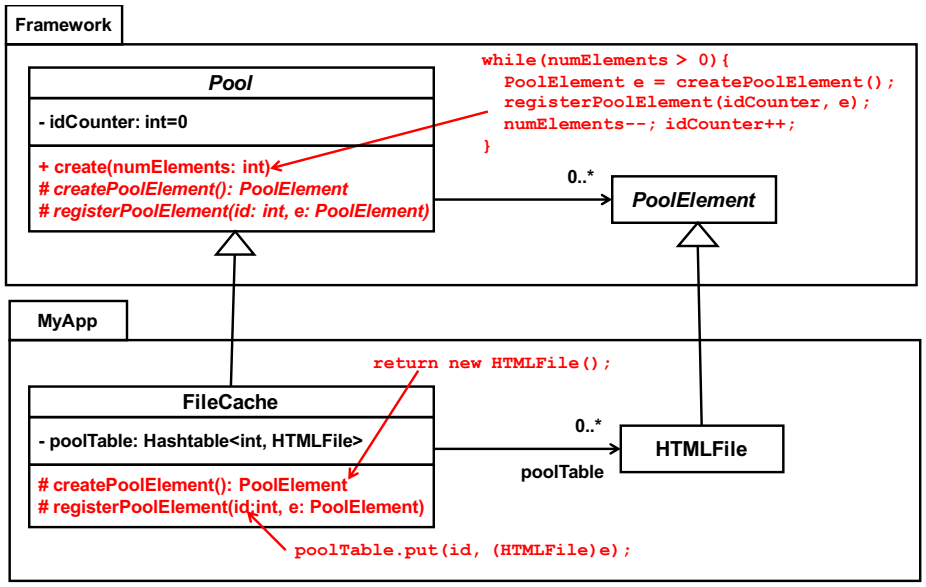
Factory Method in Object Pooling



Recap: "Protected" Visibility







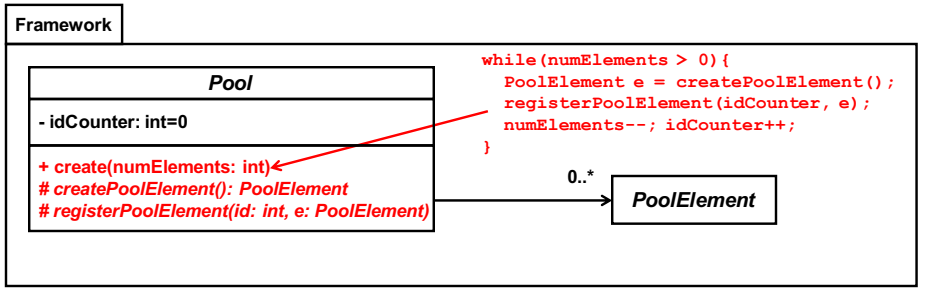

```
FileCache cache = new FileCache();
cache.create(10);
```


What's the Point?

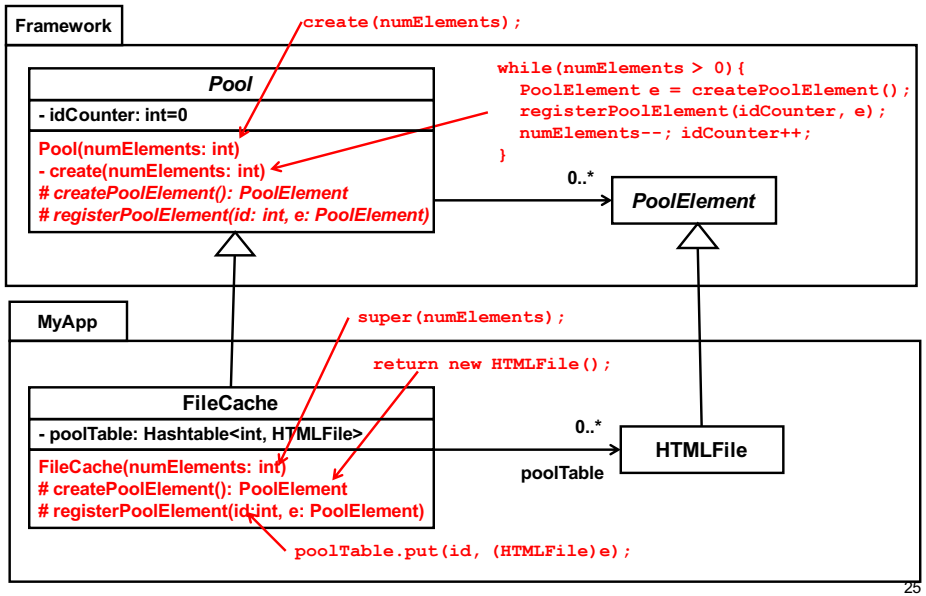
- The framework
  - Provides a skeleton (or template) for pool creation logic.
    - Partially implements pool creation logic.
  - Never specify specific types (specific class names) for a pool and its elements
- MyApp (framework client)
  - Reuses the skeleton/template of pool creation logic and completes it
    - By specifying which pool class and which pool element class are used.

Factory Method

- createPoolElement()
  - Allows create() to avoid specifying the class name for pool elements.
  - Allows the framework to be independent (or de-coupled) from individual applications (framework clients).
    - Allows applications to be pluggable to the framework.

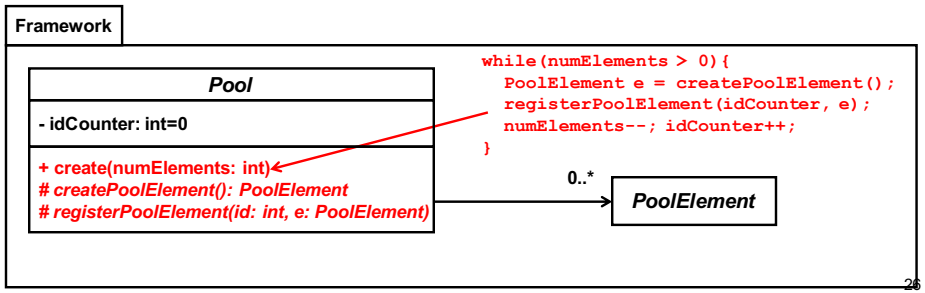


An Alternative Design



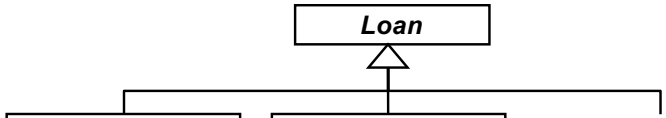
Other Considerations

- idCounter



Static Factory Method

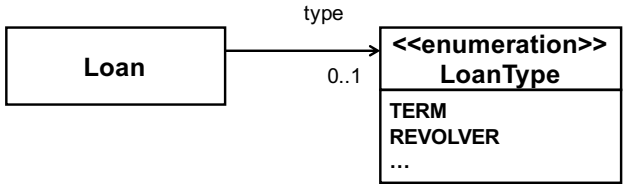
Recap: Using a class Inheritance or not



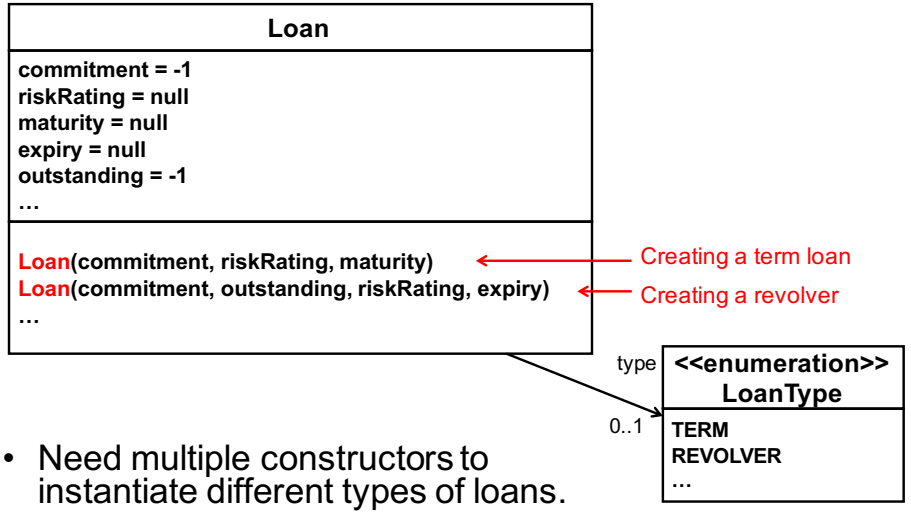
AdvisedLineLoan
SPLC
RCTL
risk adjusted versions
etc.

- Term loan
  - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
  - With a spending limit and expiry date
- Re-instantiation problem
  - A revolver can transform into a term loan when the revolver expires.

Enumeration-based Design

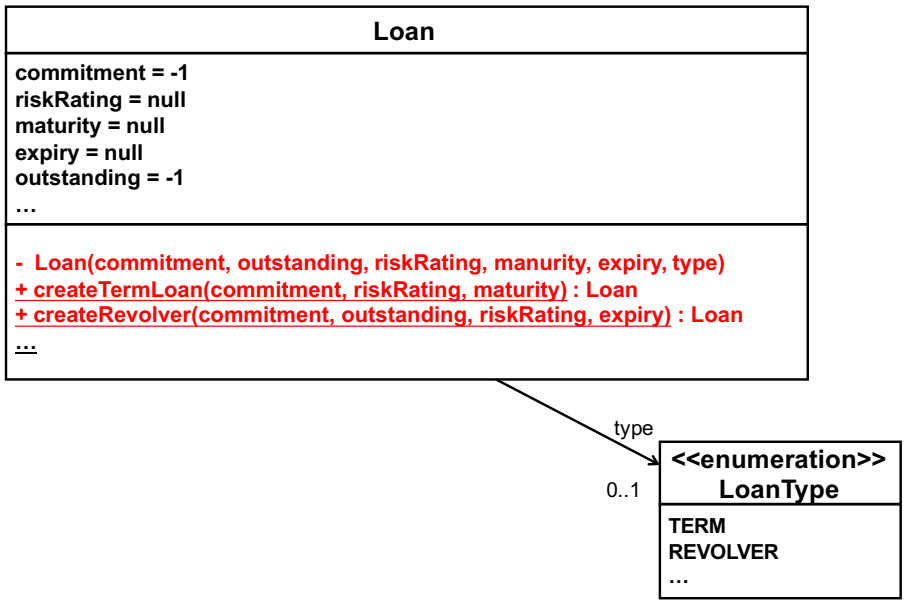


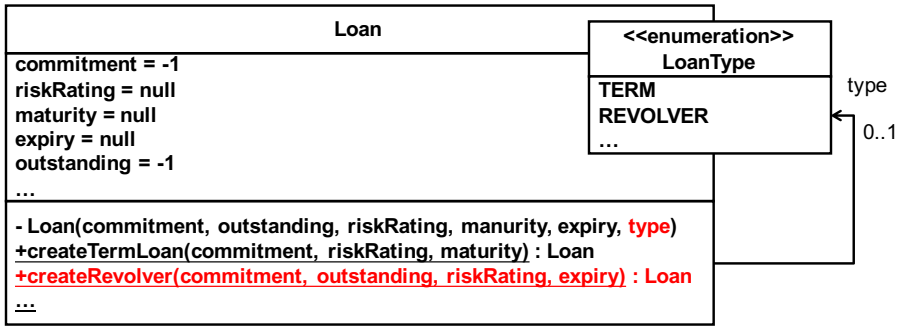
- A class inheritance should not be used here.



- Need multiple constructors to instantiate different types of loans.
- Implicit and error-prone.
  - `Loan l1 = new Loan(100, 0.9, new Date(…));`
    `l1.setLoanType( LoanType.TERM );`
  - `Loan l2 = new Loan(100, 0, 0.7, new Date(…));`
    `l2.setLoanType( LoanType.REVOLVER );`

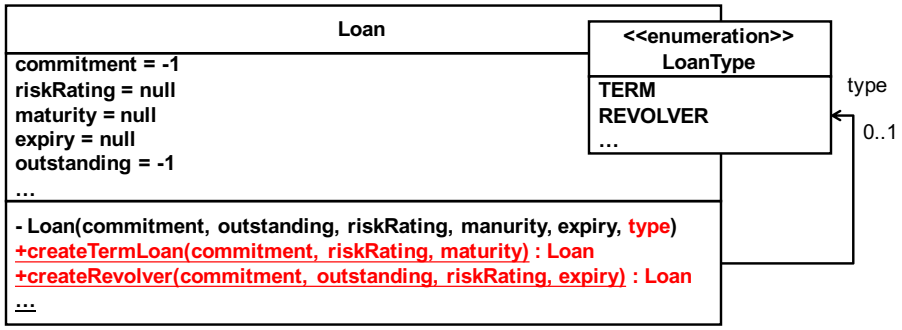
Static Factory Methods





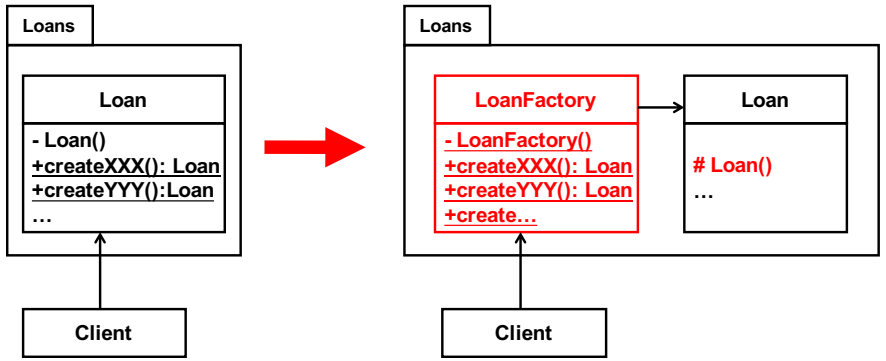
- Client/user of Loan
  - `Loan loan = Loan.createRevolver(1000,0,…,…);`

```
• Public class Loan{
    private LoanType type = null;
    ……
    private Loan(…,…,…,…,…,…){ … }
    public static Loan createRevolver( commitment, outstanding,
                                       riskRating, expiry ){
      return new Loan( commitment, outstanding, riskRating, null,
                       expiry, LoanType.REVOLVER );
    }
```



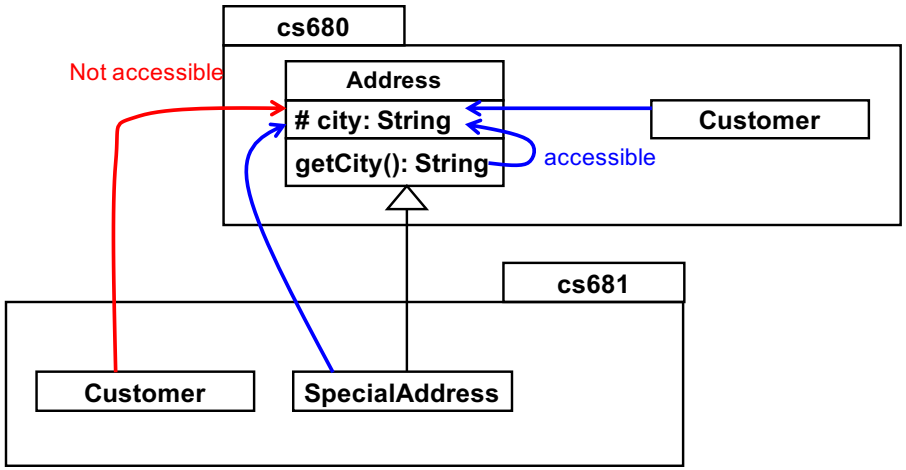
- Too many (static) factory methods in a class can obscure its primary responsibility/functionality.
  - They could dominate the class's public methods.
  - Loan no longer strongly communicates it's primary (i.e., loan-related) responsibility/functionality.

Alternative: Factory Class



- Factory class
  - A class that encapsulates (static) factory methods and isolate instantiation logic from other classes.
- Loan can directly/strongly communicate its primary responsibility/functionality.

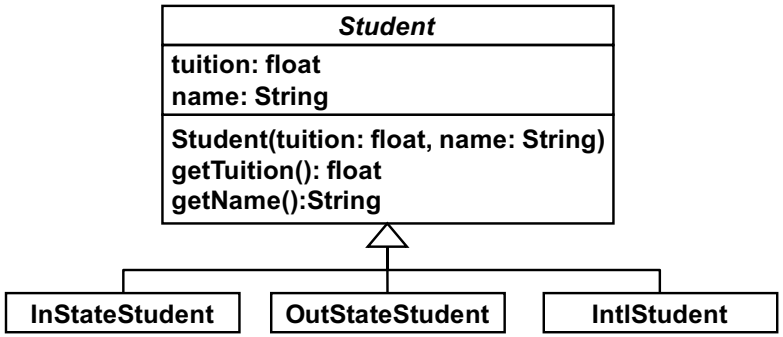
Recap: "Protected" Visibility



Design Tradeoffs

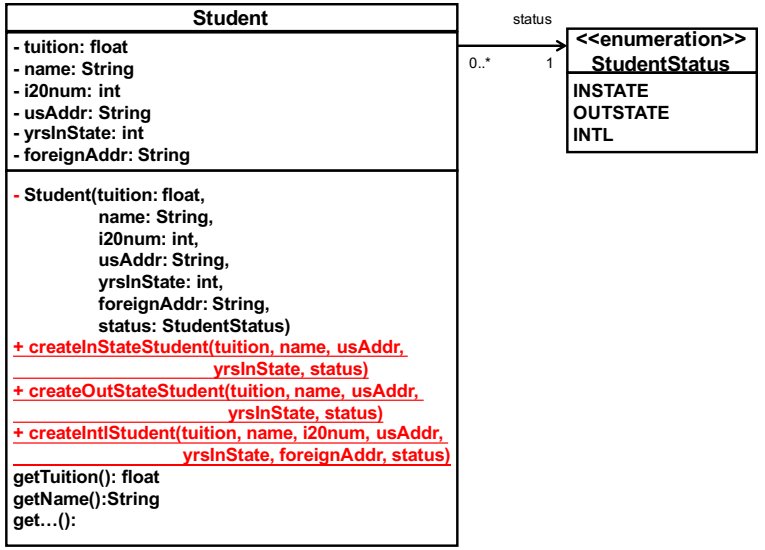
- Class inheritance
  - Pros: Straightforward.
  - Cons: Dynamic class-switching is impossible.
- Static factory method
  - Pros: No dynamic class-switching problem.
  - Cons: Potentially too many factory methods in a single class
- Factory class
  - Pros: Separates a class's primary logic and its instantiation logic
  - Cons: Non-factory classes in the same package can call protected constructors.
    - Could violate the encapsulation principle.
    - Consider an inner class.

Recap: This Design is not Good.

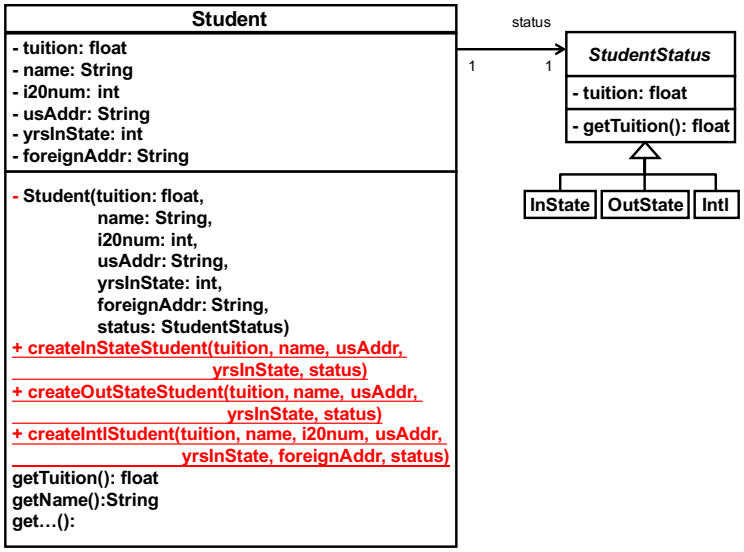


- Alternative designs
  - Use an enumeration
  - Use *Static Factory Method* and an enumeration

w/ *Static Factory Method*



w/ *Static Factory Method and State*



HW 9-2

- Complete the designs in Slides #38 and #39 and implement them.
  - Write test cases too.
  - Optional: Implement a factory class
    - Separate `StudentFactory` and `Student`

Anotehr Example: Matchers in hamcrest-all.jar

- `org.hamcrest.CoreMatchers`
- `org.hamcrest.Matchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `Matchers` is a superset of `CoreMatchers`.