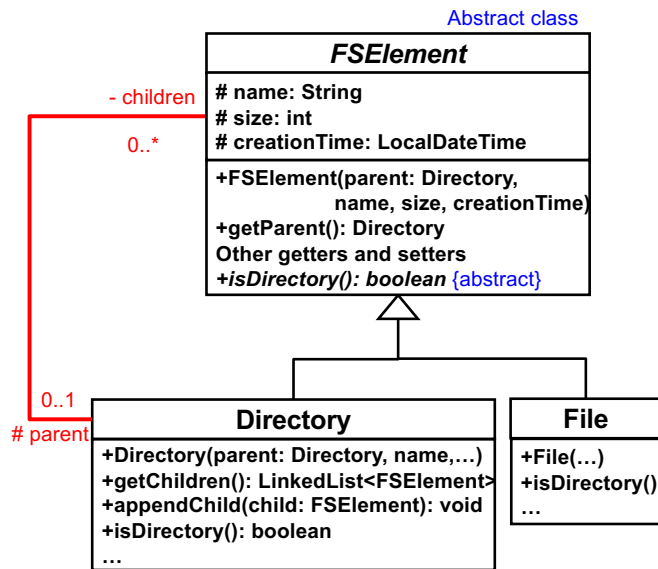
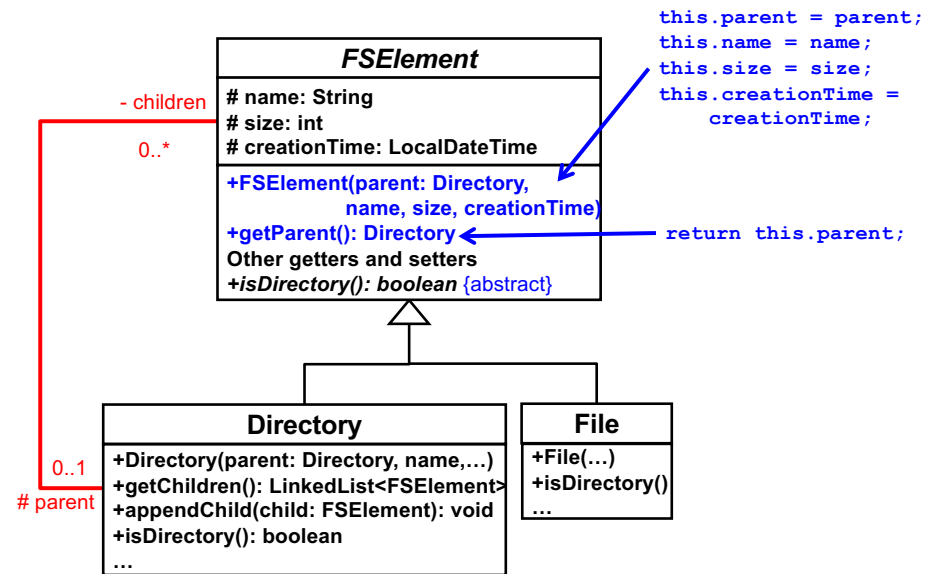


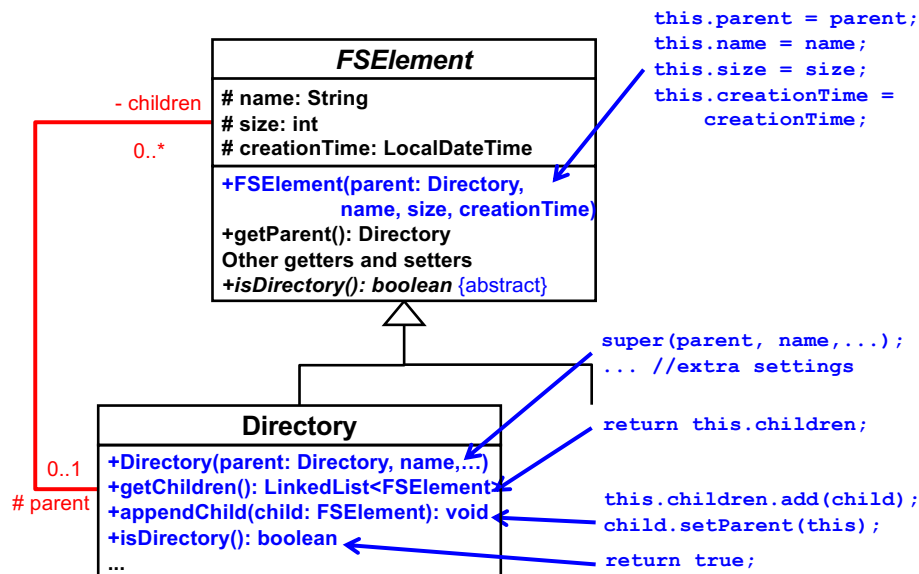
# Composite Design Pattern



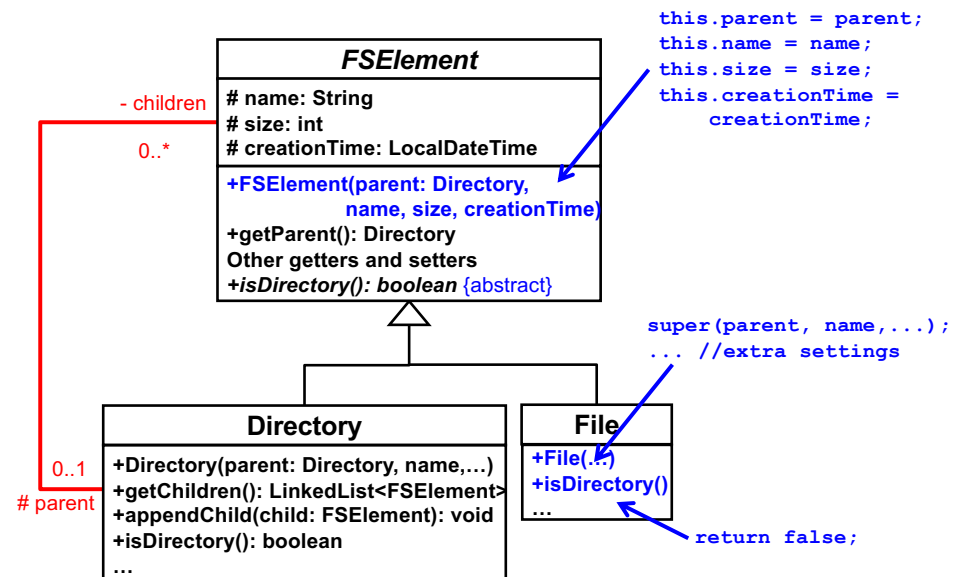
1



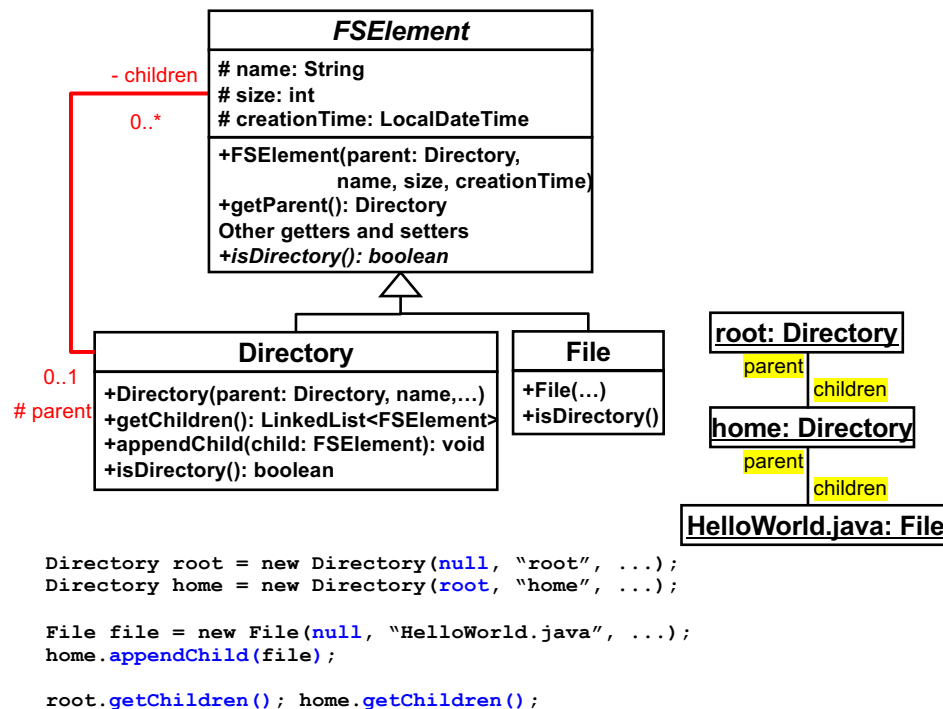
2



3



4



5

## Benefits of Composite

- Client code of a tree structure can **treat** individual objects and compositions of objects **uniformly**.

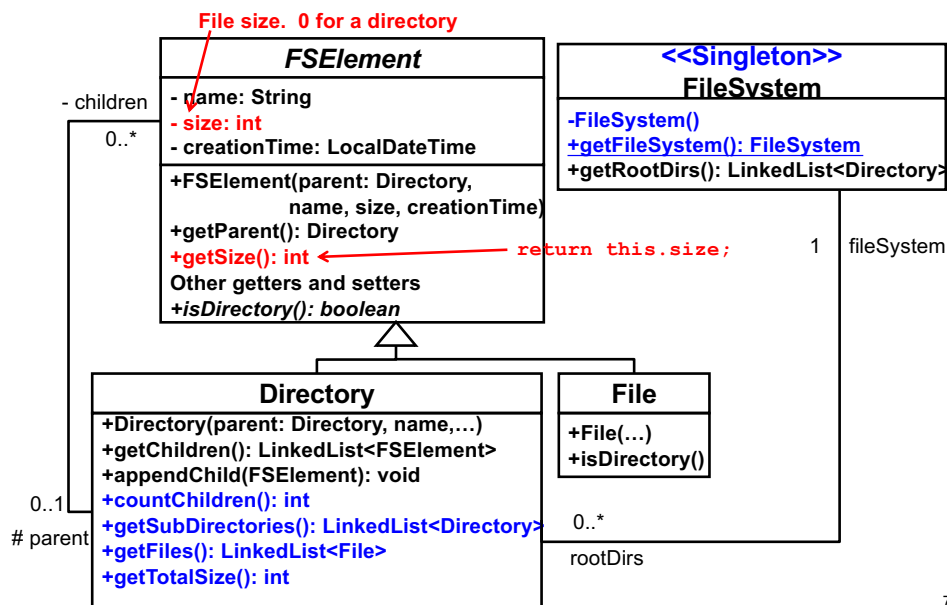
– “Treating...uniformly” means **performing polymorphism on ....**

```

- Directory dir = ...;
  for( FSElement fsElement: dir.getChildren() ){
      System.out.println( fsElement.getName() );
      System.out.println( fsElement.getSize() );
      System.out.println( fsElement.getParent().getName() ); }
  
```

6

## HW 6: Implement This



7

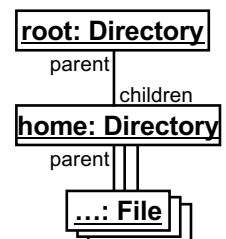
### • Directory

- **LinkedList<FSElement> children**: data field to reference files and subdirectories in the directory
- **countChildren()**: returns the number of files and subdirectories in the directory.
- **getTotalSize()**: returns the total disk consumption by all the files and subdirectories under the directory
  - Call **getTotalSize()** recursively on all sub-directories

```

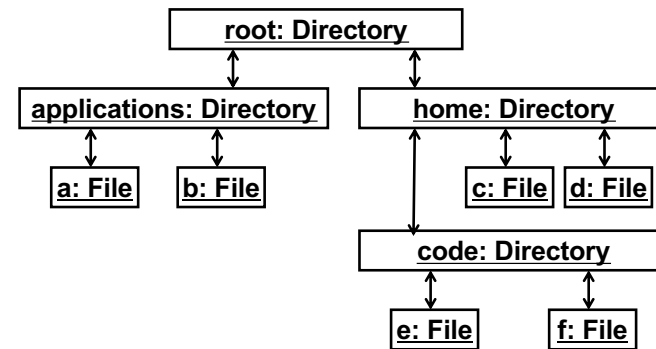
Directory root = new Directory(null, "Root", ...);
Directory home = new Directory(root, "home", ...);
File file = new File(home, "...", ...);
File file = new File(home, "...", ...);
File file = new File(home, "...", ...);

root.getTotalSize();
  
```



- **FileSystem** (*Singleton* class)
  - Implement a static factory method: `getFileSystem()`
  - Define a (empty) private constructor
- Add any extra methods in any classes as you like.
  - The class diagram in a previous slide is not complete.
    - e.g., `addRootDir()` in `FileSystem`?

- Use this file system structure in your test cases.
  - Create this file system structure as a *test fixture*.
  - Assign values to data fields (size, etc) as you like.



9

10

- Unit testing
  - Test each public method with `DirectoryTest`, `FileTest`, `FileSystemTest`
  - **DirectoryTest**
    - No need to define a test method for each getter method in `Directory`
    - Implement an equality-check for each `Directory` instance by calling getter methods of `Directory`
      - You can define your own logic for equality check.
      - c.f. HW 5, which performs equality-check for different `Car` instances
  - **FileTest**
    - Follow the instructions given for `DirectoryTest`

```

• public DirectoryTest{
  ...
  private String dirToStringArray(Directory d){
    String[] dirInfo = {
      d.isFile(), d.getName(), d.getSize(),
      d.getCreationTime(), d.getParent().getName(),
      // Call FSElement's methods and put returned values
      d.countChildren(), ...
    };
    // Call Directory's methods and put returned values };
    return dirInfo; }

  @Test
  public void verifyDirectoryEqualityRoot(){
    String[] expected = {..., ..., ...};

    Directory actual = ...;
    assertEquals(expected,
      dirToStringArray(actual) ); }

  @Test
  public void verifyDirectoryEqualityHome (){ ... }

  ...
}
  
```

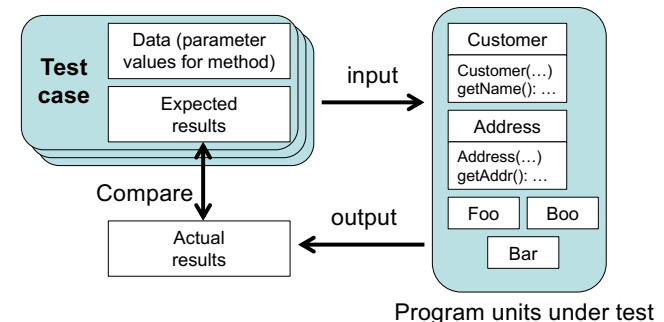
11

12

# Test Fixtures

- Deadline: Nov 5 (Tue) midnight

- Fixture
  - An instance of a class under test
  - An instance of another class that the class under test depends on
  - Input data
  - Expected result(s)
  - Set up of a file(s) and other resources
    - e.g., Socket
  - Set up of external systems/frameworks
    - e.g. Database, web server, web app framework, emulator (e.g. Android emulator)



13

14

## Setting up Fixtures

- Class under test

```
public class Calculator{
    public int multiply(int x, int y){
        return x * y;
    }
    public float divide(int x, int y){
        if(y==0) throw
            new IllegalArgumentException(
                "division by zero");
        return (float)x / (float)y;
    }
}
```

- Test class

```
public class CalculatorTest{
    @Test
    public void multiply3By4(){
        Calculator cut = new Calculator();
        float actual = cut.multiply(3,4);
        float expected = 12;
        assertEquals(expected, actual); }

    @Test
    public void divide3By2(){
        Calculator cut = new Calculator();
        float actual = cut.divide(3,2);
        float expected = 1.5f;
        assertEquals(expected, actual)); }

    @Test
    public void divide5By0(){
        Calculator cut = new Calculator();
        try{
            cut.divide(5, 0);
            fail("Division by zero"); }
        catch(IllegalArgumentException ex){
            assertEquals("division by zero",
                ex.getMessage() ); } } }
```

Setting up fixtures  
-- inline setup

15

## Implicit Setup

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.AfterAll;

public class RectangleTest{
    private Rectangle cut;

    @BeforeAll
    public void setUp(){
        cut = new Calculator(); }

    @Test
    public void multiply3By4(){
        float actual = cut.multiply(3,4);
        float expected = 12;
        assertEquals(expected, actual); }

    @Test
    public void divide3By2(){
        float actual = cut.divide(3,2);
        ... }

    @Test
    public void divide5By0(){
        ... }

    @AfterAll
    public void doSomething(){...} }
```

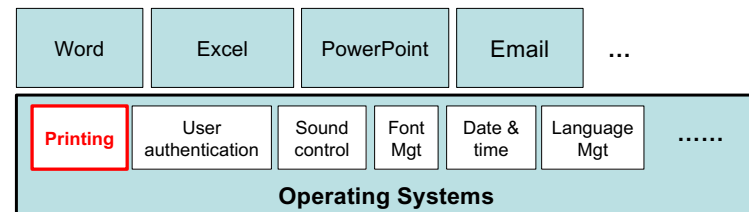
16

# Quiz: Printing API

- Implicit setup makes a test class less redundant.
- Flow of execution
  - @BeforeAll setUp()
  - @Test multiply3By4()
  - @Test divide3By2()
  - @Test divide5By0()
  - @AfterAll doSomething()
- The @BeforeAll method runs **before** all test methods.
- The @AfterAll method runs **after** all test methods.
- JUnit may run the test methods in a different order from their ordering in source code.

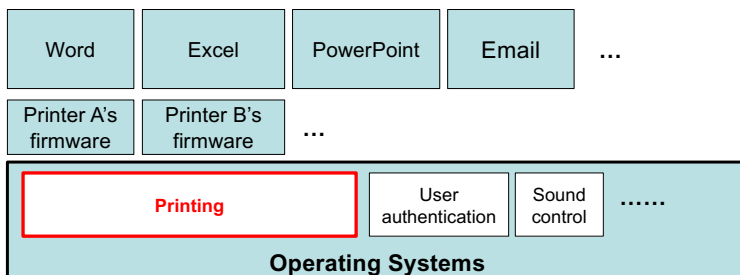
17

- Imagine a series of foundation services (or system APIs) that an operating system provides for apps.



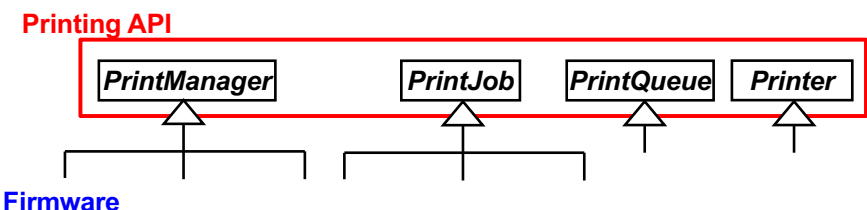
18

- Suppose you are trying to implement a **printing API** for the OS.
  - Your API will implement what are common among printer firmware.
    - e.g., print job, print queue, etc.
  - Firmware developers will use your API to implement their firmware
  - Apps will call firmware and your API to do printing.



19

- The printing API has 4 classes:
  - **PrintManager**
    - Used by an app to start printing.
    - Creates a PrintJob, sends it to a PrintQueue and manages its lifecycle.
  - **PrintJob**
    - Represents a print job.
  - **PrintQueue**
  - **Printer**
- They implement what are common among printer firmware.



20

- API classes:

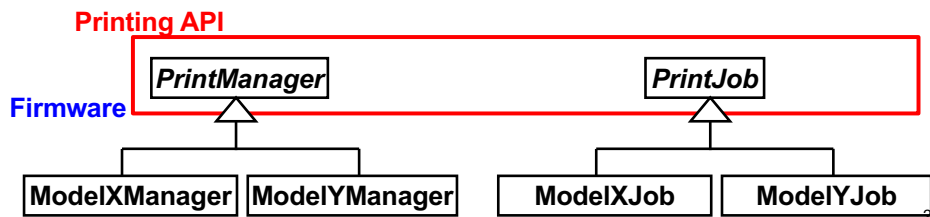
- **PrintManager**

- Used by an app to start printing.
    - Creates a PrintJob, sends it to a PrintQueue and manages its lifecycle.

- **PrintJob**

- They implement what are common among printer firmware.

- Each firmware uses/extends them to implement its own printing service.



- Let's focus on the **creation of print jobs**.

- Requirements:

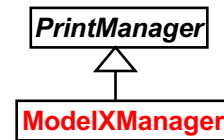
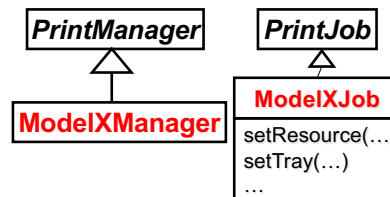
- There is a **common procedure** that the printing API wants to run when creating a new print job.

- You want to enforce this procedure in all firmware.
    - You want to implement it in PrintManager.

- e.g. Check if the printer is online and create a print job if the printer is online.

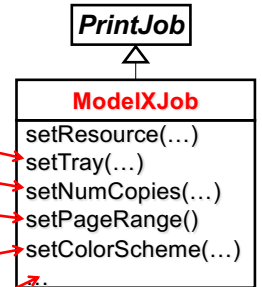
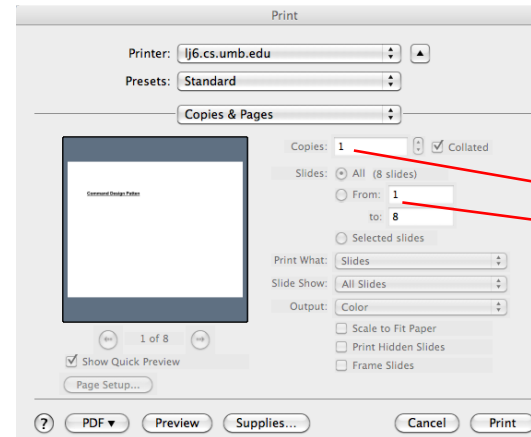
- toner level?
    - jammed?

- How to implement this without knowing about subclasses (i.e., without stating subclasses)?

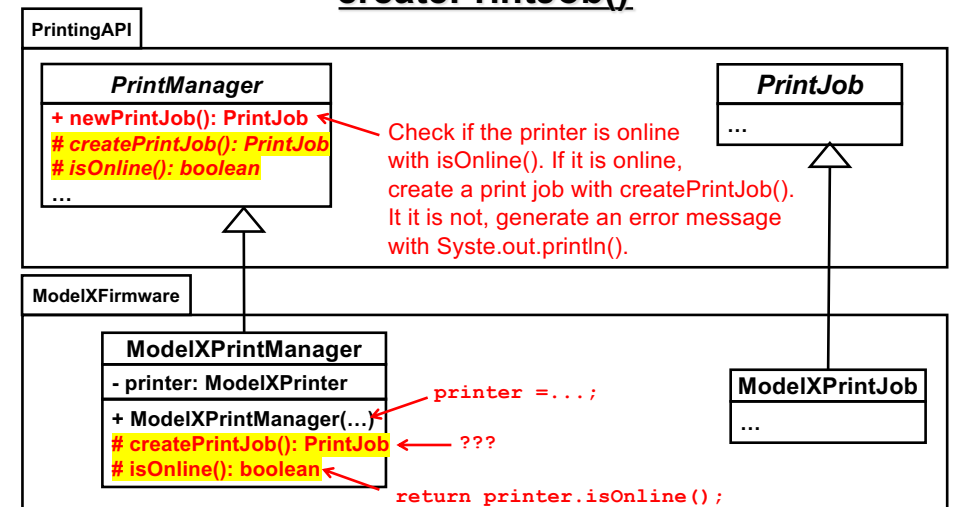


ModelXManager pops up a print configuration window and allows the user to set up a print job.

It creates an instance of **ModelXJob** once the user clicks "Print," and then sends it to a print queue.



## Write Pseudo Code for PrintManager's newPrintJob() and ModelXPrintManager's createPrintJob()

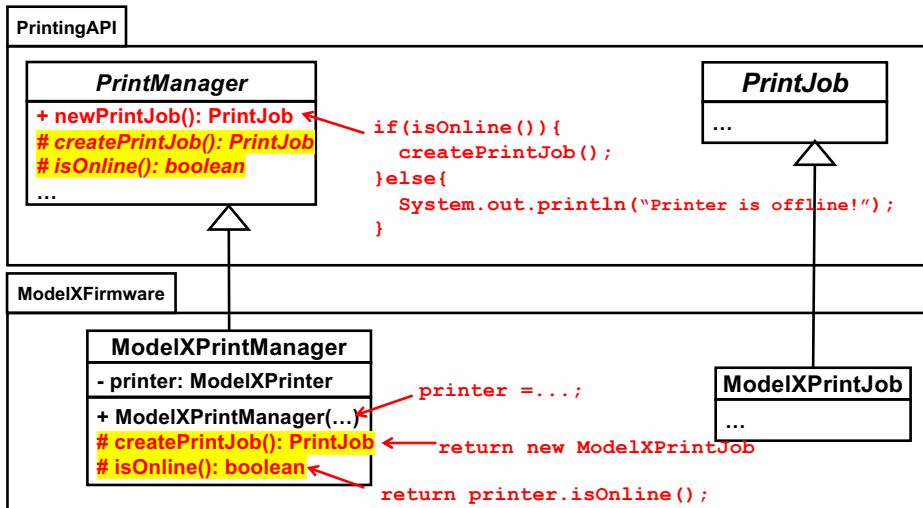


Check if the printer is online with isOnline(). If it is online, create a print job with createPrintJob(). If it is not, generate an error message with `Syste.out.println()`.

printer = ...;  
 ???  
 return printer.isOnline();

```

ModelXPrintManager manager = new ModelXPrintManager(...);
manager.newPrintJob();
  
```



```

ModelXPrintManager manager = new ModelXPrintManager(...);
manager.newPrintJob();
  
```