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

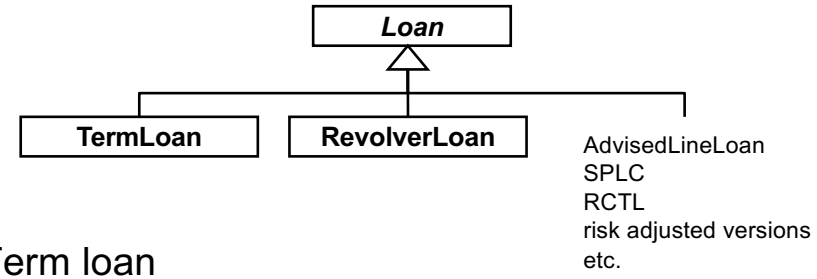
- Understand how to address reusability, maintainability and extensibility
  - By improving the *design* and *organization* of programs through design patterns

Static Factory Method

Static Factory Method

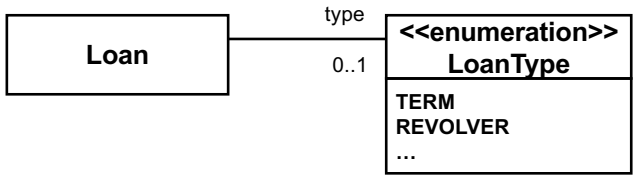
- Intent
  - Define a "communicable" method that instantiates a class
    - Constructors are the methods to instantiate a class.
    - Static factory methods are more "communicable" (more maintainable) than constructors.
- Benefits
  - Static factory methods have their own names (!).
    - Improve code 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.

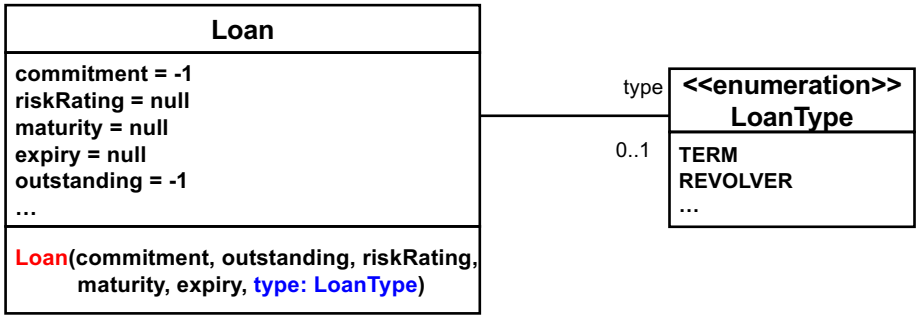


- Term loan
  - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
  - With a spending limit and expiry date
- Dynamic class change problem
  - A revolver can transform into a term loan when the revolver expires.

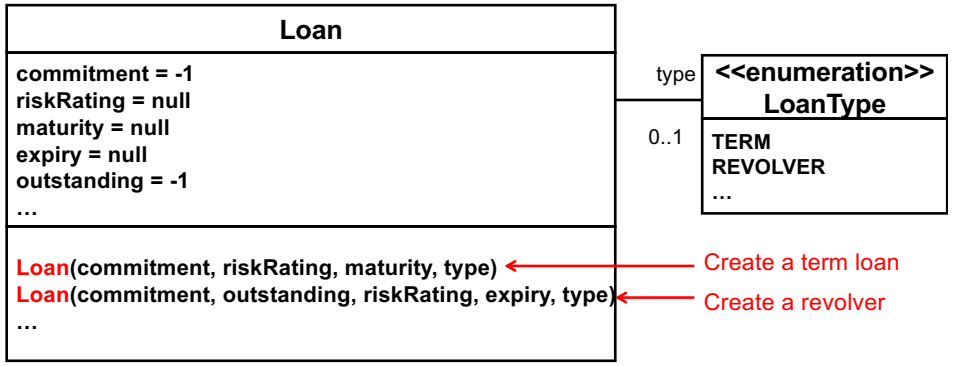
Enumeration-based Design



- A class inheritance should not be used here.

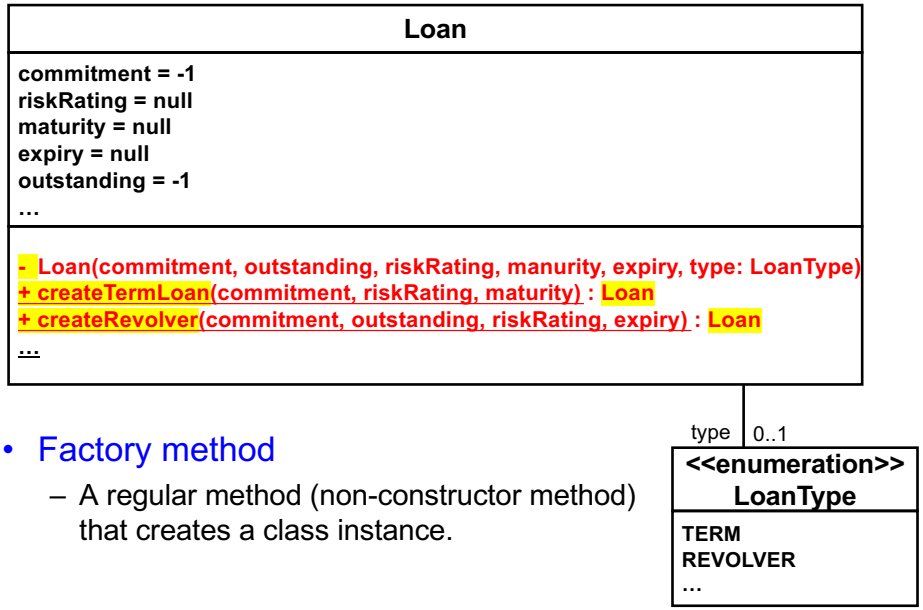


- Different loans need different sets of data to be set up.
  - A term loan needs commitment, risk rating and maturity date.
  - A revolver needs commitment, outstanding debt, risk rating and expiry date.
- The constructor is hard to understand and error-prone.
- `Loan l1 = new Loan(100, -1, 0.9, LocalDate.of(…), null, LoanType.TERM);`
- `Loan l2 = new Loan(100, 0, 0.7, null, LocalDate.of(…), LoanType.REVOLVER);`

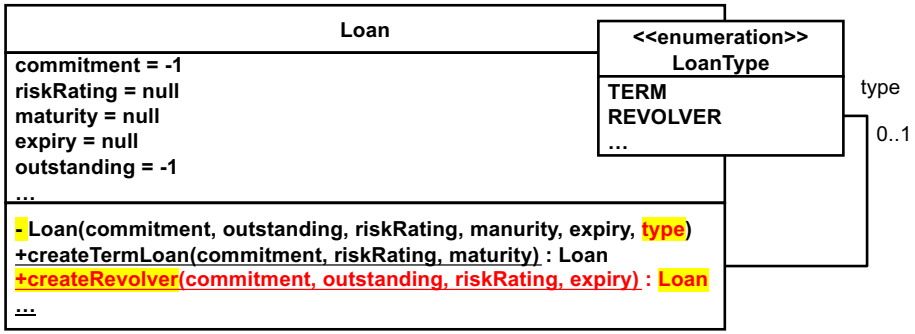


- Different constructors to create different types of loans.
- They are less error-prone, but still hard to understand.
- `Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);`
- `Loan l2 = new Loan(100, 0, 0.7, LocalDate.of(…), LoanType.REVOLVER);`

Static Factory Methods



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



- 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 );
    }
```

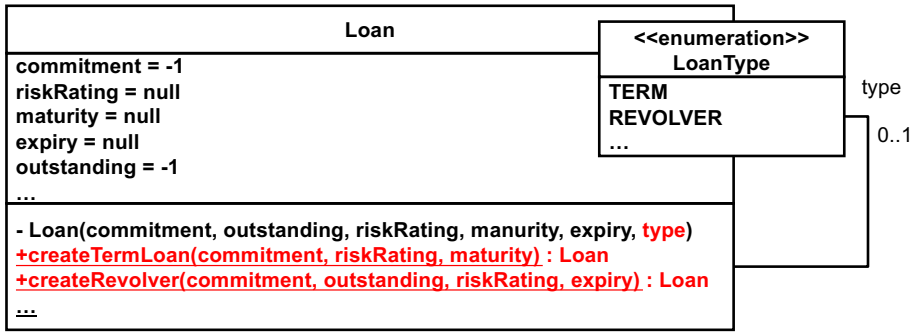

- You should NOT define public constructors.
  - You want all client code to use static factory methods.
- You should define a private constructor(s).
  - It can be empty or can do something for instance initialization.
  - If you never define constructors, your Java compiler automatically inserts an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

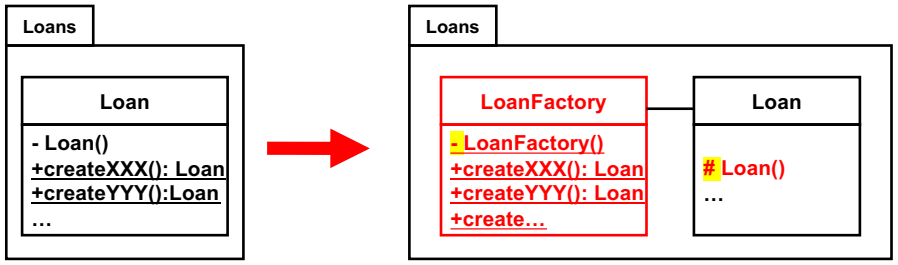
```
– Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);
– Loan l2 = Loan.createTermLoan(100, 0.9, LocalDate.of(…));
```

A Potential Issue w/ *Static Factory Method*

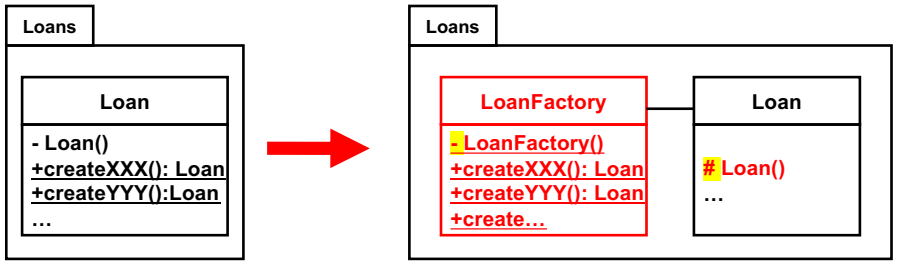


- Too many static factory methods in a class may obscure its primary responsibility/functionality.
  - They may dominate the class's public methods.
  - **`Loan`** may no longer strongly communicate it's primary functionality (e.g., loan-processing methods)

Design Revision w/ a Factory Class

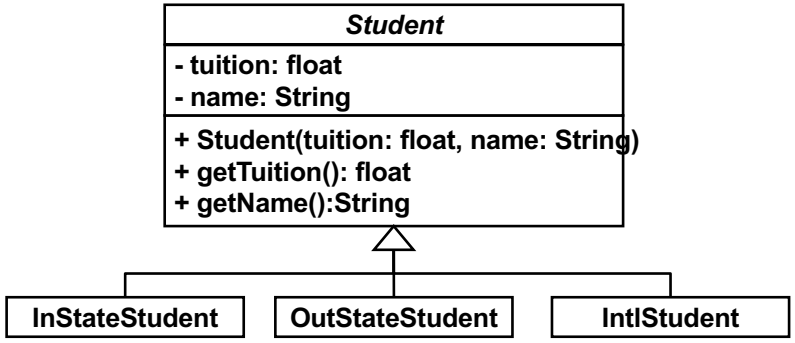


- Factory class (**`LoanFactory`**)
  - A class that consists of static factory methods and isolates instantiation logic (from **`Loan`**)
- **`Loan`** can better communicate its primary functionality (e.g., loan-processing methods)



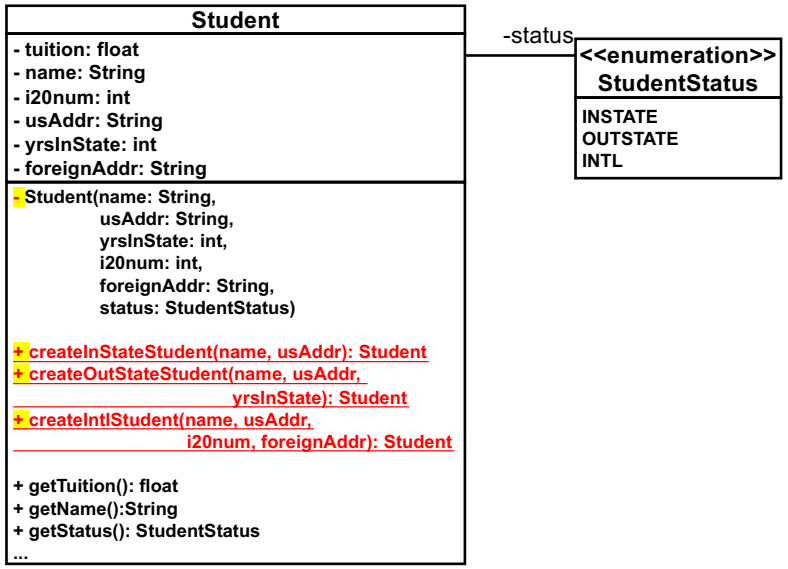
- Potential issue
  - Non-factory classes in the same package can call a protected constructor(s) in **`Loan`**
  - This could violate the encapsulation principle.
- Solutions
  - Define **`Loan`** as an inner class of **`LoanFactory`**
  - Define **`LoanFactory`** as an interface
    - Static factory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

Singleton Design Pattern

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

- 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.

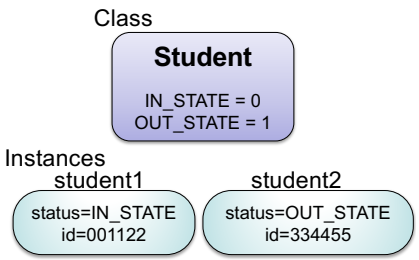
Just in Case…
Static Data Fields and Methods

- 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.

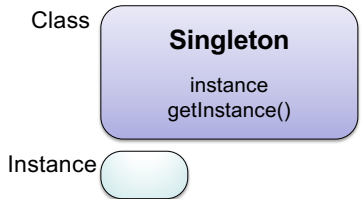
- A static data field
  - Created used on a per-class basis.
    - All instances share the data field.
- A regular (non-static) data field
  - Created and used on an instance-by-instance basis.
    - Different instances have different copies of the data field.


```
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(...);
```

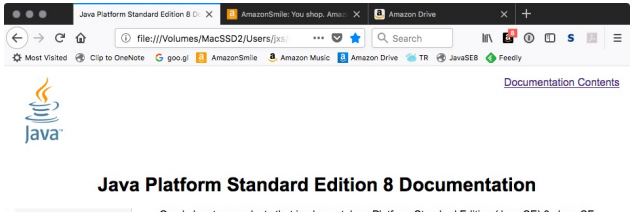


- A static method
  - Created used on a per-class basis.
  - Can access static data fields.
  - Can NOT access regular (non-static) data fields.
- A regular (non-static) method
  - Created and used on an instance-by-instance basis.
  - Can access both regular (non-static) and static data fields.



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



- 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.

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`.

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;
    } }
```

JUnit API (cont'd)

Equality and Identity

- `assertEquals( Object expected, Object actual )`
  - Asserts that `actual` is *logically equal* to `expected`
    - » By calling `expected.equals(actual)`.
      - » c.f. `Object.equals()`

- `assertSame( Object expected, Object actual )`
  - Asserts that `expected` and `actual` refer to the *identical object*
    - » by checking if `expected.hashCode()==actual.hashCode()`

    - »
```
Foo f = new Foo();
assertSame(f, f);                    // PASS
```

    - »
```
Singleton instance1 = Singleton.getInstance();
Singleton instance2 = Singleton.getInstance();
assertSame(instance1, instance2);  // PASS
```


-
```
String str = "umb";          // Syntactic sugar for
                             //     String str = new String("umb");
                             // str contains a pointer (or reference) to
                             // the String instance.
String expected = str;       // expected and actual refer to the
String actual = str;         // identical String instance.
```

-
```
assertSame(expected, actual);      // PASS
assertEquals(expected, actual);    // PASS
```

- `assertSame()` checks whether
  - `expected.hashCode()==actual.hashCode()` is true.

- `assertEquals()` checks whether
  - `expected.equals(actual)` returns true.
    - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String value.

-
```
String expected = "umb";     // Syntax sugar for:
                             //    String expected = new String("umb");
String actual = "umb0".substring(0,2);
                             // Syntax sugar for:
                             //    String temp = new String("umb0");
                             //    actual = temp.substring(0, 2);
                             // "umb0" -> "umb"
                             // expected and actual refer to
                             // different String instances.
```

-
```
assertSame(expected, actual);    // FAIL
assertEquals(expected, actual);  // PASS
```

- `assertSame()` checks whether
  - `expected.hashCode()==actual.hashCode()` is true.

- `assertEquals()` checks whether
  - `expected.equals(actual)` returns true.
    - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String values.

HW 2

- Revise `PrimeGenerator` to be a *singleton* class.
  - Name the revised class as `SingletonPrimeGenerator`.
  - Add a static factory method: `getInstance()`.
  - Remove the public constructor.
  - Have your class receive `from` and `to` values, as you like.

- Write a test class (`SingletonPrimeGeneratorTest`) with JUnit
  - Verify `getInstance()` returns a non-null value.
    - Use `Assertions.assertNonNull()`
  - Verify `getInstance()` returns the identical instance when it is called multiple times.
    - Use `assertSame()`
  - Verify `getPrimes()` returns the expected result.
    - Use `assertIterableEquals()` or `assertArrayEquals()`
  - Verify your class throws an expected exception when wrong ranges (`from`-`to` pairs) are given
    - e.g., [-10, 10], [-10, -5], [100, 1]

Recap: Equality

- `assertEquals( Object expected, Object actual )`
  - Asserts that `actual` is *logically equal* to `expected`
    - » By calling `expected.equals(actual)`.
      - » c.f. `Object.equals()`
  - Checks whether
    - `expected.equals(actual)` returns true.
      - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String values.

`Object.equals()`

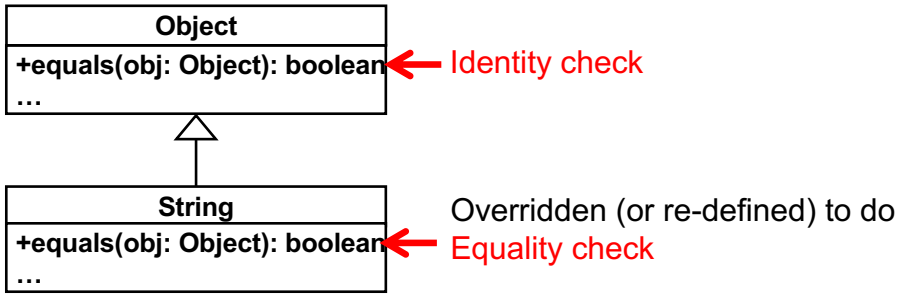
- `Object.equals(Object obj)` compares two objects with:
  - `if( this.toString()==obj.toString() ){ return true; } else if{ return false; }`
  - `Object.toString()` returns the identity of an object.
    - String data that consists of an object ID, a class name and a package name.
      - e.g., edu.umb.cs680.junit5intro.Calculator@2b2948e2
  - Performs identity check (not equality check).
    - Even though the method name says “equals.”

`Object.equals()`

- `Object.equals(Object obj)` compares two objects with:
  - `if( this.toString()==obj.toString() ){ return true; }`
    `else if{ return false; }`
  - `Object.toString()` returns the identity of an object.
    - String data that consists of an object ID, a class name and a package name.
      - e.g., edu.umb.cs680.junit5intro.Calculator@2b2948e2
    - Performs identity check (not equality check).
      - Even though the method name says "equals."
- Most Java API classes (e.g. String) override `Object.equals()` to perform appropriate equality check.
  - However, user-defined classes DO NOT… to be discussed.

`equals()` in Java API

- Most Java API classes override `Object.equals()` to perform appropriate equality check.
  - e.g., `String` overrides `Object.equals()` and returns true if two String instances contain the same String values.



Read the source code of `String.equals()` if you are interested.

Equality Check for User-defined Classes

- When you define your own class, it is implicitly treated as a subclass of `Object`
  - Your class inherits `Object.equals()`.
- Your class's `equals()` does identity check by default
  - Unless you override `equals()`.


```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
assertSame(p1, p1);          // PASS
assertSame(p1, p2);          // FAIL
assertEquals(p1, p2);        // FAIL
assertEquals(p1, p3);        // FAIL
assertEquals(p2, p3);        // FAIL
```

- `Person` inherits `Object.equals()`. The inherited method performs *identity check* by default for `Person` instances.
  - You need to override `equals()` in `Person` if you want equality check.

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
assertSame(p1, p2);                 // FAIL
assertEquals(p1, p2);               // PASS
assertSame(p1, p3);                 // FAIL
assertEquals(p1, p3);               // FAIL
```

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String + equals(anotherPerson:Object): boolean

```
if( this.firstName.equals(((Person)anotherPerson).getFirstName())
    && this.lastName.equals(((Person)anotherPerson).getLastName())){
  return true;
}
else{
  return false;
}
```


- Define `equals()` in `Person`, if your team has a consensus about the equality of `Person`s.
- If the consensus may often change, or if there is no reasonable consensus…
  - you should craft equality-check logic in your test class, not in `Person`.

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
assertEquals(p1.getFirstName(), p2.getFirstName());     // PASS
assertEquals(p1.getLastName(), p2.getLastName());       // PASS

assertNotEquals(p1.getFirstName(), p3.getFirstName()); // PASS
assertNotEquals(p1.getLastName(), p3.getLastName());   // PASS
```

- JUnit judges that a test method (test case) passes if it normally returns (i.e., if all four assertion methods return) without `AssertionFailedError`

How to Write Equality-check Logic

- As you use more information for an equality check, you need to call assertion methods more often in a single test method.
  - e.g., first and last names, DOB, zip code for home address.
    - Need to call `assertEquals()` 4 times.
  - e.g., car name, manufacturer name, production year
    - Need to call `assertEquals()` 3 times.
- Equality-check logic gets less clear.
- In general, it makes more sense to perform equality-check by calling assertion methods less often.
  - Consider a String-to-String or array-to-array comparison.

String-to-String Comparison

```
@Test
... checkPersonEqualityWithJohnJane(){
  Person p1 = new Person("John","Doe",
                         LocalDate...,
                         02125);
  Person p2 = new Person("Jane", "Doe",
                         LocalDate...,
                         02125);

  assertEquals(p1.getFirstName(),
               p2.getFirstName());
  assertEquals(p1.getLastName(),
               p2.getLastName());
  assertEquals(p1.getDOB(),
               p2.getDOB());
  assertEquals(p1.getZipCode(),
               p2.getZipCode());
}
```

```
private String eol =
    System.getProperty("line.separator");

private String personToString(Person p){
  return p.getFirstName() + eol +
         p.getLastName() + eol +
         p.getDOB().toString() + eol +
         p.getZipCode() + eol; }

private String concatenamePersonInfo(
                                String[] p){
  String personInfo;
  for(String info: p){
      personInfo += info + eol;} }

@Test
... checkPersonEqualityWithJohnJane(){
  String[] expectedArray =
      {"John", "Doe", ..., "02125"};
  String expected =
      concatenatePersonInfo(expectedArray);

  Person actual = new Person(
                "John","Doe", ..., 02125);

  assertEquals(expected,
               personToString(actual) ); }
```

Array-to-Array Comparison

```
private String[] personToStringArray( Person p ){
  String[] personInfo = {p.getFirstName(),
                         p.getLastName(),
                         p.getDOB().toString(),
                         p.getZipCode() };
  return personInfo;
}

@Test
public void checkPersonEqualityWithJohnJane(){
  String[] expected = {"John", "Doe", ..., "0215"};

  Person actual = new Person("John","Doe", ..., 02125);

  assertArrayEquals(expected,
                    personToStringArray(actual) );
}
```


HW 3

- Define the `Car` class and implement its getter methods.
  - ```
    public class Car {
      private String make, model;
      private int mileage, year;
      private float price;  }
    ```

- Write a test class (`carTest`) with JUnit
  - Include a private method `carToStringArray()`
  - Define a test method `verifyCarEqualityWithMakeModelYear()`
    - Create two `Car` instances and check their equality with array-to-array comparison
      - Use make, model and year in equality-check logic

      - ```
        String[] expected = {"Toyota", "RAV4", "2018"};
        Car actual = new Car(...);
        assertArrayEquals(expected,
                          carToStringArray(actual) );
        ```