LOYOLA
UNIVERSITY
CHICAGO

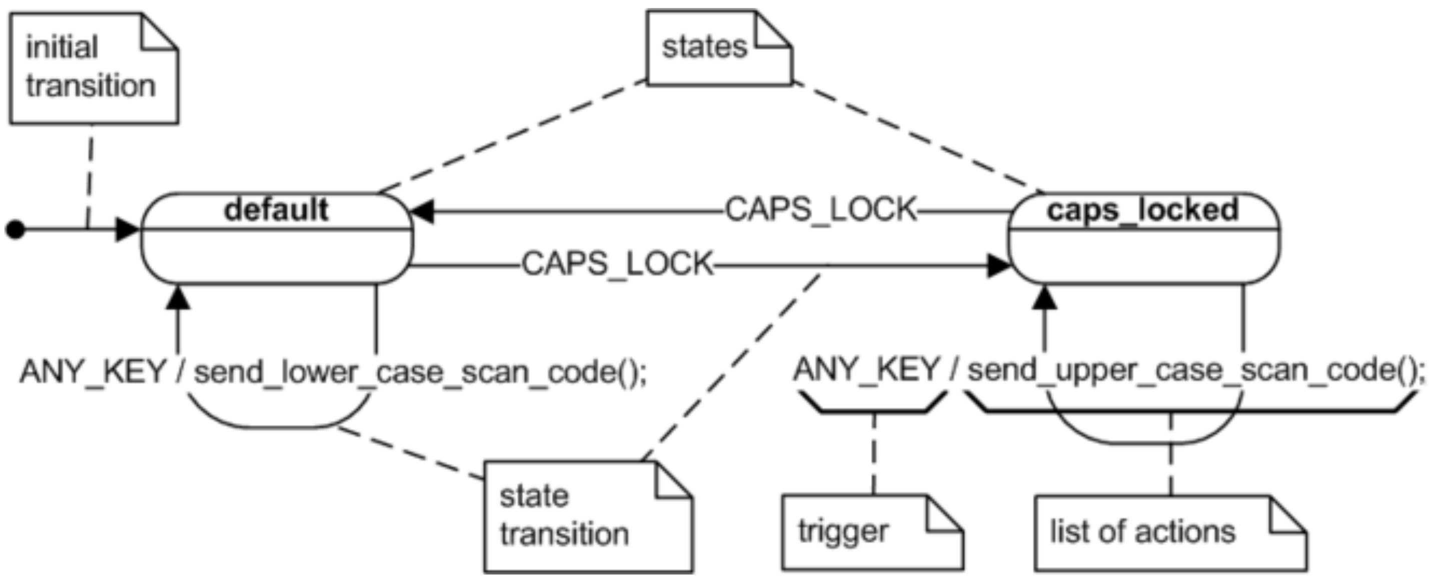
COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Week 12 Topics

- Test2 Review (if needed)
- Project 4 State Machine Exercise
  - UML Extended State Diagrams and Alternative Representation
  - Some Project 4 details
  - Group exercise to create a Project 4 Extended State Machine
- Design Smells and Refactoring
- A few Test 3 Items
  - MVA in clickcounter and stopwatch
  - MVP and MVVM
- Possibly Time to Work on Project 4 in your Groups
- Extra Topic – Immutability

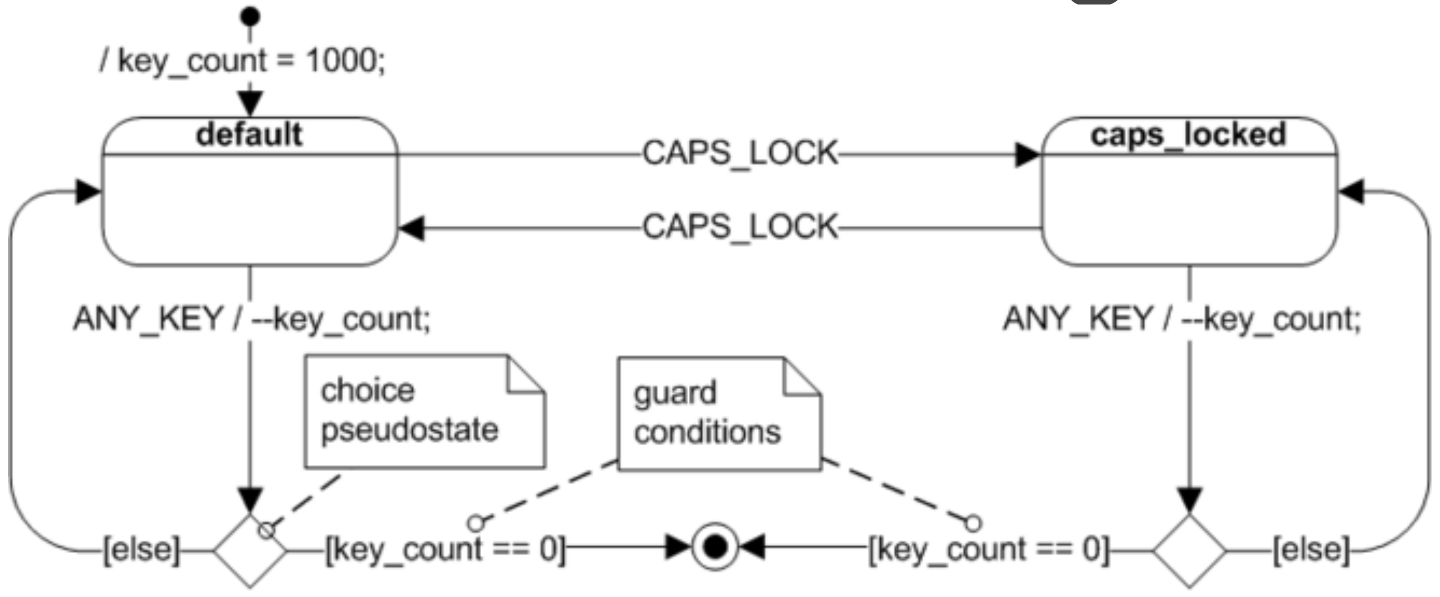
Basic UML State Diagrams



UML state diagrams are directed graphs in which nodes denote states and connectors denote state transitions. For example, Figure 1 shows a UML state diagram corresponding to the computer keyboard state machine. In UML, states are represented as rounded rectangles labeled with state names. The transitions, represented as arrows, are labeled with the triggering events followed optionally by the list of executed actions. The **initial transition** originates from the solid circle and specifies the default state when the system first begins. Every state diagram should have such a transition, which should not be labeled, since it is not triggered by an event. The initial transition can have associated actions.

Project 4 has two kinds of events, clicking the button, and the clock ticking.

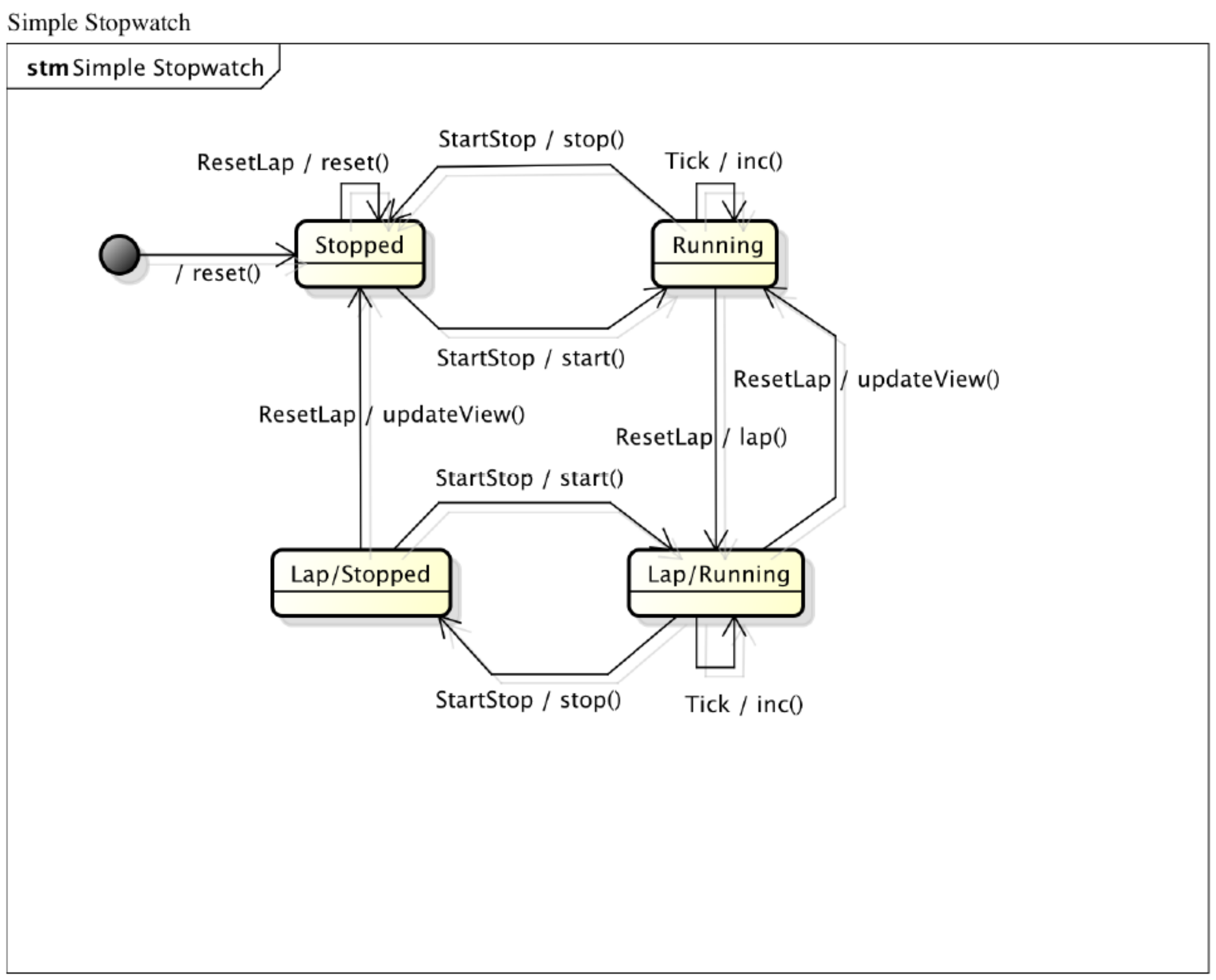
Extended UML State Diagrams



This is an example of an extended state machine, in which the complete condition of the system (called the extended state) is the combination of a qualitative aspect —the "state"—and the quantitative aspects—the extended state variables (such as the down-counter **key_count**). This keyboard “dies” after 1000 keystrokes.
The obvious advantage of extended state machines is flexibility: for example, extending the lifespan of this "cheap keyboard" from 1,000 to 10,000 keystrokes would not complicate the extended state machine at all.
This also shows the use of **guard conditions**, boolean expressions based on the value of extended state variables, like **key_count**, and/or event parameters.
Time-based guard conditions will prove useful in the Project 4 state diagram.

[stopwatch-android-java](#)

Reminder: stopwatch State Machine



Alternative Representation for the Stopwatch State Machine

<u>state</u>: triggering event [optional guard] / action → next state

stopwatch **example** (note: <u>all</u> transitions call update_view(), not shown):

States: **Stopped, Running, Lap/Running, Lab/Stopped**

Events: **StartStop, ResetLap, Tick** (ignored if not listed for a state)

Extended state variables: <u>none</u>

<u>start</u>: *none* / reset() → **Stopped; <u>Stopped</u>**: ResetLap / reset() → **Stopped**

<u>Stopped</u>: StartStop / start() → **Running; <u>Running</u>**: StartStop / stop() → **Stopped**

<u>Running</u>: Tick / inc() → **Running; <u>Running</u>**: ResetLap / lap() → **Lap/Running**

<u>Lap/Running</u>: ResetLap / updateView → **Running**

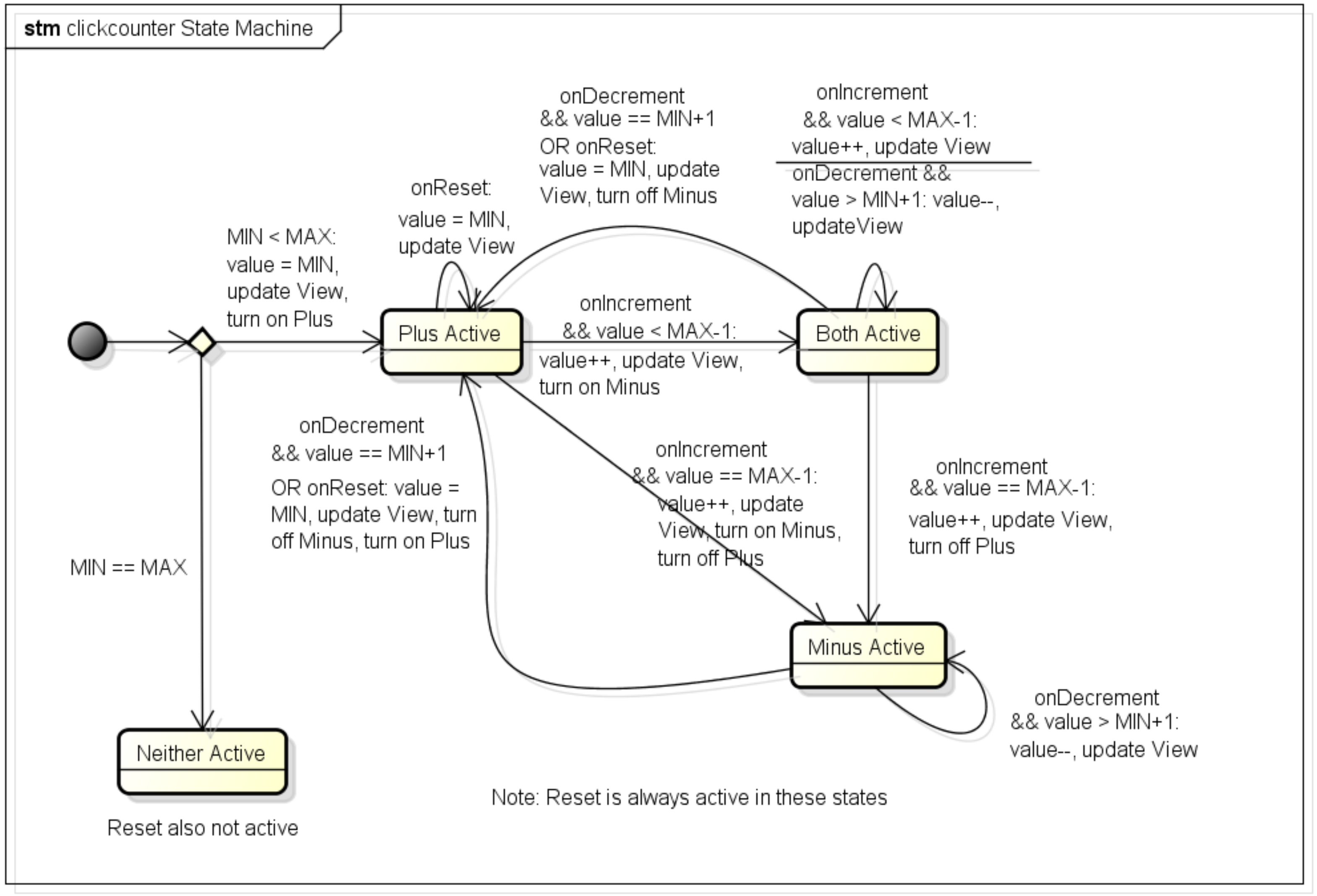
<u>Lap/Running</u>: Tick / inc() → **Lap/Running**

<u>Lap/Running</u>: StartStop / stop() → **Lap/Stopped**

<u>Lap/Stopped</u>: StartStop / start() → **Lap/Running**

<u>Lap/Stopped</u>: ResetLap / updateView() → **Stopped**

Implicit clickcounter State Machine


stm clickcounter State Machine
MIN < MAX: value = MIN, update View, turn on Plus
MIN == MAX
Neither Active
Reset also not active
Plus Active
onReset: value = MIN, update View
onDecrement && value == MIN+1 OR onReset: value = MIN, update View, turn off Minus
onIncrement && value < MAX-1: value++, update View, turn on Minus
Both Active
onIncrement && value < MAX-1: value++, update View
onDecrement && value > MIN+1: value--, updateView
onDecrement && value == MIN+1 OR onReset: value = MIN, update View, turn off Minus, turn on Plus
onIncrement && value == MAX-1: value++, update View, turn on Minus, turn off Plus
onIncrement && value == MAX-1: value++, update View, turn off Plus
Minus Active
onDecrement && value > MIN+1: value--, update View
Note: Reset is always active in these states

powered by Astah

Alternative Representation for the Clickcounter State Machine

state: triggering event [optional guard] / action →   next state

clickcounter example (note: all transitions call update_view()):

States: **start**, **inc (INCrement active)**, **dec (DECrement active)**, **both**

Events: **INC (increment), DEC (decrement), RESET**

Extended state variables: **value**; **MIN**, **MAX** (constants)

start: *none* / value = MIN, INC active, DEC inactive →   **inc**

inc or both: INC [value < MAX-1] / value++, DEC active →   **both**

inc: INC [value == MAX-1] / value++, DEC active, INC inactive →   **dec**

dec or both : DEC [value > MIN+1] / value-- →   **both**

dec or both: DEC [value == MIN+1] / value = MIN, INC active, DEC inactive → **inc**

all states: RESET / value = MIN, INC active, DEC inactive →   **inc**

Project 4 – Functional Requirements – Slide 1 of 2

- The timer has the following controls:
  - One two-digit display of the form 88.
  - One multi-function button. **(Clicking the button causes a click event.)**
- The timer behaves as follows (part 1 of 2):
  - The timer always displays the remaining time in seconds.
  - Initially, the timer is stopped and the (remaining) time is zero.
  - If the button is pressed when the timer is stopped, the time is incremented by one up to a preset maximum of **99**.
    (The button acts as an **increment** button.)
  - If the time is greater than zero and three seconds elapse from the most recent time the button was pressed, then the timer beeps once and starts running. **(When the remaining time is greater than 0 the clock model (see** stopwatch**) is used to send tick events to the state machine.)**

Project 4 – Functional Requirements – Slide 2 of 2

- The timer behaves as follows (part 2 of 2):
  - While running, the timer subtracts one from the time for every second that elapses. **(Caused by a clock model tick event.)**
  - If the timer is running and the button is pressed, the timer stops and the time is reset to zero. (The button acts as a **cancel** button.)
  - If the timer is running and the time reaches zero by itself (without the button being pressed), then the timer stops and the alarm starts beeping continually and indefinitely.
  - If the alarm is sounding and the button is pressed, the alarm stops sounding; the timer is now stopped and the (remaining) time is zero. (The button acts as a **stop** button.)
  - The timer handles rotation by continuing in its current state.
- **Your Groups will now create an extended state machine diagram for Project 4’s Simple Timer app**

LOYOLA UNIVERSITY CHICAGO

Working on the P4 State Machine

- You'll continue to work in these Groups (from the UML exercise):

Name	Group
Killham, Eric John	1
Tapia, Rene	1
Cicale, Julia	1
Rodriguez Orjuela, Jose Luis	1
Mir, Sarfaraz Ali Khan	1
Nowreen, Syeda Tashnuva	1
Goel, Neha	2
Soliz Rodriguez, Percy Gabriel	2
Misra, Anadi	2
Pacheco, Andrea	2
Mehta, Shipra Ashutosh	2
Sindhu, Pinky	2
Al Khofi, Sundas Abdullatif A	3
Liu, Siyuan	3
Meghrajani, Aman Maheshkumar	3
Oakey, Anthony William	3

- You can use **Ar** **is.org/** for this – you may want to dr nd first

Project 4 – Events

There are only two kinds of events in Project 4:

- Button Click events (always processed in all states)
- Clock Tick events (ignored in stopped and alarm states unless a Clock Tick is used to repeatedly play a “beep” for the alarm)

You have 45 minutes to work on the state machine now, followed by a 10-minute break

Submit your State Machine using the Sakai Project 4 State machine Extra Credit Assignment (worth 2%)

LOYOLA UNIVERSITY CHICAGO

Week 12 Topics

- **Design Smells and Refactoring**
  - **Software Testing & TDD**
  - Code Smells, Design Smells, and Refactoring

Software Testing and TDD → Lead-In to Refactoring

- **Software testing in an Agile context**
  - general overview of software testing
  - test-driven development
  - continuous integration/delivery

Week 12 Topics

- **Design Smells and Refactoring**
  - Software Testing & TDD
  - **Code Smells, Design Smells, and Refactoring**
    - **See online course schedule for topics, in particular, this presentation, plus the following slides**

Code Smells (Like Anti-Patterns)

- **A *code smell* is a hint that something has gone wrong somewhere in your code. Use the smell to track down the problem.**
  - Kent Beck (with inspiration from the nose of Massimo Arnoldi) seems to have coined the phrase in the "OnceAndOnlyOnce" page, where he also said that code "wants to be simple". *Bad Smells in Code* was an essay by Kent Beck and Martin Fowler, published as Chapter 3 of RefactoringImprovingTheDesignOfExistingCode.
  - Highly experienced and knowledgeable developers have a "feel" for good design. Having reached a state of "UnconsciousCompetence," where they routinely practice good design without thinking about it, they find that they can look at a design or the code and immediately get a "feel" for its quality, without getting bogged down in "logically detailed arguments".
- Note that a code smell is a *hint* that something *might* be wrong, not a certainty. A perfectly good idiom may be considered a code smell because it's often misused, or because there's a simpler alternative that works in most cases. Calling something a code smell is not an attack; it's simply a sign that a closer look is warranted.

Source: http://c2.com/xp/CodeSmell.html

Code Smells – Some Examples

Too much code, time to take something off the stove before it boils over:

- **Duplicated Code** - See: OnceAndOnlyOnce, SwitchStatementsSmell
- **Methods too big**: ComposedMethod, TallerThanMe, LongMethodSmell
- Classes with too many instance variables: ExtractComponent, ValueObject, and WholeValue
- **Classes with too much code**: OneResponsibilityRule / Single Responsibility Principle (SRP), GodClass
- **Strikingly similar subclasses**: BehaviorToState
- Parallel Inheritance Hierarchies - Violates OnceAndOnlyOnce.
- An instance variable that is only set in some circumstances
- Comparing variables to null: NullObject, NullConsideredHarmful
- Too many private (or protected) methods - MethodsShouldBePublic
- **Many messages to the same object from the same method**: MoveMethod

Code Smells – More Examples

Not enough code, better put the half-baked code back in the oven a while:

- Classes with too few instance variables
- **Classes with too little code** - See: OneResponsibilityRule / SRP
- Methods with no messages to self - See: UtilityMethod
- **Empty catch clauses** - See: EmptyCatchClause
- **Explicitly setting variables to null**: can indicate that either there are references to things that this code has no business referencing, or the structure is so complex that the programmer doesn't really understand it and feels the need to do this to be safe

Code Smells – … Examples

Not actually the code:

- Preprocessor Statements
- **Comments** - See: ToNeedComments, TooMuchDocumentation
- Excessive Logging - Lots of logs are needed to figure out what the heck the code is doing!
- **Boredom** - If you're bored, you might be doing something wrong

Problems with the way the code is changing:

- Sporadic Change Velocity - Different rates of change in the same object.
- **Shot Gun Surgery** - The same rate of change in different objects, particularly if they are disconnected. A conceptually simple change that requires modification to code in many places. The result of CopyAndPasteProgramming.

Code Smells – … Examples

Some other code problems:

- Contrived Interfaces - A smell of PrematureGeneralization.
- **Asymmetrical Code/Imbalance**
- Variable Clumps
- Dependency cycles among packages or classes within a package.
- **Concrete classes that depend on other concrete classes** (vs. abstractions)
- Methods requiring Special Formatting to be readable
- Back Pedaling (loss of context)
- **Long method names**. Seriously: If you follow good naming standards, long method names are often an indication that the method is in the wrong class.
- **Vague Identifiers**
- Procedural code masquerading as objects.
- Embedded code strings such as large chunks of SQL, HTML or XML.
- **Passing Nulls To Constructors** - use a Factory Method instead

Code Smells – … Examples

Some other code problems:

- **Too Many Parameters, Long Parameter List**
- Variable Name Same As Type
- While Not Done Loops
- **False unification of procedures**. A procedure, function, or method has a boolean that provides a variation on its behavior, but the two variations in fact have completely different semantics. It would be better to refactor the common code into another method and split the remainder into two separate methods, removing the boolean.
- False unification of interfaces. An interface has two implementors. One implementor implements half of the interface and throws an Unsupported Operation Exception for the other half. The 2nd implementor implements the other half and throws an Unsupported Operation Exception for the 1st half. It would be better to split the interface into two. Similar to RefusedBequest.
- **Hidden coupling**. Code depends on completely non-obvious characteristics, such as relying on reference equality instead of value equality.

Eliminating Code Smells → Refactoring

What is Refactoring?

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior-preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Source: http://refactoring.com/

See also: http://www.refactoring.com/catalog/

and http://sourcemaking.com/refactoring

Refactoring Catalog Examples – 1

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction
- Convert Static to Dynamic Construction
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier

Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression.

→ **Move it outside of the expression.**

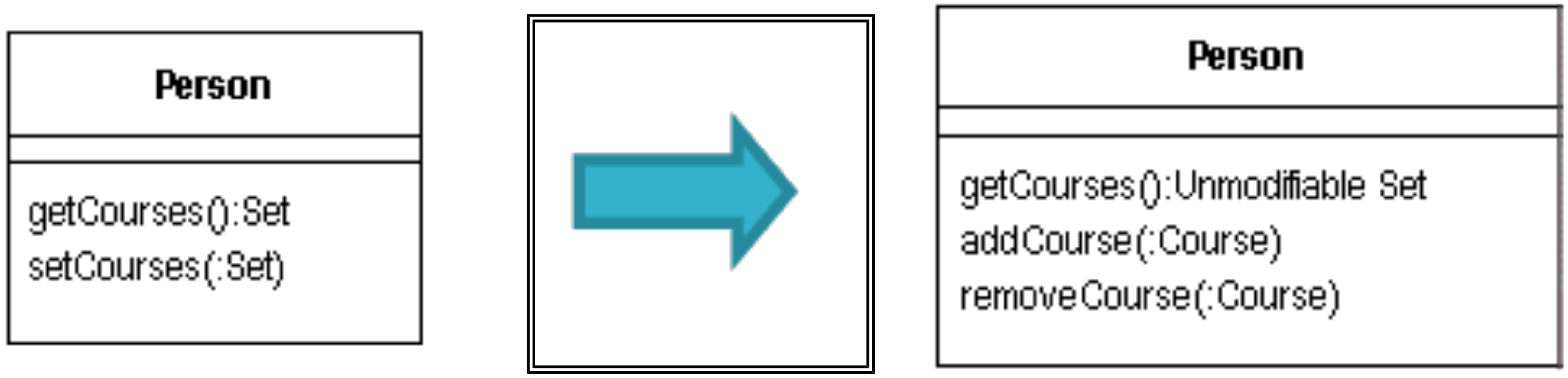
```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Encapsulate Collection

A method returns a Collection.

→ **Make it return a read-only view and provide add/remove methods.**



Encapsulate Field

There is a public field.

→ **Make it private and provide accessors and mutators.**

```
public String name;          private String _name;

                     public String ge
                       { return _name; }
                     public String setName( String arg )
                       { _name = arg; }
```

Refactoring Catalog Examples – 2

- Inline Class
- Inline Method
- Inline Temp
- Introduce A Controller
- Introduce Assertion
- Introduce Business Delegate
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- **Introduce Parameter Object**
- Introduce Synchronizer Token
- Localize Disparate Logic
- Merge Session Beans
- Move Business Logic to Session
- Move Class
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable
- Refactor Architecture by Tiers
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Double Negative

Refactoring Catalog Examples – 3

- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Replace Assignment with Initialization
- Replace Conditional with Polymorphism
- Replace Conditional with Visitor
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Iteration with Recursion
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional w/Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Recursion with Iteration
- Replace Static Variable with Parameter
- Replace Subclass with Fields
- Replace Temp with Query
- Replace Type Code with Class
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Reverse Conditional

Refactoring Catalog Examples – 4

- Self Encapsulate Field
- Separate Data Access Code
- Separate Query from Modifier
- Split Loop
- Split Temporary Variable
- Substitute Algorithm
- Use a Connection Pool
- Wrap entities with session

Some Other Example Code Smell Refactorings

- **Long method → replace method with method object**
  - That is, create a class that defines that method, and have the original class create a class object and use its method (composition)
- **Long parameter list → introduce parameter object**
  - Similar to long method above – create a new class and give its constructor the parameter list information, then create an object from that class and pass that object to the original method
  - This may be replaced by a lambda in Java 8
- **Duplicate code → extract method**
  - That is, define a (parameterized) method containing the duplicate code and call it in the various places where that code used to be

Note: IntelliJ IDEA supports some <u>code refactorings</u> directly

LOYOLA UNIVERSITY CHICAGO

Week 12 Topics

- Test2 Review (if needed)
- Project 4 State Machine Exercise
  - UML Extended State Diagrams and Alternative Representation
  - Some Project 4 details
  - Group exercise to create a Project 4 Extended State Machine
- Design Smells and Refactoring
- A few Test 3 Items
  - MVA in clickcounter and stopwatch
  - MVP and MVVM
- Possibly Time to Work on Project 4 in your Groups
- Extra Topic – Immutability

LOYOLA UNIVERSITY CHICAGO

Possible Time to Work on Project 4

Extra Topic – Immutability

Immutable Objects

From Wikipedia, the free encyclopedia

- In object-oriented and functional programming, an **immutable object** is an object whose state cannot be modified after it is created. This is in contrast to a **mutable object**, which can be modified after it is created.
- An object can be either entirely immutable or some attributes in the object may be declared immutable. *In some cases, an object is considered immutable even if some internally used attributes change, but the object's state appears to be unchanging from an external point of view.* For example, an object that uses memoization to cache the results of expensive computations could still be considered an immutable object.
- The initial state of an immutable object is usually set at its inception, but can also be set before actual use of the object (lazy initialization).
- Immutable objects are often useful because some costly operations for copying and comparing can be omitted, simplifying the program code and speeding execution. However, making an object immutable is usually inappropriate if the object contains a large amount of changeable data. Because of this, many languages allow for both immutable and mutable objects.
- See http://xpadro.blogspot.com/2014/08/java-concurrency-tutorial-thread-safe.html for information about using immutable objects for thread safe design

Immutable Objects – Benefits

- Immutable objects are simply objects whose state (the object's data) cannot (visibly) change after construction. Examples of immutable objects from the Java API include String and Integer (actually, any Java wrapped primitive is immutable).
- Immutable objects greatly simplify your program, since they :
  - are simple to construct, test, and use
  - **are automatically thread-safe and have no synchronization issues**
  - do not need a copy constructor
  - do not need an implementation of clone
  - allow hashCode to use lazy initialization, and to cache its return value
  - **do not need to be copied defensively when used as a field**
  - **make good Map keys and Set elements (these objects must not change state while in the collection)**
  - have their class invariant established once upon construction, and it never needs to be checked again
  - **always have "failure atomicity" (a term used by Joshua Bloch): if an immutable object throws an exception, it's never left in an undesirable or indeterminate state**

Source: http://www.javapractices.com/topic/TopicAction.do?Id=29

LOYOLA UNIVERSITY CHICAGO

Immutable Objects – How To

Make a class immutable by following these guidelines :

- **ensure the class cannot be overridden** – make the class final, or use static factories and keep constructors private
- **make fields private and final**
- **force callers to construct an object completely in a single step**, instead of using a no-argument constructor combined with subsequent calls to setXXX methods (that is, avoid the Java Beans convention) or instead of using a Builder Pattern
- **do not provide any methods which can change the state of the object in any way** – not just setXXX methods, but any method which can change state
- **if the class has any mutable object fields, then they must be defensively copied when passed between the class and its caller, either incoming or outgoing**
  - Incoming parameter objects with mutable fields must be copied before being put into instance reference variables
  - Outgoing returned objects with mutable fields that come from reference variables must be copied before being returned

LOYOLA UNIVERSITY CHICAGO

Special Case – Unmodifiable Collections

If you need an immutable Collection of some kind, use one of these static methods:

- static <T> Collection<T> **unmodifiableCollection**(Collection<? extends T> c)
  Returns an unmodifiable view of the specified collection (ie, an immutable view).
- static <T> List<T> **unmodifiableList**(List<? extends T> list)
  Returns an unmodifiable view of the specified list.
- static <K,V> Map<K,V> **unmodifiableMap**(Map<? extends K,? extends V> m)
  Returns an unmodifiable view of the specified map.
- static <T> Set<T> **unmodifiableSet**(Set<? extends T> s)
  Returns an unmodifiable view of the specified set.
- static <K,V> SortedMap<K,V> **unmodifiableSortedMap**(SortedMap<K,? extends V> m)
  Returns an unmodifiable view of the specified sorted map.
- static <T> SortedSet<T> **unmodifiableSortedSet**(SortedSet<T> s)
  Returns an unmodifiable view of the specified sorted set.
- static <T> List<T> **emptyList**() – returns the (immutable) empty list. Also: Collections.EMPTY_LIST.
- static <K,V> Map<K,V> **emptyMap**() – returns the (immutable) empty map. Also: Collections.EMPTY_MAP.
- static <T> Set<T> **emptySet**() – returns the (immutable) empty set. Also: Collections.EMPTY_SET.

Example: if myList is a List, then the following statement yields an immutable view of myList …

List immutableMyList = Collections.unmodifiableList(myList); // see uidemo

LOYOLA UNIVERSITY CHICAGO

Other Useful Collections Methods

Singletons:

- static <T> Set<T> **singleton**(T o)
  Returns an immutable set containing only the specified object.
- static <T> List<T> **singletonList**(T o)
  Returns an immutable list containing only the specified object.
- static <K,V> Map<K,V> **singletonMap**(K key, V value)
  Returns an immutable map, mapping only the specified key to the specified value.

"Reversed" Collections:

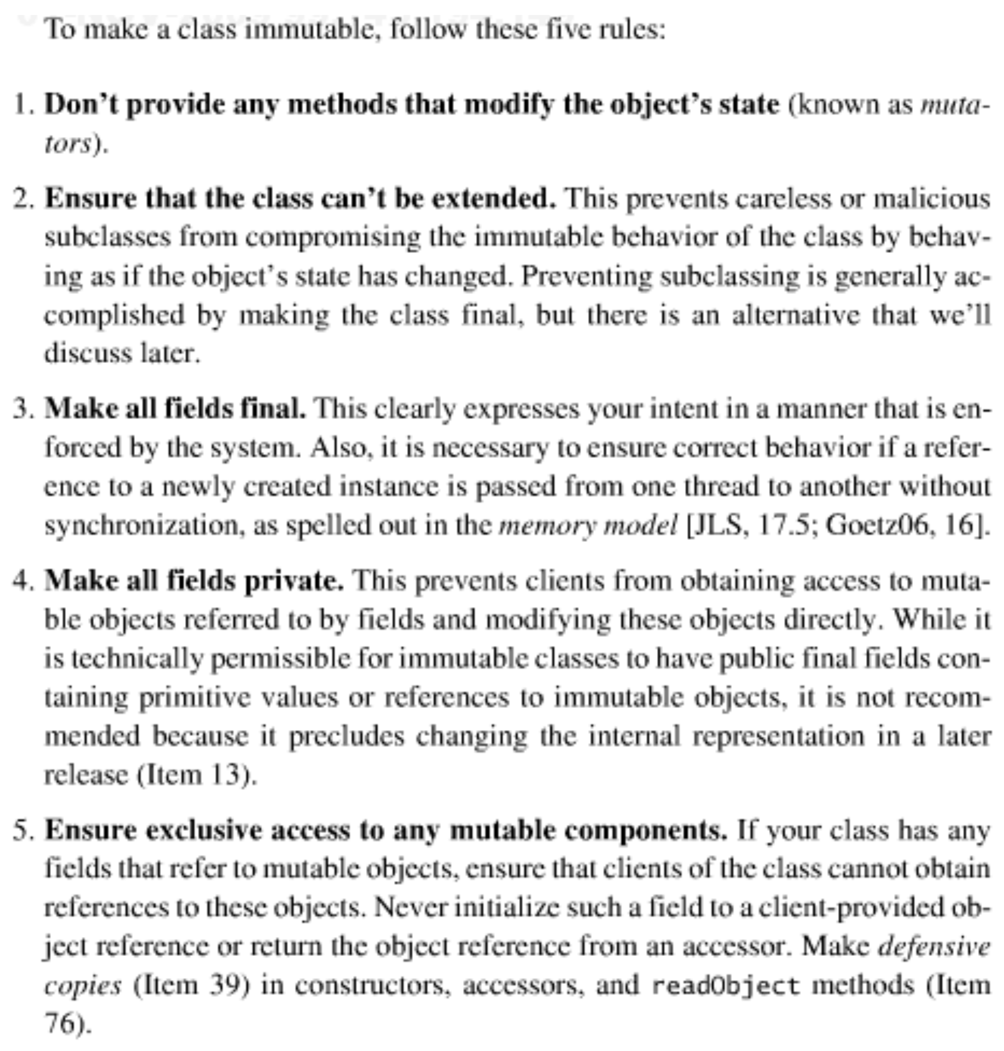
- static void **reverse**(List<?> list)
  Reverses the order of the elements in the specified list.
- static <T> Comparator<T> **reverseOrder**()
  Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.
- static <T> Comparator<T> **reverseOrder**(Comparator<T> cmp)
  Returns a comparator that imposes the reverse ordering of the specified comparator.

Check out the Collections online documentation for even more useful static methods!

Source: http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Bloch on Immutability – Item 15

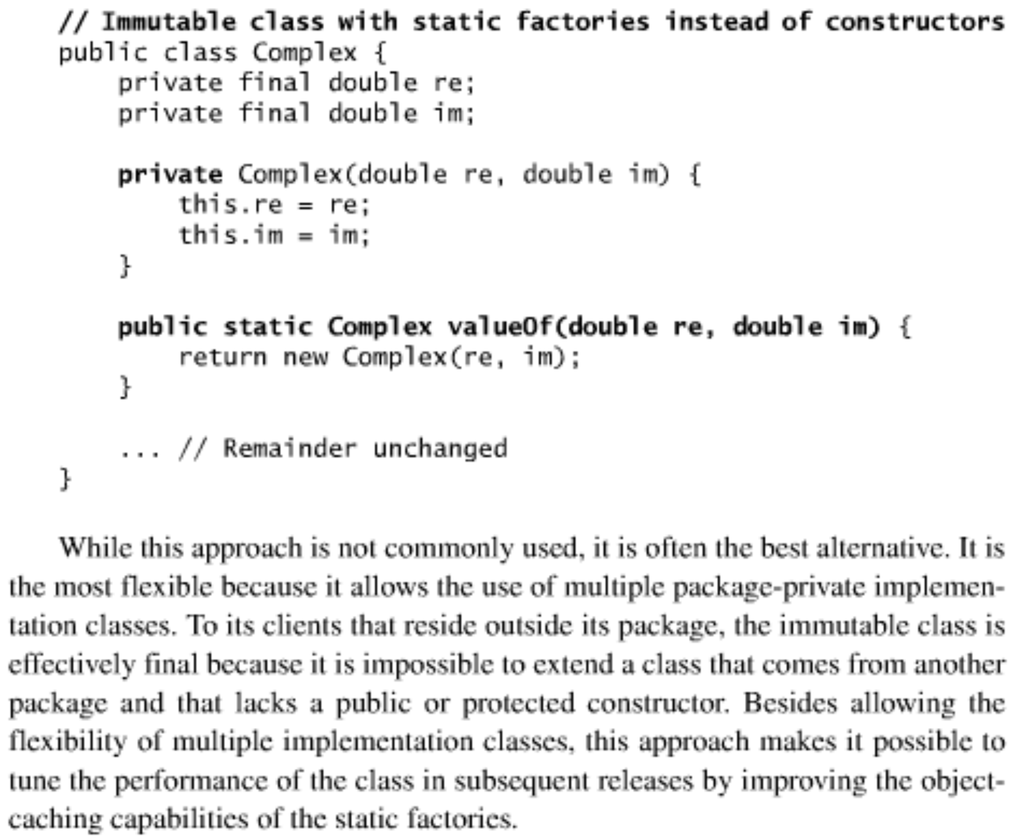
How to minimize mutability:


To make a class immutable, follow these five rules:

1. **Don't provide any methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class final, but there is an alternative that we'll discuss later.
3. **Make all fields final.** This clearly expresses your intent in a manner that is enforced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06, 16].
4. **Make all fields private.** This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly. While it is technically permissible for immutable classes to have public final fields containing primitive values or references to immutable objects, it is not recommended because it precludes changing the internal representation in a later release (Item 13).
5. **Ensure exclusive access to any mutable components.** If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects. Never initialize such a field to a client-provided object reference or return the object reference from an accessor. Make *defensive copies* (Item 39) in constructors, accessors, and `readObject` methods (Item 76).

Bloch on Immutability – Item 15

How to make a class non-subclassable using static factories:

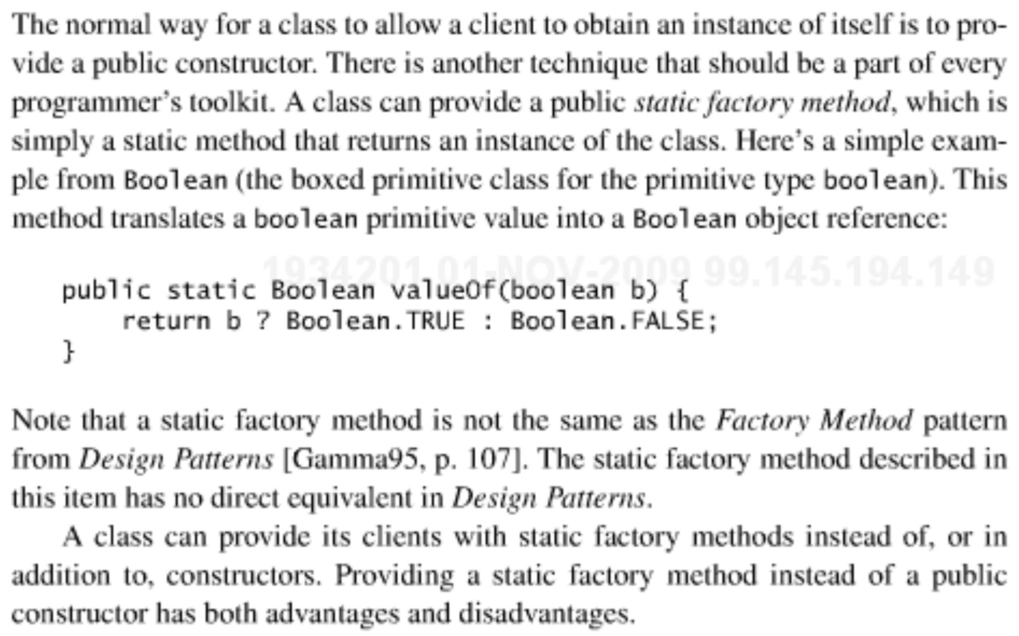


```
// Immutable class with static factories instead of constructors
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
}
```

While this approach is not commonly used, it is often the best alternative. It is the most flexible because it allows the use of multiple package-private implementation classes. To its clients that reside outside its package, the immutable class is effectively final because it is impossible to extend a class that comes from another package and that lacks a public or protected constructor. Besides allowing the flexibility of multiple implementation classes, this approach makes it possible to tune the performance of the class in subsequent releases by improving the object-caching capabilities of the static factories.

Bloch on Immutability – Item 1

What are static factory methods, and how are they used?



The normal way for a class to allow a client to obtain an instance of itself is to provide a public constructor. There is another technique that should be a part of every programmer's toolkit. A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class. Here's a simple example from `Boolean` (the boxed primitive class for the primitive type `boolean`). This method translates a `boolean` primitive value into a `Boolean` object reference:

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Note that a static factory method is not the same as the *Factory Method* pattern from *Design Patterns* [Gamma95, p. 107]. The static factory method described in this item has no direct equivalent in *Design Patterns*.

A class can provide its clients with static factory methods instead of, or in addition to, constructors. Providing a static factory method instead of a public constructor has both advantages and disadvantages.

Advantages of static factory methods over constructors, …

- they have names, so differently named static factory methods can return different (types of) objects even if they are given the same parameters
- they are not required to create a *new* object each time they are invoked
- they can return an object of any subtype of their return type