CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

02 Introduction to OOP with Java

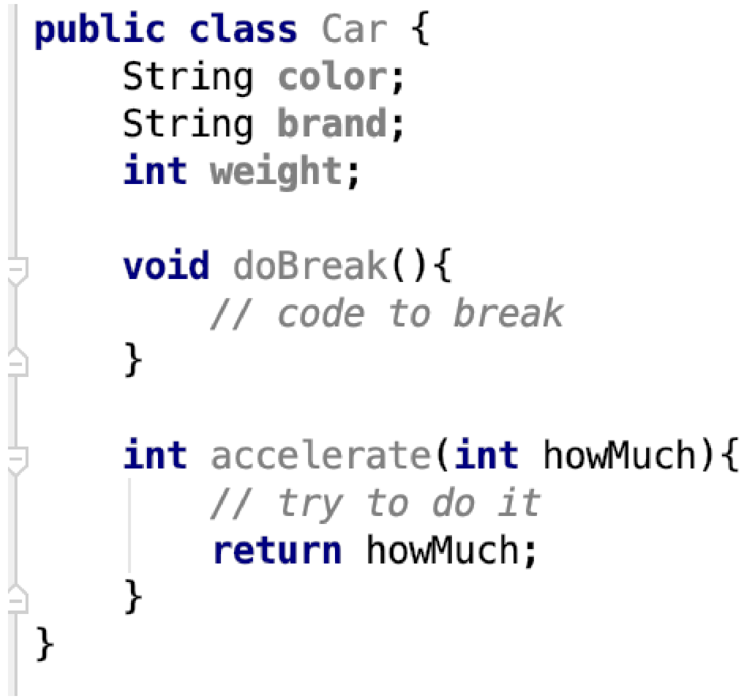
Object

- 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”

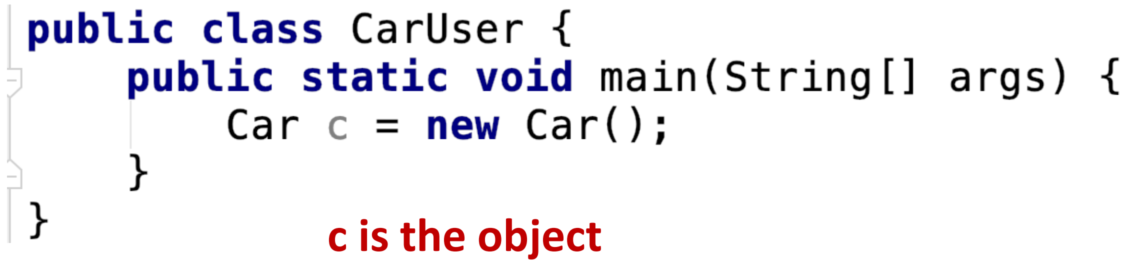
Example

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

This is a Class



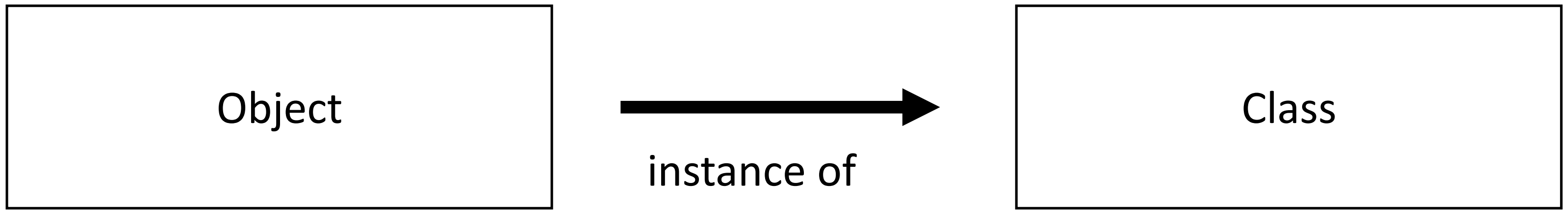
```
public class Car {
    String color;
    String brand;
    int weight;

    void doBreak(){
        // code to break
    }

    int accelerate(int howMuch){
        // try to do it
        return howMuch;
    }
}
```



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

c is the object

What is a Class ?

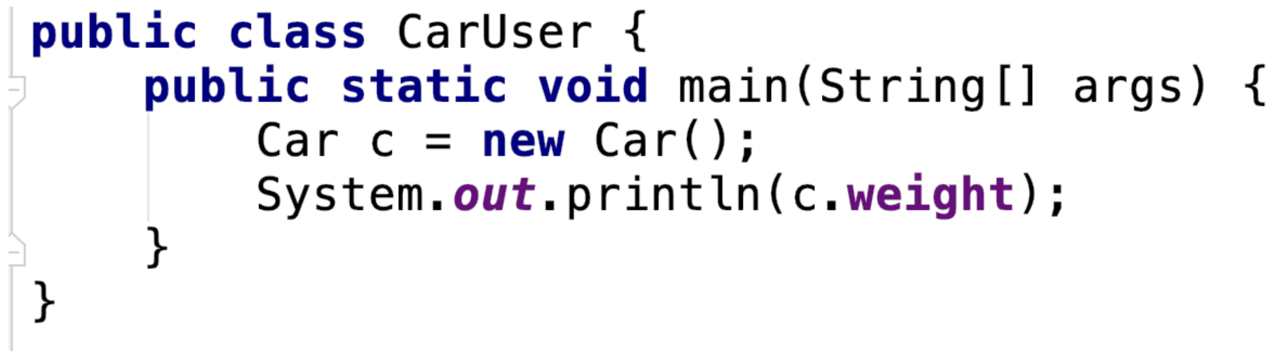


- A class can be considered as a outline of the object
- You can **construct** different objects using that outline
- Constructor is a *special* method
- It’s name is same as class name and it does not return any value

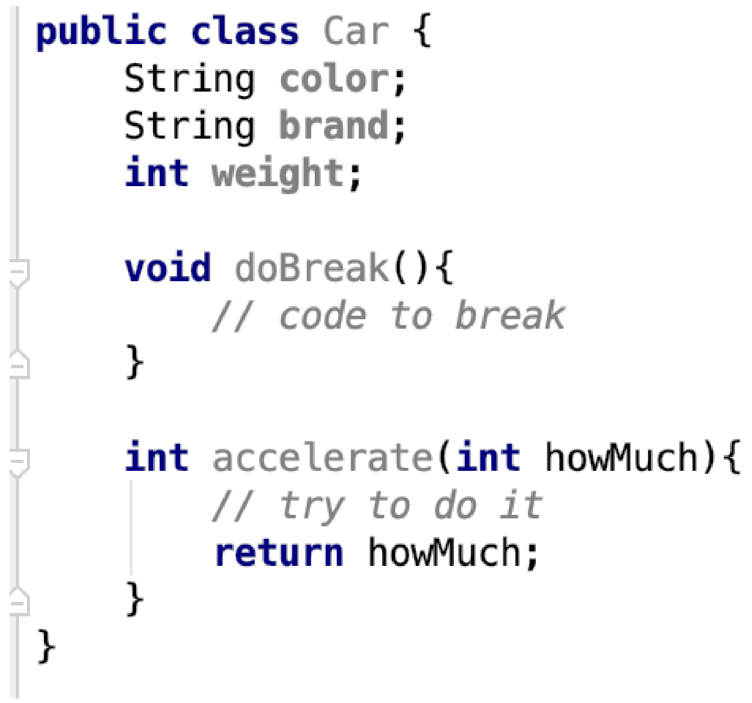
```
public class CarUser {
    public static void main(String[] args) {
        Car c = new Car();
    }
}
```

Default Constructor

What is the output?



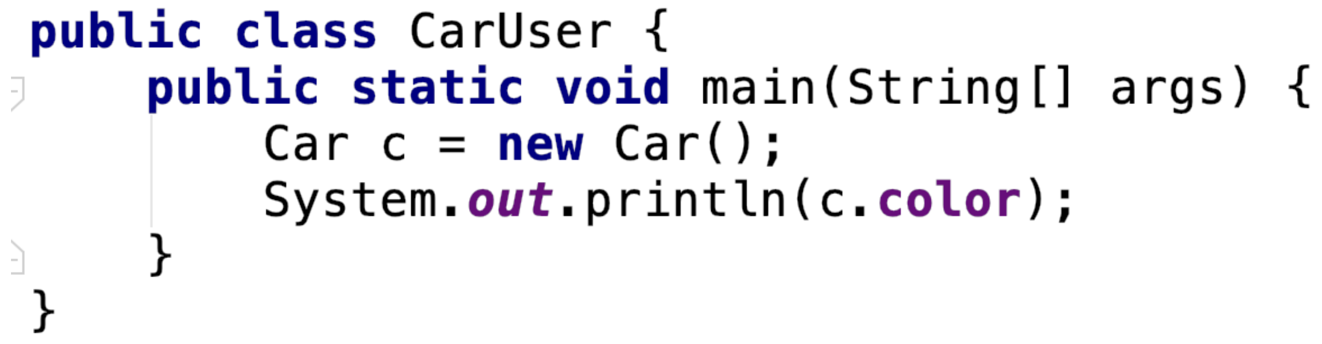
```java
public class CarUser {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println(c.weight);
    }
}
```



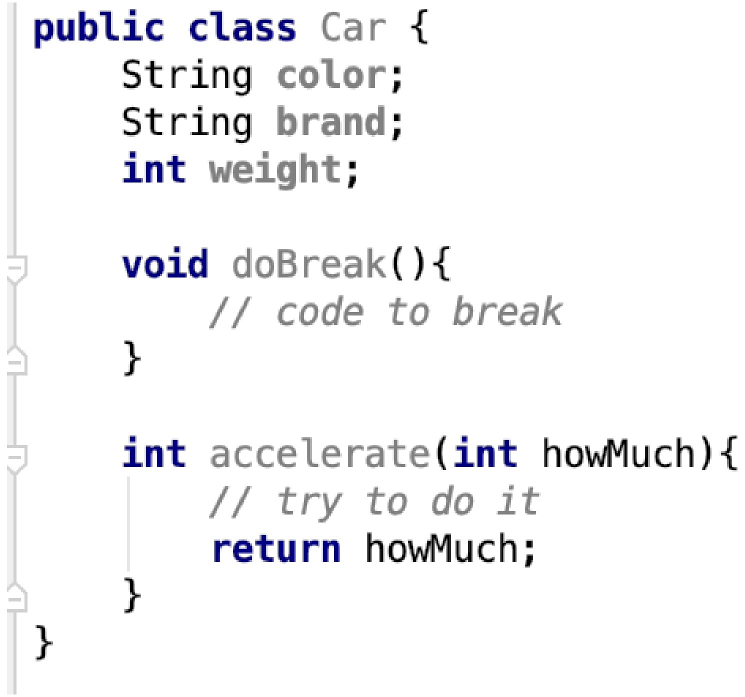
```java
public class Car {
    String color;
    String brand;
    int weight;

    void doBreak(){
        // code to break
    }

    int accelerate(int howMuch){
        // try to do it
        return howMuch;
    }
}
```

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

What is the output?



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



```java
public class Car {
    String color;
    String brand;
    int weight;

    void doBreak(){
        // code to break
    }

    int accelerate(int howMuch){
        // try to do it
        return howMuch;
    }
}
```

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

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

```java
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;
    }
}
```

```java
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

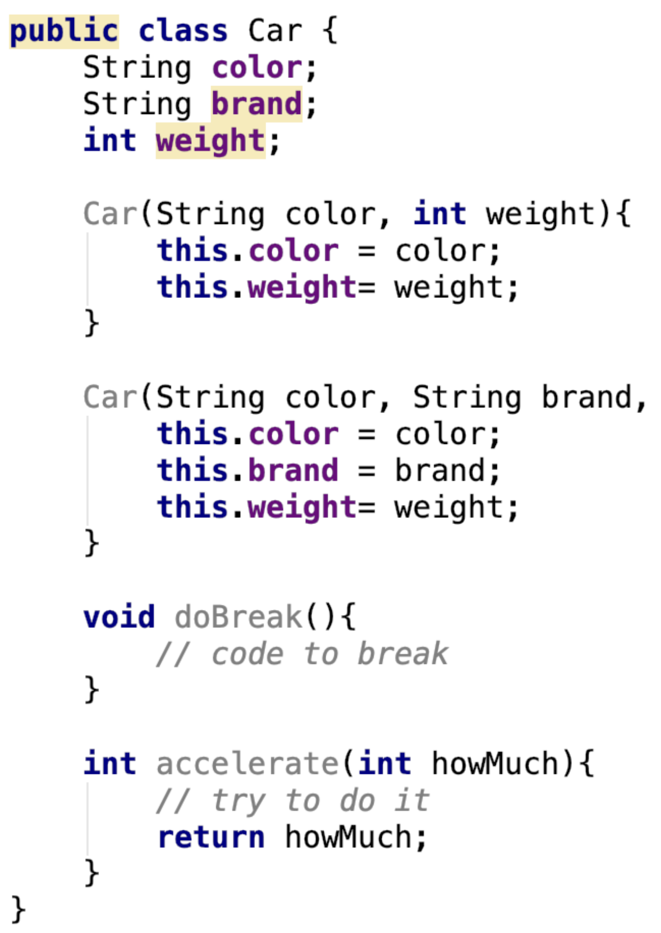
What is the output?

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

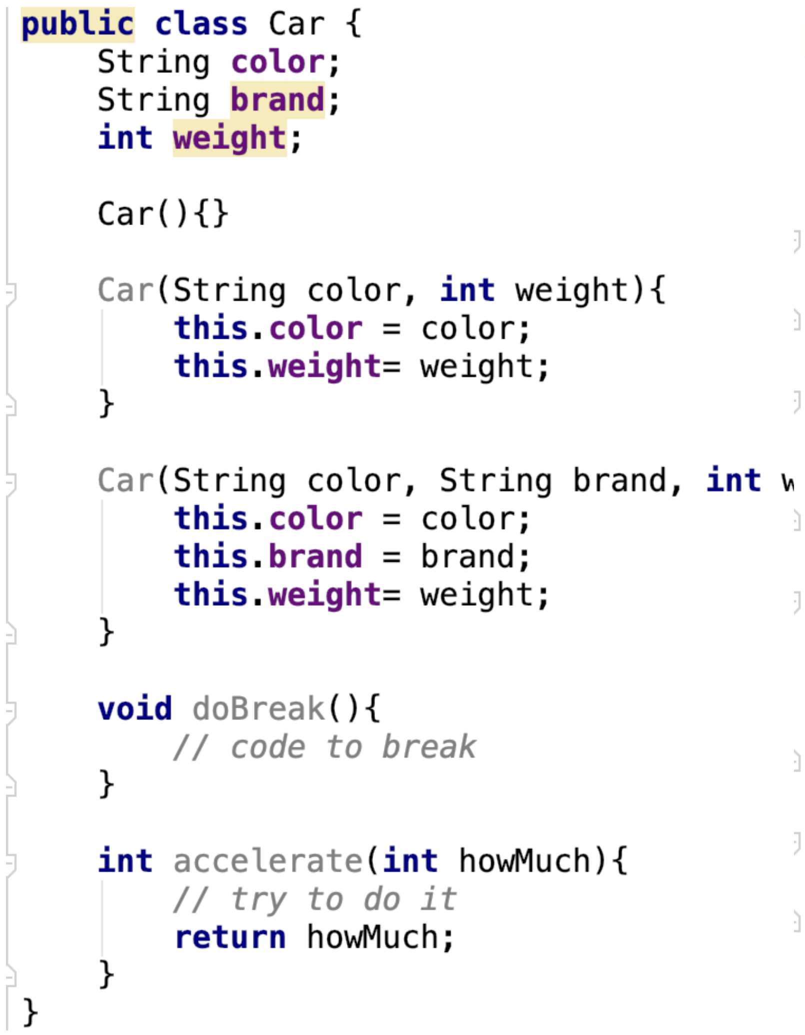
```java
public class Car {
    String color;
    String brand;
    int weight;

    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;
    }
}
```

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

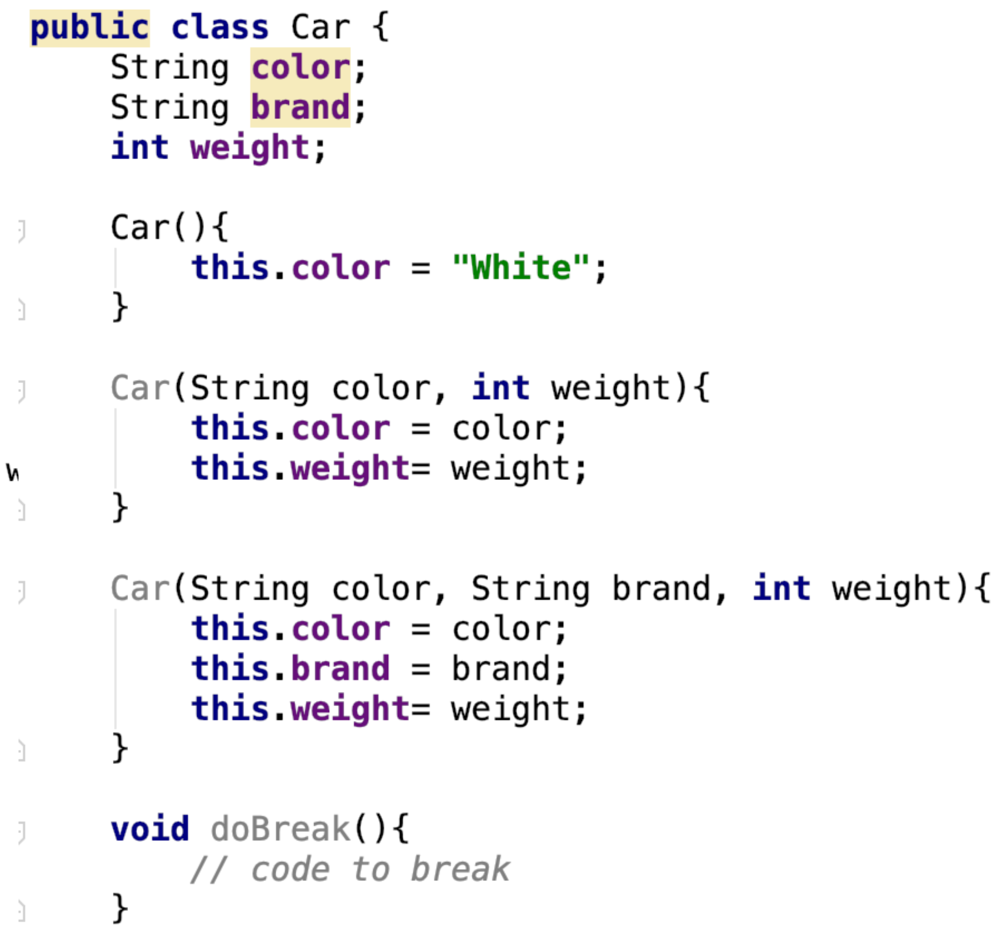
Multiple constructors



```java
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;
    }
}
```



```java
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;
    }
}
```



```java
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()

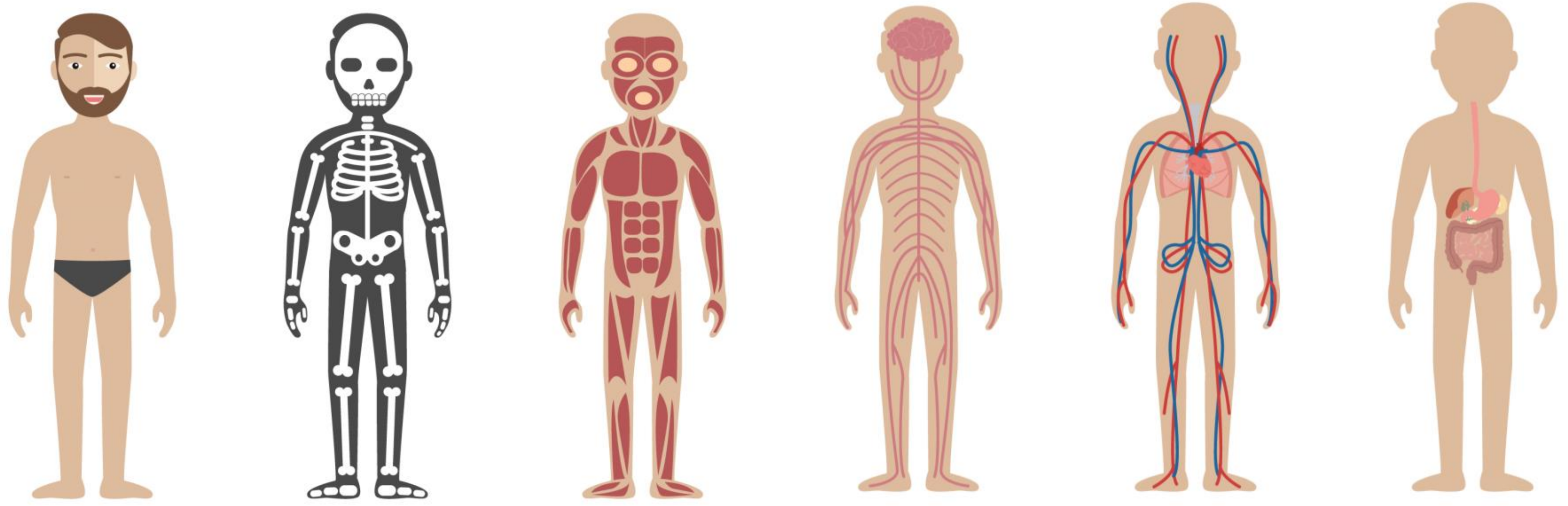
More on constructors later

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



People vector created by photoroyalty - www.freepik.com

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

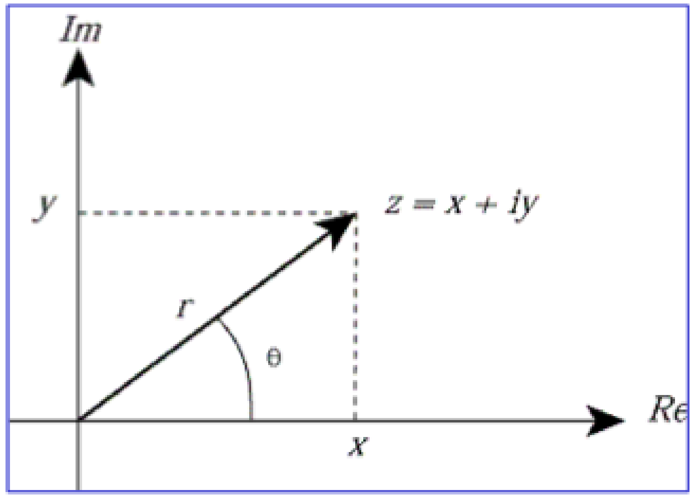
Abstraction

- Java Collections Framework is full of abstractions
  - Set is an abstraction of the mathematical set concept
  - List
  - Map

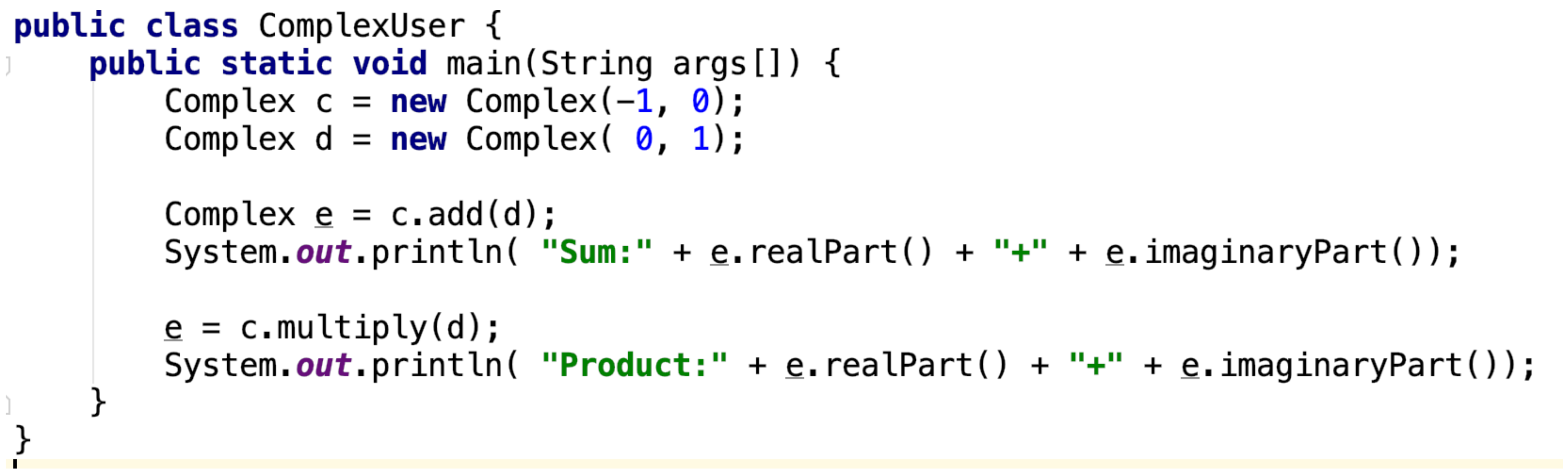
- In java, abstraction is achieved via <u>interfaces</u> and <u>abstract classes</u>
- The real abstraction is achieved via good modelling

Class example – complex numbers

```java
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



```java
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());
    }
}
```

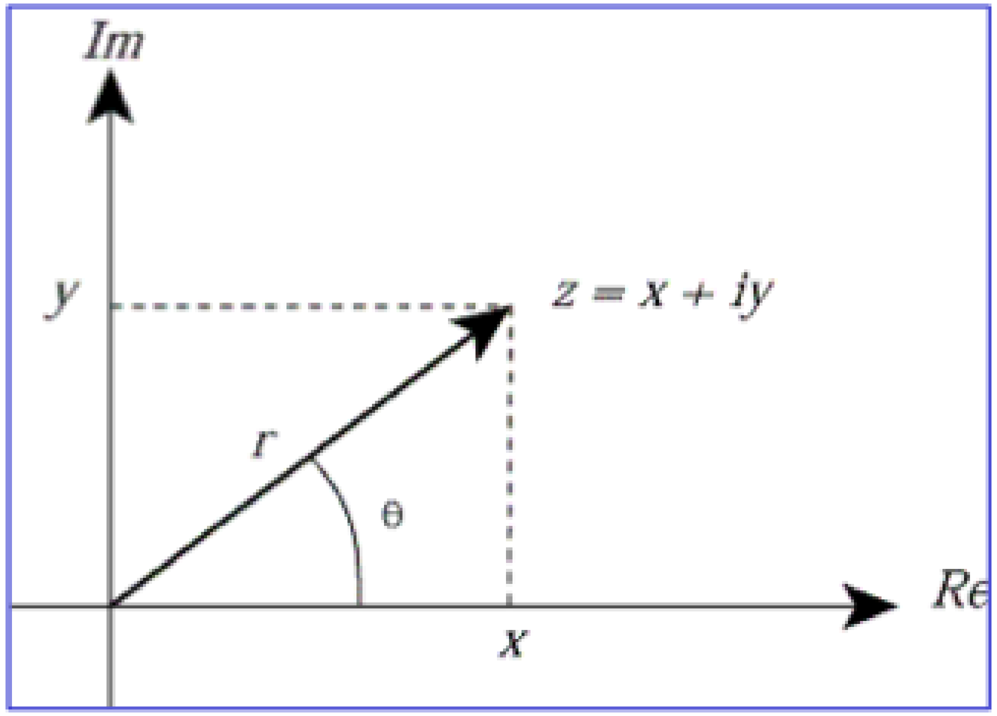
When you run this program, it prints

Sum:-1.0+1.0

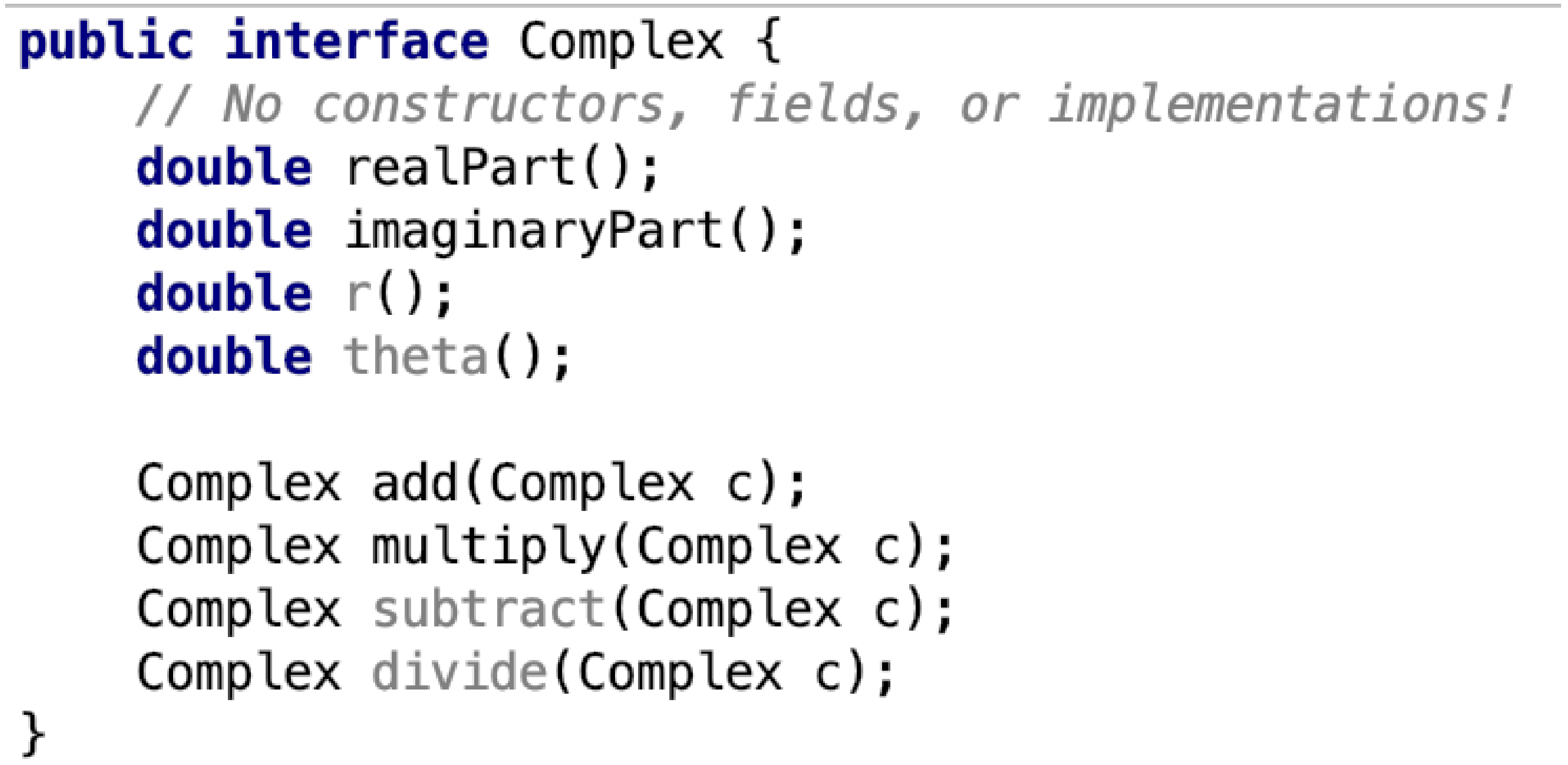
Product:-0.0+-1.0

An abstraction for Complex Numbers

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



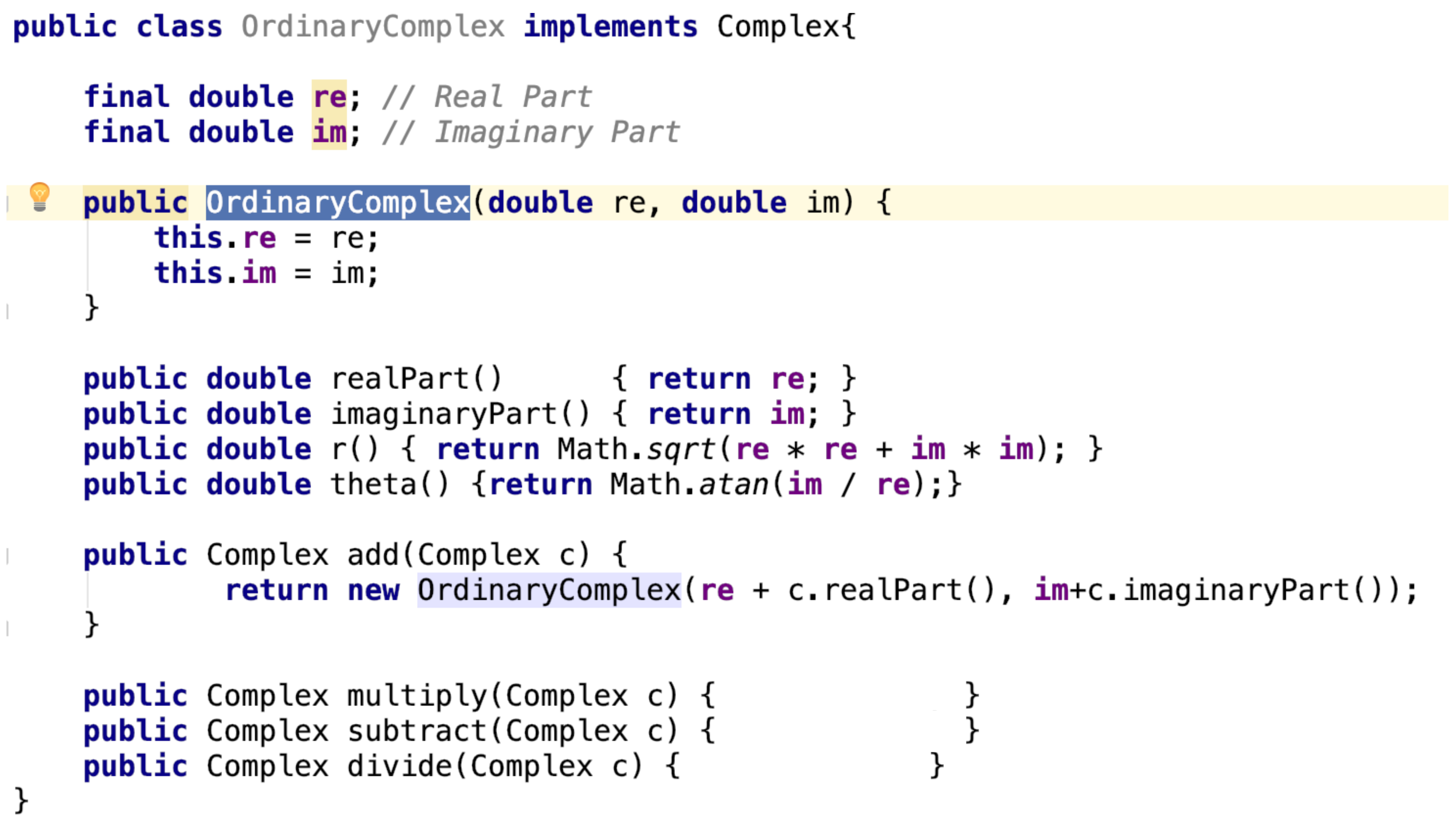
An interface to go with our class



```java
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);
}
```

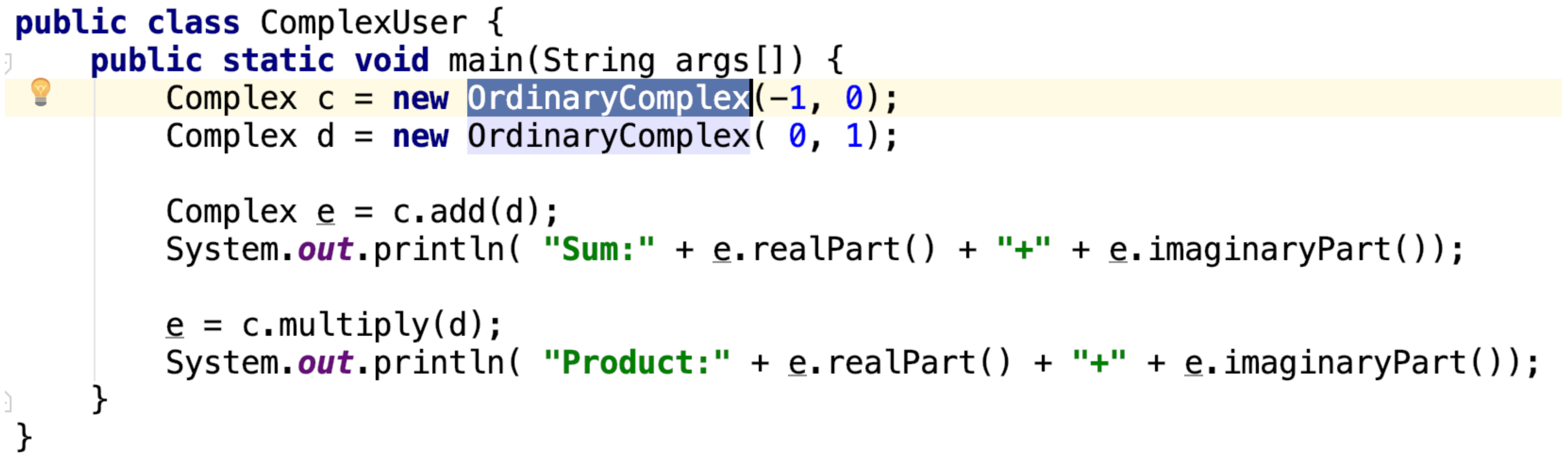
An interface defines but does not implement API

Modifying class to use interface



```java
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

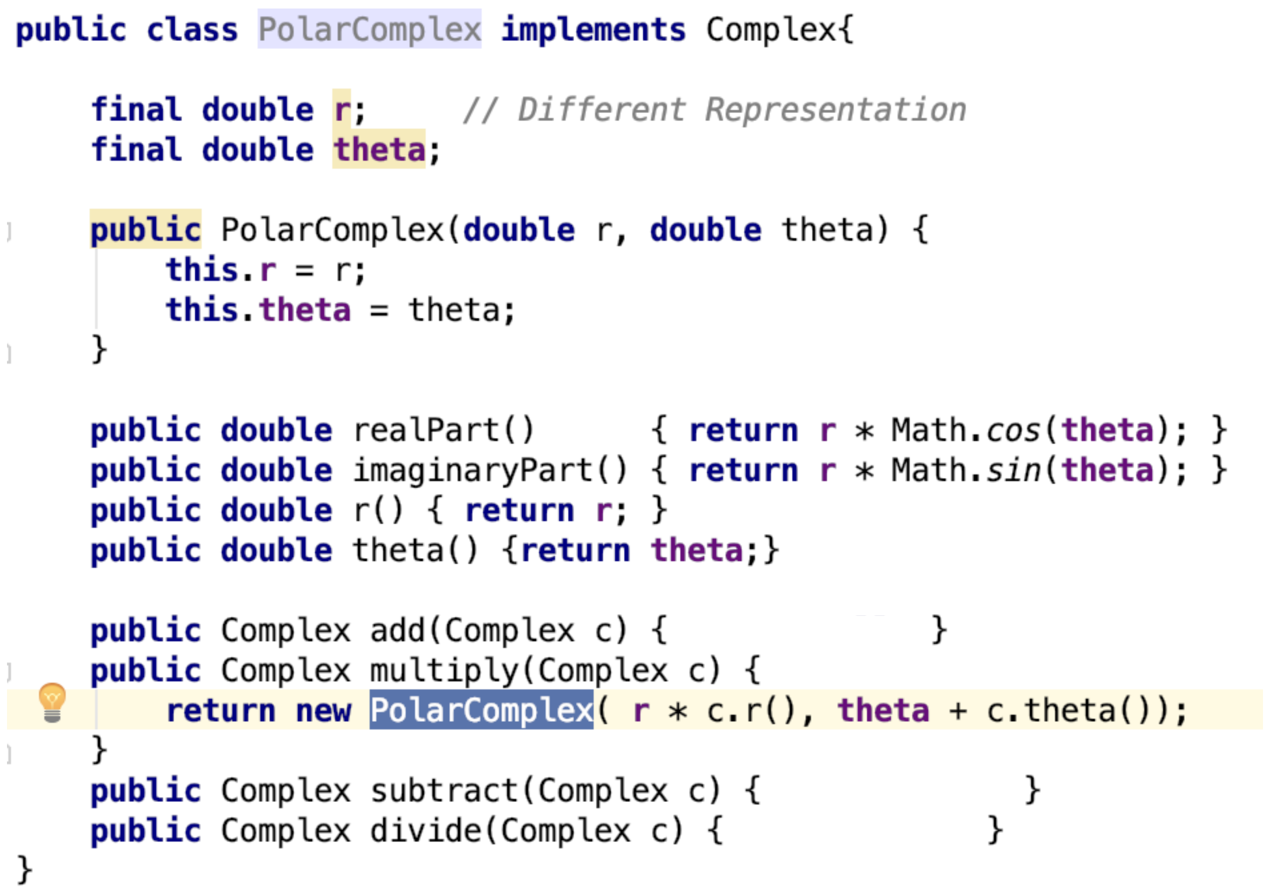


```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex( 0, 1);

        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());
    }
}
```

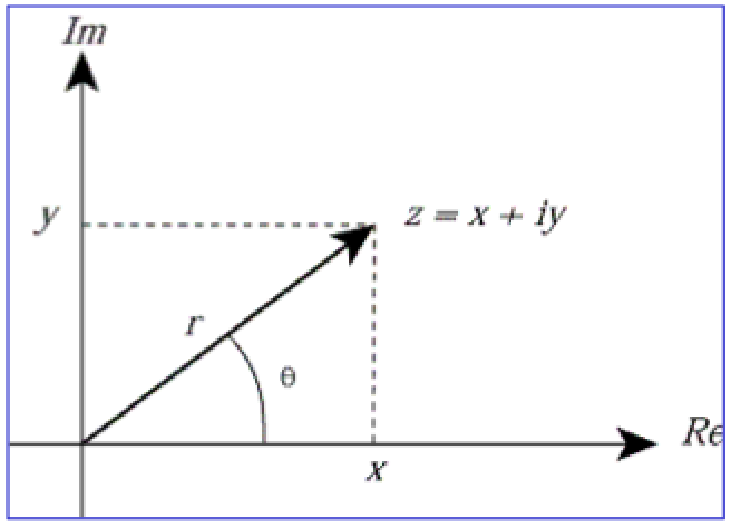
When you run this program, it STILL prints

```
Sum:-1.0+1.0
Product:-0.0+-1.0
```

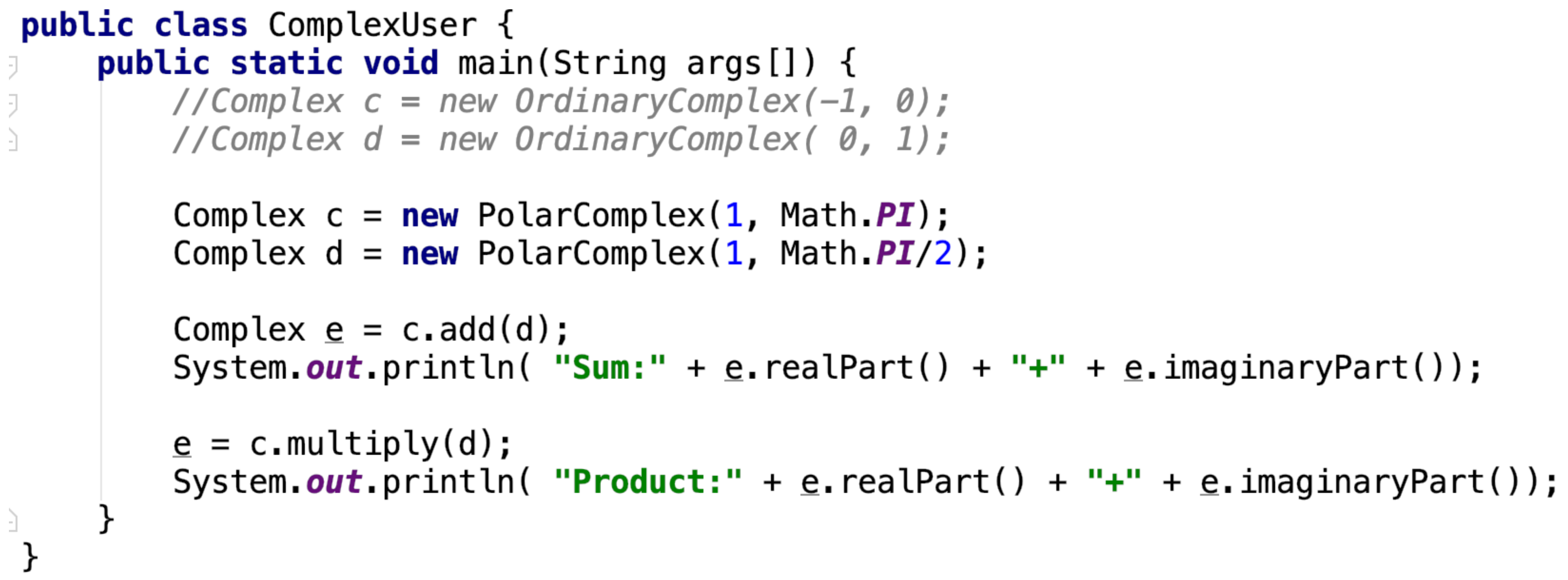
Interface enables multiple implementations



```
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

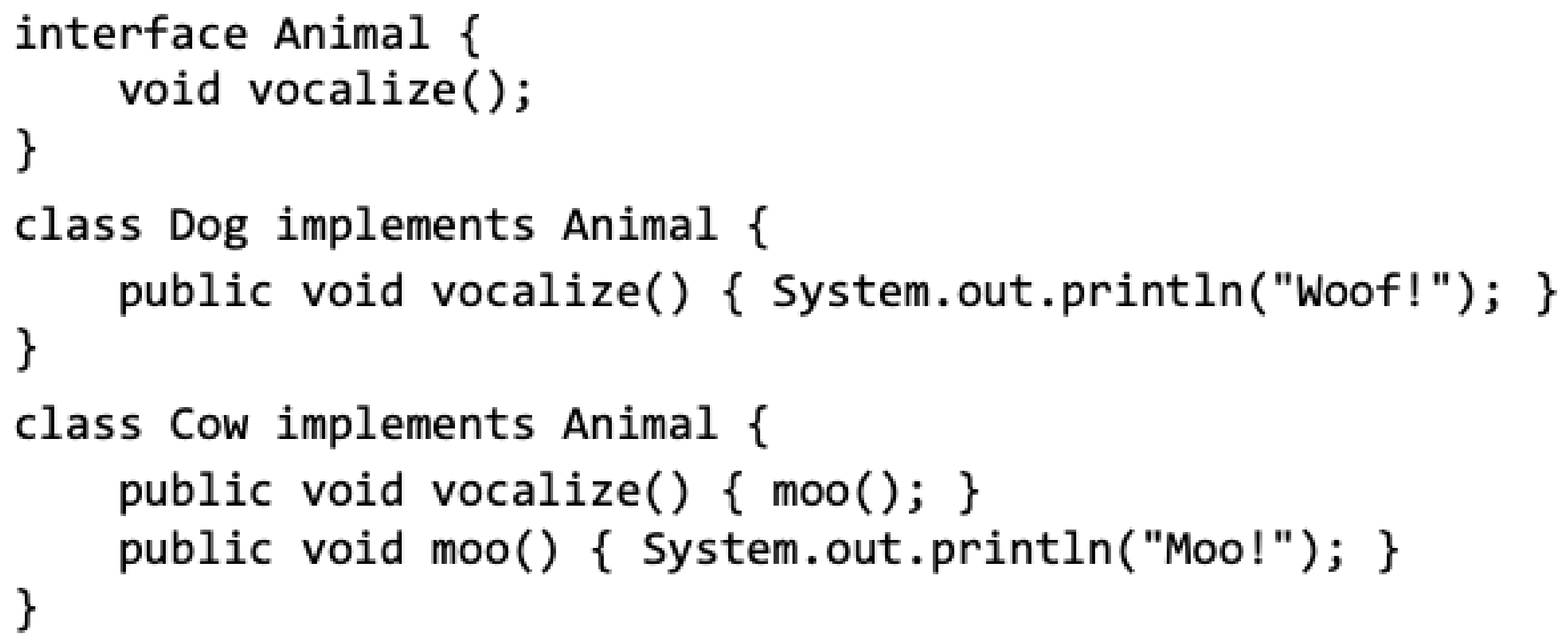
- An interface defines but does not implement API
- Multiple implementations of an API can coexist
  - Multiple classes can implement the same API
  - OrdinaryComplex, PolarComplex implement the same API
- In Java, an API is specified by *class* or *interface*
  - Class provides an API and an implementation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



```java
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

Object Oriented Programming Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

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 pirinciple 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

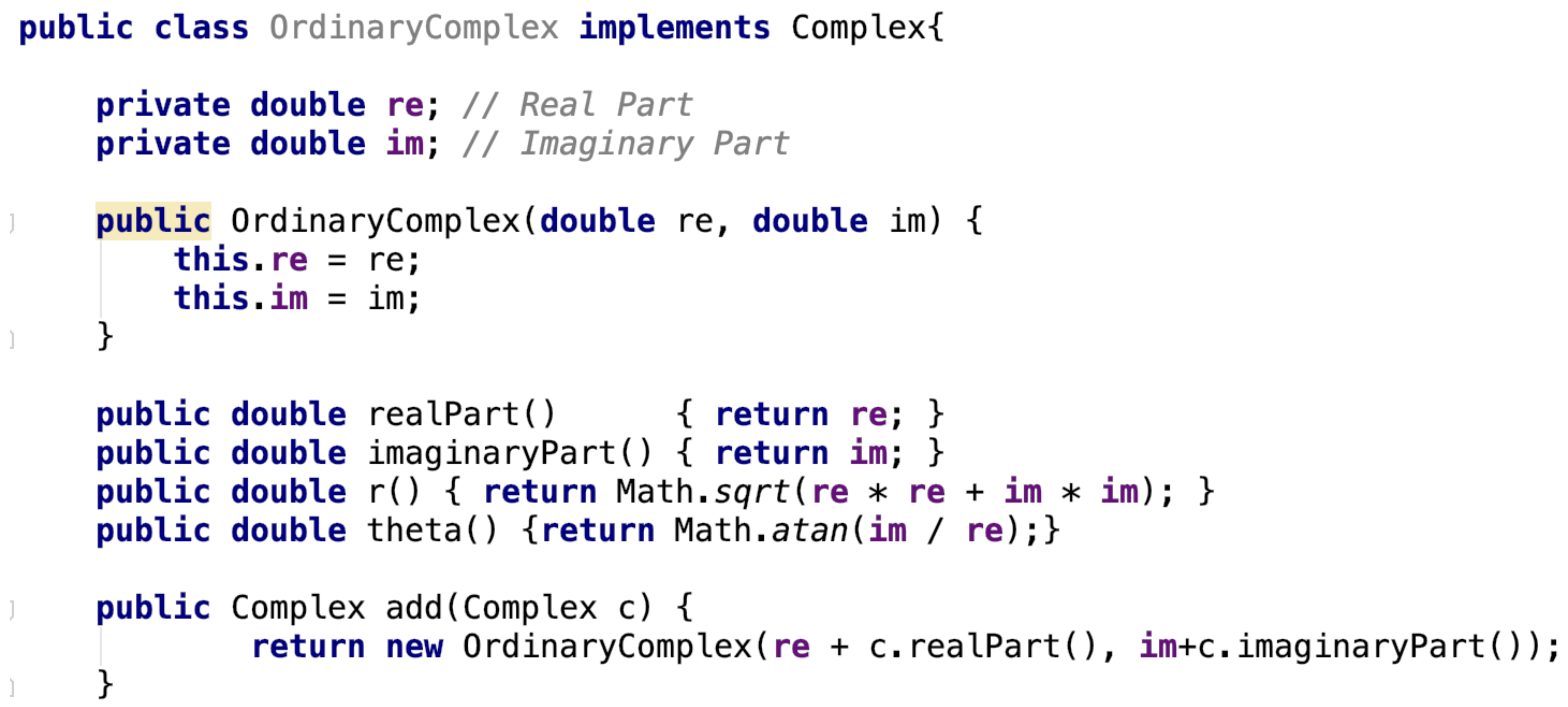
Encapsulation with interfaces

- 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

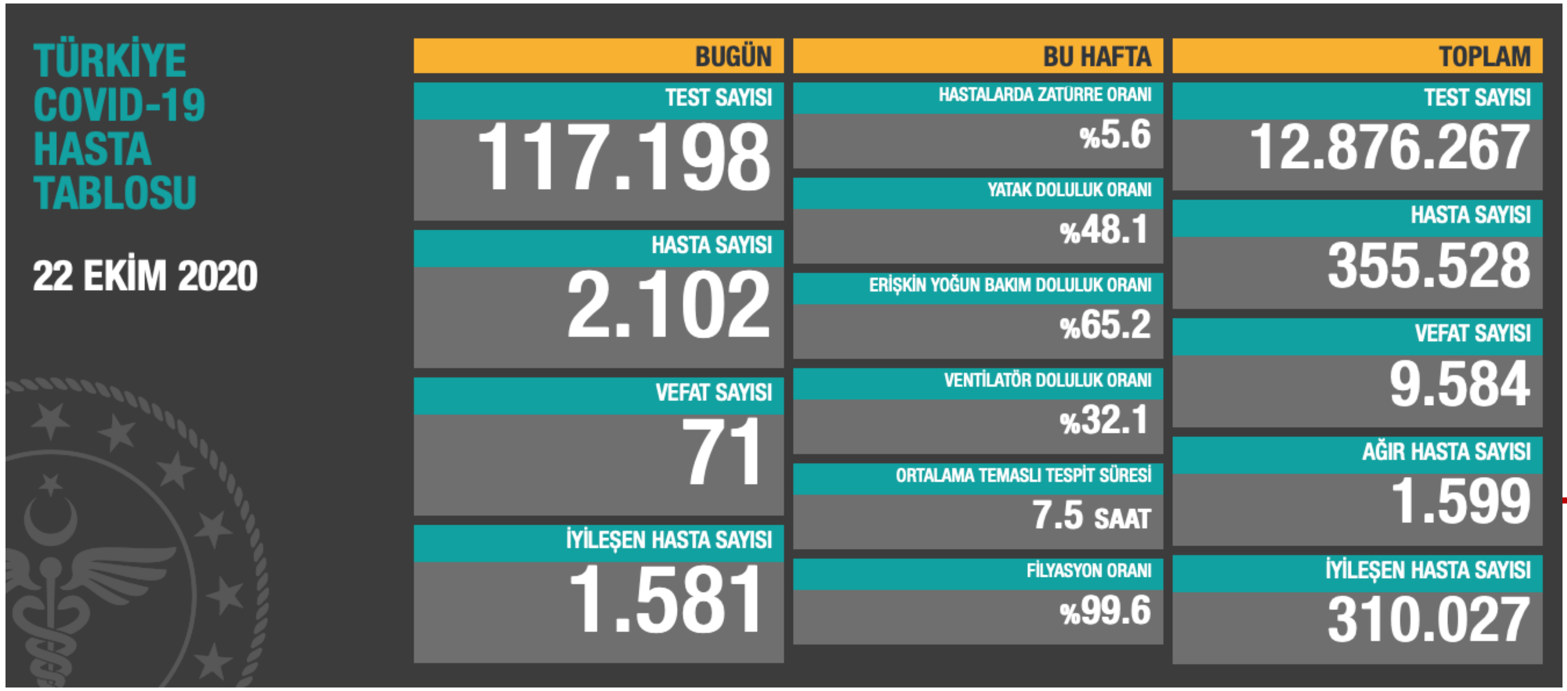


```java
public class OrdinaryComplex implements Complex{

    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

TEST SAYISI
117.198

HASTA SAYISI
2.102

VEFAT SAYISI
71

İYİLEŞEN HASTA SAYISI
1.581

BU HAFTA

HASTALARDA ZATÜRRE ORANI
%5.6

YATAK DOLULUK ORANI
%48.1

ERİŞKİN YOĞUN BAKIM DOLULUK ORANI
%65.2

VENTİLATÖR DOLULUK ORANI
%32.1

ORTALAMA TEMASLI TESPİT SÜRESİ
7.5 SAAT

FİLYASYON ORANI
%99.6

TOPLAM

TEST SAYISI
12.876.267

HASTA SAYISI
355.528

VEFAT SAYISI
9.584

AĞIR HASTA SAYISI
1.599

İYİLEŞEN HASTA SAYISI
310.027

$\rightarrow$ covidData.getSevereCases();

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.

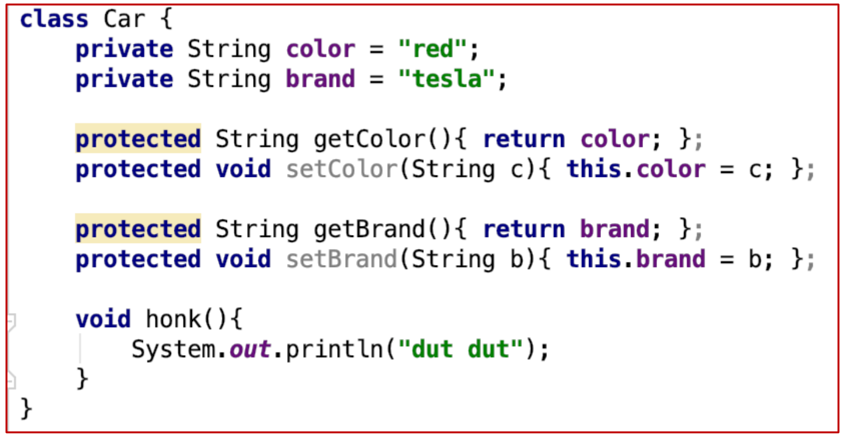
Inheritance In Java

- 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

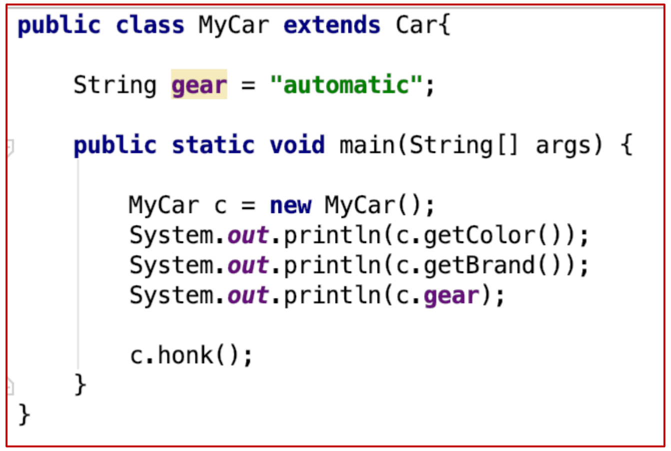
```
class A extends B {

}
```

- 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.

Example



```java
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");
    }
}
```



```java
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:

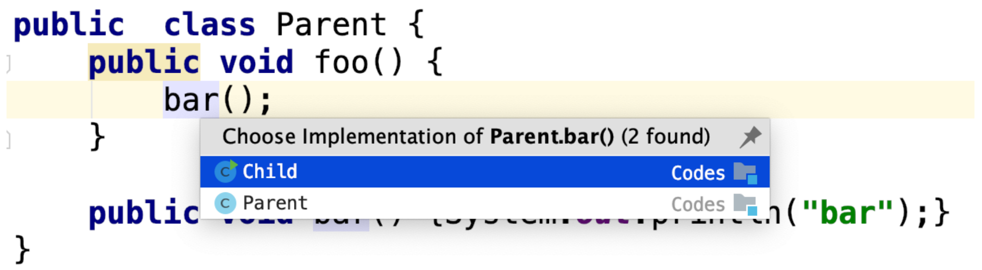
```
red
tesla
automatic
dut dut
```

What is the output?

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

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

```java
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.

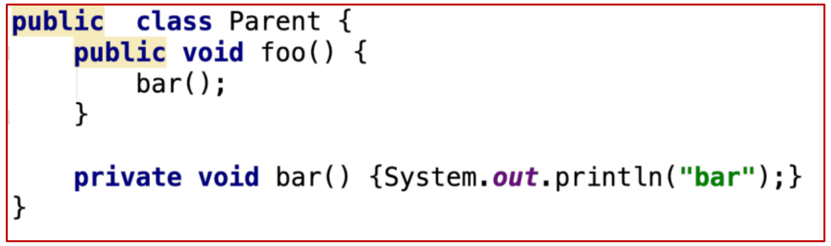


```java
public  class Parent {
    public void foo() {
        bar();
    }

    public                              n("bar");}
}
```

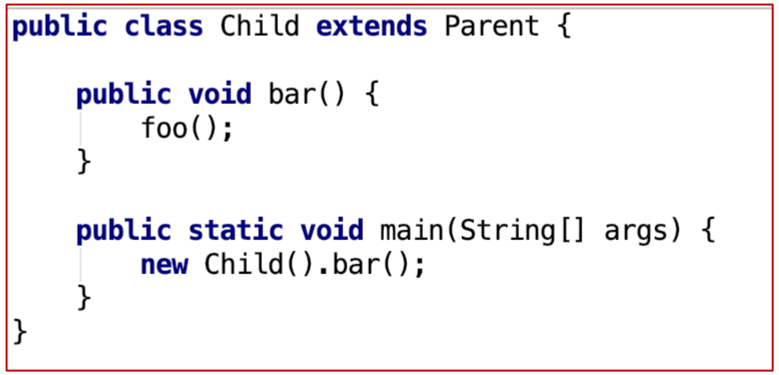
```
Choose Implementation of Parent.bar() (2 found)
Child                                   Codes
Parent                                  Codes
```

Private members are not inherited

- What is the output?



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

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



```java
public class Child extends Parent {

    public void bar() {
        foo();
    }

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

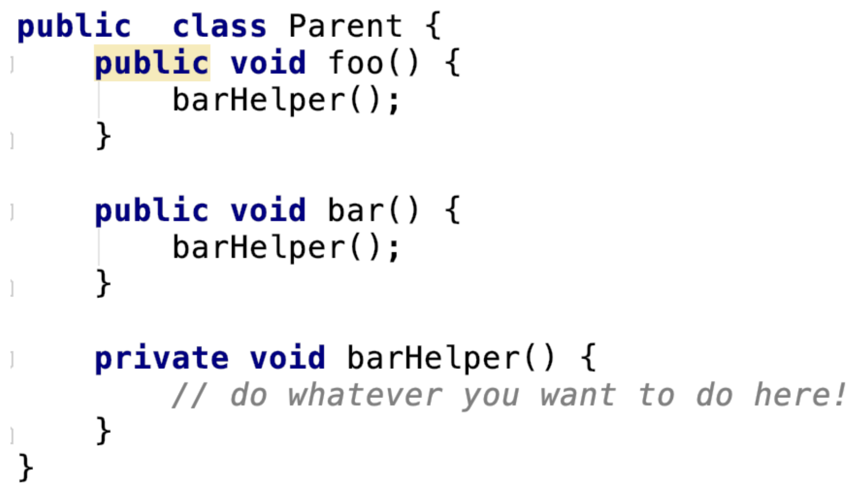
bar

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

Let’s assume Parent’s bar() has to be public

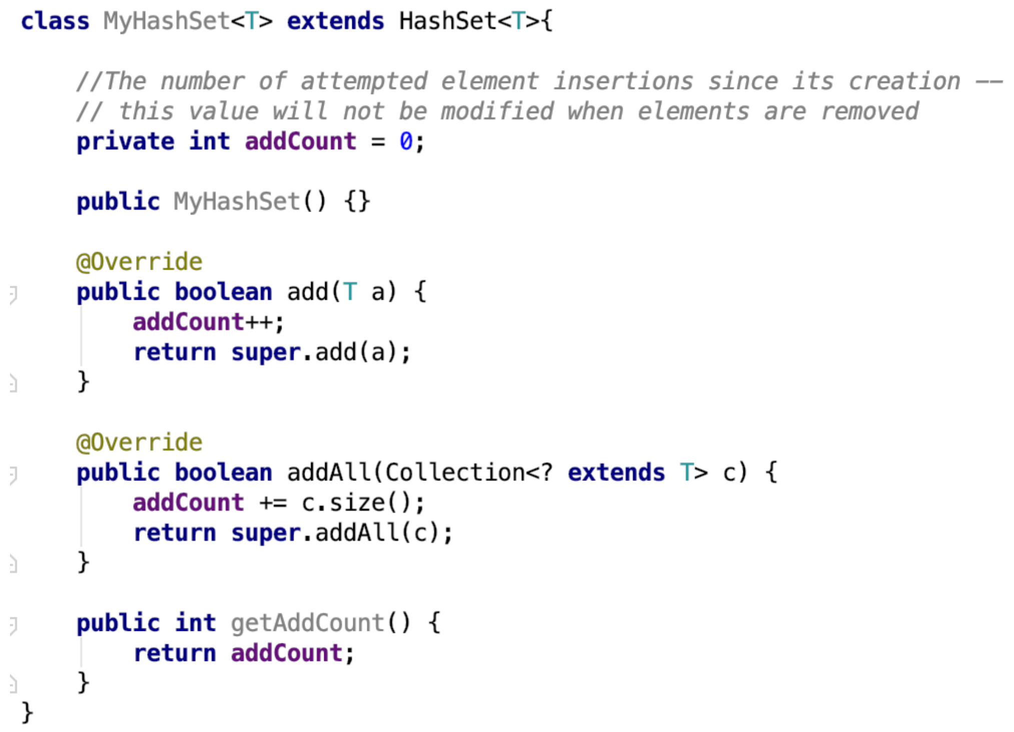
- For some reason!

- 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.



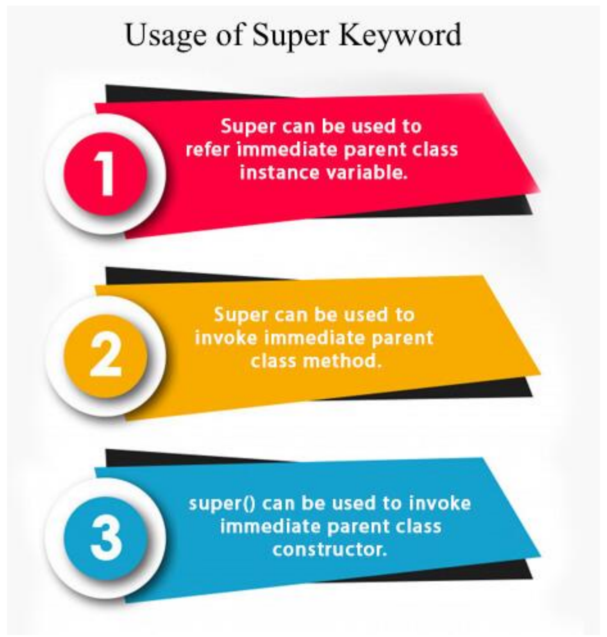
```java
public  class Parent {
    public void foo() {
        barHelper();
    }

    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;
    }
}
```

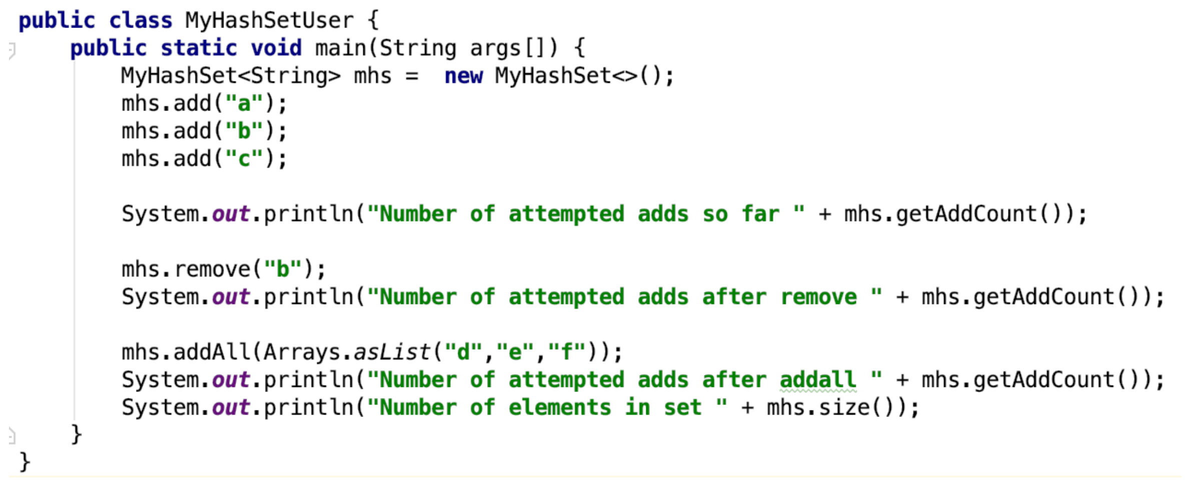
Side Note



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?

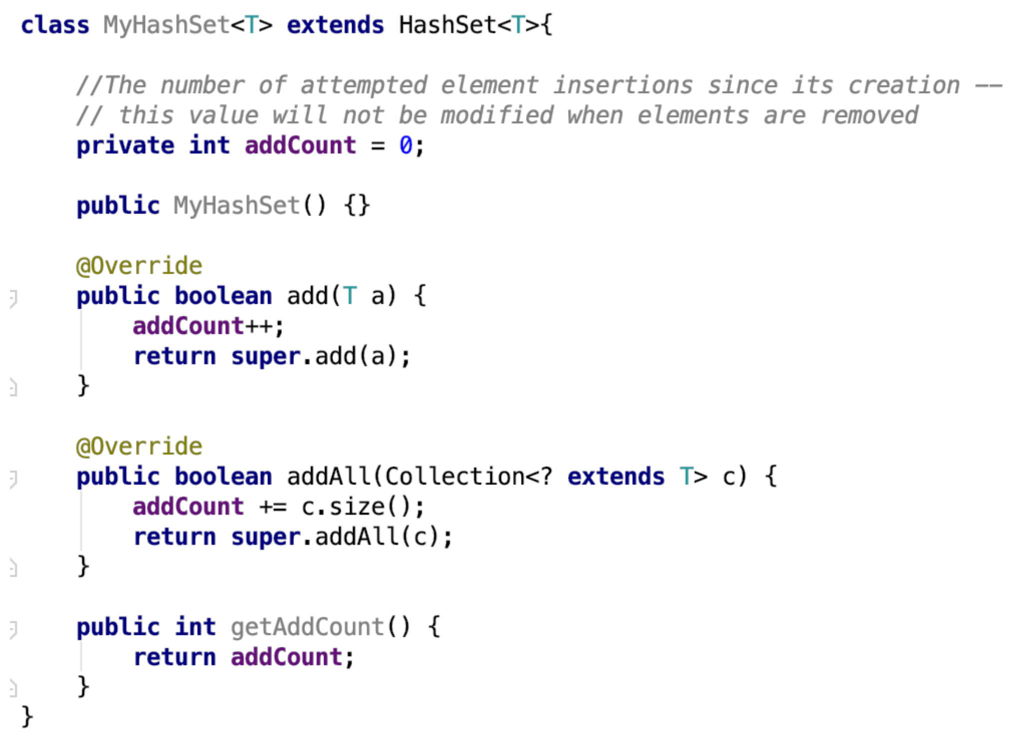


```java
public class MyHashSetUser {
    public static void main(String args[]) {
        MyHashSet<String> mhs =  new MyHashSet<>();
        mhs.add("a");
        mhs.add("b");
        mhs.add("c");

        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
Number of elements in set 5
```

$\longrightarrow$ What the heck!

My Hash Set



```java
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;
    }
}
```

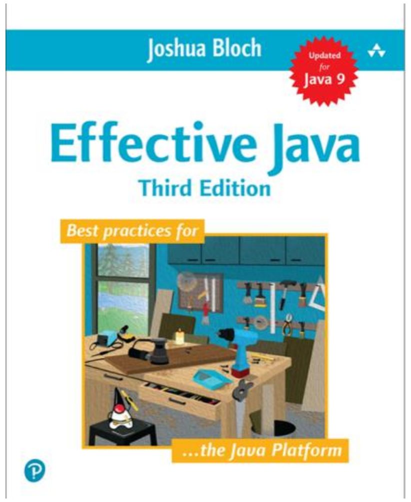
The cause of the problem is that: In the implementation of HashSet, addAll calls the add method. Therefore, we are incrementing addCount too many times in calls to addAll.

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**



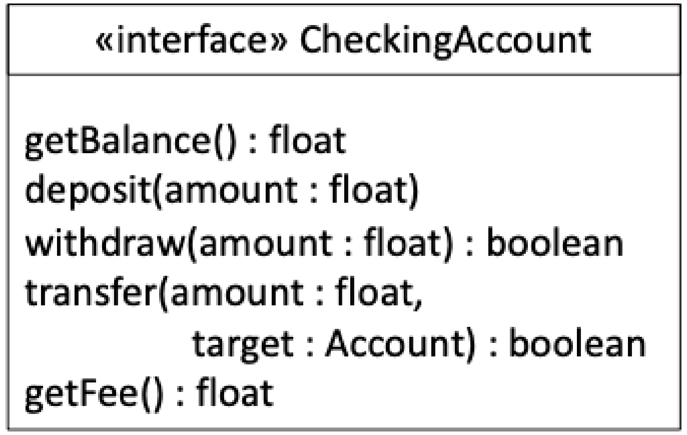
Abstract classes

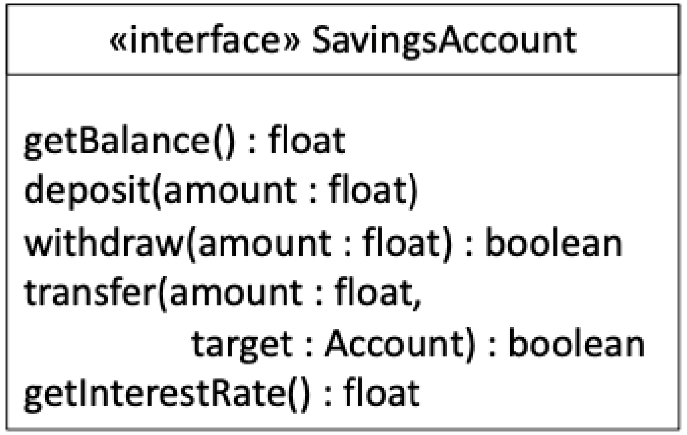
- We have seen

• Interfaces	no implementation
• (Concrete) classes	full implementation

- Abstract classes are somewhere in between
- Java classes can
  - **implements** multiple interfaces
  - **extends** a single class

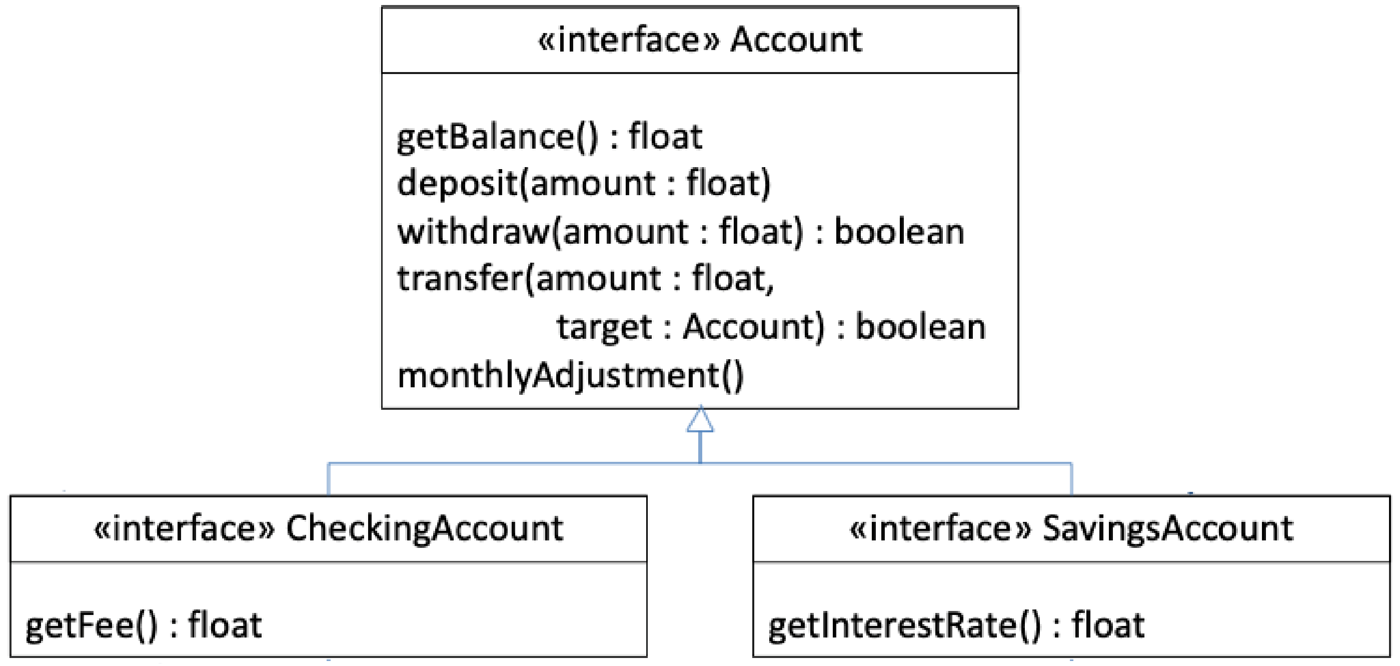
Example: Account Types


«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!

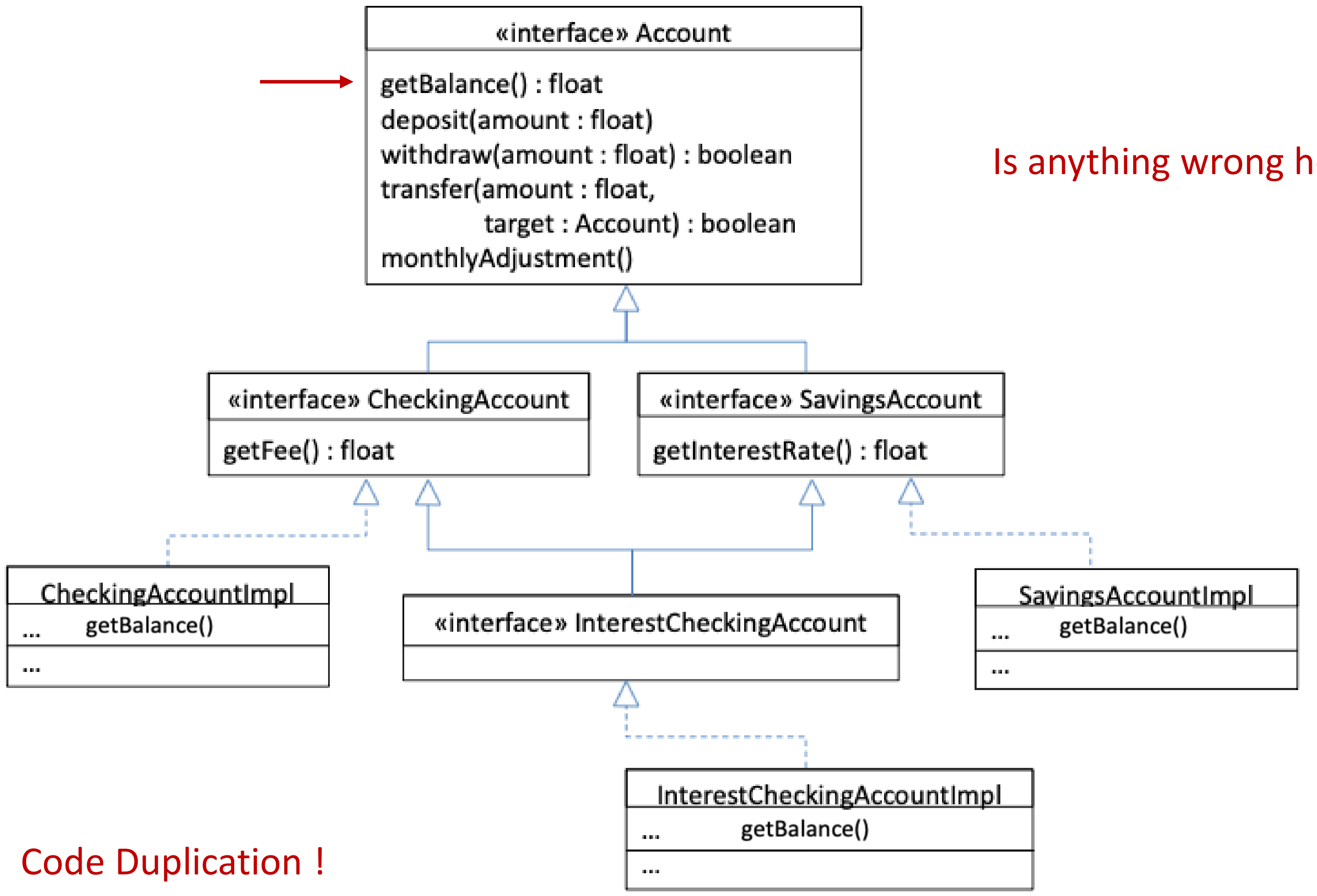
Better Design


«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean
monthlyAdjustment()
«interface» CheckingAccount
getFee() : float
«interface» SavingsAccount
getInterestRate() : float

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



Is anything wrong here?

Code Duplication !

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

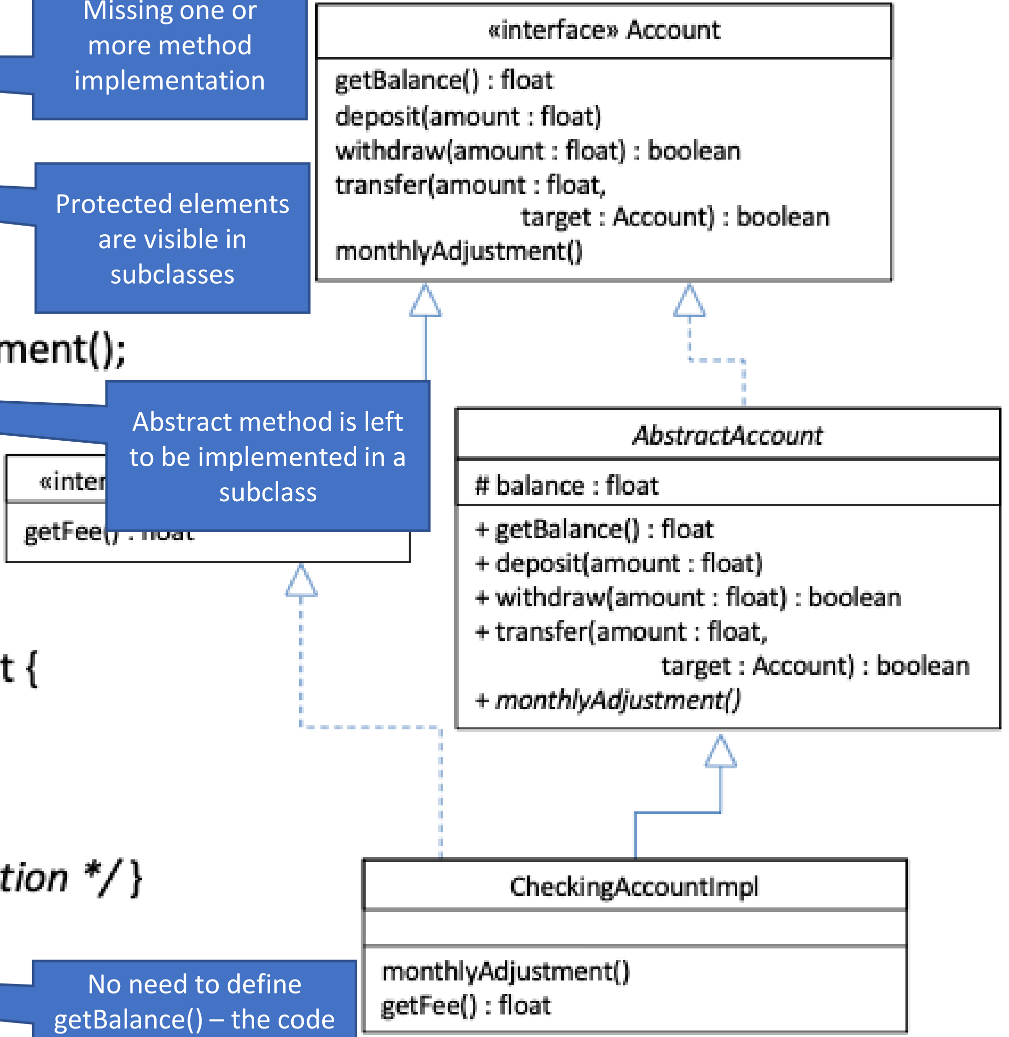
```
public abstract class AbstractAccount
        implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

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

Missing one or more method implementation

Protected elements are visible in subclasses

Abstract method is left to be implemented in a subclass

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 <u>not</u> 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

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

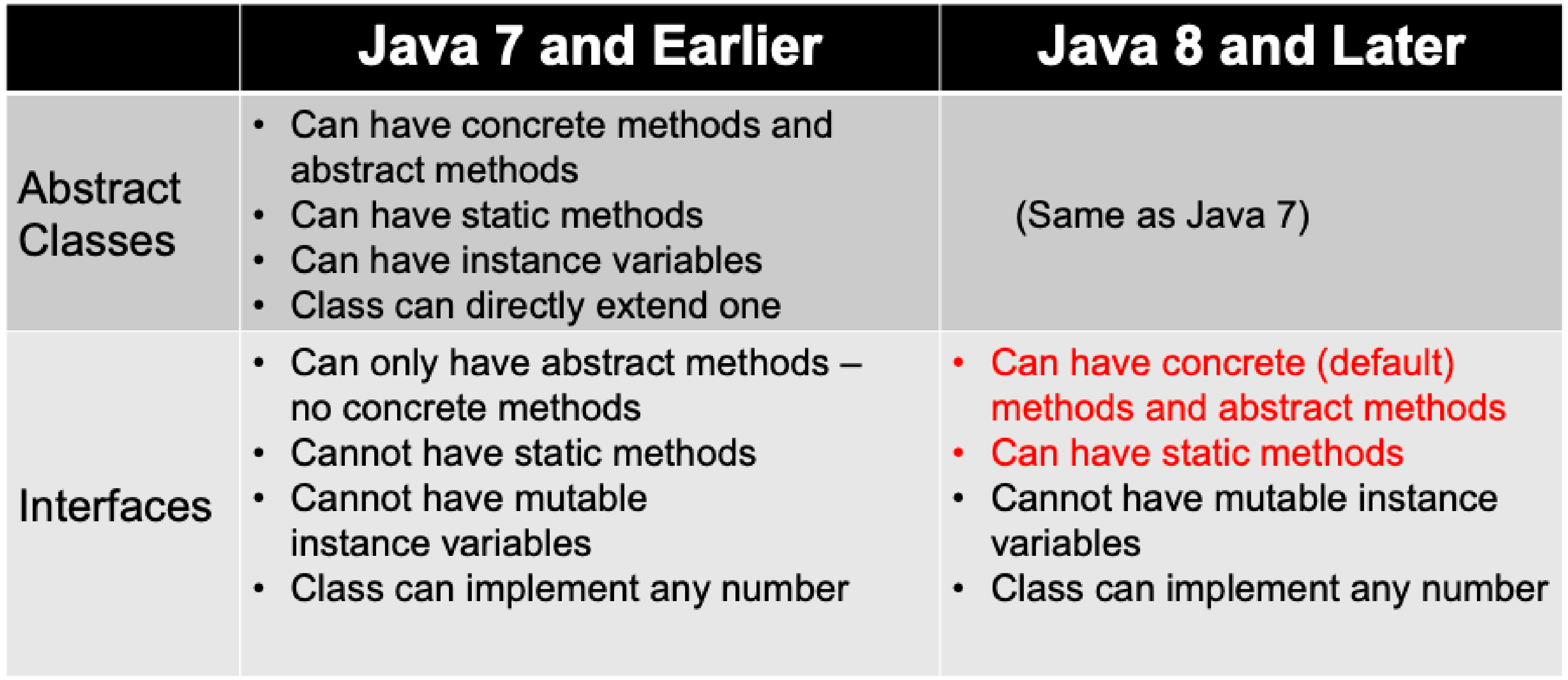
give your new class a private field that references an instance of the existing class.

```
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?

A Note on Interfaces



	Java 7 and Earlier	Java 8 and Later
Abstract Classes	• Can have concrete methods and abstract methods • Can have static methods • Can have instance variables • Class can directly extend one	(Same as Java 7)
Interfaces	• Can only have abstract methods – no concrete methods • Cannot have static methods • Cannot have mutable instance variables • Class can implement any number	• Can have concrete (default) methods and abstract methods • Can have static methods • Cannot have mutable instance variables • Class can implement any number

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.

Object Oriented Programming Concepts

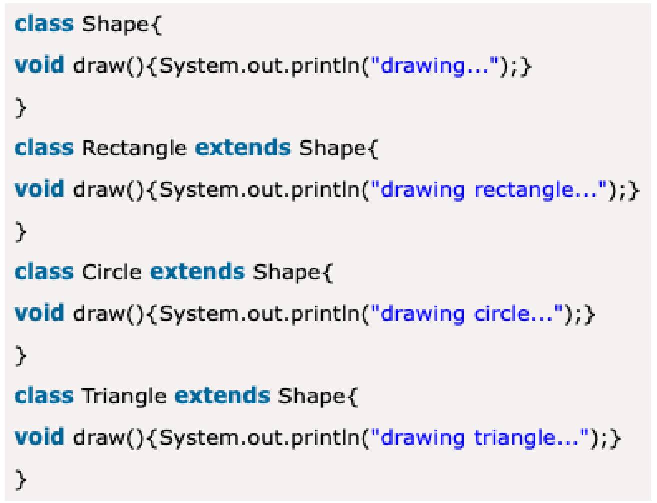
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Polymorphism

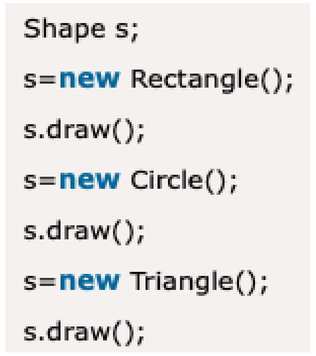
- Polymorphism allows us to perform a single action in different ways
- Polymorphism is the capability of a method to do different things based on the object that it is acting upon
- Polymorphism allows you define one interface and have multiple implementations

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

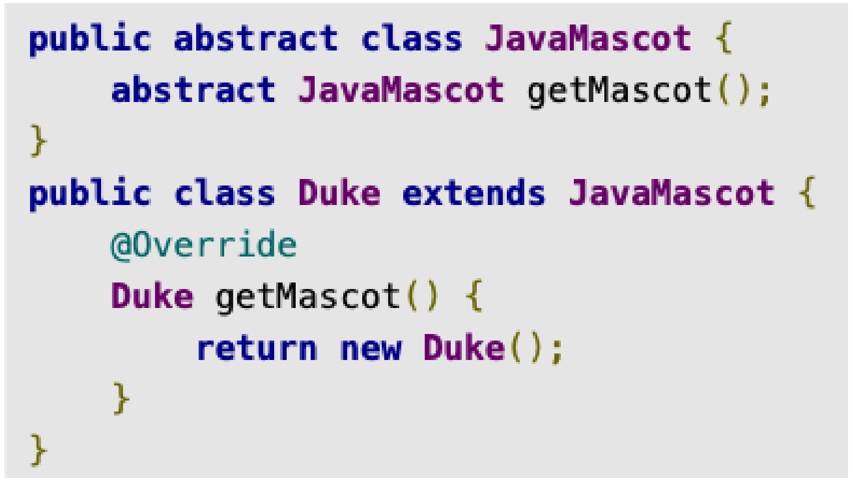
```java
abstract class SweetProducer {
    public abstract void produceSweet();
}

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

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

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

```java
// The creator class knows nothing about the runtime type
class SweetCreator {
    private SweetProducer[] sweetProducer;
    public SweetCreator(SweetProducer[] sweetProducer) {
        this.sweetProducer = sweetProducer;
    }
    public void createSweets() {
        for(SweetProducer sweet: sweetProducer)
            sweet.produceSweet();
    }
}

// The client class knows nothing about implementation details
public class Main {
    public static void main(String[] args) {
        SweetCreator sweetCreator = new SweetCreator(
            new SweetProducer[]{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.



```java
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<>();

Achtung!

- It’s a common mistake to think it’s possible to invoke a specific method without using casting.
- Another mistake is 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.
- Also 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<?> lst) {
        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 5 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
  - StringBuffer (thread-safe) and StringBuilder (not t/s) are 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:

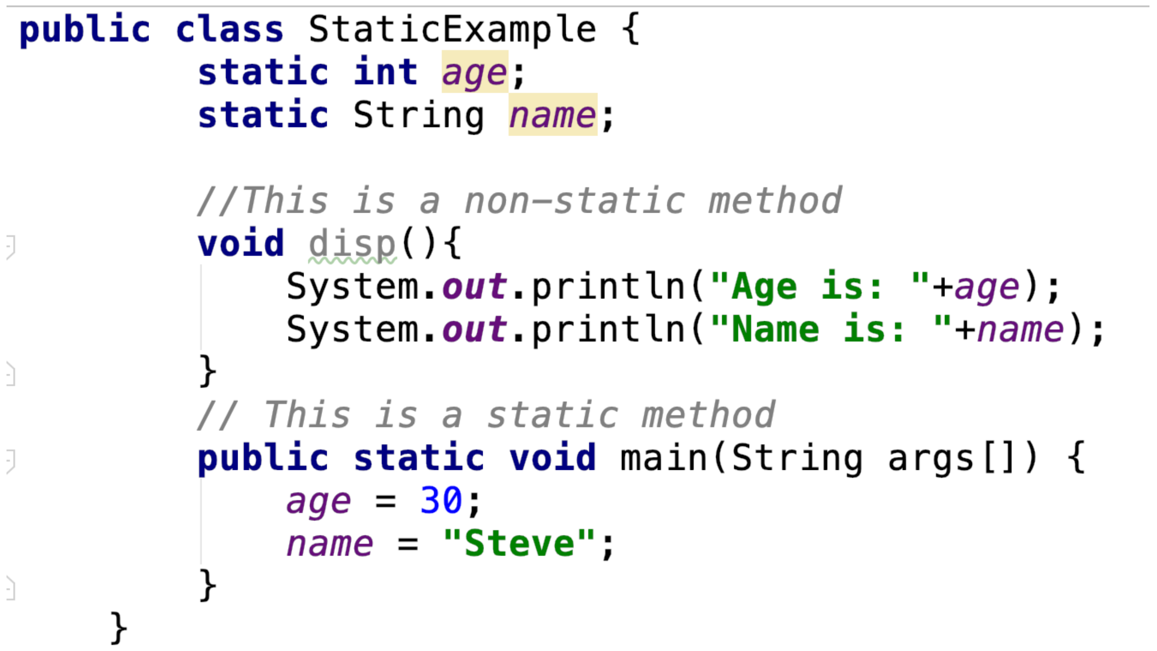
```
Obj1: count is=2
Obj2: count is=2
```

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

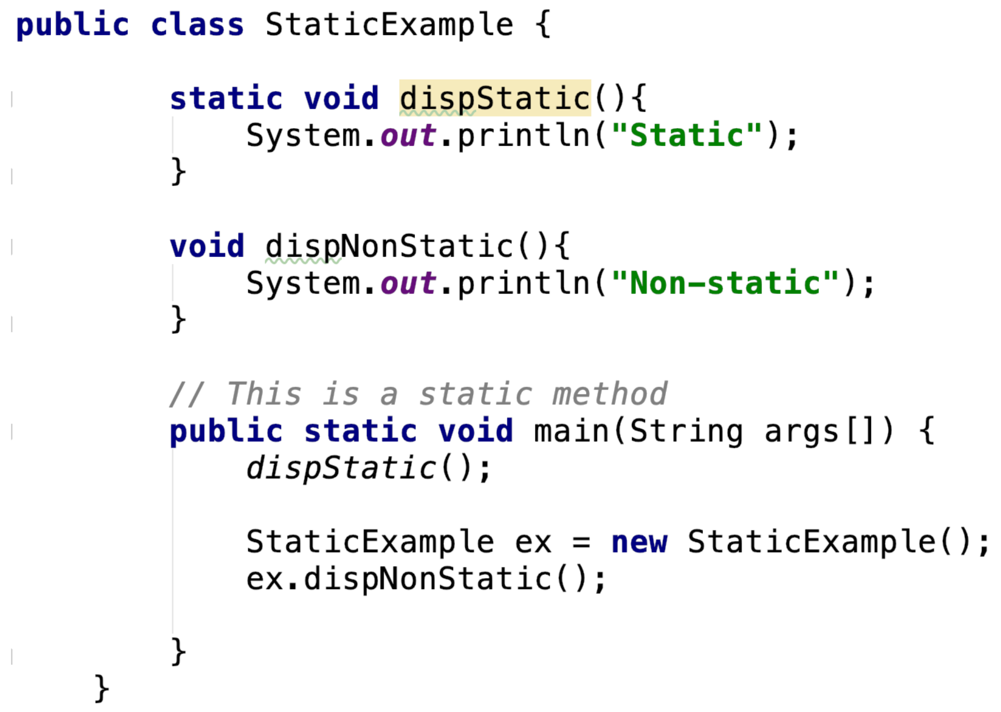
- Static Variable can be accessed directly in a static method and a non-static one



```
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();

        }
    }
```

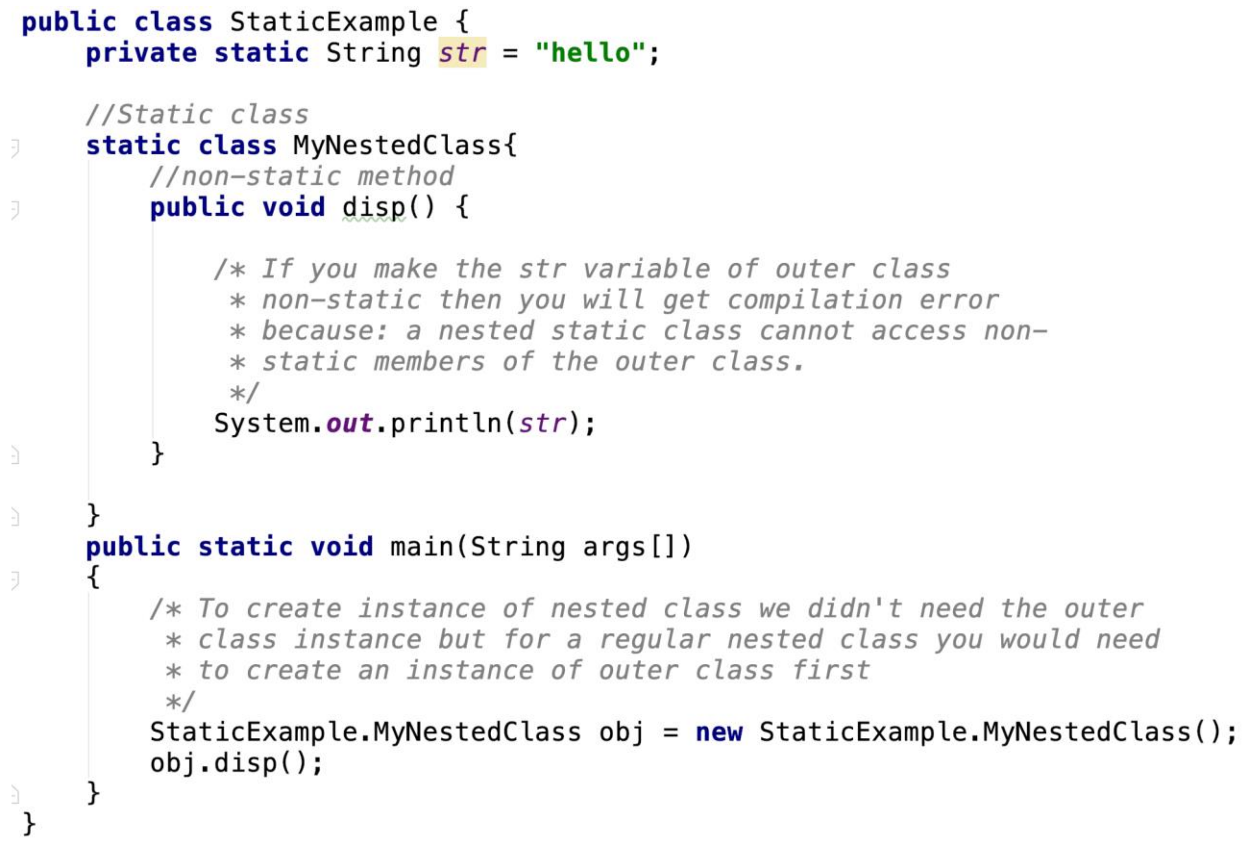
You can call them from another class as follows:
StaticExample.*dispStatic*();

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

Inner class



```
public class StaticExample {
    private static String str = "hello";

    //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();
    }
}
```

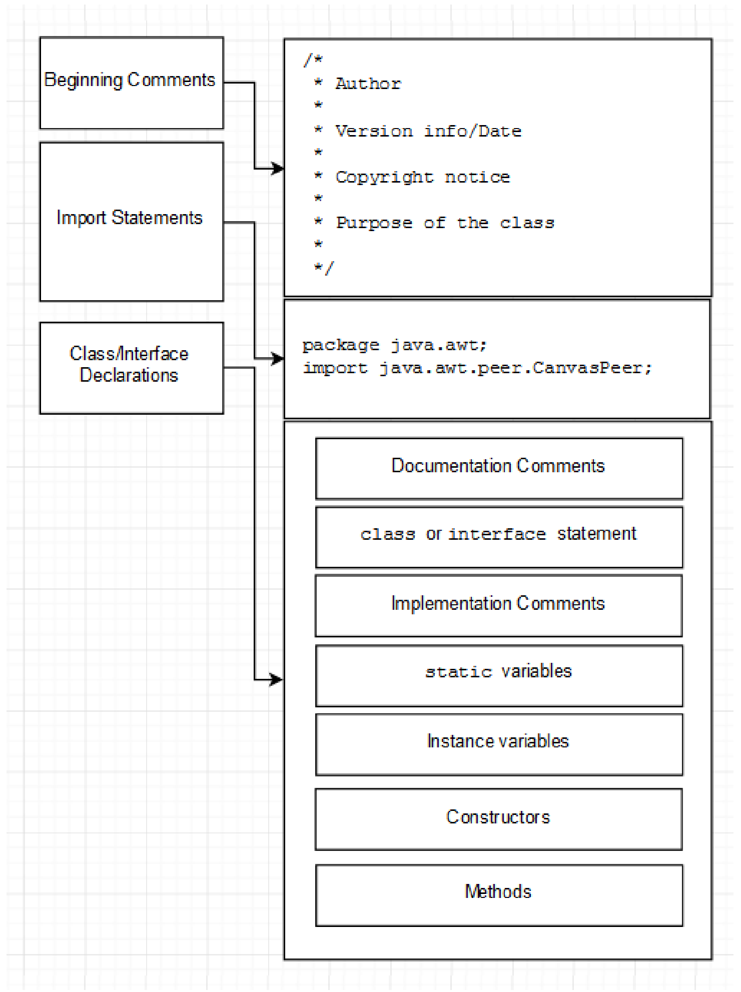
Static Quiz !

- 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.

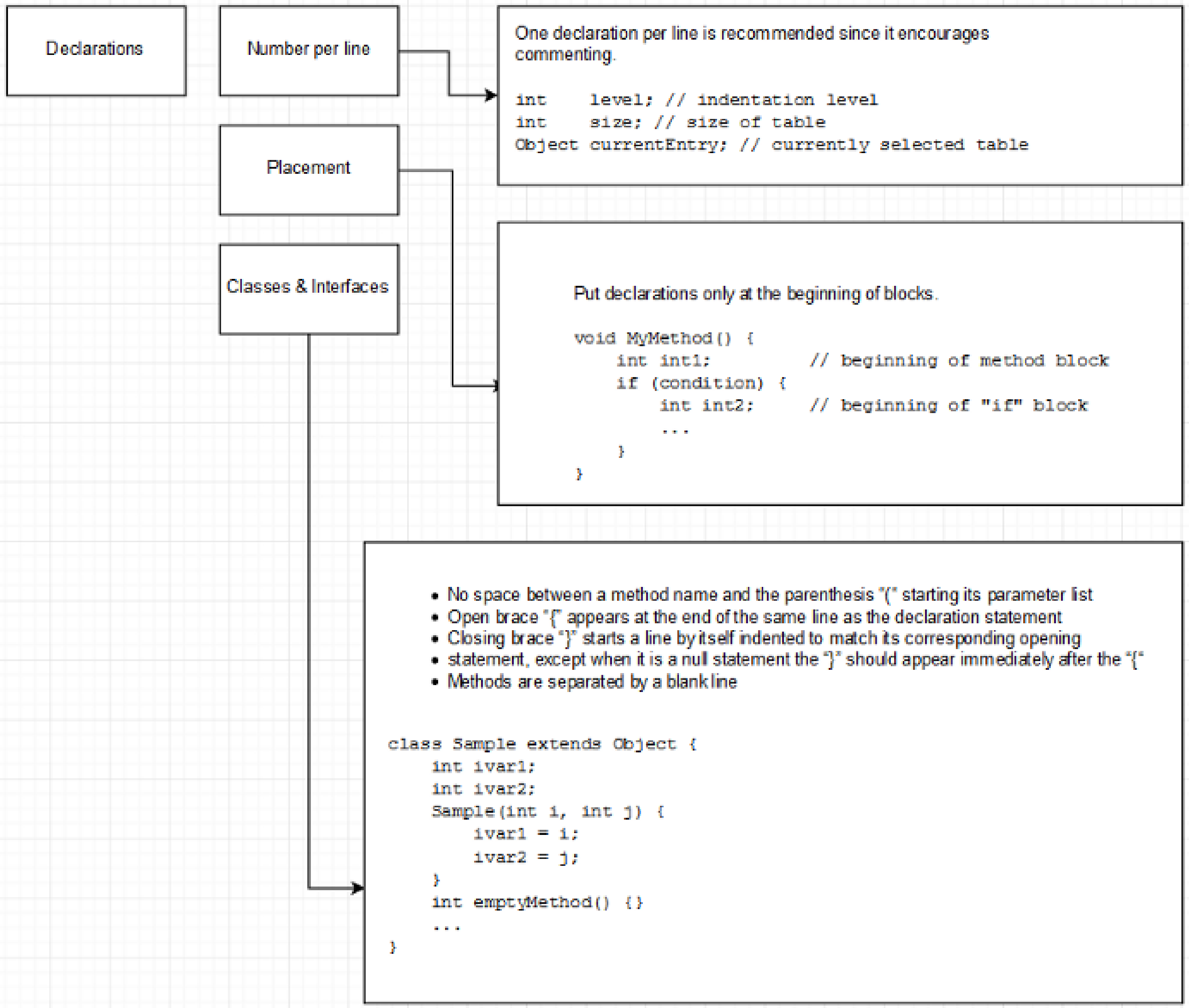
Important things

- Java coding conventions
  - https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
  - https://google.github.io/styleguide/javaguide.html
- Documentation
  - JavaDoc /** … */
  - https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
- Indentation
- Code organization

The standard source file structure



Source: https://medium.com/@thusharaj/java-code-convention-simplified-f476bd8aa719



One declaration per line is recommended since it encourages commenting.

```
int    level; // indentation level
int    size; // size of table
Object currentEntry; // currently selected table
```

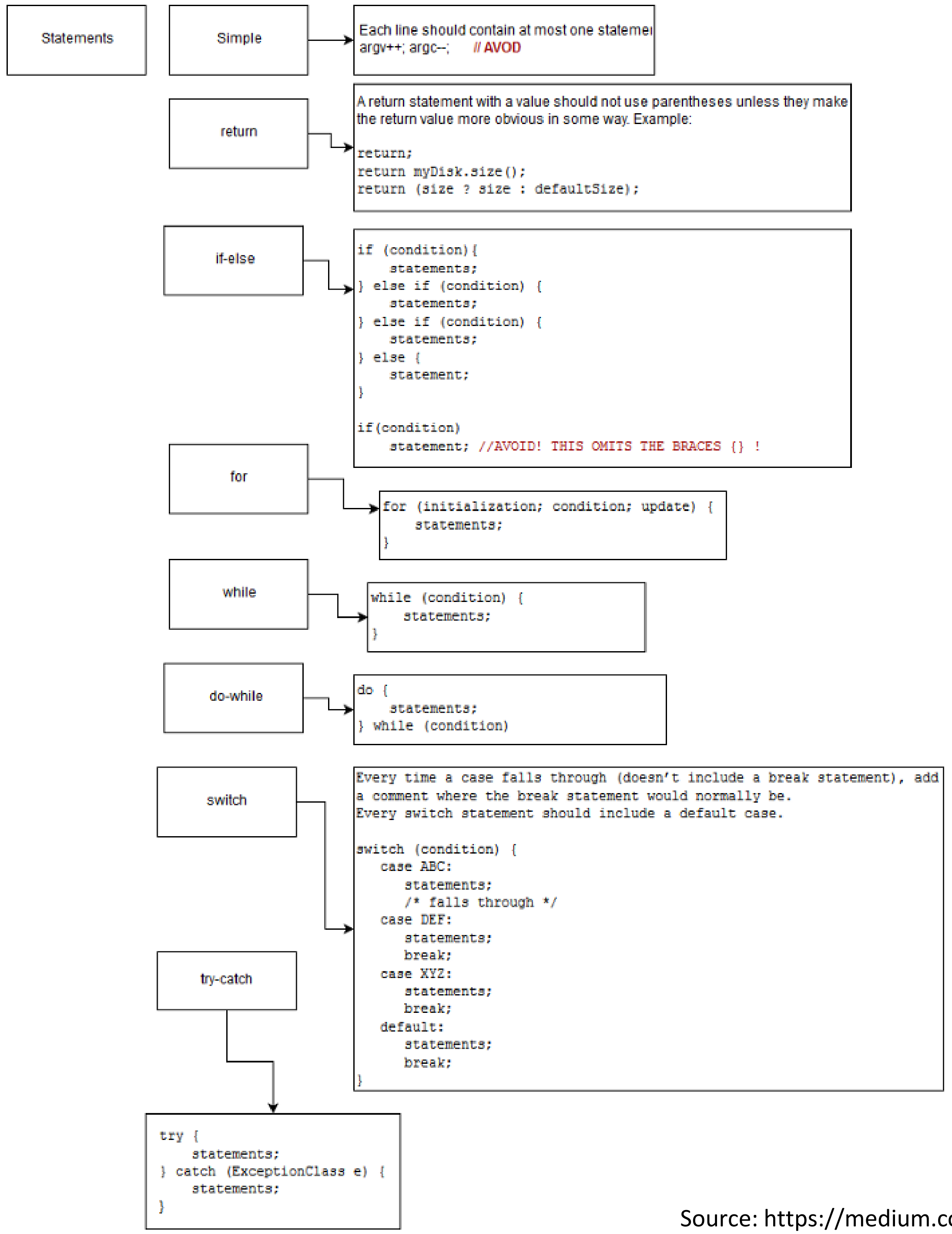
Put declarations only at the beginning of blocks.

```
void MyMethod() {
    int int1;          // beginning of method block
    if (condition) {
        int int2;      // beginning of "if" block
        ...
    }
}
```

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening
- statement, except when it is a null statement the "}" should appear immediately after the "{"
- Methods are separated by a blank line

```
class Sample extends Object {
    int ivar1;
    int ivar2;
    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }
    int emptyMethod() {}
    ...
}
```

Source: https://medium.com/@thusharaj/java-code-convention-simplified-f476bd8aa719



Each line should contain at most one statemei
argv++; argc–;   // AVOD

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

```
if (condition){
    statements;
} else if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statement;
}

if(condition)
    statement; //AVOID! THIS OMITS THE BRACES {} !
```

```
for (initialization; condition; update) {
    statements;
}
```

```
while (condition) {
    statements;
}
```

```
do {
    statements;
} while (condition)
```

```
Every time a case falls through (doesn't include a break statement), add
a comment where the break statement would normally be.
Every switch statement should include a default case.

switch (condition) {
   case ABC:
      statements;
      /* falls through */
   case DEF:
      statements;
      break;
   case XYZ:
      statements;
      break;
   default:
      statements;
      break;
}
```

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

Source: https://medium.com/@thusharaj/java-code-convention-simplified-f476bd8aa719