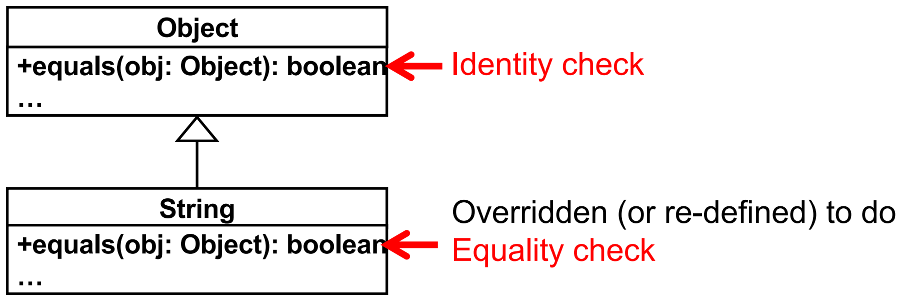
`Object.equals()`

- `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.
    - Even though the method name says "equals."

`equals()` in Java API

- Most Java API classes override (or re-define) `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.


Object
+equals(obj: Object): boolean
…
Identity check
String
+equals(obj: Object): boolean
…
Overridden (or re-defined) to do
Equality check

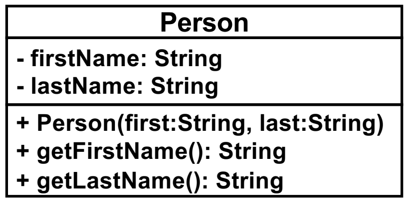
Read the source code of `String.equals()` if you are interested.

Equality Check for User-defined Classes

- When you define your own class, it inherit `Object.equals()`.
- Your class's `equals()` does identity check by default
  - Unless you override (or re-define) `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 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 `Persons`.
- If the consensus may often change, or if there is no 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 4

- Recall the *Singleton* design pattern.
- Test the `Singleton` class to make sure that its `getInstance()` returns the identical instance.
  - Write a test class (`SingletonTest`) with JUnit.
  - Use `assertSame()` in a test method
- Deadline: October 24 (Thu) midnight

HW 5

- 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 `assertArrayEquals()`
      - Use make, model and year in equality-check logic

      -
        ```
        String[] expected = {"Toyota", "RAV4", "2018"};
        Person actual = new Car(...);
        assertArrayEquals(expected,
                          carToStringArray(actual) );
        ```
- Deadline: October 24 (Thu) midnight

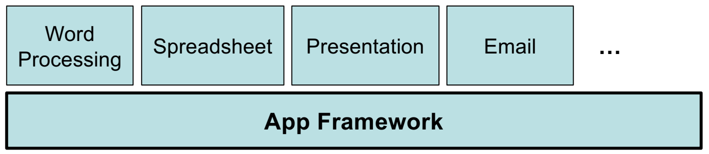
Factory Method

Factory Method

- A method to instantiate a class and initialize a class instance without using its constructors
  - Uses a regular method (i.e., non-constructor method)
  - Allows a class to *defer* instantiation to its subclasses.
    - Define an abstract class for creating an instance.
    - Allows its subclasses to decide *which class to instantiate* and *how to initialize a class instance*.

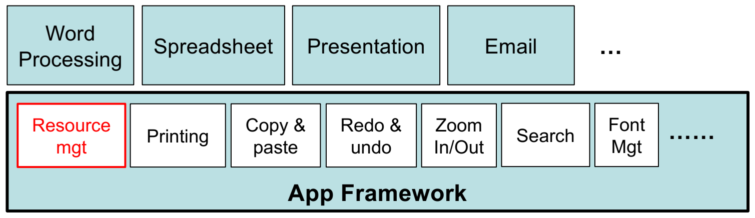
An Example: A Framework for Productivity ("Office") Applications

- Application framework
  - A set of foundation APIs to implement and run a series of apps.
    - Implement the standard/common functionalities (structures and behaviors) in individual applications
    - Make them available/reusable for individual apps.
    - Make app development easier and faster.
- Frameworks for productivity ("office") applications
  - e.g., .Net Framework, Microsoft Foundation Class (MFC), Cocoa, OpenOffice Framework, GNOME, KDE, etc.

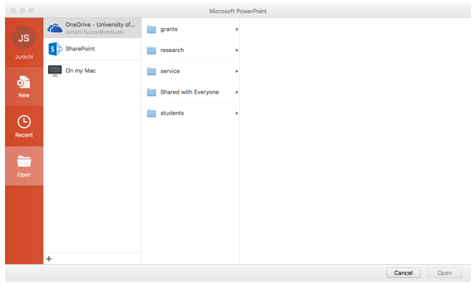


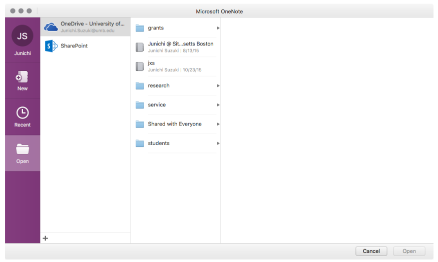
Resource Mgt in App Framework

- Resource management
  - Creating, opening and closing *resources* used in applications
    - e.g., documents, spreadsheets, presentation slides, emails and notes.
  - Saving resources in the local disk or a remote cloud.
  - Renaming resources.
  - Exporting resources in other resource types (file formats).
- Here, we focus on the ***creation*** of resources.



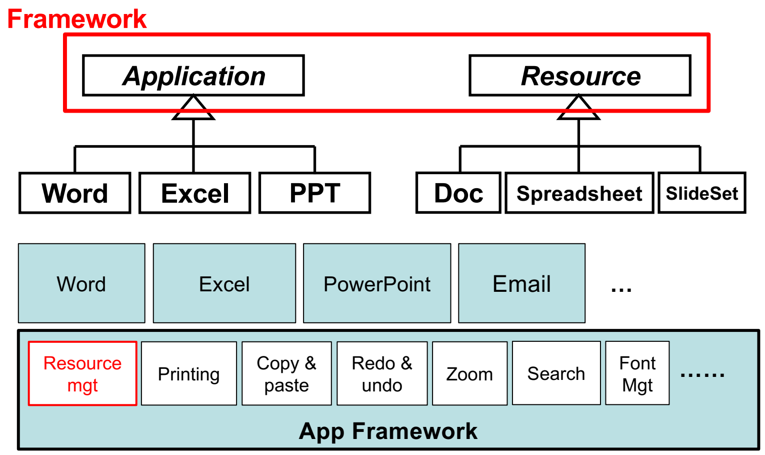
In Microsoft Office Applications…



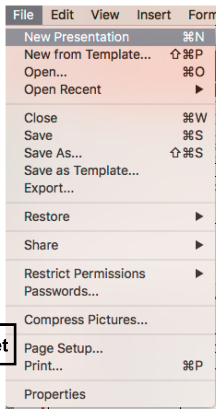


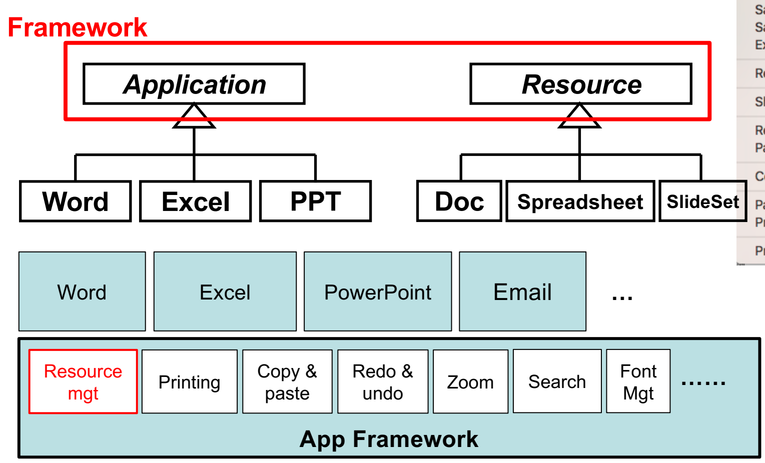
Requirements for Resource Creation

- Multiple applications run on the framework.
- Different applications create and use different types of resources.

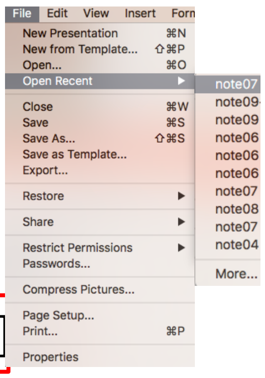


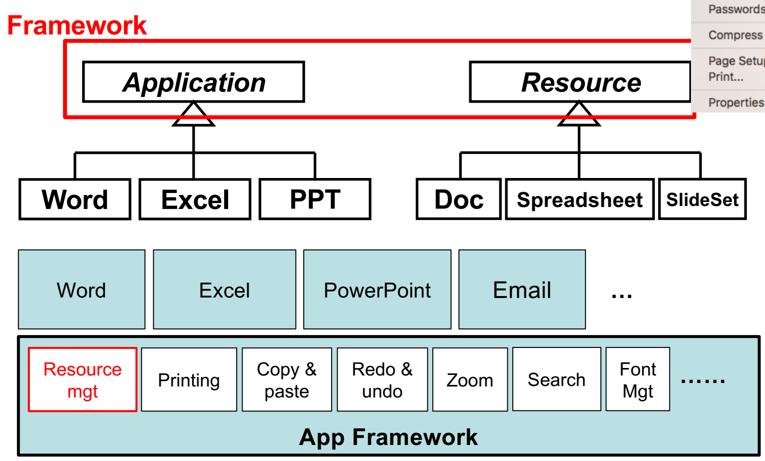
- When an application creates a new resource, it opens a blank resource.
  - Word creates a blank document.
  - Excel creates a blank spreadsheet.
  - PPT creates a blank slide set.

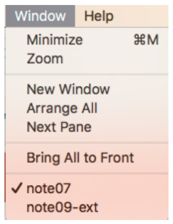




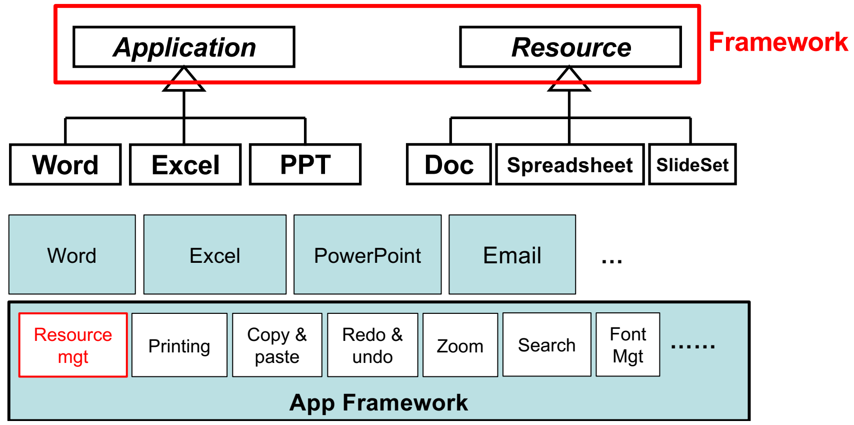
- Each application creates one resource at a time, but can keep multiple resources open.
- Each application records the list of resources that it opened recently.



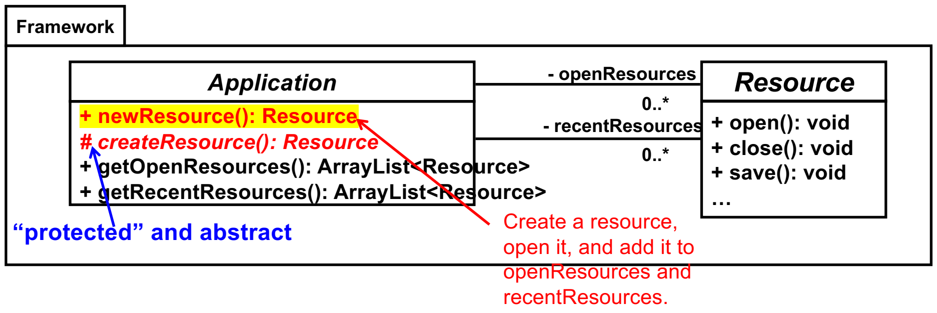


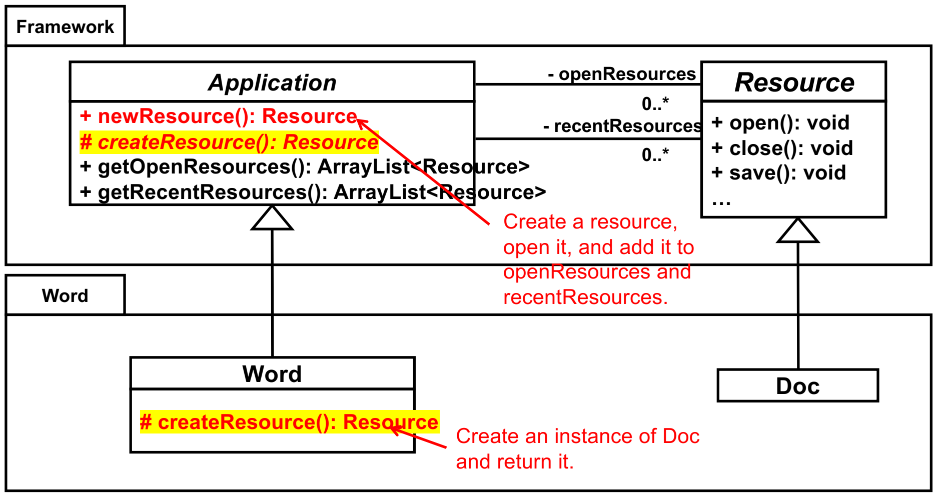


- Extra applications may be developed in the near future.
  - An app to be developed in the future should create a particular resource associated to that app.
    - We don't know that app-resource pair in advance.
- How can we implement the *common resource creation logic* at the framework level (i.e., with Application and Resource) without knowing Application's and Resource's subclasses?

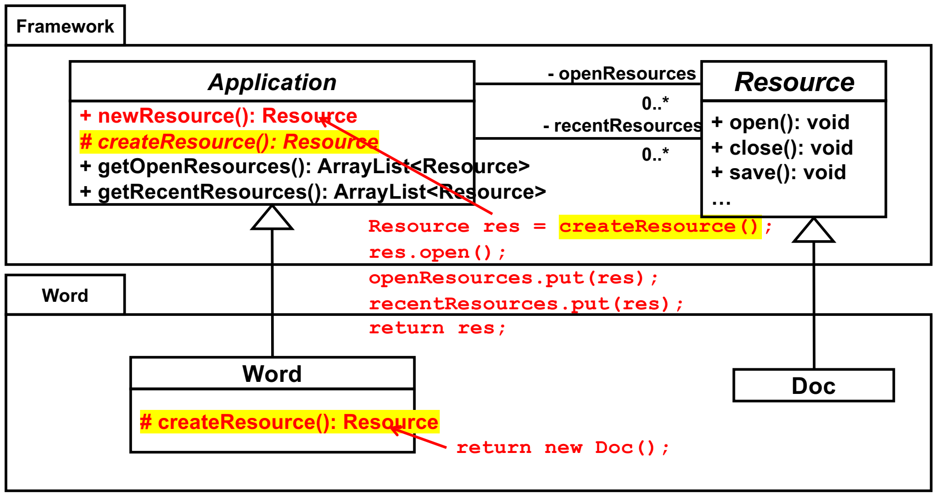


Address this Design Context with *Factory Method*


Framework
Application
+ newResource(): Resource
# createResource(): Resource
+ getOpenResources(): ArrayList<Resource>
+ getRecentResources(): ArrayList<Resource>
- openResources
0..*
- recentResources
0..*
Resource
+ open(): void
+ close(): void
+ save(): void
…
"protected" and abstract
Create a resource, open it, and add it to openResources and recentResources.


Framework
Application
+ newResource(): Resource
# createResource(): Resource
+ getOpenResources(): ArrayList<Resource>
+ getRecentResources(): ArrayList<Resource>
- openResources
0..*
- recentResources
0..*
Resource
+ open(): void
+ close(): void
+ save(): void
…
Create a resource, open it, and add it to openResources and recentResources.
Word
Word
# createResource(): Resource
Doc
Create an instance of Doc and return it.

```
Word word = new Word(...);
word.newResource();
```


Framework
Application
+ newResource(): Resource
# createResource(): Resource
+ getOpenResources(): ArrayList<Resource>
+ getRecentResources(): ArrayList<Resource>
- openResources
0..*
- recentResources
0..*
Resource
+ open(): void
+ close(): void
+ save(): void
…
Word
Word
# createResource(): Resource
Doc

```
Resource res = createResource();
res.open();
openResources.put(res);
recentResources.put(res);
return res;
```

```
return new Doc();
```

```
Word word = new Word(…);
word.newResource();
```

What's the Point?

- The framework
  - `newResource()` provides a *skeleton* (or *template*) for resource creation.
    - *Partially* implements a common procedure for resource creation.
    - Never states specific types (class names) for apps and their resources, such as `Word` and `Doc`.
- Word (framework client)
  - Reuses the skeleton/template for resource creation and *completes* it
    - By specifying which application class and which resource class are used.

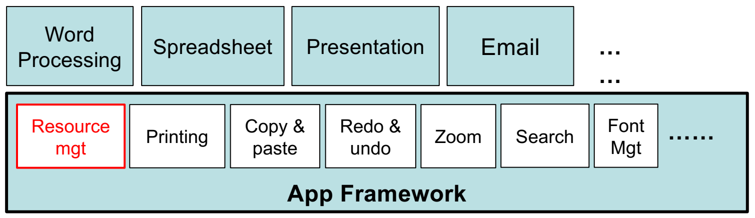
What *Factory Method* Does…

- Defines a *factory method* (`newResource()`) in `Application`.
- Has it implement a common procedure for resource creation
  - with an *empty protected method* (`createResource()`).
- Allows `Application` to *defer* instantiation to its subclasses (e.g., `Word`)
  - Allows each subclass (e.g., `Word`) to decide *which class to instantiate* and *how to instantiate it*.

Benefits

- The framework
  - Can define a common procedure for resource creation.
    - Without knowing app-resource pairs (i.e., which specific apps use which specific resources).
  - Allows individual apps to reuse the common procedure.
    - Less redundant code in apps.
    - Can "force" every single app to follow the same behavior (i.e. the same procedure for instance creation and initialization) when it creates a new resource.

- Can be independent (or de-coupled) from individual applications (framework clients).
  - Allows applications to be pluggable to the framework.


Word Processing
Spreadsheet
Presentation
Email
……
Resource mgt
Printing
Copy & paste
Redo & undo
Zoom
Search
Font Mgt
……
App Framework