

CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

Hande Alemdar

Fall 2020 - Lecture 1

Introduction to Java – Part 1

History

- Java was started as a project called “Oak” by James Gosling in June 1991.
 - Sun Microsystems – a hardware company
- The goal was to develop software that was portable so that it could be switched quickly to new hardware
- The language was first called Oak (after an oak tree outside Goslings window). However, there was already a language named Oak.
- The team went for a “Coffee break” and named the language Java
 - in recognition of the role that caffeine plays in software development

The Goal

- The goal was to implement sth much simpler than C/C++
- Java was developed with the goal to implement “Write Once, Run Anywhere” programming model.



Philosophy

- The Java programming language was built on the following five philosophies.
 1. It will use the Object-oriented programming methodology
 2. The same program should be executable on multiple operating systems.
 3. Built-in support for using computer networks.
 4. Designed to execute code from the remote sources securely.
 5. It should be easy to use, take the good features of Object-oriented programming.

A BRIEF HISTORY OF



Java is an Object-Oriented Programming Language. It's more than **20 years old** and Java is running on **billions** of devices.

1995 - JDK BETA

The first beta version of Java. Developed by James Gosling at Sun Microsystems.

23 JAN 1996 - JAVA 1

First public release. The stable version Java 1.0.2 is called Java 1.

19 FEB 1997 - JAVA 1.1

Inner Classes, Java Beans, JDBC, RMI

8 DEC 1998 - JAVA 1.2

Swing, JIT Compiler, Collections

8 MAY 2000 - JAVA 1.3

HotSpot JVM, JNDI, JPDA

6 FEB 2002 - JAVA 1.4

Assertions, RegEx Improvements, Image IO API, XML Parsers, XSLT Processors, Preferences API

30 SEP 2004 - JAVA 5

Generics API, Varargs, for-each loop, Autoboxing, Enum, Annotations, Static Imports

11 DEC 2006 - JAVA 6

JAXB 2, JDBC 4.0 support, Pluggable annotations

7 JUL 2011 - JAVA 7

String in Switch Statements, Try with Resource, Java NIO Package, Catching Multiple Exceptions in a single catch block

18 MAR 2014 - JAVA 8

18 MAR 2014 - JAVA 8

forEach() Method, default and static method in interfaces, Functional interfaces and Lambda expressions, Stream API, New Date Time API

21 SEP 2017 - JAVA 9

JShell, Module System under Project Jigsaw, Reactive Streams, HTTP 2 Client

20 MAR 2018 - JAVA 10

Local-Variable Type Inference

25 SEP 2018 - JAVA 11

Running Java program from single command, New String Class methods, var for lambda expressions

19 MAR 2019 - JAVA 12

Shenandoah Garbage Collector, Teeing Collectors, New methods in String class, Switch Expressions

17 SEP 2019 - JAVA 13

Text Blocks, Switch Expressions, Socket API reimplementation, Unicode 12.1 support, DOM and SAX Factories with Namespace Support



© <https://www.journaldev.com>

String in Switch Statements, Try with Resource, Java NIO Package, Catching Multiple Exceptions in a single catch block

What is Java?

- According to Sun Microsystems White Paper
- Java is simple, object-oriented, network-savvy, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic language

Java is Simple

- Java is partially modeled on C++ but greatly simplified and improved
- Pointers & multiple inheritance often make programming complicated
- Java replaces the multiple inheritance in C++ with a simple language construct
- Java eliminates pointers
- Java uses automatic memory allocation and garbage collection

Java is Object-Oriented

- It is an Object-Oriented programming language
- It has constructs to implement **encapsulation, polymorphism and inheritance**

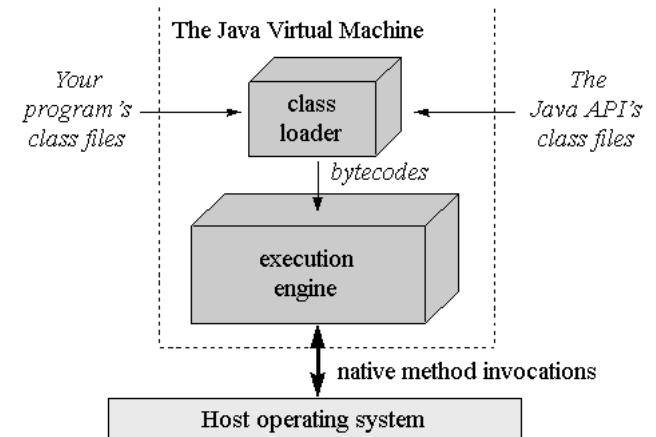
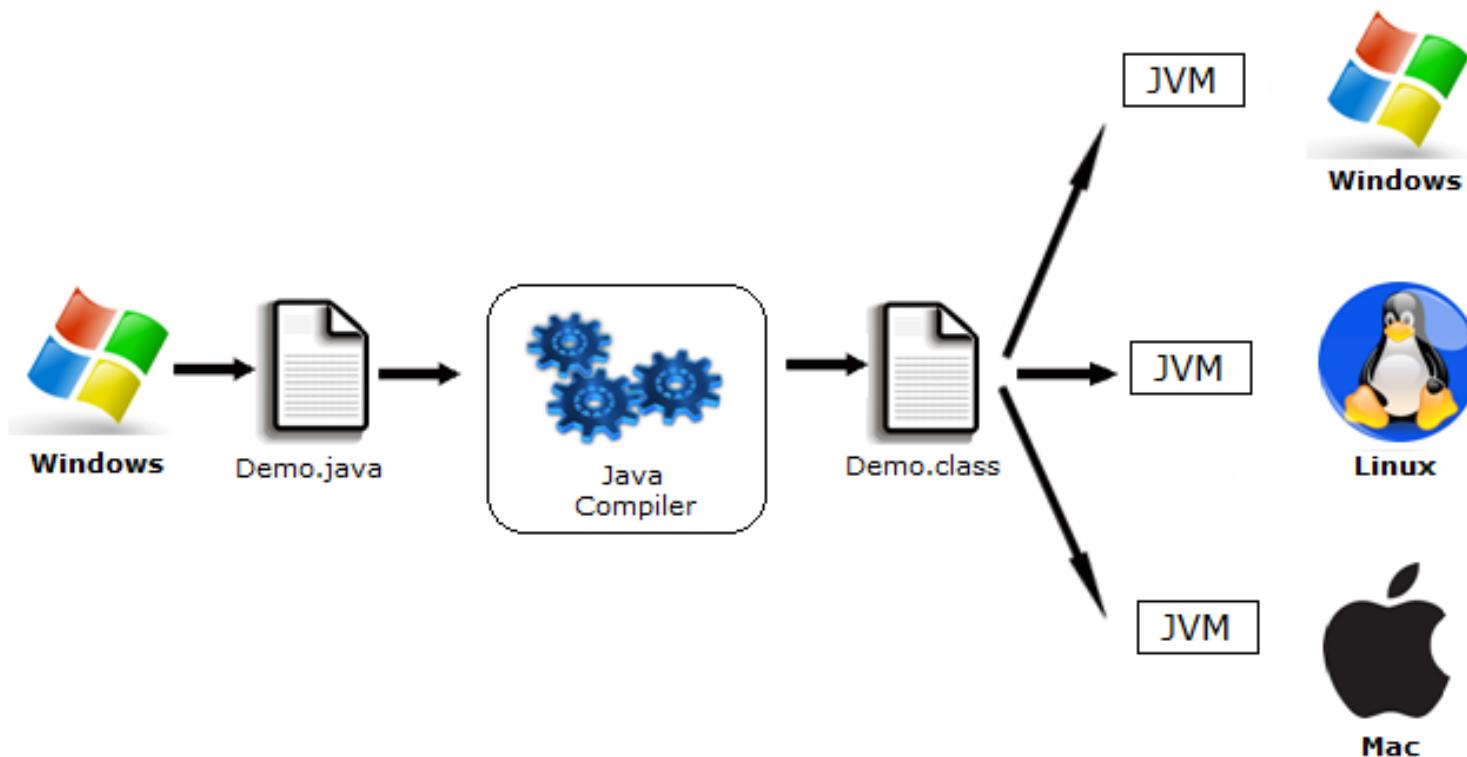
Java is Distributed

- Distributed computing involves several computers working together on a network.
- Networking capability is inherently integrated into Java so writing network programs is like sending and receiving data to and from a file.

Java is Compiled and Interpreted

- The Java platform has a compiler that translates Java source into a form called bytecodes
- Bytecode is an architecturally neutral representation of code written in the Java programming language.
- Bytecode is machine-independent and can run on any machine that has a Java interpreter.
- The bytecode rather than Java source code, is interpreted when you run a Java program.

Java Execution Model



Java is Robust

- Robust means reliable.
- Java has eliminated certain error-prone programming constructs found in other languages.
- It doesn't support pointer arithmetic, for example, thereby eliminating the possibility of overwriting memory and corrupting data.
- Java has a runtime exception-handling feature to provide programming support for robustness.
- The programmer must write the code to deal with exceptions.

Java is Secure

- Java security is based on the premise that nothing should be trusted.
- If you download a Java applet and run it on your computer, it will not damage your system because Java implements several security mechanisms.
- We don't have pointers and we cannot access out of bound arrays (you get `ArrayIndexOutOfBoundsException` if you try to do so) in java. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Java is Architecture neutral and Portable

- All Java programs must be compiled into bytecodes before the JVM can run them.
- Java programs can be run on any platform without being recompiled, making them very portable.
- There are no platform-specific features in Java (size of an int).

Java is Multithreaded

- Multithreading is the capability for a program to perform several tasks simultaneously within a program
- Used in graphical user interfaces
 - listen to an audio recording while surfing a Web page
- Used in network programming
 - a server can serve multiple clients at the same time
- Classes are provided from the base language package to create and manage threads

Java is Dynamic

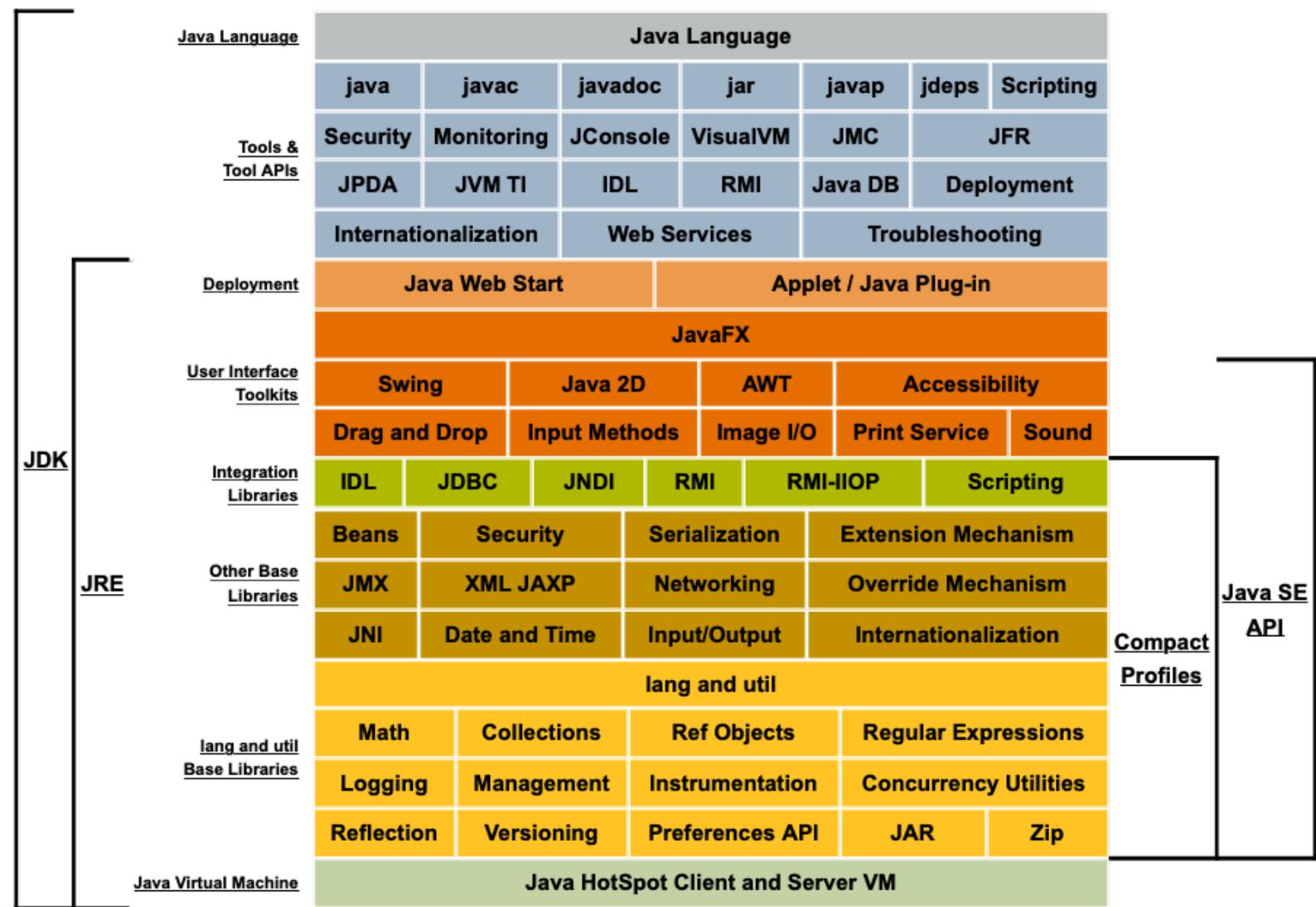
- Java was designed to adapt to an evolving environment.
- You can **freely** add new methods to a class without affecting its clients
 - Only if you follow the rules!
- Example: In the Circle class, you can add a new data property to indicate the color of the circle or a new method to obtain the circumference of the circle. The original client program that uses the circle class remains the same.
- Also, at runtime Java loads classes as they are needed.

Java's Performance

- Java was historically considered slower than the fastest 3rd generation typed languages such as C and C++
- The main reason was using Java virtual machine (JVM) rather than directly using the computer's processor as native code, as do C and C++ programs
- Since the late 1990s, the execution speed of Java programs improved significantly via introduction of just-in-time compilation (JIT) (in 1997 for Java 1.1)
- The addition of language features supporting better code analysis, and optimizations in the JVM (such as HotSpot becoming the default for Sun's JVM in 2000)
- HotSpots are parts of the bytecode executed frequently
- Hardware execution of Java bytecode, such as that offered by ARM's Jazelle, was also explored to offer significant performance improvements.

What Java is TODAY

- It has become more than just another programming language
- Set of Technologies and tools with huge community
- An open standard



CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

Hande Alemdar

Fall 2020 - Lecture 1

Introduction to Java – Part 2

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- The simplest Java Program
- If you want to run a Java program you should have method called **main**.
- It is the identifier that the JVM looks for as the starting point of the java program. **It's not a keyword**.

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- you must use a class even if you aren't doing OO programming

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- main must be public
- It is an *Access modifier*, which specifies from where and who can access the method. Making the *main()* method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- main must be static
- It is a *keyword* which is when associated with a method, makes it a class related method. The *main()* method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM.

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- main must return void
- As soon as the *main()* method terminates, the java program terminates too. Hence, it doesn't make any sense to return from *main()* method as JVM can't do anything with the return value of it.

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- main must declare command line arguments even if it doesn't use them

Hello World

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

- `println` uses the static field `System.out`

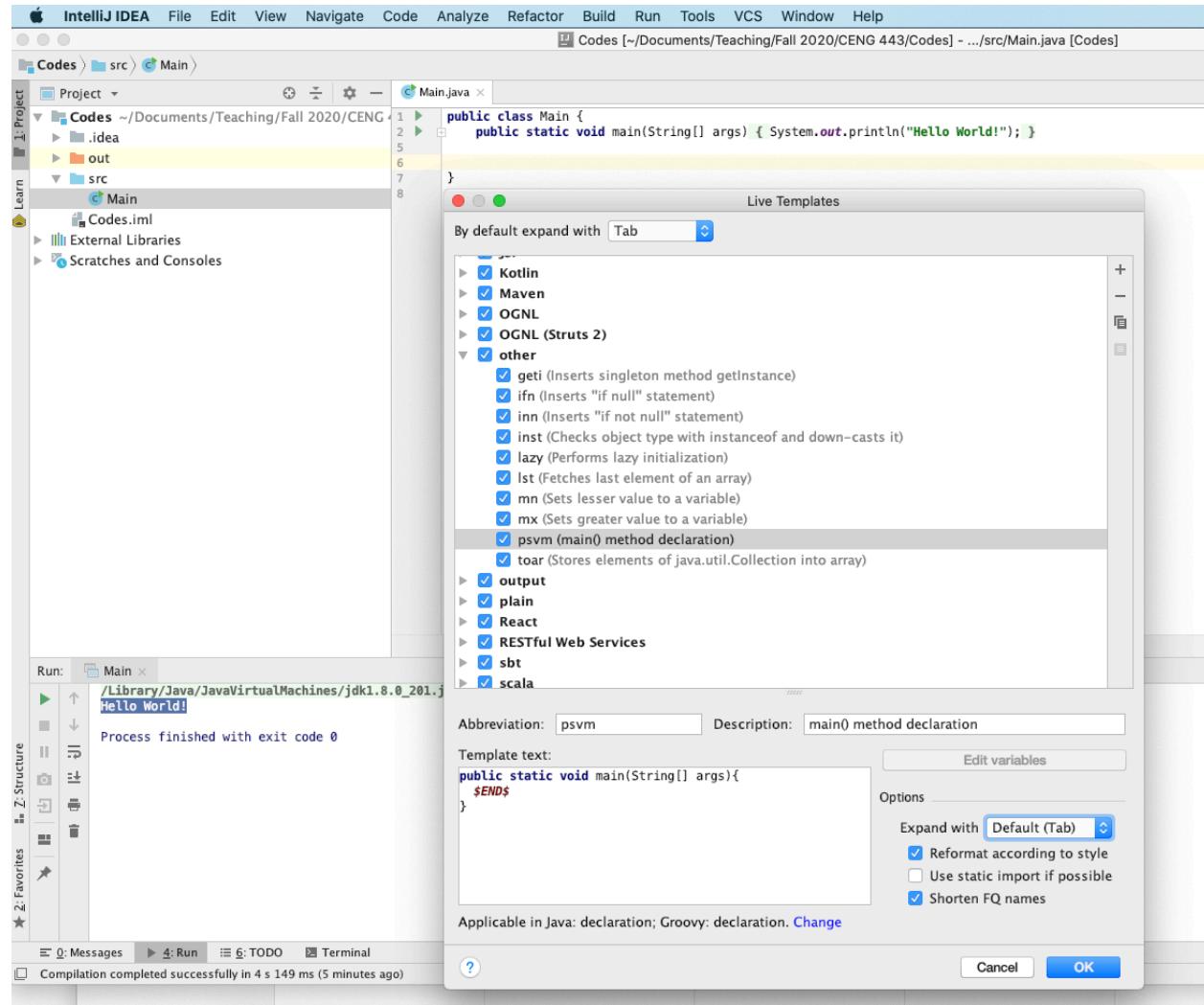
Execution is a bit complicated, too

- First you **compile** the source file
 - javac HelloWorld.java
 - Produces class file HelloWorld.class
- Then you **launch** the program
 - java HelloWorld
 - Java Virtual Machine (JVM) executes main method

On the bright side...

- Has many good points to balance shortcomings
- Some verbosity is not a bad thing
 - Can reduce errors and increase readability
- Modern IDEs eliminate much of the pain
 - Type **psvm** instead of public static void main
- Managed runtime (JVM) has many advantages
 - Safe, flexible, enables garbage collection
- It may not be best language for Hello World...
 - But Java is very good for large-scale programming!

Pick an IDE and use it to the fullest



Java Type System

- Java has a *bipartite* (2-part) type system

Primitive Types

int long short
char boolean byte
float double

Object Reference Types

Classes, interfaces, arrays,
enums, annotations, strings,
exceptions

- No identity except their value
- Immutable
- On stack, exist only when in use
- Can't achieve unity of expression
- Dirt cheap

- Have identity distinct from value
- Some mutable, some immutable
- On heap, garbage collected
- Unity of expression with generics
- More costly

Java Primitive Types

- int 32-bit signed integer
- long 64-bit signed integer
- byte 8-bit signed integer
- short 16-bit signed integer
- char 16-bit unsigned integer/character
- float 32-bit IEEE 754 floating point number
- double 64-bit IEEE 754 floating point number
- boolean Boolean value: true or false

“Deficient” primitive types

- byte, short – typically use int instead!
 - byte is broken – should have been unsigned
- float – typically use double instead!
 - Provides too little precision
 - Few compelling use cases, e.g., large arrays in resource-constrained environments

Objects

- All non-primitives are represented by objects.
- An **object** is a bundle of state and behavior
- State – the data contained in the object
 - In Java, these are called its instance **fields**
- Behavior – the actions supported by the object
 - In Java, these are called its **methods**
 - Method is just OO-speak for function
 - “Invoke a method” is OO-speak for “call a function”

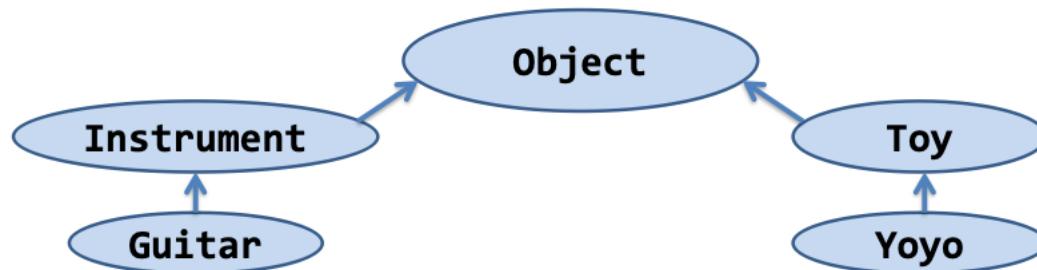
Classes

- Every object has a class
 - A class defines methods and fields
 - Methods and fields collectively known as **members**
- Class defines both type and implementation
 - Type ≈ **what** object does (hence where it can be used)
 - Implementation ≈ **how** the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
 - Defines how users interact with its instances

Java Class Hierarchy

- The root is Object (all non-primitives are objects)
- All classes except Object have one parent class
 - Specified with an **extends** clause

```
class Guitar extends Instrument { ... }
```
 - If extends clause omitted, defaults to Object
- A class is an instance of all its superclasses

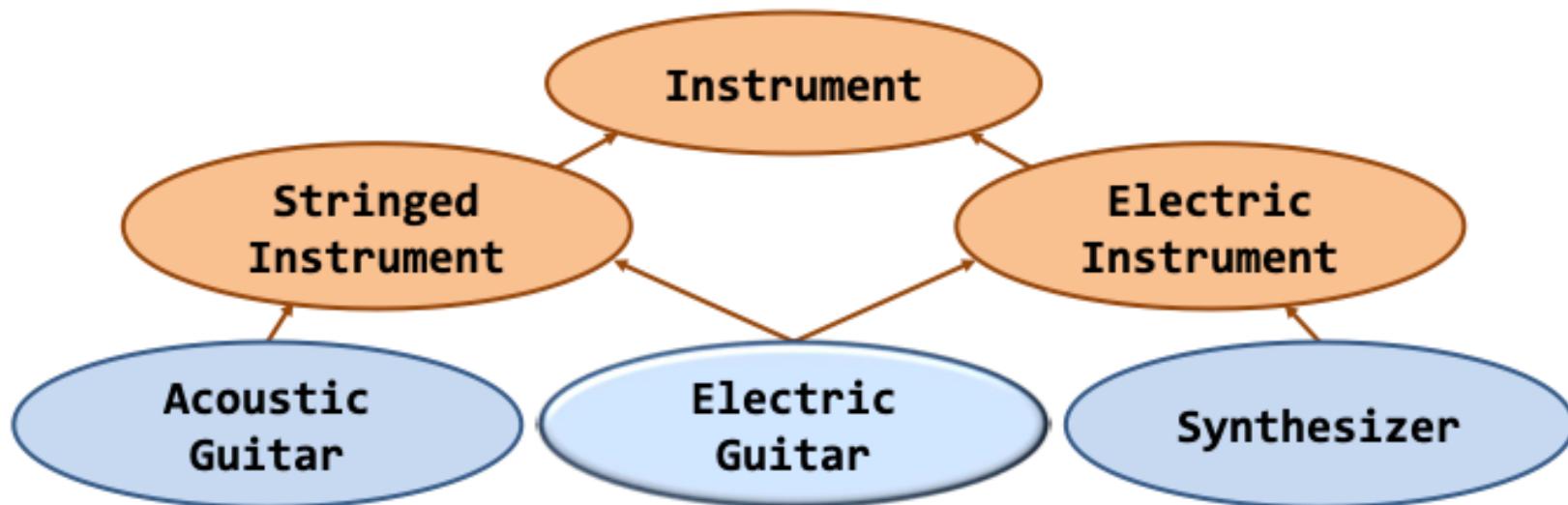


Implementation inheritance

- A class:
 - Inherits visible fields and methods from its superclasses
 - Can *override* methods to change their behavior
- Overriding method implementation must obey the contract(s) of its superclass(es)
 - Ensures subclass can be used anywhere superclass can
 - *Liskov Substitution Principle* (LSP)
 - We will talk more about this in a later class

Interface types

- Defines a type without an implementation
- Much more flexible than class types
 - An interface can extend one or more others
 - A class can implement multiple interfaces



Enum types

- Java has object-oriented enums
- In simple form, they look just like C enums

```
enum Planet { MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE }

Planet location = ...;
if (location.equals(Planet.EARTH)) {
    System.out.println("Honey, I'm home!"); }
```

- But they have **many** advantages!
 - Compile-time type safety
 - Multiple enum types can share value names
 - Can add or reorder without breaking existing uses
 - High-quality Object methods are provided
 - Screaming fast collections (EnumSet, EnumMap)
 - Can iterate over all constants of an enum

You can add data to enums

```
public enum Planet {  
    MERCURY(3.302e+23, 2.439e6), VENUS (4.869e+24, 6.052e6),  
    EARTH(5.975e+24, 6.378e6), MARS(6.419e+23, 3.393e6);  
  
    private final double mass;    // In kg.  
    private final double radius;  // In m.  
    private static final double G = 6.67300e-11; // N m2/kg2  
  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
    public double mass() { return mass; }  
    public double radius() { return radius; }  
    public double surfaceGravity() { return G * mass / (radius * radius); }  
}
```

You can add behavior too

```
public enum Planet {  
    . . . // As on previous slide  
  
    public double surfaceWeight(double mass) {  
        return mass * surfaceGravity; // F = ma  
    }  
}
```

Example

```
public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight / EARTH.surfaceGravity();

    for (Planet p : Planet.values()) {
        System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
    }
}

$ java WeightOnPlanet 180
Your weight on MERCURY is 68.023205
Your weight on VENUS is 162.909181
Your weight on EARTH is 180.000000
Your weight on MARS is 68.328719
```

You can even add value-specific behavior

```
public enum Operation {  
    PLUS ("+", (x,y)->x+y),  
    MINUS ("-", (x, y) -> x - y),  
    TIMES ("*", (x, y) -> x * y),  
    DIVIDE("/", (x, y) -> x / y);  
  
    private final String symbol;  
    private final DoubleBinaryOperator op;  
    Operation(String symbol, DoubleBinaryOperator op) {  
        this.symbol = symbol;  
        this.op = op;  
    }  
  
    @Override  
    public String toString() { return symbol; }  
  
    public double apply(double x, double y) {  
        return op.applyAsDouble(x, y);  
    }  
}
```

Example

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);

    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));
}

$ java TestOperation 4 2
4.000000 + 2.000000 = 6.000000
4.000000 - 2.000000 = 2.000000
4.000000 * 2.000000 = 8.000000
4.000000 / 2.000000 = 2.000000
```

Enums are your friend

- Use them whenever you have a type with a fixed number of values known at compile time

Boxed primitives

- Immutable containers for primitive types
- Boolean, Integer, Short, Long, Character, Float, Double
- Let you “use” primitives in contexts requiring objects
- Language does *autoboxing* and *auto-unboxing*
- **Don’t use boxed primitives unless you have to!**
- **Use Boxed primitives when**
 1. Using parameterized types (list Collection). Parameterized types do not permit primitives.
 2. Using value as a key or value in Collections, e.g. HashSet<Integer>
 3. Using reflective method invocation (another don't do). e.g. class.forName("java.lang.Integer");

```
public class Test
{
    Integer i; // Initialized to null
    public static void main(String[] args)
    {
        if ( i == 45 )      // 1. auto-unbox i (convert Integer to int)
                            // 2. NullPointerException as i value is null.
        {
            System.out.println( "i is 45." );
        }
    }
}
```

Comparing values

- `x == y` compares x and y “directly”:
- **primitive values:** returns true if x and y **have the same value**
- **objects refs:** returns true if x and y **refer to same object**
- `x.equals(y)` compares the ***values of the objects referred to by x and y***

True or False?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);
```

The moral of the example

- **Always use .equals to compare object refs!**
 - (Except for enums, which are special)
- The == operator can fail silently and unpredictably when applied to object references
- Same goes for !=

Output

- Unformatted
 - `System.out.println("Hello World");`
 - `System.out.println("Radius: " + r);`
 - `System.out.println(r * Math.cos(theta));`
 - `System.out.println();`
 - `System.out.print("*");`
- Formatted – very similar to C
`System.out.printf("%d * %d = %d%n", a, b, a * b); // Varargs`

Command line input example

- *Echos all its command line arguments*

```
class Echo {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.print(arg + " ");  
        }  
    }  
}
```

```
$ java Echo Woke up this morning, had them weary blues  
Woke up this morning, had them weary blues
```

Command line input with parsing

- Prints the GCD of its two command line arguments

```
class Gcd {  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        int j = Integer.parseInt(args[1]);  
        System.out.println(gcd(i, j));  
    }  
  
    static int gcd(int i, int j) {  
        return i == 0 ? j : gcd(j % i, i);  
    }  
}
```

```
$ java Gcd 11322 35298  
666
```

Scanner input

- *Counts the words on standard input*

```
class Wc {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        long result = 0;  
  
        while (sc.hasNext()) {  
            sc.next(); // Swallow token  
            result++;  
        }  
        System.out.println(result); }  
}
```

```
$ java Wc < Wc.java
```

CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

Hande Alemdar

Fall 2020 - Lecture 1

Introduction to Java – Part 2

Answers from last session

- Operator overloading is not permitted in Java
 - Implemented in compiler
- You can have multiple main methods in a class
 - But with different **signatures**
 - method's name + parameter types

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
        main();  
        main( 3 );  
        System.out.println( "main " + main( s: "s" ));  
    }  
  
    public static void main() {  
        System.out.println("main 2");  
    }  
  
    public static void main(int i) {  
        System.out.println("main 3");  
    }  
  
    public static int main(String s) {  
        return 4;  
    }  
}
```

Enum patterns

```
public static final int APPLE_FUJI      = 0;  
public static final int APPLE_PIPPIN    = 1;  
public static final int APPLE_GRANNY_SMITH = 2;  
  
public static final int ORANGE_NAVEL   = 0;  
public static final int ORANGE_TEMPLE  = 1;  
public static final int ORANGE_BLOOD   = 2;
```

- This technique, known as the *int enum pattern*, has many shortcomings.
- It provides nothing in the way of type safety and little in the way of expressive power.
- The compiler won't complain if you pass an apple to a method that expects an orange, compare apples to oranges with the == operator

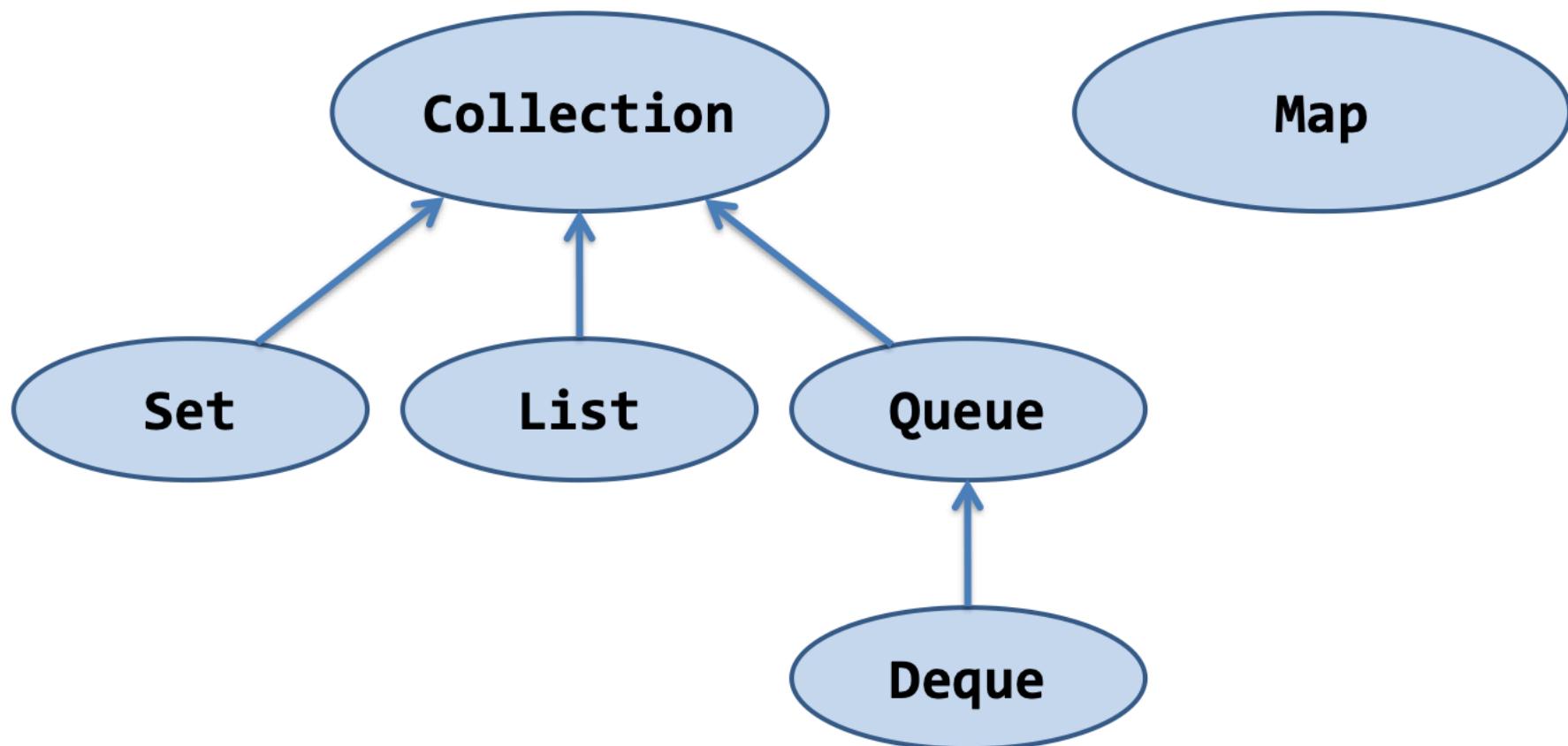
```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }  
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

Java Collections

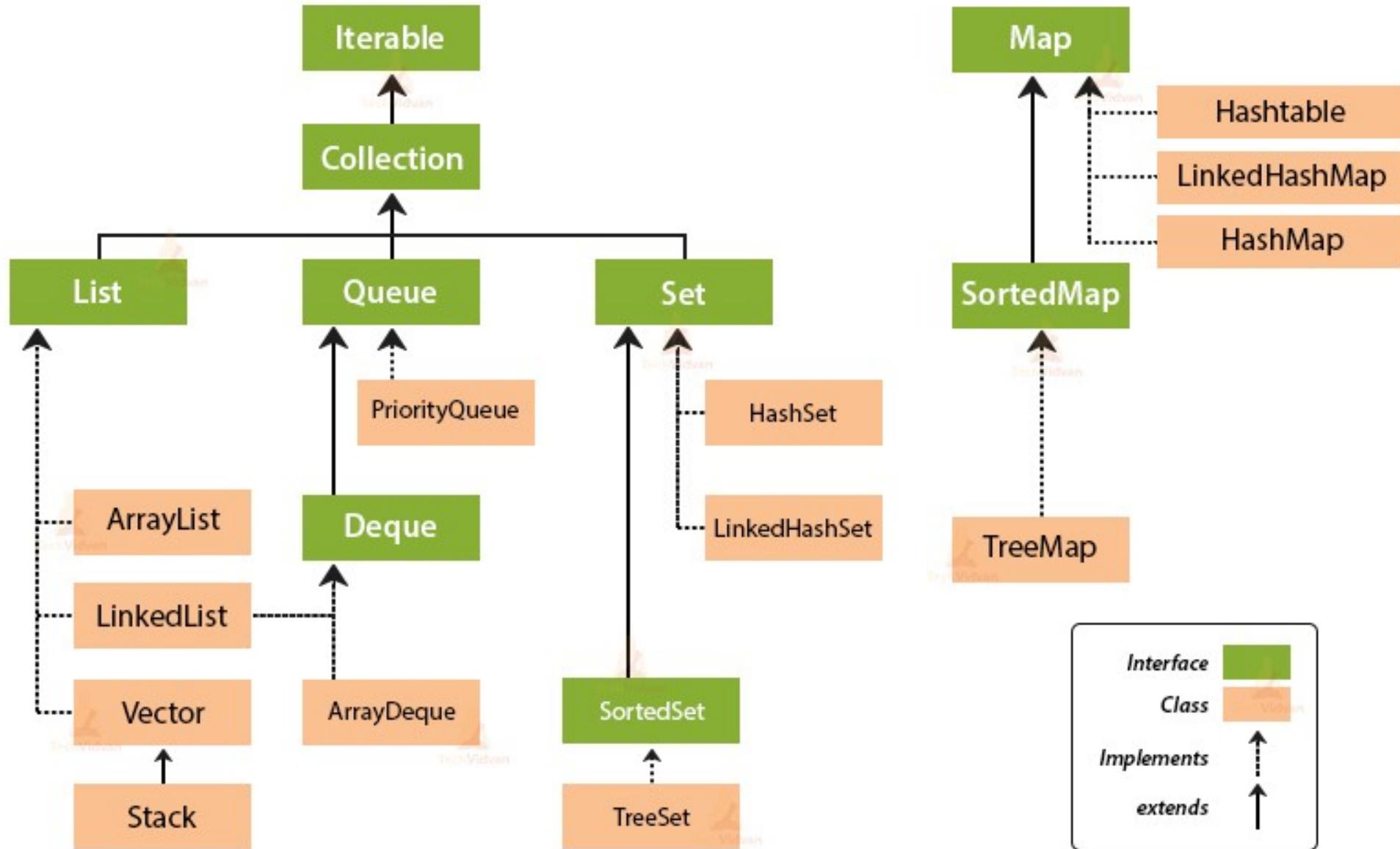
- A collection is an object that represents a group of objects
- Java Collections Framework
 - Interfaces for common abstract data structures
 - Classes that implement those data structures
 - Includes **algorithms** (e.g. searching, sorting)
 - Algorithms are *polymorphic*: can be used on many different implementations of collection interfaces.

Java Collections

- Primary collection interfaces



Collection Framework Hierarchy in Java



Collections usage example 1

- *Squeezes duplicate words out of command line*

```
public class Squeeze {  
    public static void main(String[] args) {  
        Set<String> s = new LinkedHashSet<>();  
        for (String word : args)  
            s.add(word);  
        System.out.println(s);  
    }  
}
```

```
$ java Squeeze I came I saw I conquered  
[I, came, saw, conquered]
```

Collections usage example 2

- *Prints unique words in alphabetical order*

```
public class Lexicon {  
    public static void main(String[] args) {  
        Set<String> s = new TreeSet<>();  
        for (String word : args)  
            s.add(word);  
        System.out.println(s);  
    }  
}
```

```
$ java Lexicon I came I saw I conquered  
[I, came, conquered, saw]
```

Collections usage example 3

- *Prints the index of the first occurrence of each word*

```
class Index {  
    public static void main(String[] args) {  
        Map<String, Integer> index = new TreeMap<>();  
  
        // Iterate backwards so first occurrence wins  
        for (int i = args.length - 1; i >= 0; i--) {  
            index.put(args[i], i);  
        }  
        System.out.println(index);  
    }  
}
```

```
$ java Index if it is to be it is up to me to do it  
{be=4, do=11, if=0, is=2, it=1, me=9, to=3, up=7}
```

What about arrays?

- Arrays aren't a part of the collections framework
- But there is an adapter: `Arrays.asList`
- Arrays and collections don't mix well
- If you try to mix them and get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
- But arrays of primitives (e.g., `int[]`) are preferable to lists of boxed primitives (e.g., `List<Integer>`)

Java Arrays

- Conceptually represented as an object
 - Provides .length, runtime bounds checking

```
String[] answers = new String[42];
if (answers.length == 42) {
    answers[42] = "no"; // ArrayIndexOutOfBoundsException
}
```

Methods common to all objects

- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on Object
 - **equals** - returns true if the two objects are “equal”
 - **hashCode** - returns an int that must be equal for equal objects, and is likely to differ on unequal objects
 - **toString** - returns a printable string representation

Object Implementations

- Provide **identity semantics**
 - `equals(Object o)` – returns true if `o` refers to this object
 - `hashCode()` – returns a near-random int that never changes over the object lifetime
 - `toString()` – returns a nasty looking string consisting of the type and hash code
 - `java.lang.Object@659e0bfd`

Overriding Object implementations

- No need to override equals and hashCode if you want identity semantics
 - When in doubt do not override
 - Identity semantics are often what you want
 - It is easy to get it wrong
- Nearly always override `toString`
 - `println` invokes it automatically
 - Why settle for ugly?

Overriding `toString` is easy and beneficial

```
final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
    ...  
    @Override public String toString() {  
        return String.format("(%03d) %03d-%04d",  
            areaCode, prefix, lineNumber);  
    }  
}
```

```
Number jenny = ...;  
System.out.println(jenny);  
Prints: (707) 867-5309
```

Java Annotations

- Annotations mark code without any immediate functional effect
- @Override, @Deprecated, @SuppressWarnings

```
class Bicycle {  
    ...  
    @Override  
    public String toString() {  
        return ...;  
    }  
}
```

Summary

- Java is well suited to large programs; small ones may seem a bit verbose
- Bipartite type system – primitives & object refs
- Single implementation inheritance
- Multiple interface inheritance
- Easiest output – `println`, `printf`
- Easiest input – Command line args, `Scanner`
- Collections framework is powerful & easy to use

CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

Hande Alemdar

Fall 2020 – Lecture 2

Introduction to OOP – Part 1

Object

- An **object** is a bundle of state and behavior
- State – the data contained in the object
 - In Java, these are called its instance **fields**
- Behavior – the actions supported by the object
 - In Java, these are called its **methods**
 - Method is just OO-speak for function
 - “Invoke a method” is OO-speak for “call a function”

Example

- Object : Car
- State
 - Color, brand, weight, model
- Behavior
 - Break, accelerate, slow down, gear change

```
public class CarUser {  
    public static void main(String[] args) {  
        Car c = new Car();  
    }  
}
```

c is the object

This is a Class

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

What is a Class ?



- A class can be considered as a outline of the object
- You can **construct** different objects using that outline
- Constructor is a *special* method
- It's name is same as class name and it does not return any value

```
public class CarUser {  
    public static void main(String[] args) {  
        Car c = new Car();  
    }  
}
```

Default Constructor

What is the output?

```
public class CarUser {  
    public static void main(String[] args) {  
        Car c = new Car();  
        System.out.println(c.weight);  
    }  
}
```

- a. 0
- b.
- c. Runtime Error
- d. Compile Error

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

What is the output?

```
public class CarUser {  
    public static void main(String[] args) {  
        Car c = new Car();  
        System.out.println(c.color);  
    }  
}
```

- a.
- b. null
- c. Runtime Error
- d. Compile Error

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

Default Constructor

- When we do not explicitly define a **constructor** for a class, **java compiler** creates a **default constructor** for the class.
- It is a non-parameterized **constructor**.
- The **default constructor**'s job is to call the super class **constructor** and initialize all instance fields.

Parametrized Constructors

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(String color, String brand, int weight){  
        this.color = color;  
        this.brand = brand;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(String color, int weight){  
        this.color = color;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

this keyword is used to refer to the object instance

What is the output?

```
public class CarUser {  
    public static void main(String[] args) {  
        Car c = new Car();  
        System.out.println(c.weight);  
    }  
}
```

- a. 0
- b.
- c. Runtime Error
- d. Compile Error

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(String color, String brand, int weight){  
        this.color = color;  
        this.brand = brand;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

Multiple constructors

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(String color, int weight){  
        this.color = color;  
        this.weight= weight;  
    }  
  
    Car(String color, String brand,  
        this.color = color;  
        this.brand = brand;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(){  
    }  
  
    Car(String color, int weight){  
        this.color = color;  
        this.weight= weight;  
    }  
  
    Car(String color, String brand, int w  
        this.color = color;  
        this.brand = brand;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
  
    int accelerate(int howMuch){  
        // try to do it  
        return howMuch;  
    }  
}
```

```
public class Car {  
    String color;  
    String brand;  
    int weight;  
  
    Car(){  
        this.color = "White";  
    }  
  
    Car(String color, int weight){  
        this.color = color;  
        this.weight= weight;  
    }  
  
    Car(String color, String brand, int weight){  
        this.color = color;  
        this.brand = brand;  
        this.weight= weight;  
    }  
  
    void doBreak(){  
        // code to break  
    }  
}
```

Constructors

- 3 Types
 - Default constructor : the body of the constructor is empty.
 - no-arg constructor : the body of the constructor is NOT empty.
 - Parameterized constructor
- Constructors are not methods and they don't have any return type.
- Constructor name should match with class name
- If you don't implement any constructor within the class, compiler will do it for you
- A constructor can also invoke another constructor of the same class – By using this()

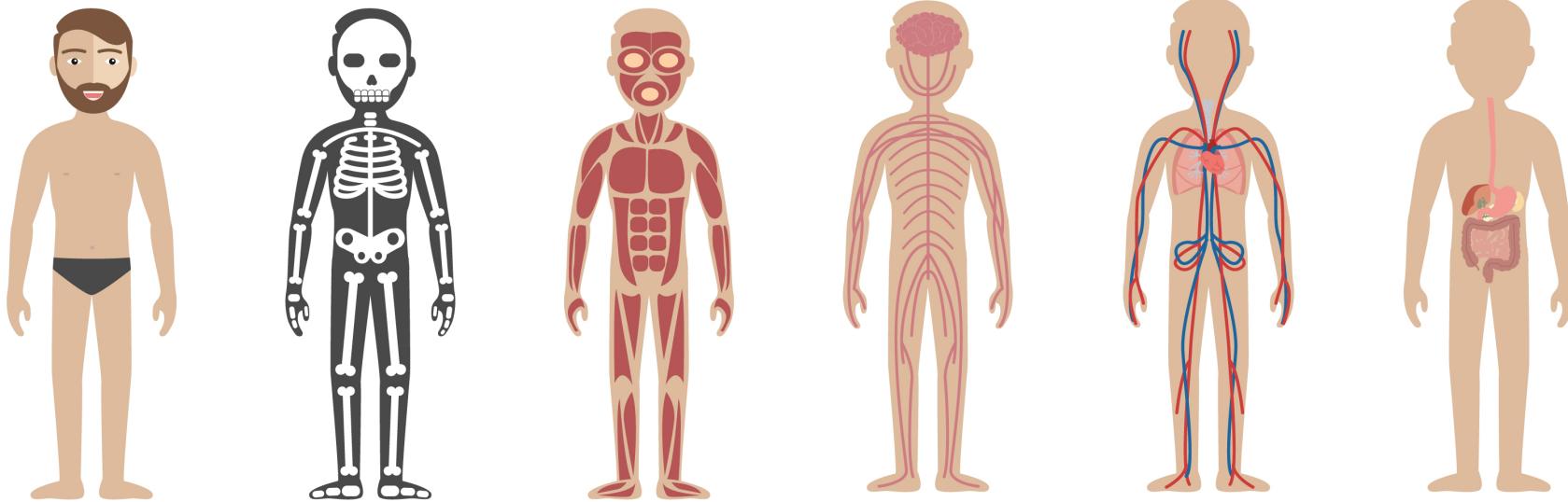
More on constructors later

Object Oriented Programming Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- Definition
 - the process of removing physical, spatial, or temporal details in the study of objects or systems to focus attention on details of greater importance it is similar in nature to the process of generalization



Please introduce yourself to me

Abstraction

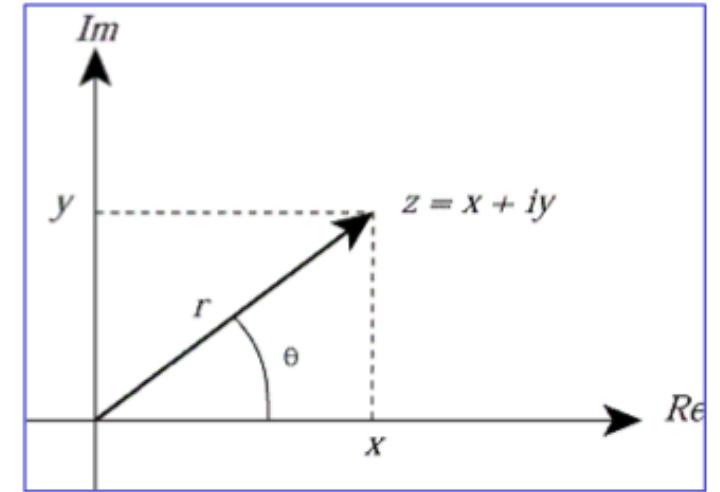
- We make use of abstraction constantly in our day to day lives
 - Hi, I am Hande, vs.
 - Hi, I'm a member of the mammalian species *Homo sapien*, a group of ground-dwelling, primates that are characterized by bipedalism and the capacity for speech and language, with an erect body carriage that frees the hands for manipulating objects!
- **“the quality of dealing with ideas rather than events”**
- Abstraction is about ***maximizing relevant information*** by omitting the unnecessary ones
- A way to handle the complexity
- Finding the correct abstraction for the concepts is extremely important

Abstraction

- Java Collections Framework is full of abstractions
 - Set is an abstraction of the mathematical set concept
 - List
 - Map
- In java, abstraction is achieved via interfaces and abstract classes
- The real abstraction is achieved via good modelling

Class example – complex numbers

```
public class Complex {  
  
    final double re; // Real part  
    final double im; // Imaginary part  
  
    public Complex(double re, double im) {  
        this.re = re;  
        this.im = im;  
    }  
  
    public double realPart() { return re; }  
    public double imaginaryPart() { return im; }  
    public double r() { return Math.sqrt(re * re + im * im); }  
  
    public double theta() { return Math.atan(im / re); }  
  
    public Complex add(Complex c) {  
        return new Complex(re + c.re, im + c.im);  
    }  
  
    public Complex multiply(Complex c) {  
        return new Complex( re * c.re - im * c.im , re * c.im + im * c.re);  
    }  
}
```



Complex Class usage example

```
public class ComplexUser {  
    public static void main(String args[]) {  
        Complex c = new Complex(-1, 0);  
        Complex d = new Complex(0, 1);  
  
        Complex e = c.add(d);  
        System.out.println("Sum:" + e.realPart() + "+" + e.imaginaryPart());  
  
        e = c.multiply(d);  
        System.out.println("Product:" + e.realPart() + "+" + e.imaginaryPart());  
    }  
}
```

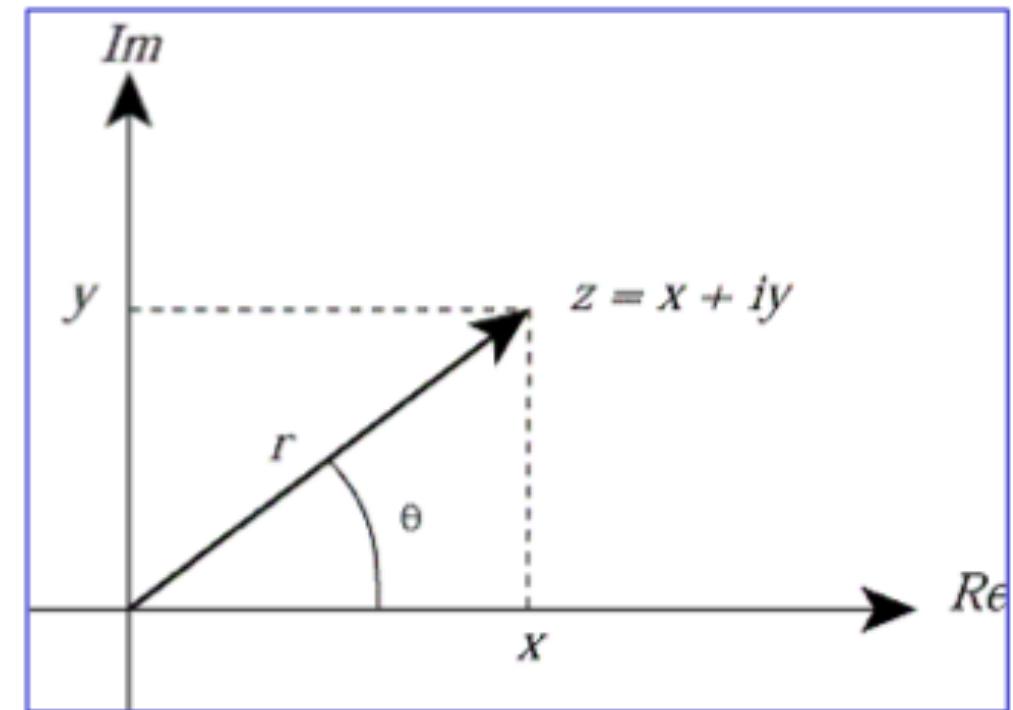
When you run this program, it prints

Sum:-1.0+1.0

Product:-0.0+-1.0

An abstraction for Complex Numbers

- Fields
 - Imaginary Part
 - Real Part
 - r
 - theta
- Operations
 - Add
 - Subtract
 - Multiply
 - Divide



An interface to go with our class

```
public interface Complex {  
    // No constructors, fields, or implementations!  
    double realPart();  
    double imaginaryPart();  
    double r();  
    double theta();  
  
    Complex add(Complex c);  
    Complex multiply(Complex c);  
    Complex subtract(Complex c);  
    Complex divide(Complex c);  
}
```

An interface defines but does not implement API

Modifying class to use interface

```
public class OrdinaryComplex implements Complex{

    final double re; // Real Part
    final double im; // Imaginary Part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im+c.imaginaryPart());
    }

    public Complex multiply(Complex c) { }
    public Complex subtract(Complex c) { }
    public Complex divide(Complex c) { }

}
```

Modifying client to use interface

```
public class ComplexUser {  
    public static void main(String args[]) {  
        Complex c = new OrdinaryComplex(-1, 0);  
        Complex d = new OrdinaryComplex( 0, 1);  
  
        Complex e = c.add(d);  
        System.out.println( "Sum:" + e.realPart() + "+" + e.imaginaryPart());  
  
        e = c.multiply(d);  
        System.out.println( "Product:" + e.realPart() + "+" + e.imaginaryPart());  
    }  
}
```

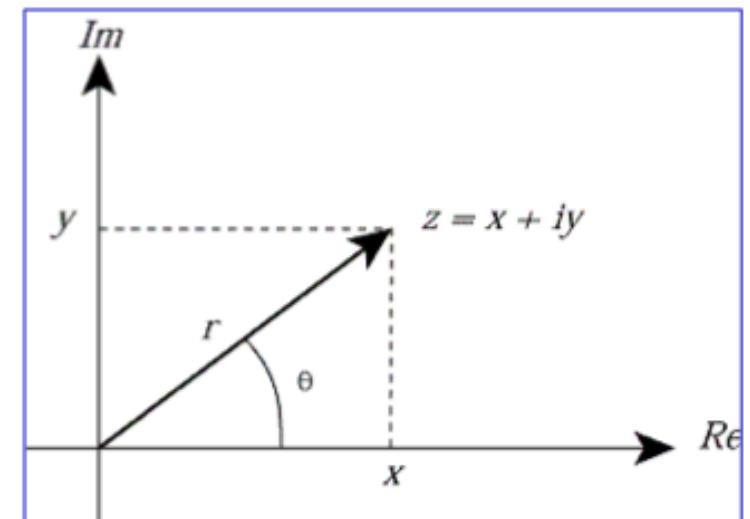
When you run this program, it STILL prints

Sum:-1.0+1.0

Product:-0.0+-1.0

Interface enables multiple implementations

```
public class PolarComplex implements Complex{  
  
    final double r;      // Different Representation  
    final double theta;  
  
    public PolarComplex(double r, double theta) {  
        this.r = r;  
        this.theta = theta;  
    }  
  
    public double realPart() { return r * Math.cos(theta); }  
    public double imaginaryPart() { return r * Math.sin(theta); }  
    public double r() { return r; }  
    public double theta() {return theta;}  
  
    public Complex add(Complex c) {}  
    public Complex multiply(Complex c) {  
        return new PolarComplex(r * c.r(), theta + c.theta());  
    }  
    public Complex subtract(Complex c) {}  
    public Complex divide(Complex c) {}  
}
```



Interface decouples client from implementation

```
public class ComplexUser {  
    public static void main(String args[]) {  
        //Complex c = new OrdinaryComplex(-1, 0);  
        //Complex d = new OrdinaryComplex( 0, 1);  
  
        Complex c = new PolarComplex(1, Math.PI);  
        Complex d = new PolarComplex(1, Math.PI/2);  
  
        Complex e = c.add(d);  
        System.out.println("Sum:" + e.realPart() + "+" + e.imaginaryPart());  
  
        e = c.multiply(d);  
        System.out.println("Product:" + e.realPart() + "+" + e.imaginaryPart());  
    }  
}
```

When you run this program, it STILL prints

Sum:-1.0+1.0

Product:-0.0+1.0

Why multiple implementations?

- Different **performance**
 - Choose implementation that works best for your use
- Different **behavior**
 - Choose implementation that does what you want
 - Behavior *must* comply with interface spec (“contract”)
- Often **performance and behavior both** vary
 - Provides a functionality – performance tradeoff
 - Example: HashSet, LinkedHashSet, TreeSet

Classes - Reminder

- Every object has a **class**
 - A class defines methods and fields
- Class defines both **type** and **implementation**
 - Type ≈ **what** the object is and **where** it can be used
 - Implementation ≈ **how** the object does things
- The methods of a class are its **Application Programming Interface (API)**
 - Defines how users interact with instances

Interfaces and implementations

- An interface defines but does not implement API
- Multiple implementations of an API can coexist
 - Multiple classes can implement the same API
 - OrdinaryComplex, PolarComplex implement the same API
- In Java, an API is specified by *class* or *interface*
 - Class provides an API and an implementation
 - Interface provides *only* an API
 - A class can implement multiple interfaces

Prefer interfaces to classes as types

- Use interface types for parameters and variables unless a single implementation will suffice
 - Supports change of implementation
 - Prevents dependence on implementation details

```
Complex c = new PolarComplex(1, Math.PI);  
Complex d = new PolarComplex(1, Math.PI/2);
```

- But sometimes a single implementation will suffice
 - In those cases write a class and be done with it - **DON'T OVERDO IT**

```
interface Animal {  
    void vocalize();  
}  
  
class Dog implements Animal {  
    public void vocalize() { System.out.println("Woof!"); }  
}  
  
class Cow implements Animal {  
    public void vocalize() { moo(); }  
    public void moo() { System.out.println("Moo!"); }  
}
```

Animal a = new Animal(); a.vocalize(); Compile error

Dog b = new Dog(); b.vocalize(); Woof!

Animal c = new Cow(); c.vocalize(); Moo!

Animal d = new Cow(); d.moo(); Compile error

Encapsulation

- Hiding the implementation details from users
- Single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides internal data and other implementation details from other modules
- Well-designed code hides *all* implementation details
 - Cleanly separates API from implementation
 - Modules communicate *only* through APIs
 - They are unaware of each others' inner workings
- Fundamental principle of software design

Benefits of encapsulation

- **Decouples** the classes that comprise a system
 - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
 - Classes can be developed in parallel
- **Eases burden of maintenance**
 - Classes can be understood more quickly and debugged with little fear of harming other modules
- **Enables effective performance tuning**
 - “Hot” classes can be optimized in isolation
- **Increases software reuse**
 - Loosely-coupled classes often prove useful in other contexts

Encapsulation with interfaces

- Declare variables using interface types
- Client can use only interface methods
- Fields and implementation-specific methods not accessible from client code
- But this takes us only so far
 - Client can access non-interface members directly
 - In essence, it's **voluntary encapsulation**

Mandatory encapsulation

- *Visibility modifiers* for members
 - **private** – Accessible *only* from declaring class
 - **package-private** – Accessible from any class in the package where it is declared
 - Technically known as *default* access
 - You get this if no access modifier is specified – Don't do it!
 - **protected** – Accessible from subclasses of declaring class (and within package)
 - **public** – Accessible from any class

Hiding internal state in OrdinaryComplex

```
public class OrdinaryComplex implements Complex{

    private double re; // Real Part
    private double im; // Imaginary Part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart()      { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() {return Math.atan(im / re);}

    public Complex add(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im+c.imaginaryPart());
    }
}
```

Best practices for encapsulation

- The rule of thumb is simple: **make each class or member as inaccessible as possible.**
- Carefully design your API
- Provide *only* functionality required by clients
 - All other members should be private
- Use the most restrictive access modifier possible
 - You can always make a private member public later without breaking clients but not vice-versa!
 - You can change whatever hidden as much as you want, nobody is affected

Example

TÜRKİYE COVID-19 HASTA TABLOSU		
22 EKİM 2020		
		
BUGÜN	BU HAFTA	TOPLAM
TEST SAYISI 117.198	HASTALarda ZATURRE ORANI %5.6	TEST SAYISI 12.876.267
HASTA SAYISI 2.102	YATAK DOLULUK ORANI %48.1	HASTA SAYISI 355.528
VEFAT SAYISI 71	ERİŞKİN YÖĞÜN BAKIM DOLULUK ORANI %65.2	VEFAT SAYISI 9.584
İYILEŞEN HASTA SAYISI 1.581	VENTİLATÖR DOLULUK ORANI %32.1	AĞIR HASTA SAYISI 1.599
	ORTALAMA TEMASLI TESPİT SÜRESİ 7.5 SAAT	İYILEŞEN HASTA SAYISI 310.027
	FİLYASYON ORANI %99.6	

→ covidData.getSevereCases();

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020 – Lecture 2

Introduction to OOP – Part 2

Object Oriented Programming Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Inheritance

- The process by which one class acquires the properties and functionalities of another class
- Define a new class based on an existing class by extending its common data members and methods.
- Inheritance allows us to reuse of code, it improves reusability in your java application.
- The parent class is called the **base class** or **super class**.
- The child class that extends the base class is called the derived class or **sub class** or **child class**.
- The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

Inheritance In Java

- Single inheritance
 - only extend a single class
 - B is the base-class, A is the sub-class
 - B is the parent, A is the child
-
- The sub-class class inherits all the members and methods that are declared as **public** or **protected**.
 - If the members or methods of super class are declared as private then the derived class cannot use them directly.
 - The private members can be accessed only in its own class.
 - Such private members can only be accessed using public or protected getter and setter methods of super class.
- ```
class A extends B {
}
}
```

# Example

```
class Car {
 private String color = "red";
 private String brand = "tesla";

 protected String getColor(){ return color; };
 protected void setColor(String c){ this.color = c; };

 protected String getBrand(){ return brand; };
 protected void setBrand(String b){ this.brand = b; };

 void honk(){
 System.out.println("dut dut");
 }
}
```

```
public class MyCar extends Car{

 String gear = "automatic";

 public static void main(String[] args) {

 MyCar c = new MyCar();
 System.out.println(c.getColor());
 System.out.println(c.getBrand());
 System.out.println(c.gear);

 c.honk();
 }
}
```

## Output:

```
red
tesla
automatic
dut dut
```

# What is the output?

```
public class Parent {
 public void foo() {
 bar();
 }

 public void bar() {System.out.println("bar");}
}
```

```
public class Child extends Parent {

 // foo in Parent is actually calling this bar!
 public void bar() {
 foo();
 }

 public static void main(String[] args) {
 new Child().bar();
 }
}
```

Inheritance creates a tight coupling between base and derived classes.

```
public class Parent {
 public void foo() {
 bar();
 }
}
Choose Implementation of Parent.bar() (2 found)
Child
Parent
public void bar() {System.out.println("bar");}
```

# Private members are not inherited

- What is the output?

```
public class Parent {
 public void foo() {
 bar();
 }

 private void bar() {System.out.println("bar");}
}
```

```
public class Child extends Parent {

 public void bar() {
 foo();
 }

 public static void main(String[] args) {
 new Child().bar();
 }
}
```

bar

The rule of thumb is : make each class or member as inaccessible as possible.

# Let's assume Parent's bar() has to be public

- For some reason!
- Make sure overridable methods don't call other overridable methods.
- Have them use either private helper methods, or
- Make them final
  - Methods declared as final cannot be overridden.

```
public class Parent {
 public void foo() {
 barHelper();
 }

 public void bar() {
 barHelper();
 }

 private void barHelper() {
 // do whatever you want to do here!
 }
}
```

# My Hash Set

```
class MyHashSet<T> extends HashSet<T>{

 //The number of attempted element insertions since its creation --
 // this value will not be modified when elements are removed
 private int addCount = 0;

 public MyHashSet() {}

 @Override
 public boolean add(T a) {
 addCount++;
 return super.add(a);
 }

 @Override
 public boolean addAll(Collection<? extends T> c) {
 addCount += c.size();
 return super.addAll(c);
 }

 public int getAddCount() {
 return addCount;
 }
}
```

## Side Note

### Usage of Super Keyword

- 1** Super can be used to refer immediate parent class instance variable.
- 2** Super can be used to invoke immediate parent class method.
- 3** super() can be used to invoke immediate parent class constructor.

# What is the output?

```
public class MyHashSetUser {
 public static void main(String args[]) {
 MyHashSet<String> mhs = new MyHashSet<>();
 mhs.add("a");
 mhs.add("b");
 mhs.add("c");

 System.out.println("Number of attempted adds so far " + mhs.getAddCount());

 mhs.remove("b");
 System.out.println("Number of attempted adds after remove " + mhs.getAddCount());

 mhs.addAll(Arrays.asList("d","e","f"));
 System.out.println("Number of attempted adds after addall " + mhs.getAddCount());
 System.out.println("Number of elements in set " + mhs.size());
 }
}
```

Number of attempted adds so far 3

Number of attempted adds after remove 3

Number of attempted adds after addall 9 → What the heck!

Number of elements in set 5

# My Hash Set

```
class MyHashSet<T> extends HashSet<T>{

 //The number of attempted element insertions since its creation --
 // this value will not be modified when elements are removed
 private int addCount = 0;

 public MyHashSet() {}

 @Override
 public boolean add(T a) {
 addCount++;
 return super.add(a);
 }

 @Override
 public boolean addAll(Collection<? extends T> c) {
 addCount += c.size();
 return super.addAll(c);
 }

 public int getAddCount() {
 return addCount;
 }
}
```

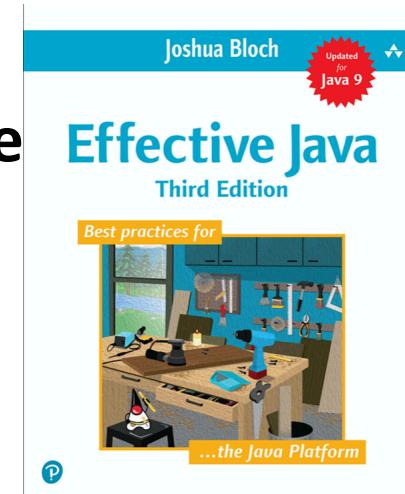
The cause of the problem is that:  
In the implementation of HashSet, addAll calls the add method.  
Therefore, we are incrementing addCount too many times in calls to addAll.

# The Real Problem

- Inheritance violates encapsulation !
- It forces you to know the implementation details of the class you are extending

# Takeaway Message

- Inheritance is one of the fundamental principles in an object-oriented paradigm, bringing a lot of values in software design and implementation.
- However, there are situations where even the correct use of inheritance breaks the implementations.
- **Item 19: Design and document for inheritance or else prohibit it**
- **Item 18: Favor composition over inheritance**



# Abstract classes

- We have seen
  - Interfaces no implementation
  - (Concrete) classes full implementation
- Abstract classes are somewhere in between
- Java classes can
  - **implements** multiple interfaces
  - **extends** a single class

# Example: Account Types

«interface» CheckingAccount

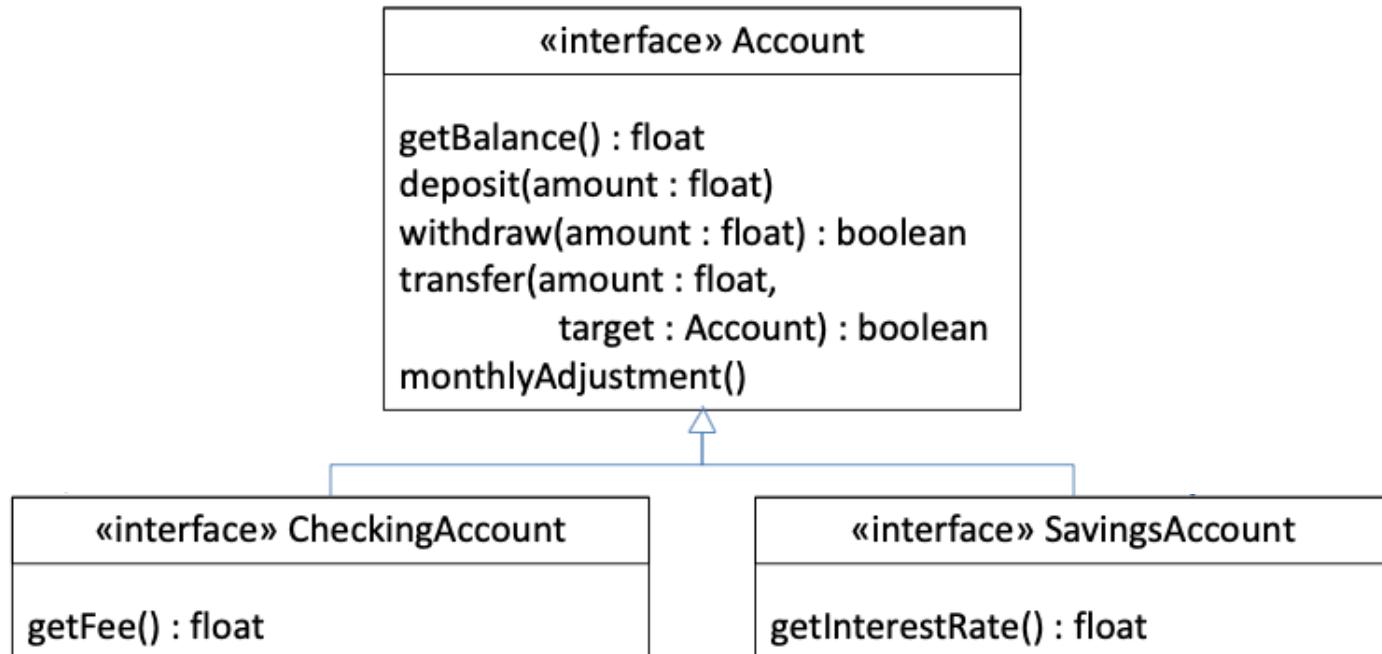
```
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
 target : Account) : boolean
getFee() : float
```

«interface» SavingsAccount

```
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
 target : Account) : boolean
getInterestRate() : float
```

Lots of common methods!

# Better Design



# The code

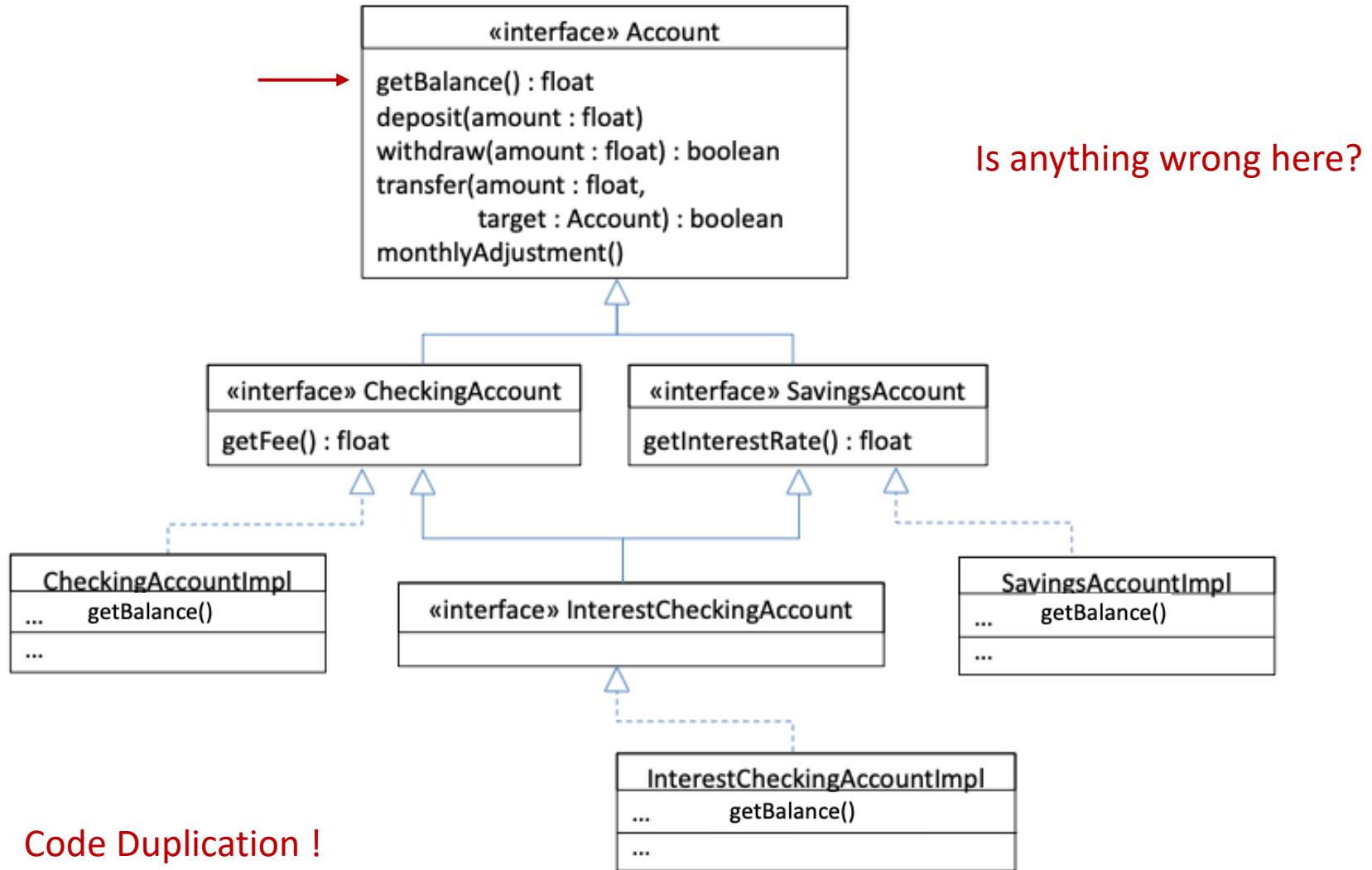
```
public interface Account {
 public long getBalance();
 public void deposit(long amount);
 public boolean withdraw(long amount);
 public boolean transfer(long amount, Account target);
 public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
 public long getFee();
}

public interface SavingsAccount extends Account {
 public double getInterestRate();
}

public interface InterestCheckingAccount
 extends CheckingAccount, SavingsAccount {
}
```

# Add the implementations



# Abstract class is the remedy

- An *abstract class* is a convenient hybrid between an interface and a full implementation.
- Can have
  - Abstract methods (no body)
  - Concrete methods (w/ body)
  - Data fields
- An *interface* defines expectations / commitment for clients
  - can declare methods but cannot implement them
  - All methods are *abstract methods*

# Code Reuse with Abstract Methods

```
public abstract class AbstractAccount
 implements Account {
 protected float balance = 0.0;
 public float getBalance() {
 return balance;
 }
 abstract public void monthlyAdjustment();
 // other methods...
}
```

Missing one or more method implementation

Protected elements are visible in subclasses

«interface» Account

|                                    |
|------------------------------------|
| getBalance() : float               |
| deposit(amount : float)            |
| withdraw(amount : float) : boolean |
| transfer(amount : float,           |
| target : Account) : boolean        |
| monthlyAdjustment()                |

```
public class CheckingAccountImpl
 extends AbstractAccount
 implements CheckingAccount {
 public void monthlyAdjustment() {
 balance -= getFee();
 }
 public float getFee() /* fee calculation */
}
```

Abstract method is left to be implemented in a subclass

«interface» CheckingAccount

|                  |
|------------------|
| getFee() : float |
|------------------|

AbstractAccount

|                                      |
|--------------------------------------|
| # balance : float                    |
| + getBalance() : float               |
| + deposit(amount : float)            |
| + withdraw(amount : float) : boolean |
| + transfer(amount : float,           |
| target : Account) : boolean          |
| + monthlyAdjustment()                |

CheckingAccountImpl

|                     |
|---------------------|
| monthlyAdjustment() |
| getFee() : float    |

No need to define getBalance() – the code is inherited from AbstractAccount

# Interface – Abstract Class – Concrete Class

- An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Abstract methods (no implementation)
  - Concrete methods (with implementation)
  - Data fields
- Unlike a concrete class, an *abstract class*
  - *Cannot be instantiated*
  - *Can declare abstract methods*
    - Which *must* be implemented in all *concrete* subclasses
- An abstract class may *implement* an interface
  - But need not implement all methods of the interface
  - Implementation of some of them is left to subclasses

# Constructors revisited

- Every class has a constructor whether it's concrete or abstract
- Interfaces **do not have constructors**
- Constructors can not be inherited --> we can't override a constructor
- Abstract class can have constructor and it gets invoked when a concrete class, which implements it, is instantiated. (i.e. object creation of concrete class)

# Constructors revisited

- Constructor can use any access specifier, they can be declared as **private** also.
- A constructor can also invoke another constructor of the same class
  - By using `this()` or `this(parameter list)`
- **`this()` and `super()` should be the first statement in the constructor code**
- If you don't implement any constructor, compiler will do it for you
- If Super class doesn't have a no-arg (default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario

```
public class CheckingAccountImpl
 extends AbstractAccount implements CheckingAccount {

 private long fee;

 public CheckingAccountImpl(long initialBalance, long fee) {
 super(initialBalance);
 this.fee = fee;
 }

 public CheckingAccountImpl(long initialBalance) {
 this(initialBalance, 500);
 }
 /* other methods... */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class

# Constructors revisited

- Constructors are not methods and they don't have any return type
- Constructor name should match with class name
- There can be methods with the same name as the class – **they must have return types**
- Constructor overloading is possible

# Mini quiz

- Can an interface extend more than one interface in Java? -Yes
- What will happen if a class implements two interfaces and both interfaces have a method with the same signature but different return types?
  - In this case, a conflict will arise because the compiler will not be able to link a method call due to ambiguity. You will get a compile time error in Java.

# Back to Inheritance

- + allows code reuse
  - + provides design flexibility
  - - breaks encapsulation
  - - creates entanglement
- 
- Are there other ways of code reuse?
  - **Item18: Favor composition over inheritance**

# Composition

- Instead of extending an existing class, give your new class a private field that references an instance of the existing class.
- This design is called *composition* because the existing class becomes a component of the new one.

# Using composition to reuse code

- Consider `java.util.List`

```
public interface List<E> {
 public boolean add(E e);
 public E remove(int index);
 public void clear();
 ...
}
```

- Suppose we want a list that logs its operations to the console

# Solution

give your new class a private field that references an instance of the existing class.

```
public class LoggingList<E> implements List<E> {
 private final List<E> list;
 public LoggingList<E>(List<E> list) { this.list = list; }
 public boolean add(E e) {
 System.out.println("Adding " + e);
 return list.add(e);
 }
 public E remove(int index) {
 System.out.println("Removing at " + index);
 return list.remove(index);
 }
 ...
```

# Summary

- Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass.
- In other words, a class *B* should extend a class *A* only if an “is-a” relationship exists between the two classes.
- Composition builds “has-a” relationship
- How to test “is-a” relationship?

CENG 443  
Introduction to Object-Oriented Programming  
Languages and Systems

Hande Alemdar

Fall 2020 – Lecture 2

Introduction to OOP – Part 3

# Object Oriented Programming Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

But before that

# A Note on Interfaces

|                  | <b>Java 7 and Earlier</b>                                                                                                                                                                                                             | <b>Java 8 and Later</b>                                                                                                                                                                                                                |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Abstract Classes | <ul style="list-style-type: none"><li>• Can have concrete methods and abstract methods</li><li>• Can have static methods</li><li>• Can have instance variables</li><li>• Class can directly extend one</li></ul>                      | (Same as Java 7)                                                                                                                                                                                                                       |
| Interfaces       | <ul style="list-style-type: none"><li>• Can only have abstract methods – no concrete methods</li><li>• Cannot have static methods</li><li>• Cannot have mutable instance variables</li><li>• Class can implement any number</li></ul> | <ul style="list-style-type: none"><li>• Can have concrete (default) methods and abstract methods</li><li>• Can have static methods</li><li>• Cannot have mutable instance variables</li><li>• Class can implement any number</li></ul> |

**More on this later!**

# @Override

- Not mandatory, but best practice
- **Catches errors at compile time instead of run time**
  - If you make a typo in the name or signature of overridden method, it would still compile but would give wrong answer at compile time
  - This point applies only to extending regular classes, not to extending abstract classes or implementing interfaces
- **Expresses design intent**
  - Tells later maintainer “the meaning of this method comes from the parent class, I am not just inventing a new method”
- **It improves the readability of the code**
  - if you change the signature of overridden method then all the sub classes that overrides the particular method would throw a compilation error, which would eventually help you to change the signature in the sub classes.
  - If you have lots of classes in your application then this annotation would really help you to identify the classes that require changes when you change the signature of a method.

# Example

```
public class Ellipse implements Shape {
 public double getArea() { ... }
}
```

```
public class Circle extends Ellipse {
 public double getarea() { ... }
}
```

Java is case sensitive!

```
public class Circle extends Ellipse {
 @Override
 public double getarea() { ... }
}
```

This tells the compiler "I think that I am overriding a method from the parent class". If there is no such method in the parent class, code won't compile. If there is such a method in the parent class, then `@Override` has no effect on the code. Recommended but optional.  
More on `@Override` in later sections.

# Polymorphism

- Polymorphism allows us to perform a single action in different ways
- Polymorphism is the capability of a method to do different things based on the object that it is acting upon
- Polymorphism allows you define one interface and have multiple implementations

# Types of Polymorphism

- 1) **Static Polymorphism** also known as compile time polymorphism
  - Ex: Method *overloading*
- 2) **Dynamic Polymorphism** also known as runtime polymorphism
  - Ex: Method *overriding*

```
class Shape{
 void draw(){System.out.println("drawing...");}
}

class Rectangle extends Shape{
 void draw(){System.out.println("drawing rectangle...");}
}

class Circle extends Shape{
 void draw(){System.out.println("drawing circle...");}
}

class Triangle extends Shape{
 void draw(){System.out.println("drawing triangle...");}
}
```

```
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
```

# What's the purpose of polymorphism?

- decouple the client class from implementation code
- client class receives the implementation to execute the necessary action
- the client class knows just enough to execute its actions, which is an example of loose coupling

# Example

```
public abstract class SweetProducer {
 public abstract void produceSweet();
}

public class CakeProducer extends SweetProducer {
 @Override
 public void produceSweet() {
 System.out.println("Cake produced");
 }
}

public class ChocolateProducer extends SweetProducer {
 @Override
 public void produceSweet() {
 System.out.println("Chocolate produced");
 }
}

public class CookieProducer extends SweetProducer {
 @Override
 public void produceSweet() {
 System.out.println("Cookie produced");
 }
}
```

The creator class knows nothing about the runtime type

```
public class SweetCreator {
 private List<SweetProducer> sweetProducer;
 public SweetCreator(List<SweetProducer> sweetProducer) {
 this.sweetProducer = sweetProducer;
 }
 public void createSweets() {
 sweetProducer.forEach(sweet -> sweet.produceSweet());
 }
}
```

The client class knows nothing about implementation details

```
public static void main(String... args) {
 SweetCreator sweetCreator = new SweetCreator(Arrays.asList(
 new ChocolateProducer(), new CookieProducer()));
 sweetCreator.createSweets();
}
```

Loosely coupled design

# Covariant return types in method overriding

- It's possible to change the return type of an overridden method if it is a covariant type
- A *covariant type* is basically a subclass of the return type.

```
public abstract class JavaMascot {
 abstract JavaMascot getMascot();
}

public class Duke extends JavaMascot {
 @Override
 Duke getMascot() {
 return new Duke();
 }
}
```

Because Duke is a JavaMascot, we are able to change the return type when overriding.

# Polymorphism with the core Java classes

- `List<String> list = new ArrayList<>();`
- `List<String> list = new LinkedList<>();`

# Mini Quiz

```
static abstract class Simpson {
 void talk() {
 System.out.println("Simpson!");
 }
 protected void prank(String prank) {
 System.out.println(prank);
 }
}

static class Bart extends Simpson {
 String prank;
 Bart(String prank) { this.prank = prank; }
 protected void talk() {
 System.out.println("Eat my shorts!");
 }
 protected void prank() {
 super.prank(prank);
 System.out.println("Knock Homer down");
 }
}

static class Lisa extends Simpson {
 void talk(String toMe) {
 System.out.println("I love Sax!");
 }
}
```

```
public static void main(String... doYourBest) {
 new Lisa().talk("Sax :");
 Simpson simpson = new Bart("D'oh");
 simpson.talk();
 Lisa lisa = new Lisa();
 lisa.talk();
 ((Bart) simpson).prank();
}
```

Don't use an IDE to figure this out!  
The point is to improve your code analysis skills, so try to determine the output for yourself.

- |                                                                                      |                                                                                 |
|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| A) I love <b>Sax!</b><br>D'oh<br>Simpson!<br>D'oh                                    | C) <b>Sax :</b><br>D'oh<br>Simpson!<br>Knock Homer down                         |
| B) <b>Sax :)</b><br>Eat my shorts!<br>I love <b>Sax!</b><br>D'oh<br>Knock Homer down | D) I love <b>Sax!</b><br>Eat my shorts!<br>Simpson!<br>D'oh<br>Knock Homer down |

# Common Mistakes

- To think it's possible to invoke a specific method without using casting.
- Being unsure what method will be invoked when instantiating a class polymorphically.
  - Remember that the method to be invoked is the method of the created instance.
- remember that method overriding is not method overloading.
- It's impossible to override a method if the parameters are different.
- It *is possible* to change the return type of the overridden method if the return type is a subclass of the superclass method.

# Polymorphism - summary

- The created instance will determine what method will be invoked when using polymorphism.
- The `@Override` annotation obligates the programmer to use an overridden method; if not, there will be a compiler error.
- Polymorphism can be used with normal classes, abstract classes, and interfaces.
- Most design patterns depend on some form of polymorphism.
- The only way to use a specific method in your polymorphic subclass is by using casting.
- It's possible to design a powerful structure in your code using polymorphism.

# Binding

- Association of method call to the method body is known as binding
- **Static Binding** that happens at compile time
  - The binding of overloaded methods is static
  - Binding of private, static and final methods is static since these methods cannot be overridden
- **Dynamic Binding** that happens at runtime
  - Method Overriding - both parent and child classes have the same method and in this case the **type** of the object determines which method is to be executed.

# What is the output?

```
public class CollectionClassifier {
 public static String classify(Set<?> s) {
 return "Set";
 }

 public static String classify(List<?> l) {
 return "List";
 }

 public static String classify(Collection<?> c) {
 return "Unknown Collection";
 }

 public static void main(String[] args) {
 Collection<?>[] collections = {
 new HashSet<String>(),
 new ArrayList<BigInteger>(),
 new HashMap<String, String>().values()
 };

 for (Collection<?> c : collections)
 System.out.println(classify(c));
 }
}
```

- A. Set, List, Unknown Collection
- B. Runtime error
- C. Unknown Collection, Unknown Collection, Unknown Collection
- D. Compile error

The answer is C because **classify** method is *overloaded*, and **the choice of which overloading to invoke is made at compile time**

**Item 52: Use overloading judiciously**

# You can fix the code as follows

- Using the ugly instanceof and ternary operator

```
public static String classify(Collection<?> c) {
 return c instanceof Set ? "Set" :
 c instanceof List ? "List" : "Unknown Collection";
}
```

DO NOT DO THAT UNLESS YOU REALLY HAVE TO

# instanceof

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
 long adj = 0;
 if (acct instanceof CheckingAccount) {
 checkingAcct = (CheckingAccount) acct;
 adj = checkingAcct.getFee();
 } else if (acct instanceof SavingsAccount) {
 savingsAcct = (SavingsAccount) acct;
 adj = savingsAcct.getInterest();
 }
}
```

- avoid instanceof if possible
- Never(?) use instanceof in a superclass to check type against subclass

# Use polymorphism to avoid instanceof

```
public interface Account {
 ...
 public long getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
 ...
 public long getMonthlyAdjustment() {
 return getFee();
 }
}

public class SavingsAccount implements Account {
 ...
 public long getMonthlyAdjustment() {
 return getInterest();
 }
}
```

~~public void doSomething(Account acct) {  
 long adj = 0;  
 if (acct instanceof CheckingAccount) {  
 checkingAcct = (CheckingAccount) acct;  
 adj = checkingAcct.getFee();  
 } else if (acct instanceof SavingsAccount) {  
 savingsAcct = (SavingsAccount) acct;  
 adj = savingsAcct.getInterest();  
 }  
 ...  
}~~

Instead:

```
public void doSomething(Account acct) {
 long adj = acct.getMonthlyAdjustment();
 ...
}
```

# type-casting in Java

- Sometimes you want a different type than you have
  - double pi = 3.14;  
`int indianaPi = (int) pi;`
- Useful if you know you have a more specific subtype
  - Account acct = ...;  
`CheckingAccount checkingAcct = (CheckingAccount) acct;`  
`long fee = checkingAcct.getFee();`
  - Will get a ClassCastException if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

# final

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
- Can an interface have final methods?

# Mutability / Immutability

- If an object of a class can be changed, then this class is called mutable class
  - if you can change/update the values of an object **without creating another object** then it's mutable
- If an object of a class cannot be changed, then this class is called immutable class
- final vs immutable
  - final → cannot be reinitialized, we cannot assign a new object to that variable, but we can change the existing object!
  - immutable → cannot be changed without creating a new object
  - final is a reserved word!

# To make a class immutable, follow these five rules

1. Don't provide methods that modify the object's state
2. Ensure that the class can't be extended
  - make the class final
  - make the constructor private and construct instances in factory methods
3. Make all fields final
4. Make all fields private
5. Ensure exclusive access to any mutable components
  - If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects

**Item17: Minimize mutability**

# String is Immutable

- String s = "hello"; // String s = new String("hello");
- s.concat(" world"); // String temp = " world";  
// String temp2 = "hello world";  
// s = temp2;

**The major disadvantage of immutable classes is that they require a separate object for each distinct value.**

Immutable classes are easier to design, implement, and use.  
They are less prone to error and are more secure.  
Immutable objects are inherently thread-safe; they require no synchronization.

# Use immutable objects carefully!

- String, Boxed primitives, BigInteger and BigDecimal are all immutable
- Creating these objects can be costly, especially if they are large
- Suppose that you have a million-bit BigInteger and you want to change its low-order bit:
- ```
BigInteger moby = ...;
moby = moby.flipBit(0);
```
- The flipBit method creates a new BigInteger instance, also a million bits long, that differs from the original in only one bit. The operation requires time and space proportional to the size of the BigInteger
- Look for mutable companions!
 - BitSet is mutable companion for BigInteger
 - StringBuilder is mutable companion for String

static

- Static keyword can be used with class, variable, method and block
- Static members belong to the class instead of a specific instance
 - if you make a member static, you can access it without object
- When we make a member static it becomes class level
- Static members are common for all the instances(objects) of the class but non-static members are separate for each instance of class

Java Static Variables

- A static variable is common to all the instances (or objects) of the class because it is a class level variable
- In other words you can say that only a single copy of static variable is created and shared among all the instances of the class
- Memory allocation for such variables only happens once when the class is loaded in the memory.
- Static variables are also known as Class Variables.
- static variables can be accessed directly in static and non-static methods

Example

```
class VariableDemo
{
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        VariableDemo obj1=new VariableDemo();
        VariableDemo obj2=new VariableDemo();
        obj1.increment();
        obj2.increment();
        System.out.println("Obj1: count is="+obj1.count);
        System.out.println("Obj2: count is="+obj2.count);
    }
}
```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

Static variable initialization

- Static variables are initialized when class is loaded
- Static variables are initialized before any object of that class is created
- Static variables are initialized before any static method of the class executes
- Default values for static and non-static variables are same.
 - primitive integers(long, short etc): 0
 - primitive floating points(float, double): 0.0
 - boolean: false
 - object references: null
- The static final variables are constants
 - **Constant variable name should be in Caps! you can use underscore(_) between.**
 - public static final int MY_VAR=27;

Static Variable access

- Static Variable can be accessed directly in a static method and a non-static one

```
public class StaticExample {  
    static int age;  
    static String name;  
  
    //This is a non-static method  
    void disp(){  
        System.out.println("Age is: "+age);  
        System.out.println("Name is: "+name);  
    }  
    // This is a static method  
    public static void main(String args[]) {  
        age = 30;  
        name = "Steve";  
    }  
}
```

Static Methods

- Static methods can be accessed directly in static and non-static methods

```
public class StaticExample {  
    static void dispStatic(){  
        System.out.println("Static");  
    }  
  
    void dispNonStatic(){  
        System.out.println("Non-static");  
    }  
  
    // This is a static method  
    public static void main(String args[]){  
        dispStatic();  
  
        StaticExample ex = new StaticExample();  
        ex.dispNonStatic();  
    }  
}
```

You can call them from another class as follows:
StaticExample.*dispStatic()*;

Static Classes

- A class can be made **static** only if it is a nested (inner) class.
- Inner static class doesn't need reference of Outer class
- A static class cannot access non-static members of the Outer class

```
Outer class           public class StaticExample {  
                        private static String str = "hello";  
  
Inner class           //Static class  
                     ↴   static class MyNestedClass{  
                         //non-static method  
                         ↴   public void disp() {  
  
                             /* If you make the str variable of outer class  
                             * non-static then you will get compilation error  
                             * because: a nested static class cannot access non-  
                             * static members of the outer class.  
                             */  
                             System.out.println(str);  
                         }  
                     }  
                     ↴   public static void main(String args[])  
                     {  
                         /* To create instance of nested class we didn't need the outer  
                         * class instance but for a regular nested class you would need  
                         * to create an instance of outer class first  
                         */  
                         StaticExample.MyNestedClass obj = new StaticExample.MyNestedClass();  
                         obj.disp();  
                     }  
                 }
```

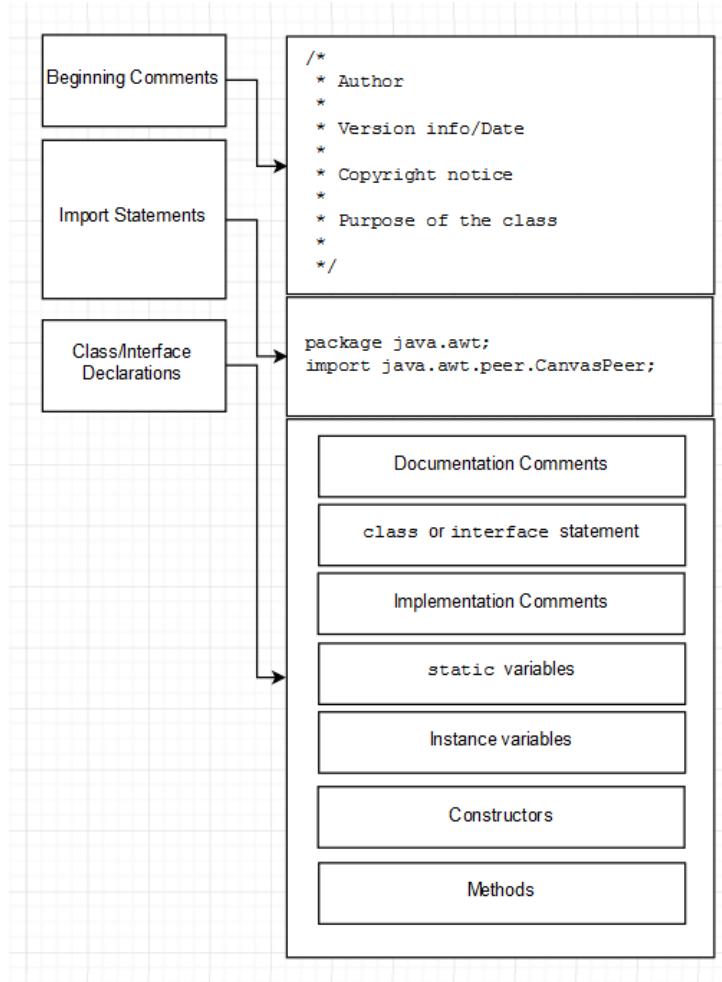
Static Quiz !

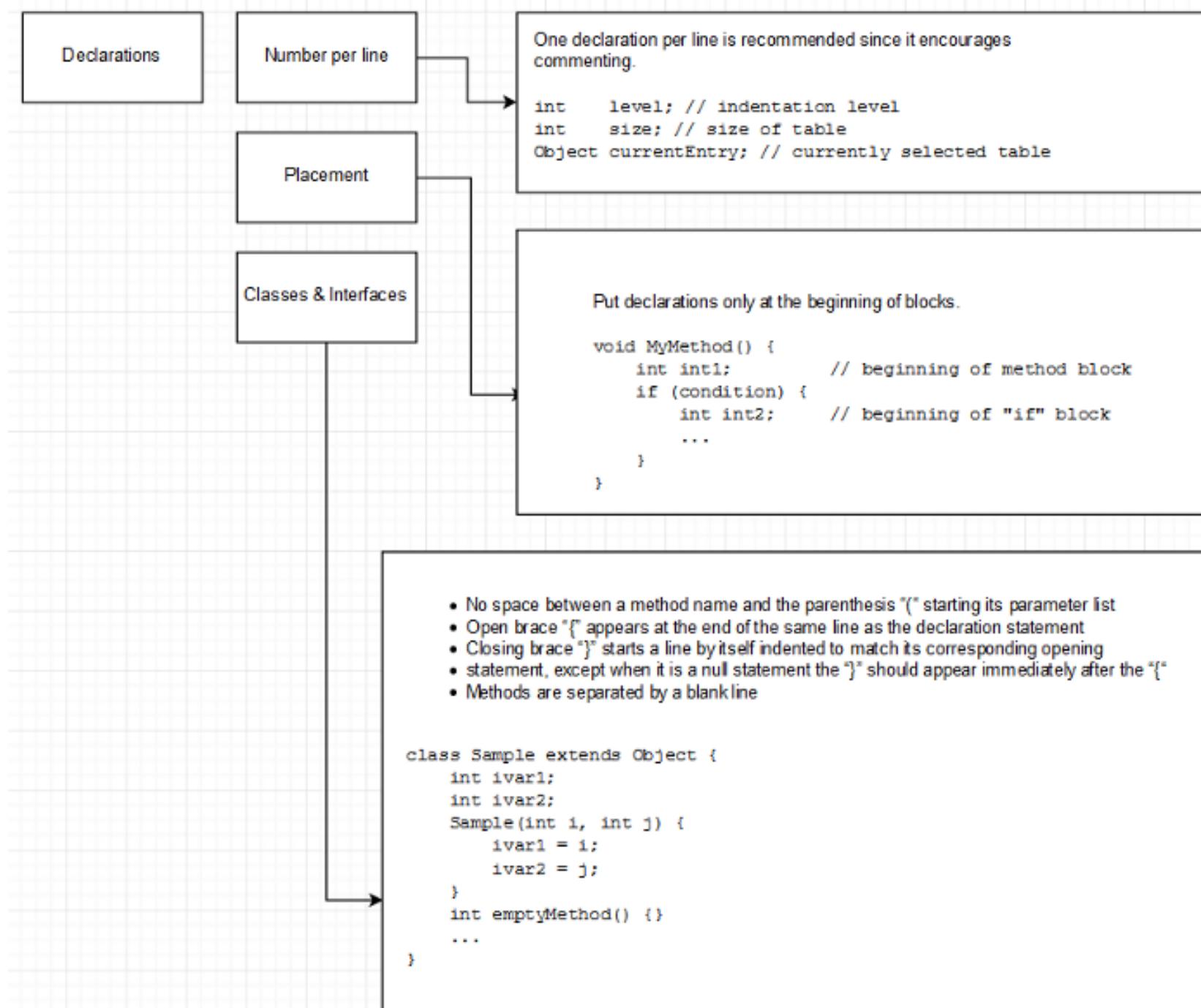
- Can an interface have static variables?
 - Yes but they have to be final also! → CONSTANTS
- Can a class have static constructor?
 - No!
- Can we override static methods?
 - No, because **method overriding** is based on dynamic binding at runtime and the **static methods** are bonded using **static binding** at compile time.

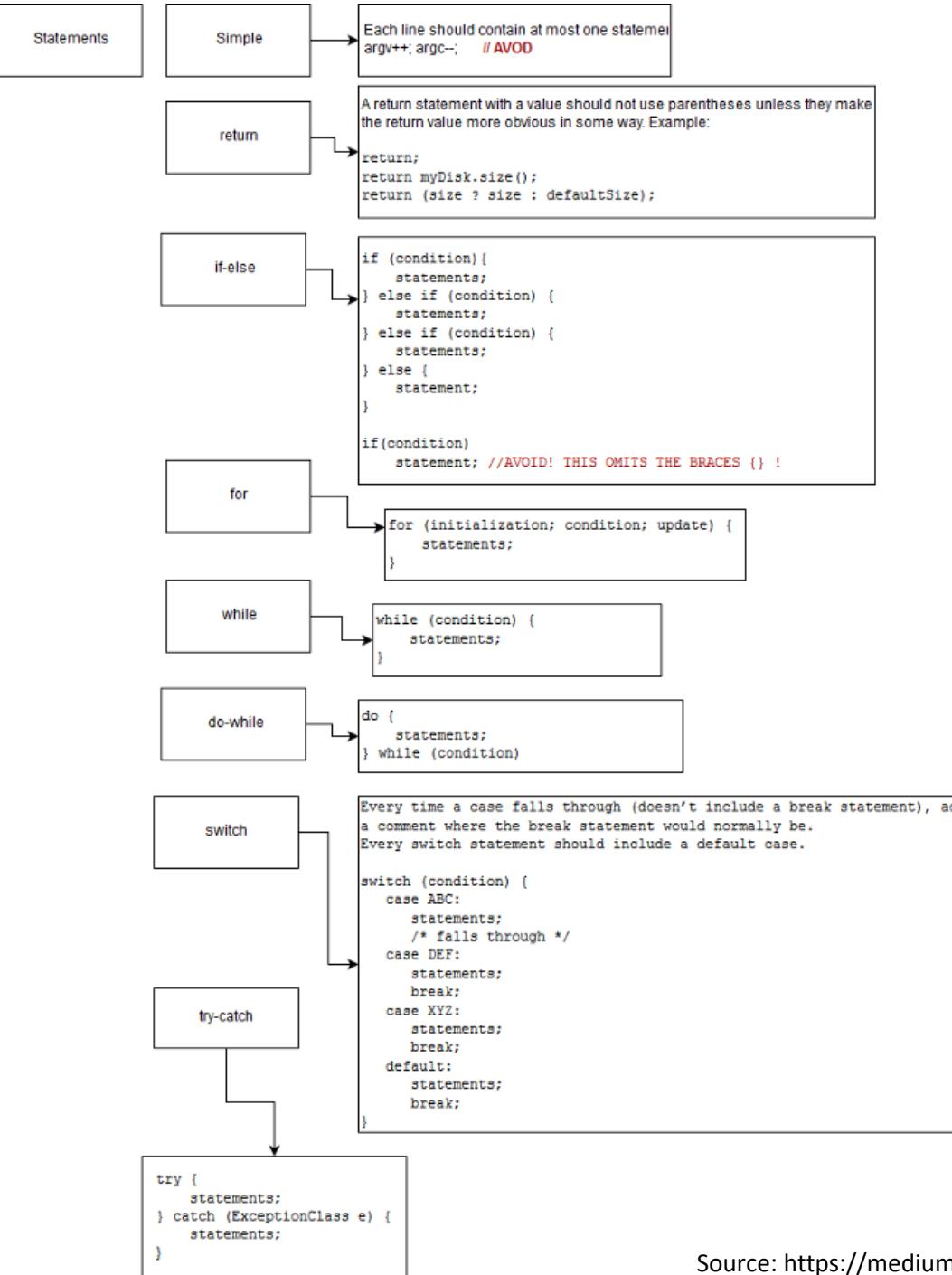
Important things

- Java coding conventions
 - <https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
 - <https://google.github.io/styleguide/javaguide.html>
- Documentation
- Indentation
- Code organization

The standard source file structure







CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Java Garbage Collection

What is GC?

- The Java virtual machine's heap stores all objects created by a running Java application
- Objects are created by the program through *new* keyword, but never freed explicitly by the program
 - No need to call free()
- Garbage collection is the process of automatically freeing objects that are no longer needed
- An object is determined to be “no longer needed” when there is no other object referencing to it
 - Each object has a reference counter - when it becomes 0, it means there is no other object referencing to it

Advantages of GC

- Programmer is free from memory management
 - Less error prone code
- System cannot crash due to memory management
 - More reliable application
 - Memory-leak is still possible
 - You can use memory profiler to find out where memory-leaking code is located

Disadvantages of GC

- GC could add overhead
 - Many GC schemes are focused on minimizing GC overhead
- GC can occur in an non-deterministic way

When Does GC Occur?

- JVM performs GC when it determines the amount of free heap space is below a threshold
 - This threshold can be set when a Java application is run
- You explicitly request garbage collection

How Does JVM Perform GC?

- Find the objects no longer reachable
 - MyObject obj = new MyObject();
obj = null;
- Find the objects references copied to other reference
 - MyObject obj1 = new MyObject();
MyObject obj2 = new MyObject();
obj2 = obj1;

the instance (object) pointed by (referenced by) obj2 is not reachable and available for garbage collection.

How Does JVM Perform GC?

- The garbage collector must somehow determine which objects are no longer referenced and make available the heap space occupied by such unreferenced objects.
- The simplest and most crude scheme is to keep reference counter to each object
- There are many different schemes - years of research

GC Related Java API

- *finalize() method* in Object class
 - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- gc() method in System class
 - Runs the garbage collector.
 - Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse.
 - When control returns from the method call, the Java Virtual Machine has made a **best effort** to reclaim space from all discarded objects.

Example

```
public class Example{
    public static void main(String args[]){
        /* Here we are intentionally assigning a null
         * value to a reference so that the object becomes
         * non reachable
         */
        Example obj=new Example();
        obj=null;

        /* Here we are intentionally assigning reference a
         * to the another reference b to make the object referenced
         * by b unusable.
         */
        Example a = new Example();
        Example b = new Example();
        b = a;
        System.gc();
    }

    protected void finalize() throws Throwable          Overridden finalize() method
    {
        System.out.println("Garbage collection is performed by JVM");
    }
}
```

Garbage collection is performed by JVM
Garbage collection is performed by JVM

Process finished with exit code 0

Item 8: Avoid finalizers and cleaners

- Finalizers are unpredictable, often dangerous, and generally unnecessary.
- As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries.
- The Java 9 replacement for finalizers is *cleaners*.
Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.
- One shortcoming of finalizers and cleaners is that there is no guarantee they'll be executed promptly
- It can take arbitrarily long between the time that an object becomes unreachable and the time its finalizer or cleaner runs.
 - you should **never do anything time-critical in a finalizer or cleaner.**

Item 6: Avoid creating unnecessary objects

- `String s = new String("hello");`
- `String s = "hello";`

The argument to the String constructor ("hello") is itself a String instance !

Can you spot the problem?

```
private static long sum() {  
    Long sum = 0L;  
    for (long i = 0; i <= Integer.MAX_VALUE; i++)  
        sum += i;  
  
    return sum;  
}
```

The variable sum is declared as a Long instead of a long, which means that the program constructs about 2^{31} unnecessary Long instances!

prefer primitives to boxed primitives, and watch out for unintentional autoboxing

Item 7: Eliminate obsolete object references

- GC does not mean that you don't have to think about memory management
- Memory Leak can happen and in fact happens all the time

```
public class Stack {  
    private Object[] elements;  
    private int size = 0;  
    private static final int DEFAULT_INITIAL_CAPACITY = 16;  
  
    public Stack() {  
        elements = new Object[DEFAULT_INITIAL_CAPACITY];  
    }  
  
    public void push(Object e) {  
        ensureCapacity();  
        elements[size++] = e;  
    }  
  
    public Object pop() {  
        if (size == 0)  
            throw new EmptyStackException();  
        Object result = elements[--size];  
        elements[size] = null; // Eliminate obsolete reference  
        return result;  
    }  
    /**  
     * Ensure space for at least one more element, roughly  
     * doubling the capacity each time the array needs to grow.  
     */  
    private void ensureCapacity() {  
        if (elements.length == size)  
            elements = Arrays.copyOf(elements, 2 * size + 1);  
    }  
}
```

Can you find the memory leak?

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Java Packages

Packages

- A package is a grouping of related **types** providing access protection and name space management
- Types refer to classes, interfaces, enumerations, and annotation types
- Types are often referred to as classes and interfaces since enumerations and annotation types are special kinds of classes and interfaces

Advantages of using a package in Java

- **Reusability:** While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
- **Better Organization:** in large projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
- **Name Conflicts:** We can define two classes with the same name in different packages so to avoid name collision, we can use packages

Types of packages

- User defined package: The package we create is called user-defined package.
- Built-in package: The already defined packages
 - `java.io.*`, `java.lang.*`, `java.util.*`

Creating a package

- To create a package, you choose a name for the package and put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package
- If you do not use a package statement, your type ends up in an unnamed package
- Use an unnamed package only for small or temporary applications

Placing a Class in a Package

- To place a class in a package, we write the following as the first line of the code (except comments)

package <packageName>;

package myownpackage;

Example package

- Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points
- You also write an interface, Draggable, that classes implement if they can be dragged with the mouse

```
//in the Draggable.java file  
public interface Draggable {}
```

```
//in the Graphic.java file  
public abstract class Graphic {}
```

```
//in the Circle.java file  
public class Circle extends Graphic implements Draggable {}
```

```
//in the Rectangle.java file  
public class Rectangle extends Graphic implements Draggable {}
```

```
//in the Point.java file  
public class Point extends Graphic implements Draggable {}
```

```
//in the Line.java file  
public class Line extends Graphic implements Draggable {}
```

Placing a class to a package

```
package SchoolClasses;  
  
public class StudentRecord {  
    private String name;  
    private String address;  
    private int age;  
    :  
}
```

the StudentRecord.class file must be placed under the directory named SchoolClasses.

In case of an unnamed package, the current directory is implied.

Using Classes from Other Packages

- To use a public package member (classes and interfaces) from outside its package, you must do one of the following
 - Import the package member using import statement
 - Import the member's entire package using import statement
 - Refer to the member by its fully qualified name (without using import statement)

Importing Packages

- To be able to use classes outside of the package you are currently working in, you need to import the package of those classes.
- By default, all your Java programs import the `java.lang.*` package, that is why you can use classes like `String` and `Integers` inside the program even though you haven't imported any packages.
- The syntax for importing packages is as follows:
- **`import <nameOfPackage>;`**

Importing a Class or a Package

```
// Importing a class  
import java.util.Date;
```

```
// Importing all classes in the  
// java.util package
```

```
import java.util.*;
```

"*" stands for classes / interfaces only, not sub-packages if any.

Using Classes of other packages via fully qualified path

```
public static void main(String[] args) {  
    java.util.Date x = new java.util.Date();  
}
```

- ... if `java.util.Date` is not imported.

Package & Directory Structure

- Packages can also be nested.
- Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.
- There should have same directory structure,
./myowndir/myownsubdir/myownpackage directory
for the following package statement

package myowndir.myownsubdir.myownpackage;

Example

- import java.util.Scanner
- Here:
 - **java** is a top level package
 - **util** is a sub package
 - and **Scanner** is a class which is present in the sub package **util**.

Subpackages with *

- The wild card import like `package.*` should be used carefully when working with subpackages.
- For example:
 - we have a package **abc** and inside that package we have another package **foo**, now **foo** is a subpackage.
 - classes inside abc are: Example1, Example 2, Example 3
classes inside foo are: Demo1, Demo2
- `import abc.*;`
 - will only import classes Example1, Example2 and Example3 but it will not import the classes of sub package.
- To import the classes of subpackage you need to
 - `import abc.foo.*;`
 - will import Demo1 and Demo2 but it will not import the Example1, Example2 and Example3.
- To import all the classes present in package and subpackage, we need to use two import statements like this:
 - `import abc.*;`
 - `import abc.foo.*;`

Packages and imports

- A class can have only one package declaration but it can have more than one package import statements.

```
package abcpackage; //This should be one
import xyzpackage;
import anotherpackage;
import anything;
```

Class name conflict

- Lets say we have two packages **abcpackage** and **xyzpackage** and both the packages have a class with the same name: JavaExample.java.
- Now suppose a class import both these packages like this:
 - import abcpackage.*;
 - import xyzpackage.*;
 - This will throw compilation error.
- To avoid such errors you need to use the fully qualified name method
 - abcpackage.JavaExample obj = new abcpackage.JavaExample();
 - xyzpackage.JavaExample obj2 = new xyzpackage.JavaExample();
 - This way you can avoid the import package statements and avoid that name conflict error.

Java static import

- Static import allows you to access the static member of a class directly without using the fully qualified name.

```
import static java.lang.System.out;
import static java.lang.Math.*;
class Demo2{
    public static void main(String args[])
    {
        //instead of Math.sqrt need to use only sqrt
        double var1= sqrt(5.0);
        //instead of Math.tan need to use only tan
        double var2= tan(30);
        //need not to use System in both the below statements
        out.println("Square of 5 is:"+var1);
        out.println("Tan of 30 is:"+var2);
    }
}
```

When to use?

- If you are going to use static variables and methods a lot then it's fine to use static imports.
 - for a code with lot of mathematical calculations, you may want to use static import.
- **Drawback:** It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it
- You can also use wildcard(*) imports.

Managing Source and Class Files

- Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although *The Java Language Specification* does not require this.
- Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is .java. For example:

```
//in the Rectangle.java file  
package graphics;  
public class Rectangle { ... }
```

Managing Source and Class Files

- Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

.....\graphics\Rectangle.java

- **class name** – graphics.Rectangle
- **pathname to file** – graphics\Rectangle.java
- By convention a company uses its reversed Internet domain name for its package names.
- The Example company, whose Internet domain name is example.com, would precede all its package names with com.example.
- Each component of the package name corresponds to a subdirectory.
- So, if the Example company had a com.example.graphics package that contained a Rectangle.java source file, it would be contained in a series of subdirectories like this:
 -\com\example\graphics\Rectangle.java

Managing Source and Class Files

- When you compile a source file, the compiler creates a different output file for each type defined in it.
- The base name of the output file is the name of the type, and its extension is .class.
- The compiled files in the example will be located at:
<path to the parent directory of the output files>\com\example\graphics\Rectangle.class
- Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name.
- However, the path to the .class files does not have to be the same as the path to the .java source files.

Managing Source and Class Files

- By doing this, you can give the classes directory to other programmers without revealing your sources.
- The full path to the classes directory,
 - <path_two>\classes, is called the *class path*, and is set with the CLASSPATH system variable.
 - Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.
- For example, if <path_two>\classes is your class path, and the package name is com.example.graphics, then the compiler and JVM look for .class files in
 - <path_two>\classes\com\example\graphics.

Setting the CLASSPATH System Variable

- A class path may include several paths, separated by a semicolon (Windows) or colon (UNIX). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

To display the current CLASSPATH variable, use these commands in Windows and UNIX (Bourne shell):

In Windows: C:\> set CLASSPATH

In UNIX: % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use these commands:

In Windows: C:\> set CLASSPATH=

In UNIX: % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable, use these commands (for example):

In Windows: C:\> set CLASSPATH=C:\users\george\java\classes

In UNIX: % CLASSPATH=/home/george/java/classes; export CLASSPATH

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Nested Classes

Overview

- Sometimes we define a class whose purpose is only to “help out” another class.
- In that kind of situation, it would be nice if the helper class didn't stand on its own, but instead was more closely associated with the class itself.
 - Like being a part of the associated class

Nested classes

- Two categories: static and non-static.
- Nested classes that are declared static are called *static nested classes*.
- Non-static nested classes are called *inner classes*.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

Nested classes

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*.
- Outer classes can only be declared public or *package private*

Why Use Nested Classes?

- **It is a way of logically grouping classes that are only used in one place**
 - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**
 - Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

- A static nested class interacts with the instance members of its outer class (and other classes) **just like any other top-level class.**
- In fact, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static Nested Classes

- A class can be made **static** only if it is a nested (inner) class.
- Inner static class doesn't need reference of Outer class
- A static class cannot access non-static members of the Outer class

```
Outer class      public class StaticExample {  
                  private static String str = "hello";  
  
Inner class    //Static class  
                static class MyNestedClass{  
                  //non-static method  
                  public void disp() {  
  
                    /* If you make the str variable of outer class  
                     * non-static then you will get compilation error  
                     * because: a nested static class cannot access non-  
                     * static members of the outer class.  
                     */  
                    System.out.println(str);  
                  }  
                }  
                public static void main(String args[])  
                {  
                  /* To create instance of nested class we didn't need the outer  
                   * class instance but for a regular nested class you would need  
                   * to create an instance of outer class first  
                   */  
                  StaticExample.MyNestedClass obj = new StaticExample.MyNestedClass();  
                  obj.disp();  
                }  
              }
```

Inner Classes

- **Inner class** are defined inside the body of another class (known as **outer class**).
- These classes can have access modifier or even can be marked as abstract and final.
- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
- Since an inner class is associated with an instance, it cannot define any static members itself.

Inner Class

- Inner class acts as a member of the enclosing class and can have any access modifiers: abstract, final, public, protected, private, static
- Inner class can access all members of the outer class including those marked private

```
//Top level class definition
class MyOuterClassDemo {
    private int myVar= 1;

    // inner class definition
    class MyInnerClassDemo {
        public void seeOuter () {
            System.out.println("Value of myVar is :" + myVar);
        }
    } // close inner class definition
} // close Top level class definition
```

Private
variables are
accessible

Instantiating an inner class

```
class MyOuterClassDemo {  
    private int x= 1;  
    public void innerInstance()  
{  
        MyInnerClassDemo inner = new MyInnerClassDemo();  
        inner. seeOuter();  
    }  
    public static void main(String args[]){  
        MyOuterClassDemo obj = new MyOuterClassDemo();  
        obj.innerInstance();  
    }  
    // inner class definition  
    class MyInnerClassDemo {  
        public void seeOuter () {  
            System.out.println("Outer Value of x is :" + x);  
        }  
    } // close inner class definition  
} // close Top level class definition
```

- To instantiate an instance of inner class, there should be a live instance of outer class.
- An inner class instance can be created only from an outer class instance.

Output:

```
Outer Value of x is :1
```

Instantiating an inner class from outside the outer class Instance Code

```
class MyOuterClassDemo {  
    private int x= 1;  
    public void innerInstance()  
{  
        MyInnerClassDemo inner = new MyInnerClassDemo();  
        inner. seeOuter();  
    }  
    public static void main(String args[]){  
        MyOuterClassDemo.MyInnerClassDemo inner = new MyOuterClassDemo().new MyInnerClassDemo();  
        inner. seeOuter();  
    }  
    // inner class definition  
    class MyInnerClassDemo {  
        public void seeOuter () {  
            System.out.println("Outer Value of x is :" + x);  
        }  
    } // close inner class definition  
} // close Top level class definition
```

Inner class types

- There are two special kinds of inner classes
 - local classes
 - defined in a *block*, which is a group of zero or more statements between balanced braces.
 - You typically find local classes defined in the body of a method
 - anonymous classes
 - are like local classes except that they do not have a name
 - Anonymous classes enable you to make your code more concise.
 - They enable you to declare and instantiate a class at the same time.
 - Use them if you need to use a local class only once

Local classes

- Local classes are similar to inner classes because they cannot define or declare any static members
- Local classes in static methods can only refer to static members of the enclosing class
- Local classes are non-static because they have access to instance members of the enclosing block
- A local class can have static members provided that they are constant variables

Example

```
//Top level class definition
class MyOuterClassDemo {
    String farewell = "Bye bye";

    public void sayGoodbyeInEnglish() {
        class EnglishGoodbye {

            public void sayGoodbye() {
                System.out.println(farewell);
            }
        }
        EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();
        myEnglishGoodbye.sayGoodbye();
    }
}

} // close Top level class
```

Anonymous Inner Classes

- It is a type of inner class which
 - has no name
 - can be instantiated only once
 - is usually declared inside a method or a code block, a curly braces ending with semicolon
 - Is accessible only at the point where it is defined
 - does not have a constructor simply because it does not have a name
 - cannot be static

Example

```
public class HelloWorld extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) {  
        primaryStage.setTitle("Hello World!");  
        Button btn = new Button();  
        btn.setText("Say 'Hello World'");  
        btn.setOnAction(new EventHandler<ActionEvent>()  
  
            @Override  
            public void handle(ActionEvent event) {  
                System.out.println("Hello World!");  
            }  
        );  
  
        StackPane root = new StackPane();  
        root.getChildren().add(btn);  
        primaryStage.setScene(new Scene(root, 300, 250));  
        primaryStage.show();  
    }  
}
```

btn.setOnAction specifies what happens when you select the button.

This method requires an object of type EventHandler<ActionEvent>. The EventHandler<ActionEvent> interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the btn.setOnAction method.

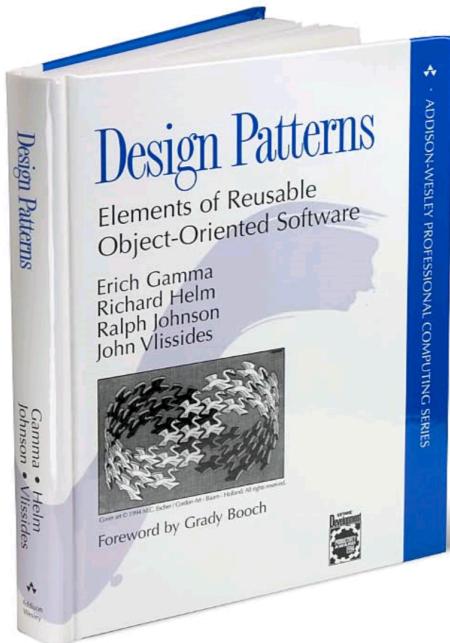
CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Design Patterns

Design Patterns



Gang of Four, 1994

From the Preface

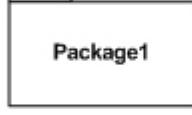
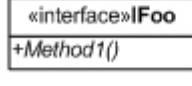
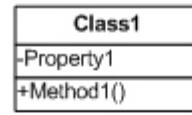
This book isn't an introduction to object-oriented technology or design. Many books already do a good job of that. This book assumes you are reasonably proficient in at least one object-oriented programming language, and you should have some experience in object-oriented design as well. You definitely shouldn't have to rush to the nearest dictionary the moment we mention "types" and "polymorphism," or "interface" as opposed to "implementation" inheritance.

A word of warning and encouragement: Don't worry if you don't understand this book completely on the first reading. We didn't understand it all on the first writing! Remember that this isn't a book to read once and put on a shelf. We hope you'll find yourself referring to it again and again for design insights and for inspiration.

Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

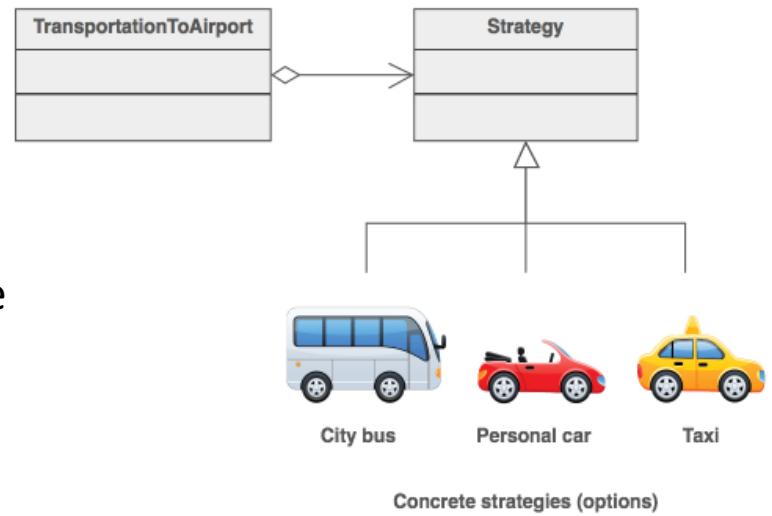
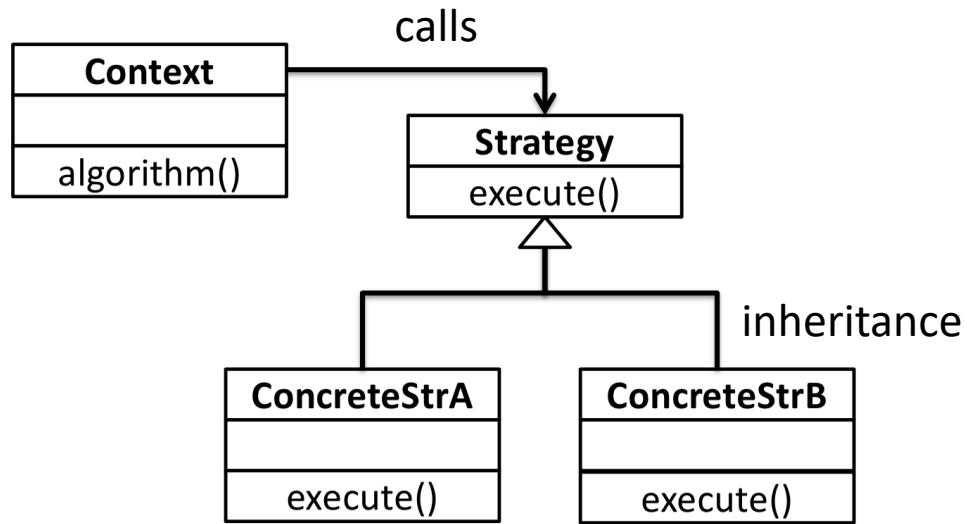
B —————> A	Inheritance - B inherits from A. "is-a" relationship.
B ——————> A	Generalization - B implements A,
A ————— B	Association - A and B call each other
A —————> B	One way Association. A can call B's properties/methods, but not visa versa.
A ◊———— B	Aggregation A "has-a" instance of B. B can survive if A is disposed.
A ♦———— B	Composition A has an instance of B, B cannot exist without A.

	Description
	Package A collection of interfaces and classes.
	Interface Microsoft guidelines specify that interfaces should start with I. This graphic can also sometimes be used as an abstract class.
	Class Properties or attributes sit at the top, methods or operations at the bottom. + indicates public and # indicates protected.

Strategy pattern

- **Problem:** Clients need different variants of an algorithm
- **Solution:** Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- **Consequences:**
 - Easily extensible for new algorithm implementations
 - Separates algorithm from client context
 - Introduces an extra interface and many classes:
 - Code can be harder to understand
 - Lots of overhead if the strategies are simple

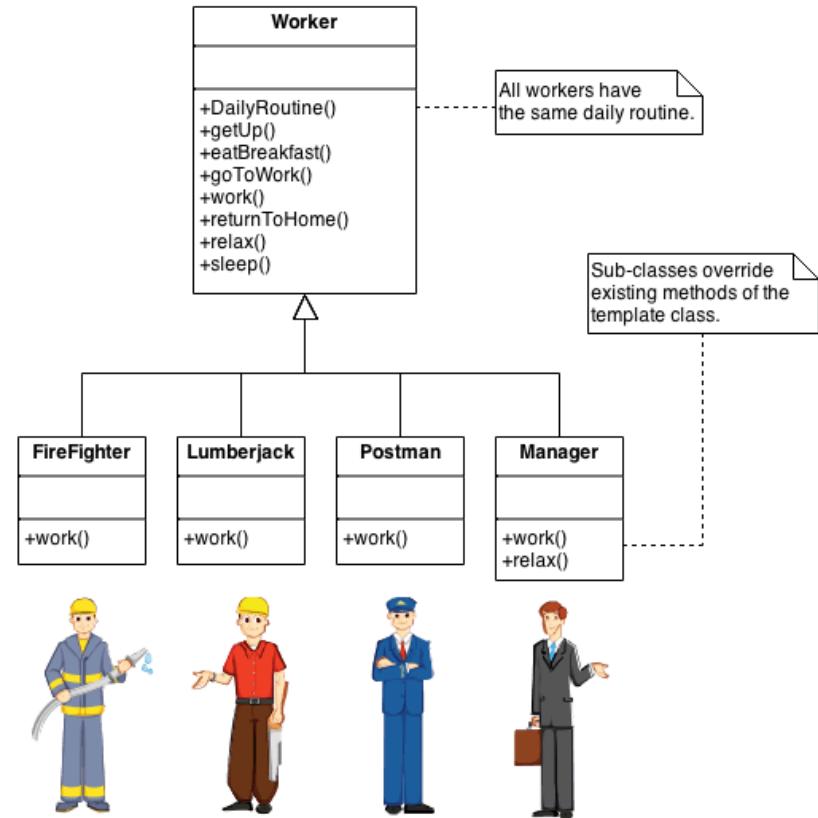
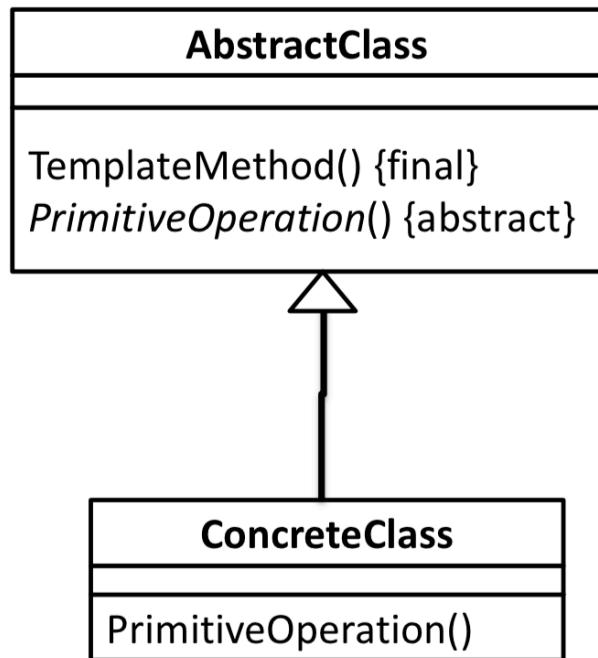
Strategy pattern



Template method pattern

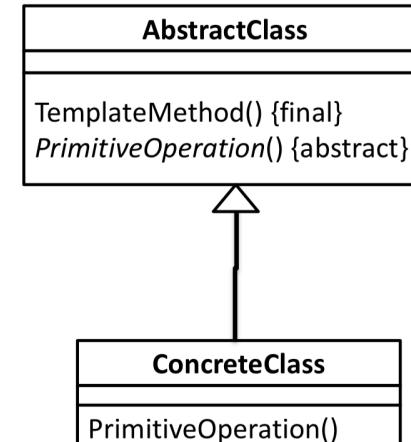
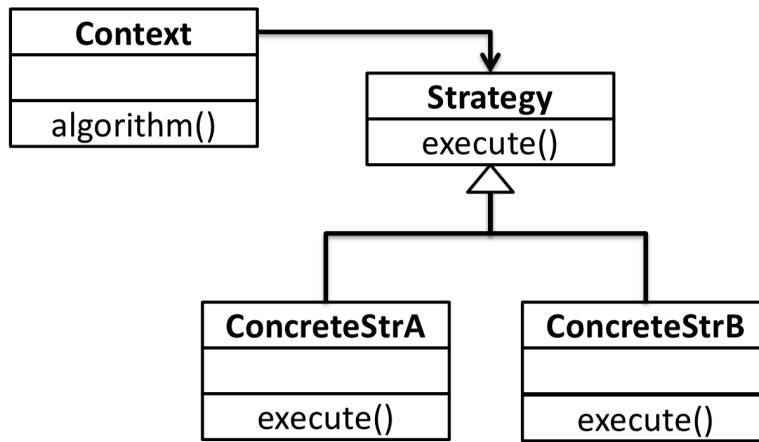
- **Problem:** An algorithm consists of customizable parts and invariant parts
- **Solution:** Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- **Consequences:**
 - Code reuse for the invariant parts of algorithm
 - Customization is restricted to the primitive operations

Template method pattern



Template method vs. the strategy

- Template method uses **inheritance** to vary part of an algorithm
 - Template method implemented in supertype, primitive operations implemented in subtypes
- Strategy pattern uses **delegation** to vary the entire algorithm
 - Strategy objects are reusable across multiple classes
 - Multiple strategy objects are possible per class



Some patterns have the
same structure

Patterns are more than just structure

Command pattern

- **Problem:** Clients need to execute some (possibly flexible) operation without knowing the details of the operation
- **Solution:** Create an interface for the operation, with a class (or classes) that actually executes the operation
- **Consequences:**
 - Separates operation from client context
 - Can specify, queue, and execute commands at different times
 - Introduces an extra interface and classes:
 - Code can be harder to understand
 - Lots of overhead if the commands are simple

This resembles a lot like the Strategy Pattern

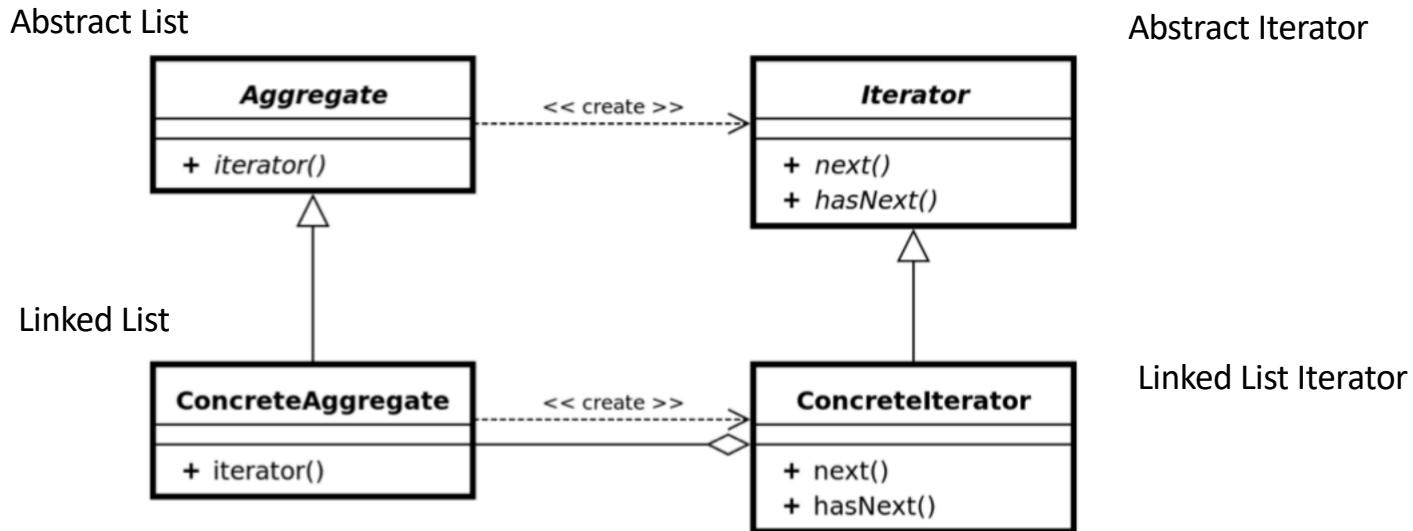
Difference

- The Command pattern is used to make an object out of ***what needs to be done***
 - Execute → teach, study, work
- The Strategy pattern is used to specify ***how something should be done***, and plugs into a larger object or method to provide a specific algorithm
 - Sorting → MergeSort, QuickSort

The Iterator Pattern

- **Problem:** Clients need uniform strategy to access all elements in a container, independent of the container type
 - All items in a list, set, tree; the Fibonacci numbers; all permutations of a set
 - Order is unspecified, but access every element once
- **Solution:** A strategy pattern for iteration
- **Consequences:**
 - Hides internal implementation of underlying container
 - Easy to change container type
 - Facilitates communication between parts of the program

The Iterator Pattern



Traversing a Collection

- Since Java 1.0:

```
List<String> arguments = ...;  
for (int i = 0; i < arguments.size(); ++i) {  
    System.out.println(arguments.get(i));  
}
```

- Java 1.5: for-each loop

```
List<String> arguments = ...;  
for (String s : arguments) {  
    System.out.println(s);  
}
```

- For-each loop works for every implementation of Iterable

```
public interface Iterable<E> {  
    public Iterator<E> iterator();  
}
```

The Iterator Interface

```
public interface java.util.Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); // removes previous returned item  
} // from the underlying collection
```

Fibonacci Iterator

```
class FibIterator implements Iterator<Integer> {  
    public boolean hasNext() {  
  
        public Integer next() {  
  
    }  
    public void remove() {  
  
    }  
}
```

Fibonacci Iterator

```
class FibIterator implements Iterator<Integer> {  
    public boolean hasNext() { return true; }  
    private int a = 1;  
    private int b = 1;  
    public Integer next() {  
        int result = a;  
        a = b;  
        b = a + result;  
        return result;  
    }  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
}
```

A Warning on Iterator

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
 - You will get a ConcurrentModificationException
 - If you simply want to remove an item:

```
List<String> arguments = ...;  
for (Iterator<String> it = arguments.iterator();  
     it.hasNext(); ) {  
    String s = it.next();  
    if (s.equals("Charlie"))  
        arguments.remove("Charlie"); // runtime error  
}
```

A Warning on Iterator

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used

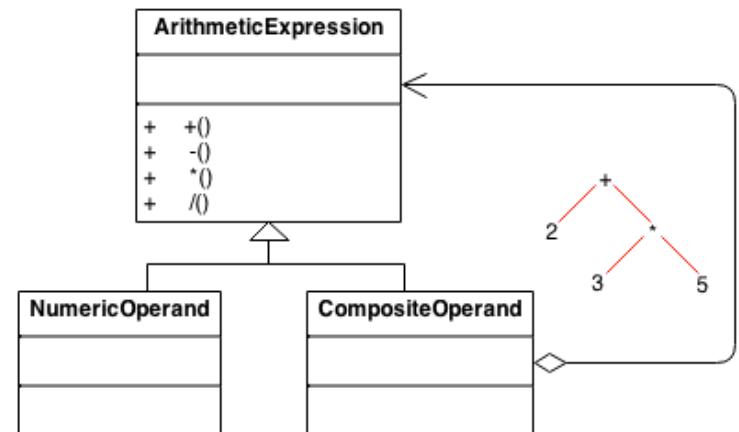
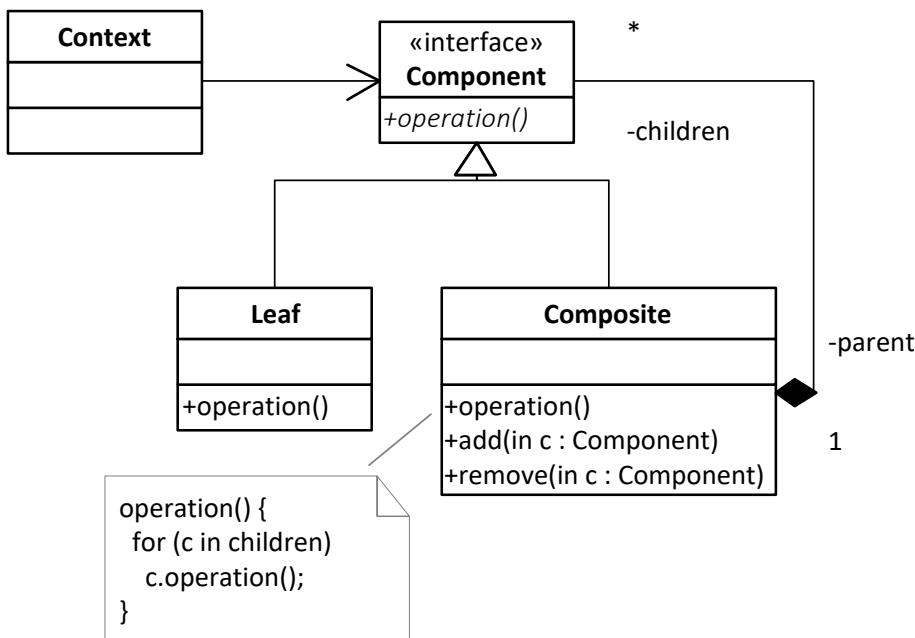
- You will get a ConcurrentModificationException
- If you simply want to remove an item:

```
List<String> arguments = ...;  
for (Iterator<String> it = arguments.iterator();  
     it.hasNext(); ) {  
    String s = it.next();  
    if (s.equals("Charlie"))  
        it.remove();  
}
```

The Composite Design Pattern

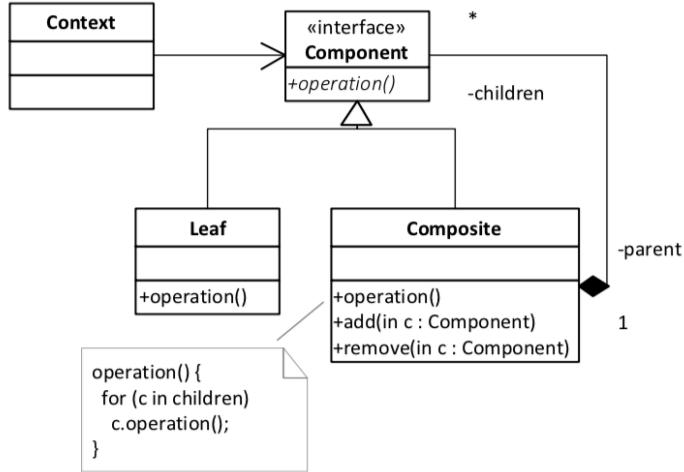
- **Problem:**
 - You want to represent part-whole hierarchies of objects
 - You want to be able to ignore the difference between compositions of objects and individual objects
- **Consequences:**
 - Makes the client simple, since it can treat objects and composites uniformly
 - Makes it easy to add new kinds of components
 - Can make the design overly general
 - Operations may not make sense on every class
 - Composites may contain only certain components

The Composite Design Pattern



Example

```
interface Item {  
    double getWeight();  
}  
  
class Letter implements Item {  
    double weight;  
    double getWeight() {...}  
}  
  
class Box implements Item {  
    ArrayList<Item> items=new ArrayList<>();  
    double getWeight() {  
        double weight = 0.0  
        for(Item item : items) {  
            weight += item.getWeight();  
        }  
    }  
    void add(Item item){  
        items.add(item);  
    }  
}
```

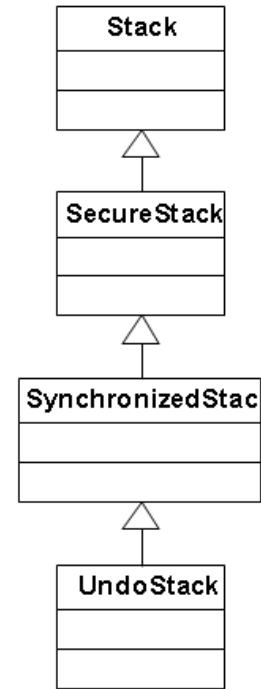
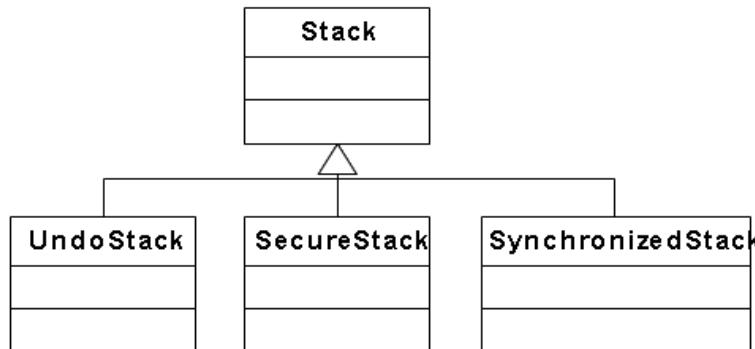


Limitations of inheritance

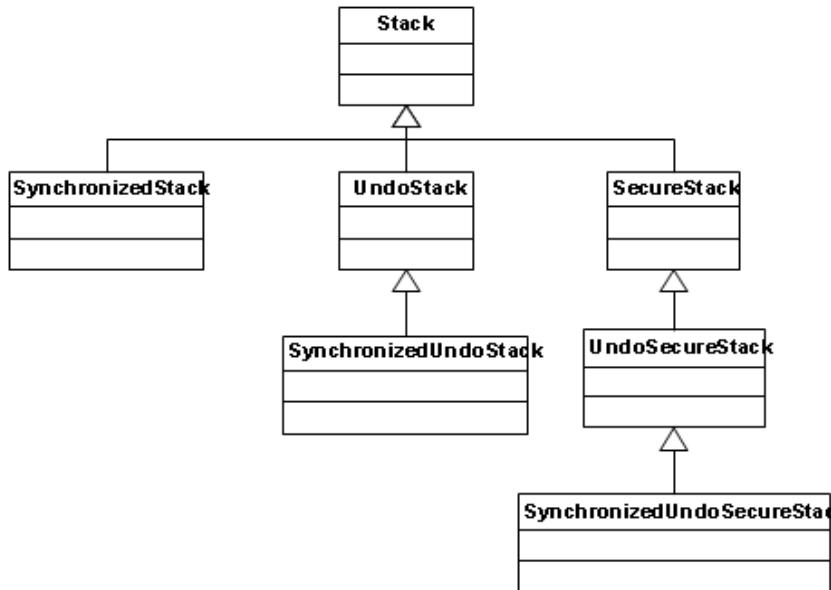
- Suppose you want various extensions of a Stack data structure...
 - UndoStack: A stack that lets you undo previous push or pop operations
 - SecureStack: A stack that requires a password
 - SynchronizedStack: A stack that serializes concurrent accesses
 - **SecureUndoStack: A stack that requires a password, and also lets you undo previous operations**
 - **SynchronizedUndoStack: A stack that serializes concurrent accesses, and also lets you undo previous operations**
 - **SecureSynchronizedStack: ...**
 - **SecureSynchronizedUndoStack: ...**

Goal: arbitrarily composable extensions

Limitations of inheritance

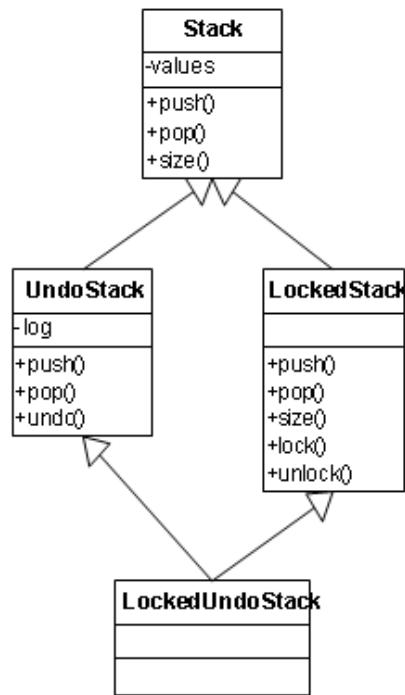


Combining Inheritance Hierarchies?



- Combinatorical explosion
- Massive Code Replication

Multiple Inheritance?



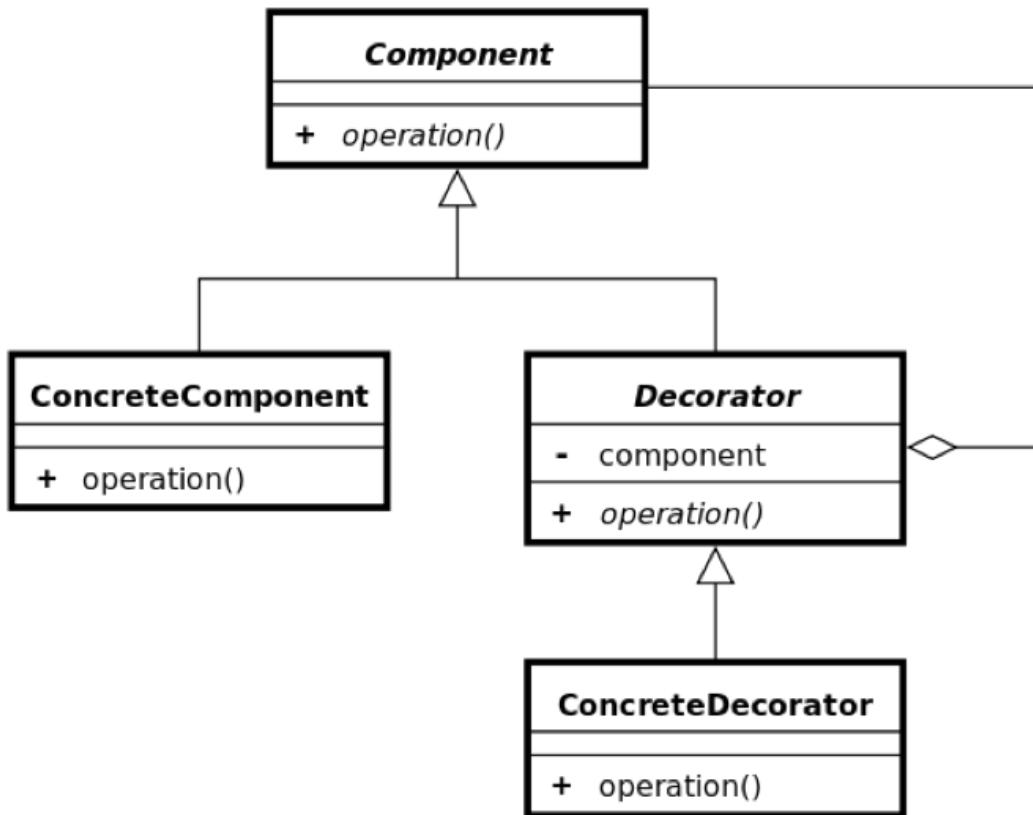
- Diamond Problem

Don't know which feature is inherited from which parent

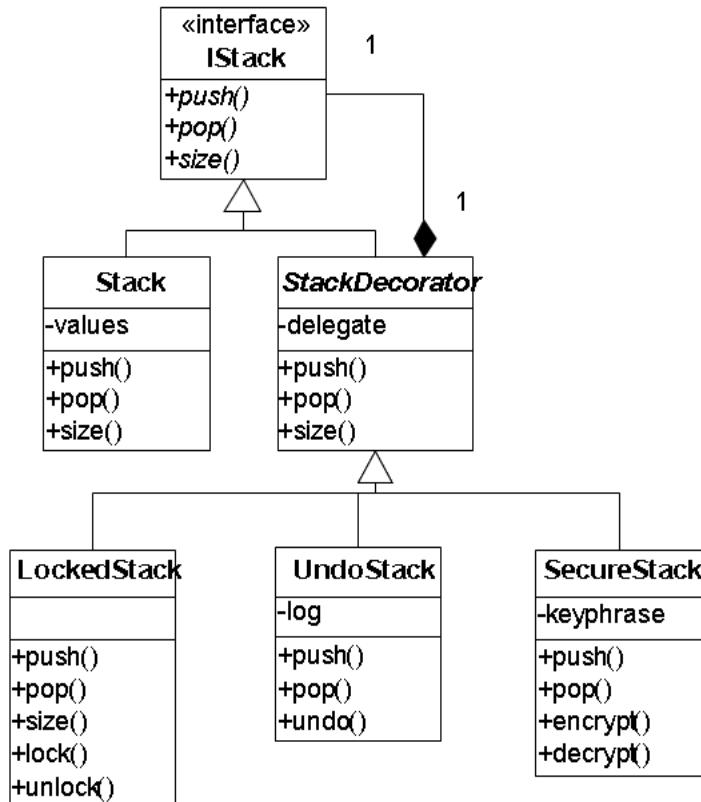
The Decorator Design Pattern

- **Problem:** Need arbitrary / dynamically composable extensions to individual objects.
- **Solution:**
 - Implement common interface as the object you are extending
 - But delegate primary responsibility to an underlying object.
- **Consequences:**
 - More flexible than static inheritance
 - Customizable, cohesive extensions
 - Breaks object identity, self- references

The Decorator Design Pattern



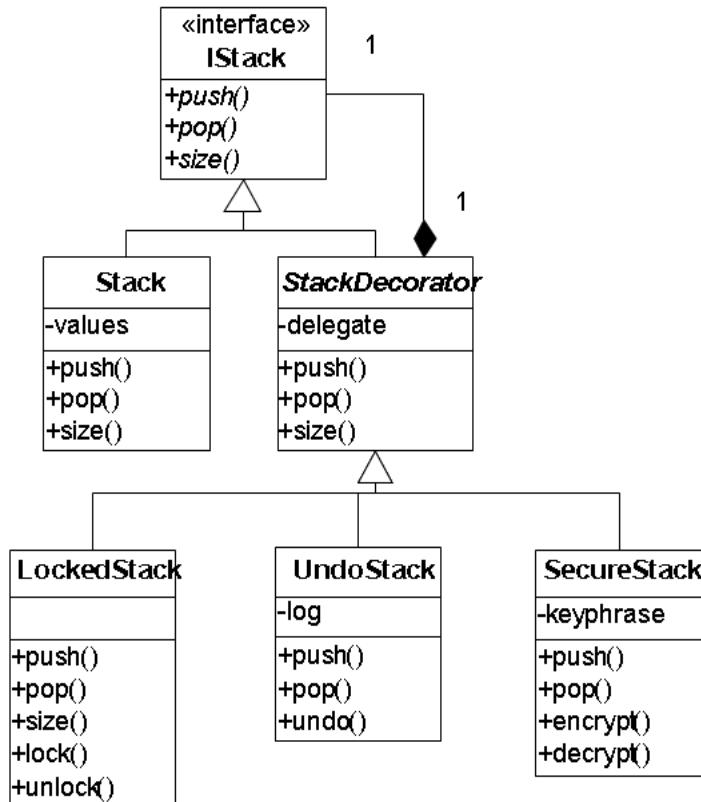
Solving Stack Problem with Decorator



The abstract forwarding class

```
public abstract class StackDecorator
    implements IStack {
    private final IStack stack;
    public StackDecorator(IStack stack) {
        this.stack = stack;
    }
    public void push(Item e) {
        stack.push(e);
    }
    public Item pop() {
        return stack.pop();
    }
    ...
}
```

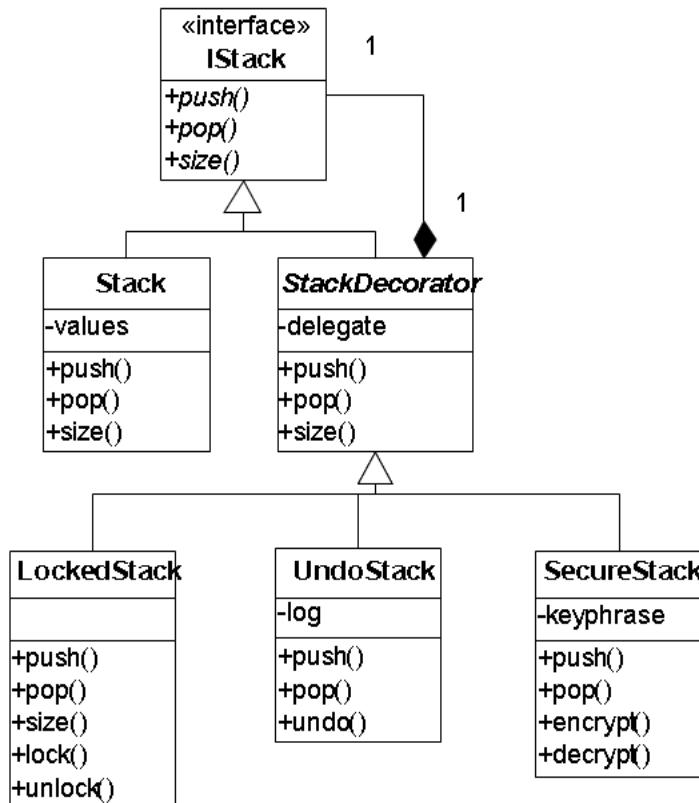
Solving Stack Problem with Decorator



A concrete decorator class

```
public class UndoStack
    extends StackDecorator
    implements IStack {
    private final UndoLog log = new UndoLog();
    public UndoStack(IStack stack) {
        super(stack);
    }
    public void push(Item e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
    }
    ...
}
```

Solving Stack Problem with Decorator



Using the decorator classes

- To construct a plain stack:
`Stack s = new Stack();`
- To construct an plain undo stack:
`UndoStack s = new UndoStack(new Stack());`
- To construct a secure synchronized undo stack:
`SecureStack s = new SecureStack(new SynchronizedStack(new UndoStack(new Stack()))));`

Decorators from java.util.Collections

- Turn a mutable list into an immutable list:

```
static List<T> unmodifiableList(List<T> lst);
static Set<T> unmodifiableSet( Set<T> set);
static Map<K,V> unmodifiableMap( Map<K,V> map);
```

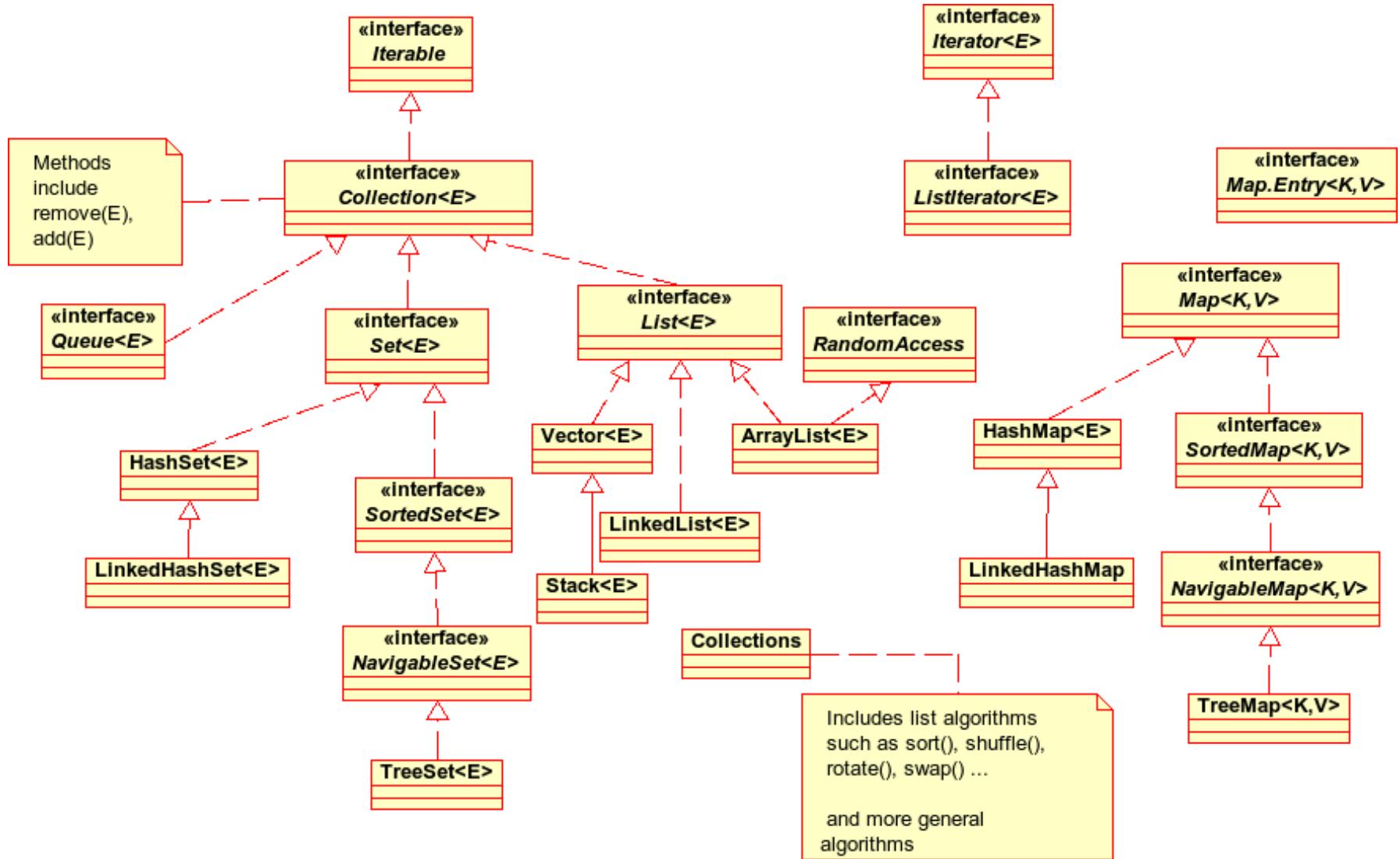
- Similar for synchronization:

```
static List<T> synchronizedList(List<T> lst);
static Set<T> synchronizedSet( Set<T> set);
static Map<K,V> synchronizedMap( Map<K,V> map);
```

The decorator pattern vs. inheritance

- Decorator composes features at runtime
 - Inheritance composes features at compile time
- Decorator consists of multiple collaborating objects
 - Inheritance produces a single, clearly-typed object
- Can mix and match multiple decorations
 - Multiple inheritance has conceptual problems

Java Collections



Designing a data structure library

- Different data types: lists, sets, maps, stacks, queues, ...
- Different representations
 - Array-based lists vs. linked lists
 - Hash-based sets vs. tree-based sets – ...
- Many alternative design decisions
 - Mutable vs. immutable
 - Sorted vs. unsorted
 - Accepts null or not
 - Accepts duplicates or not
 - Concurrency/thread-safe or not – ...

The philosophy of the Collections framework

- Powerful and general
- Small in size and conceptual weight
 - Must feel familiar
 - Only include fundamental operations
 - "Fun and easy to learn and use"
- GOAL:
 - Understand the design challenges of collection libraries.
 - Recognize the design patterns used and how those design patterns achieve design goals.

The Collection Interface

```
public interface Collection<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);          // Optional  
    boolean remove(Object element); // Optional  
    Iterator<E> iterator();  
  
    Object[] toArray();  
    T[] toArray(T a[]);  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); // Optional  
    boolean removeAll(Collection<?> c);   // Optional  
    boolean retainAll(Collection<?> c);   // Optional  
    void clear();                      // Optional  
  
    ...  
}
```

The List interface: an ordered collection

```
public interface List<E> extends Collection<E> {  
    E get(int index);  
    E set(int index, E element);      // Optional  
    void add(int index, E element);   // Optional  
    Object remove(int index);        // Optional  
    boolean addAll(int index, Collection<? extends E> c);  
                                    // Optional  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    List<E> subList(int from, int to);  
  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
}
```

The Set interface: collection of distinct items

```
public interface Set<E> extends Collection<E> {  
}
```

- Adds no methods!
- Specification requires no duplicate items in collection
- The *marker interface* design pattern
 - Marker interfaces add invariants, no code
- Joshua Bloch, "Effective Java (Third edition)," Item 41
Use marker interfaces to define types

TheMap interface: key-value mapping

```
public interface Map<K,V> {  
    int size();  
    boolean isEmpty();  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    Object get(Object key);  
    Object put(K key, V value);    // Optional  
    Object remove(Object key);    // Optional  
    void putAll(Map<? extends K, ? extends V> t); // Opt.  
    void clear();                // Optional  
  
    // Collection views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
}
```

The Collection Interface

```
public interface Collection<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);           // Optional  
    boolean remove(Object element);  // Optional  
    Iterator<E> iterator();  
  
    Object[] toArray();  
    T[] toArray(T a[]);  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); // Optional  
    boolean removeAll(Collection<?> c);   // Optional  
    boolean retainAll(Collection<?> c);   // Optional  
    void clear();                      // Optional  
  
    ...  
}
```

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    // Optional  
}
```

The factory method design pattern

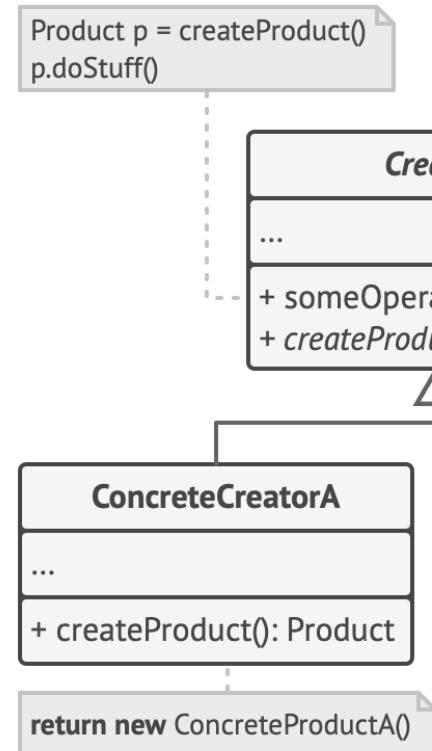
- **Problem:** Subclasses need to control the type of object created
- **Solution:** Define a method that constructs the object; subclasses can override the method.
- **Consequences:**
 - Names can be meaningful, self-documenting
 - Can avoid constructing a new object
 - Might be hard to distinguish factory method from other methods

3

The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.



1

The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2

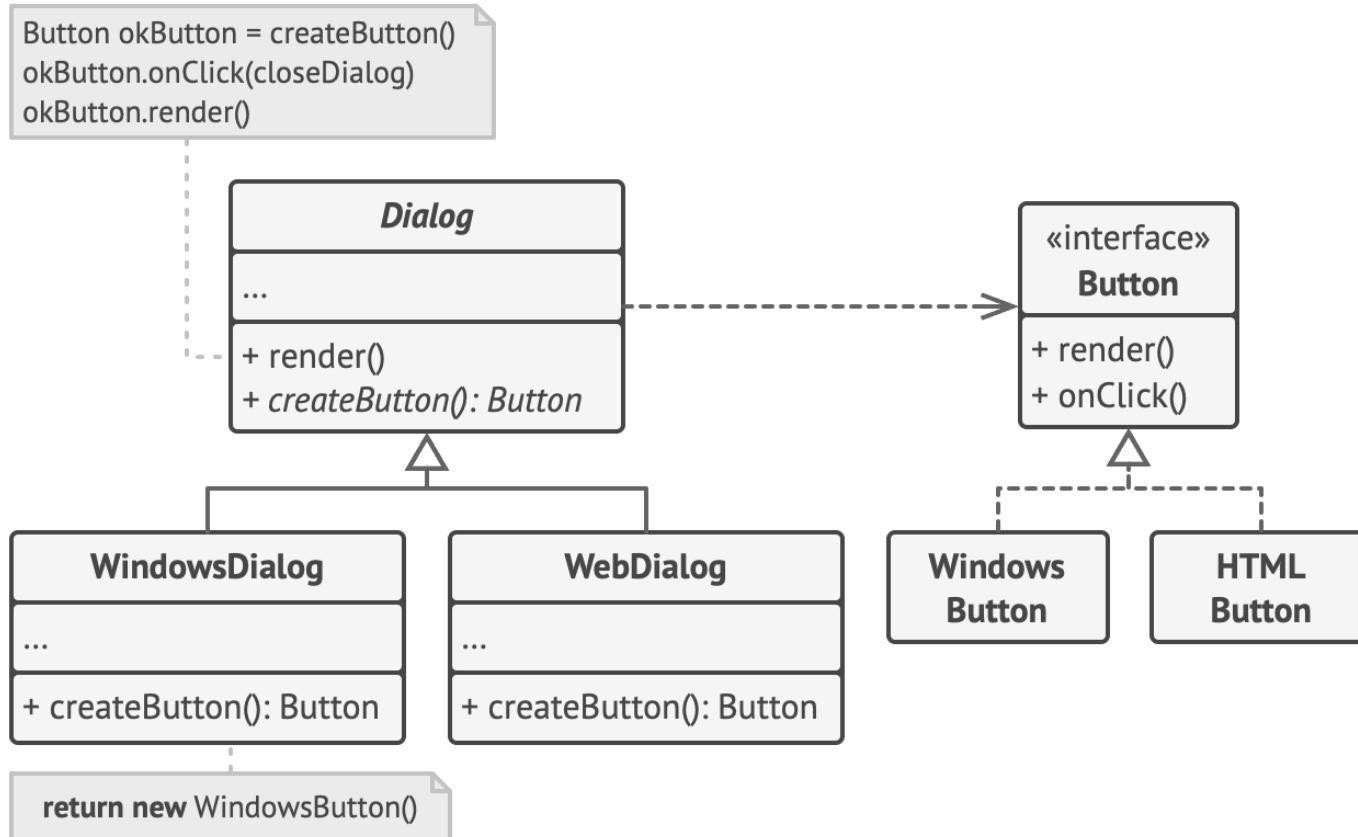
Concrete Products are different implementations of the product interface.

4

Concrete Creators override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

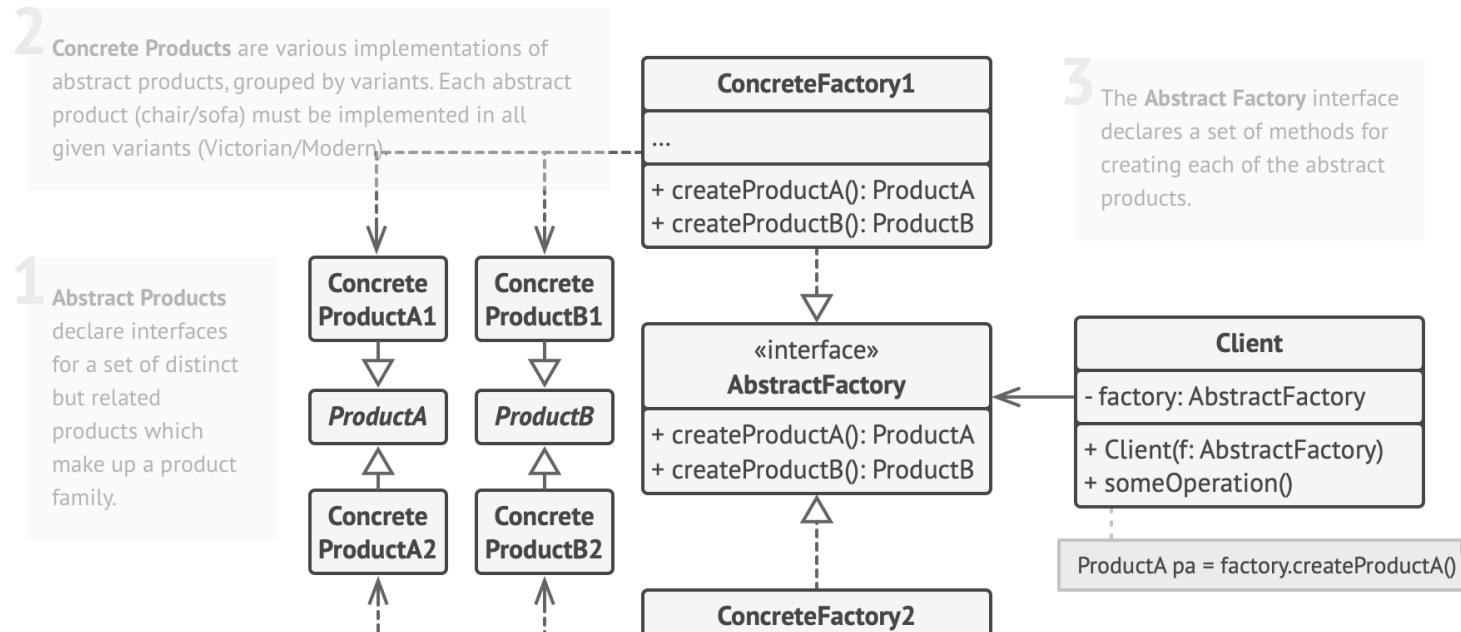
Example



The cross-platform dialog example.

Abstract Factory

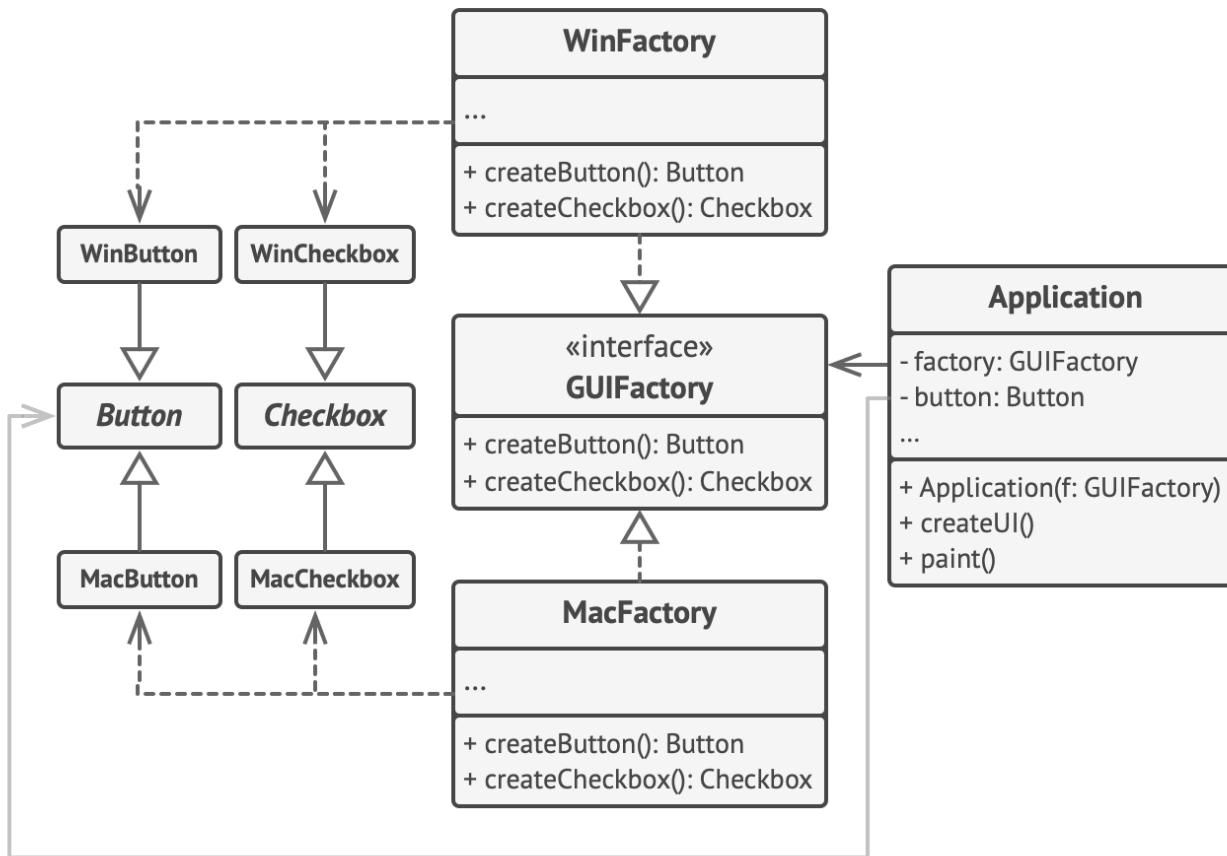
lets you produce families of related objects without specifying their concrete classes.



4 Concrete Factories implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.

5 Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding *abstract* products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

Example



The cross-platform UI classes example.

General-purpose implementations of Collections

Interface	Implementation
Set	HashSet
List	ArrayList
Queue	ArrayDeque
Deque	ArrayDeque
[stack]	ArrayDeque
Map	HashMap

Interface	Implementation(s)
List	LinkedList
Set	LinkedHashSet TreeSet EnumSet
Queue	PriorityQueue
Map	LinkedHashMap TreeMap EnumMap

Wrapper and special-purpose implementations

- Unmodifiable collections (from `java.util.Collections`):

```
Collection<T> unmodifiableCollection(Collection<? extends T> c);  
List<T> unmodifiableList(List<? extends T> list);  
Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m);
```

- Synchronized collections (from `java.util.Collections`):

```
Collection<T> synchronizedCollection(Collection<? extends T> c);  
List<T> synchronizedList(List<? extends T> list);  
Map<K,V> synchronizedMap(Map<? extends K, ? extends V> m);
```

- A List backed from an array (from `java.util.Arrays`):

```
List<T> asList(T... a);
```



The adapter pattern: Returns a specialized list implementation that adapts the array API to the `java.util.List` API

The Adapter Pattern

- **Problem:** You have a client that expects one API for a service provider, and a service provider with a different API
- **Solution:** Write a class that implements the expected API, converting calls to the service provider's actual API
- **Consequences:**
 - Easy interoperability of unrelated clients and libraries
 - Client can use unforeseen future libraries
 - Adapter class is coupled to concrete service provider, can make it harder to override service provider behavior

The UnmodifiableCollection class

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c)
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E>
    implements Collection<E>, Serializable {

    final Collection<E> c;

    UnmodifiableCollection(Collection<> c) {this.c = c; }

    public int         size()           {return c.size();}
    public boolean     isEmpty()        {return c.isEmpty();}
    public boolean     contains(Object o) {return c.contains(o);}
    public Object[]    toArray()        {return c.toArray();}
    public <T> T[]     toArray(T[] a)   {return c.toArray(a);}
    public String      toString()       {return c.toString();}
    public boolean     add(E e)          {throw new UnsupportedOperationException();}
    public boolean     remove(Object o)  {throw new UnsupportedOperationException();}
    public boolean     containsAll(Collection<?> coll) {return c.containsAll(coll);}
    public boolean     addAll(Collection<? extends E> coll) {throw new
                                                               UnsupportedOperationException();}
```

What design pattern
is this?

UnmodifiableCollection
decorates Collection
by removing functionality

Abstract implementations for easy reuse

```
public abstract AbstractList<E> extends AbstractCollection<E>
                                implements List<E> {
    abstract public int size();
    abstract public E get(int index);

    public boolean isEmpty() { return size() == 0; }
    public boolean contains(Object element) { ... }
    public boolean add(int index, E element) { throw new UnsupportedOperationException(); }
    public boolean remove(int index) { throw new UnsupportedOperationException(); }

    ...
}
```

**What design pattern
is this?**

**The template method design pattern:
size and get are primitive operations,
other methods are template methods.**

Reusable algorithms in java.util.Collections

```
static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<?> list);
static void shuffle(List<?> list);
```

Sorting a Collection

```
public static void main(String[] args) {  
    List<String> list = Arrays.asList(args);  
    Collections.sort(list);  
    for (String s : list) {  
        System.out.println(s);  
    }  
}
```

Sorting your own types of objects

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

- General contracts:
 - `a.compareTo(b)` should return:
 - < 0 if `a` is less than `b`
 - 0 if `a` and `b` are equal
 - > 0 if `a` is greater than `b`
 - Should define a total order:
 - If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c)` should be < 0
 - If `a.compareTo(b) < 0`, then `b.compareTo(a)` should be > 0
 - Should usually be consistent with `.equals`:
 - `a.compareTo(b) == 0` iff `a.equals(b)`

Comparable objects – an example

```
public class Integer implements Comparable<Integer> {  
    private final int val;  
    public Integer(int val) { this.val = val; }  
    ...  
    public int compareTo(Integer o) {  
        if (val < o.val) return -1;  
        if (val == o.val) return 0;  
        return 1;  
    }  
}
```

Make Name comparable

```
public class Name {  
    private final String first, last;  
    public Name(String first, String last) {  
        if (first == null || last == null)  
            throw new NullPointerException();  
        this.first = first;  this.last = last;  
    }  
    ...  
}
```

Hint: Strings implement Comparable<String>

Make Name comparable

```
public class Name implements Comparable<Name> {  
    private final String first, last;  
    public Name(String first, String last) {  
        if (first == null || last == null)  
            throw new NullPointerException();  
        this.first = first;  this.last = last;  
    }  
    public int compareTo(Name o) {  
        int lastComparison = last.compareTo(o.last);  
        if (lastComparison != 0) return lastComparison;  
        return first.compareTo(o.first);  
    }  
}
```

Alternative comparisons

```
public class Employee implements Comparable<Employee> {  
    protected Name name;  
    protected int salary;  
    ...  
}
```

- What if we want to sort Employees by name, usually, but sometimes sort by salary?
- Answer: There's a Strategy pattern interface for that:

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

Writing a Comparator object

```
public class Employee implements Comparable<Employee> {  
    protected Name name;  
    protected int salary;  
    public int compareTo(Employee o) {  
        return name.compareTo(o.name);  
    }  
}  
  
public class SalaryComparator implements Comparator<Employee> {  
    public int compare (Employee o1, Employee o2) {  
        return o1.salary - o2.salary;  
    }  
}
```

Summary

- Collections as reusable and extensible data structures
 - – design for reuse
 - – design for change
- Iterators to abstract over internal structure
- Decorator to attach behavior at runtime
- Template methods and factory methods to support customization in subclasses
- Adapters to convert between implementations

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Lecture 5 - Part 1 Java Generics

Motivation

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Provide a way to re-use the same code with different inputs
- Benefits
 - Stronger type checks at compile time
 - Fixing compile-time errors is easier than fixing runtime errors
 - Elimination of casts
 - Enabling programmers to implement generic algorithms

Prevention of casting

- Without the generics

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- With generics

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);    // no cast
```

Box class

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

- Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types.
- There is no way to verify, at compile time, how the class is used.
- One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

Generic Version of Box

```
/**  
 * Generic version of the Box class.  
 * @param <T> the type of the value being boxed  
 */  
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

- all occurrences of Object are replaced by T.
- A type variable can be any **non-primitive** type you specify: any class, interface, array, etc.

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
 - Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
 - E - Element (used extensively by the Java Collections Framework)
 - K - Key
 - N - Number
 - T - Type
 - V - Value
 - S,U,V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

- `Box<Integer> integerBox;`
- To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:
 - `Box<Integer> integerBox = new Box<Integer>();`
- In Java SE 7 and later, you can invoke the constructor of a generic class with an empty set of type arguments (`<>`) as long as the compiler can determine the type arguments from the context.
- This pair of angle brackets, `<>`, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:
 - `Box<Integer> integerBox = new Box<>();`

Generic Methods

- public static <T> T best(List<T> entries, ...) { ... }
- This says that the best method takes a List of T's and returns a T
- The <T> at the beginning means T is not a real type, but a type that Java will figure out from the method call
- Java will figure out the type of T by looking at parameters to the method call

```
List<Person> people = ...;
```

```
Person bestPerson = Utils.best(people, ...);
```

```
List<Car> cars = ...;
```

```
Car bestCar = Utils.best(cars, ...);
```

Example

```
public class RandomUtils {  
    ...  
  
    public static <T> T randomElement(T[] array) {  
        return (array[randomIndex(array)]);  
    }  
}
```

- In rest of method, T refers to a type.
- Java will figure out what type T is by looking at the parameters of the method call.
- Even if there is an existing class actually called T, it is irrelevant here.

This says that the method takes in an array of T's and returns a T. For example, if you pass in an array of Strings, you get out a String; if you pass in an array of Employees, you get out an Employee. No typecasts required in any of the cases.

You can limit T as <T extends Number>

Then, it can be the class (or interface) Number or any of its subclass.

Example

```
public static <T> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e > elem)  
            ++count;  
    return count;  
}
```

- the greater than operator (`>`) applies only to primitive types such as short, int, double, long, float, byte, and char.
- You cannot use the `>` operator to compare objects.
- To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

Correct Code

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e.compareTo(elem) > 0)  
            ++count;  
    return count;  
}
```

Generics, Inheritance, and Subtypes

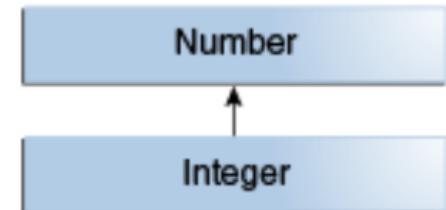
- it is possible to assign an object of one type to an object of another type provided that the types are compatible

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; // OK
```

- This is called an "is a" relationship. Since an Integer *is a* kind of Object, the assignment is allowed. Integer is also a kind of Number, so the following code is valid as well

```
public void someMethod(Number n) { /* ... */ }

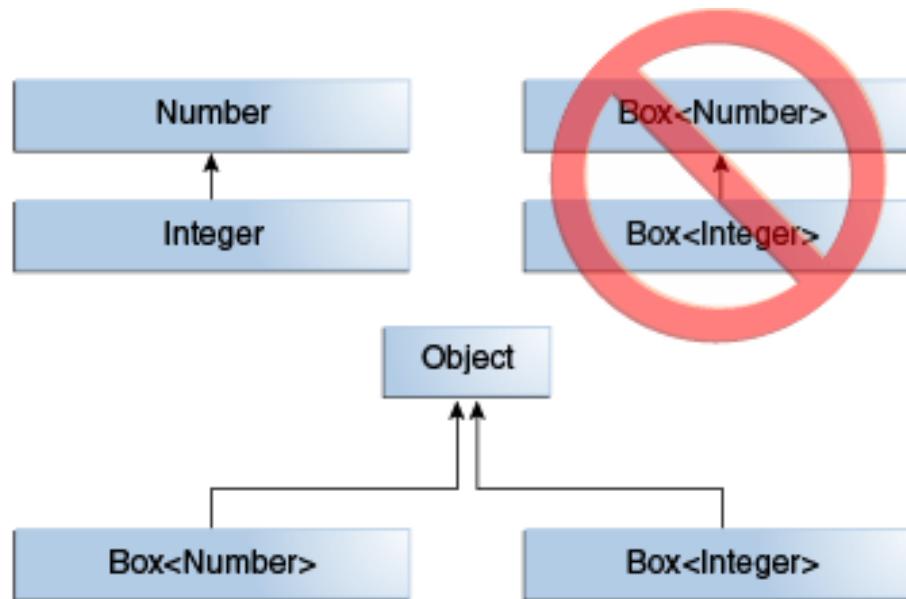
someMethod(new Integer(10)); // OK
someMethod(new Double(10.1)); // OK
```



- The same is also true with generics

Example

- public void boxTest(Box<Number> n) { /* ... */ }
- Are you allowed to pass in Box<Integer> or Box<Double>?



Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass, regardless of whether or not A and B are related.

Generic Classes or Interfaces

```
public class ArrayList<E> {  
  
    public E get(int index) { ... }  
  
    ...  
}  
  
...  
  
public boolean add(E element) { ... }
```

This says that get returns an E. So, if you created `ArrayList<Employee>`, get returns an Employee. No typecast required in the code that calls get.

This says that add takes an E as a parameter. So, if you created `ArrayList<Circle>`, add can take only a Circle.

In rest of class, E does not refer to an existing type. Instead, it refers to whatever type was defined when you created the list. E.g., if you did `ArrayList<String> words = ...;` then E refers to String.

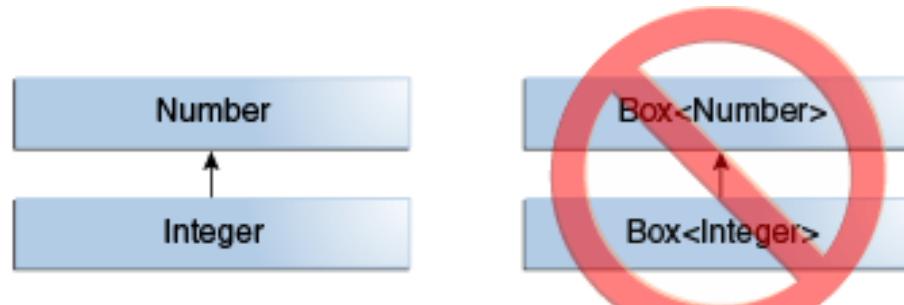
Wildcards

- In generic code, the question mark (?), called the *wildcard*, represents an unknown type
- **Upper Bounded Wildcards**
 - `public static void process(List<? extends Foo> list) { /* ... */ }`
- The upper bounded wildcard, `<? extends Foo>`, where Foo is any type, matches Foo and any subtype of Foo.
- The process method can access the list elements as type Foo:

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) { // ... }
}
```

Wildcards

- **Lower Bounded Wildcards**
 - A lower bounded wildcard is expressed using the wildcard character ('?'), followed by the super keyword, followed by its *lower bound*: <? super A>.
- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.
- **Unbounded Wildcards**
 - List<Object> and List<?> are not the same.



What does Effective Java think about Generics?

- Positive things in general!

5 Generics.....

Item 26: Don't use raw types
Item 27: Eliminate unchecked warnings.....
Item 28: Prefer lists to arrays
Item 29: Favor generic types.....
Item 30: Favor generic methods
Item 31: Use bounded wildcards to increase API flexibility .
Item 32: Combine generics and varargs judiciously.....
Item 33: Consider typesafe heterogeneous containers

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Lecture 5 - Part 2 Reflection

Reflection

- In computer science, it is the process by which a computer program can observe and modify its own structure and behavior.
- The programming paradigm driven by reflection is called reflective programming.
 - an extension to the object-oriented programming paradigm
 - to add self-optimization to application programs, and
 - to improve their flexibility

Reflection in Java

- Allows you to find out information about any object, including its methods and fields, at run time.
- Reflection Can be Used To
 - construct new class instances and new arrays
 - access and modify fields of objects and classes
 - invoke methods on objects and classes
 - access and modify elements of arrays

Reflection API

- `java.lang.reflect`, offers programmatic access to arbitrary classes.
- Given a `Class` object, you can obtain `Constructor`, `Method`, and `Field` instances representing the constructors, methods, and fields of the class represented by the `Class` instance. These objects provide programmatic access to the class's member names, field types, method signatures, and so on.
- `Constructor`, `Method`, and `Field` instances let you manipulate their underlying counterparts *reflectively*:
 - you can construct instances, invoke methods, and access fields of the underlying class by invoking methods on the `Constructor`, `Method`, and `Field` instances.
 - For example, `Method.invoke` lets you invoke any method on any object of any class (subject to the usual security constraints).
 - Reflection allows one class to use another, even if the latter class did not exist when the former was compiled.

The cost of reflection

- You lose all the benefits of compile-time type checking
 - including exception checking. If a program attempts to invoke a nonexistent or inaccessible method reflectively, it will fail at runtime
- Performance suffers
 - Reflective method invocation is much slower than normal method invocation
- The code required to perform reflective access is clumsy and verbose
 - It is tedious to write and difficult to read

```
Class c1 = Class.forName( className );
Class[] paramTypes = new Class[] { String[].class };
Method m = c1.getDeclaredMethod( "main", paramTypes );
Object[] args = new Object[]
{ new String[] { "Breathing", "Fire" } }
m.invoke( null, args );
```

```
Class c1 = Class.forName("Villain");
Class[] paramTypes = new Class[] {String.class,
Integer.TYPE };
Constructor m = c1.getConstructor( paramTypes );
Object[] arguments = new Object[] { "Darth Vader",
new Integer(20) };
Villan v = (Villan) m.newInstance(arguments);
```

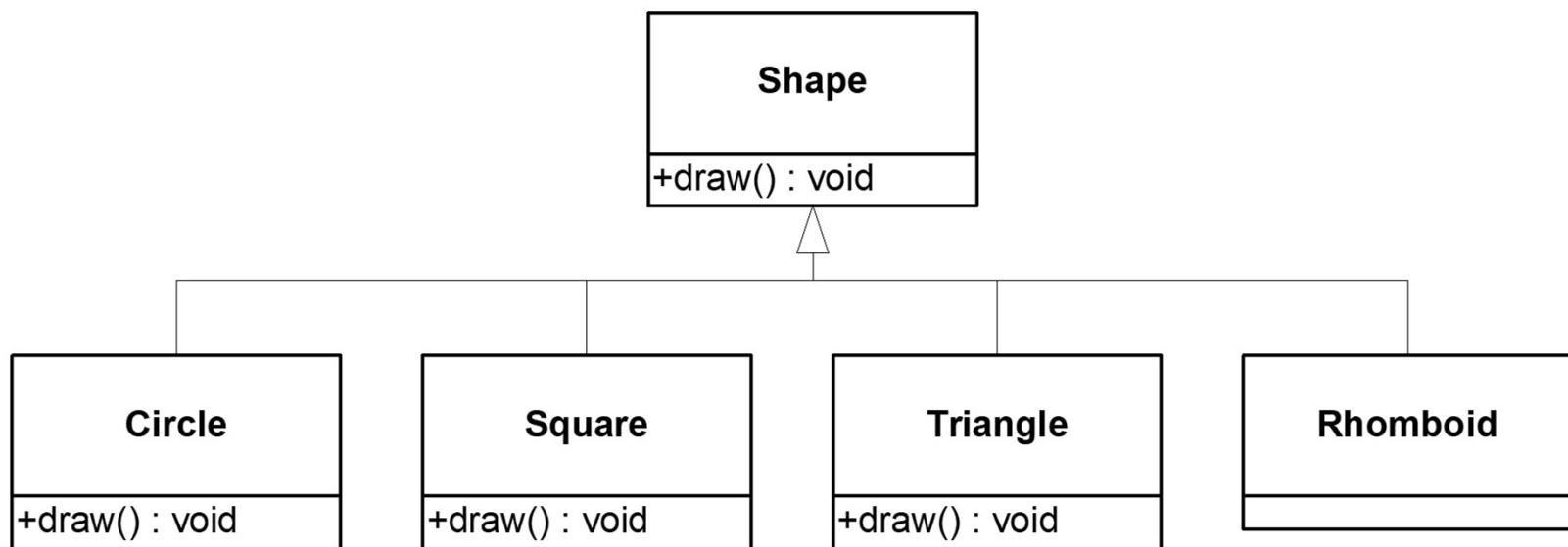
Where to use it

- There are a few sophisticated applications that require reflection.
- Examples include code analysis tools and dependency injection frameworks.
 - Even such tools have been moving away from reflection of late, as its disadvantages become clearer.
- If you have any doubts as to whether your application requires reflection, it probably doesn't.

Item 65: Prefer interfaces to reflection

Example

- Shape objects to instantiate and manipulate



Factory Method Design Pattern

- **Problem:** Subclasses need to control the type of object created
- **Solution:** Define a method that constructs the object; subclasses can override the method.

Factory Method without Reflection

```
public static Shape createShape(String s) {
    Shape temp = null;
    if (s.equals("Circle"))
        temp = new Circle();
    else
        if (s.equals("Square"))
            temp = new Square();
        else
            if (s.equals("Triangle"))
                temp = new Triangle();
            else
                // ...
                // continues for each kind of shape
    return temp;
}
```

- switch-case or cascading if-else statements should scream “redesign me” to the developer
 - Eliminate the switch/case statement
 - Consider a dynamic, better, approach

Factory Method with Reflection

```
public static Shape createShape(String s) {  
    Shape temp = null;  
    try {  
        temp = (Shape) Class.forName(s).newInstance();  
    }  
    catch (Exception e) {}  
    return temp;  
}
```

- Every time you need an instance of some Shape, you have to call
 - Class.forName(s).newInstance()
- which is really a slow process
- Performance problem when you call it many times!
- Any solution?

Factory Object with Reflection

- Given some factory object (per class) which knows how to create some Shape

```
class SquareFactory {  
    public Shape createShape () {  
        return new Square ();  
    }  
}
```

- Now all you have to do is
 - instantiate only one instance of SquareFactory, and
 - call its createShape method whenever you need some Square
- No such performance problem left!
- But, you need to define a XxxFactory class for each of the Xxx class extending Shape

Factory Object with Reflection

- Instantiating only one instance of XxxFactory

```
public static ShapeFactory createShapeFactory(String s) {  
    Shape temp = null;  
    try {  
        temp = (ShapeFactory) Class.forName(s).newInstance();  
    }  
    catch(Exception e){}  
    return temp;  
}
```

- Call its createShape method whenever you need some Shape

```
ShapeFactory squareFactory=createShapeFactory("SquareFactory");  
...  
Shape square=squareFactory.createShape();
```

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Hande Alemdar

Fall 2020

Lecture 5 - Part 3 Exceptions

Exceptions

- The Java programming language uses *exceptions* to handle errors and other exceptional events.
- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Exceptions in Java separates error handling from main business logic
- Java has a uniform approach for handling all errors
 - From very unusual (e.g. out of memory) to more common ones your program should check itself (e.g. index out of bounds)
 - From Java run-time system errors (e.g., divide by zero) to errors that programmers detect and raise deliberately

Throwing and catching

- An error case can throw an exception
 - `throw <exception object>;`
- By default, exceptions result in the thread terminating after printing an error message
- However, exception handlers can catch specified exceptions and recover from error
 - `catch (<exception type> e) {
 //statements that handle the exception
}`

Exceptional flow of control

- Exceptions break the normal flow of control.
- When an exception occurs, the statement that would normally execute next is not executed.
- What happens instead depends on:
 - whether the exception is caught,
 - where it is caught,
 - what statements are executed in the ‘catch block’,
 - and whether you have a ‘finally block’.

Approaches to handling an exception

1. Prevent the exception from happening
2. Catch it in the method in which it occurs, and either
 - a. Fix up the problem and resume normal execution
 - b. Fix partially and re-throw it *
 - c. Handle it then throw a different exception *
3. Do not catch it *
4. With 1. and 2.a. the caller never knows there was an error.
5. With 2.b., 2.c., and 3., the caller must handle it.
6. If no one from the method where exception occurred to main catches the exception, the program will terminate and display a stack trace.

* Then you declare that the method may throw an exception

Catching an exception

```
try { // statement that could throw an exception
    }
catch (<exception type> e) {
    // statements that handle the exception
}
catch (<exception type> e) { //e higher in hierarchy
    // statements that handle the exception
}
finally {
    // release resources
}
//other statements
```

- At most one catch block executes
- finally block always executes once

Execution of try catch blocks

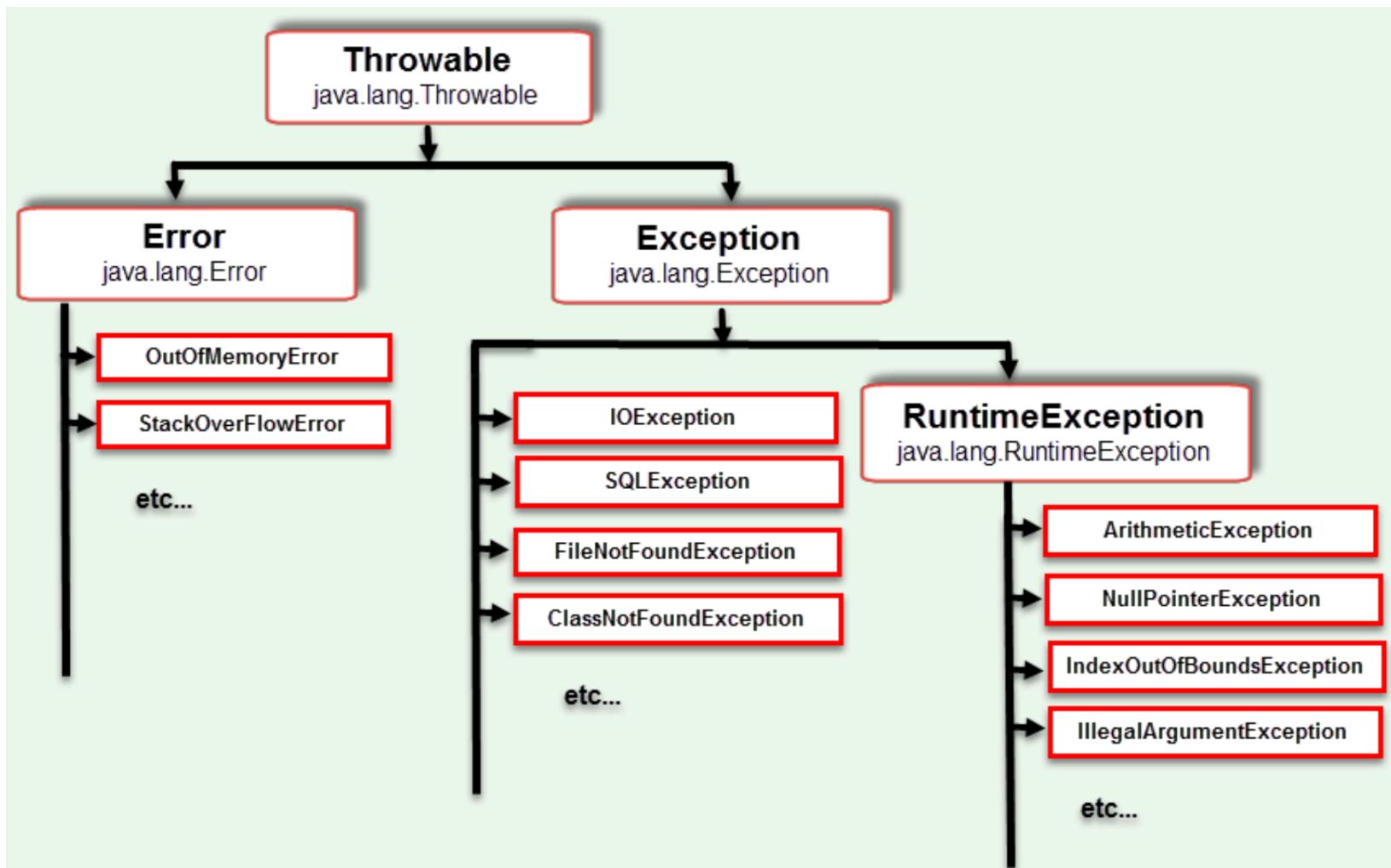
- For normal execution:
 - try block executes, then finally block executes, then other statements execute
- When an error is caught and the catch block throws an exception or returns:
 - try block is interrupted
 - catch block executes (until throw or return statement)
 - finally block executes
- When error is caught and catch block doesn't throw an exception or return:
 - try block is interrupted
 - catch block executes
 - finally block executes
 - other statements execute
- When an error occurs that is not caught:
 - try block is interrupted
 - finally block executes

**finally block is
always executed
unless you call
System.exit()**

Catch processing

- When an exception occurs, the catch statements are searched top-to-bottom & inner-to-outer order for a catch parameter matching the exception class
- A catch parameter is said to match the exception if it:
 - is the same class as the exception; or
 - is a superclass of the exception; or
 - if the parameter is an interface, the exception class implements the interface.
- The first try/catch statement that has a parameter that matches the exception has its catch statement executed.
- After the catch statement executes, execution resumes with the finally statement, if any, then the statements after the try/catch statement.

Exception hierarchy



Java is strict

- Unlike C++, is quite strict about catching exceptions
- If it is a **checked exception** Java compiler forces the caller must either catch it or explicitly declare that it may result in an exception.
- By enforcing this, Java guarantees exception correctness at compile time.
- Here's a method that ducks out of catching an exception by explicitly re-throwing it:
 - `void f() throws tooBig, tooSmall, divZero { }`
 - The caller of this method now must either catch these exceptions or declare them in its specification as it may result in them when it calls `f()`.

Checked vs Unchecked

- Checked exceptions are checked at compile-time.
 - if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.
- Unchecked exceptions are not checked at compile time
 - If your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error.
 - Most of the times these exception occurs due to the bad data provided by user during the user-program interaction.
 - It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately.
 - All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Example

```
import java.io.*;
class Example {
    public static void main(String args[])
    {
        FileInputStream fis = null;
        try{
            fis = new FileInputStream("B:/myfile.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("The specified file is not " +
                               "present at the given path");
        }
        int k;
        try{
            while(( k = fis.read() ) != -1)
            {
                System.out.print((char)k);
            }
            fis.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
        }
    }
}
```

will not compile

Option 1 : throw it

Option 2: handle it

Unchecked Exception Example

```
class Example {  
    public static void main(String args[])  
{  
    int num1=10;  
    int num2=0;  
    /*Since I'm dividing an integer with 0  
     * it should throw ArithmeticException  
     */  
    int res=num1/num2;  
    System.out.println(res);  
}  
}
```

If you compile this code, it would compile successfully however when you will run it, it would throw `ArithmaticException`.

Unchecked Exception Example

```
class Example {  
    public static void main(String args[])  
{  
        int arr[] ={1,2,3,4,5};  
        /* My array has only 5 elements but we are trying to  
         * display the value of 8th element. It should throw  
         * ArrayIndexOutOfBoundsException  
         */  
        System.out.println(arr[7]);  
    }  
}
```

It **doesn't mean** that compiler is not checking these exceptions so we shouldn't handle them.

```
class Example {  
    public static void main(String args[]) {  
        try{  
            int arr[] ={1,2,3,4,5};  
            System.out.println(arr[7]);  
        }  
        catch(ArrayIndexOutOfBoundsException e){  
            System.out.println("The specified index does not exist " +  
                "in array. Please correct the error.");  
        }  
    }  
}
```

Important notes

- All possible
 - try-catch
 - try-catch-catch...
 - try-catch-finally
 - try-catch-catch--finally
 - try-finally
 - Nesting try-catch blocks
- The circumstances that prevent execution of the code in a finally block are:
 - The death of a Thread
 - Using of the System.exit() method.
 - Due to an exception arising in the finally block.

```
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
        try {
            in.close();
        }catch (IOException exp) {
            System.out.println(exp);
        }
    }
}
```

Example

```
class Example2{
    public static void main(String args[]){
        try{
            System.out.println("First statement of try block");
            int num=45/0;
            System.out.println(num);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException");
        }
        finally{
            System.out.println("finally block");
        }
        System.out.println("Out of try-catch-finally block");
    }
}
```

```
First statement of try block
finally block
Exception in thread "main" java.lang.ArithmetricException: / by zero
```

User defined exception in java

```
class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

public class Example1
{
    void productCheck(int weight) throws InvalidProductException{
        if(weight<100){
            throw new InvalidProductException("Product Invalid");
        }
    }

    public static void main(String args[])
    {
        Example1 obj = new Example1();
        try
        {
            obj.productCheck(60);
        }
        catch (InvalidProductException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
    }
}
```

try-with-resources

```
// try-finally is ugly when used with more than one resource!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

Hande Alemdar

Fall 2020

Lecture 6 - Principles of Object-Oriented Design

The Object-Oriented Paradigm

- What is object-oriented (OO) paradigm?
 - provide a set of techniques for analysing, decomposing, and modularising software system architectures
 - characterized by structuring the system architecture on the basis of its *objects* (and classes of objects) rather than the *actions* it performs
- What is the rationale for using OO?
 - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time
 - Use it for **large systems**
 - Use it for **systems that change often**

Signs of Rotting Design

1. Rigidity

- code difficult to change
- every change causes a cascade of subsequent changes in dependent modules
- management reluctance to change anything becomes policy

2. Fragility

- even small **changes** can cause cascading effects
- code breaks in unexpected places

3. Immobility

- code is so tangled that it's impossible to **reuse** anything
- the software is simply rewritten instead of reused

Signs of Rotting Design

4. Viscosity

- **viscosity of the design**

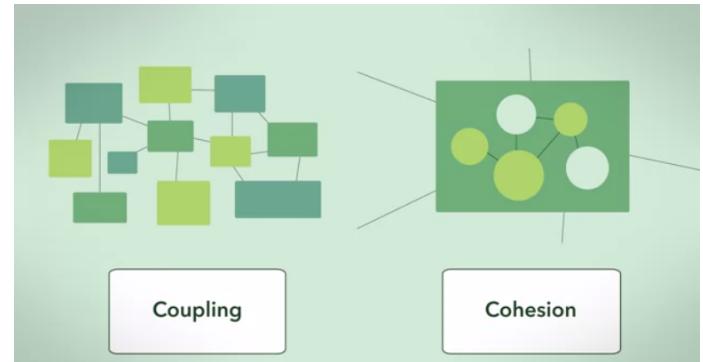
- When the design preserving changes are harder to employ than the hacks, then the viscosity of the design is high.
- It is easy to do the wrong thing, but hard to do the right thing.

- **viscosity of the environment**

- comes about when the development environment is slow and inefficient
- if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view

Causes of Rotting Design

- Dependency Management
 - coupling and cohesion
 - Coupling refers to the interdependencies between modules
 - Loose coupling vs tight coupling
 - Cohesion describes how related the functions within a single module are.
 - Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.
 - It can be controlled!
 - create *dependency* firewalls so that dependencies do not propagate



SOLID Principles

- SRP – Single Responsibility Principle
 - A class should have only one reason to change
- OCP – Open / Closed Principle
 - Software entities should be open for extension, but closed for modification
- LSP – Liskov Substitution Principle
 - Inheritance should ensure that any property proved about supertype objects also holds for subtype objects
- ISP – Interface Segregation Principle
 - Clients should not be forced to depend on methods that they do not use.
- DIP – Dependency Inversion Principle
 - High-level modules should not depend on low-level modules.

From: Agile Software Development: Principles, Patterns, and Practices. Robert C. Martin, Prentice Hall, 2002

Single Responsibility Principle (SRP)

- A class should have only one reason to change



every module or class should have **responsibility** over a **single** part of the functionality provided by the software, and that **responsibility** should be entirely encapsulated by the class.

Responsibility and Cohesion

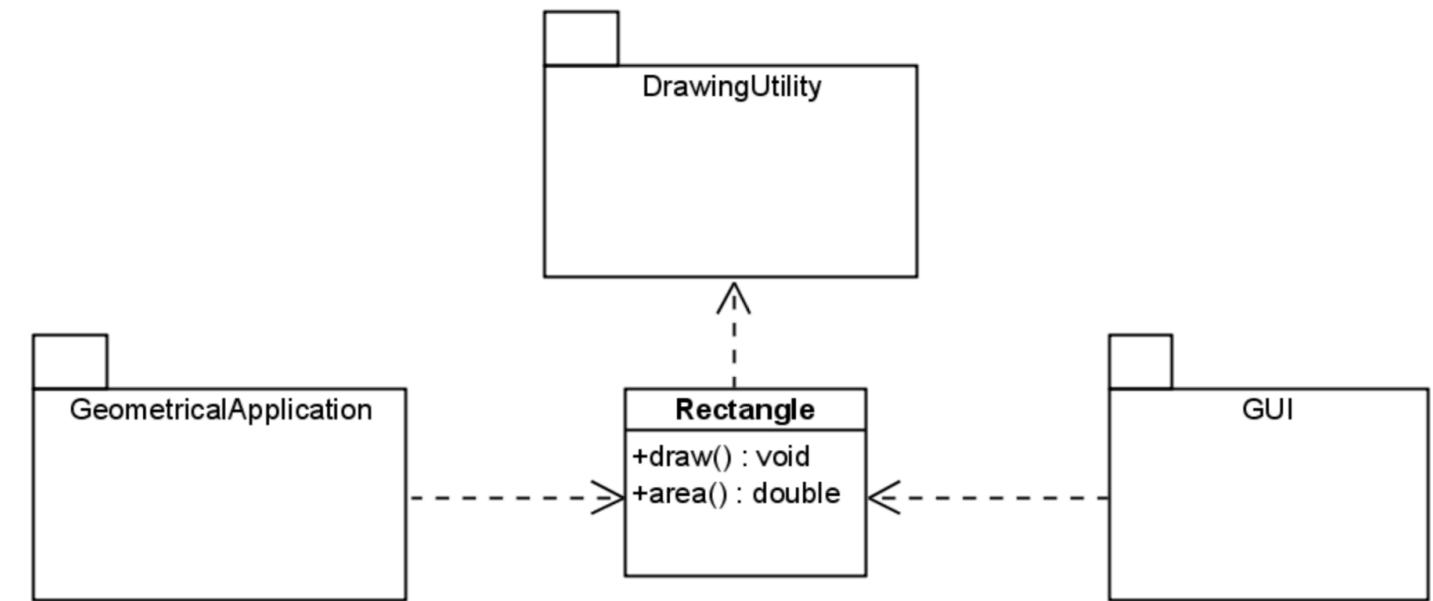
- A class is assigned the responsibility to know or do something
 - Class PersonData is responsible for knowing the data of a person.
 - Class CarFactory is responsible for creating Car objects.
- **A responsibility is an axis of change.**
 - If new functionality must be achieved, or existing functionality needs to be changed, the responsibilities of classes must be changed.
- A class with only one responsibility will have only one reason to change!

Responsibility and Cohesion

- **Cohesion measures the degree of togetherness among the elements of a class.**
- In a class with very high cohesion every element is part of the implementation of one concept.
- The elements of the class work together to achieve one common functionality.
- A class with high cohesion implements only one responsibility
- Therefore, a **class with low cohesion violates SRP**

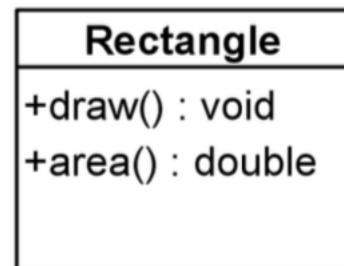
Example

- GUI package uses Rectangle to draw rectangle shapes in the screen. Rectangle uses DrawingUtility to implement draw.
- GeometricalApplication is a package for geometrical computations which also uses Rectangle (area()).



Problems of Rectangle

- Rectangle has multiple responsibilities!
 - 1) Geometrics of rectangles represented by the method area()
 - 2) Drawing of rectangles represented by the method draw()
- Rectangle has low cohesion!
 - Geometrics and drawing do not naturally belong together.



Why is it a problem?

Problems of Rectangle

Rectangle
+draw() : void
+area() : double

- **Rectangle is hard to use!**

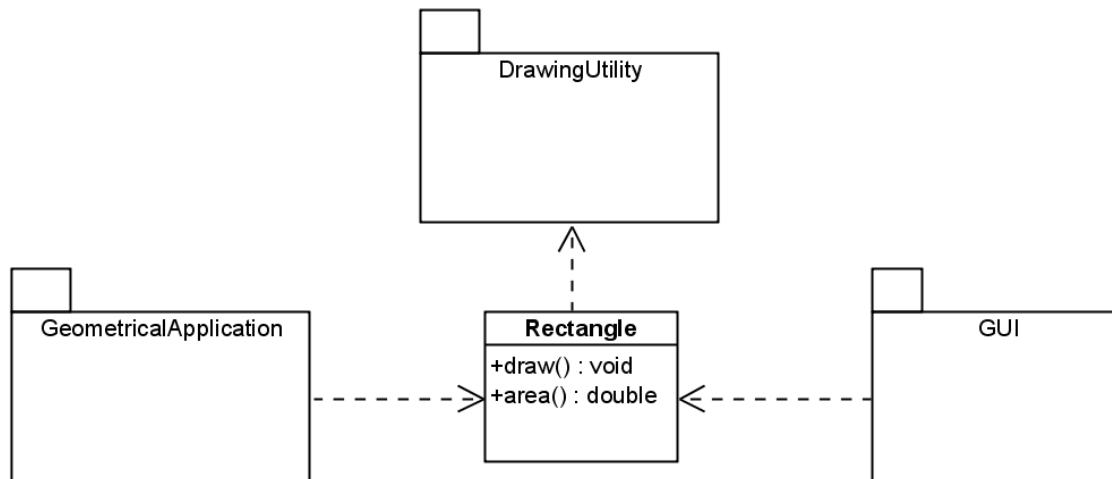
- It has multiple reasons to change.
- Even if we want to use only one of its responsibilities, we must depend on both of them.
- We inherit the effects of changes along every possible axis of change (= responsibility)

- **Rectangle is easily misunderstood!**

- It is not only a representation of a rectangle shape, but also part of a process concerned with drawing rectangle shapes in the screen.
It was not created as a representation of a certain concept, but as a bundle of needed functionality without careful consideration of their cohesion.

Undesired Effects of Change

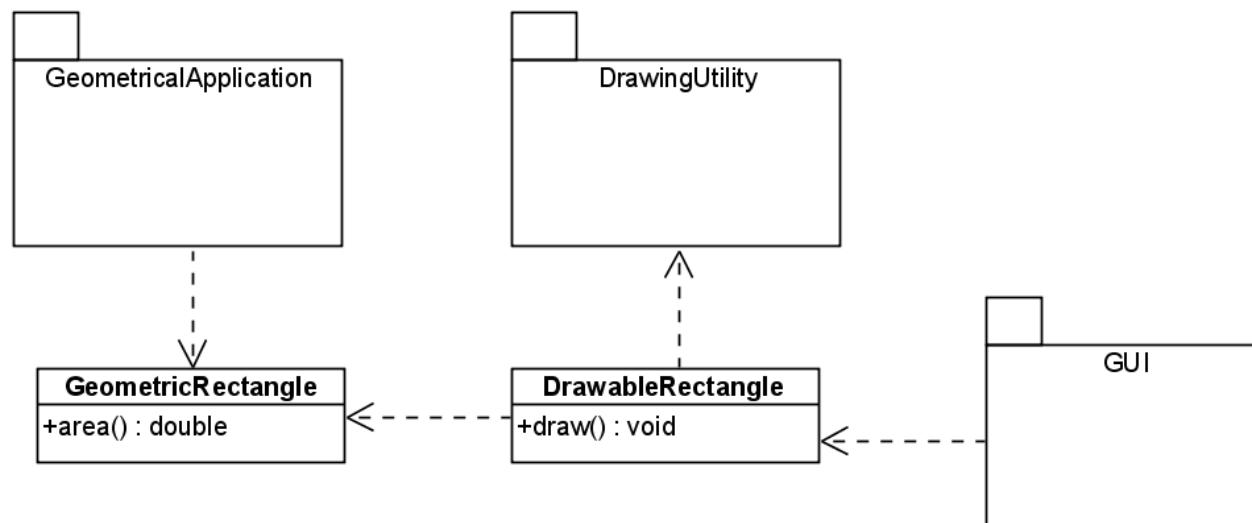
- Unnecessary dependency between GeometricApplication and DrawingUtility (DrawingUtility classes have to be deployed along with Rectangle) even if we only want to use the geometrical functions of rectangles.



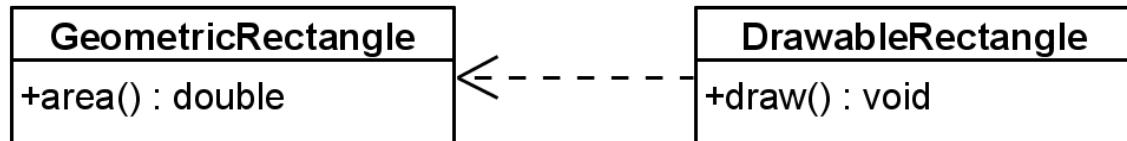
- **Problem:** If drawing functionality changes in the future, we need to retest Rectangle also in the context of GeometricalApplication!

A SRP-Compliant Design

- Split Rectangle according to its responsibilities.
 - GeometricRectangle models a rectangle by its geometric properties.
 - DrawableRectangle models a graphical rectangle by its visual properties.
- GeometricalApplication uses only GeometricRectangle. It only depends on the geometrical aspects.
- GUI uses DrawableRectangle and indirectly GeometricRectangle. It needs both aspects and therefore has to depend on both.



Two Classes with High Cohesion



- **Both classes can be (re)used easily!**
 - Only changes to the responsibilities we use will affect us.
- **Both classes are easily understood!**
 - Each implements one concept.
 - GeometricRectangle represents a rectangle shape by his size.
 - DrawableRectangle encapsulates a rectangle with visual properties.

Employee Example

- Consider the class Employee which has two responsibilities:
 - 1) Calculating the employees pay.
 - 2) Storing the employee data to the database.

Employee
+calculatePayment() : double
+storeToDatabase() : void

Should we split responsibilities?

Employee Represents a Typical SRP-Violation

- Calculating the payment of an employee is part of the **business rules**.
 - It corresponds to a real-world concept the application shall implement.
- Storing the employee information in the database is a **technical aspect**.
 - It is a necessity of the IT architecture that we have selected; does not correspond to a real-world concept.
- **Mixing business rules and technical aspects is calling for trouble!**
 - From experience we know that both aspects are extremely unpredictable.

Most probably we should split in this case.

Employee
+calculatePayment() : double
+storeToDatabase() : void

Modem Example

- The class Modem has also two responsibilities:
 1. Connection management (dial and hangup)
 2. Data communication (send and receive)

Modem
+dial(number : String) : void
+hangup() : void
+send(c : char) : void
+receive() : char

Should we split?

To Split or Not to Split Modem?

- Break down the question to:
 1. Do we expect connection management and data communication to constitute **different axes of change**?
 2. Do we expect them to change together, or independently.
 3. Will these responsibilities be **used by different clients**?
 4. Do we plan to provide **different configurations of modems to different customers**?

To Split or Not to Split Modem?

- **Split if:**

- Responsibilities will change separately.
- Responsibilities are used / will probably be used by different clients.
- We plan to provide different configurations of modems with varying combinations of responsibilities (features).

- **Do not split if:**

- Responsibilities will only change together, e.g. if they both implement one common protocol.
- Responsibilities are used together by the same clients.
- Both correspond to non optional features.

Modem Example

- The class Modem has also two responsibilities:
 1. Connection management (dial and hangup)
 2. Data communication (send and receive)

Modem
+dial(number : String) : void
+hangup() : void
+send(c : char) : void
+receive() : char

Should we split?

To Apply or Not to Apply

- Decide based on the nature of responsibilities:
 - changed together / not changed together
 - used together / not used together
 - optional / non optional
- **Only apply a principle, if there is a symptom!**
 - An axis of change is an axis of change only, if the change actually occurs.

Strategic Application

- Choose the kinds of changes to guide SRP application
 - Guess the most likely kinds of changes
 - Separate to protect from those changes
- Prescience derived from experience
 - Experienced designer hopes to know the user and an industry well enough to judge the probability of different kinds of changes.
 - Invoke SRP against the most probable changes.
- **Be agile**
 - Predictions will often be wrong.
 - Wait for changes to happen and modify the design when needed.
 - Simulate change.

Simulate Change

- Write tests first
 - Testing is one kind of usage of the system
 - Force the system to be testable
 - Force developers to build the abstractions needed for testability
- Use **short development (iteration) cycles**
- Develop features before infrastructure
 - show them to stakeholders
- Develop the **most important features first**
- Release software early and often
 - get it in front of users and customers as soon as possible

Summary

- Applying SRP maximizes the cohesion of classes.
- Classes with high cohesion:
 - can be reused easily,
 - are easily understood,
 - protect clients from changes that should not affect them.
- Be strategic in applying SRP.
- Carefully study the context and make informed trade-offs.
- Guess at most likely axes of change and separate along them.
- Be agile: Simulate changes as much as possible; apply SRP when changes actually occur.
- Separation may not be straightforward with typical OO mechanisms

Open-Closed Principle (OCP)

- "*Software Systems change during their life time*"
 - both better designs and poor designs have to face the changes;
 - good designs are stable

*Software entities should be open for extension,
but closed for modification*

B. Meyer, 1988 / quoted by R. Martin, 1996

- Be open for extension
 - ▶ module's behavior can be extended
- Be closed for modification
 - ▶ source code for the module must not be changed
- *Modules should be written so they can be extended without requiring them to be modified*

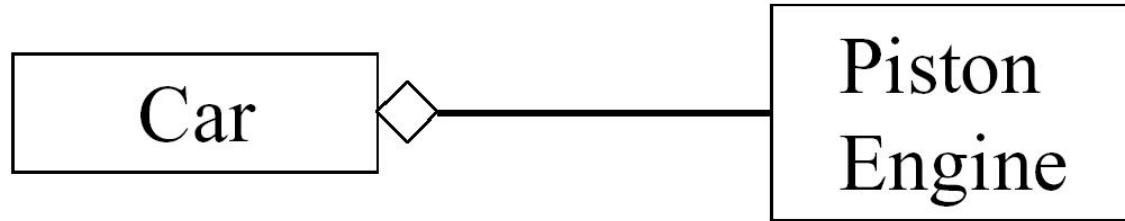
Extension and Modification

- **Extension:** Extending the *behavior* of an module.
- **Modification:** Changing the *code* of an module.
- **Open for extension:**
As requirements of the application change, we can extend the module with new behaviors that satisfy those changes. We change what the module does.
- **Closed for modification:**
Changes in behavior do not result in changes in the modules source or binary code.

Why Closed for Modifications?

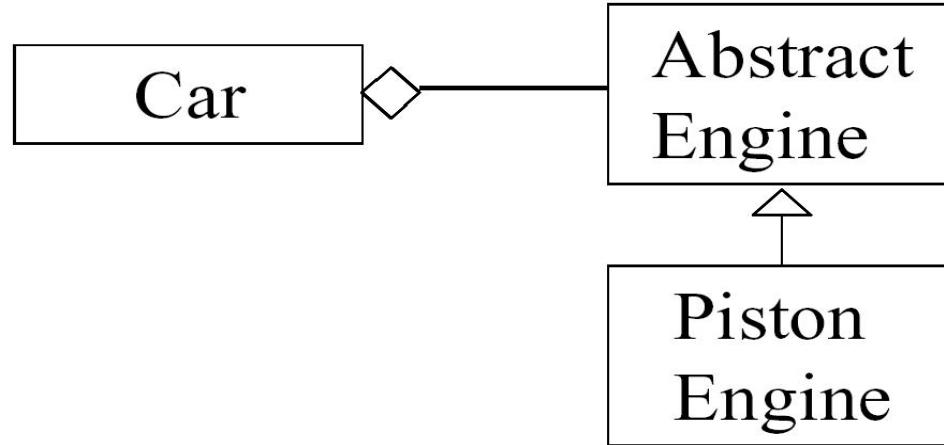
- Question: Why not simply change the code if I needed?
 1. Module was **already delivered to customers**, a change will not be accepted.
 - If you need to change something, hopefully you opened your module for extension!
 2. Module is a **third-party library only available as binary code**.
 - If you need to change something, hopefully the third-party opened the module for extension!
 3. **Most importantly:** not changing existing code for the sake of implementing extensions enables incremental compilation, testing, debugging.

Open the door ...



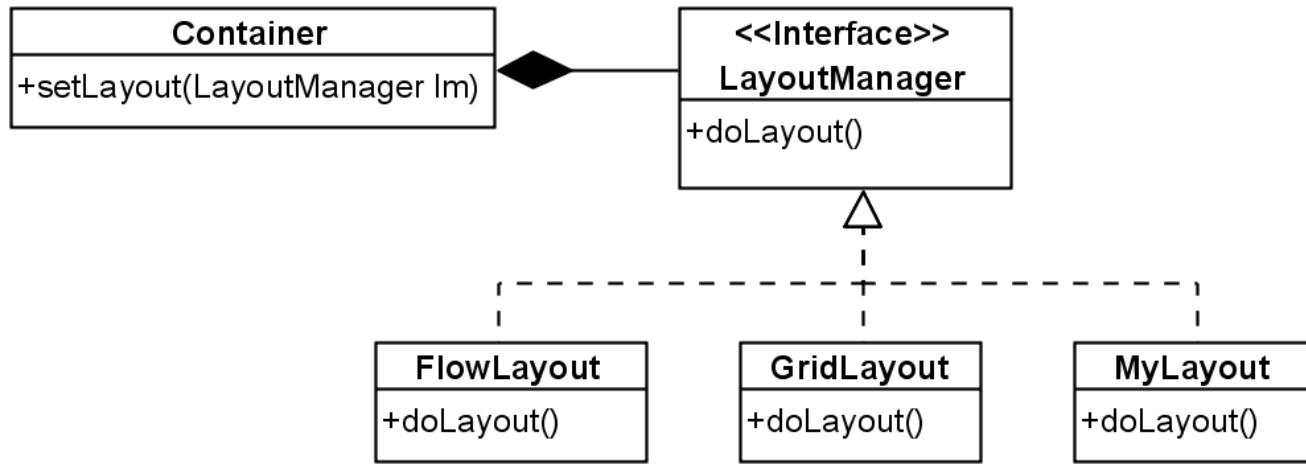
- How to make the **Car** run efficiently with a **TurboEngine**?
- Only by changing the Car in the given design!

... But Keep It Closed!



- A class must not depend on a concrete class!
- It must depend on an **abstract** class using **polymorphic** dependencies (calls)
- Abstraction is the key

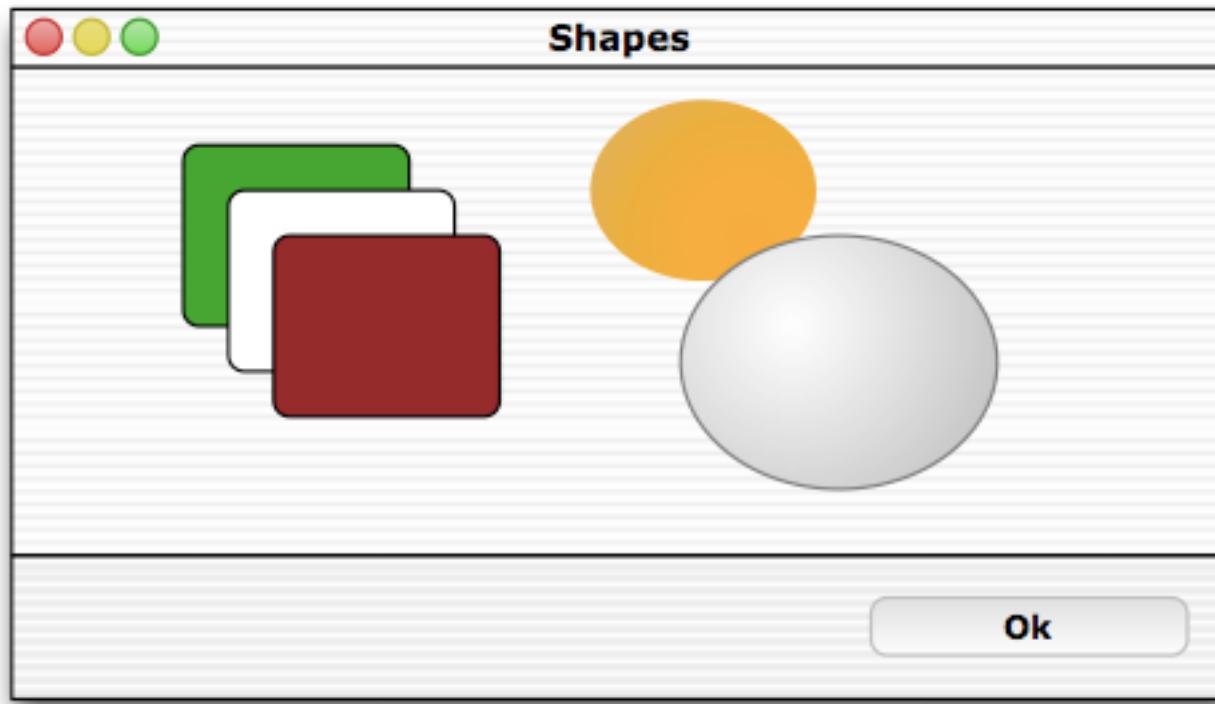
Abstracting over Variations



- Container delegates the layout functionality to an abstraction. The rest of its functionality is implemented against this abstraction.
- To change the behavior of an instance of Container we configure it with the LayoutManager of our choice.
- We can add new behavior by implementing our own LayoutManager.

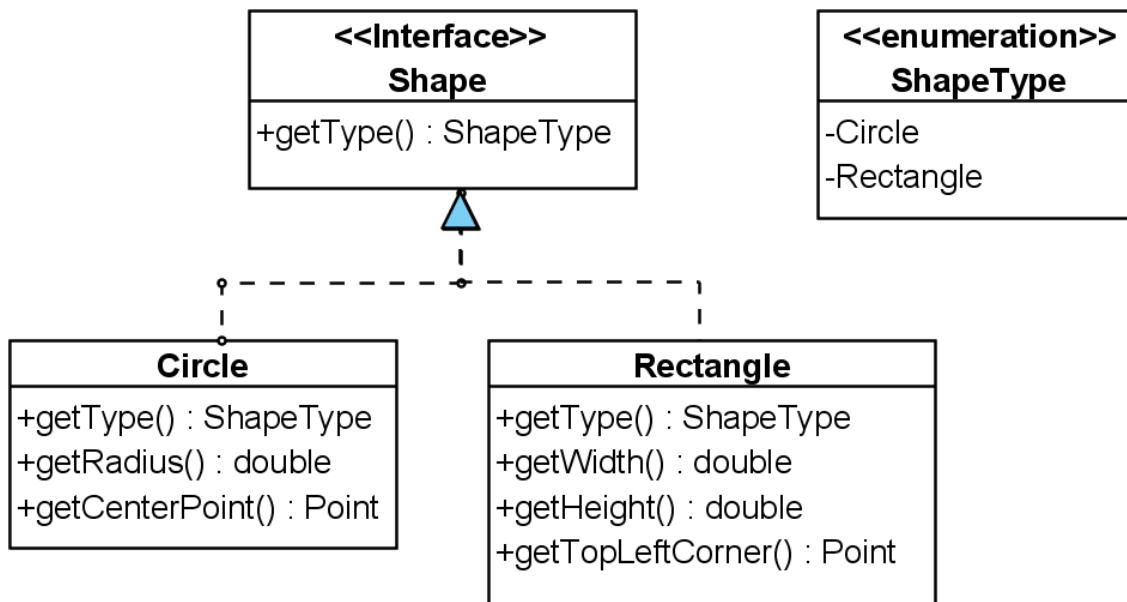
OCP by Example

- Consider an application that draws shapes - circles and rectangles – on a standard GUI.



A Possible Design

- Realizations of Shape identify themselves via the enumeration ShapeType
- Realizations of Shape declare specialized methods for the shape type they represent; they mostly serve as containers for storing the geometric properties of shapes.



A Possible Design

- Drawing is implemented in separate methods (say of Application class)

```
public void drawAllShapes(List<Shape> shapes) {  
    for(Shape shape : shapes) {  
        switch(shape.getType()) {  
            case Circle:  
                drawCircle((Circle)shape);  
                break;  
            case Rectangle:  
                drawRectangle((Rectangle)shape);  
                break;  
        }  
    }  
}  
  
private void drawCircle(Circle circle) {  
    ...  
}  
  
private void drawRectangle(Rectangle rectangle) {  
    ...  
}
```

Evaluating the Design

- **Adding new shapes (e.g., Triangle) is hard;** we need to:
 - Implement a new realization of Shape.
 - Add a new member to ShapeType.
 - This possibly leads to a recompile of all other realizations of Shape.
 - drawAllShapes (and every method that uses shapes in a similar way) must be changed.

Hunt for every place that contains conditional logic to distinguish between the types of shapes and add code to it.
- **drawAllShapes is hard to reuse!**
 - When we reuse it, we have to bring along Rectangle and Circle.

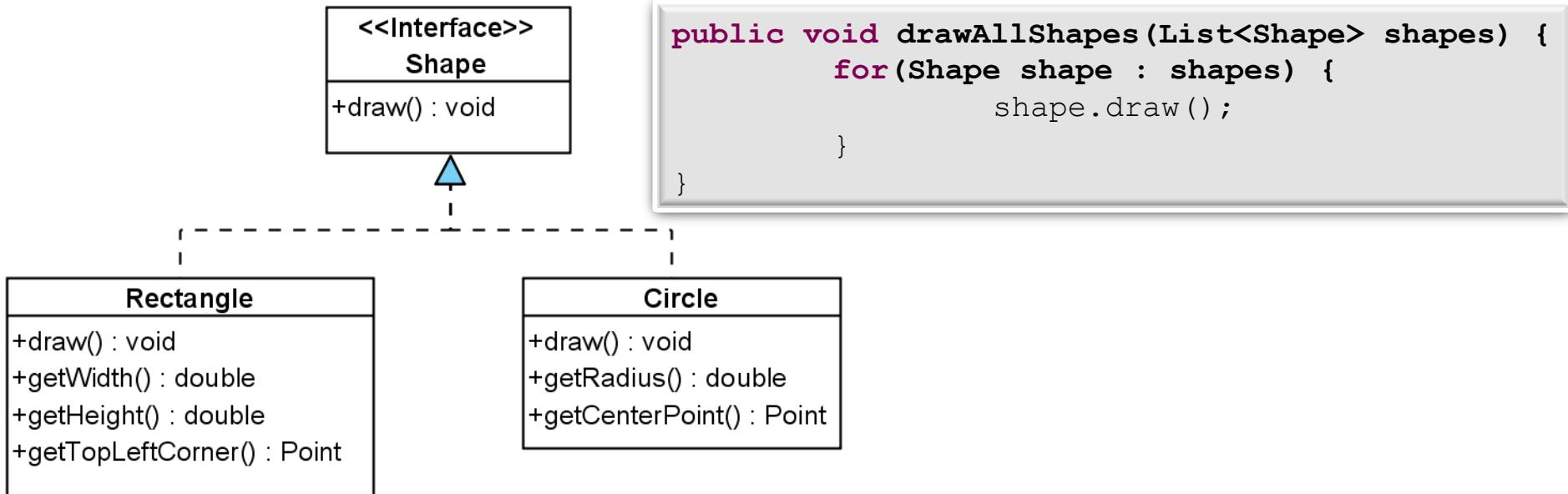
Rigid, Fragile, Immobile Designs

- **Rigid designs** are hard to change – every change causes many changes to other parts of the system.
 - Our example design is rigid: Adding a new shape causes many existing classes to be changed.
- **Fragile designs** tend to break in many places when a single change is made.
 - Our example design is fragile: Many switch/case (if/else) statements that are both hard to find and hard to decipher.
- **Immobile designs** contain parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too big.
 - Our example design is immobile: DrawAllShapes is hard to reuse.

Evaluating the Design

- The design violates OCP with respect to extensions with new kinds of shapes.
- We need to open our module for this kind of change by building appropriate abstractions.

An Alternative Design

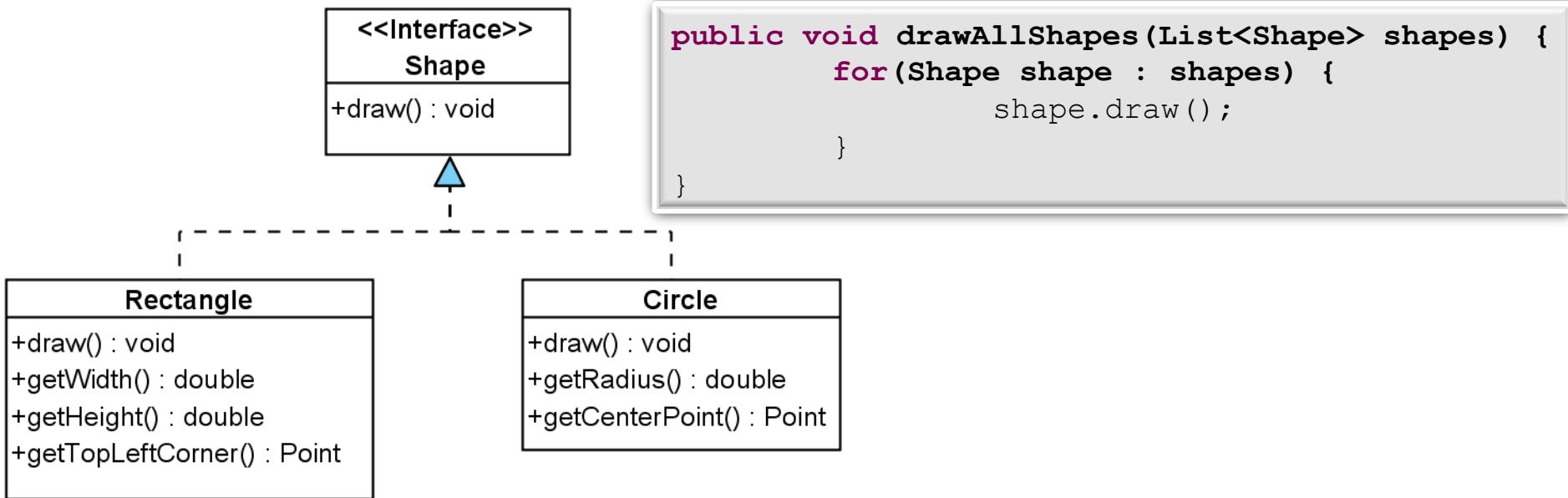


- **New abstraction:** `Shape.draw()` `ShapeType` is not necessary anymore.

- **Extensibility:**

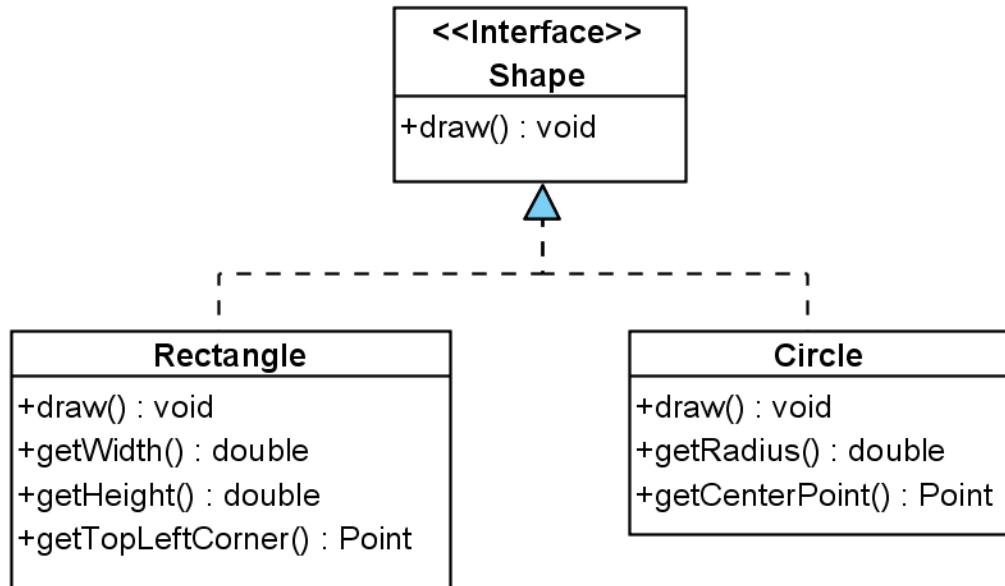
Adding new shapes is easy! Just implement a new realization of `Shape`. `drawAllShapes` only depends on `Shape`! We can reuse it efficiently.

An Alternative Design



Problematic Changes

- Consider **extending** the design with further **shape functions**
 - shape transformations,
 - shape dragging,
 - calculating of shape intersection, shape union, etc.
- Consider **adding support for different operating systems**.
 - The implementation of the drawing functionality varies for different operating systems.



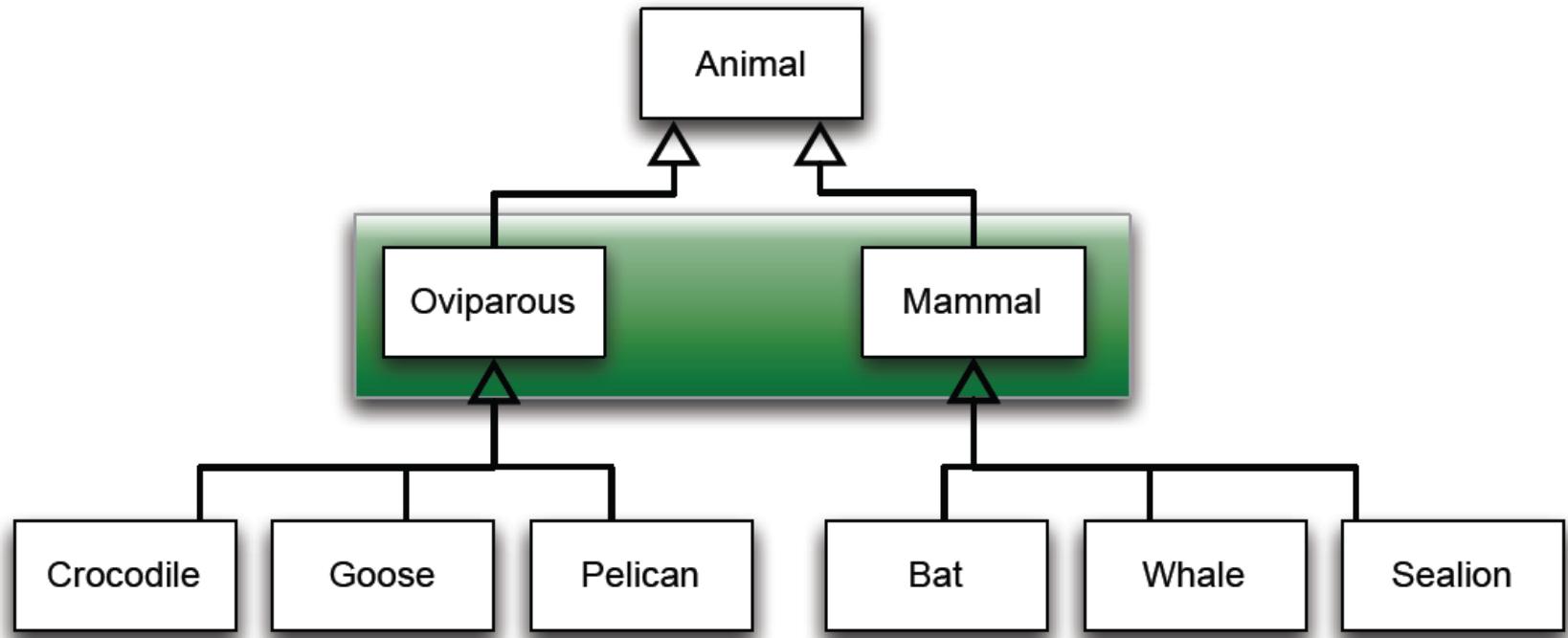
Abstractions are the key

- May Support or Hinder Change
 - Change is easy if change units correspond to abstraction units
 - Change is tedious if they do not correspond to abstraction units
- Reflect a viewpoint
 - No matter how “open” a module is, there will always be some kind of change that requires modification
 - There is no model that is natural to all contexts.

Viewpoints Illustrated: The Case of a Zoo

- Imagine: Development of a "Zoo Software".
- Three stakeholders:
 - Veterinary surgeon
 - What matters is how the animals reproduce!
 - Animal trainer
 - What matters is the intelligence!
 - Keeper
 - What matters is what they eat!

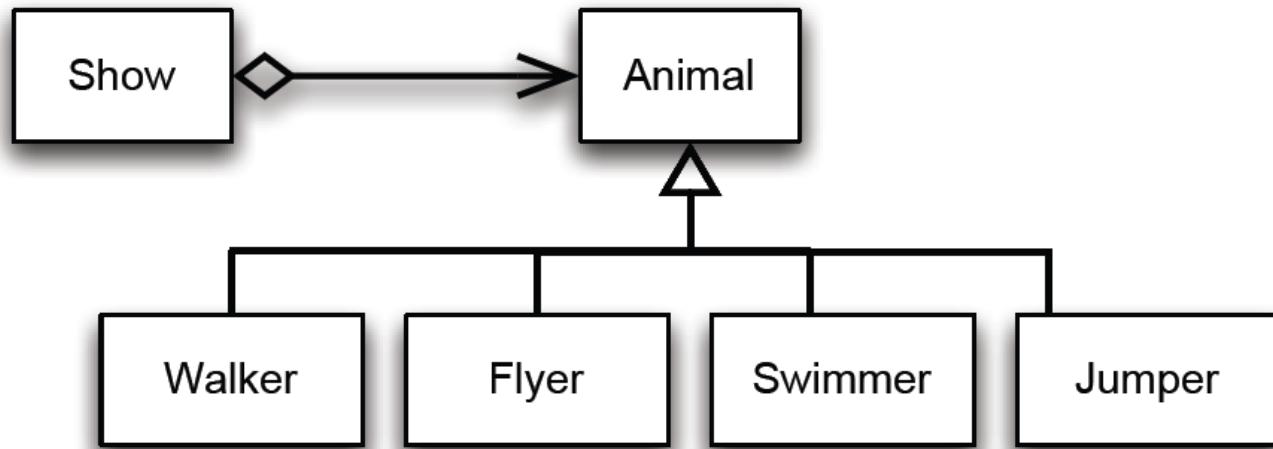
One Possible Class Hierarchy



The veterinary surgeon has won !

The World from Trainer's Viewpoint

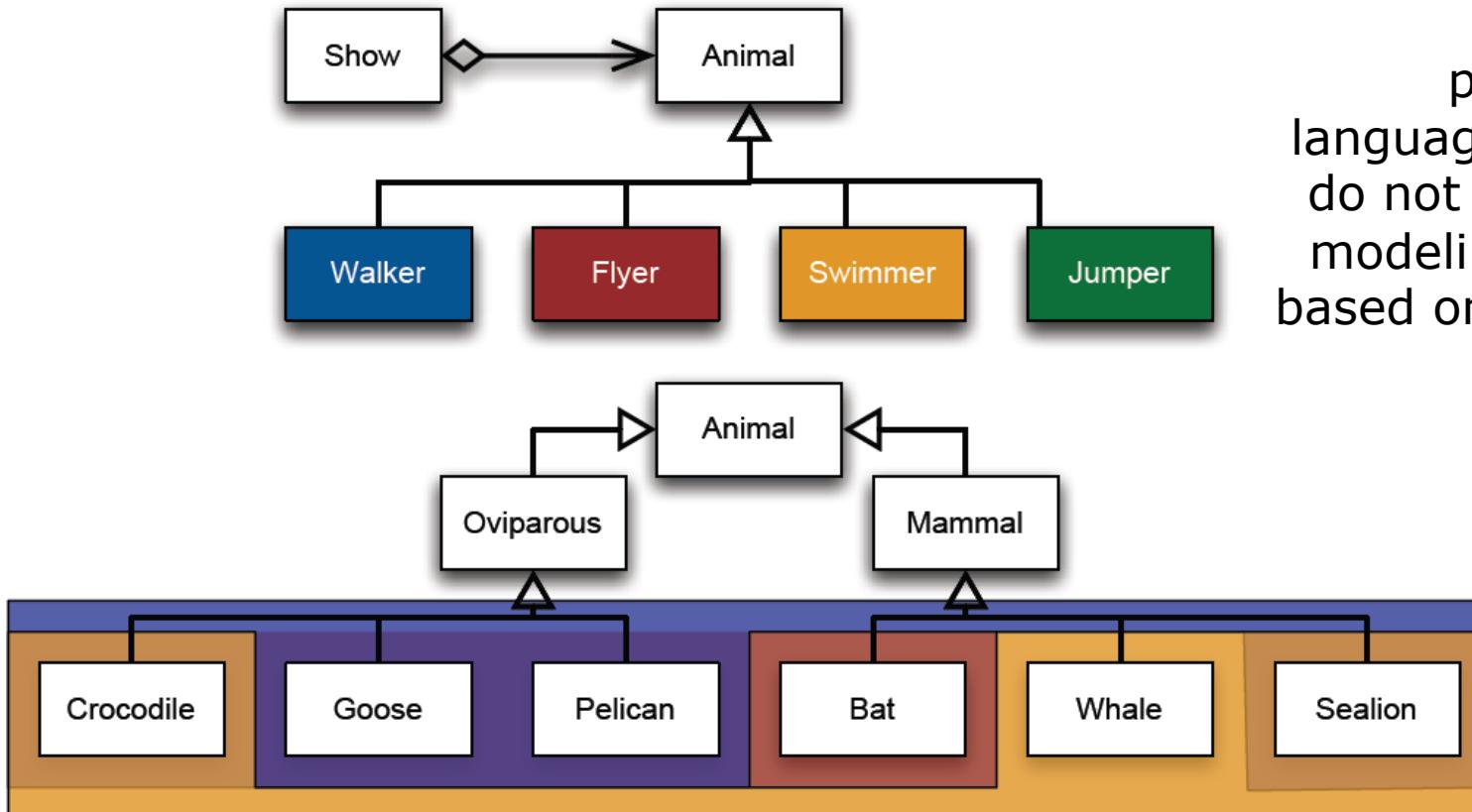
“The show shall start with the pink pelicans and the African geese flying across the stage. They are to land at one end of the arena and then walk towards a small door on the side. At the same time, a killer whale should swim in circles and jump just as the pelicans fly by. After the jump, the sea lion should swim past the whale, jump out of the pool, and walk towards the center stage where the announcer is waiting for him.”



Models Reflecting Different Viewpoints

Overlap

- Overlapping: Elements of a category in one model correspond to several categories in the other model and the other way around.
- Adopting the veterinary viewpoint hinders changes that concern trainer's viewpoint and the other way around.



Our current programming languages and tools do not support well modeling the world based on co-existing viewpoints.

An Interim Takeaway

No matter how “closed” a module is, there will always be some kind of change against which it is not closed.

Strategic Closure

"No significant program can be 100% closed"

R.Martin, "The Open-Closed Principle," 1996

- Closure not *complete* but *strategic*
- Closure Against What?
 - You have to choose which changes you'll isolate yourself against.
 - What if we have to draw all circles first? Now DrawAllShapes must be edited (or we have to hack something)
- Opened Where?
 - Somewhere, someone has to instantiate the individual shapes.
 - It's best if we can keep the dependencies confined

Takeaway

- Abstraction is the key to supporting OCP.
- No matter how “open” a module is, there will always be some kind of change which requires modification.
- Limit the Application of OCP to changes that are likely.
 - After all wait for changes to happen.
 - Stimulate change (agile spirit).

OCP Heuristics

Make all object-data private
No Global Variables!

- **Changes** to public data are always at risk to “open” the module
 - They may have a rippling effect requiring changes at many unexpected locations;
 - Errors can be difficult to completely find and fix. Fixes may cause errors elsewhere.

OCP Heuristics (2)

RTTI is Ugly and Dangerous!

- RTTI is ugly and dangerous
 - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
 - recognize them by type **switch**-es or **if-else-if** structures
- Not all these situations violate OCP all the time
 - when used only as a "filter"

Liskov Substitution Principle (LSP)

- Subtypes must be behaviorally substitutable for their base types.

—Barbara Liskov, 1988 (ACM Turing Award Receiver)

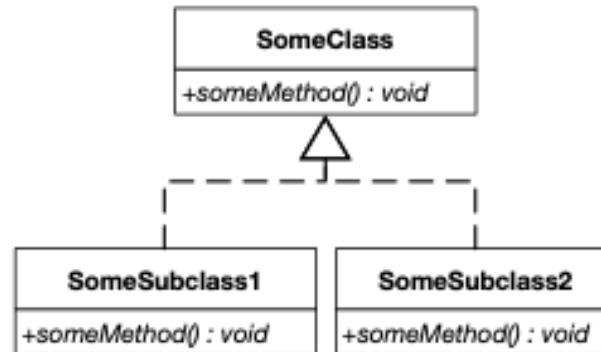
The Liskov Substitution Principle

- Class inheritance and subtype polymorphism as primary mechanisms for supporting the open-closed principle (OCP) in object-oriented designs
- The Liskov Substitution Principle
 - ... gives us a way to characterize good inheritance hierarchies.
 - ... increases our awareness about traps that will cause us to create hierarchies that do not conform to the open-closed principle.

Substitutability

- In object-oriented programs, subclasses are substitutable for superclasses in client code:

```
void clientMethod(SomeClass sc)
{
    ...
    sc.someMethod();
    ...
}
```

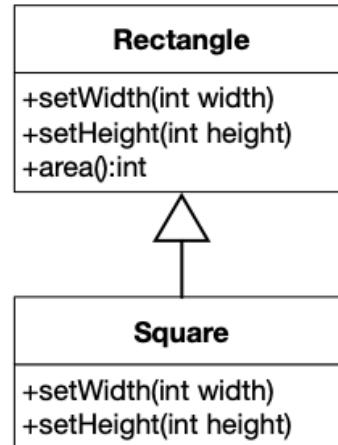


- In clientMethod, sc may be an instance of SomeClass or any of its subclasses.
Hence, if clientMethod works with instances of SomeClass, it does so with instances of any subclass of SomeClass. They provide all methods of SomeClass and eventually more.

Example

```
class Rectangle {  
    public void setWidth(int width) { this.width = width; }  
    public void setHeight(int height) { this.height = height; }  
    public void area() { return height * width; }  
}
```

```
class Square extends Rectangle {  
    public void setWidth(int width) {  
        super.setWidth(width);  
        super.setHeight(width);  
    }  
  
    public void setHeight(int height) {  
        super.setWidth(height);  
        super.setHeight(height);  
    }  
}
```

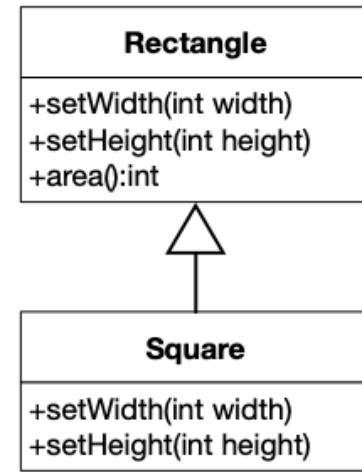


Assume that we want to implement a class **Square** and want to maximize reuse.

What do you think about this design?

Critique

```
class Square extends Rectangle {  
  
    public void setWidth(int width) {  
        super.setWidth(width);  
        super.setHeight(width);  
    }  
  
    public void setHeight(int height) {  
        super.setWidth(height);  
        super.setHeight(height);  
    }  
}
```



- Is it mathematically correct?
 - A square has the mathematical properties of a rectangle: A square has four edges and only right angles and is therefore a rectangle.
- Is this implementation correct?
 - With this overriding of `setHeight` and `setWidth` – to set both dimensions to the same value – instances of **Square** remain mathematically valid squares.
- Does Java type system allow substitution?
 - We can pass **Square** wherever **Rectangle** is expected, as far as the Java type system is concerned.
- **But, we may break assumptions that clients of **Rectangle** make about the “behavior” of a **Rectangle**.**

The client is the keyword here!

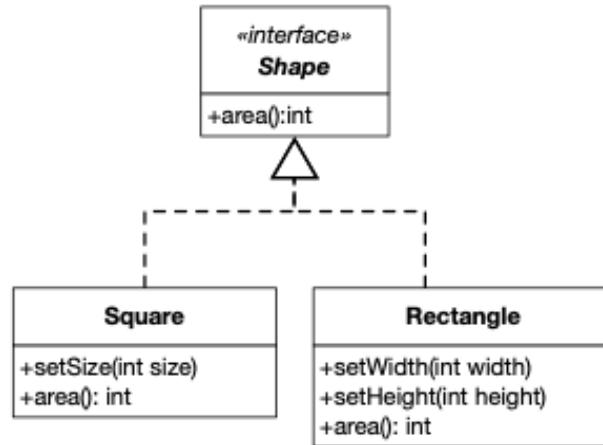
```
void clientMethod(Rectangle rec)
{
    rec.setWidth(5);
    rec.setHeight(4);
    assert(rec.area() == 20);
}
```

- This client that works with Rectangle but not with Square
- The clientMethod method makes an assumption that is true for Rectangle
 - setting the width respectively height has no effect on the other attribute. This assumption does not hold for Square.
- The Rectangle/Square hierarchy violates the Liskov Substitution Principle (LSP)!
 - Square is behaviorally not a correct substitution for Rectangle.
- A Square does not comply with the behavior of a rectangle:
 - Changing the height/width of a square behaves differently from changing the height/width of a rectangle.
 - Actually, it doesn't make sense to distinguish between the width and the height of a square.

Interim Takeaway Messages

- Programmers do not define entities that are something, but entities that behave somehow.
- A model viewed in isolation can not be meaningfully validated!
- The validity of a model depends on the clients that use it.
 - Inspecting the Square/Rectangle hierarchy in isolation did not show any problems. In fact it even seemed like a self-consistent design. We had to inspect the clients to identify problems.
- The validity of a model must be judged against the possible uses of the model. We need to anticipate the assumptions that clients will make about our classes.

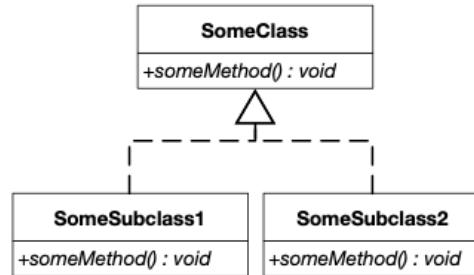
LSP-compliant Design for Rectangle/Square



- Clients of **Shape** cannot make any assumptions about the behavior of setter methods.
- When clients want to change properties of the shapes, they have to work with the concrete classes.
- When clients work with the concrete classes, they can make true assumptions about the computation of the area.

Behavioral Substitutability

- What does the Liskov Substitution Principle add to the common object-oriented subtyping rules?



- It's not enough that instances of SomeSubclass1 and SomeSubclass2 provide all methods declared in SomeClass. These methods should also behave like their heirs!
- A client method should not be able to distinguish the behavior of objects of SomeSubclass1 and SomeSubclass2 from that of objects of SomeClass.

Design by Contract

- Solution to the validation problem: A technique for explicitly stating what may be assumed.
- Pre- and Post-conditions
 - Declared for every method of the class.
 - Preconditions must be true for the method to execute.
 - Post-conditions must be true after the execution of the method.
- Invariants
 - Properties that are always true for instances of the class.
 - May be broken temporarily during a method execution, but otherwise hold.
- We can specify contracts using Pre-, Post-Conditions and Invariants. They must be respected by subclasses and clients can rely on them.

Behavioral Subtyping Rules

- Rule for Preconditions:

- Preconditions of a class imply preconditions of its subclasses.
- Preconditions may be replaced by (equal or) **weaker** ones.

- Rule for Postconditions:

- Postconditions of a class are implied by those of its subclasses.
- Postconditions may be replaced by equal or **stronger** ones.

- Rule for Invariants:

- can be replaced by **stronger** ones

- A derived class must not impose more obligations on clients.
- Conditions that clients obey to before executing a method on an object of the base class should suffice to call the same method on instances of subclasses.
- Properties assumed by clients after executing a method on an object of the base class still hold when the same method is executed on instances of subclasses.
- The guarantees that a method gives to clients can only become stronger.

BigNatural vs BigInteger

```
//@mathmodel n integer      //@mathmodel n integer
//@constraint n >= 0        //@constraint
interface BigNatural {     interface BigInteger {

    //@alters n             //@alters n
    //@ens n = #n+1          //@ens n = #n+1
    void increment();       void increment();

    //@alters n             //@alters n
    //@ens n=max(0,#n-1)     //@ens n = #n-1
    void decrement();       void decrement();

}}
```

Should BigNatural extend BigInteger?

For behavioral subtyping, ask

- Is BigNatural's invariant *stronger*?
- Do all BigNatural methods *require less*?
- Do all BigNatural methods *ensure more*?

```
//@mathmodel n integer      //@mathmodel n integer
//@constraint n >= 0          //@constraint
interface BigNatural {      interface BigInteger {

    //@alters n                  //@alters n
    //@ens n = #n+1              //@ens n = #n+1
    void increment();           void increment();

    //@alters n                  //@alters n
    //@ens n=max(0,#n-1)         //@ens n = #n-1
    void decrement();           void decrement();
}

}
```

Is BigNatural's invariant *stronger*?

- BigNatural invariant is $n \geq 0$
- BigInteger invariant is true

Yes

```
//@mathmodel n integer      //@mathmodel n integer
//@constraint n >= 0        //@constraint
interface BigNatural {     interface BigInteger {
```

Do all BigNatural methods *require less*?

- increment() requires the same (true) in both
- decrement() requires the same (true) in both

Yes

```
interface BigNatural {    interface BigInteger {  
  
    //@alters n          //@alters n  
  
    void increment();    void increment();  
  
    //@alters n          //@alters n  
    //  
    void decrement();    void decrement();  
}  
}
```

Do all BigNatural methods *ensure more*?

- BigNatural decrement() ensures $\#n > 0 \implies n = \#n - 1$
- BigInteger decrement() ensures $n = \#n - 1$

No

```
interface BigNatural {    interface BigInteger {  
  
    //@ens n = #n+1          //@ens n = #n+1  
    void increment();        void increment();  
  
    //@ens n=max(0,#n-1)      //@ens n = #n-1  
    void decrement();        void decrement();  
}  
}
```

Postconditions cannot be weakened in the derivative.
(You cannot guarantee less than the parent.)

Example violating client

- `noop()` is correct for `BigInteger`, but not for `BigNatural`

```
BigInteger noop(BigInteger i) {  
    i.decrement();  
    i.increment();  
    return i;  
}
```

Java Modeling Language

- In JML, specifications are written as Java annotation comments to the Java program, which hence can be compiled with any Java compiler.
- To process JML specifications several tools exist:
 - an assertion-checking compiler (jmlc) which performs runtime verification of assertions,
 - a unit testing tool (jmlunit),
 - an enhanced version of javadoc (jmldoc) that understands JML specifications and
 - an extended static checker ([ESC/Java](<http://en.wikipedia.org/wiki/ESC/Java>)) a static verification tool that uses JML as its front-end.

```
public class Rectangle implements Shape {  
  
    private int width;  
    private int height;  
  
    /*@  
     * @ requires w > 0;  
     * @ ensures height = \old(height) && width = w;  
     */  
    public void setWidth(double w) {  
        this.width = w;  
    }  
}
```

Straightforward examples of violations of the Liskov Substitution Principle

- Derivates that override a method of the superclass by an empty method.
- Derivates that document that certain methods inherited from the superclass should not be called by clients.
- Derivates that throw additional (unchecked) exceptions.

LSP Violations in Java Platform Classes

- Properties inherits from Hashtable
 - Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non- String key or value, the call will fail.
- Stack inherits from Vector
- In both cases, composition would have been preferable.

Takeaway messages

- Behavioral subtyping extends “standard” OO subtyping.
Additionally ensures that assumptions of clients about the behavior of a base class are not broken by subclasses.
- Behavioral subtyping helps with supporting OCP.
Only behavioral subclassing (subtyping) truly supports open-closed designs.
- Design-by-Contract is a technique for supporting LSP. Makes the contract of a class to be assumed by the clients and respected by subclasses explicit (and checkable)

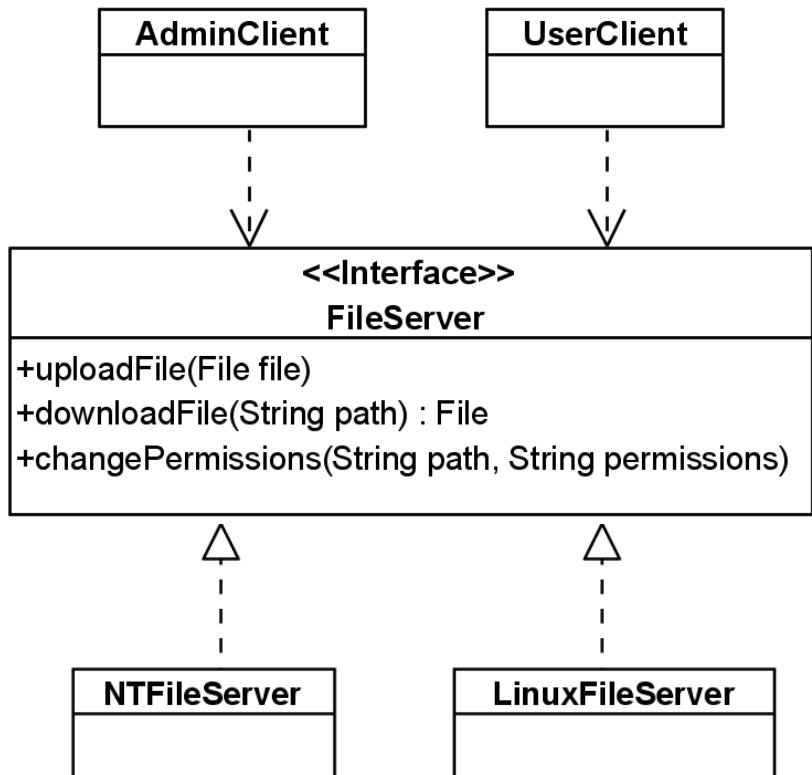
Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods that they do not use.
- When clients are forced to depend on methods they do not use, they become subject to changes to these methods, which other clients force upon the class.
- This causes coupling between all clients.

Introduction to ISP by Example

- Consider the design of a file server system.
- The interface FileServer declares methods provided by any file server.
- Various classes implement this interface for different operating systems.
- Two clients are implemented for the file server:
 - AdminClient, uses all methods.
 - UserClient, uses only the upload/download methods

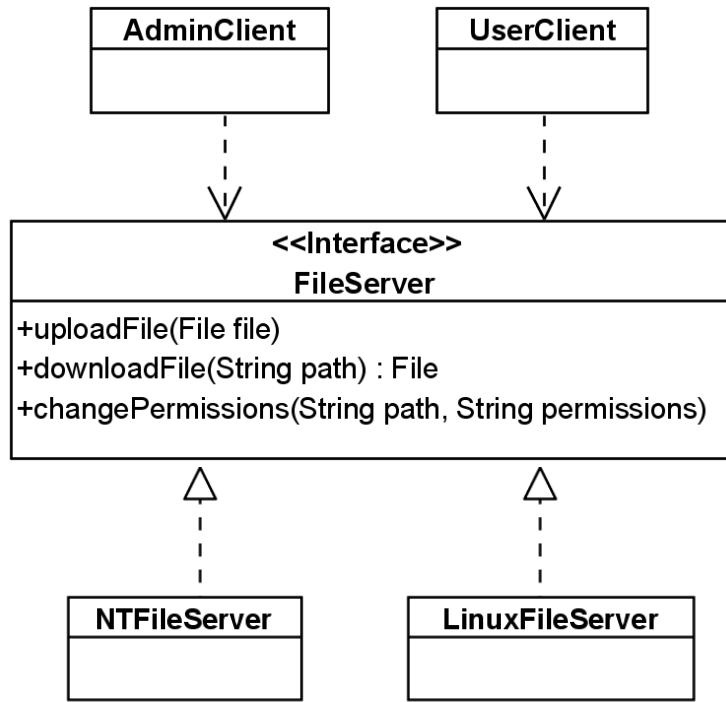
 Realization / Implementation
 Dependency



Any problems?

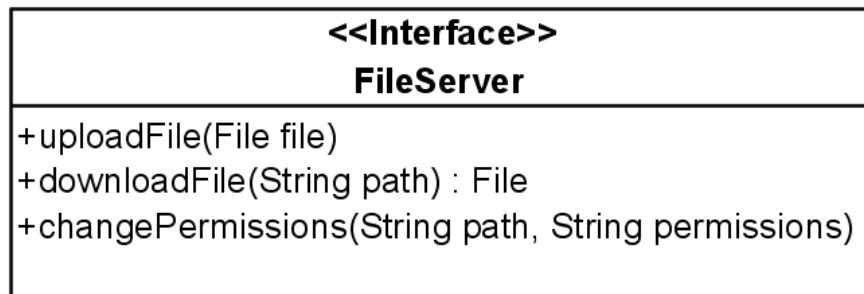
Problems of the Proposed Design

- Having the option of calling `changePermissions()` does not make sense when implementing `UserClient`.
 - The programmer must avoid calling it by convention instead of by design!
- Modifications to `changePermissions()` triggered by needs of `AdminClient` may affect `UserClient`, even though it does not use `changePermissions()`.
 - Mainly an issue with binary compatibility. A non-issue with dynamic linking.
- There may be servers that do not use a permission system.
If we wanted to reuse `UserClient` for these servers, they would be forced to implement `changePermissions`, even though it won't be used.

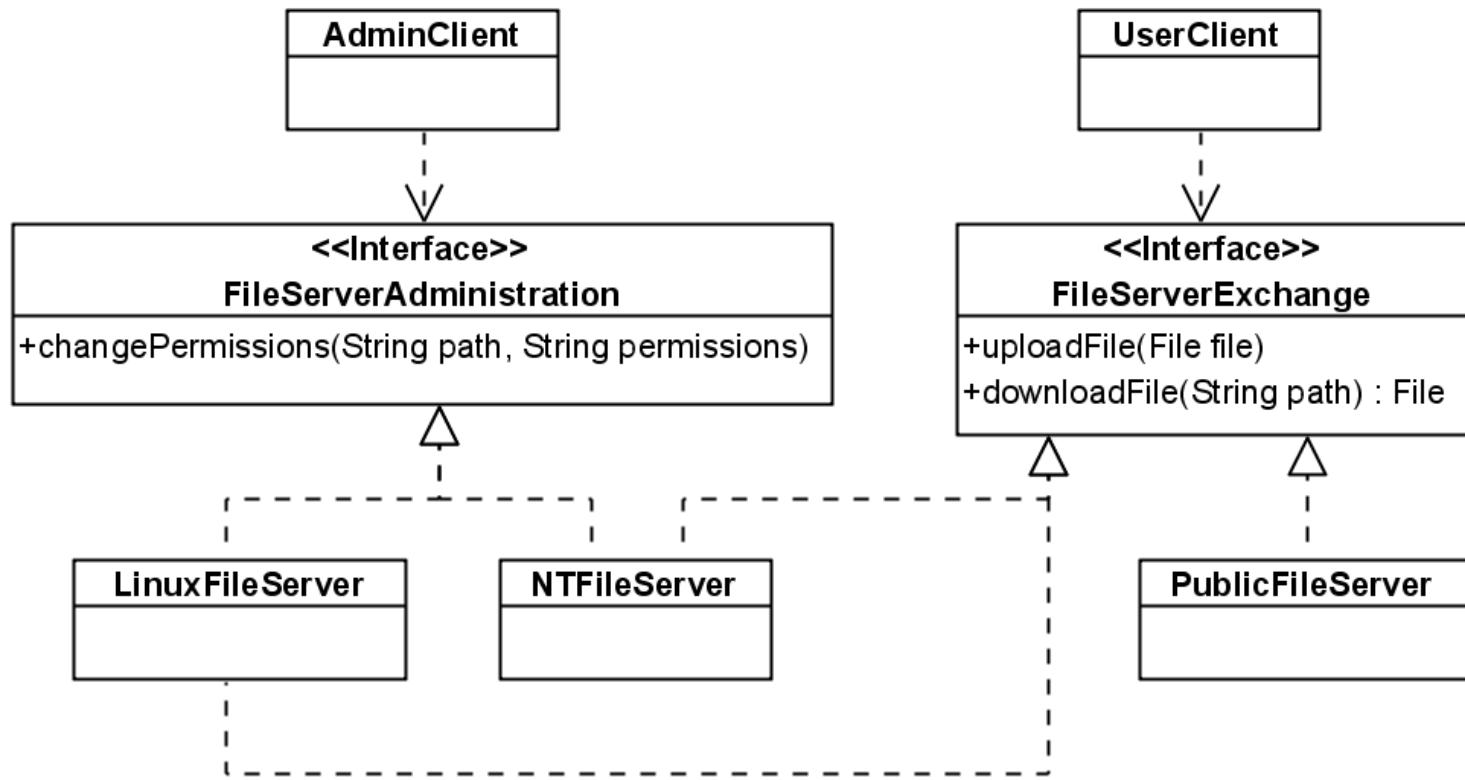


A Polluted Interface

- FileServer is a polluted interface.
- It declares methods that do not belong together.
- It forces classes to depend on unused methods and therefore depend on changes that should not affect them.
- ISP states that such interfaces should be split.



An ISP Compliant Solution



Proliferation of Interfaces

- ISP should not be overdone!
Otherwise you will end up with $2^n - 1$ interfaces for a class with n methods.
- A class implementing many interfaces may be a sign of a SRP-violation!
- Try to group possible clients of a class and have an interface for each group.

Takeaway

- Interfaces that declare unrelated methods force clients to depend on changes that should not affect them.
- Polluted interfaces should be split.
- But, be careful with interface proliferation.

Dependency Inversion Principle

I. High-level modules should ***not*** depend on low-level modules.

Both should depend on abstractions.

II. Abstractions should not depend on details.

Details should depend on abstractions

R. Martin, 1996

- OCP states the **goal**; DIP states the **mechanism**
- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions

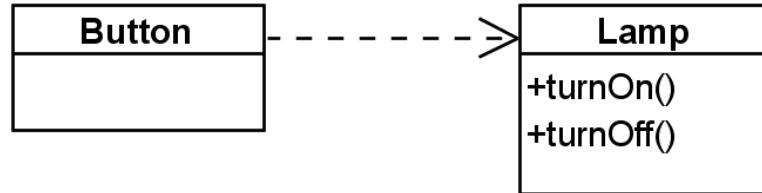
The Rationale of DIP

- **High-level, low-level Modules.**
- Good software designs are structured into modules.
 - High-level modules contain the important policy decisions and business models of an application
 - The identity of the application.
 - Low-level modules contain detailed implementations of individual mechanisms needed to realize the policy.
- High-level policy:
 - The abstraction that underlies the application;
 - the truth that does not vary when details are changed;
 - the system inside the system;
 - the metaphor.

The Rationale of DIP

- High-level policies and business processes is what we want to reuse.
- If high-level modules depend on the low-level modules changes to the lower level details will force high-level modules to change.
- It becomes harder to use them in other contexts.
- It is the high-level modules that should influence the low-level details

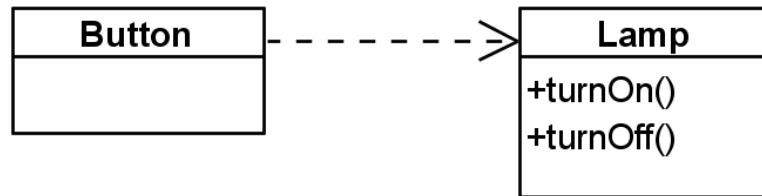
Introduction to DIP by Example



- Consider a design excerpt from a smart home scenario.
- Button
 - Is capable of “sensing” whether it has been activated/deactivated by the user.
 - Once a change is detected, it turns the Lamp on respectively off.

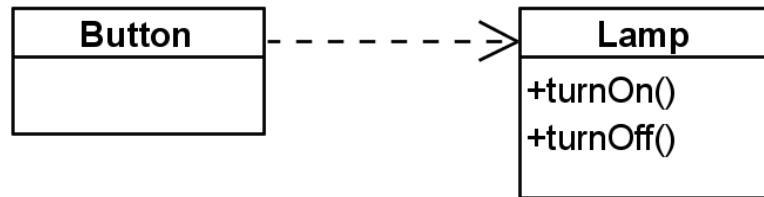
Any problems?

Problems with Button/Lamp



- We cannot reuse Button since it depends directly on Lamp. But there are plenty of other uses for Button.
- Button should not depend on the details represented by Lamp.
- These are symptoms of the real problem (Violation of DIP):
- The high-level policy underlying this (mini) design is not independent of the low-level details.

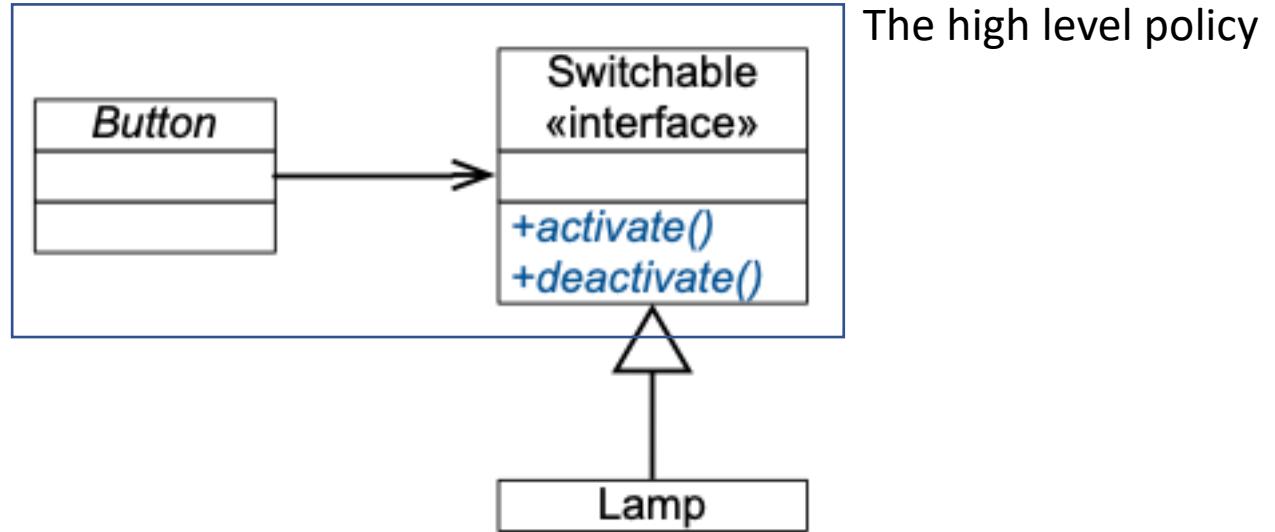
The High-Level Policy



The underlying abstraction is the detection of on/off gestures and their delegation to a server object that can handle them.

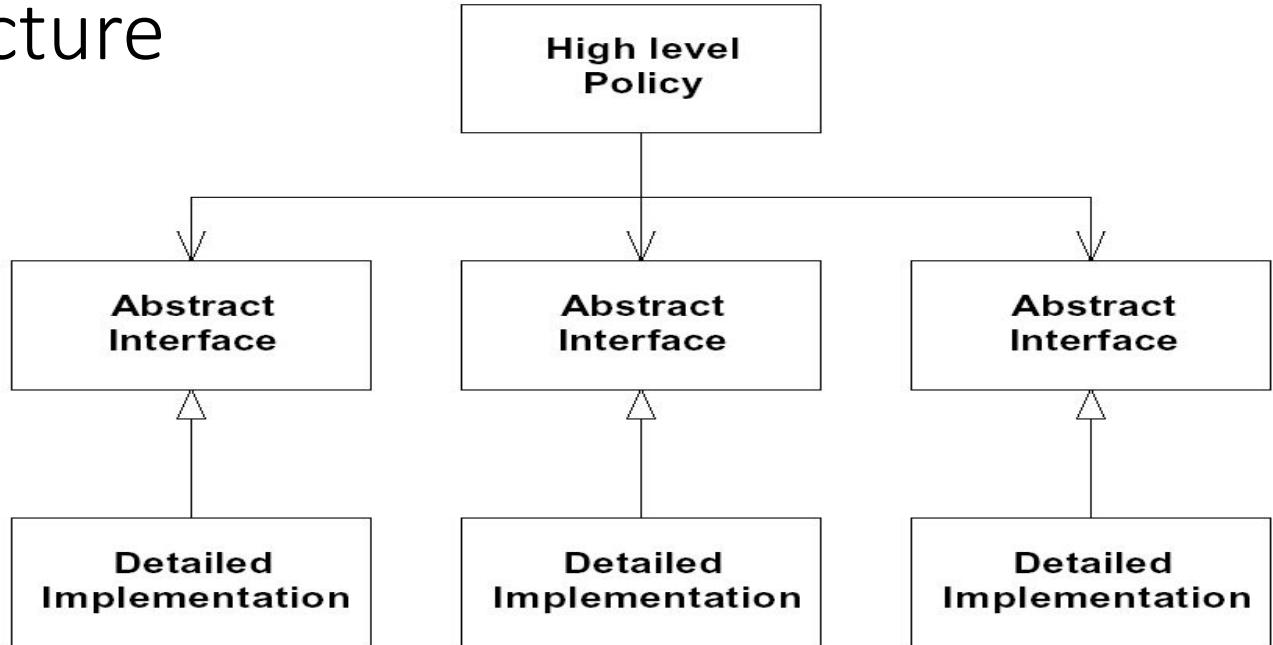
- If the interface of Lamp is changed, Button has to be adjusted, even though the policy that Button represents is not changed!
- To make the high-level policy independent of details we should be able to define it independent of the details of Lamp or any other specific device.

A DIP-Compliant Solution



- Now Button only depends on abstractions!
It can be reused with various classes that implement Switchable.
- Changes in Lamp will not affect Button!
- The dependencies have been inverted:
 - Lamp now has to conform to the interface defined by Button.
- Actually: both depend on an abstraction!

OO Architecture

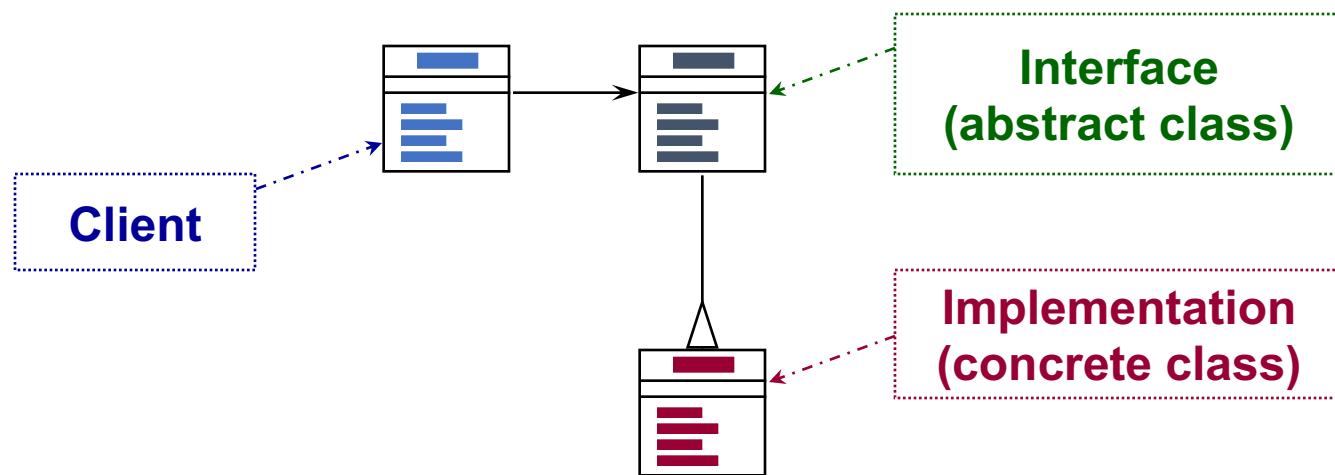


- Good software designs are structured into High-level, low-level modules.
 - High-level modules contain the important policy decisions and business models of an application – The identity of the application.
 - Low-level modules contain detailed implementations of individual mechanisms needed to realize the policy.

DIP Related Heuristic

Design to an interface,
not an implementation!

- Use inheritance to avoid direct bindings to classes:



Design to an Interface

- **Abstract classes/interfaces:**
 - tend to change much less frequently
 - abstractions are ‘hinge points’ where it is easier to extend/modify
 - shouldn’t have to modify classes/interfaces that represent the abstraction (OCP)
- Exceptions
 - Some classes are very unlikely to change;
 - therefore little benefit to inserting abstraction layer
 - Example: String class
 - In cases like this can use concrete class directly
 - as in Java or C++

DIP Related Heuristic

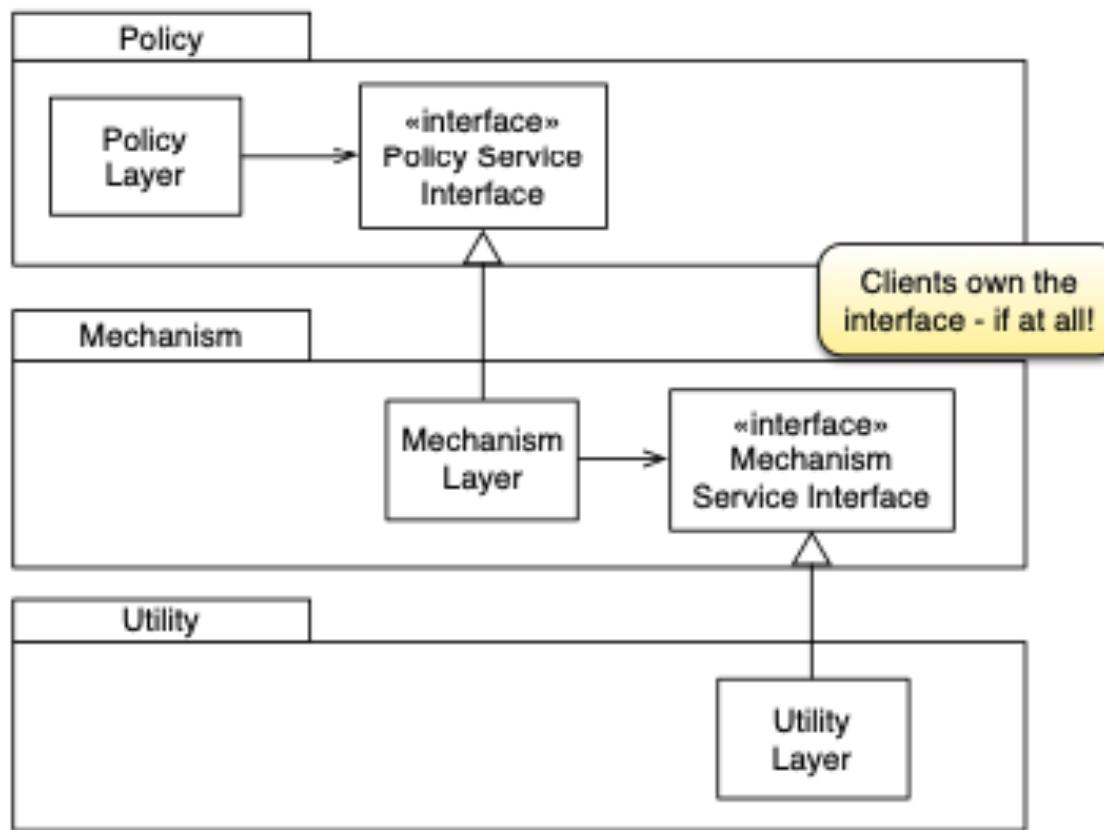
Avoid Transitive Dependencies

- Avoid structures in which higher-level layers depend on lower-level abstractions:
 - In example below, Policy layer is ultimately dependant on Utility layer.



Solution to Transitive Dependencies

- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:



Naive Heuristic for Ensuring DIP

- DO NOT DEPEND ON A CONCRETE CLASS
- All relationships in a program should terminate on an abstract class or an interface.
 - No class should hold a reference to a concrete class.
 - No class should derive from a concrete class.
 - No method should override an implemented method of any of its base classes.

Takeaway

- Traditional structural programming creates a dependency structure in which policy depends on detail.
 - Policies become vulnerable to changes in the details.
- Object-orientation enables to invert the dependency:
 - Policy and details depend on abstractions.
 - Service interfaces are owned by their clients.
- Inversion of dependency is the hallmark of good object-oriented design.

The Founding Principles

- The three principles are closely related
- Violating either LSP or DIP invariably results in violating OCP
 - LSP violations are latent violations of OCP
- It is important to keep in mind these principles to get most out of OO development...