Until now…

- You didn't have to use Ant when you did unit testing with JUnit.
- You didn't have to use JUnit when you used Ant (build-calc.xml).

Running Unit Tests in a Build Process

- From now, you will use BOTH Ant and JUnit.
  - You will run your Ant build script to
    - Compile your program(s),
      - e.g., `PrimeGenerator.java`
    - Compile your test class(es),
      - e.g., `PrimeGeneratorTest.java`
    - Run your test class(es) with Junit, and
      - e.g., `PrimeGeneratorTest.class`
    - Run your program (optional)
      - e.g., `PrimeGenerator.class`
- Your code base requires, or depends on, JUnit library files (i.e., JUnit JAR files).
  - Your Ant build script needs to reference those JAR files and set their paths into CLASSPATH
    - so that it can compile and run your test classes

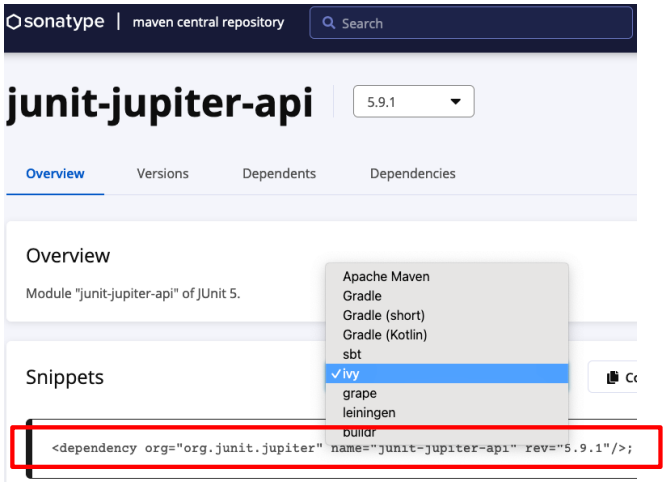
JUnit JAR Files

- JUnit API JAR files
  - `junit-jupitar-api.jar`
  - `junit-jupitar-engine.jar`
  - `junit-jupitar-params.jar`
  - `apiguardian-api.jar`
  - `opentest4j.jar`
  - DO NOT use `junit-vintage-*.jar` (Those are for JUnit version 4.)
- JUnit Platform JAR files
  - `junit-platform-commons.jar`
  - `junit-platform-engine.jar`
  - `junit-platform.launcher.jar`
  - `junit-platform-runner.jar:`
  - `junit-platform-suite-api.jar`
- Your Ant build script needs to reference ALL these JAR files and set their paths in CLASSPATH.

How to Make JUnit JAR files Available for Your Build Script

- Use Apache Ivy with Ant
  - https://ant.apache.org/ivy/
  - Install it in addition to Ant (c.f. Ivy's reference manual)
- Ivy does "dependency management" for Ant.
  - Ivy allows your Ant build script to download any external library files (JAR files) to the "lib" directory, by default.
    - so your build script can set their paths into CLASSPATH.
  - ivy.xml lists the JAR files you want to use.
  - Your build script states `<ivy:retrieve/>`
    - This task will download the JAR files listed in ivy.xml into the "lib" directory.

- In ivy.xml, you specify each JAR file you want to use with `<dependency>`.
  - <dependency org="…"
    name="…"
    rev="…" />
- Search each JAR file you want at https://search.maven.org/



- Your build script:
  - `<ivy:retrieve/>`

```
<path id="classpath">
  <pathelement location="bin" />
  <pathelement location="test/bin" />
  <fileset dir="lib">
</path>

<javac …>
<classpath refid="classpath"/>
    …
</javac>

<junitlauncher …>
<classpath refid="classpath"/>
    …
</junitlauncher>
```

  - Useful task (for debugging) to print CLASSPATH
    - `<echo message="${toString:classpath}" />`

- Use the <junitlauncher> task in Ant to run JUnit.
  - Refer to Ant's and JUnit's manuals.
    - https://ant.apache.org/manual/Tasks/junitlauncher.html

```
- <junitlauncher printSummary="true">
    <classpath refid="classpath" />
    <test outputdir="test"
          name="edu.umb.cs680.hw01.CalculatorTest"/>
    <listener type="legacy-plain" sendSysOut="true"/>
  </junitlauncher>

- <junitlauncher printSummary="true">
    <classpath refid="classpath" />
    <testclasses outputdir="test">
      <fileset dir="${test.bin}">
        <include name="edu/umb/cs680/hw01/*Test.class"/>
      </fileset>
      <listener type="legacy-plain" sendSysOut="true"/>
    </testclasses>
  </junitlauncher>
```

  - Use this (the second) option.

Exercise

- Extend your ex1 solution. Set up the following directory structure
  - <proj dir>
    - ex1.xml
    - ivy.xml
    - src [source code directory]
      - edu/umb/cs680/ex1/Calculator.java
    - bin [binary code directory]
      - Calculator.class will be placed under this directory.
    - test [test code directory]
      - src
        - » edu/umb/cs680/ex1/CalculatorTest.java
      - bin
        - » CalculatorTest.class will be placed under this directory.

- Do: ant -f ex1.xml
  - Will create the "lib" directory.
  - Will download required JAR files into the "lib" directory.
  - Will add the "lib" directory to CLASSPATH
  - Compile source code.
  - Run a test class.

HW 1

- Re-do ex2 with Ant, Ivy and JUnit.
  - In ex2, you didn't use Ant and Ivy.
  - Use Ant and Ivy to
    - Download JUnit JAR files.
    - Do pre-compilation configurations (e.g., directory structures, CLASSPATH settings, etc.)
    - Compile `PrimeGenerator` and `PrimeGeneratorTest`
    - Run test cases (test methods) in `PrimeGeneratorTest`
  - Follow the directory structure you used for HW 1.

- **FIRM** deadline: October 31 (Tue), midnight
  - For each HW, do your best to turn in your solution sometime soon (e.g. in a week or two).
    - Will give you extra points if you regularly do so.

Expected Directory Structure

- <proj dir>
  - build.xml
  - ivy.xml
  - src [source code directory]
    - edu/umb/cs680/hw01/PrimeGenerator.java
  - bin [binary code directory]
    - PrimeGenerator.class will be placed under this directory.
  - test [test code directory]
    - src
      - » edu/umb/cs680/hw01/PrimeGeneratorTest.java
    - bin
      - » PrimeGeneratorTest.class will be placed under this directory.

- Make sure that your build script runs properly.
  - on both your shell and IDE

- Turn in:
  - build script
  - ivy.xml
  - "src" directory
  - "test/src" directory

- DO NOT include binary files (the "lib" directory and .class files).
  - Configure `.gitignore` if you like.
  - I will NOT grade your work if you turn in binary files.

Important Notice

- You must work alone for each HW.
  - You can discuss HW assignments with others. However, you must work on your coding yourself.
- "Alone" means:
  - DO NOT start with anyone else's code.
  - Write your code from scratch.
- It is an academic crime to
  - Copy (or steal) someone else's code and submit it as your own work.
  - Allow someone else to copy (or steal) your code and submit it as his/her work.
    - Use a private repo to avoid this.
- You will end up with a serious situation if you commit this crime.
  - The University, College, Department and I have no mercy about it.

Design Patterns

Design Patterns

- Tested, proven and documented solutions for recurring design problems in given contexts.
- Benefits
  - Useful information source to learn and practice good designs
  - Useful communication tool among developers
    - c.f. Recursion, collections (array, list, set, map, etc.), sorting, buffers, infinite loops, integer overflow, polymorphism, etc.

Recap: Brief History to OOD

- In good, old days… programs had no structures.
  - One dimensional code.
- As the size and complexity of software increased, programming languages needed structures (or modularity).
  - Module: A chunk of code
  - Modularity: Making modules self-contained and independent from others
  - Goal: Improve productivity and maintainability
    - Higher productivity through higher reusability
      - Lead to less production costs
    - Higher maintainability through clearer separation of concerns
      - Lead to less maintenance costs

Modules in SLs and OOPLs

- Modules in Structured Prog. Languages (SPLs)
  - Structure = a set of variables
  - Function = a block of code
- Modules in Object-Oriented PLs (OOPLs)
  - Class = a set of variables (data fields) and functions (methods)
  - Interface = a set of functions (methods)
- Key design questions/challenges:
  - how to define modules?
  - how to separate a module from others?
  - how to let modules interact with each other?

OOPLs v.s. SPLs

- OOPLs
  - Intend coarse-grained modularity
    - The size of each module is often bigger in OOPLs.
  - Extensibility in mind to enhance productivity and maintainability further.
    - How easy (cost effective) to add and revise existing modules (classes and interfaces) to implement new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to leverage modularity to address reusability, maintainability and extensibility?
    - Design patterns can give you good examples.

A Key Topic in CS680

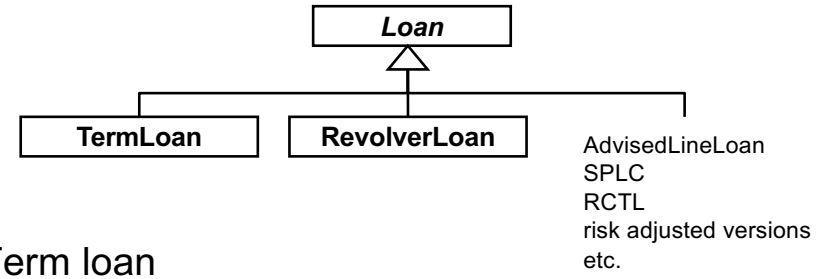
- Understand how to address reusability, maintainability and extensibility
  - By improving the *design* and *organization* of programs through design patterns

Static Factory Method

Static Factory Method

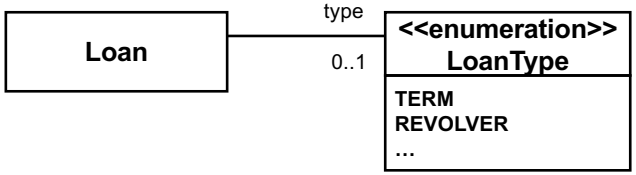
- Intent
  - Define a "communicable" method that instantiates a class
    - Constructors are the methods to instantiate a class.
    - Static factory methods are more "communicable" (more maintainable) than constructors.
- Benefits
  - Static factory methods have their own names (!).
    - Improve code maintainability.
      - The name can explicitly tell what object to be created/returned.
      - A class to be instantiated and client code gets easier to understand.

Recap: This Design is not Good.

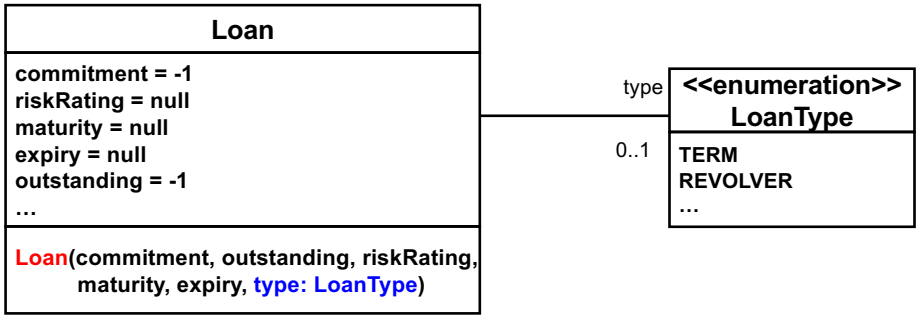


- Term loan
  - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
  - With a spending limit and expiry date
- Dynamic class change problem
  - A revolver can transform into a term loan when the revolver expires.

Enumeration-based Design



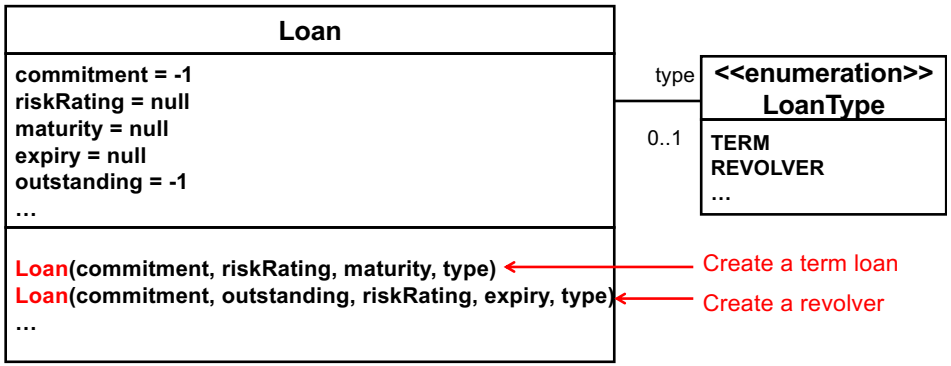
- A class inheritance should not be used here.



- Different loans need different sets of data to be set up.
  - A term loan needs commitment, risk rating and maturity date.
  - A revolver needs commitment, outstanding debt, risk rating and expiry date.
- The constructor is hard to understand and error-prone.

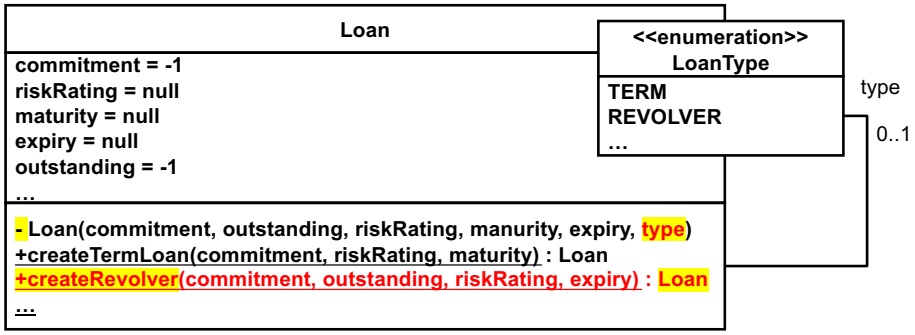
```
Loan l1 = new Loan(100, -1, 0.9, LocalDate.of(…), null, LoanType.TERM);
```

```
Loan l2 = new Loan(100, 0, 0.7, null, LocalDate.of(…), LoanType.REVOLVER);
```



- Different constructors to create different types of loans.
- They are less error-prone, but still hard to understand.

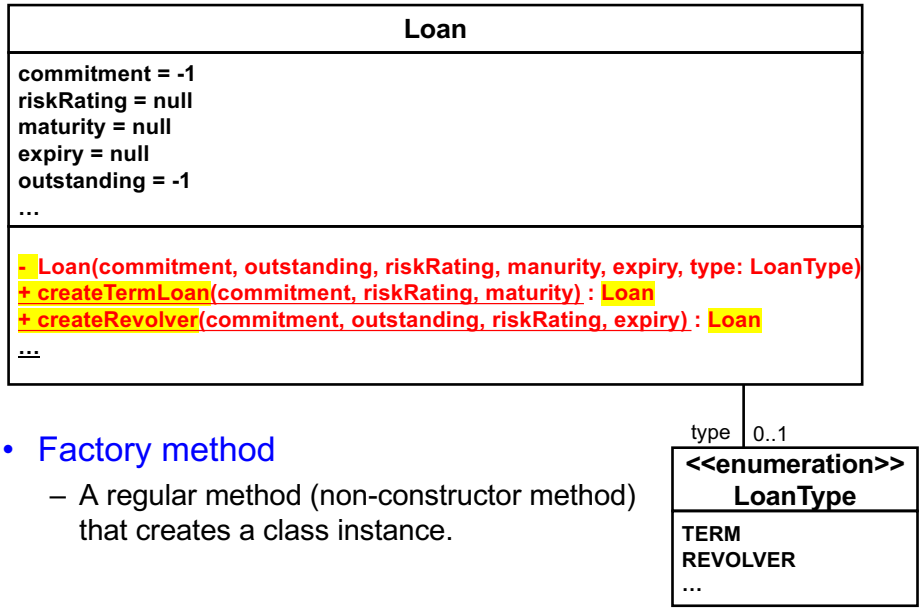
```
• Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);
• Loan l2 = new Loan(100, 0, 0.7, LocalDate.of(…), LoanType.REVOLVER);
```

- Client/user of Loan
  - `Loan loan = Loan.createRevolver(1000, 0, 0.7, LocalDate.of(...));`

```
• 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 );
    }
```


Static Factory Methods



- Factory method
  - A regular method (non-constructor method) that creates a class instance.

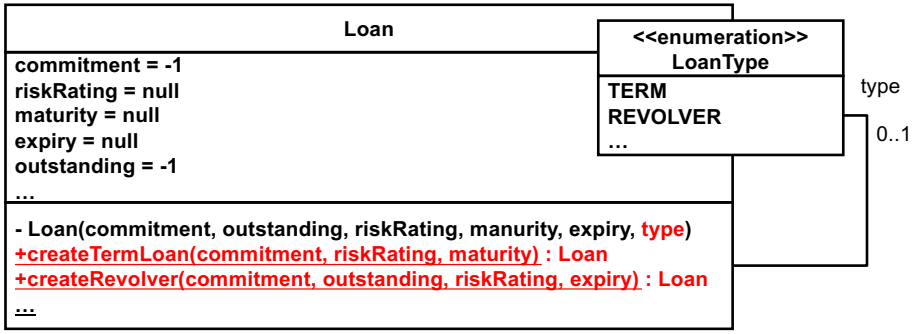
- You should NOT define public constructors.
  - You want all client code to use static factory methods.
- You should define a private constructor(s).
  - It can be empty or can do something for instance initialization.
  - If you never define constructors, your Java compiler automatically inserts an empty public constructor, so client code can instanciate the class.

Benefits of *Static Factory Method*

- Static factory methods have their own names.
  - Improve code maintainability.
    - The name can explicitly tell what object is created and what data is required to set it up.
    - A class to be instantiated and client code gets clean and easier to understand.

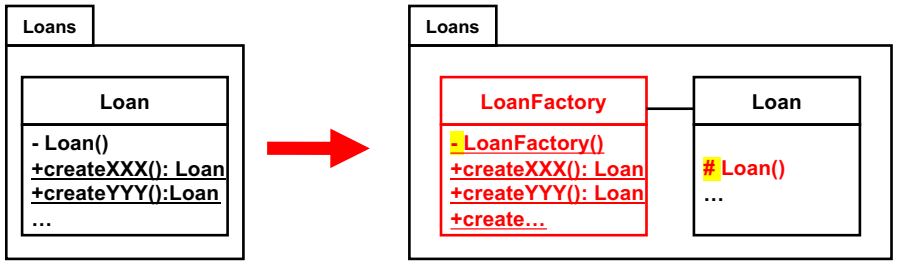
```
Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);
Loan l2 = Loan.createTermLoan(100, 0.9, LocalDate.of(…));
```


A Potential Issue w/ *Static Factory Method*

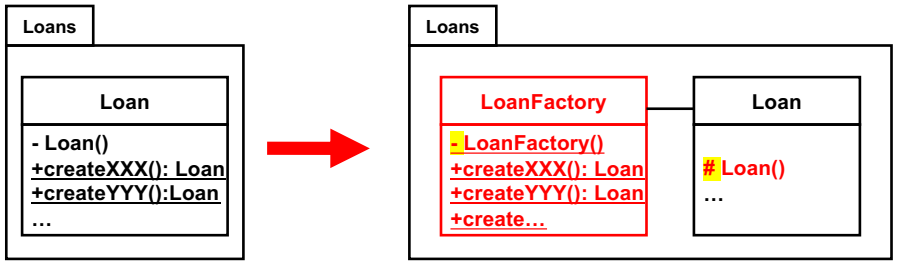


- Too many static factory methods in a class may obscure its primary responsibility/functionality.
  - They may dominate the class's public methods.
  - **`Loan`** may no longer strongly communicate it's primary functionality (e.g., loan-processing methods)

Design Revision w/ a Factory Class

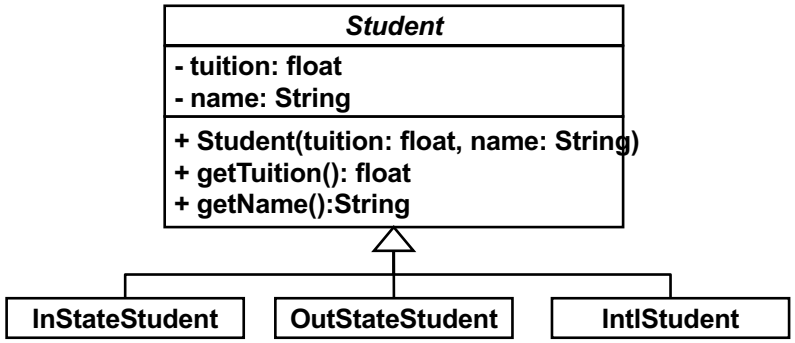


- Factory class (**`LoanFactory`**)
  - A class that consists of static factory methods and isolates instantiation logic (from **`Loan`**)
- **`Loan`** can better communicate its primary functionality (e.g., loan-processing methods)



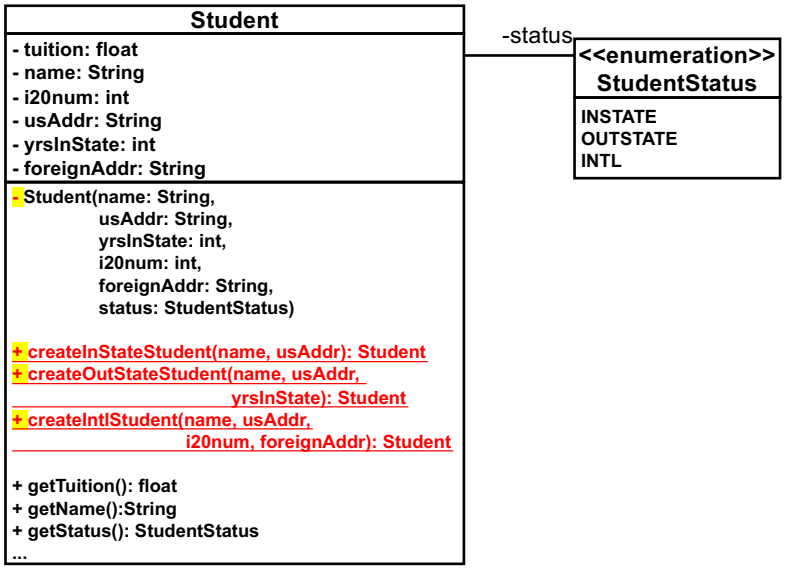
- Potential issue
  - Non-factory classes in the same package can call a protected constructor(s) in **`Loan`**
  - This could violate the encapsulation principle.
- Solutions
  - Define **`Loan`** as an inner class of **`LoanFactory`**
  - Define **`LoanFactory`** as an interface
    - Static factory methods to be implemented as static interface methods

Recap: This Design is not Good.



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

Design Improvement w/ *Static Factory Method*



Suggested Read

- Chapter 2 (Creating and destroying Objects) of *Effective Java*
  - Joshua Bloch, Addison
  - http://bit.ly/2ydblp8

Just in Case… Date and Time API in Java

- `java.util.Date` (since JDK 1.0)
  - Poorly designed: Never try to use this class
    - It still exists only for backward compatibility
- `java.util.Calendar` (since JDK 1.1)
  - Deprecated many methods of `java.util.Date`
  - Limited capability: Try not to use this class
- Date and Time API (`java.time`)
  - Since JDK 1.8
  - Always try to use this API.

Date and Time API: `Instant`

- Represents an instantaneous point on the timeline, which starts at 01/01/1970 (on the prime Greenwich meridian).
  - Can be used as a timestamp.

- **`Duration`**
  - Represents an amount of time in between two `InstantS`

```
Instant start = Instant.now();
...
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long timeElapsedMSec = timeElapsed.toMillis();
```

Date and Time API: "Local" Classes

- **`LocalDate, LocalTime, LocalDateTime`**
  - Used to represent date and time without a time zone (time difference)
  - Apply leap-year rules automatically.

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(2009, 9, 10);
LocalDate 18thBirthday = birthday.plusYears(18);
birthday.getDayOfWeek().getValue();
```

- **`Period`**
  - Represents an amount of time in between two local date/time.

```
Period period = today.until( 18thBirthday );
period.getDays();
```

Date and Time API: Other Classes

- **`TemporalAdjusters`**
  - Utility class (i.e., a set of static methods) that implements various calendaring operations.
    - e.g., Getting the first Sunday of the month.

- **`ZonedDateTime`**
  - Similar to `LocalDateTime`, but considers time zones (time difference) and time-zone rules such as daylight savings.

- **`DateTimeFormatter`**
  - Useful to parse and print date-time objects.

Singleton Design Pattern

Singleton

- Intent
  - Guarantee that a class has only one instance.

```
public class Singleton{
    private Singleton(){};
    private static Singleton instance = null;

    public static Singleton getInstance(){
        if(instance==null)
            instance = new Singleton ();
        return instance;
    }
}
```

- You should not define public constructors.
- Your should define a private constructor(s). Otherwise...


```
Singleton instance = Singleton.getInstance();
instance.hashCode();
Singleton instance = Singleton.getInstance();
instance.hashCode();
```

- `hashCode()` returns a unique ID (int) for each instance.
  - Different class instances have different IDs.
- `getInstance()` can take parameters, if necessary.
  - It can pass those parameters to a constructor.
- *Singleton* is an application of *Static Factory Method*.
  - `getInstance()` is a static factory method.
  - *Singleton* focuses on a requirement to have a class keep only one instance.

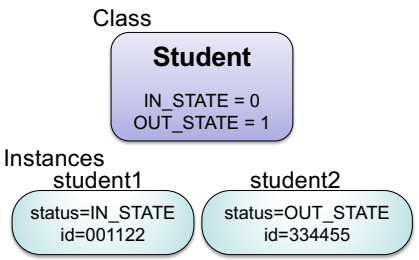
Just in Case…
Static Data Fields and Methods

- A static data field
  - Created used on a per-class basis.
    - All instances share the data field.
- A regular (non-static) data field
  - Created and used on an instance-by-instance basis.
    - Different instances have different copies of the data field.

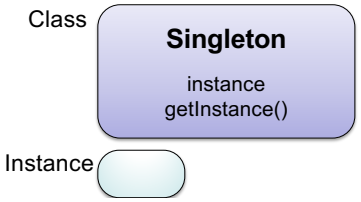
- A static data field
  - Created used on a per-class basis.
    - All instances share the data field.
- A regular (non-static) data field
  - Created and used on an instance-by-instance basis.
    - Different instances have different copies of the data field.


```
public class Student {
  private static    int IN_STATE = 0;
  private static    int OUT_STATE = 1;
  private                 int status;
  private                 int id;  }
```

```
Student student1 = new Student(...);
Student student2 = new Student(...);
```

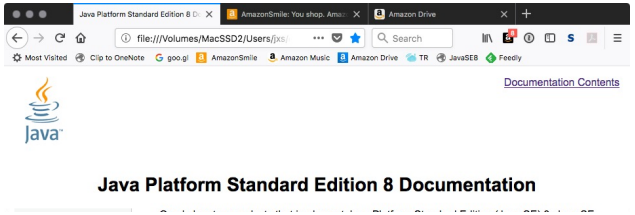


- A static method
  - Created used on a per-class basis.
  - Can access static data fields.
  - Can NOT access regular (non-static) data fields.
- A regular (non-static) method
  - Created and used on an instance-by-instance basis.
  - Can access both regular (non-static) and static data fields.



What Can be a *Singleton*?

- Object pools
  - Pool of a certain type of objects
    - e.g., database connections, network connections, threads, tabs open in a web browser, etc.
- Logger
- Step counter
- Configuration/plug-in manager
- Access counter
- Game loop
- Cache



Alternative Implementation with Null Checking API in Java 7

- `java.util.Objects`, extending `java.lang.Object`
  - A utility class (i.e., a set of static methods) to deal with the instances of `java.lang.Object` and its subclasses.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(str);
    } }
```

  - `requireNonNull()` throws a `NullPointerException` if `str==null`. Otherwise, it simply returns `str`.

```
class Foo{
    private String str;
    public Foo()(String str){
        this.str = Objects.requireNonNull(
                    str, "str must be non-null!!!");
    } }
```

  - `requireNonNull()` can receive an error message, which is to be contained in a `NullPointerException`.

- Traditional null checking

```
if(str == null)
    throw new NullPointerException();
this.str = str;
```

- With `Objects.requireNonNull()`

```
this.str = Objects.requireNonNull(str);
```

- Can eliminate an explicit conditional and make code a bit simpler.

Implementing *Singleton* with `Objects.requireNonNull()`

- ```
  public class SingletonNullCheckingJava7{
      private SingletonNullCheckingJava7(){};
      private static SingletonNullCheckingJava7 instance = null;

      public static SingletonNullCheckingJava7 getInstance(){
        try{
            return Objects.requireNonNull(instance);
        }
        catch(NullPointerException ex){
            instance = new SingletonNullCheckingJava7();
            return instance;
      } }
  ```

- ```
  public class Singleton{
      private Singleton(){};
      private static Singleton instance = null;

      public static Singleton getInstance(){
              if(instance==null)
                      instance = new Singleton ();
              return instance;
      } }
  ```