Mahidol University
Faculty of Information
and Communication Technology

LECTURE 15

Final Review

ITCS123 Object Oriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

(Some materials in the lecture are done by Aj. Suppawong Tuarob)

Ref: Java Concepts Early Objects by Cay Horstmann

Semester 2/2023 25 April 2024

Overall Topics

Before Midterm

- Class and Object
- Data Type
    - Primitive Type vs Object Reference
- Decision and Loop
- Methods
- Inheritance
- Polymorphism

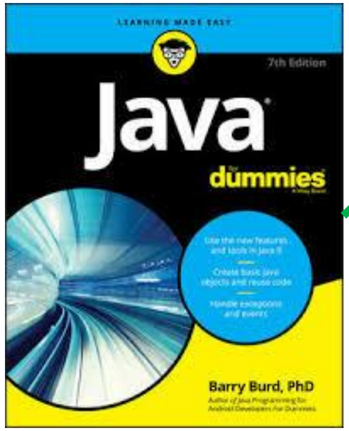
After Midterm

- Interface and Abstract
- Java Collection (Set, List, Map)
- Exception (try, catch, throw)
- File Management and RegEx
- Recursion
- Sorting

Class and Object

Multiple Associated Variables without Class

How to represent these two books?



```
public static void main (String[] args){

   String b1title = "Java Concepts";
   String b1ISBN = "1118423011";
   String b1author =  "Cay S. Horstmann";
   int b1edition = 7;
   int b1pages = 848;

   String b2title = "Java for Dummies";
   String b2ISBN = "1119235553";
   String b2author =  "Barry Burd";
   int b2edition = 7;
   int b2pages = 504;

   …

}
```

What will happen if I want to have **1000 books'** information in the library ?

Data items of Objects

- Each book has the following of data associated with them such as
  - Title
  - ISBN
  - Author(s)
  - Publisher
  - Edition
  - Number of pages
  - Etc.

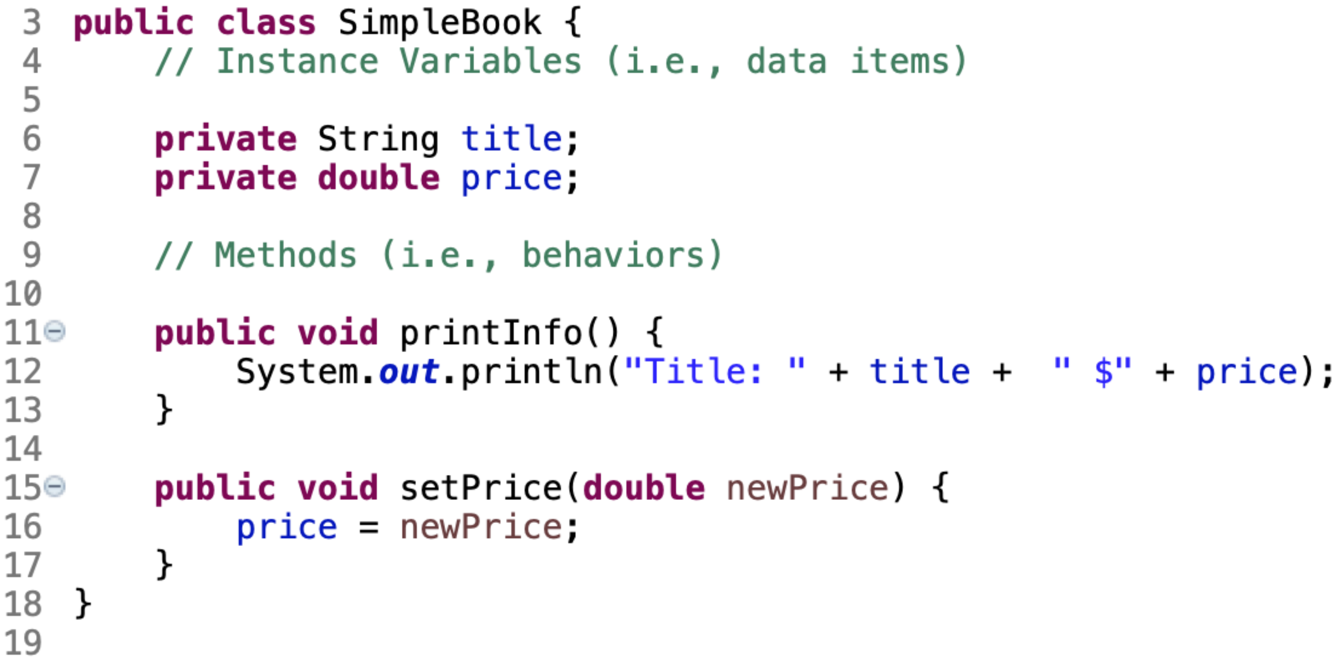
- Most useful programs do not just manipulate numbers and strings. They deal with ***data items*** or ***properties*** that more closely represent real-word **objects**.

Behaviors of Objects

- To access or change values of data items of objects, OOP programs usually use "***methods***" of objects.
- These methods are resemble to the **behaviors** of real-world objects such as
  - Display book's information
  - Return number of authors
  - Update book's price when you have a discount
  - Etc.
- Each method consists of a **sequence of statements** that can access the internal data items of an object.
- Each method must clearly define: method's name, arguments (take any input?), return (produce an output?), what they do.

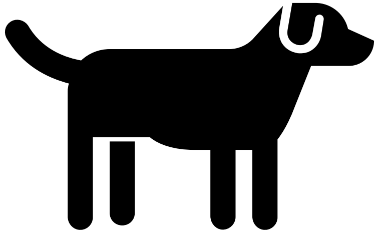
Instance Variables and Methods

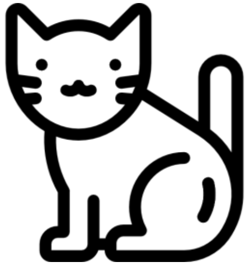
- An object consists of **data items** and **behaviors**.
  - **Data items** -> Instance Variables = Instance Fields = Attributes
  - **Behaviors** -> Methods = Function (in C programming)

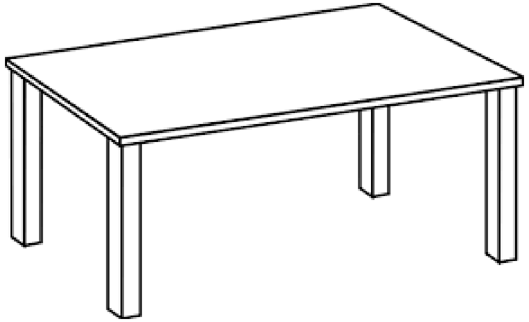


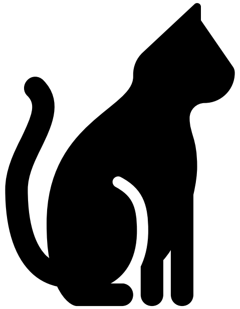
```java
public class SimpleBook {
    // Instance Variables (i.e., data items)

    private String title;
    private double price;

    // Methods (i.e., behaviors)

    public void printInfo() {
        System.out.println("Title: " + title +  " $" + price);
    }

    public void setPrice(double newPrice) {
        price = newPrice;
    }
}

```

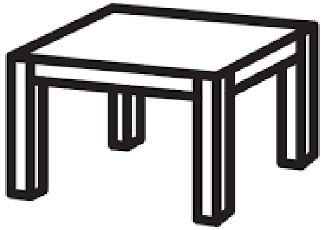
1.4 Classes: Modeling world with classes

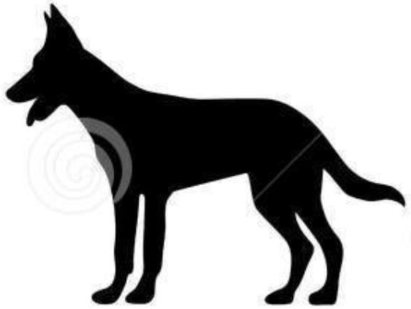




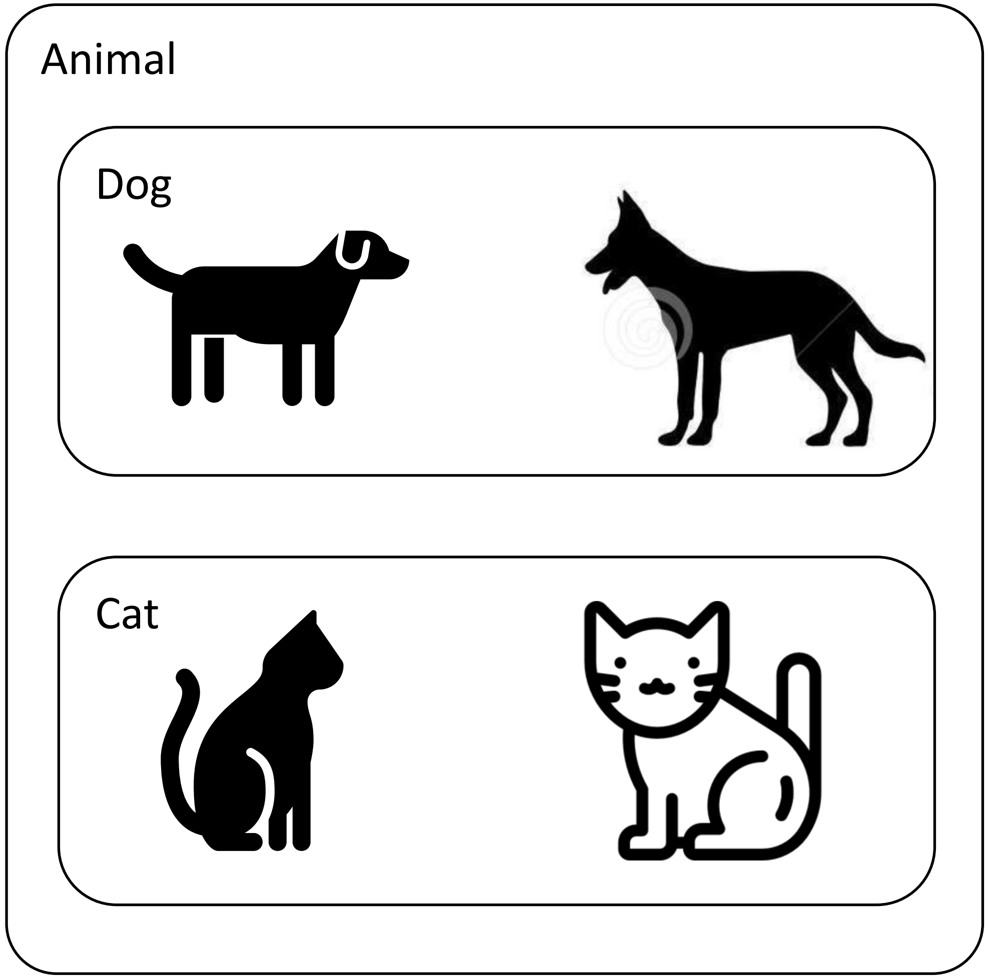








Example

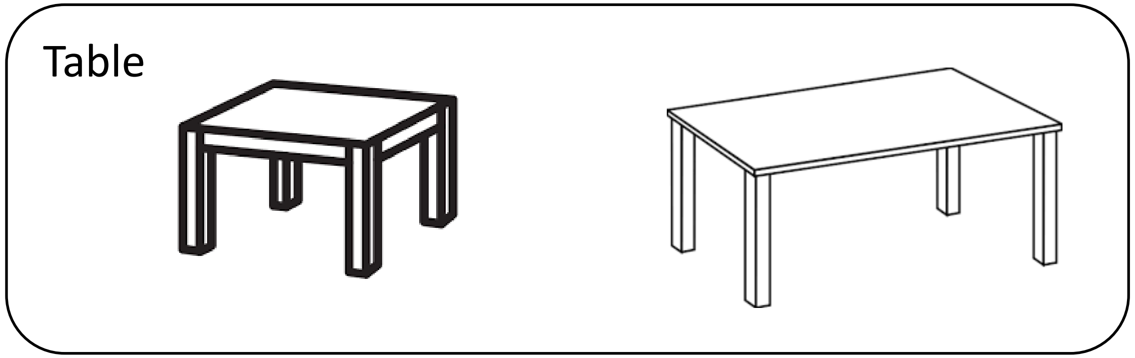


Animal

Dog

Cat

These classes are abstrations of the reality.



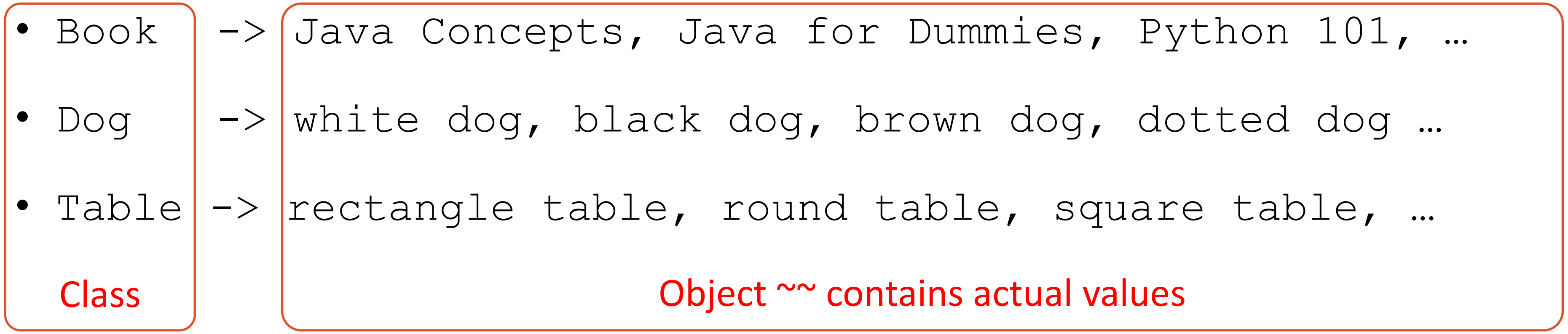
Table

We can have many other classes, depending on how broad we want the class to be.

We can come up with a class that fit all things together for example, “four-legged-things” class.

Classes

- Class is a **conceptual model** or an **abstraction** of reality.
- Class describes the commonalities of similar objects



- Book -> Java Concepts, Java for Dummies, Python 101, …
- Dog -> white dog, black dog, brown dog, dotted dog …
- Table -> rectangle table, round table, square table, …

Class

Object ~~ contains actual values

- Object is an **instance of** a class. In another word, a class is a blueprint of an object.
- Objects of the same class share the same kind of **properties** and **behavior**.

Using Objects

- In OOP paradigm, you will have to put **objects** together as its building blocks.
- Some objects are premade* and ready to use.
- But sometimes, you may need to design your own objects -> The design or blueprint of your objects is called a “class”
- To use an object, we need to **construct** an object from its class and **call the methods** defined by its class.

*Premade: prepared or made beforehand

Data Type

Data Type

- In Java, every variable either
  - a reference to an **object** (class-types)

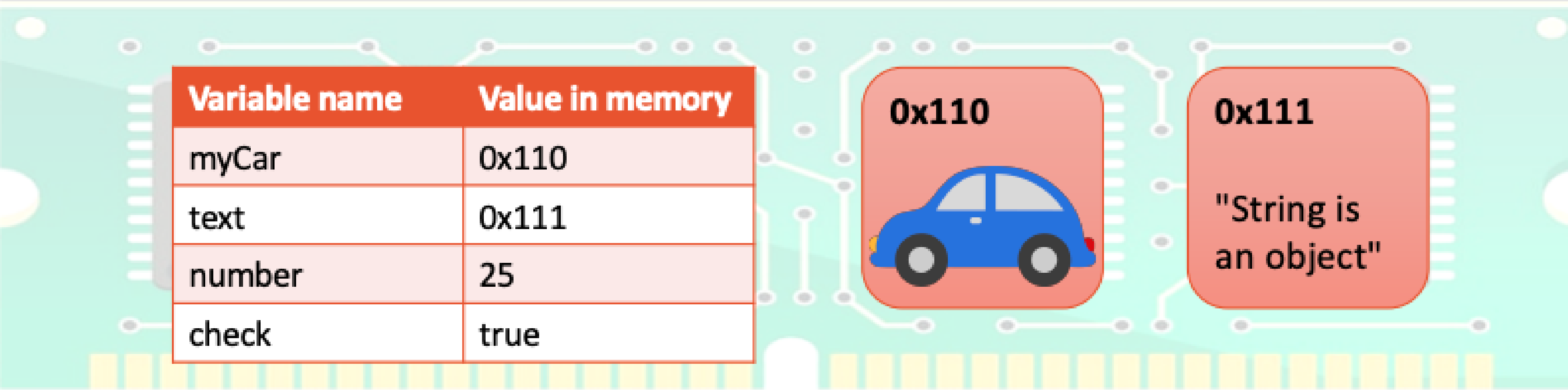
```
e.g. Car myCar = new Car()
     String text = "String is an object";
```

  - belongs to one of **8 primitive types**

```
e.g. int number = 25;
     boolean check = true;
```

How to notice:
1. Type starts with a capital letter:
   Car
2. Normal font style in Eclipse

How to notice:
1. Type starts with a lowercase letter:
   *i*nt, *b*oolean
2. **Purple-bold** font style in Eclipse



Variable name	Value in memory
myCar	0x110
text	0x111
number	25
check	true

0x110

0x111

"String is an object"

= VS ==

- Be careful and Don’t confuse between = and ==
  - The == operator tests for equality

```
x == 0        // return true if x is 0,
              // otherwise false
```

    - usually use in if-else condition and loop
    - e.g., `if(x == 0)`

```
        System.out.println("x is 0");
```

  - The = operator assigns a value to a variable

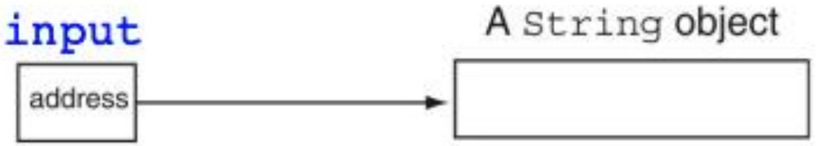
```
x = 0;        // assign 0 to x
```

Comparing STRINGS

- Do NOT use == for Strings!

Check the address!

```
String input = new String("Y");
if (input == "Y"){…}          // Always FALSE!!!
```



- Use equals method:

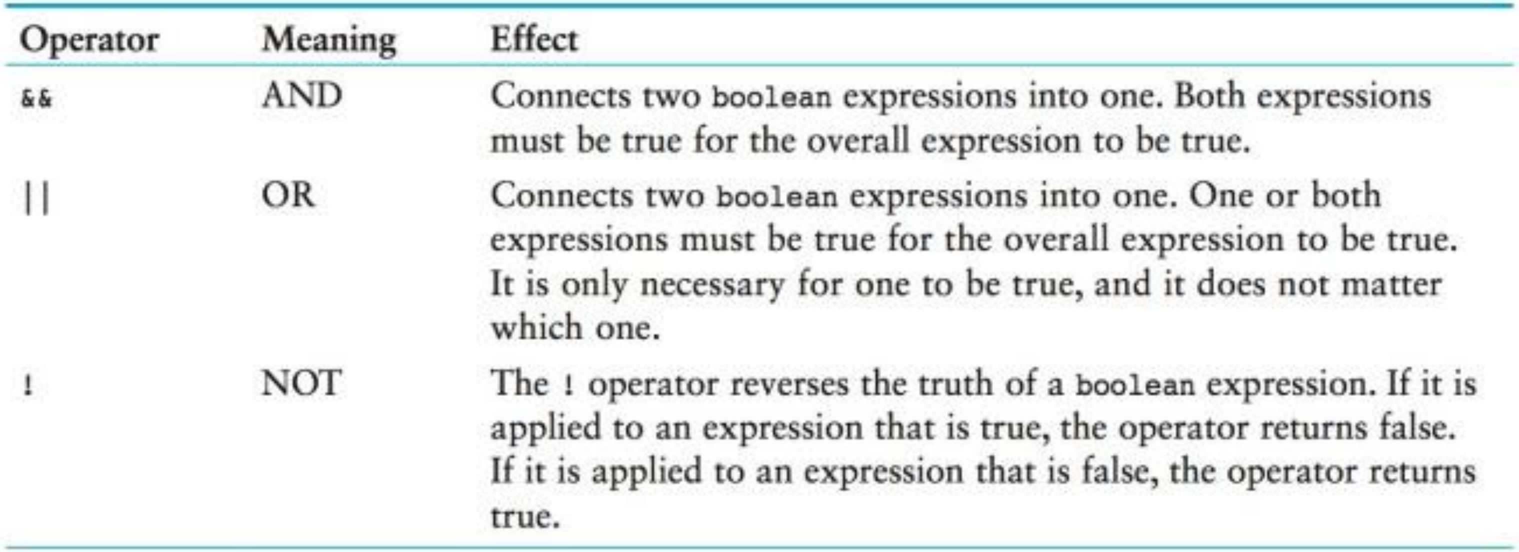
```
if (input.equals("Y")){…}  // true
```

Check the content!

- Note! Case sensitive test ("Y" != "y"), to ignore the letter case use equalsIgnoreCase method

```
if (input.equalsIgnoreCase("y")){…}      // true
```

Boolean Expression: Logical Ops



Operator	Meaning	Effect
`&&`	AND	Connects two `boolean` expressions into one. Both expressions must be true for the overall expression to be true.
`||`	OR	Connects two `boolean` expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.
`!`	NOT	The `!` operator reverses the truth of a `boolean` expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

A	B	A || B	A && B	!A
0	0	0	0	1
1	0	1	0	0
0	1	1	0	1
1	1	1	1	0

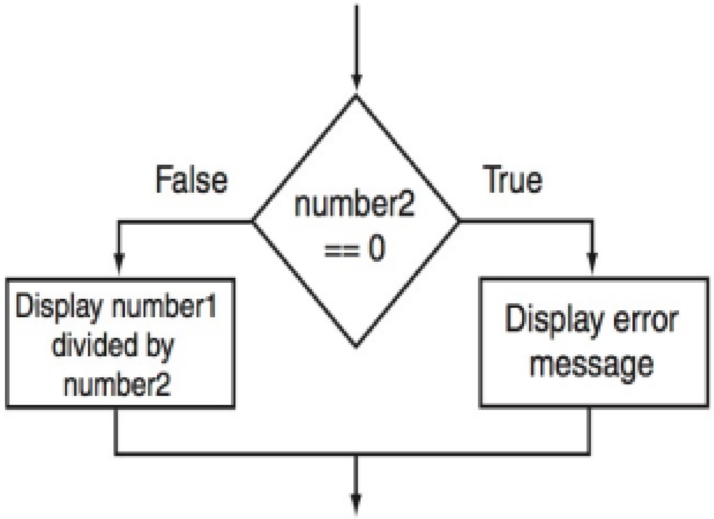
```
if( score > 70 && score <= 80)
    grade = 'B';

if( x == 10 || y == 20)

if( !(x > y) )
```

Decision and Loop

if-else statement

- The `if-else` statement is an expansion of `if` statement
- It will execute one group of statements if its boolean expression is `true`, or another group if its boolean expression is `false`.

```
if (BooleanExpression)
    statement or block;
else
    statement or block;
```

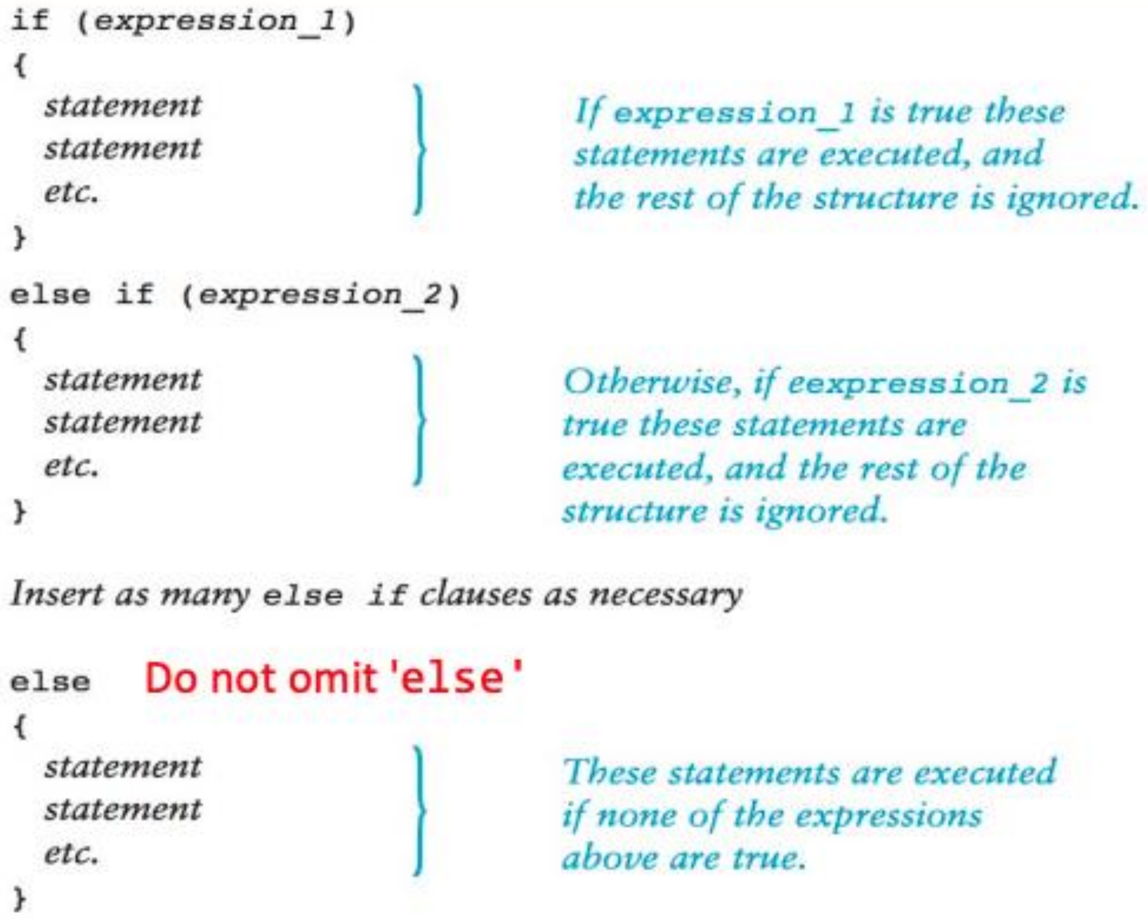


```
if (number2 == 0){
    System.out.println("Number1 cannot
            be divided by 0.");
} else{
    System.out.println(number1/number2);
}
```

if-else-if Statement

The `if-else-if` statement tests a series of conditions.

It is often simpler to test a series of conditions with the `if-else-if` statement than with a set of nested `if-else` statements.



```
if (expression_1)
{
  statement
  statement
  etc.
}
else if (expression_2)
{
  statement
  statement
  etc.
}
```

If expression_1 is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if eexpression_2 is true these statements are executed, and the rest of the structure is ignored.

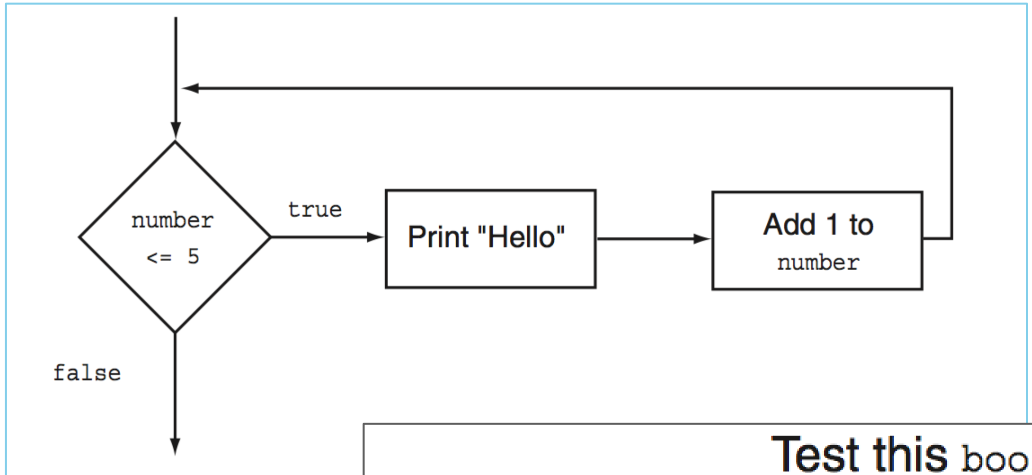
Insert as many `else if` *clauses as necessary*

```
else
{
  statement
  statement
  etc.
}
```

Do not omit 'else'

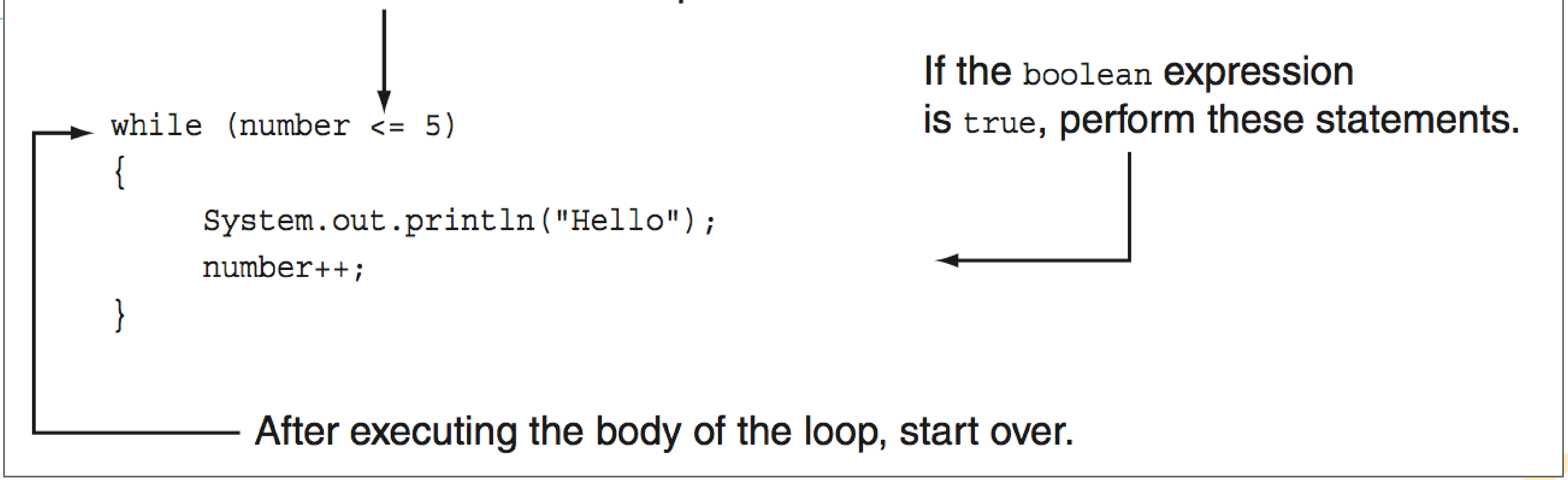
These statements are executed if none of the expressions above are true.

while Loop



```
while (BooleanExpression) {
    statement;
    statement;
    // Place as many statements
    // here as necessary.
}
```

Test this boolean expression.

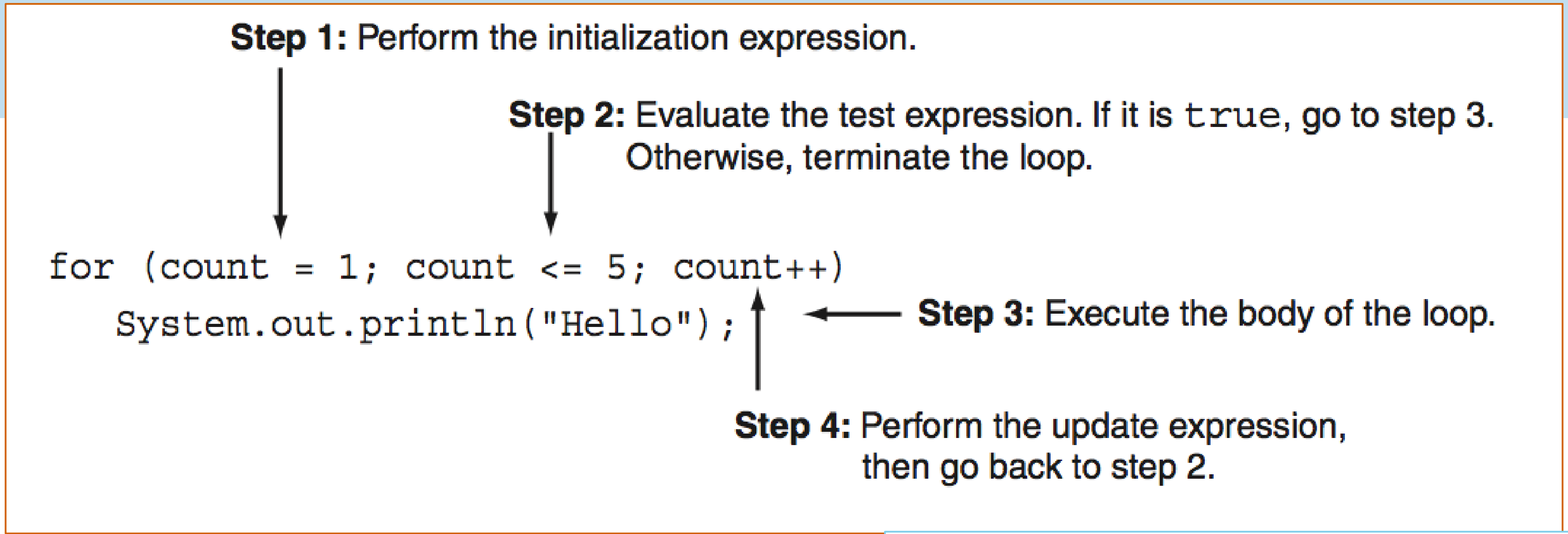


```
while (number <= 5)
{
     System.out.println("Hello");
     number++;
}
```

If the boolean expression is true, perform these statements.

After executing the body of the loop, start over.

for Loop



Step 1: Perform the initialization expression.

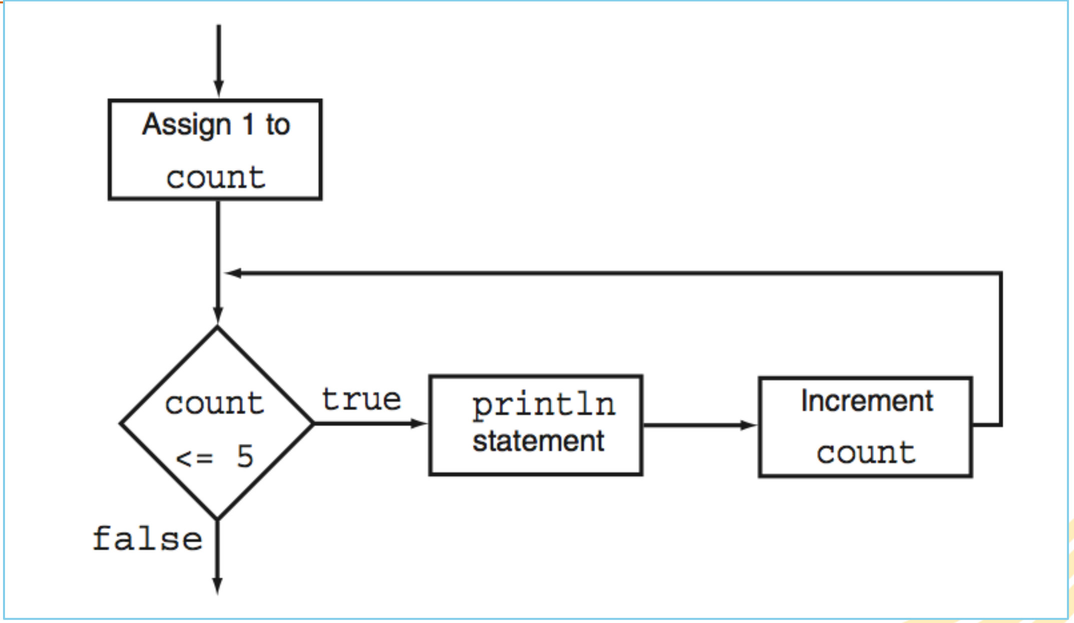
Step 2: Evaluate the test expression. If it is `true`, go to step 3. Otherwise, terminate the loop.

```
for (count = 1; count <= 5; count++)
    System.out.println("Hello");
```

Step 3: Execute the body of the loop.

Step 4: Perform the update expression, then go back to step 2.

```
for (initialization; condition; update) {
    statement;
    statement;
    // Place as many statements here
    // as necessary.
}
```



Examples

```
int number;
for (number = 1; number <= 10; number++) {
    System.out.print(number + " ");
}
```

Output: 1 2 3 4 5 6 7 8 9 10

```
for (int number = 1; number <= 10; number++) {
    System.out.print(number + " ");
}
```

Output: 1 2 3 4 5 6 7 8 9 10

```
for (int number = 2; number <= 100; number += 2) {
    System.out.println(number);
}
```

Output: Display **even numbers** from 2 through 100

```
for (int number = 2; number <= 100; number += 2) {
    System.out.println(number);
}
System.out.println(number);
```

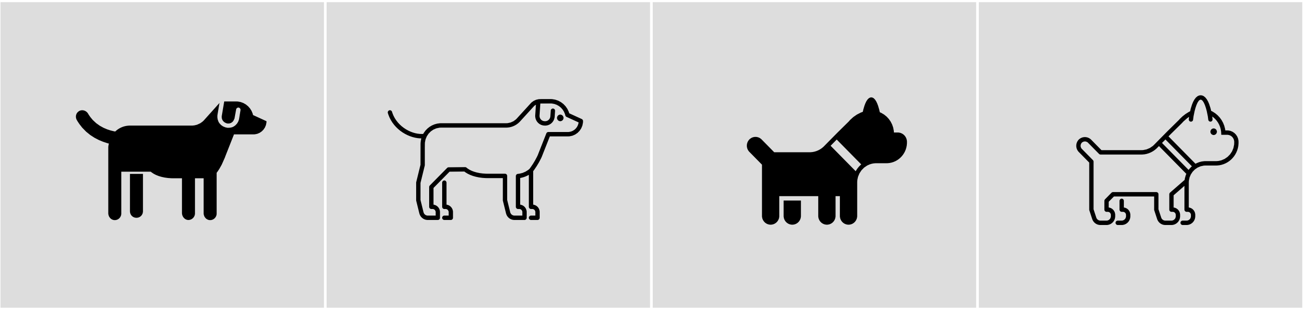
Output: Compilation Fails

Array

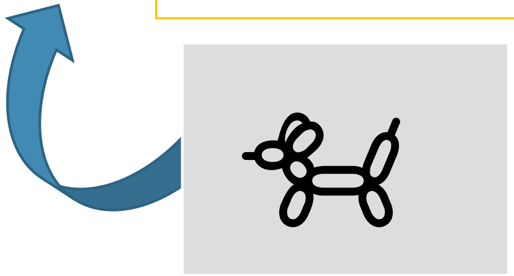
50

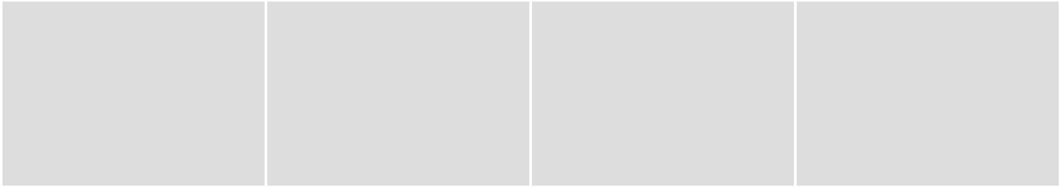
10	20	30	40

```
int[] nums = new int[4];
-- or --
int[] nums = {10, 20, 30, 40};

int x = nums[2]; // x = 30

nums[0] = 5;
```

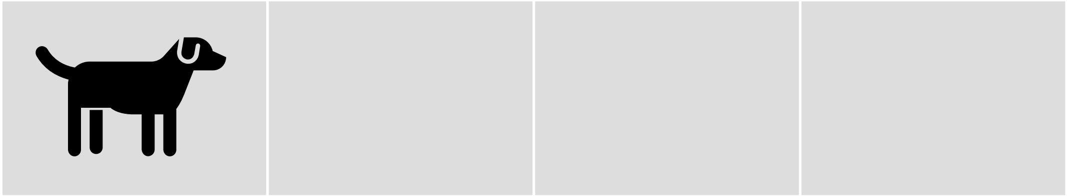


```
Dog[] dogs = new Dog[4];

dogs[0] = new Dog(12, "black");

System.out.println(dogs[0]);
```

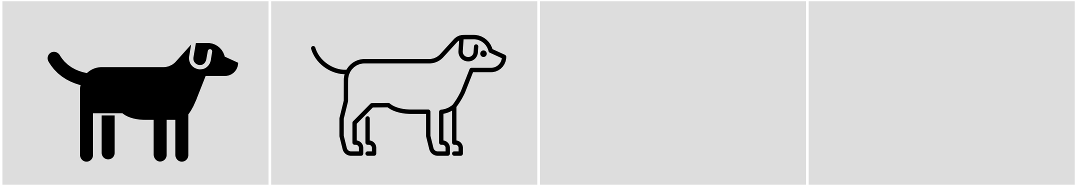




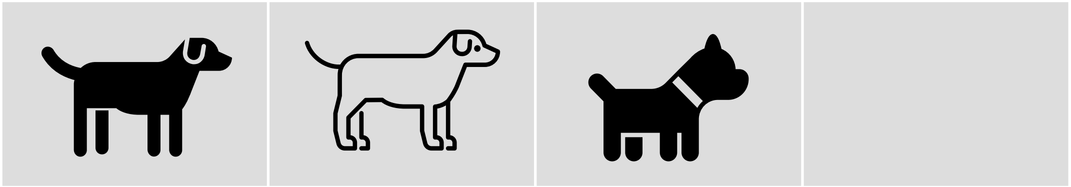
```
Dog[] dogs = new Dog[4];
```



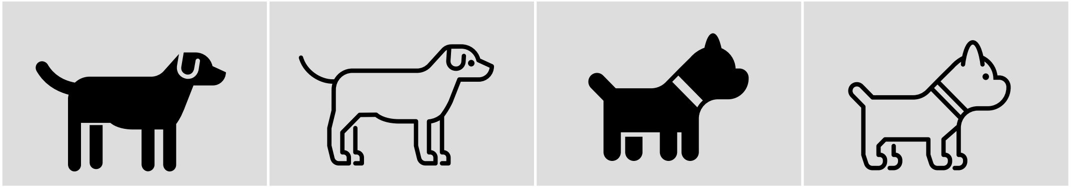
```
dogs[0] = new Dog(12, "black");
```



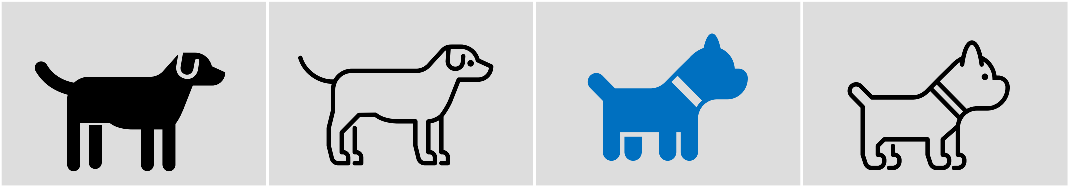
```
dogs[1] = new Dog(12, "white");
```



```
dogs[2] = new Dog(3, "black");
```

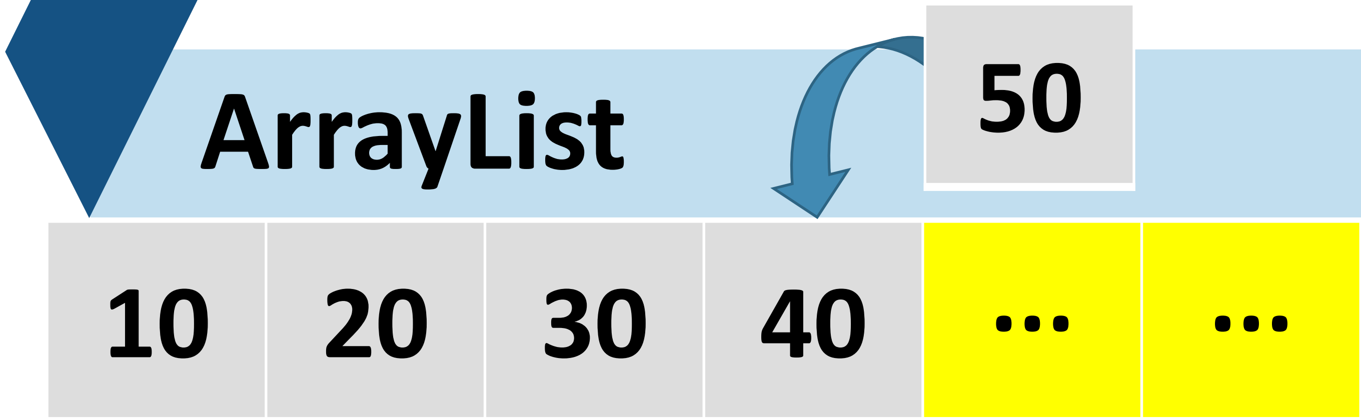


```
dogs[3] = new Dog(3, "white");
```



```
// change color of a dog
dogs[2].color = "blue";

dogs[2].setColor("blue");
```

ArrayList

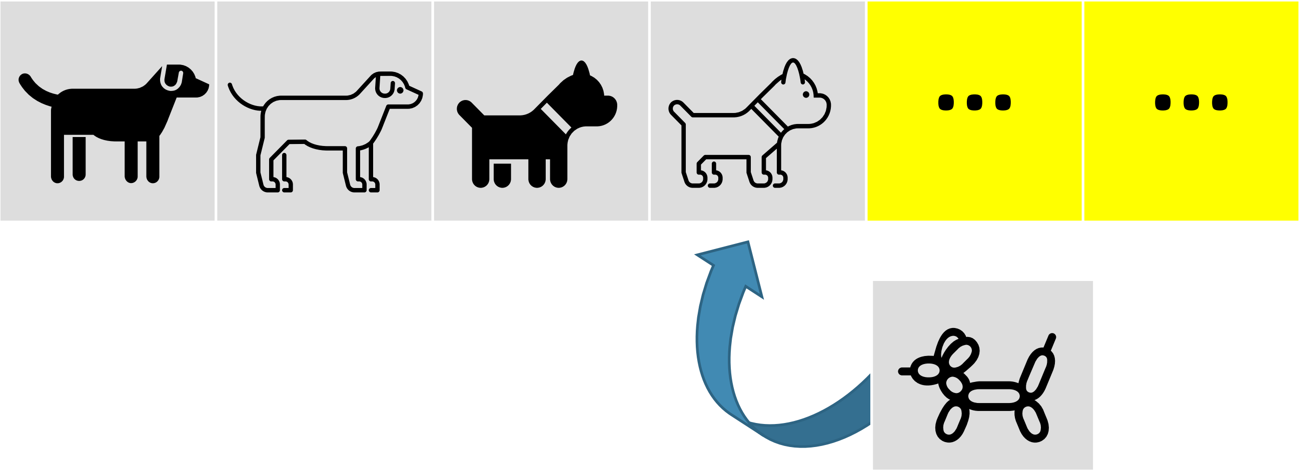


```
nums.add(10);
nums.add(20);
nums.add(30);
nums.add(40);
```

```
ArrayList<Integer> nums = new ArrayList<Integer>();
```

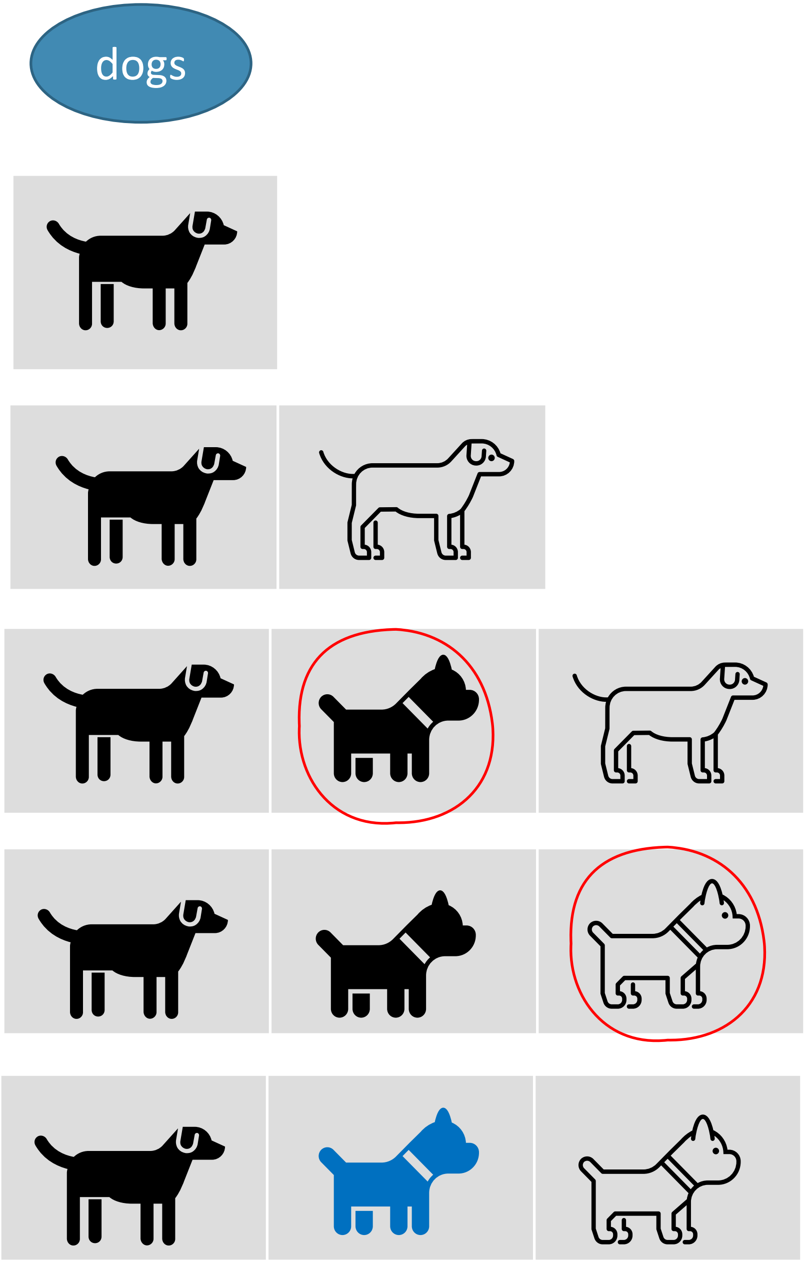
```
int x = nums.get(2); // x = 30
```

```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```

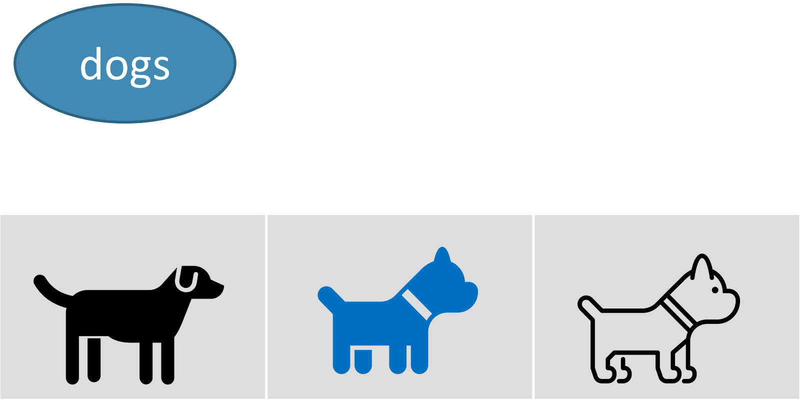


```
dogs.add(new Dog(12, "black"));
dogs.add(new Dog(12, "white"));
dogs.add(new Dog(3, "black"));
dogs.add(new Dog(3, "white"));
```

```
Dog d = dogs.get(2); // [3, black]
```



```
ArrayList<Dog> dogs = new ArrayList<Dog>();

dogs.add(new Dog(12, "black"));

dogs.add(new Dog(12, "white"));

dogs.add(1, new Dog(3, "black"));

dogs.set(2, new Dog(3, "white"));

// change color of dog at index 1
dogs.get(1).color = "blue";
// -- or --
dogs.get(1).setColor("blue");
```



```
ArrayList<Dog> dogs = new ArrayList<Dog>();
```

```
// count dog who are younger than 4 months
int count = 0;

for(Dog d : dogs){
   if(d.getAge() < 4)
      count++;
}

System.out.println(count);  // 2
```

```
// print dog with index
for(int i = 0; i < dogs.size(); i++){
   System.out.println("index:" + i + ", " + dogs.get(i));
}
```

```
index:0, DOG[12,black]
index:1, DOG[3,blue]
index:2, DOG[3,white]
```

Methods

Methods

- no input, no return

```
public void count(){
        count++;                    // to increase the value of the counter by one
}
```

- With input, no return

```
public void setCount(int val){
        this.count = val;           // to set a new value to the counter
}
```

- No input, with return

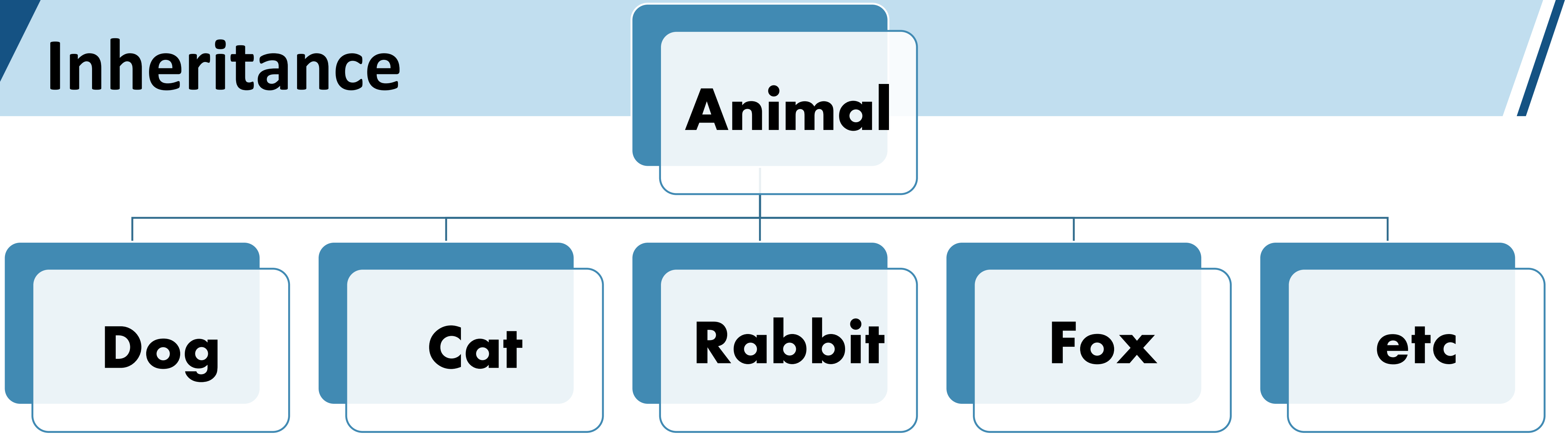
```
public int getCount(){
        return count;               // to return the value of the counter
}
```

- With input, with return

```
public int addCount(int val){
        count = count + val;        // to add value to the counter, and return current value
        return count;
}
```

Inheritance

Inheritance



Animal

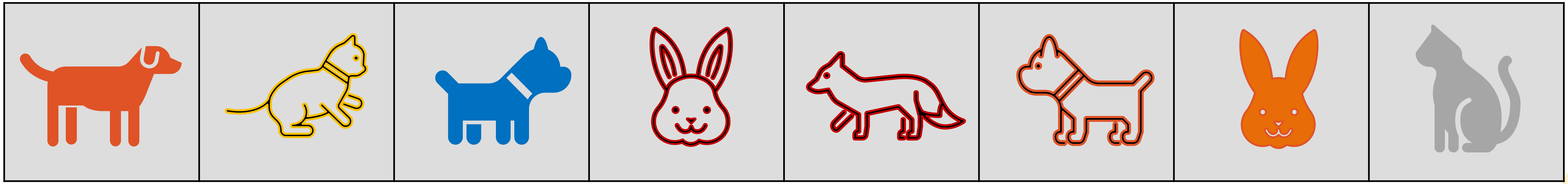
Dog

Cat

Rabbit

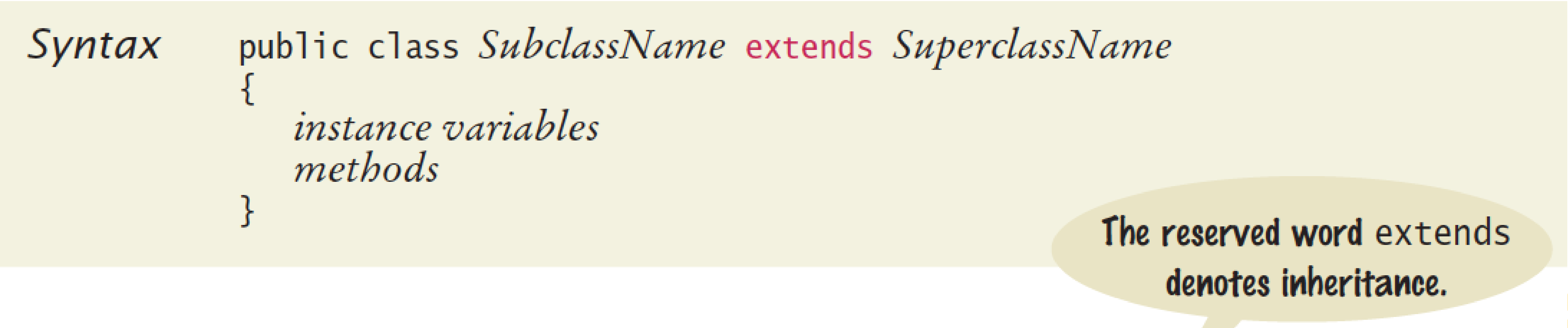
Fox

etc



Implementing Subclass

- Subclass inherit superclass by adding “**extends**” keyword.
- Subclass only include what makes the subclass **different form** its superclass
- Subclass objects automatically have the **instance variables** that are declared in the superclass. So you only declare instance variables that are not part of the superclass. (e.g., power in EnergyDrink)
- Subclass objects can call all inherited method from the superclass. You only implement any specialized method for the subclass. For example, EnergyDrink has getPower() method



Syntax

```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```

The reserved word extends denotes inheritance.

Instance Fields & Constructor Methods

```java
public class Food {
    public String name;
    private int level;

    Food(String name, int level){
        this.name = name;
        this.level = level;
    }

    void printInfo(){
        System.out.println(
                name + ": " + level);
    }

    int getLevel(){
        return level;
    }
}
```

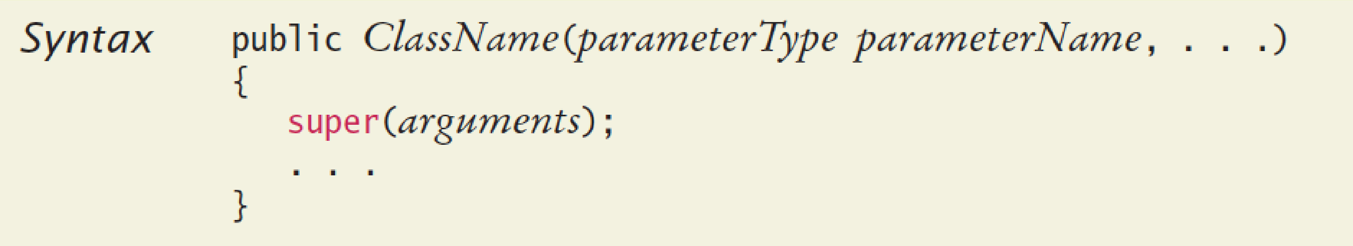
```java
//EnergyDrink is a subclass, and Food is a superclass
public class EnergyDrink extends Food{
    private int power;

    EnergyDrink(String name, int level, int power){
        // call constructor method of the superclass
        super(name, level);

        // assign a input value to a new variable (power)
        this.power = power;
    }
}
```

How to create subclass's **constructor** method.

A subclass constructor can only initialize the subclass instance variables.

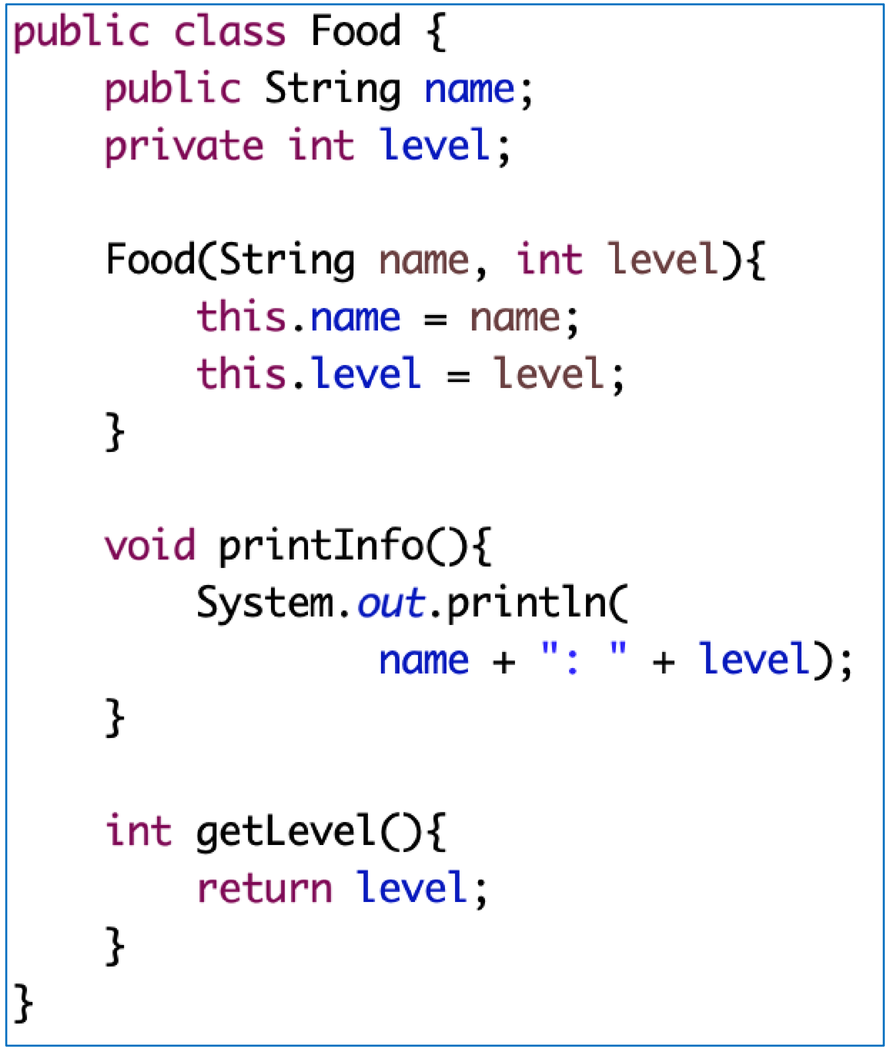
But the superclass instance variables also need to be initialized.

-> via **superclass constructor**, use the **super** reserved word in the ***first statement*** of the subclass constructor.

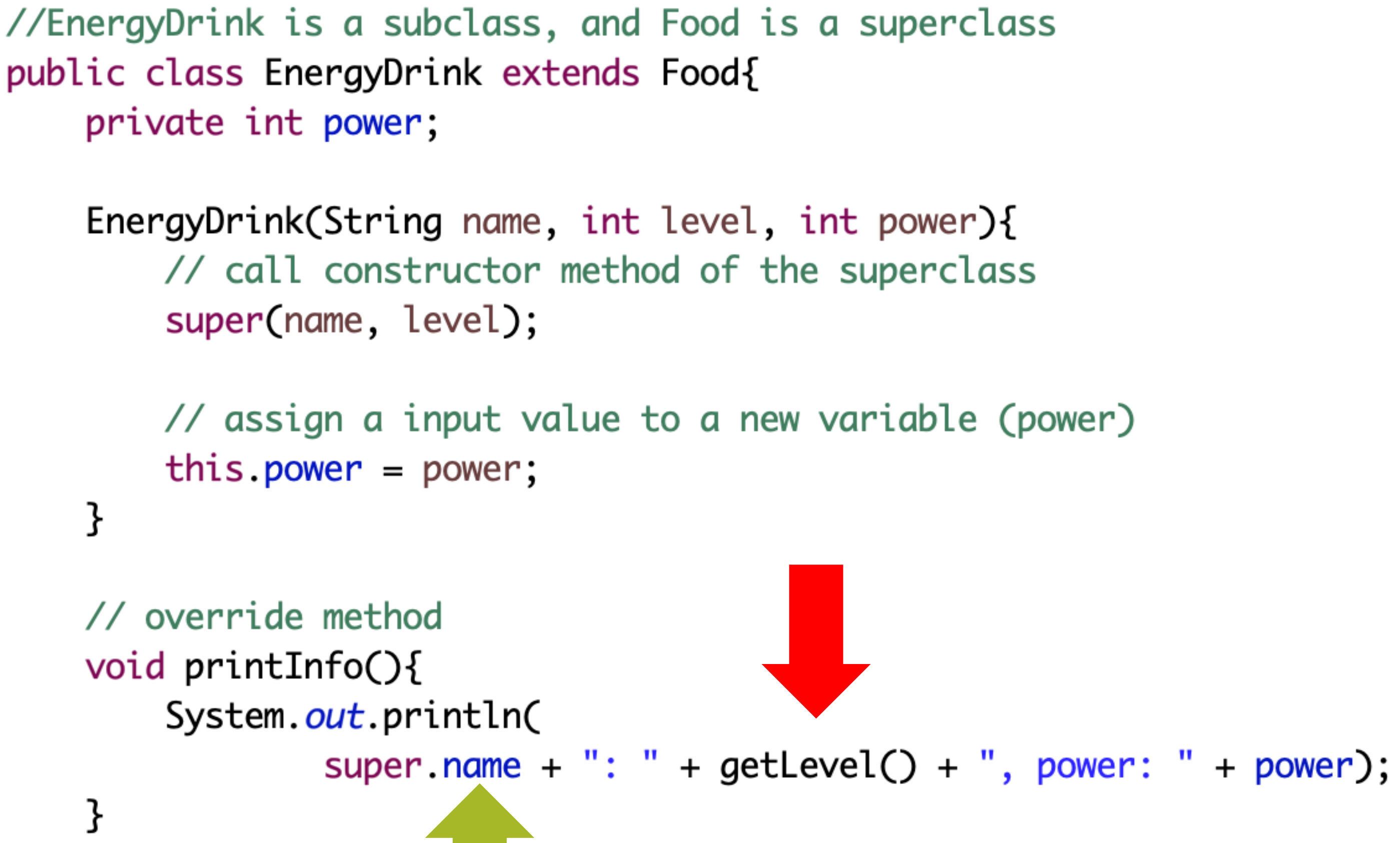


```
Syntax    public ClassName(parameterType parameterName, . . .)
          {
             super(arguments);
             . . .
          }
```

Accessing Inherited Instance Fields



```java
public class Food {
    public String name;
    private int level;

    Food(String name, int level){
        this.name = name;
        this.level = level;
    }

    void printInfo(){
        System.out.println(
                name + ": " + level);
    }

    int getLevel(){
        return level;
    }
}
```



```java
//EnergyDrink is a subclass, and Food is a superclass
public class EnergyDrink extends Food{
    private int power;

    EnergyDrink(String name, int level, int power){
        // call constructor method of the superclass
        super(name, level);

        // assign a input value to a new variable (power)
        this.power = power;
    }

    // override method
    void printInfo(){
        System.out.println(
                super.name + ": " + getLevel() + ", power: " + power);
    }
```

The instance variable "name" inherited from superclass is **public**, so the subclass can access it directly using `super.name` or just name.

But, "`level`" inherited from superclass is **private**, this subclass cannot access it directly. You have to get its value by calling `getLevel()` method.

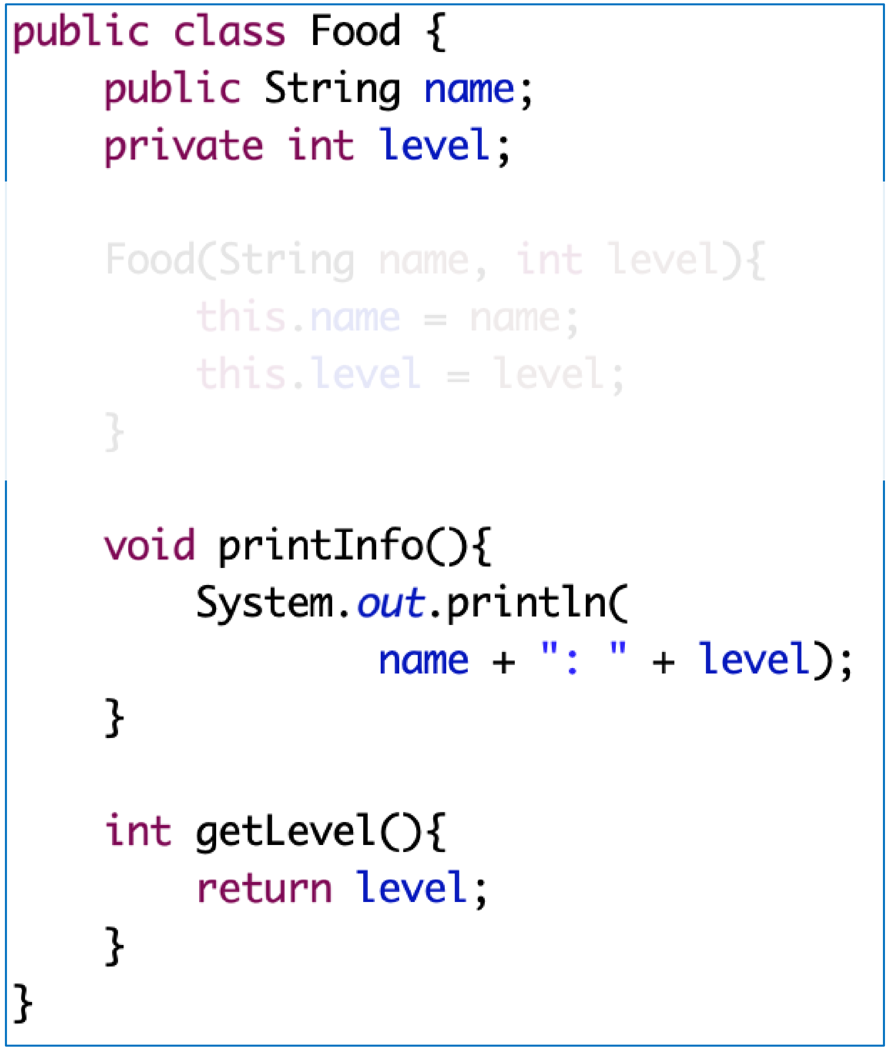
Shadowing Instance Field (DON'T do this)

```java
public class Food {
    public String name;
    private int level;

    Food(String name, int level){
        this.name = name;
        this.level = level;
    }

    void printInfo(){
        System.out.println(
                name + ": " + level);
    }

    int getLevel(){
        return level;
    }
}
```

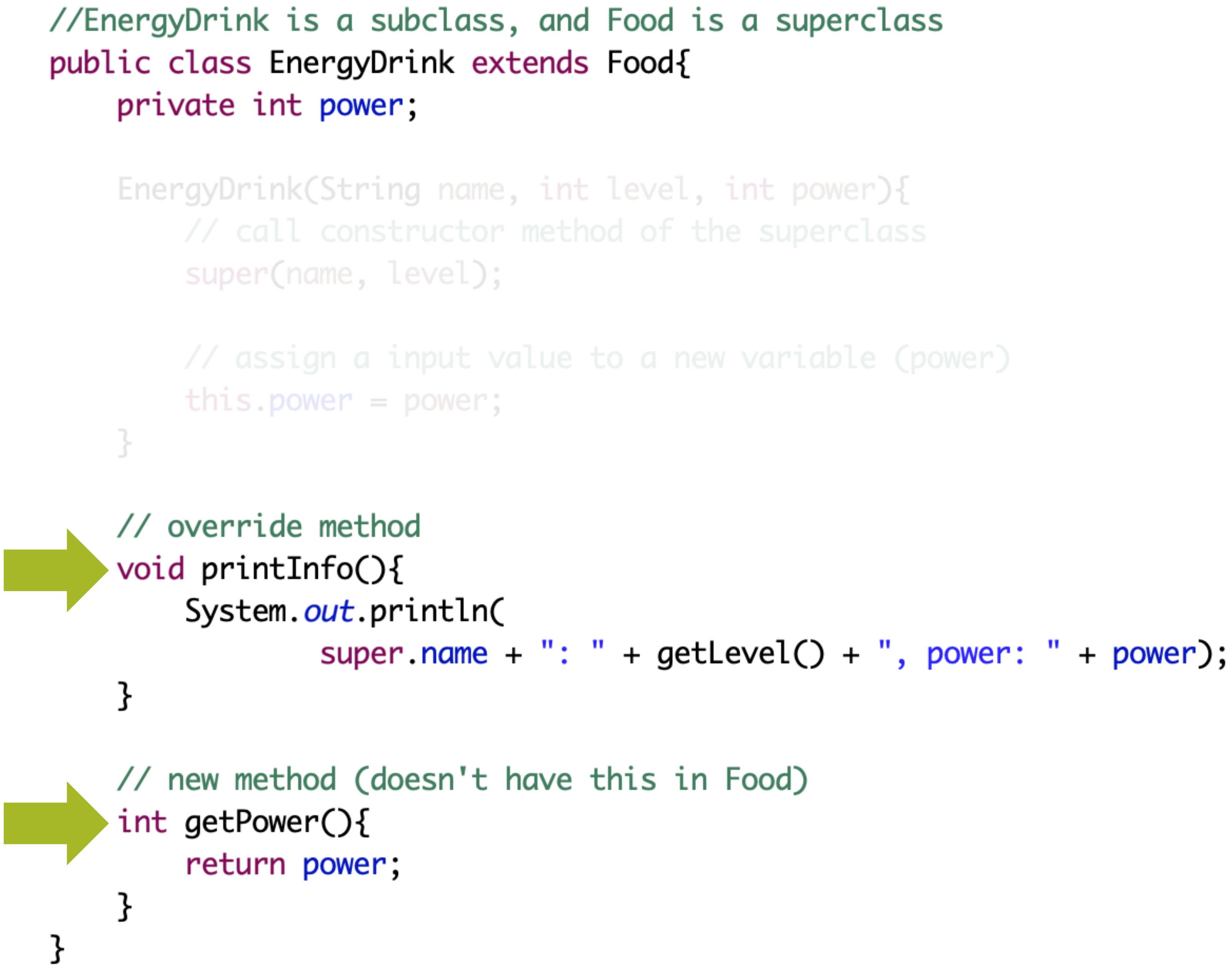
```java
//EnergyDrink is a subclass, and Food is a superclass
public class EnergyDrink extends Food{
    private int power;
    private int level;

    EnergyDrink(String name, int level      po
        // call constructor method of      pe
        super(name, level);

        // assign a input value to a new variable (     r)
        this.power = power;
    }
    // override method
    void printInfo(){
        System.out.println(
                super.name + ": " + level + ", power: " + power);
    }
```

```
name: M-100
level: 1 (inherit from Food)
power: 10
level: 1 (new variable in EnergyDrink)
```

Inherit, Override, New Methods



```java
public class Food {
    public String name;
    private int level;

    Food(String name, int level){
        this.name = name;
        this.level = level;
    }

    void printInfo(){
        System.out.println(
                name + ": " + level);
    }

    int getLevel(){
        return level;
    }
}
```

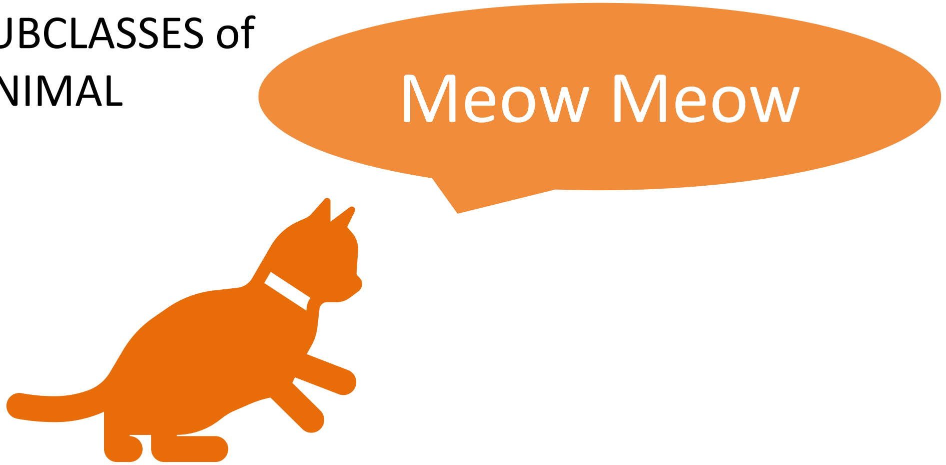


```java
//EnergyDrink is a subclass, and Food is a superclass
public class EnergyDrink extends Food{
    private int power;

    EnergyDrink(String name, int level, int power){
        // call constructor method of the superclass
        super(name, level);

        // assign a input value to a new variable (power)
        this.power = power;
    }

    // override method
    void printInfo(){
        System.out.println(
                super.name + ": " + getLevel() + ", power: " + power);
    }

    // new method (doesn't have this in Food)
    int getPower(){
        return power;
    }
}
```

Polymorphism

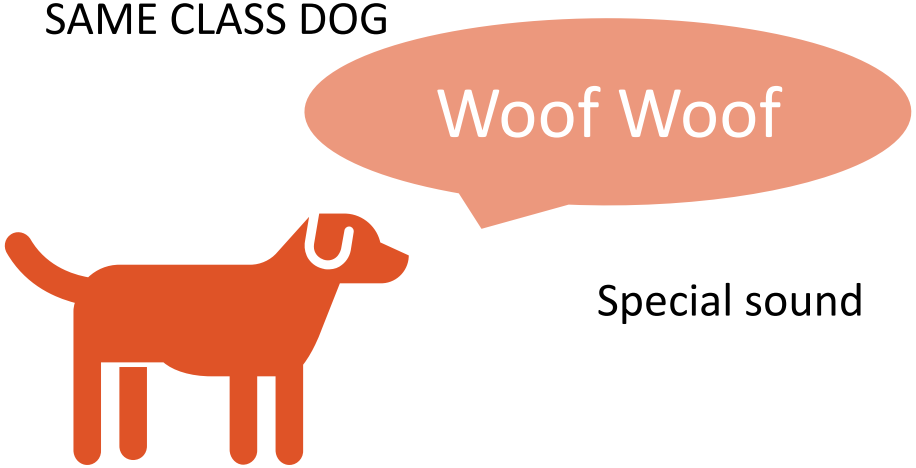
Override vs Overload

DOG and CAT are SUBCLASSES of ANIMAL



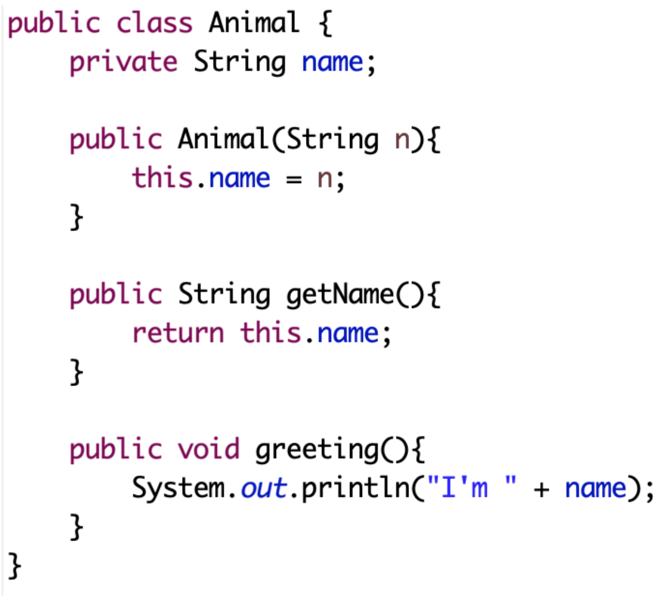
Default sound

SAME CLASS DOG

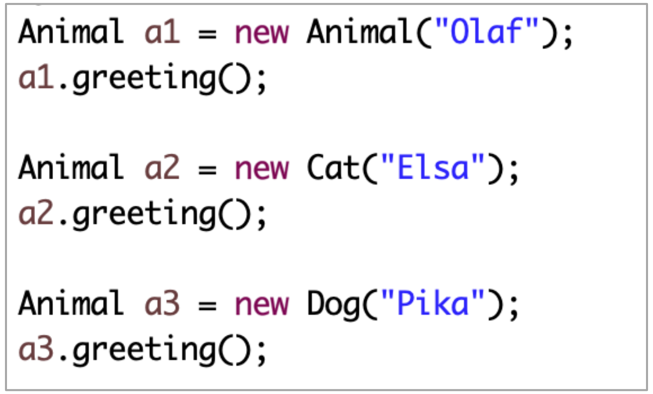


Special sound

Can we do this?



```
public class Animal {
    private String name;

    public Animal(String n){
        this.name = n;
    }

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }
}
```



```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```
public static void main(String args[]) {
    System.out.println("Hi! I love java.");
    System.out.println(100);
    System.out.println(true);
    System.out.println(99.99999 + "%");

    myPrint(5);          // call myPrint(   ?   )
    myPrint(5.0);        // call myPrint(   ?   )
}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    System.out.println("double d = " + d);
}
```

So.. What is Polymorphism?

- **‘poly’ = many,** and **‘morph” = forms**
- **Polymorphism = “having multiple forms”** allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
  - In the real world, you can use universal remote control to control all kinds of digital TV. Even though each TV, each bran may execute differently
  - In oop, it describes the concept that you can access objects of different types through the same interface. This makes program *easily extensible*. (simply say – calling the same method but doing different ways depends on the objects)

I don’t care what brand or model you are, as long as you are a digital TV, I can control you.
Evolution Kit
SMART HUB

- Java support two kinds of polymorphism -> **overload** and **override**

Polymorphism

Overloading

- Two or more methods with different signatures

Overriding

(Note that we already leared this last week!!)

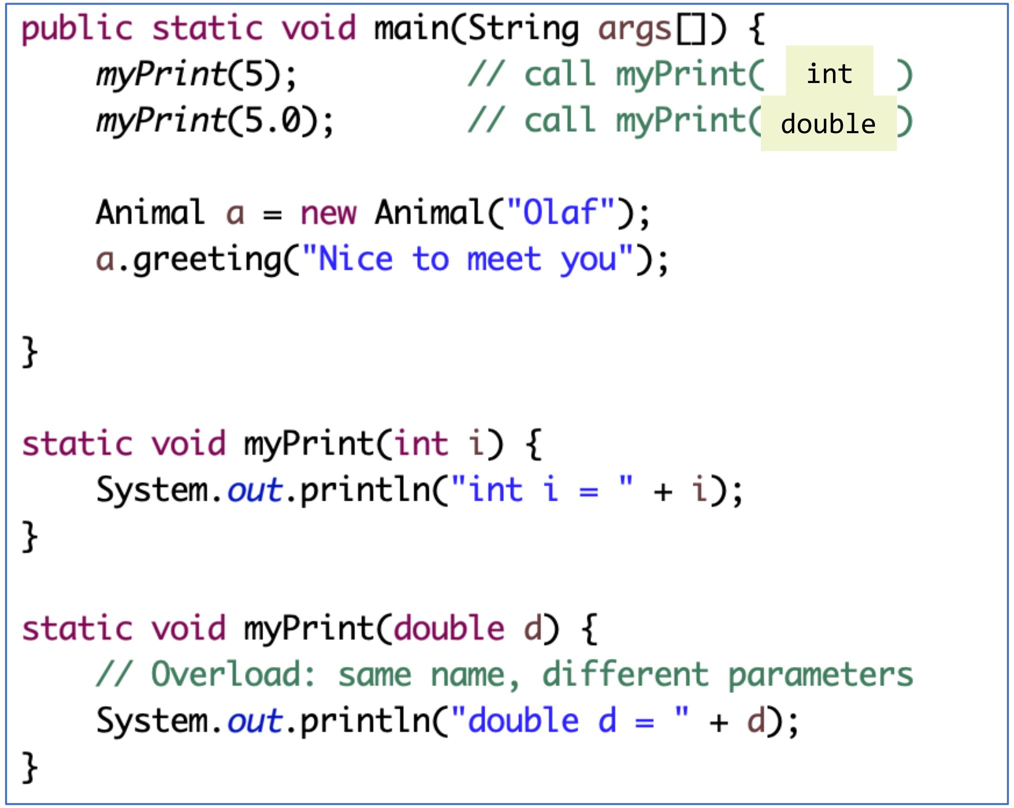
- Replacing an inherited method with another method having the same signature

Signature in Java

`foo(int i)` and `foo(int i, int j)` are different

`foo(int i)` and `foo(int k)` are the same

`foo(int i, double d)` and `foo(double d, int i)` are different



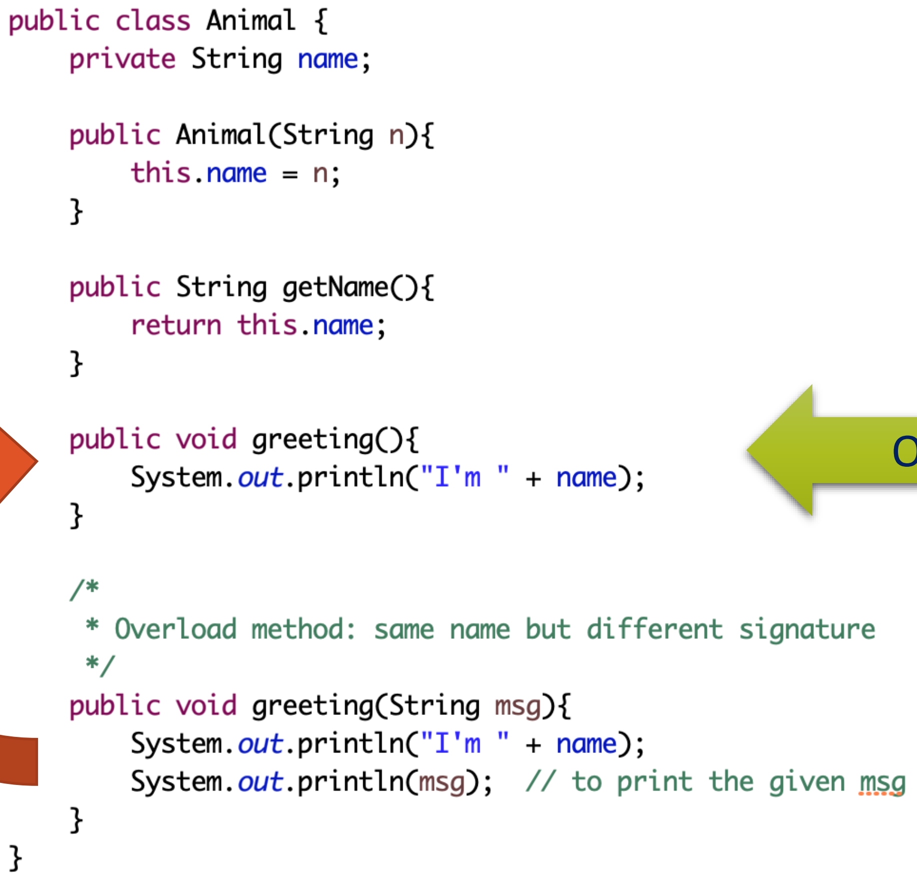
```
public static void main(String args[]) {
    myPrint(5);          // call myPrint( int )
    myPrint(5.0);        // call myPrint( double )

    Animal a = new Animal("Olaf");
    a.greeting("Nice to meet you");

}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    // Overload: same name, different parameters
    System.out.println("double d = " + d);
}
```

Overloading

Overload: in the same class, two methods can have the same name, provided they differ in their parameter types. These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated.

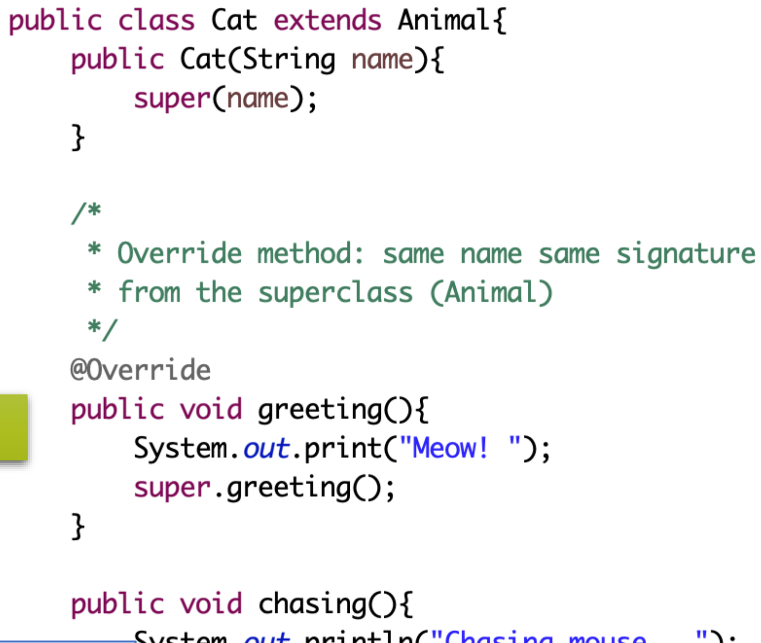
Overriding: where a subclass method provides an implementation of a method whose parameter variables have the same types.

[Note! If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method.]



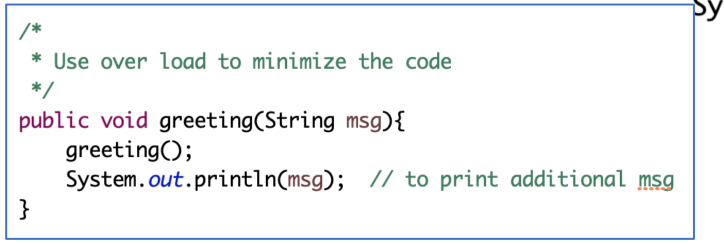
```java
public class Animal {
    private String name;

    public Animal(String n){
        this.name = n;
    }

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }

    /*
     * Overload method: same name but different signature
     */
    public void greeting(String msg){
        System.out.println("I'm " + name);
        System.out.println(msg);   // to print the given msg
    }
}
```

Override greeting() method



```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    /*
     * Override method: same name same signature
     * from the superclass (Animal)
     */
    @Override
    public void greeting(){
        System.out.println("Meow!  ");
        super.greeting();
    }

    public void chasing(){
        System.out.println("Chasing mouse...");
```

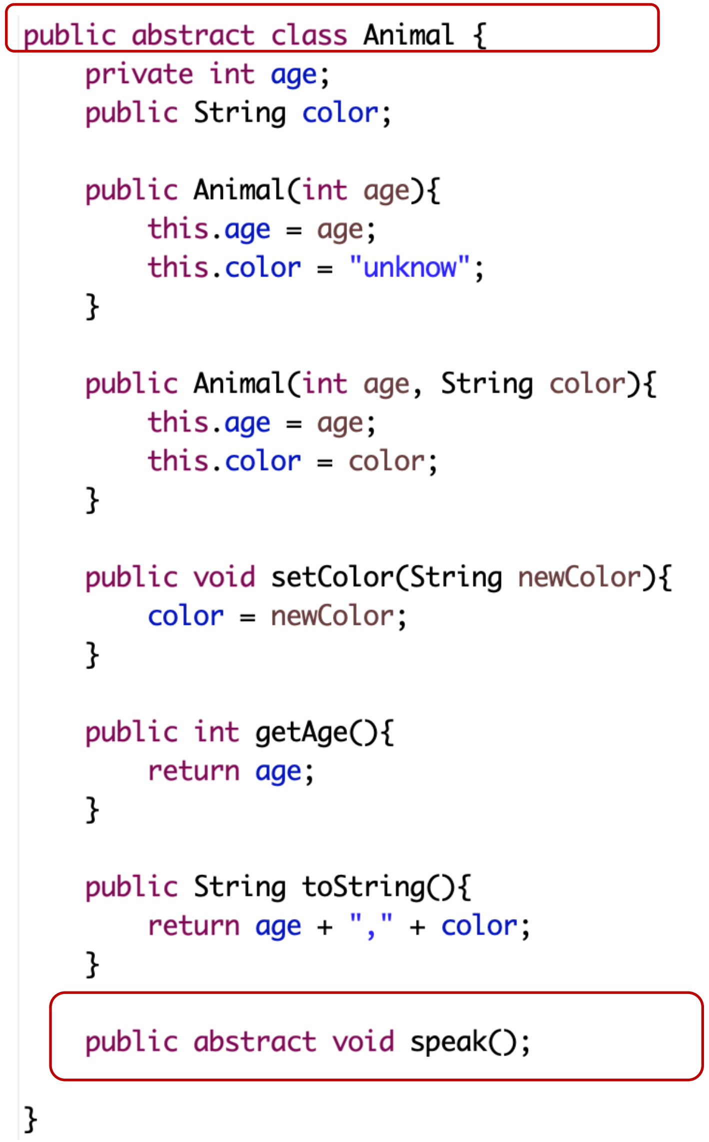
Example overloading usage



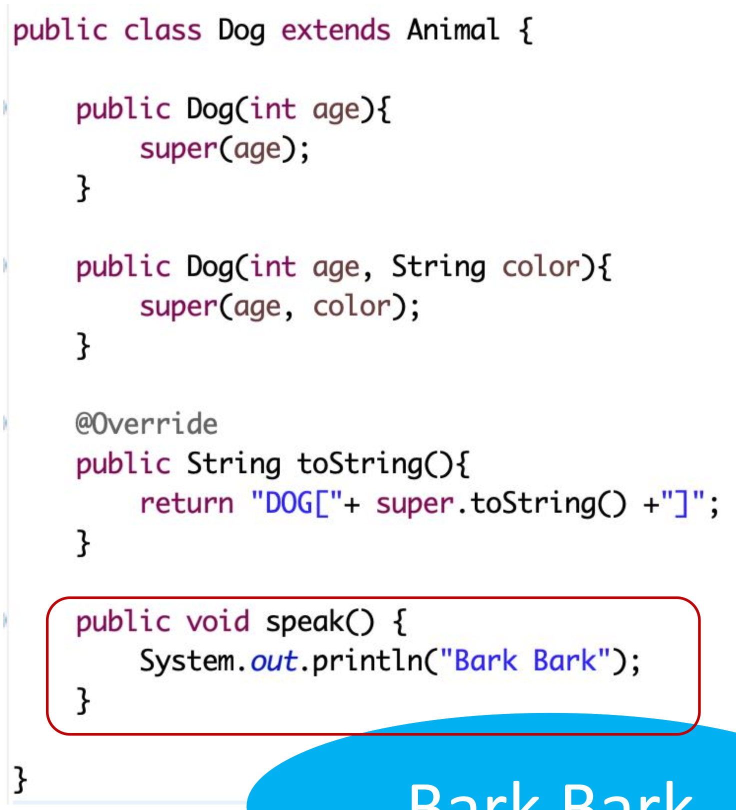
```java
/*
 * Use over load to minimize the code
 */
public void greeting(String msg){
    greeting();
    System.out.println(msg);  // to print additional msg
}
```

Interface and Abstract

Abstract Class & Abstract Method



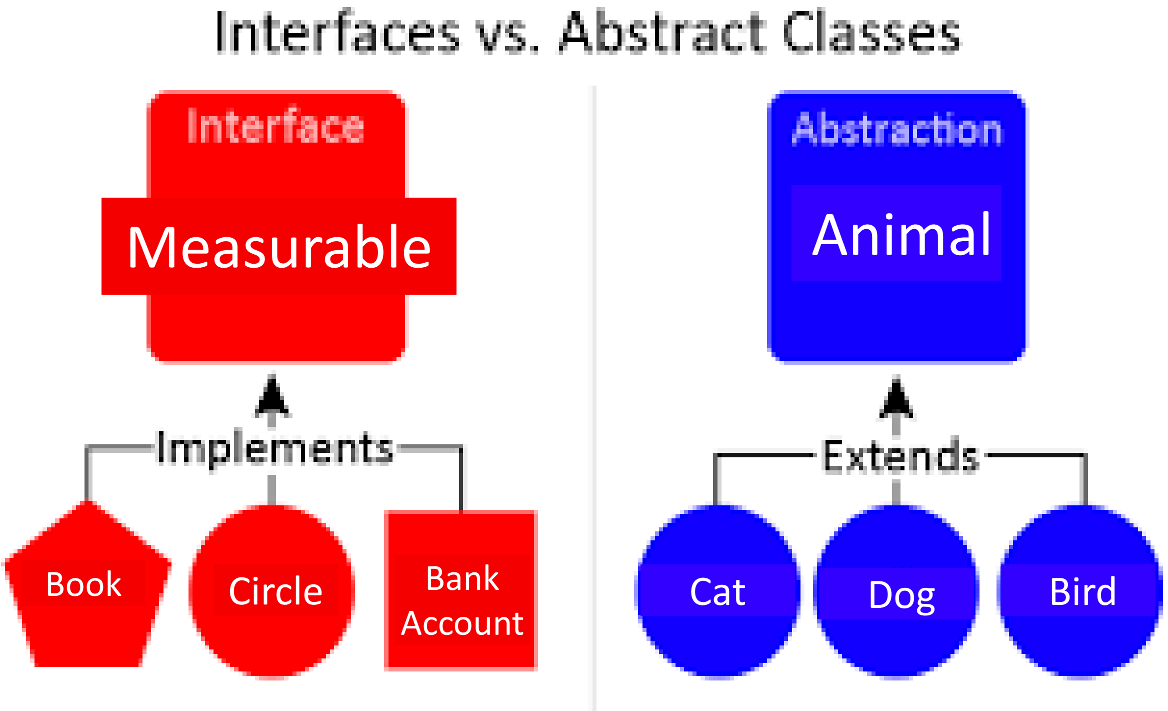
```
public abstract class Animal {
    private int age;
    public String color;

    public Animal(int age){
        this.age = age;
        this.color = "unknow";
    }

    public Animal(int age, String color){
        this.age = age;
        this.color = color;
    }

    public void setColor(String newColor){
        color = newColor;
    }

    public int getAge(){
        return age;
    }

    public String toString(){
        return age + "," + color;
    }

    public abstract void speak();

}
```



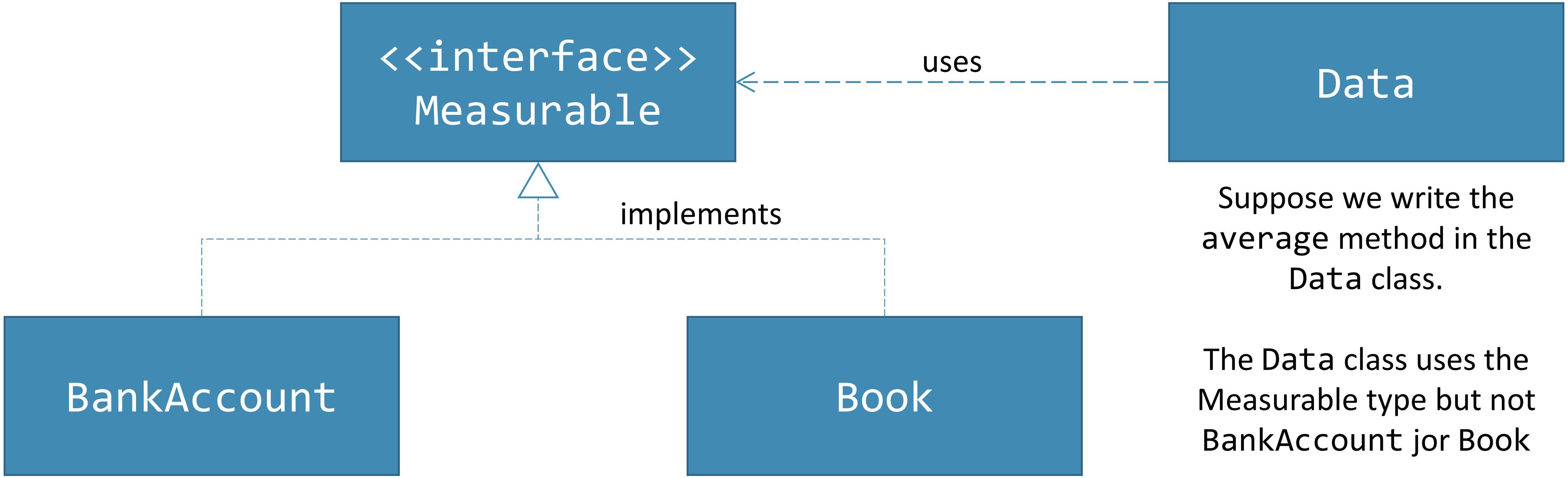
```
public class Dog extends Animal {

    public Dog(int age){
        super(age);
    }

    public Dog(int age, String color){
        super(age, color);
    }

    @Override
    public String toString(){
        return "DOG["+ super.toString() +"]";
    }

    public void speak() {
        System.out.println("Bark Bark");
    }

}
```

Review Interface

- Declare and implement interfaces
- Using interface for algorithm reuse
  - e.g., average(Measurable[] obj)
- Interface vs Abstract Class
  - Implements many interfaces // OK
- Comparable Java Standard Interface
  - Support sort() method in java



UML Diagram

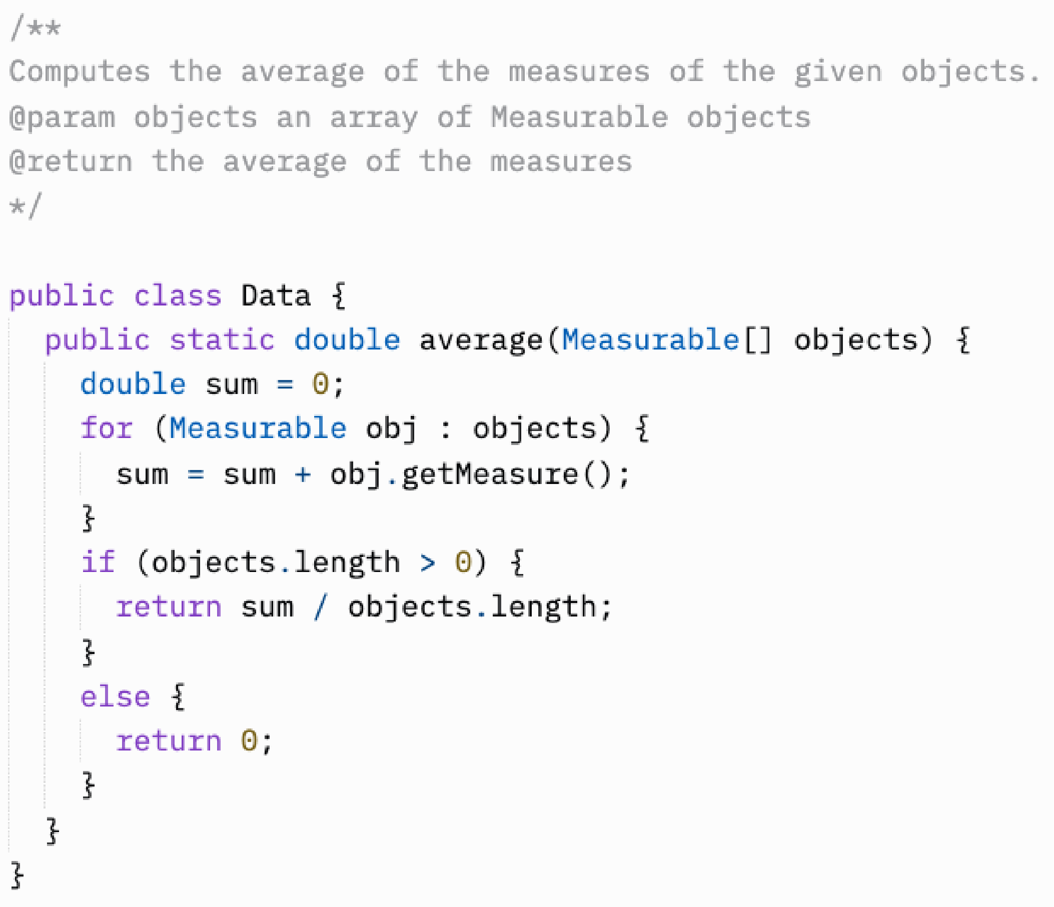


Suppose we write the average method in the Data class.

The Data class uses the Measurable type but not BankAccount jor Book

The BankAccount and Book classes implement the Measurable interface type

```
public static void main(String[] args) {
    BankAccount[] accounts = new BankAccount[3];
    accounts[0] = new BankAccount("001", 100);
    accounts[1] = new BankAccount("002", 200);
    accounts[2] = new BankAccount("003", 300);

    double avgBalance = Data.average(accounts);
    System.out.println("Avg balance:
          Expect= 200, actual= " + avgBalance);


    Book[] books = new Book[3];
    books[0] = new Book(50);
    books[1] = new Book(40);
    books[2] = new Book(30);

    double avgPrice = Data.average(books);
    System.out.println("Avg balance:
         Expect= 40, actual= " + avgPrice);

  }
```



```
/**
Computes the average of the measures of the given objects.
@param objects an array of Measurable objects
@return the average of the measures
*/

public class Data {
  public static double average(Measurable[] objects) {
    double sum = 0;
    for (Measurable obj : objects) {
      sum = sum + obj.getMeasure();
    }
    if (objects.length > 0) {
      return sum / objects.length;
    }
    else {
      return 0;
    }
  }
}
```

Checkpoint: Which statements cause ERROR?

- Suppose there are two classes and two interfaces as follow:
- public class *ClassA*
- public abstract class *ClassB*
- public interface *InterfaceC*
- public interface *InterfaceD*

Class Declaration

```
a) public class X extends ClassA
b) public class Y extends ClassB
c) public class Z implements InterfaceC
d) public class AC extends ClassA implements InterfaceC
e) public class AB extends ClassA, ClassB
f) public class CD implements InterfaceC, InterfaceD
```

Instantiate Objects

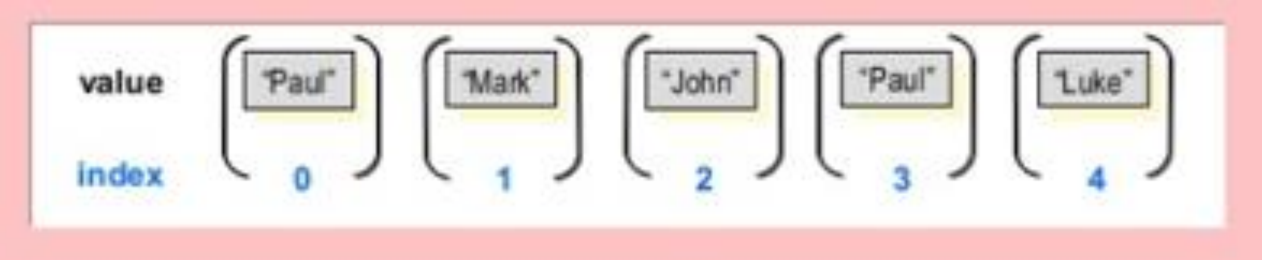
```
1) ClassA var = new ClassA();
2) ClassB var = new ClassB();
3) InterfaceC var = new InterfaceC();
4) ClassA var = new X();
5) ClassB var = new Y();
6) InterfaceC var = new CD();
```

Answer:
Statements e, 2, and 3

Java Collection

Set, List, and Map

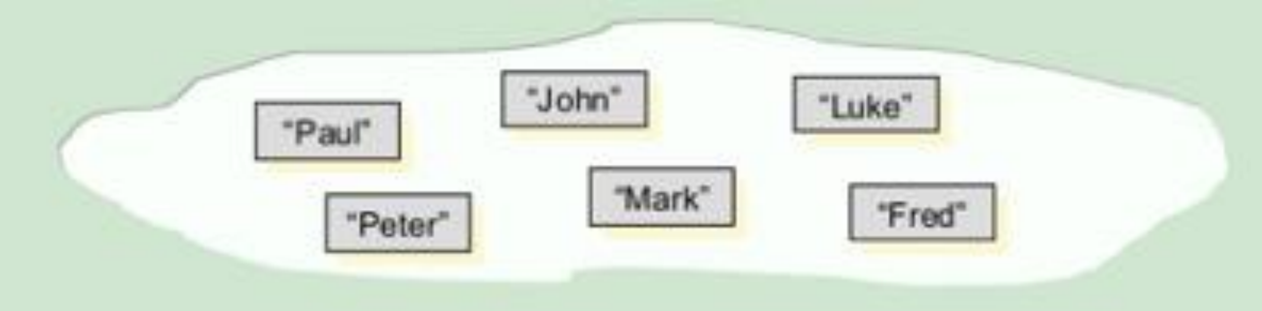
Review Collection



List: Lists of things (classes that implement List)

* cares about the index

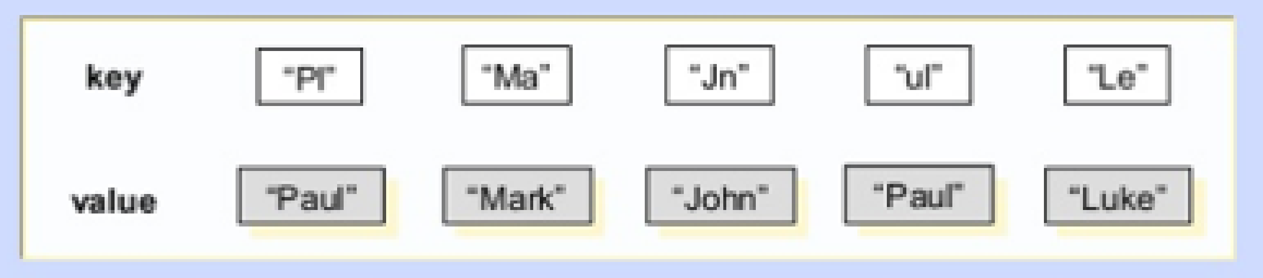
e.g., **ArrayList, Vector, LinkedList**



Set: Unique things (classes that implement Set)

* cares about uniqueness, no duplicate

e.g., **HashSet, LinkedHashSet, TreeSet**



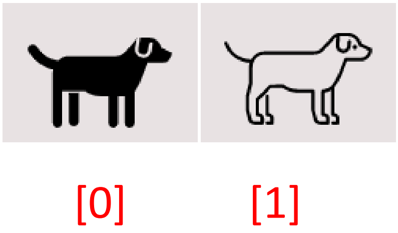
Map: Things with a unique ID
(classes that implement Map)

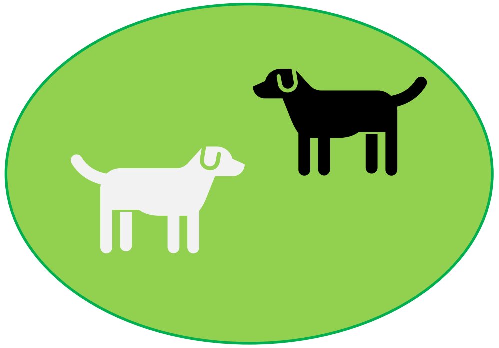
* cares about unique identifiers (key-value pair)

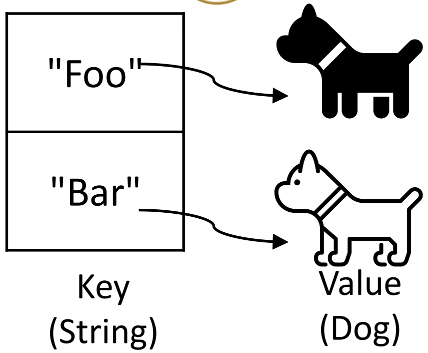
e.g., **HashMap, HashTable, TreeMap**

Summary

dogs =







	ArrayList	Set	
Add/Insert new element	dogs.add(new Dog(12, "black”));	dogs.add(new Dog(12, "black”));	dogs.put("key", new Dog(12, "black”));
Retrieve element	dogs.get(0); // index	Cannot directly get by index	dogs.get("key"); // key
Remove element	dogs.remove(0); // index	dogs.remove(dogObject); // Object	dogs.remove("key"); // key

Size / is it empty?	dogs.size();
	dogs.isEmpty()
		dogs.size(); dogs.isEmpty();	dogs.size(); dogs.isEmpty();

Iterate (loop through all elements)	for(Dog d: dogs) {
	 /* do s.th */
	};
		for(Dog d: dogs) {
		  /* do s.th */
		};
			Loop using keySet(); method

Find specific element	dogs.contains(dogObject);	dogs.contains(dogObject);	dogs.containsKey("key"); // OR
			dogs.containsValue(dogObject);

2. Common Ways to Traverse a Collection

- **Normal For Loop**

```
for(int i=0; i < objects.size(); i++) { . . . }
```

- **For-Each Loop**

```
for(Object obj: objects) { . . . }
```

- **Iterator & While Loop**

```
Iterator<Object> it = objects.iterator();
while(it.hasNext()) { . . . }
```

- `forEach()` **method**

```
objects.forEach(obj -> { . . . } );
```

For-Each loop vs forEach() method

```
//ArrayList
for(Dog dog: dogList){
  System.out.println("List => " + dog);
}

// Set
for(Dog dog: dogSet){
  System.out.println("Set => " + dog);
}

// Map -> have to loop through keySet() instead
for(String key: dogMap.keySet()){
  System.out.println("Map => key: " + key
    + ", value " + dogMap.get(key));
}
```

```
// ArrayList
dogList.forEach(dog -> {
  System.out.println("List => " + dog);
});

// Set
dogSet.forEach(dog -> {
  System.out.println("Set => " + dog);
});

// Map
dogMap.forEach((k, v)-> {
  System.out.println("key: " + k + ", value " + v);
});
```

Example Iterator: List vs Set vs Map

```
Iterator<Dog> cursorList = dogList.iterator();

while(cursorList.hasNext()) {
  System.out.println("List => " + cursorList.next());
}
```

```
Iterator<Dog> cursorSet = dogSet.iterator();

while(cursorList.hasNext()) {
  System.out.println("Set => " + cursorList.next());
}
```

Move cursor to next element

```
// For Map, we cannot get value (Dog) directly

// 1. Getting a Set of key-value pairs
Set entrySet = dogMap.entrySet();

// 2. Obtaining an interator for the entry set (key-value)
Iterator<Map.Entry<String, Dog>> it = entrySet.iterator();

while(it.hasNext()) {
  Map.Entry mapElement = it.next();
  System.out.println("Map => key: " + mapElement.getKey() +
                     ", value " + mapElement.getValue());
}
```

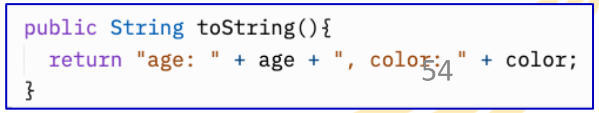
Map.Entry<key, value>
In this example, key is String and value is Dog.

mapElement.getKey() -> return key
mapElement.getValue() -> return value

OUTPUT

```
Map => key: Bar, value age: 10, color: white
Map => key: Foo, value age: 12, color: black
```

Dog Class

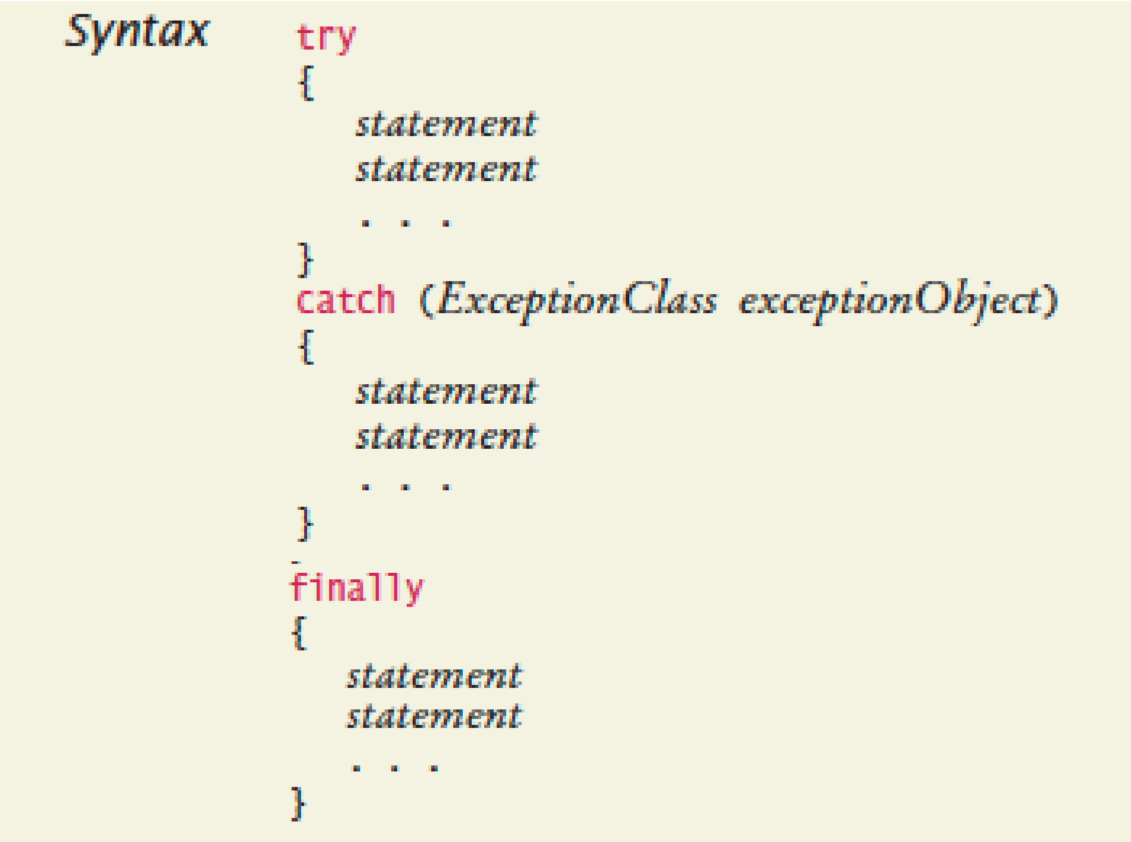


```
public String toString(){
  return "age: " + age + ", color: " + color;
}
```

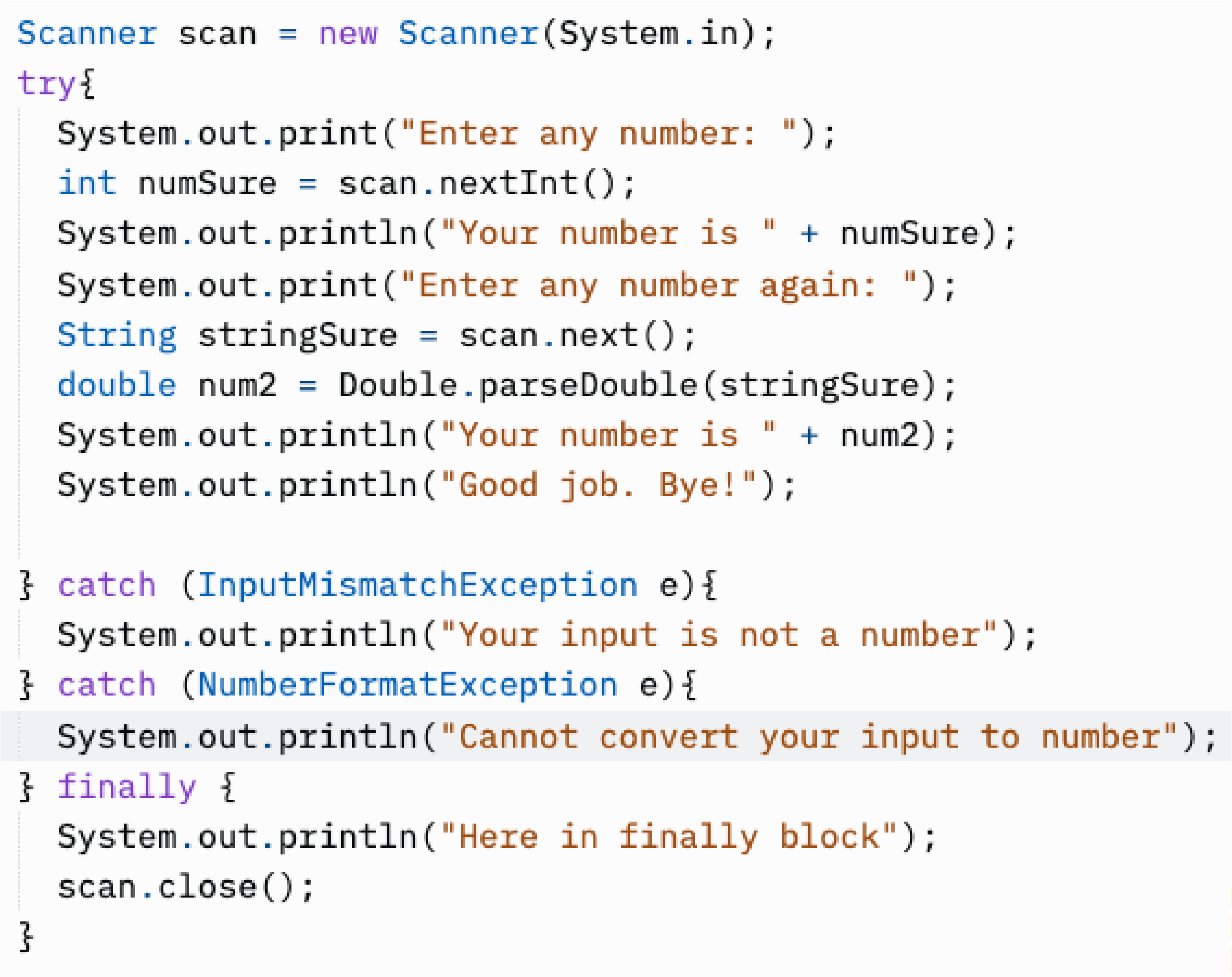
Exception

try, catch, finally, throw, throws

Review Exception Handling: Try, Catch, Finally



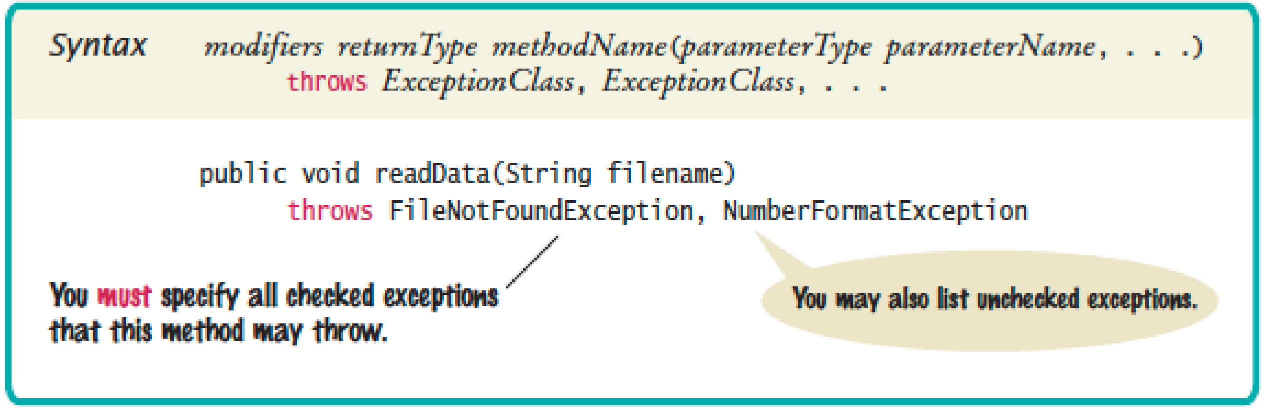
```
Syntax    try
          {
             statement
             statement
             . . .
          }
          catch (ExceptionClass exceptionObject)
          {
             statement
             statement
             . . .
          }
          finally
          {
             statement
             statement
             . . .
          }
```



```java
Scanner scan = new Scanner(System.in);
try{
  System.out.print("Enter any number: ");
  int numSure = scan.nextInt();
  System.out.println("Your number is " + numSure);
  System.out.print("Enter any number again: ");
  String stringSure = scan.next();
  double num2 = Double.parseDouble(stringSure);
  System.out.println("Your number is " + num2);
  System.out.println("Good job. Bye!");

} catch (InputMismatchException e){
  System.out.println("Your input is not a number");
} catch (NumberFormatException e){
  System.out.println("Cannot convert your input to number");
} finally {
  System.out.println("Here in finally block");
  scan.close();
}
```

Review Exception Handling: throw, throws

The throws Clause



Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
throws *ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Principle:

Throw an exception as soon as a problem is detected.

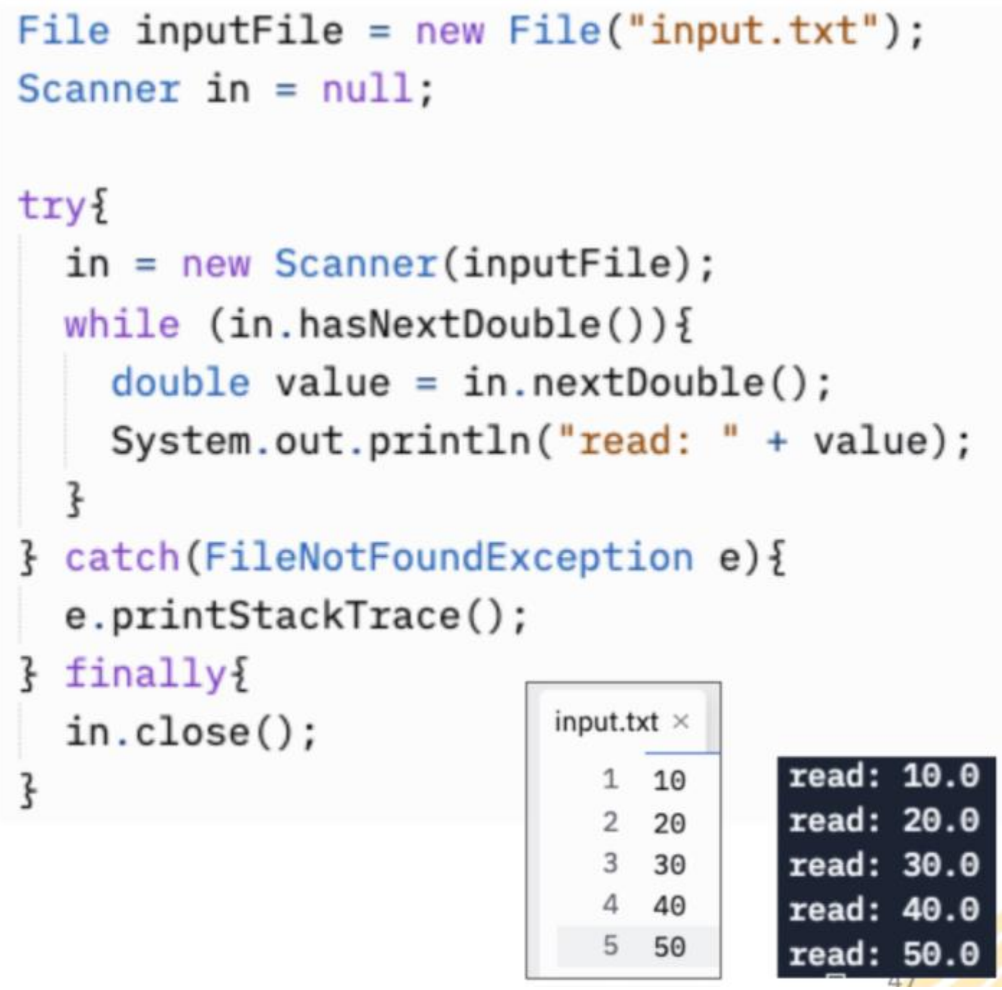
Catch it only when the problem can be handled.

```
public void withdraw(double amount) throws IllegalArgumentException {
    if (amount > balance) {
        throw new IllegalArgumentException("Amount exceeds balance");
    } else {
        balance = balance – amount;
    }
}
```

File Management

Review Read File with Scanner

- The most covenient mechanism for reading text file: ‘Scanner’ class
- To read input from a disk file, you need ‘File’ Class as well
  - File describes file’s name and directory
- Then, you can use the Scanner methods such as **nextInt**, **nextDouble**, and **next** to read data from the input file.



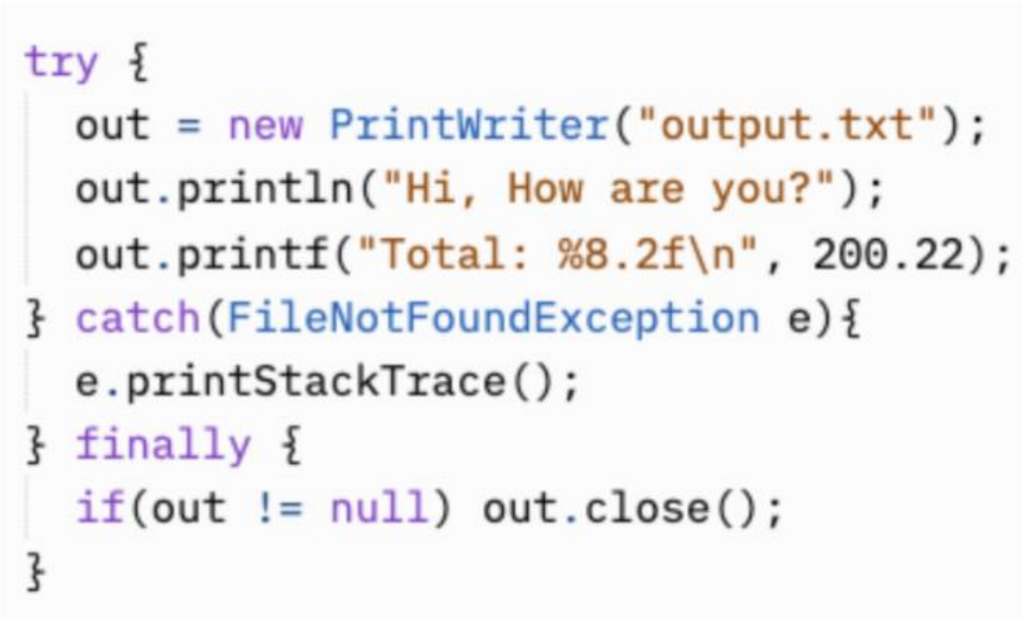
```
File inputFile = new File("input.txt");
Scanner in = null;

try{
  in = new Scanner(inputFile);
  while (in.hasNextDouble()){
    double value = in.nextDouble();
    System.out.println("read: " + value);
  }
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally{
  in.close();
}
```

```
10
20
30
40
50
```

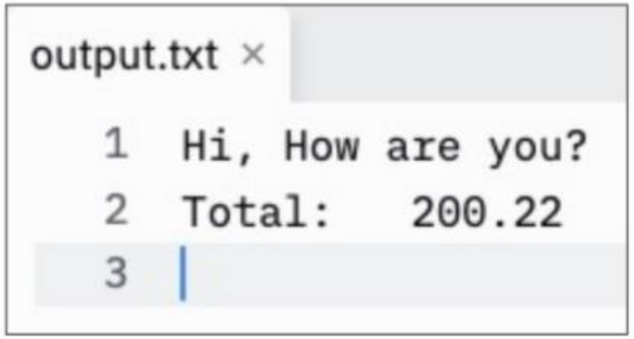
```
read: 10.0
read: 20.0
read: 30.0
read: 40.0
read: 50.0
```

Review Write File with FileWriter

- To write output to a file, you can use ‘PrintWriter’ object with the specific file name
- If the file already exit, the new data will replace the old one.
- If not, an empty file is created and the data is written.
- You can use the **print**, **println**, and **prinf** methods (similar to PrintStrem class)

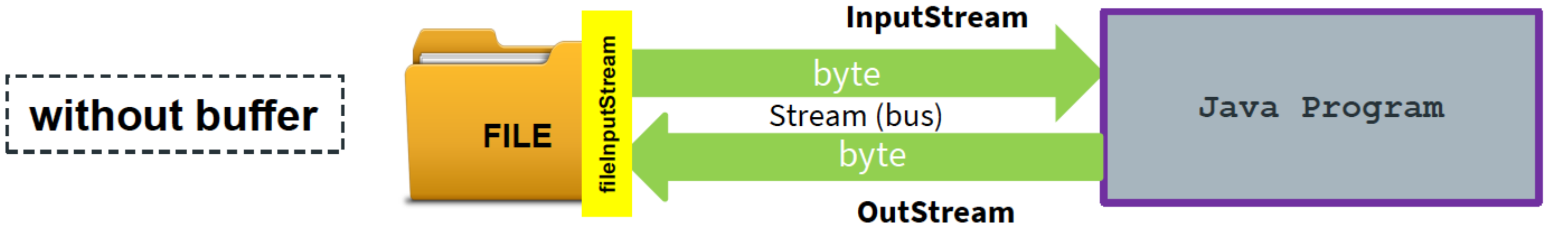


```
try {
  out = new PrintWriter("output.txt");
  out.println("Hi, How are you?");
  out.printf("Total: %8.2f\n", 200.22);
} catch(FileNotFoundException e){
  e.printStackTrace();
} finally {
  if(out != null) out.close();
}
```

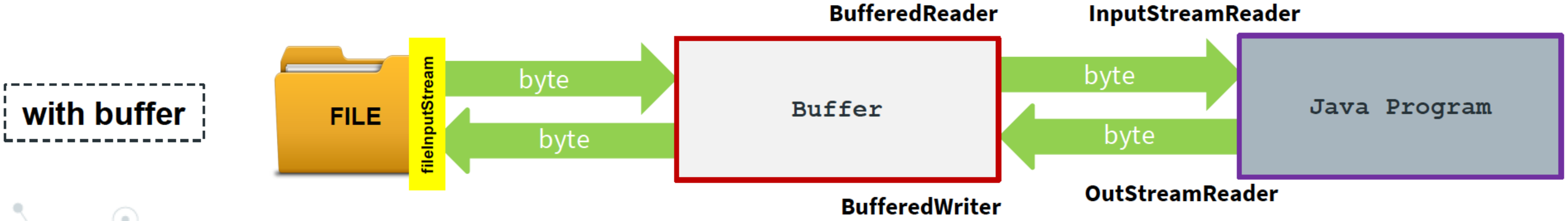


```
Hi, How are you?
Total:   200.22

```

Review: File Management



Scanner, FileWriter Class



Efficient way to read/write fiels. A data buffere is temporaly memeory that can be faster access than disk.

2. Reading/Writing a CSV File

- CSV files can be large. **It is advised to read and write a CSV file in a buffer manner.** That is you should not store a bunch of string and dump it all to a file at once, nor should you read all the content into memory before processing.
- There are many ways to read/write CSV files. Due to its simple format, you can write your own CSV reader/writer from scratch.
- But here, we will show you how to use **Apache Commons CSV**, which can save you some implementation time and help you deal with some rare cases.

Apache Commons™
http://commons.apache.org/

Commons CSV™

https://commons.apache.org/proper/commons-csv/

Writing CSV File

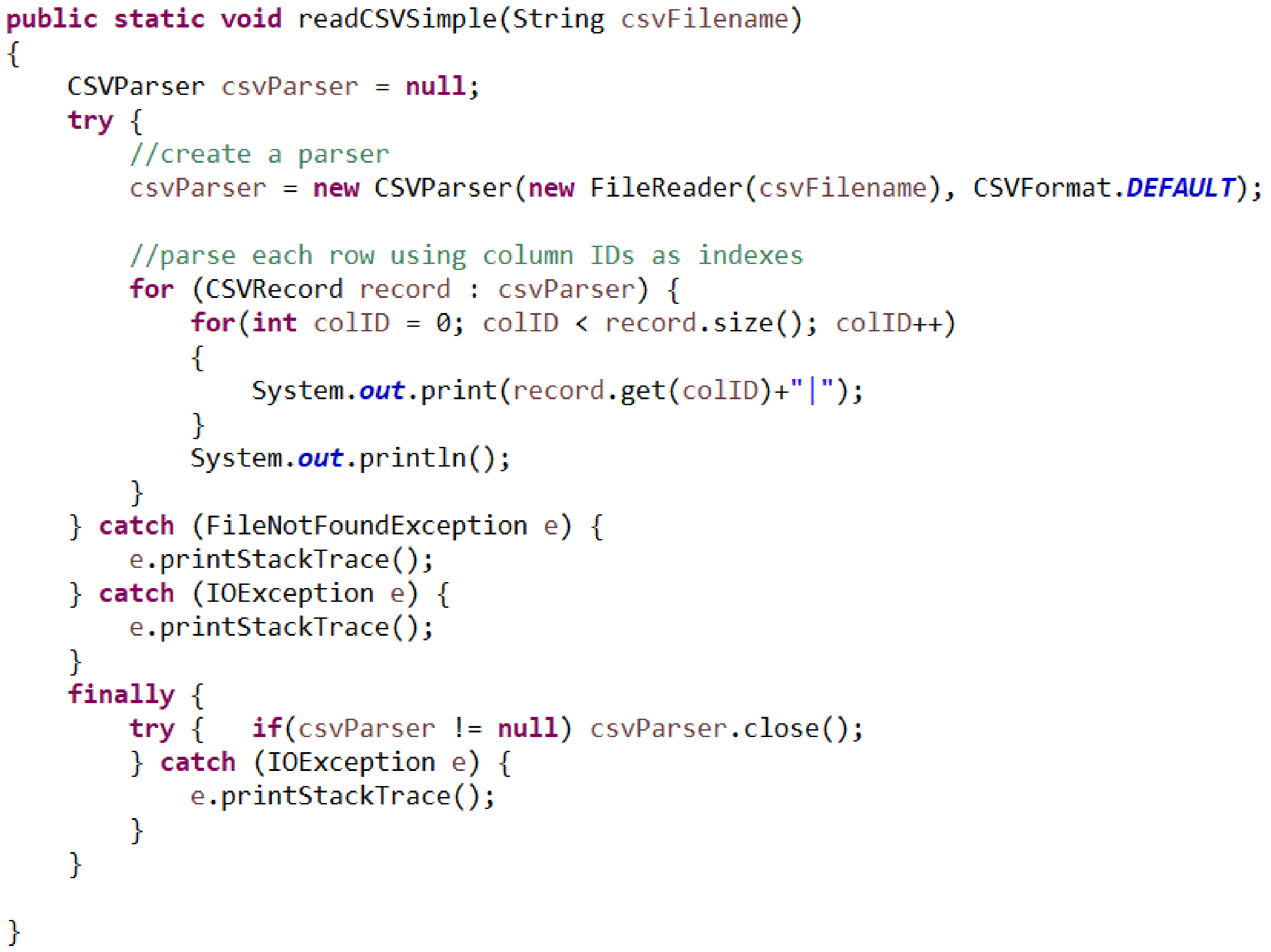
```java
public static final String[] header = {"ID", "Name", "E-mail"};
public static final String[][] students = {
        {"6488125", "David Beckham", "dback@school.edu"},
        {"6488126", "Christina Aguilera", "caguilera@school.edu"},
        {"6488127", "Lady Gaga", "lgaga@school.edu"}
};


public static void writeCSV(String outCsvFilename, String[] header, String[][] input)
{
    CSVPrinter printer = null;
    try {
        //create a CSV printer handler
        printer = new CSVPrinter(new FileWriter(outCsvFilename), CSVFormat.DEFAULT);

        //print headers
        printer.printRecord(Arrays.asList(header));

        //print data each row
        for(String[] row: input)
        {
            printer.printRecord(Arrays.asList(row));
        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally
    {   if(printer != null) try {   printer.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

```java
writeCSV("test-students.csv", header, students);
```

test-students.csv

```
ID,Name,E-mail
6488125,David Beckham,dback@school.edu
6488126,Christina Aguilera,caguilera@school.edu
6488127,Lady Gaga,lgaga@school.edu
```

Reading CSV (Do not care about headers)



```java
public static void readCSVSimple(String csvFilename)
{
    CSVParser csvParser = null;
    try {
        //create a parser
        csvParser = new CSVParser(new FileReader(csvFilename), CSVFormat.DEFAULT);

        //parse each row using column IDs as indexes
        for (CSVRecord record : csvParser) {
            for(int colID = 0; colID < record.size(); colID++)
            {
                System.out.print(record.get(colID)+"|");
            }
            System.out.println();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {   if(csvParser != null) csvParser.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

```java
readCSVSimple("small-no-headers.csv");
```

small-no-headers.csv

```
1995,Java,James Gosling
"1995","Java","James Gosling"
"March 22, 2022",Java, Quotes in "Cell"
```

Editor console

```
1995|Java|James Gosling|
1995|Java|James Gosling|
March 22, 2022|Java| Quotes in "Cell"|
```

Reading (Header-less) CSV with Custom Headers

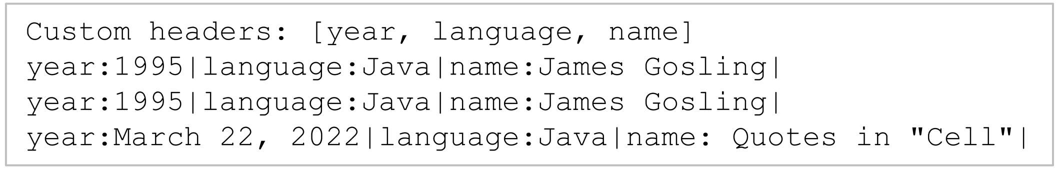
```
public static void readCSVwithCustomHeaders(String csvFilename, String[] headers)
{
    System.out.println("Custom headers: "+Arrays.toString(headers));
    CSVParser csvRecordsWithHeader = null;
    try {
        //create a parser with assigned custom headers
        csvRecordsWithHeader = CSVFormat.DEFAULT.~~withHeader~~(headers).parse(new FileReader(csvFilename));
        //Though deprecated, still usable

        //parse each row using String headers as indexes
        for (CSVRecord record : csvRecordsWithHeader) {
            for(int colID = 0; colID < headers.length; colID++)
            {
                System.out.print(headers[colID]+":"+record.get(headers[colID])+"|");
            }
            System.out.println();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {   if(csvRecordsWithHeader != null) csvRecordsWithHeader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

```
readCSVwithCustomHeaders("small-no-headers.csv",
new String[]{"year", "language", "name"});
```

small-no-headers.csv

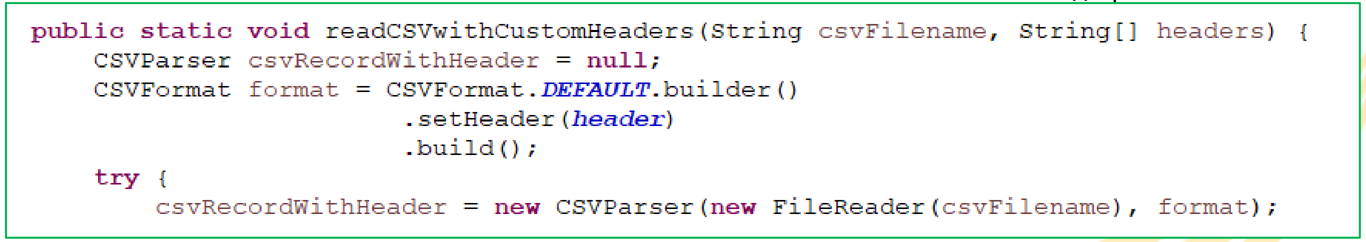
```
1995,Java,James Gosling
"1995","Java","James Gosling"
"March 22, 2022",Java, Quotes in "Cell"
```

Editor console



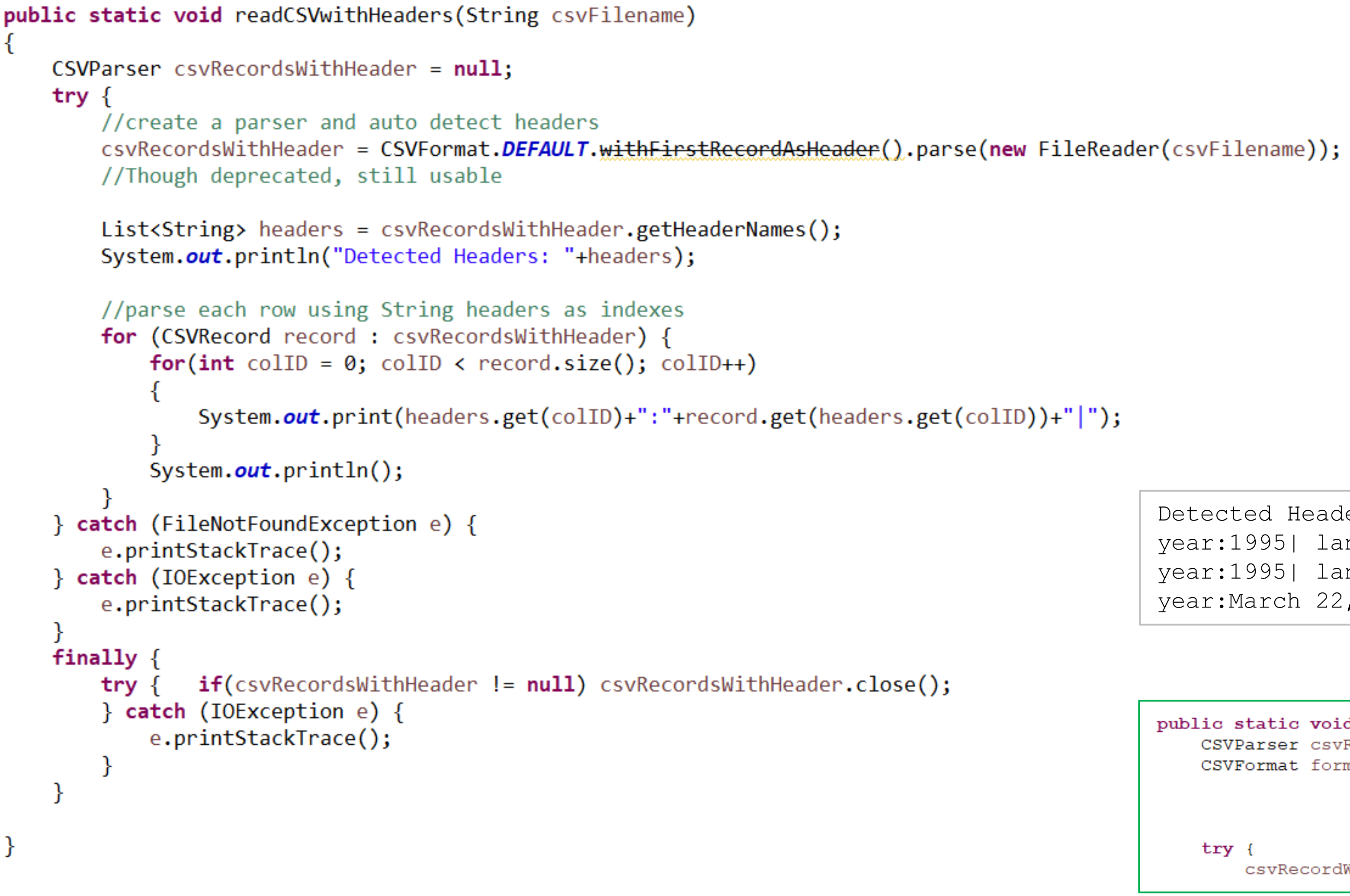
```
Custom headers: [year, language, name]
year:1995|language:Java|name:James Gosling|
year:1995|language:Java|name:James Gosling|
year:March 22, 2022|language:Java|name: Quotes in "Cell"|
```

//updated: 30Mar2023



```
public static void readCSVwithCustomHeaders(String csvFilename, String[] headers) {
    CSVParser csvRecordWithHeader = null;
    CSVFormat format = CSVFormat.DEFAULT.builder()
                            .setHeader(header)
                            .build();
    try {
        csvRecordWithHeader = new CSVParser(new FileReader(csvFilename), format);
```

Reading CSV with Headers



```
public static void readCSVwithHeaders(String csvFilename)
{
    CSVParser csvRecordsWithHeader = null;
    try {
        //create a parser and auto detect headers
        csvRecordsWithHeader = CSVFormat.DEFAULT.~~withFirstRecordAsHeader~~().parse(new FileReader(csvFilename));
        //Though deprecated, still usable

        List<String> headers = csvRecordsWithHeader.getHeaderNames();
        System.out.println("Detected Headers: "+headers);

        //parse each row using String headers as indexes
        for (CSVRecord record : csvRecordsWithHeader) {
            for(int colID = 0; colID < record.size(); colID++)
            {
                System.out.print(headers.get(colID)+":"+record.get(headers.get(colID))+"|");
            }
            System.out.println();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {   if(csvRecordsWithHeader != null) csvRecordsWithHeader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

```
readCSVwithHeaders("small-with-headers.csv");
```

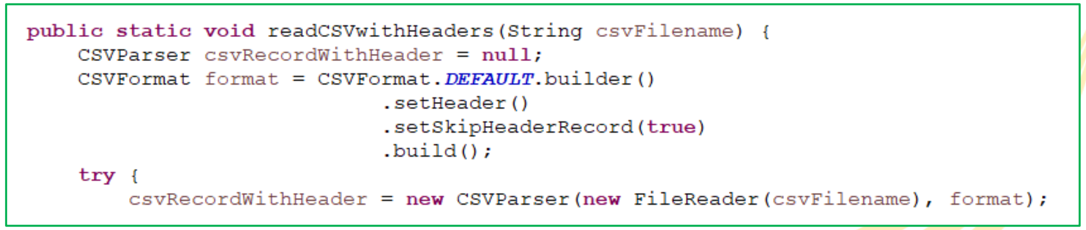
small-with-headers.csv

```
year, language, name
1995,Java,James Gosling
"1995","Java","James Gosling"
"March 22, 2022",Python, Quotes in "Cell"
```

Editor console

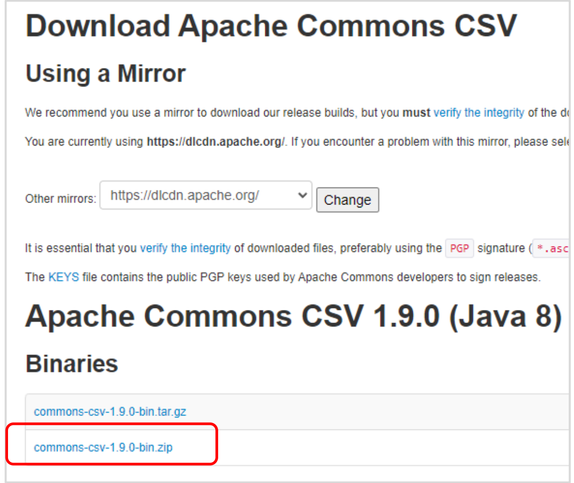
```
Detected Headers: [year,  language,  name]
year:1995| language:Java| name:James Gosling|
year:1995| language:Java| name:James Gosling|
year:March 22, 2022| language:Python| name: Quotes in "Cell"|
```

//updated: 30Mar2023



```
public static void readCSVwithHeaders(String csvFilename) {
    CSVParser csvRecordWithHeader = null;
    CSVFormat format = CSVFormat.DEFAULT.builder()
                            .setHeader()
                            .setSkipHeaderRecord(true)
                            .build();
    try {
        csvRecordWithHeader = new CSVParser(new FileReader(csvFilename), format);
```

Download and Import Commons CSV Library



Download Apache Commons CSV

Using a Mirror

We recommend you use a mirror to download our release builds, but you **must** verify the integrity of the d

You are currently using **https://dlcdn.apache.org/**. If you encounter a problem with this mirror, please sele

Other mirrors: https://dlcdn.apache.org/ Change

It is essential that you verify the integrity of downloaded files, preferably using the PGP signature (*.asc

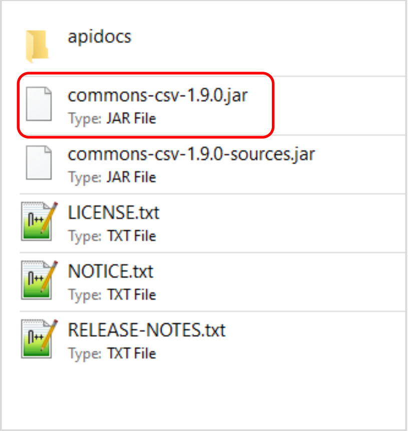
The KEYS file contains the public PGP keys used by Apache Commons developers to sign releases.

Apache Commons CSV 1.9.0 (Java 8)

Binaries

commons-csv-1.9.0-bin.tar.gz

commons-csv-1.9.0-bin.zip



apidocs

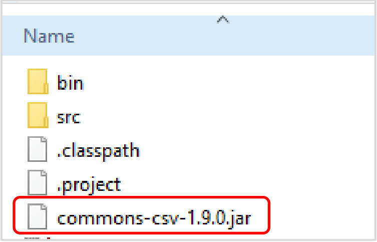
commons-csv-1.9.0.jar
Type: JAR File

commons-csv-1.9.0-sources.jar
Type: JAR File

LICENSE.txt
Type: TXT File

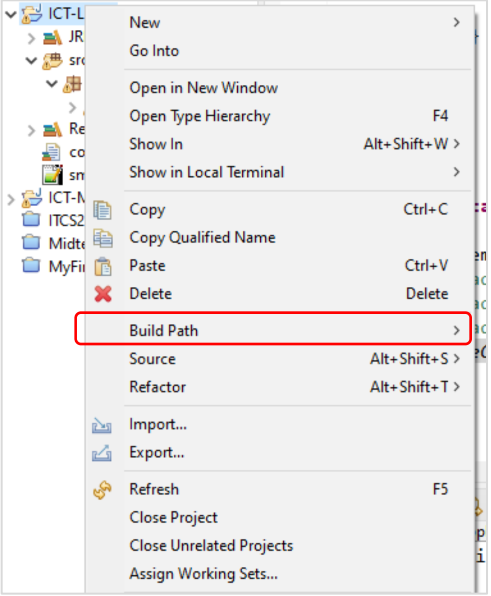
NOTICE.txt
Type: TXT File

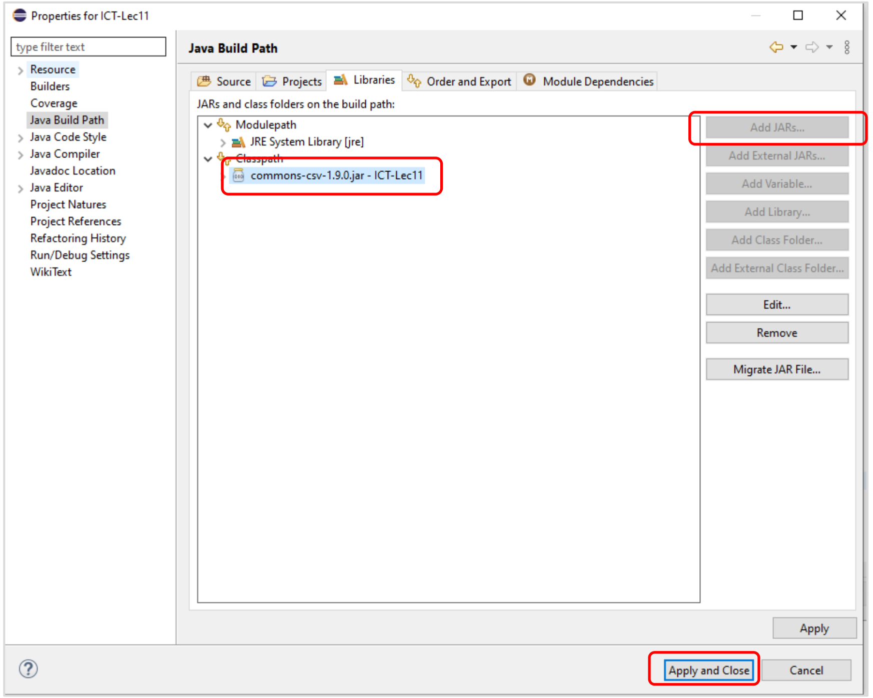
RELEASE-NOTES.txt
Type: TXT File



Name

bin
src
.classpath
.project
commons-csv-1.9.0.jar





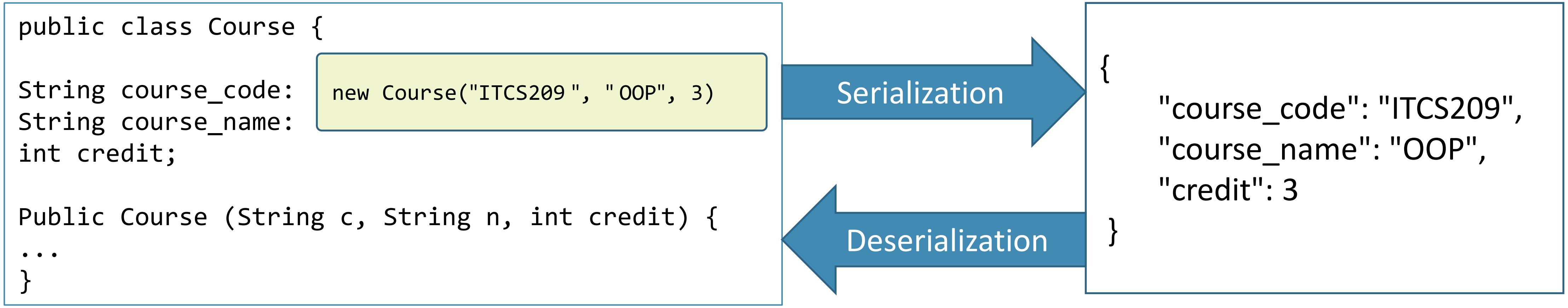
Properties for ICT-Lec11

JARs and class folders on the build path:

- Modulepath
  - JRE System Library [jre]
- Classpath
  - commons-csv-1.9.0.jar - ICT-Lec11

Working with JSON Format

- Two things are required:
  - **Serialization**: encode Java Object to its JSON representation
  - **Deserialization**: decode String back to an equivalent Java Object



```
public class Course {

String course_code:
String course_name:
int credit;

Public Course (String c, String n, int credit) {
...
}
```

```
new Course("ITCS209 ", " OOP", 3)
```

```
{
      "course_code": "ITCS209",
      "course_name": "OOP",
      "credit": 3
 }
```

- How to do that in Java?
  - Option 1: Write your own code to parse JSON text file // NOT recommended
  - Option 2: Using external Java library // YES YES YES

Simple Serialization GSON

```java
public class Course {
   private String name;
   private int credit;
   private List<String> instructors;

   public Course(String name, int credit, List<String> instructors) {
      this.name = name;
      this.credit = credit;
      this.instructors = instructors;
   }
   ...
}

// -- main method --
Course course = new Course("OOP", 3, Arrays.asList("Siripen", "Petch", "Suppawong"));
String serializedCourse = new Gson().toJson(course);
System.out.println(serializedCourse);
```

OUTPUT

```
{"name":"OOP","credit":3,"instructors":["Siripen","Petch","Suppawong"]}
```

Simple Deserialization GSON

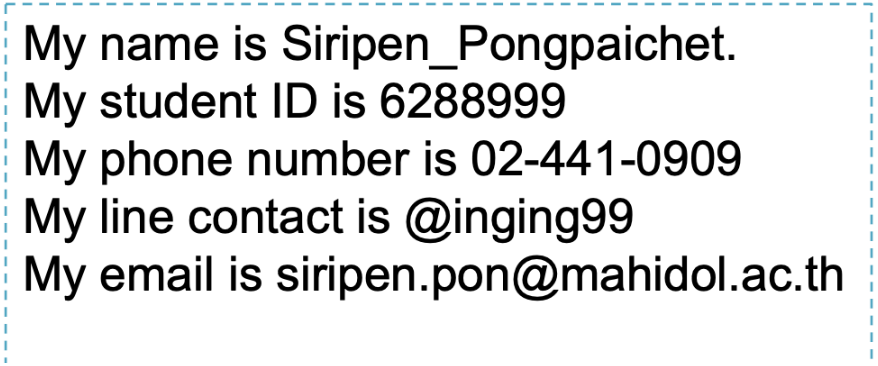
```java
public class Course {
   private String name;
   private int credit;
   private List<String> instructors;

   public Course(String name, int credit, List<String> instructors) {
      this.name = name;
      this.credit = credit;
      this.instructors = instructors;
   }

   public String toString() {
      return "name=" + name + "::credit=" + credit + "::instructors=" + instructors.toString();
   }
}

// -- main method --
String courseJson ="{\"name\":\"OOP\",\"credit\":3," +
    "\"instructors\":[\"Siripen\",\"Petch\",\"Suppawong\"]}";

Course oopCourse = new Gson().fromJson(courseJson, Course.class);
System.out.println(oopCourse);
```

OUTPUT

```
name=OOP::credit=3::instructors=[Siripen, Petch, Suppawong]
```

Regular Expression (RegEx)

Review: RegEx



My name is Siripen_Pongpaichet.
My student ID is 6288999
My phone number is 02-441-0909
My line contact is @inging99
My email is siripen.pon@mahidol.ac.th

Example Patterns

Information	Observation	Actual RegEx Patter
Name	{one or more letter}_{one or more letter}	`[a-zA-Z]+_[a-zA-Z]+`
Student ID	{7digits number only}	`[0-9]{7}`
Phone Number	{2digits}-{3digits}-{4digits}	`[0-9]{2}-[0-9]{3}-[0-9]{4}`
Line	@{text/number}	`@[\w]+`
Email	{one or more letter}.{3letter}@{text}.{th|com}	`[\w]+.[a-zA-Z]{3}@[\w.]+.[th|com]`

Basic Pattern

abc	**exactly this sequence of three letters**
[abc]	**any *one* of the letters a, b, or c**
[^abc]	**any character *except* one of the letters a, b, or c** (immediately within an open bracket, ^ means "not," but anywhere else it just means the character ^)
[a-z]	**any *one* character from a through z, inclusive**
[a-zA-Z0-9]	**any *one* letter or digit**

.	any one character except a line terminator
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\x0B\f\r]
\S	a non-whitespace character: [^\s]
\w	a word character: [a-zA-Z_0-9]
\W	a non-word character: [^\w]

Notice the space.
Spaces are **significant**
in regular expressions!

Quantifier (Assum X represetns some pattern)

X?	Optional, X occurs once or not	X{n}	X occurs exactly n times
X*	X occurs zero or more times	X{n,}X	X occurs at least n times
X+	X occurs one or more times	X{n,m}	X occurs at least n but not more than m times

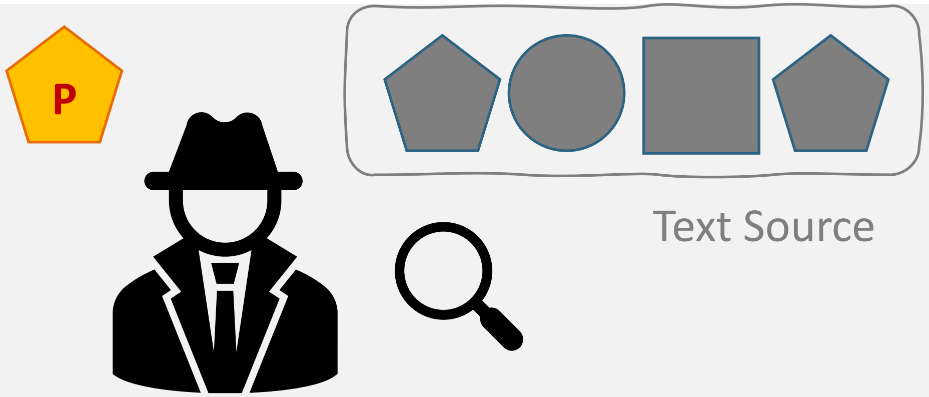
Alternative using veritcal bar, | -> e.g., abc|xyz

RegEx in Java

1. Import library (`import java.util.regex.*;`)
2. Define **RegEx** pattern in String format
3. Create and compile **`Pattern`** from the RegEx String in step 2
4. Create **`Matcher`** from the Pattern in step 3
5. *Scan the input sequence and find the subset of text that matches the pattern*

RegEx Code in Java

```
1  import java.util.regex.*;

   public class Part1_BasicRegEx {

         public static void main(String[] args) {

               String text = "abcdefgabcd";

2              String regex = "abc";

               Pattern p = Pattern.compile(regex);

4              Matcher m = p.matcher(text);

               System.out.println("matches the entire string: " + m.matches());

               System.out.println("matches at the beginning of string: " + m.lookingAt());

               System.out.println("matches any part of the text string: " + m.find());
         }
   }
```



3 Notice that neither Pattern nor Matcher has a public constructor; you create them using methods in the Pattern class.

5

```
false

true

true
```

Methods of Matcher class

- **Now that we have a matcher m,**
- `m.matches()` returns `true` if the pattern matches the *entire text string*, and `false` otherwise
- `m.lookingAt()` returns `true` if the pattern matches at *the beginning of the text string*, and `false` otherwise
- `m.find()` returns `true` if the pattern matches *any part of the text string*, and `false` otherwise
  - If called again, `m.find()` will start searching from where the last match was found
  - `m.find()` will return `true` for as many matches as there are in the string; after that, it will return `false`
  - When `m.find()` returns `false`, matcher m will be *reset* to the beginning of the text string (and may be used again)

Let's explore Step 5:

How many times subset of text matches the pattern?

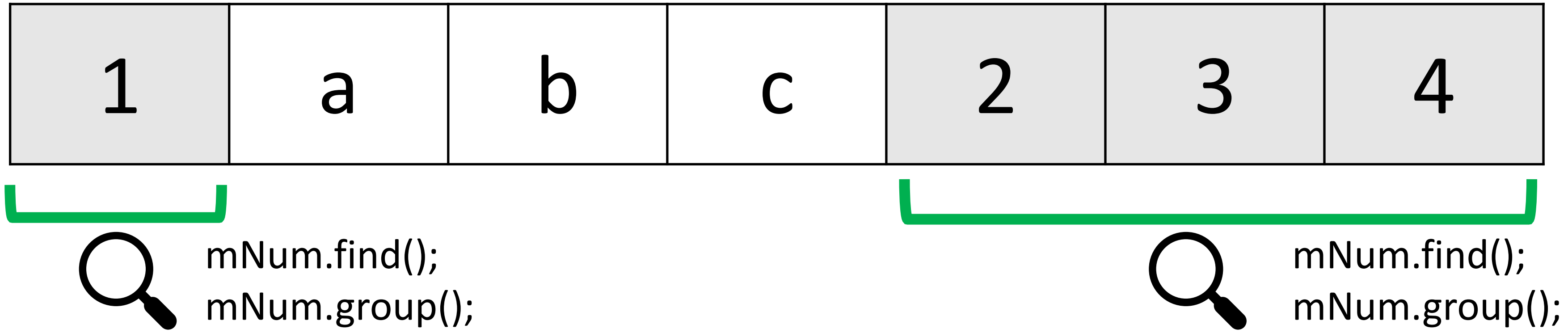
```
String text = "1abc234";

String regexNum = "\\d+";

Pattern pNum = Pattern.compile(regexNum);

Matcher mNum = pNum.matcher(text);

System.out.println("\n\nFrist group of text found: " + mNum.find());

System.out.println("Second group of text found: " + mNum.find());

System.out.println("Third group of txt found: " + mNum.find());
```

1	a	b	c	2	3	4

```
true

true

false
```

Let's explore Step 5:

How to get "all" text that matches the pattern?



1	a	b	c	2	3	4

mNum.find();
mNum.group();

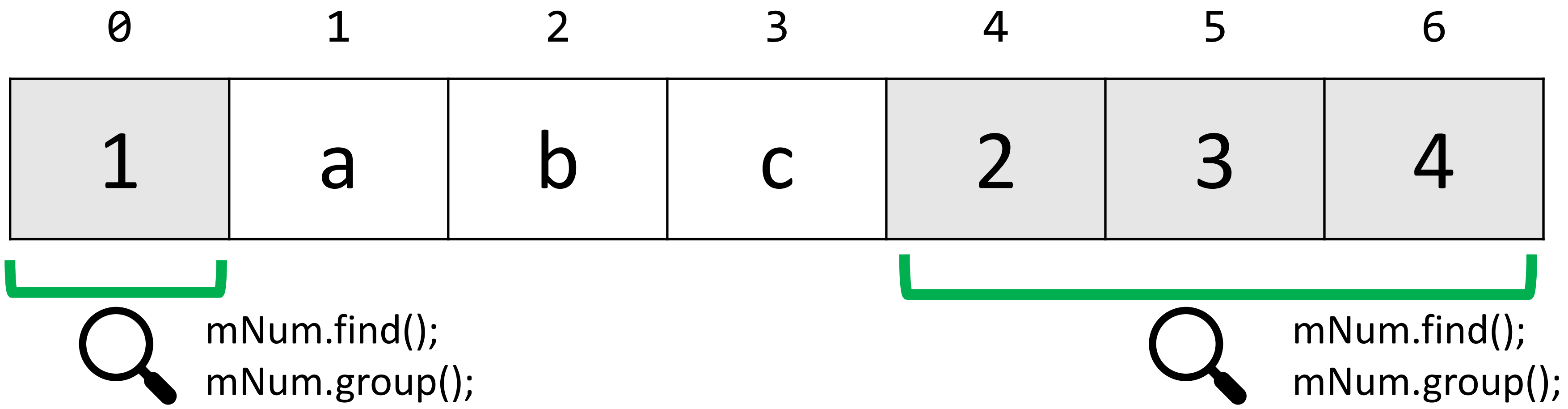
mNum.find();
mNum.group();

```
mNum.reset(); // The matcher's region is set to index 0

System.out.println("\ntext: " + text + ", regex: " + regexNum);
while(mNum.find()) {
    System.out.println("found: " +  mNum.group()) ;
}
```

```
text: 1abc234, regex: \d+
found: 1
found: 234
```

Let's explore Step 5:

How to get "all" text that matches the pattern?



0	1	2	3	4	5	6
1	a	b	c	2	3	4

mNum.find();
mNum.group();

mNum.find();
mNum.group();

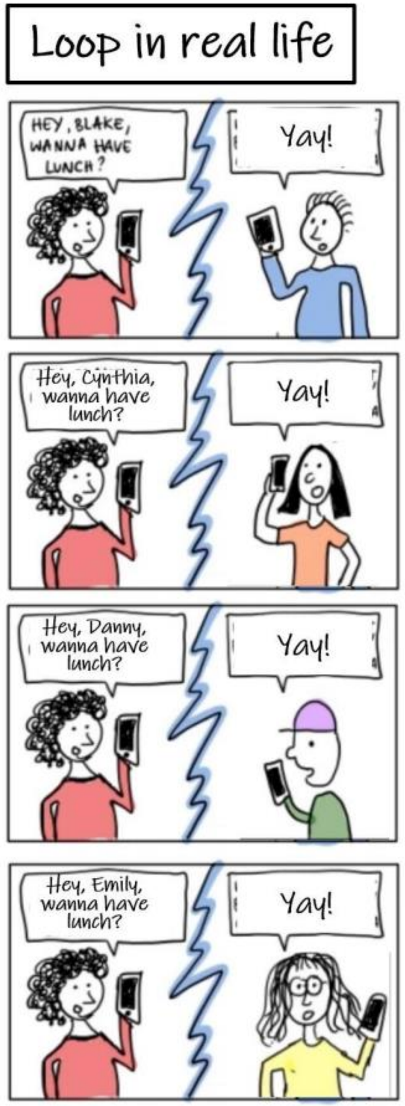
```
mNum.reset();
while(mNum.find()) {
        String found = text.substring(mNum.start(), mNum.end());
        System.out.println("["+ mNum.start() + "," + mNum.end() + "] -> " + found) ;
}
```

```
[0,1] -> 1
[4,7] -> 234
```

After a successful match, `m.start()` will return the index of the first character matched, and `m.end()` will return the index of the last character matched, *plus* one.

Recursion

<u>Review</u> Loop vs Recursion in Real Life: Lunch Invitation

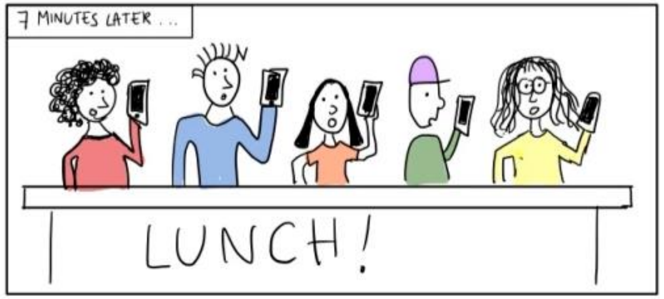

Loop in real life
HEY, BLAKE, WANNA HAVE LUNCH?
Yay!
Hey, Cynthia, wanna have lunch?
Yay!
Hey, Danny, wanna have lunch?
Yay!
Hey, Emily, wanna have lunch?
Yay!

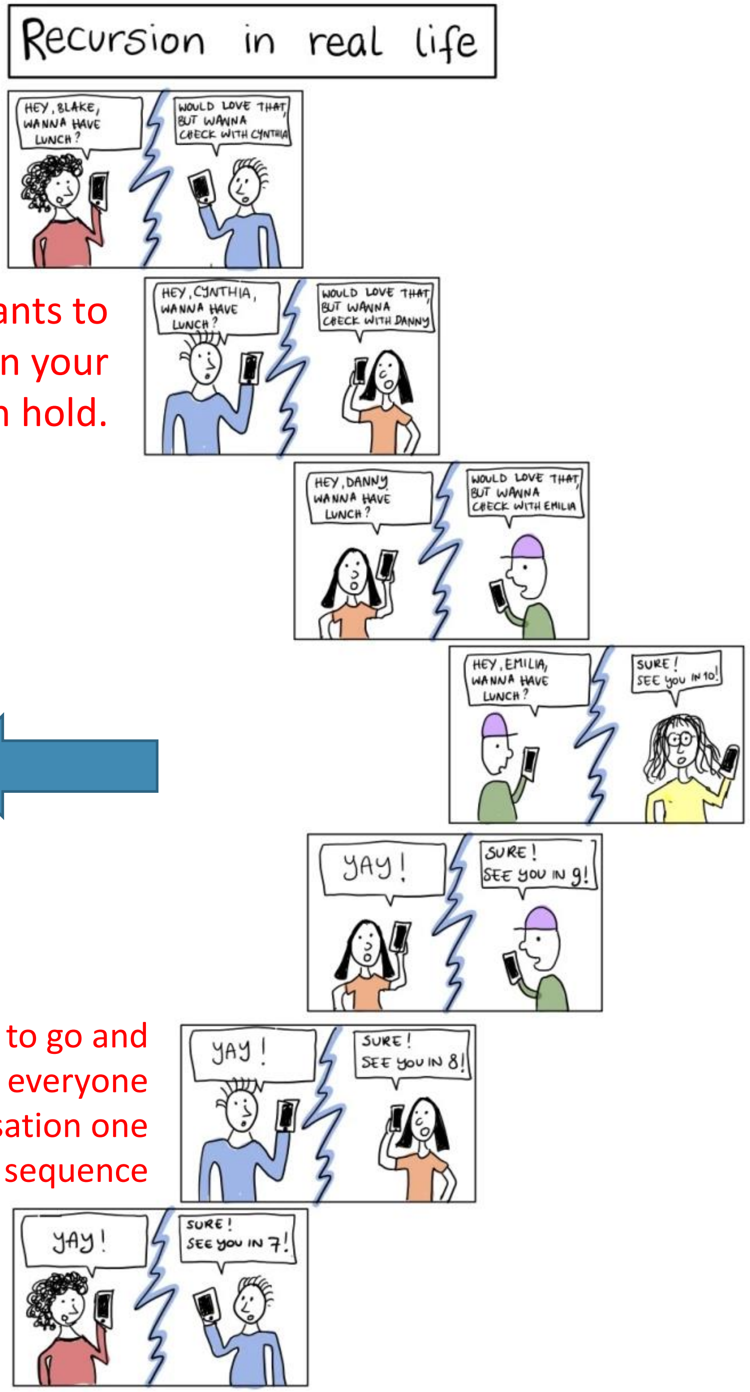
See you in 10 minutes

See you in 9 minutes

See you in 8 minutes

See you in 7 minutes

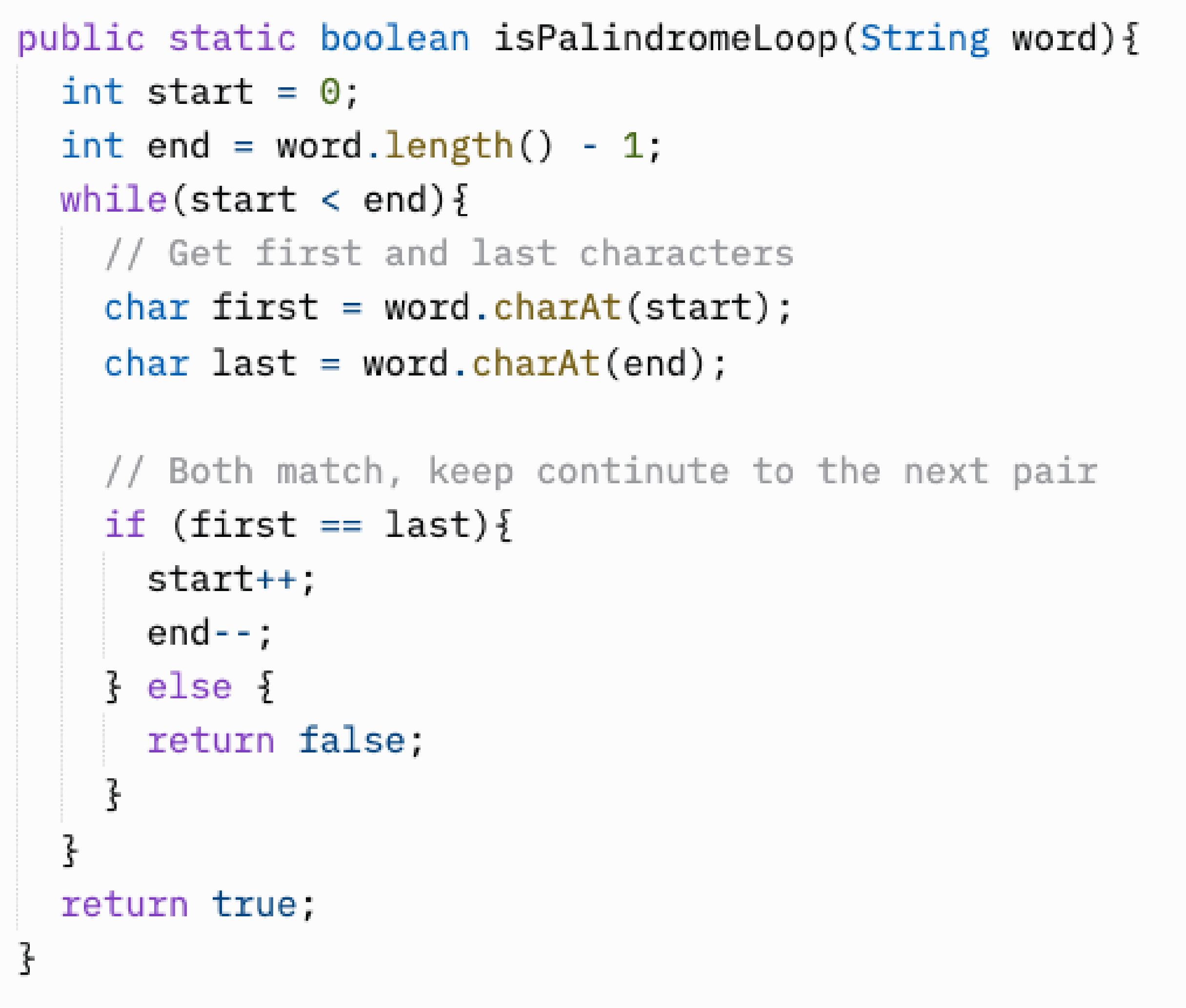



Recursion in real life
HEY, BLAKE, WANNA HAVE LUNCH?
WOULD LOVE THAT, BUT WANNA CHECK WITH CYNTHIA
HEY, CYNTHIA, WANNA HAVE LUNCH?
WOULD LOVE THAT, BUT WANNA CHECK WITH DANNY
HEY, DANNY, WANNA HAVE LUNCH?
WOULD LOVE THAT, BUT WANNA CHECK WITH EMILIA
HEY, EMILIA, WANNA HAVE LUNCH?
SURE! SEE YOU IN 10!
YAY!
SURE! SEE YOU IN 9!
YAY!
SURE! SEE YOU IN 8!
YAY!
SURE! SEE YOU IN 7!

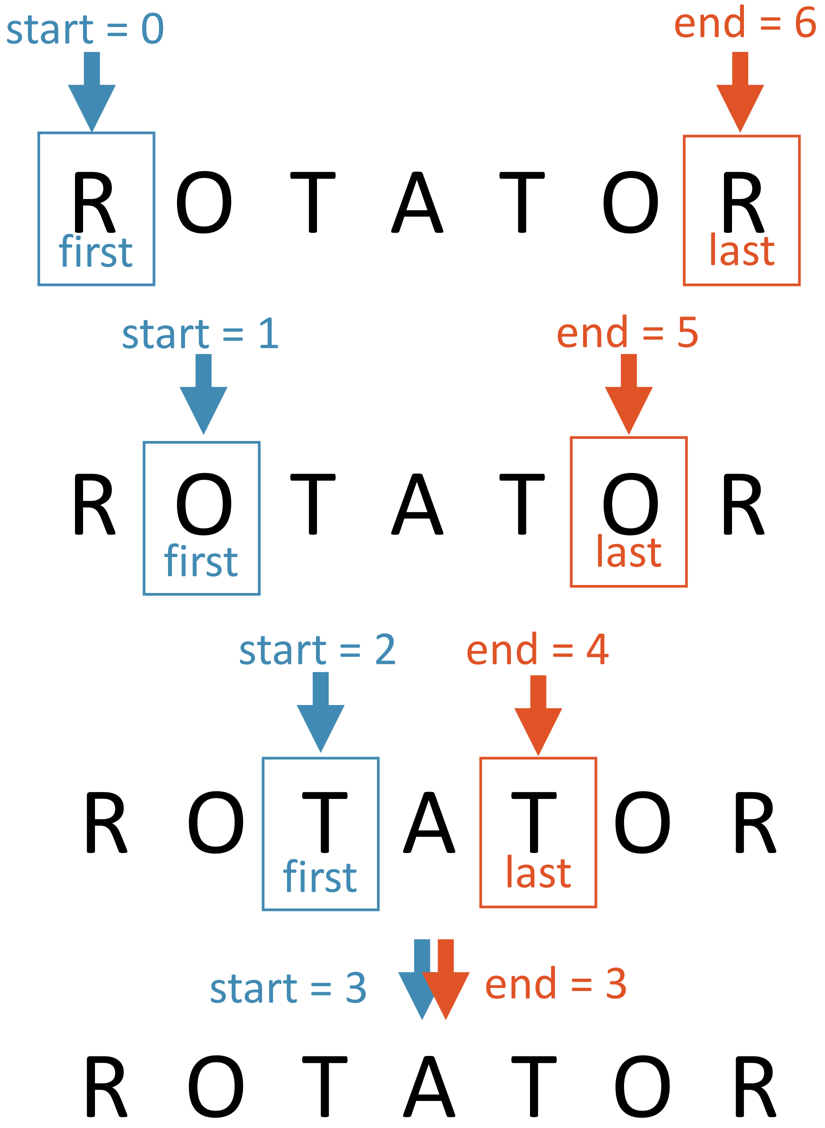
When your friend wants to invite others, then your conversation will be on hold.

<u>Until</u> the last person decides to go and ends the call (no more asking), everyone returns to their previous conversation one by one in sequence

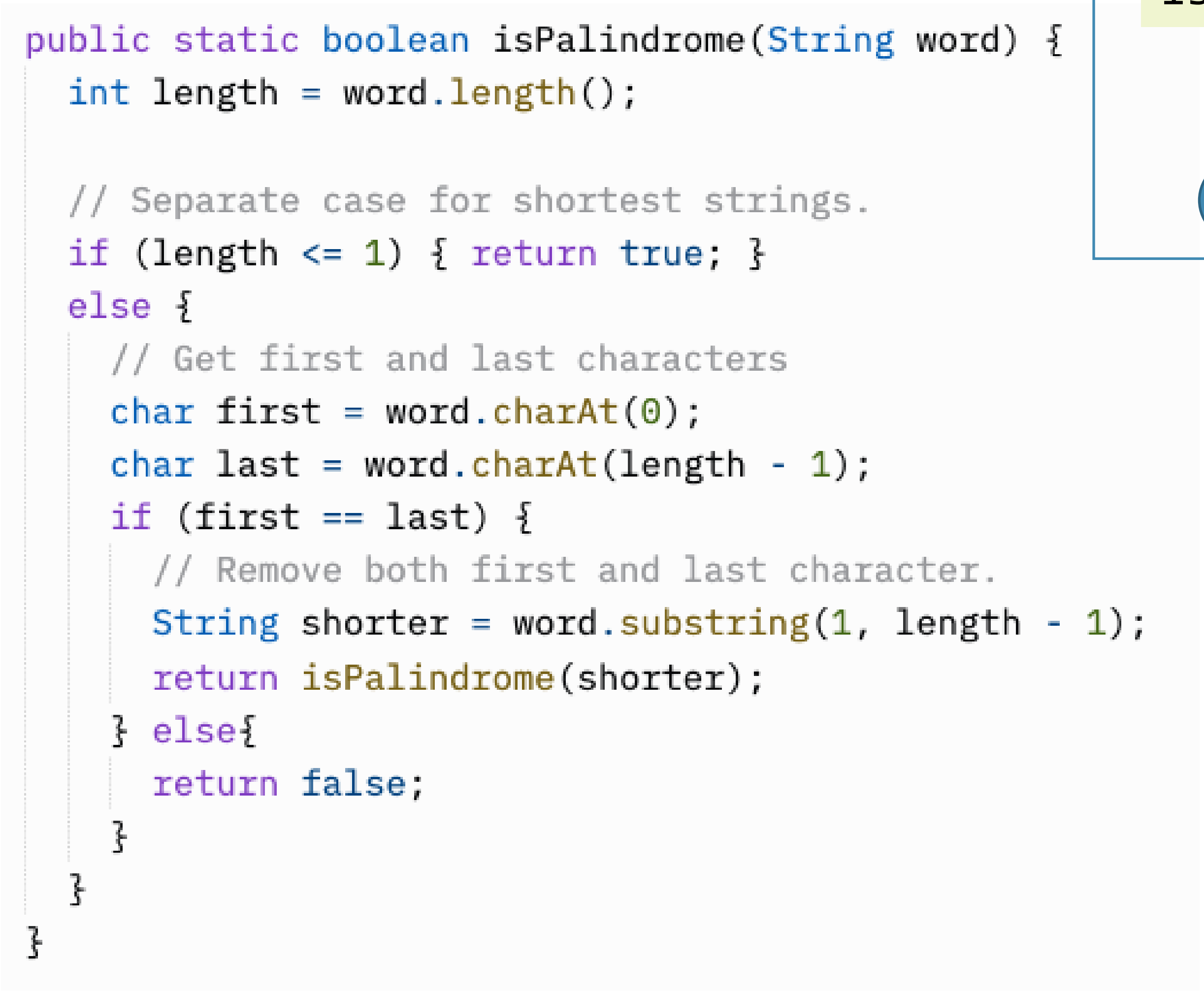
Review Loop vs Recursion Algorithm: Check Palindrome



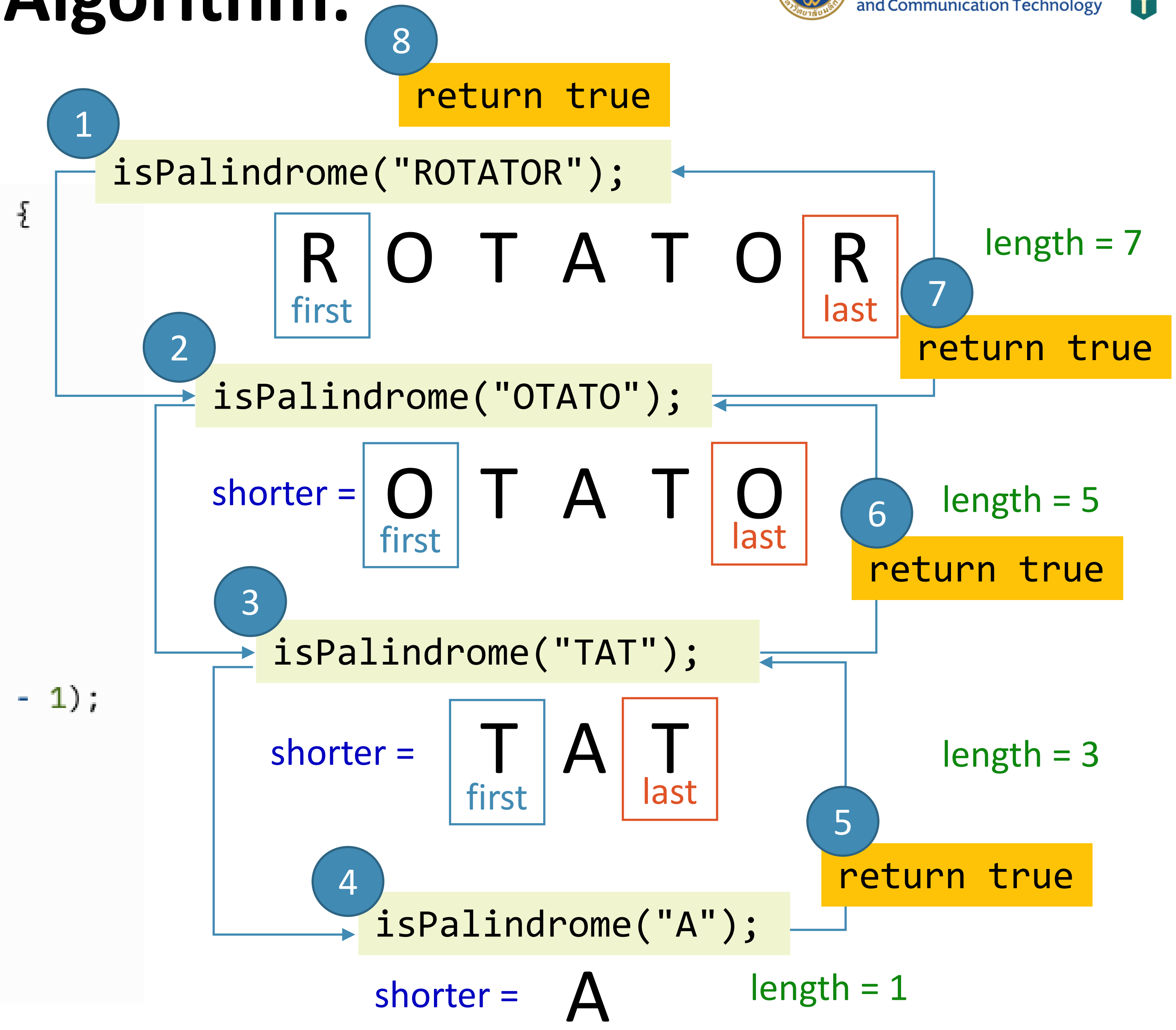
```
public static boolean isPalindromeLoop(String word){
  int start = 0;
  int end = word.length() - 1;
  while(start < end){
    // Get first and last characters
    char first = word.charAt(start);
    char last = word.charAt(end);

    // Both match, keep continute to the next pair
    if (first == last){
      start++;
      end--;
    } else {
      return false;
    }
  }
  return true;
}
```



Review Loop vs Recursion Algorithm: Check Palindrome



```
public static boolean isPalindrome(String word) {
  int length = word.length();

  // Separate case for shortest strings.
  if (length <= 1) { return true; }
  else {
    // Get first and last characters
    char first = word.charAt(0);
    char last = word.charAt(length - 1);
    if (first == last) {
      // Remove both first and last character.
      String shorter = word.substring(1, length - 1);
      return isPalindrome(shorter);
    } else{
      return false;
    }
  }
}
```



8 return true

1 isPalindrome("ROTATOR");

R O T A T O R

first

last

length = 7

7 return true

2 isPalindrome("OTATO");

shorter = O T A T O

first

last

6 length = 5

return true

3 isPalindrome("TAT");

shorter = T A T

first

last

length = 3

5 return true

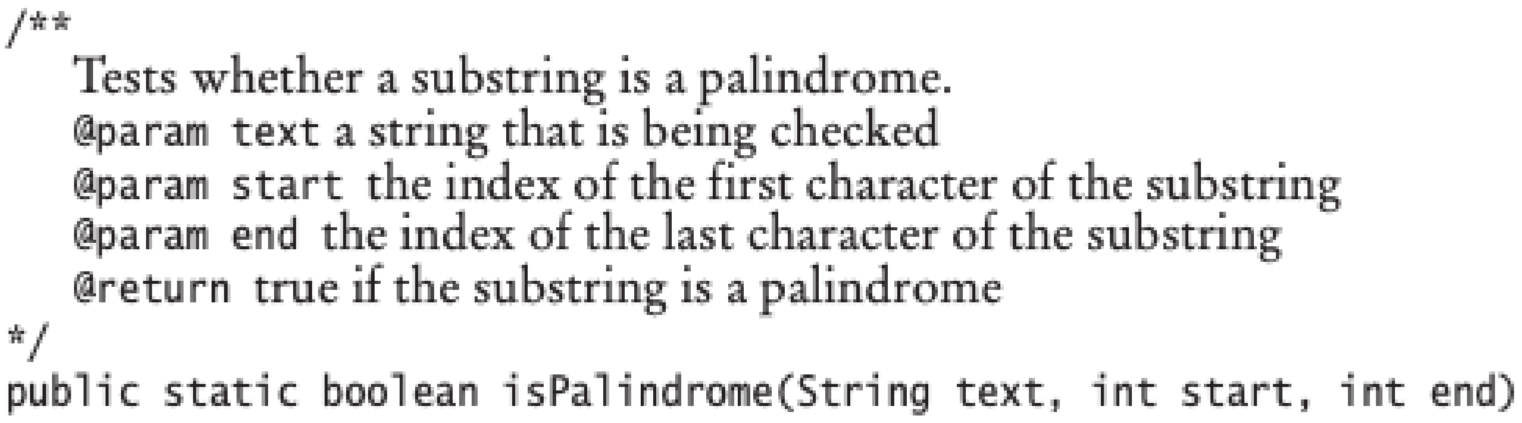
4 isPalindrome("A");

shorter = A

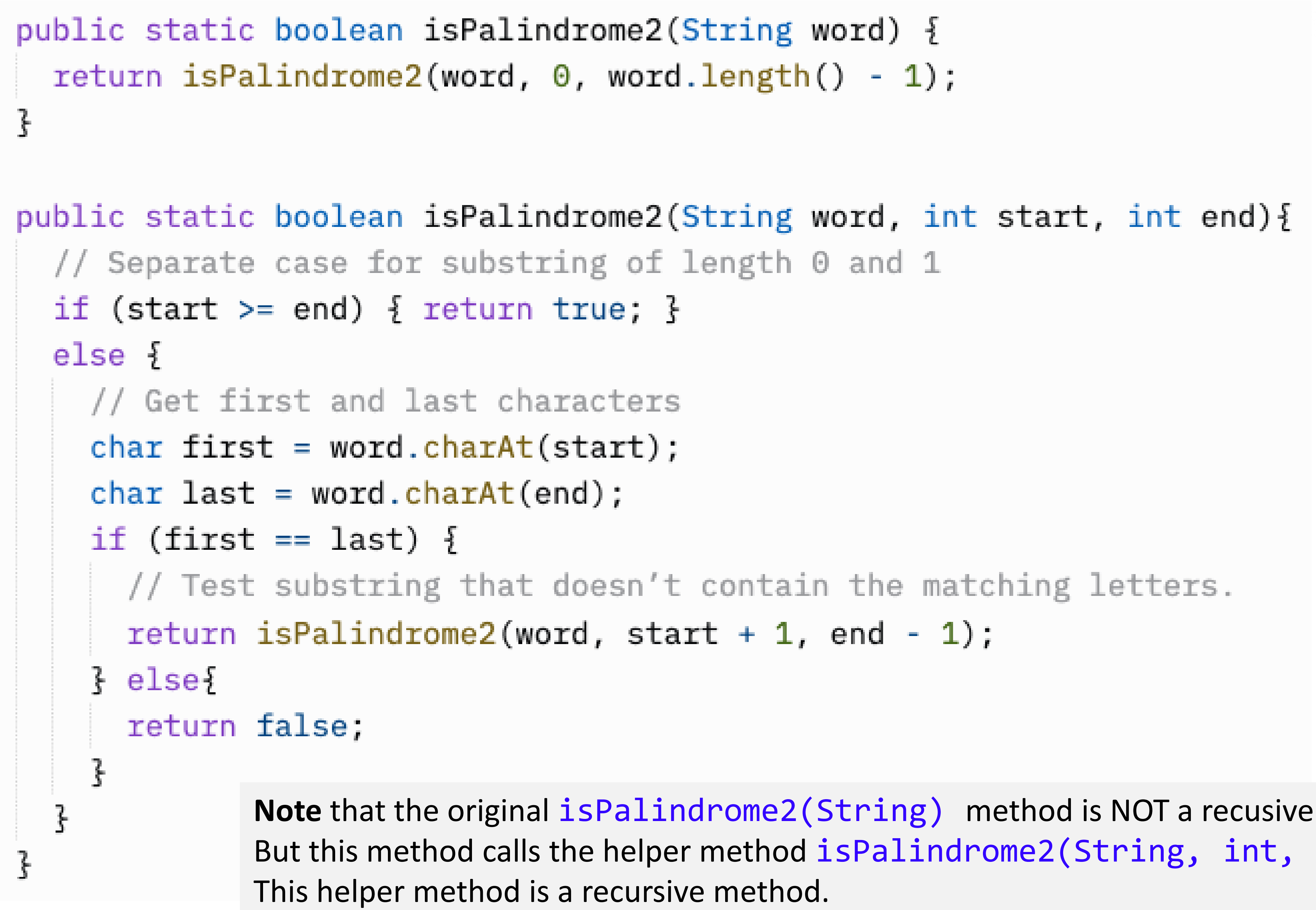
length = 1

Recursive Helper Methods

- To find recursive solution, sometimes it is easier to slightly change the original problem. Then let the origial problem call a **recursive helper methods**.
- For palindrome problem, it is inefficient to construct a new string in every step. We can change a little bit to check whether a substring is a palindrome.



```
/**
   Tests whether a substring is a palindrome.
   @param text a string that is being checked
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public static boolean isPalindrome(String text, int start, int end)
```



```java
public static boolean isPalindrome2(String word) {
  return isPalindrome2(word, 0, word.length() - 1);
}

public static boolean isPalindrome2(String word, int start, int end){
  // Separate case for substring of length 0 and 1
  if (start >= end) { return true; }
  else {
    // Get first and last characters
    char first = word.charAt(start);
    char last = word.charAt(end);
    if (first == last) {
      // Test substring that doesn’t contain the matching letters.
      return isPalindrome2(word, start + 1, end - 1);
    } else{
      return false;
    }
  }
}
```

Note that the original `isPalindrome2(String)` method is NOT a recusive method. But this method calls the helper method `isPalindrome2(String, int, int)`. This helper method is a recursive method.

Review: Thinking Recursively

- **Problem Statement: Test whether a given word is a palindrome**
- **Step 1:** Consider various ways to simplify inputs
  - Remove both the first and the last characters
- **Step 2:** Combine solutions with simpler inputs into a solution of the original problem
  - The word is palindrome if and only if **the first and the last letter match** AND **the shorter word obtained by removing the first and the last letter is a palindrome**.
- **Step 3:** Find the solutions to the *simplest* inputs
  - A recursive computation keeps simplifying its inputs. At the end, it arrives very simple inputs and recursion comes to a stop! -> **All string of length 0 or 1 are palindromes.**
- **Step 4:** Implement the solution by combining the simple cases and the reduction step.
  - Make a separate cases for the simplest inputs in Step 3.
  - If the input is not one of the simplest cases, then implement the logic we setup in Step 2.

Review: Thinking Recursively

- **Problem Statement: Find Max value in Binary Tree**
- **Step 1:** Consider various ways to simplify inputs
  - Find max value of left subtree and right subtree
- **Step 2:** Combine solutions with simpler inputs into a solution of the original problem
  - Get the **max value from left subtree**, **max value from the right subtree**, and compare with the **value at this node** to find the *maximum value amoung these three values*.
- **Step 3:** Find the solutions to the *simplest* inputs
  - A recursive computation keeps simplifying its inputs. At the end, **the input itself is null**, *the recusion stops and returns -1*
  - Or i**t arrives at the leaft node (no more left subtree nor right subtree) and recursion comes to a stop!** *The maximum value is the value of leaf node itself*
- **Step 4:** Implement the solution by combining the simple cases and the reduction step.
  - Make a separate cases for the simplest inputs in Step 3.
  - If the input is not one of the simplest cases, then implement the logic we setup in Step 2.

Object Oriented Design (OOD)

Object-Oriented Development (OOD)

OOP Development Activities

OOP Development Activities

1. Identifying Classes and Objects
2. Identifying Variables and Methods
3. Identifying Class Relationships
4. ~~Interfaces~~
5. ~~Enumerated Types Revisited~~
6. ~~Method Design~~
7. ~~Testing~~
8. ~~GUI Design and Layout~~

1. Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

2. Identifying Variables and Methods

- Part of identifying the classes we need is the process of *assigning characteristics (variables) and responsibilities* (Method) to each class.
- Every *activity* that a program must accomplish must be represented by *one or more variables+methods* in one or more classes
- We generally use nouns for variables and *verbs* for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with *primary responsibilities* and evolve the design.

"Perfection is the enemy of {progress, productiveness, good, etc.}" - **Many people**

"Good enough is better than perfect" - **Gretchen Rubin**

2. Identifying Variables and Methods

Describe Behavior (Method)

- The set of methods also dictate *how your objects interact* with each other to produce a solution.
- *Sequence diagrams is a tool that* can help tracing object methods and interactions.

3. Identifying Class Relationships

Class Relationships

- Classes in a software system can have various types of relationships to each other
- To Design a Software the **UML Diagram** is used to represent Class Relationships

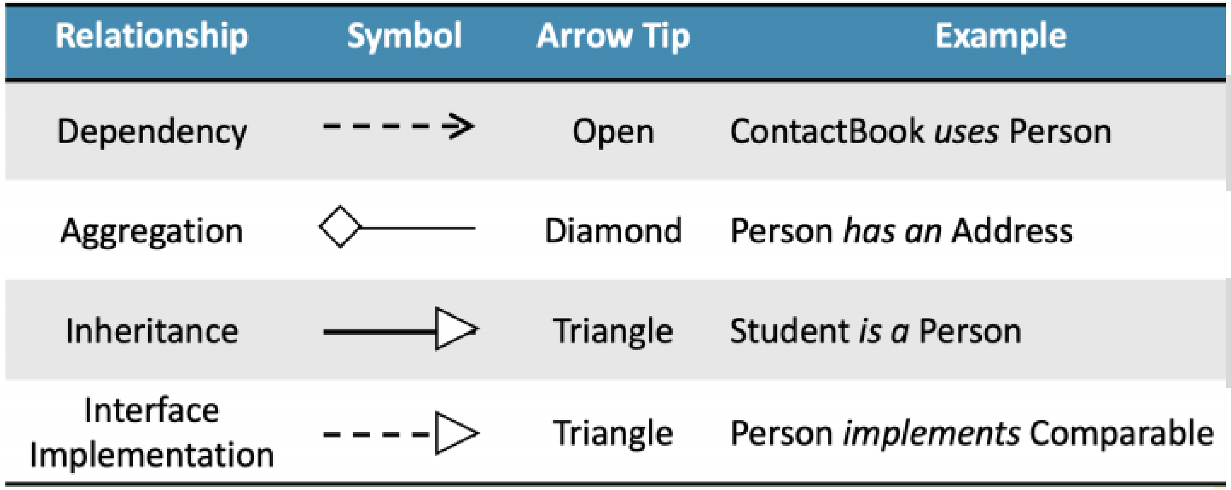
3. Identifying Class Relationships

- **UML Diagram is a picture of**
  - The Class in OOP system
  - *Fields* and *Methods*
  - Relationship between Classes

3. Identifying Class Relationships

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Four of the most common relationships:

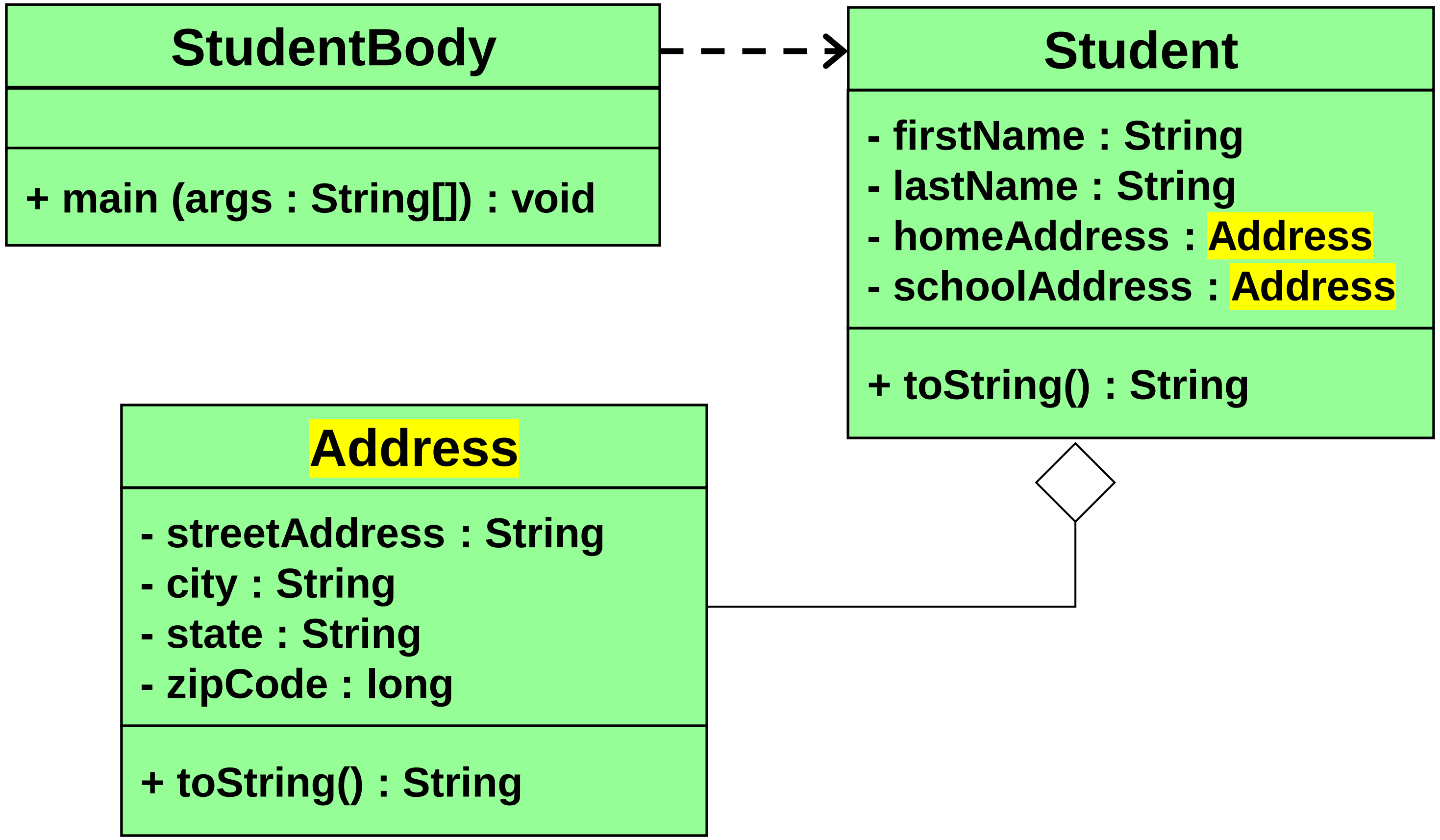


Relationship	Symbol	Arrow Tip	Example
Dependency	- - - - ->	Open	ContactBook *uses* Person
Aggregation	◇———	Diamond	Person *has an* Address
Inheritance	———▷	Triangle	Student *is a* Person
Interface Implementation	- - - -▷	Triangle	Person *implements* Comparable

- Let's discuss *dependency* and *aggregation* further

3. Identifying Class Relationships

- Aggregation in UML



StudentBody

+ main (args : String[]) : void

Student

- firstName : String
- lastName : String
- homeAddress : Address
- schoolAddress : Address

+ toString() : String

Address

- streetAddress : String
- city : String
- state : String
- zipCode : long

+ toString() : String

Thank you for your efforts in OOP

~ Good Luck ~