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.

Resources

- *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Eric Gamma et al., Addison-Wesley
- *Head First Design Patterns*
  - Elizabeth Freeman et al., O'Reilly
- *Game Programming Patterns*
  - Robert Nystrom, Genever Benning
  - http://gameprogrammingpatterns.com/
- Web
  - http://en.wikipedia.org/wiki/Design_patterns_(computer_science)
  - http://sourcemaking.com/design_patterns

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 give you great examples.

A Key Topic in CS680

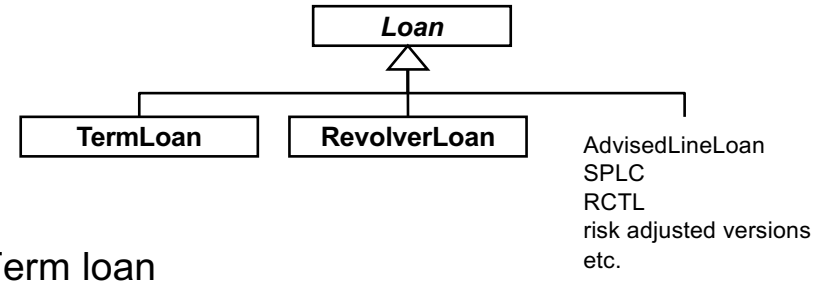
- Understand how to address reusability, maintainability and extensibility
  - By improving the *design* and *organization* of programs
  - By learning about major design patterns

Static Factory Method

Static Factory Method

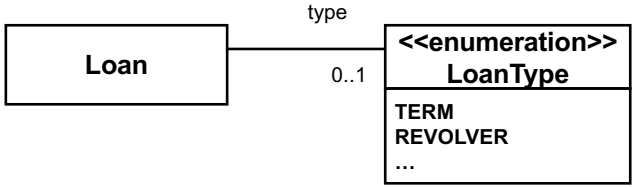
- Intent
  - Define a "communicable" method that instantiates a class
    - Constructors are the methods to instantiate a class.
    - Static factory methods are more "communicable" (more maintainable) than constructors.
- Benefits
  - Static factory methods have names (!).
    - Improve code maintainability.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

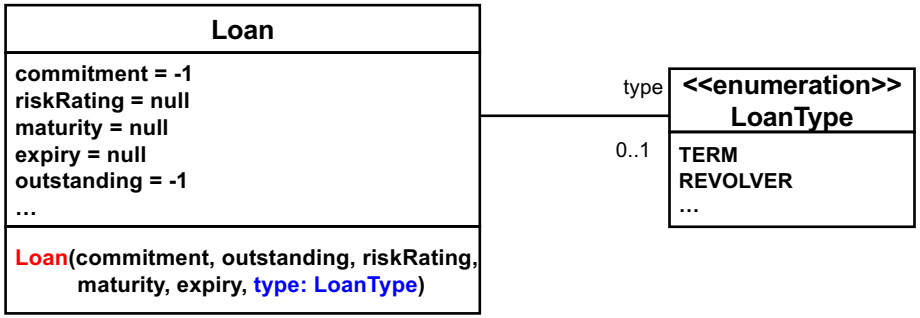


- Term loan
  - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
  - With a spending limit and expiry date
- Dynamic class change problem
  - A revolver can transform into a term loan when the revolver expires.

Enumeration-based Design



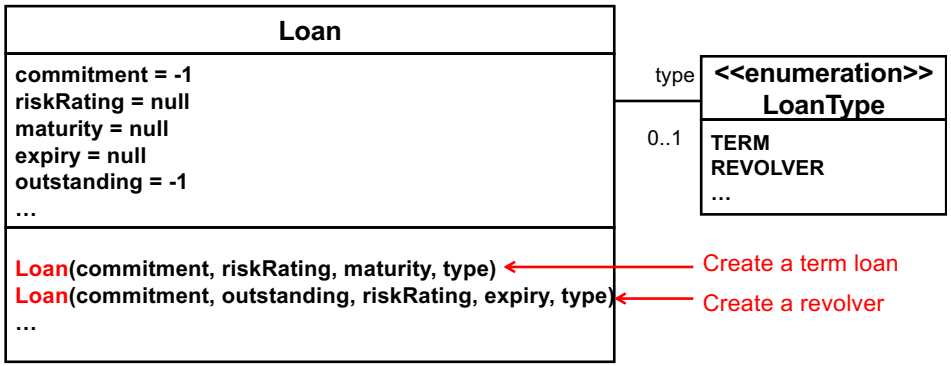
- A class inheritance should not be used here.



- 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.

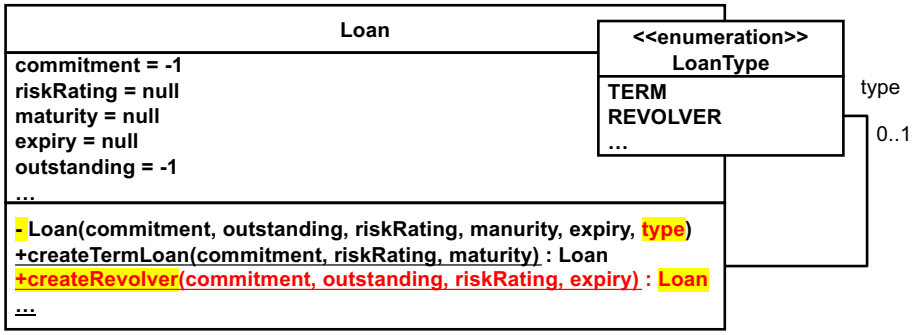
```
Loan l1 = new Loan(100, -1, 0.9, LocalDate.of(…), null, LoanType.TERM);
```

```
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.

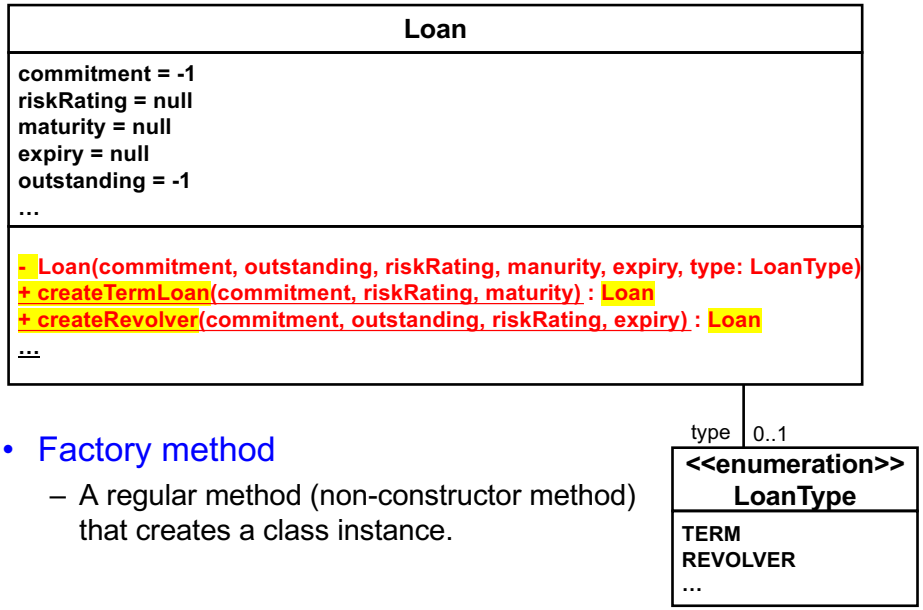
```
• Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);
• Loan l2 = new Loan(100, 0, 0.7, LocalDate.of(…), LoanType.REVOLVER);
```

- 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.

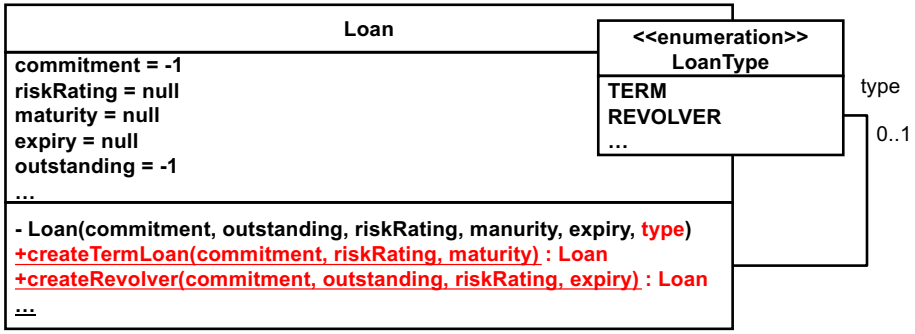
- You should not define public constructors.
  - You want all client code to use static factory methods.
- You should define a private constructor(s).
  - It can be empty or can do something for instance initialization.
  - If you never define constructors, your Java compiler automatically inserts a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

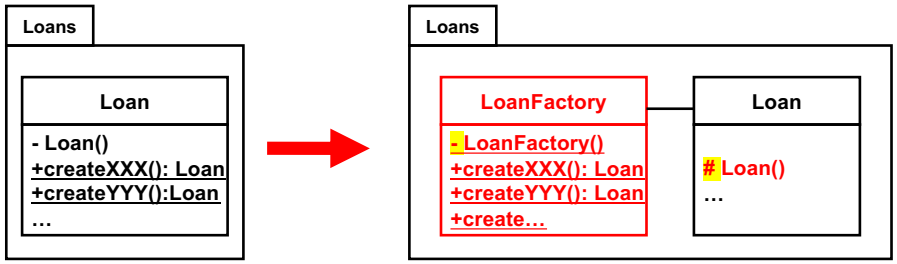
```
Loan l1 = new Loan(100, 0.9, LocalDate.of(…), LoanType.TERM);
Loan l2 = Loan.createTermLoan(100, 0.9, LocalDate.of(…));
```


A Potential Issue w/ *Static Factory Method*

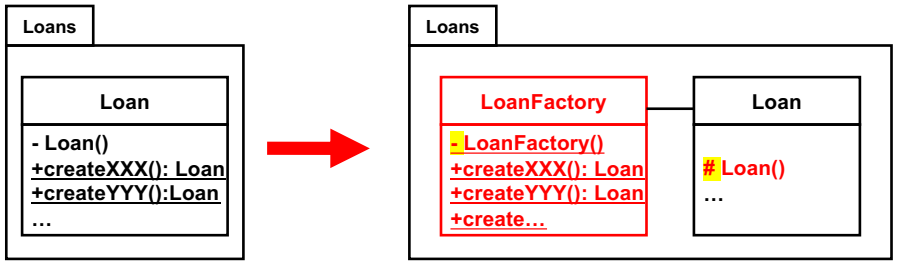


- Too many static factory methods in a class may obscure its primary responsibility/functionality.
  - They may dominate the class's public methods.
  - **`Loan`** may no longer strongly communicate it's primary functionality (e.g., loan-processing methods)

Alternative: Factory Class

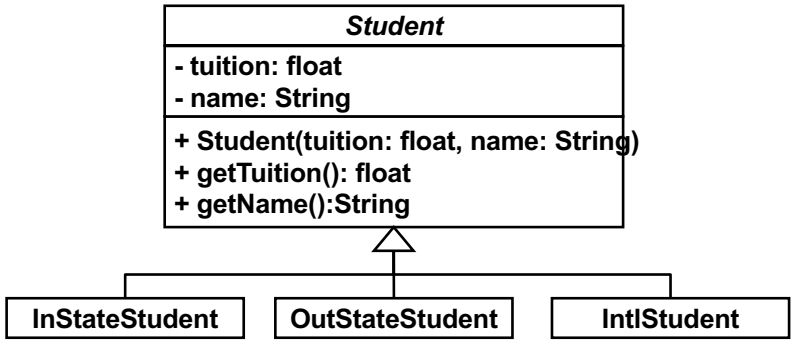


- Factory class (**`LoanFactory`**)
  - A class that consists of static factory methods and isolates instantiation logic (from **`Loan`**)
- **`Loan`** can better communicate its primary functionality (e.g., loan-processing methods)



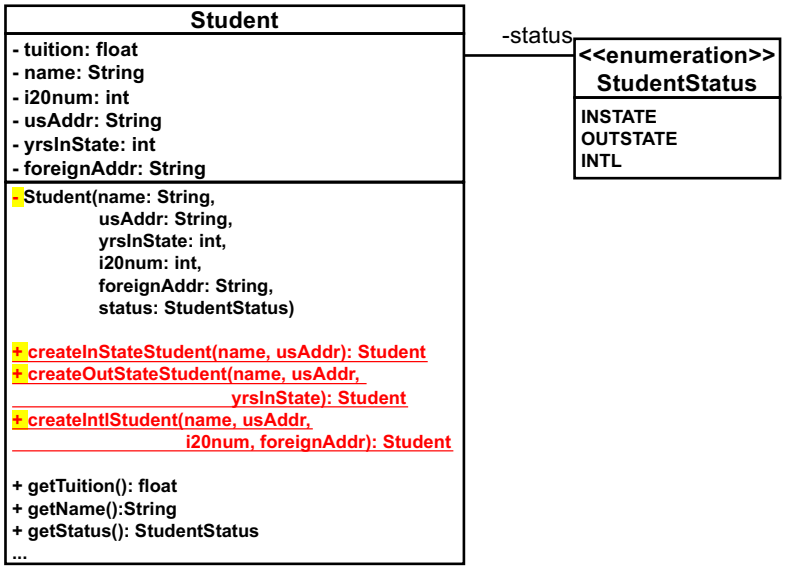
- Potential issue
  - Non-factory classes in the same package can call a protected constructor(s) in **`Loan`**
  - This could violate the encapsulation principle.
- Solution
  - Define **`Loan`** as an inner class of in **`LoanFactory`**

Recap: This Design is not Good.



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

Design Improvement w/ *Static Factory Method*



HW 3

- Complete the `Student` class and test its (3) static factory methods.
  - Write test cases with JUnit.
  - Optional: Implement a factory class
    - Separate `StudentFactory` and `Student`
- Deadline: October 22 (Tue) midnight

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 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;
    }
}
```

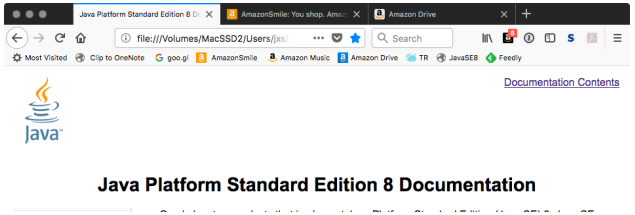
- You should not define public constructors.
- Your should define a private constructor(s). Otherwise...


```
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 an 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.

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
- 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;
    } }
```


JUnit API (cont'd)

Assertions.assertEquals()

- `org.junit.jupiter.api.Assertions`
  - Contains a series of *static* assertion methods.
  - `assertEquals( int expected, int actual )`
  - `assertEquals( float expected, float actual )`
    `...`
  - `assertEquals( Object expected, Object actual )`
    - » Defined for each primitive type and `Object`
    - » Returns if two values (expected and actual values) match.
      - »
```
float expected = 12;
float actual = cut.multiply(3,4);
assertEquals( expected, actual );
```
    - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
      - » JUnit catches it; your test cases don't have to.
    - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

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


-
```
String str = "umb";          // Syntax sugar for
                             //     String str = new String("umb");
                             // str contains a pointer 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)` is true.
    - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String values.

-
```
String expected = "umb";     // Syntax sugar for:
                             //    String expected = new String("umb");
String actual = "umb0".substring(0,2);
                             // Syntax sugar for:
                             //    String actual = new String("umb0");
                             //    actual.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)` is 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 String data that consists of an object ID, a class name and a package name.
    - e.g., edu.umb. cs680.calc.Calculator@2b2948e2
    - Returns the identity of an object.
  - Performs identity check.

- Most Java API classes (e.g. String) override `Object.equals()` to perform appropriate equality check.
  - However, user-defined classes DO NOT… to be discussed.
    ```