

CS2030 AY18/19

SEM 2

WEEK 6 | 22 FEB 19

TA GAN CHIN YAO

DISCLAIMER

Slides are made by me, **unofficial, optional**
Available to download at **bit.ly/cs2030_gan**
Slides (if any) will be uploaded on
Friday weekly

CONCEPTS

1. Consider the following classes: `FormattedText` that adds formatting information to the text. We call `toggleUnderline()` to add or remove underlines from the text. A `URL` is a `FormattedText` that is always underlined.

```
class FormattedText {
    public String text;
    public boolean isUnderlined;
    public void toggleUnderline() {
        isUnderlined = (!isUnderlined);
    }
}

class URL extends FormattedText {
    public URL() {
        isUnderlined = true;
    }

    public void toggleUnderline() {
        return;
    }
}
```

Does the above violate Liskov Substitution Principle? Explain.

Yes, it violates Liskov Substitution Principle (LSP). The “desirable property” here is that `toggleUnderline()` toggles the `isUnderlined` flag. Since `URL` changes the behavior of `toggleUnderline()`, this property no longer holds for subclass `URL`. Places in a program where the super-class (i.e. `FormattedText`) is used cannot be simply replaced by the sub-class (i.e `URL`).

In general, a subclass is also a superclass (i.e. a `URL` is also a `FormattedText`). So anywhere we are expecting a superclass object, we should be able to give a subclass object in place of the superclass object. If there are cases where we cannot give a subclass object for places expecting a superclass object (e.g. some properties will be broken), then this is an indication that LSP is violated, and the subclass should not inherit from the superclass at all. For this case, if we expect a `FormattedText` object and proceed to want to `toggleUnderline()` from off to on, or from on to off, a `URL` object cannot serve this purpose.

2. Consider the following program fragment.

```
class A {  
    int x;  
    A(int x) {  
        this.x = x;  
    }  
    public A method() {  
        return new A(x);  
    }  
}
```

```
class B extends A {  
    B(int x) {  
        super(x);  
    }  
    @Override  
    public B method() {  
        return new B(x);  
    }  
}
```

Does it compile? What happens if we switch the method definitions between class A and class B instead? Give reasons for your observations.

It will compile. There is no compilation error in the given program fragment as any existing code that invokes A's method prior to being inherited would still work if the code invokes B's method instead, after B inherits A. Saying that LSP is not violated is not exactly right, as Java does not check LSP violations during compilation (notice that the questions asks whether the program compiles, and not whether the program violates LSP).

`public A method() {}` means returning a reference to an object of type A. It does not mean return a reference to an A object. Notice the difference (return object of type A NOT EQUALS return A object). Hence, when you override it and return a reference to a B object instead, you are still returning a reference to an object of type A, because a B object is also an object of type A since B extends A.

Side note: This question was actually a AY17/18 SEM2 Mid-term question.
This was the answer key given:

Solution: Existing code that has been written to invoke A's copy would still work if the code invokes B's copy instead after B inherits from A.

The following answers are insufficient / wrong.

- B is subtype of A so it is OK. This does not explain in the context of the return type in a overriding method. For instance, Java does not allow `foo(B x)` to override `foo(A x)`. But it is close since subtyping implies that code written expecting objects of type A can be used with objects of type B. (2 marks) However, if you elaborated on subtyping of B from A to any of the following, then you are not applying the concept of subtyping correctly to answer this question.
- B's `copy()` returns an object of type B, so it is safe to change the return type to B. This does not answer the question, which is why Java allows overriding (not why Java allows the return type to be B, which is rather obvious). (0 marks)
- It does not violate LSP. B's copy still returns a copy of the object. This has to do with the semantic of the program. Java's compiler does not check for the semantic and violation of LSP. (0 marks).
- There is no ambiguity to which version of copy will be invoked; Or, the return type is not part of a method signature. This does not explain why Java does not allow, say, `int copy()` B to override A `copy()` in A. (0 marks)
- Assigning B to A is a widening conversion so it is OK. Again, this does not explain why Java allows overriding. (0 marks).

Side note: What happens if you swap instead, i.e. the following:

```
class A {
    public B method() {
        return new B();
    }
}

class B extends A {
    @Override
    public A method() {
        return new A();
    }
}
```

It will not compile. *A*'s method now returns a reference to a *B* object, but overriding it with a method that returns a reference-type *A* does not guarantee that the object is an object of type *B*. So the overriding is not allowed and results in a compilation error.

```
class A {
    public B method() {
        return new B();
    }
}

class B extends A {
    @Override
    public A method() {
        return new C();
    }
}

// Note C extends A, not B
class C extends A {}
```

Because I can end up in situation like this, where I return a *C* instead. This *C* is not related to *B* at all.

i.e. Even though my subclass *A* can point to a *B* object, it can very well point to other object not related to *B* at all. Hence, java don't allow you to do this.

3. What is the output of the following program fragment? Explain.

```
class A {  
    static void f() throws Exception {  
        try {  
            throw new Exception();  
        } finally {  
            System.out.print("1");  
        }  
    }  
  
    static void g() throws Exception {  
        System.out.print("2");  
        f();  
        System.out.print("3");  
    }  
  
    public static void main(String[] args) {  
        try {  
            g();  
        } catch (Exception e) {  
            System.out.print("4");  
        }  
    }  
}
```

Output: 214

- Since main() calls g(), 2 will be printed first.
- f() is then executed, which leads to an exception being thrown.
- Before f() returns, it executes the finally block, leading to 1 being printed.
- After returning to g(), since an exception was thrown, 3 will NOT be printed, with control returning to main(), which catches the exception.
- The catch block is executed and 4 is then printed.

4. You are given two classes MCQ and TFQ that implements a question-answer system:

- MCQ: multiple-choice questions comprising answers: A B C D E
- TFQ: true/false questions comprising answers: T F

```
class MCQ {  
    String question;  
    char answer;  
  
    public MCQ(String question) {  
        this.question = question;  
    }  
  
    void getAnswer() {  
        System.out.print(question + " ");  
        answer = (new Scanner(System.in)).next().charAt(0);  
        if (answer < 'A' || answer > 'E') {  
            throw new InvalidMCQException("Invalid MCQ answer");  
        }  
    }  
}
```

```

class TFQ {
    String question;
    char answer;

    public TFQ(String question) {
        this.question = question;
    }

    void getAnswer() {
        System.out.print(question + " ");
        answer = (new Scanner(System.in)).next().charAt(0);
        if (answer != 'T' && answer != 'F') {
            throw new InvalidTFQException("Invalid TFQ answer");
        }
    }
}

```

In particular, an invalid answer to any of the questions will cause an exception (either `InvalidMCQException` or `InvalidTFQException`) to be thrown.

```

class InvalidMCQException extends IllegalArgumentException {
    public InvalidMCQException(String msg) {
        super(msg);
    }
}

class InvalidTFQException extends IllegalArgumentException {
    public InvalidTFQException(String msg) {
        super(msg);
    }
}

```

By employing the various object-oriented design principles, design a more general question-answer class QA that can take the place of both MCQ and TFQ types of questions (and possibly more in future, each with their own type of exceptions).

Here are some design issues that needs to be addressed:

- Abstract out common code in TFQ and MCQ to a common place QA
- MCQ and TFQ inherits from QA
- QA's abstract method `getAnswer()` throws a more general exception than MCQ and TFQ
- Create the general exception class `InvalidQuestionException`

Sample answer from the prof:

```

import java.util.*;

abstract class QA {
    String question;
    char answer;

    public QA(String question) {
        this.question = question;
    }

    abstract void getAnswer() throws InvalidQuestionException;

    char askQuestion() {
        Scanner sc = new Scanner(System.in);
        System.out.print(question + " ");
        return sc.next().charAt(0);
    }
}

class MCQ extends QA {
    public MCQ(String question) {
        super(question);
    }

    void getAnswer() {
        answer = askQuestion();
        if (answer < 'A' || answer > 'E') {
            throw new InvalidMCQException("Invalid MCQ answer");
        }
    }
}

```

```

class TFQ extends QA {
    public TFQ(String question) {
        super(question);
    }

    void getAnswer() {
        answer = askQuestion();
        if (answer != 'T' && answer != 'F') {
            throw new InvalidTFQException("Invalid TFQ answer");
        }
    }
}

class InvalidQuestionException extends IllegalArgumentException {
    public InvalidQuestionException(String msg) {
        super(msg);
    }
}

class InvalidMCQException extends InvalidQuestionException {
    public InvalidMCQException(String msg) {
        super(msg);
    }
}

class InvalidTFQException extends InvalidQuestionException {
    public InvalidTFQException(String msg) {
        super(msg);
    }
}

```

Sample client class:

```
class Main {  
    public static void main(String[] args) {  
        try {  
            QA mcq = new MCQ("What is the answer to this MCQ?");  
            QA tfq = new TFQ("What is the answer to this TFQ?");  
  
            mcq.getAnswer();  
            tfq.getAnswer();  
        } catch (InvalidQuestionException ex) {  
            System.err.println(ex);  
        }  
    }  
}
```

5. In each of the following program fragments, will it compile? If so, what will be printed?

```
(a) class Main {  
    static void f() throws IllegalArgumentException {  
        try {  
            System.out.println("Before throw");  
            throw new IllegalArgumentException();  
            System.out.println("After throw");  
        } catch (IllegalArgumentException e) {  
            System.out.println("Caught in f");  
        }  
    }  
}
```

Compile error.

error: unreachable statement

Unreachable statement because of
throwing Exception before this line

```
    public static void main(String[] args) {  
        try {  
            System.out.println("Before f");  
            f();  
            System.out.println("After f");  
        } catch (Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

```
(b) class Main {  
    static void f() throws IllegalArgumentException {  
        try {  
            throw new IllegalArgumentException();  
        } catch (IllegalArgumentException e) {  
            System.out.println("Caught in f");  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            System.out.println("Before f");  
            f();  
            System.out.println("After f");  
        } catch (Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

Compile ok.

Output:

Before f

Caught in f

After f

Unlike part (a), this line is reachable. Note that **throws** does not mean always throwing **Exception**. It just mean the method **can throw** **Exception**. Hence, **f()** may or may not throw **Exception**. The compiler cannot tell, therefore the next line after **f()** is reachable.

In part (a), the compile can tell because the line is directly after **throw new IllegalArgumentException();**

```
(c) class Main {  
    static void f() throws IllegalArgumentException {  
        try {  
            throw new Exception();  
        } catch (IllegalArgumentException e) {  
            System.out.println("Caught in f");  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            System.out.println("Before f");  
            f();  
            System.out.println("After f");  
        } catch (Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

Compile error.

error: unreported exception Exception; must be caught or declared to be thrown

Exception is treated as a checked exception. When you throw a checked exception, you must either catch it, or declare a **relevant** throws clause. In this case, the catch is not catching the Exception (it is catching another unrelated exception - IllegalArgumentException), and the throws is unrelated to Exception as well, hence this checked exception is not handled.

Note: when you specify throws for a checked exception, typically you will specify the same name as the exception being thrown. You can specify a superclass of the exception being thrown in the throws clause, but not a subclass of it.


```
(d) class Main {  
    static void f() throws Exception {  
        try {  
            throw new IllegalArgumentException();  
        } catch (Exception e) {  
            System.out.println("Caught in f");  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            System.out.println("Before f");  
            f();  
            System.out.println("After f");  
        } catch (Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

Compiles fine.

Output:

Before f

Caught in f

After f

Ok. Superclass variable can refer to subclass object.
`IllegalArgumentException` is a subclass of `Exception`.
Hence, we can catch `IllegalArgumentException` with
`IllegalArgumentException` itself or any of its superclass.

Therefore, the `IllegalArgumentException` thrown is
caught in this catch clause by `Exception e`

Since `IllegalArgumentException` is already caught in method `f()`,
this catch will not be trigger when `f()` returns

```
(e) class Main {  
    static void f() throws Exception {  
        try {  
            throw new ArrayIndexOutOfBoundsException();  
        } catch (IllegalArgumentException e) {  
            System.out.println("Caught in f");  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            System.out.println("Before f");  
            f();  
            System.out.println("After f");  
        } catch (Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

Compiles fine.

Output:
Before f
Caught in main

`IllegalArgumentException` is **not** a superclass of `ArrayIndexOutOfBoundsException`. Hence, `IllegalArgumentException` cannot catch an `ArrayIndexOutOfBoundsException` which was thrown. Therefore, the exception propagates back to the caller of `f()` via the `throws Exception` and it is not handled in `f()` itself

This print will not be executed since an `ArrayIndexOutOfBoundsException` was thrown back by the `f()` method. Control then proceeds to the bottom catch to handle the exception

```
(f) class Main {
    static void f() throws Exception {
        try {
            throw new ArrayIndexOutOfBoundsException();
        } catch (IllegalArgumentException e) {
            System.out.println("Caught IA exception in f");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught AIOOB exception in f");
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("Before f");
            f();
            System.out.println("After f");
        } catch (Exception e) {
            System.out.println("Caught in main");
        }
    }
}
```

Compiles fine.

Output:

Before f

Caught AIOOB exception in f

After f

If there are multiple catches, it works in a top-down manner. The first catch clause is evaluated, and if it does not catch, proceed to the next, until one catch clause can catch the exception thrown. Then, all other catch clause below will be skipped.

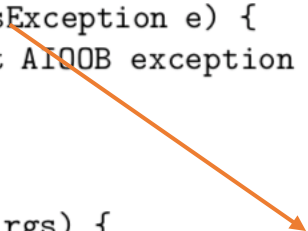
Since `f()` method has a `catch(ArrayIndexOutOfBoundsException)` to handle the exception, no error is thrown back from `f()`, hence this catch is never executed.

```
(g) class Main {
    static void f() throws Exception {
        try {
            throw new ArrayIndexOutOfBoundsException();
        } catch (Exception e) {
            System.out.println("Caught exception in f");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught AIOOB exception in f");
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("Before f");
            f();
            System.out.println("After f");
        } catch (Exception e) {
            System.out.println("Caught in main");
        }
    }
}
```

Compile error.

error: exception ArrayIndexOutOfBoundsException has already been caught



Since multiple catches work in a top-down manner, `ArrayIndexOutOfBoundsException` must have already been caught earlier in the `catch(Exception e)` clause. This is because `ArrayIndexOutOfBoundsException` is a subclass of `Exception`. Recall superclass variable can refer to subclass object. Hence, this line is redundant and will never be reached if `ArrayIndexOutOfBoundsException` is indeed thrown.

```

(h) class Main {
    static void f() throws Exception {
        try {
            throw new ArrayIndexOutOfBoundsException();
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught AIOOB exception in f");
        } catch (Exception e) {
            System.out.println("Caught exception in f");
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("Before f");
            f();
            System.out.println("After f");
        } catch (Exception e) {
            System.out.println("Caught in main");
        }
    }
}

```

Compiles fine.

Output:

Before f

Caught AIOOB exception in f

After f

Ok can. The general idea is to catch a more specific exception first (i.e. subclass), then move on to more general (i.e. superclass) to “cover other possible cases”. In this case, the `catch(Exception e)` works like a last resort and always placed in the last line in case all other previous catches fail. Note that `Exception` can catch all exceptions (because `Exception` is the superclass of all exceptions)

```
(i) class Main {
    static void f() throws Exception {
        try {
            throw new ArrayIndexOutOfBoundsException();
        } catch (IllegalArgumentException e) {
            System.out.println("Caught in f");
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("Before f");
            f();
            System.out.println("After f");
        } catch (Exception e) {
            System.out.println("Caught in main");
        }
    }
}
```

Compiles fine.

Output:
Before f
Caught in main

Exactly the same qn as Qn5 e

SUMMARY CONCEPTS

- Liskov Substitution Principle (LSP)
- Inheritance
- Exception
- Checked Exception vs Unchecked
- throw vs throws

Quick note about Exception

```
void method() throws Exception {  
    throw new Exception();  
}  
//Compile ok
```

throw: Explicitly throwing new object
syntax: with bracket ()
throws: put after method before
braces, syntax: no bracket ()

```
void method() throws Exception {  
}  
//Compile ok
```

Valid. **throws** means the method **may** throw the exception. It does not have to throw. In this case, it doesn't throw any exception, and it is valid.

```
void method() throws FileNotFoundException {  
    // some code  
    throw new FileNotFoundException();  
}  
//Compile ok
```

Checked exception, if you did not use try-catch, you must explicitly state **throws**. For unchecked exception, you don't need to state **throws**, but you can if you want.

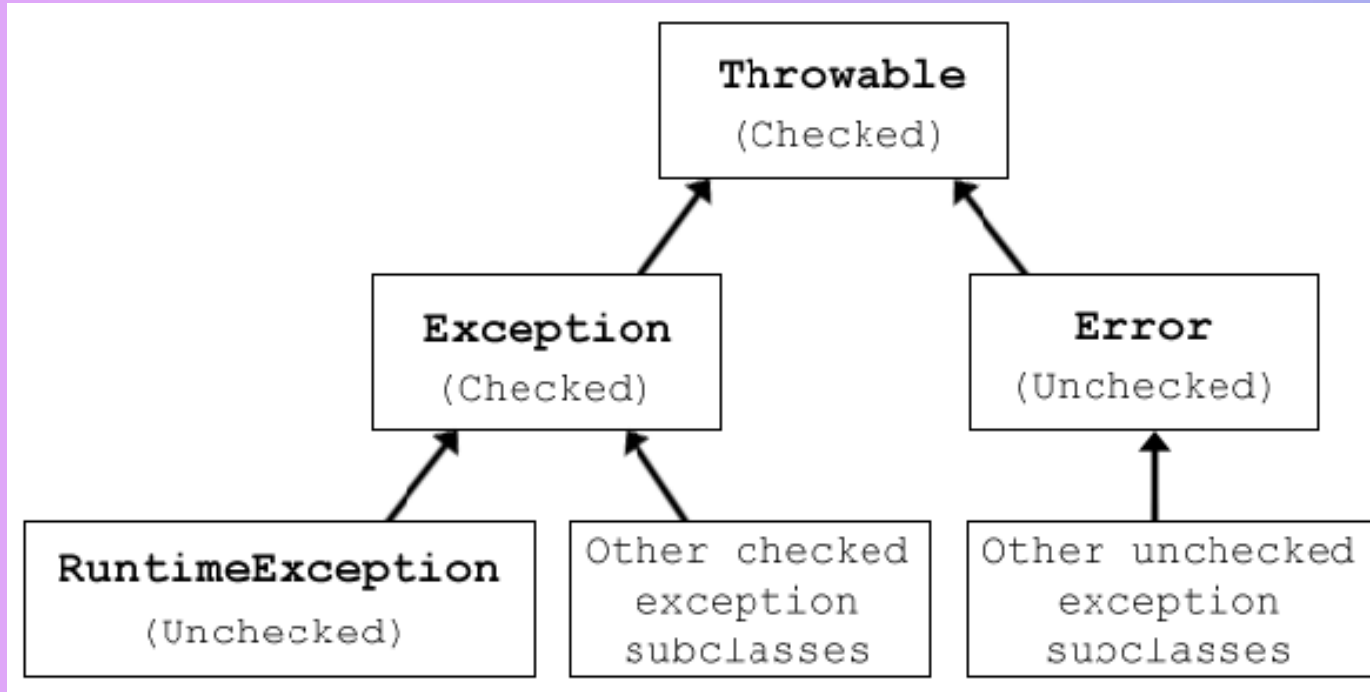
```
void method() {  
    // some code  
    throw new FileNotFoundException();  
}  
//Compile error for checked exception
```

```
void method() {  
    // some code  
    throw new NullPointerException();  
}  
//Compile ok for unchecked exception
```



```
void method() {  
    // some code  
    try {  
        throw new FileNotFoundException();  
    } catch (FileNotFoundException e) {}  
} //Compile ok
```

Valid even without **throws**. Because we use try-catch to handle the exception already, hence there is no need to indicate the method may throw an exception



Examples of checked exception:

NoSuchFieldException
ClassNotFoundException
NoSuchMethodException
FileNotFoundException

Examples of unchecked exception:

IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
ArithmeticException
NullPointerException

Some advice for Practical Lab

Note: All TAs did not see the actual question, hence take this with a pinch of salt

Tips which might be useful for you:

- 1) There is Java API on the desktop which you can access anytime during the practical assessment.
- 2) The practical assessment is a brand new question not related to any previous lab questions.
- 3) Make sure you know basic Vim commands
- 4) Make sure you know basic Unix command and able to navigate in the ssh environment
- 5) Make sure you save your work before the 90mins mark. Your computer will auto shut down at the 90mins mark.
- 6) Make sure your code can compile. If your code has compilation error, it is as good as not submitting a work.
- 7) Make sure you are familiar with the Scanner class (e.g. how to read an int or a line)
- 8) Make sure you are familiar with common import statements (e.g. `import java.util.ArrayList;`)
- 9) Make sure you are comfortable to code using Java without internet access
- 10) Make sure you revise common java concepts such as inheritance, polymorphism, access modifier, getter / setter, constructor, methods, fields, abstract, interface, static, final etc.
- 11) Make sure your output format is exactly the same as specified. Any additional spaces or lack of line break will render your output wrong.

Other Useful Commands (UNIX)

Change directory: `cd lab0`

Editing files: `vim Circle.java`

Compiling: `javac Circle.java` Compile ALL
or `javac *.java` ← files in that
directory

Running: `java Circle`

Printing: `System.out.println(...)`

Setup (Vim)

Insert mode: `i`

Command mode: `esc`

Navigation

Up, down, left, right: `k`, `j`, `h`, `l`

Jump to beginning to next word: `w`

Jump to beginning of previous word: `b`

Jump to end of line: `$`

Jump to start of line: `0`

Even More Useful Commands (UNIX)

Change directory: `cd lab0`

Go back: `cd ..`

Go back to home: `cd`

Print current working directory: `pwd`

Copy: `cp`

Move: `mv`

Remove: `rm`

Make directory: `mkdir`

Other useful vim commands:

Save in vim: `:w`

Quit vim: `:q`

Save and quit in vim: `:wq`

Quit without saving in vim: `:q!`

Move cursor fast in vim: `shift [` or `shift]`

Delete one entire line: `dd`

Undo: `u`

Auto indent: `gg=G`

How to compile a java file, how to run a java file. (`javac Main.java`) (`java Main`)

How to take in an input file when you run a java program (`java Main < test1.in`)

How to output your file into an external file to compare (`java Main < test1.in > myOutput.txt`)

How to compare 2 files (`diff filename1.txt filename2.txt`) No output means both the 2 files contain same lines. If there's output on the screen, those are the lines that are different in the 2 files.

Recommended:

Open 2 ssh windows. One for writing code, one to compile. Don't waste your time closing java file then compile, run, then open the file again. Just do it in 2 windows so you don't need to close your java file.

Alternatively, you can figure out how to do a split screen command to split your ssh window into multiple screens.

Save your java file as frequent as possible. The lab computers suck. You may be the unlucky one to get a computer that auto shuts down or hangs during the lab (happens last sem). Don't panic, inform your lab TA, note down the time you lost. Ask them to recover the file back for you and ask for time extension based on the amount of time you lost.

Last sem practical lab was a question on Food. These are the suggested java solutions uploaded by the prof. I put here for you to have a rough idea on the complexity and expectation for your practical.

Note that the question is definitely doable within 90minutes. All the best for your practical! Try to get to the highest level and complete the entire question.

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        Menu menu = new Menu();
        while(sc.next().equals("add")) {
            String type = sc.next();
            if (type.equals("Combo")) {
                menu.add(type, sc.nextInt(), sc.nextInt(), sc.nextInt());
            } else {
                menu.add(type, sc.next(), sc.nextInt());
            }
        }
        menu.print();

        Order order = new Order(menu);
        while (sc.hasNext()) {
            order.add(sc.nextInt());
        }
        System.out.println("--- Order ---");
        System.out.println(order);
    }
}
```

```
abstract class Item {
    protected static int numOfItems = 0;
    protected String desc;
    protected int price;
    protected int id;

    protected Item() {
        this.id = numOfItems++;
    }

    int getPrice() {
        return this.price;
    }

    @Override
    public String toString() {
        return "#" + id + " " + desc + " (" + getPrice() + ")";
    }
}
```

```
class Combo extends Item {
    private Burger burger;
    private Snack snack;
    private Drink drink;

    Combo(Burger burger, Snack snack, Drink drink) {
        this.burger = burger;
        this.snack = snack;
        this.drink = drink;
    }

    @Override
    int getPrice() {
        return burger.getPrice() + snack.getPrice() + drink.getPrice() - 50;
    }

    @Override
    public String toString() {
        return "#" + id + " Combo " + "(" + getPrice() + ")" +
            "\n    " + burger.toString() + "\n    " + snack.toString() + "\n    " + drink.toString();
    }
}
```

Burger.java

×

```
class Burger extends Item {  
    Burger(String desc, int price) {  
        this.desc = "Burger: " + desc;  
        this.price = price;  
    }  
}
```

Drink.java

×

```
class Drink extends Item {  
    Drink(String desc, int price) {  
        this.desc = "Drink: " + desc;  
        this.price = price;  
    }  
}
```

Snack.java

×

```
class Snack extends Item {  
    Snack(String desc, int price) {  
        this.desc = "Snack: " + desc;  
        this.price = price;  
    }  
}
```



```

Menu.java
import java.util.ArrayList;
import java.util.List;

public class Menu {
    private List<Item> items = new ArrayList<>();
    private List<Burger> burgers = new ArrayList<>();
    private List<Snack> snacks = new ArrayList<>();
    private List<Drink> drinks = new ArrayList<>();
    private List<Combo> combos = new ArrayList<>();

    public void add(String type, String desc, int price) {
        if (type.equals("Burger")) {
            Burger item = new Burger(desc, price);
            burgers.add(item);
            items.add(item);
        } else if (type.equals("Snack")) {
            Snack item = new Snack(desc, price);
            snacks.add(item);
            items.add(item);
        } else if (type.equals("Drink")) {
            Drink item = new Drink(desc, price);
            drinks.add(item);
            items.add(item);
        }
    }

    public void add(String type, int burgerId, int snackId, int drinkId) {
        try {
            Combo item = new Combo((Burger)items.get(burgerId), (Snack)items.get(snackId), (Drink)items.get(drinkId));
            combos.add(item);
            items.add(item);
        } catch (IndexOutOfBoundsException | ClassCastException e) {
            System.err.println("Error: Invalid combo input " +
                burgerId + " " + snackId + " " + drinkId);
        }
    }

    public Item get(int index) {
        return items.get(index);
    }

    private void print(List<? extends Item> items) {
        for (Item i : items) {
            System.out.println(i);
        }
    }

    public void print() {
        print(burgers);
        print(snacks);
        print(drinks);
        print(combos);
    }
}

```

```
import java.util.ArrayList;
import java.util.List;

class Order {
    private Menu menu;
    private List<Item> items;

    Order(Menu menu) {
        this.menu = menu;
        items = new ArrayList<>();
    }

    void add(int index) {
        items.add(menu.get(index));
    }

    @Override
    public String toString() {
        String s = "";
        int total = 0;
        for (Item item : items) {
            s = s + item.toString() + "\n";
            total += item.getPrice();
        }

        s = s + "Total: " + total;
        return s;
    }
}
```

QUESTIONS?