BITS Pilani
Pilani Campus

Object Oriented Programming
CS F213
Amit Dua

Slides Taken from the slides prepared by Dr. Jennifer innovate achieve lead

BITS Pilani
Pilani Campus

SOLID design principles

Topics for today

- SOLID

Introduction

- conceptualized by Robert C. Martin
- Michael Feathers, introduced the SOLID acronym
- what is SOLID and how does it help us write better code?
- **design principles encourage us to create more maintainable, understandable, and flexible software**
- **as our applications grow in size, we can reduce their complexity**

SOLID

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

1. Single Responsibility

- **a class should only have one responsibility.**
- **Furthermore, it should only have one reason to change.**

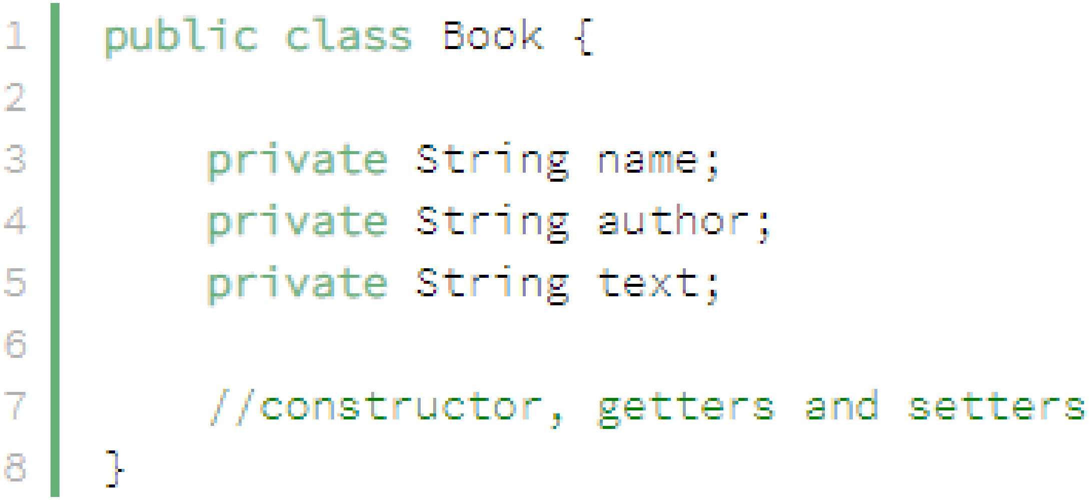
Usage:

Testing – A class with one responsibility will have far fewer test cases

Lower coupling – Less functionality in a single class will have fewer dependencies

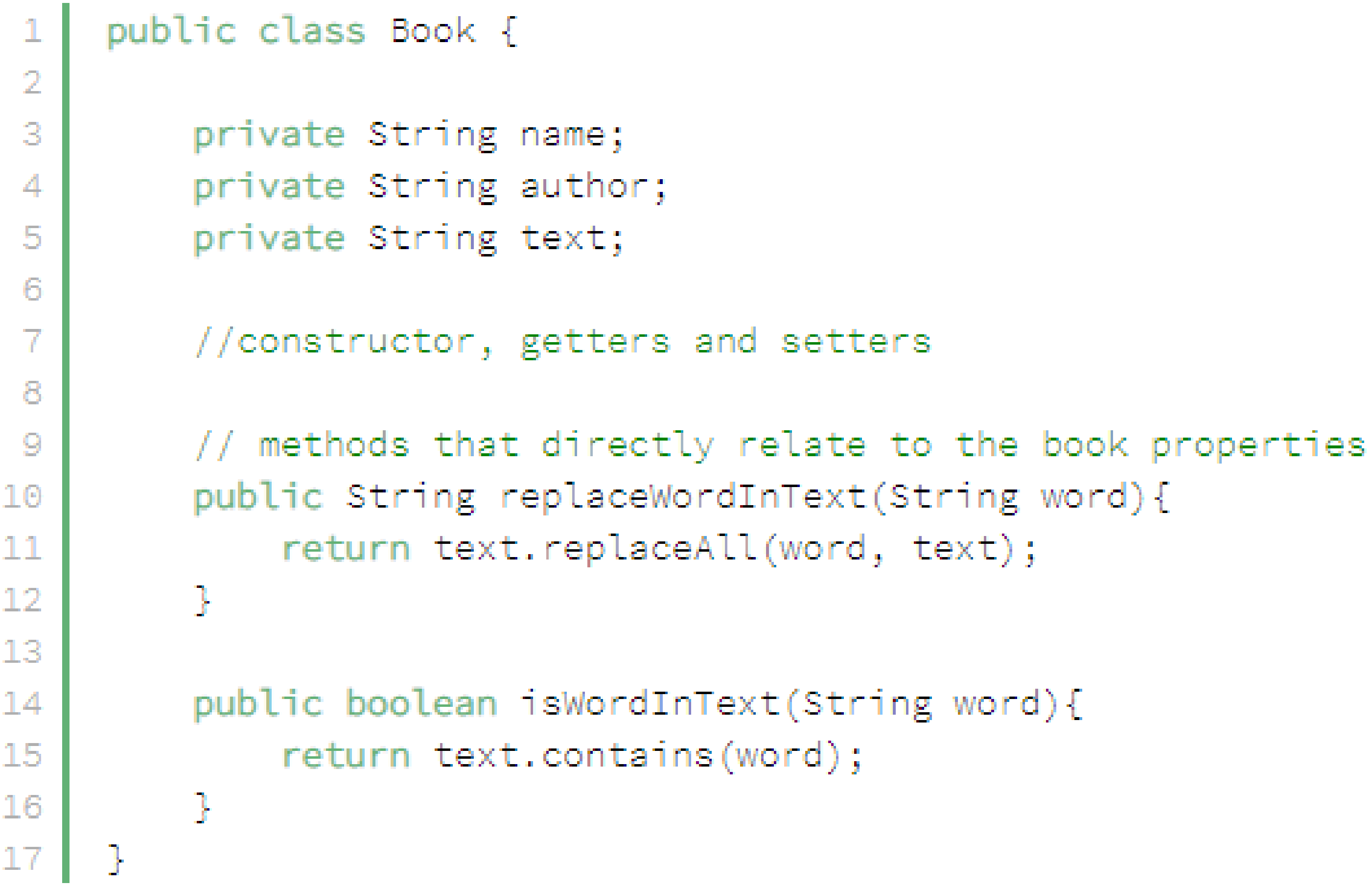
Organization – Smaller, well-organized classes are easier to search than monolithic ones

Book class



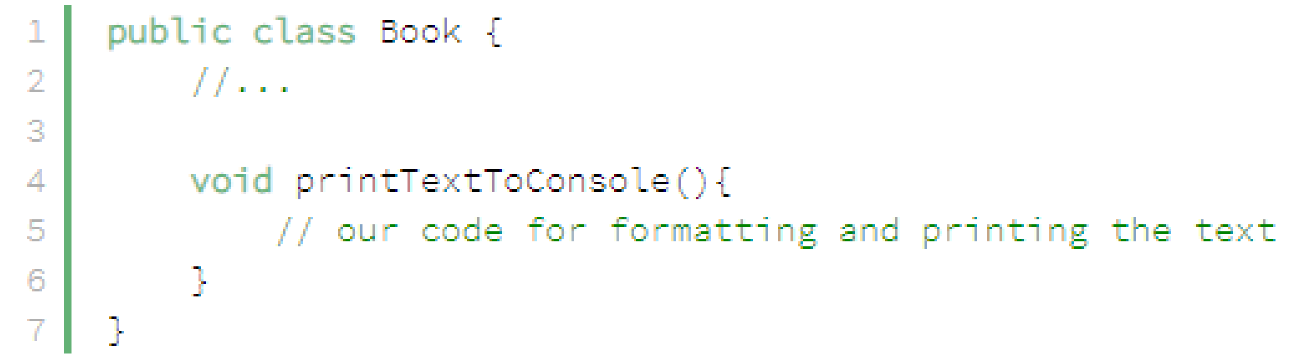
```java
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters
}
```

Book class contd..

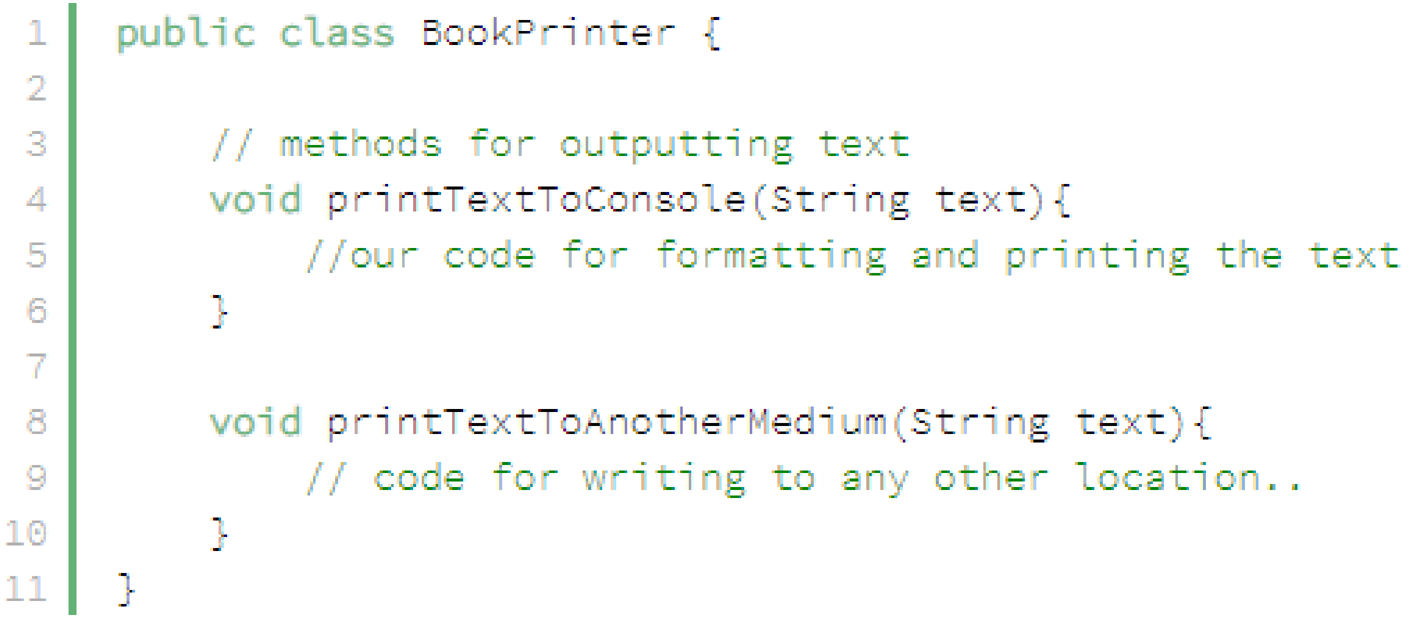


```java
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word){
        return text.replaceAll(word, text);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

Violation of ‘S’ principle and solution



```
public class Book {
    //...

    void printTextToConsole(){
        // our code for formatting and printing the text
    }
}
```



```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

2. Open for Extension, Closed for Modification

open-closed principle

- **classes should be open for extension,**
- **but closed for modification.**
- In doing so, we stop ourselves from modifying existing code and
- causing potential new bugs

Example

Phone company

-ISP

-phone subscriber

-common properties and methods

-new user VOIP. Create a new class?

-repetitions.

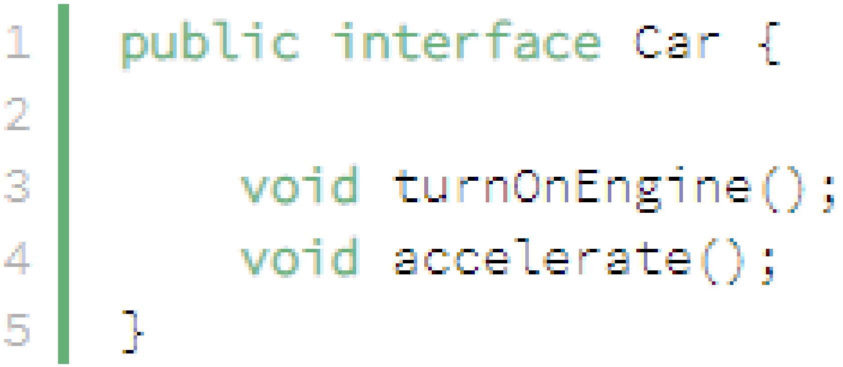
-so create an abstract class (close it for modification)

-have a method in it that allows abstract class extension

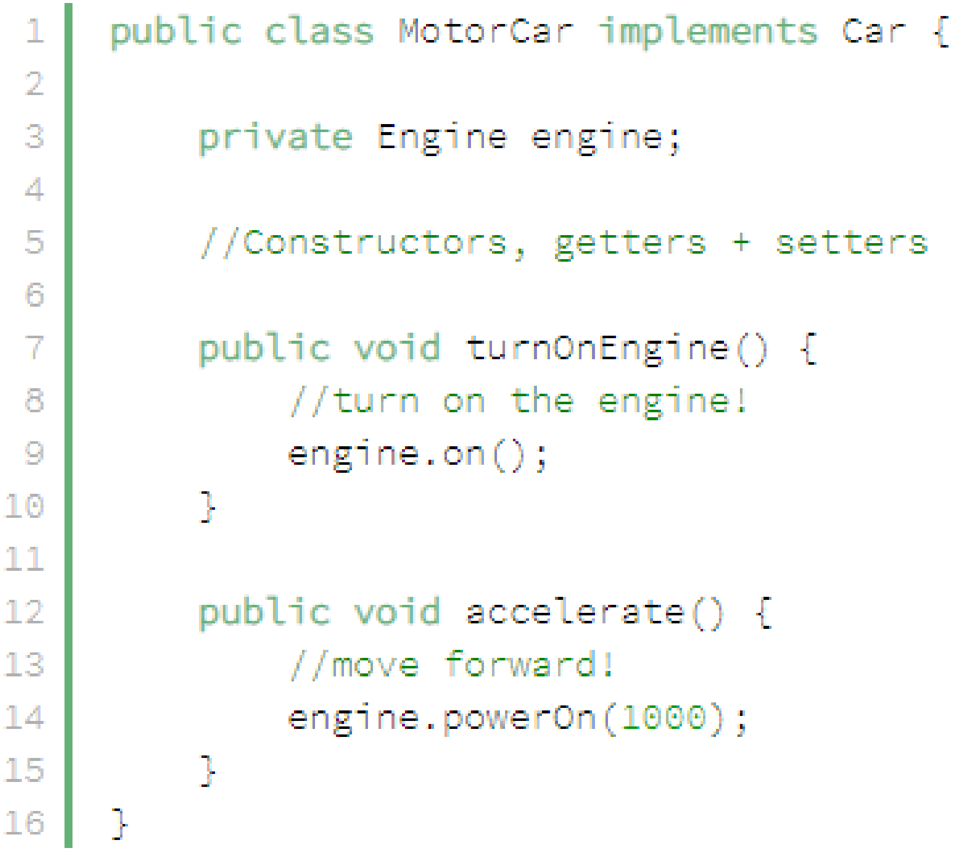
3. Liskov Substitution

- **if class *A* is a subtype of class *B*,**
- **then we should be able to replace *B* with *A* without disrupting the behavior of our program**

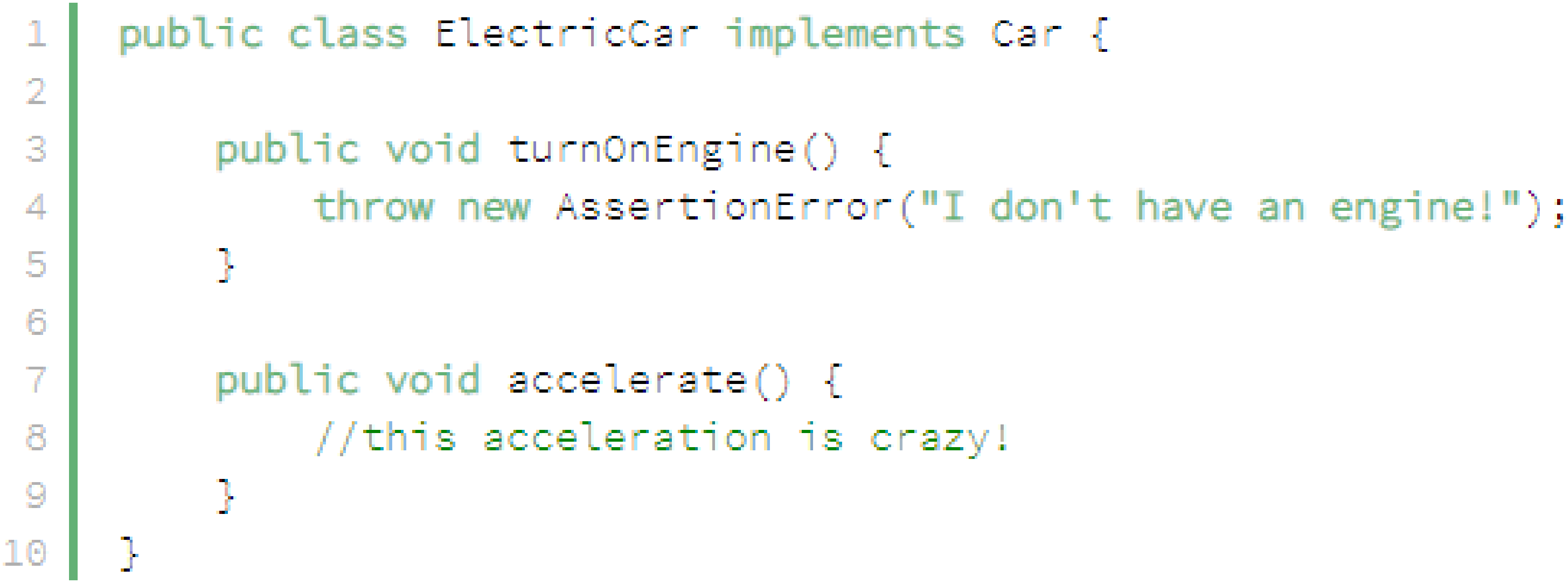
Example



```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```



```
public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```



```java
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

4. Interface Segregation

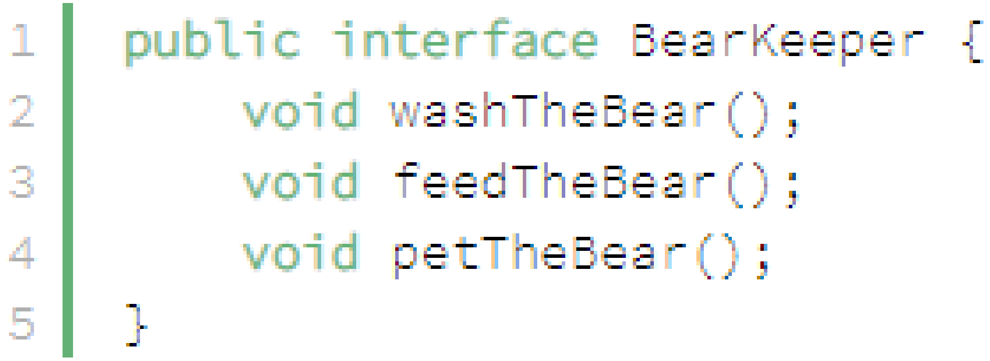
- **Clients should not be dependent on interfaces that they don’t use**

- **larger interfaces should be split into smaller ones.**
- **By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

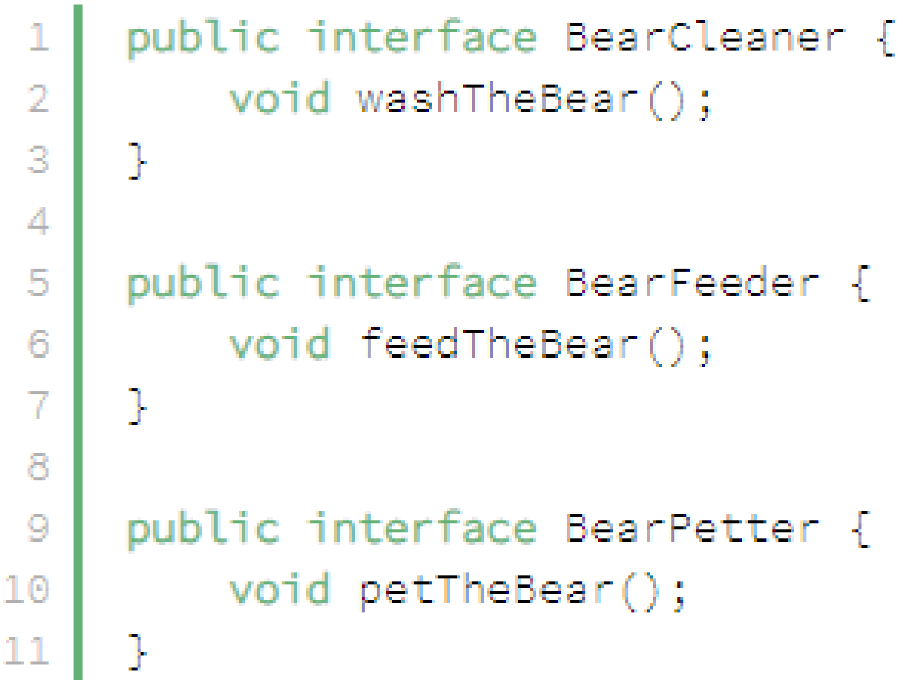
Classes with

- **empty method implementation**
- **throws UnsupportedOperationException**
- **Return null/dummy/default**

Example Interface segregation

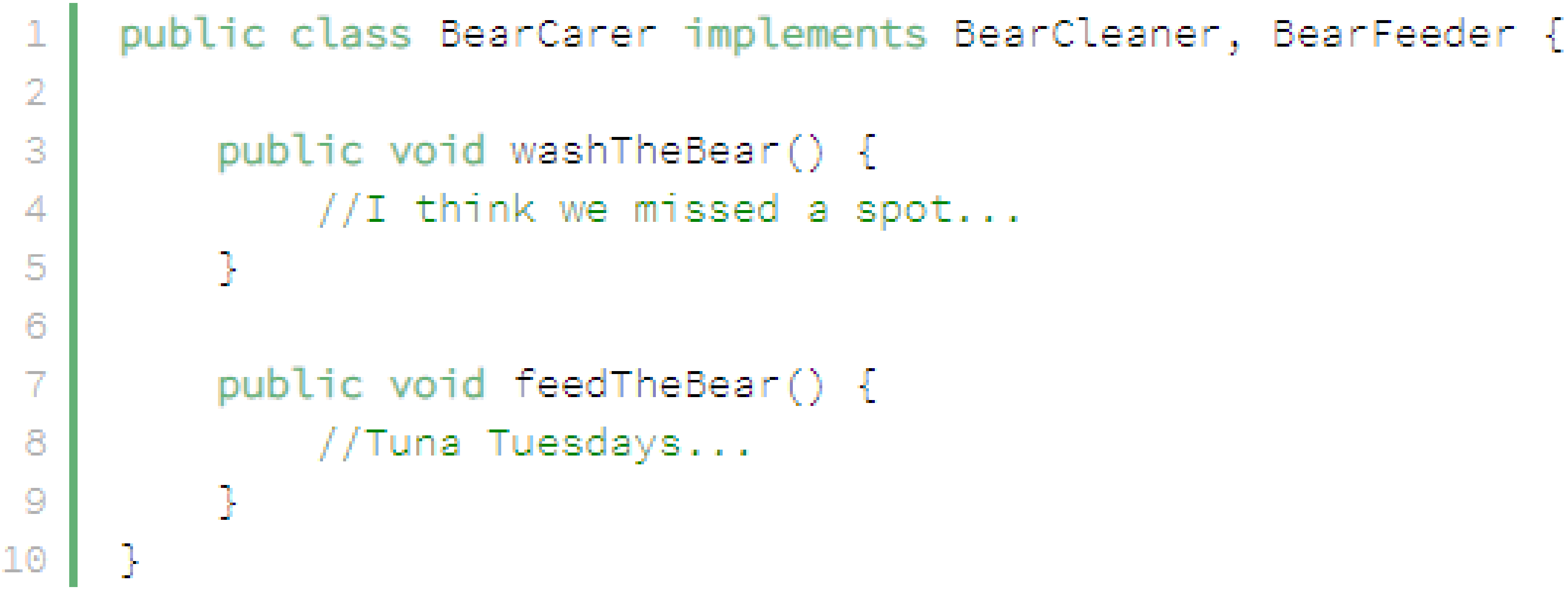


```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

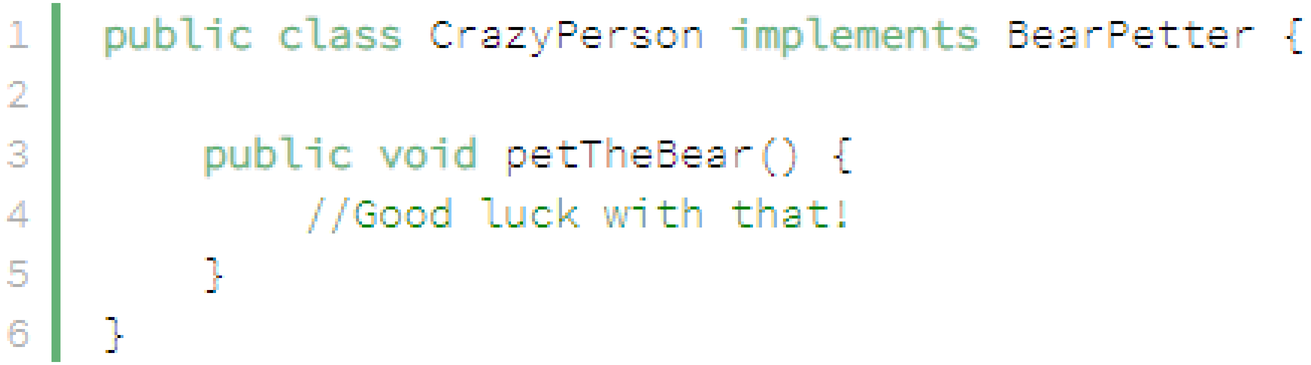


```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

Better approach



```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```



```
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

5. Dependency Inversion

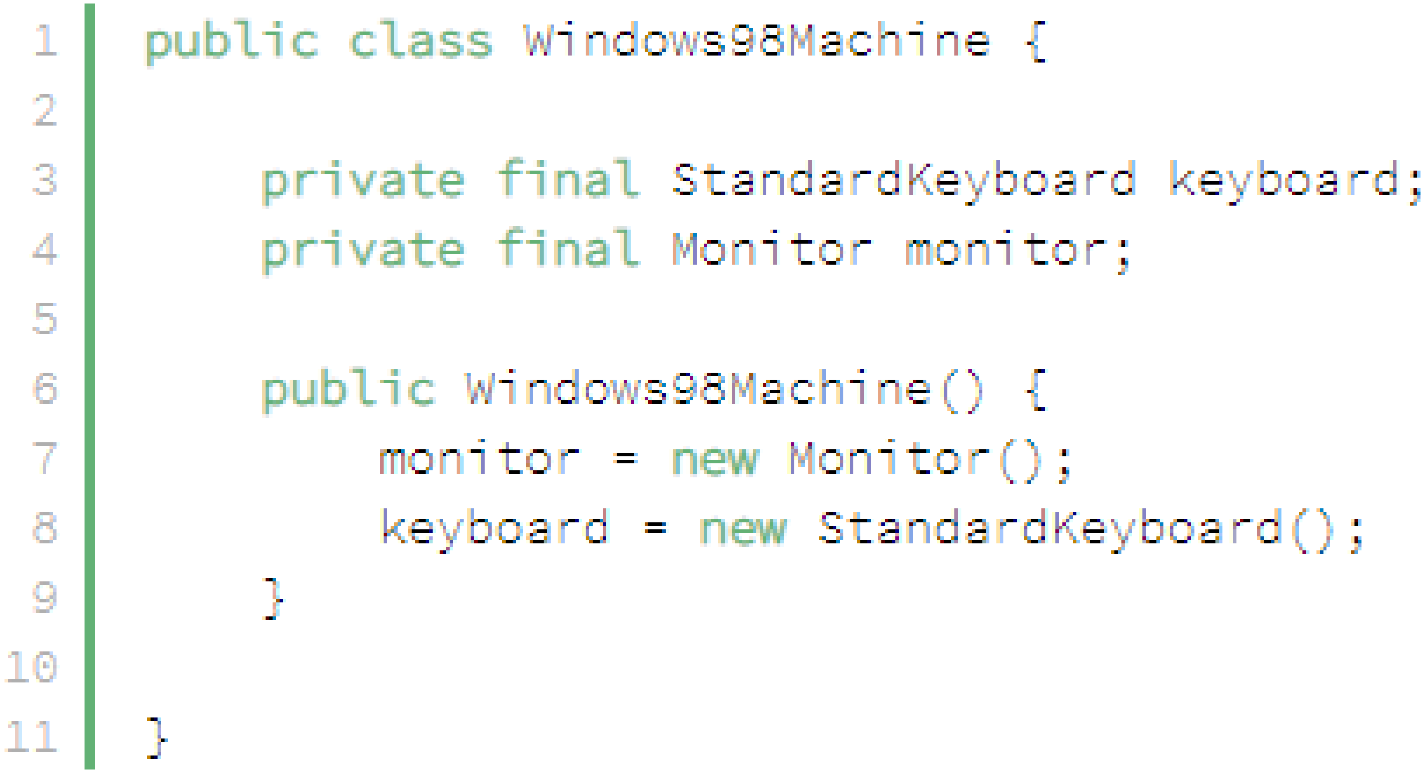
A. High level module should not depend on low level modules. Both should depend on abstractions

B. Abstractions should not depend on details. Details should depend on abstractions.

The principle of Dependency Inversion refers to the decoupling of software modules.

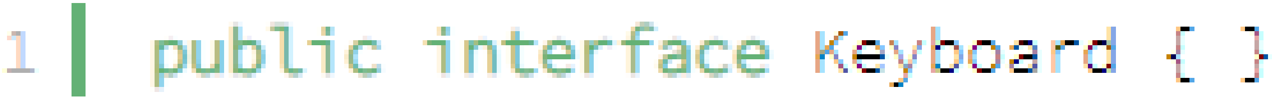
This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

Problem of violating ‘Dependency preservation’ principle

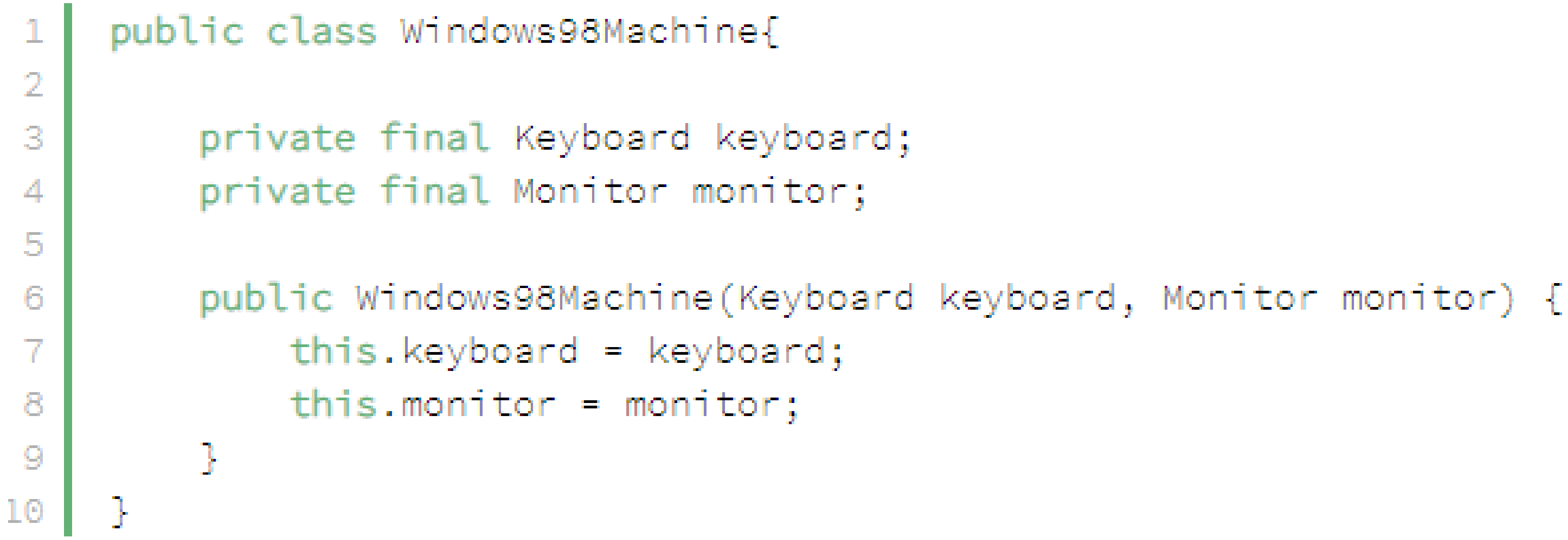


```
public class Windows98Machine {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

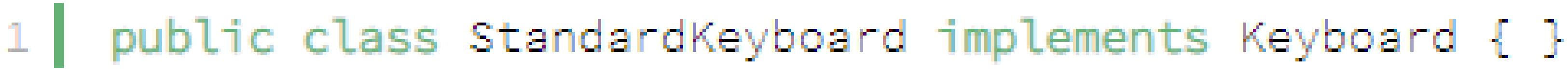
Solution



```
public interface Keyboard { }
```



```
public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```



```
public class StandardKeyboard implements Keyboard { }
```

Features of enum

- Enum is internally implemented using class

```
/* internally above enum Color is converted to
class Color {
public static final Color RED = new Color();
public static final Color BLUE = new Color();
public static final Color GREEN = new Color(); }*/
```

- Constants represents an object of type enum
- Constants are always implicitly public static final
  - It can be accessed using enum name
  - Child enums can not be created.
- It can be passed as an argument to switch statements

Features of enum

- All enums implicitly extend java.lang.Enum class
- toString() returns the enum constant name
- values() method can be used to return all values present inside enum
- ordinal() method is used to retrieve the constant index
- Enum can contain constructor and it is executed separately for each enum constant at the time of class loading.
- We cant create enum objects explicitly and hence we cannot invoke the enum constructor directly
- Enum can contain concrete method and not abstract methods.

Enum Example

Enumex.java

Garbage collector

- finalize()
  - This method is called before garbage collection when an object has no more references.
  - It could be overridden to dispose system resources, perform clean up and minimize memory leaks.
  - finalize() method is called just once on an object
  - protected void finalize()
- gc()
  - It is used to invoke the garbage collector to perform clean up
  - It is found in System and Runtime classes.
  - public static void gc()

Java Runtime class

- It is used to interact with the Java runtime environment

- It provides methods to execute a process, invoke GC, get total and free memory etc.

- Only one instance of the java.lang.Runtime class is available for one Java application

Garbage Collector : gc()

GarbageCollector.java

Finalize()

- The *finalize()* method called by Garbage Collector not JVM. Although Garbage Collector is one of the module of JVM.
- Object class *finalize()* method has empty implementation, thus it is recommended to override *finalize()* method to dispose of system resources or to perform other cleanup.
- The *finalize()* method is never invoked more than once for any given object.
- If an uncaught exception is thrown by the *finalize()* method, the exception is ignored and finalization of that object terminates.