CSC207 Week 3

Larry Zhang

Announcements

- Readings will be posted before the lecture
- Lab 1 marks available in your repo
    - 1 point for creating the correct project.
    - 1 point for creating the correct classes.
    - 1 point for creating the correct methods.
    - If there is a notes.txt file saying that you need to use branches for Git, ignore it for now.
- Sololearn: an online learning-game for Java (as well as other languages)
    - https://www.sololearn.com/Play/Java#

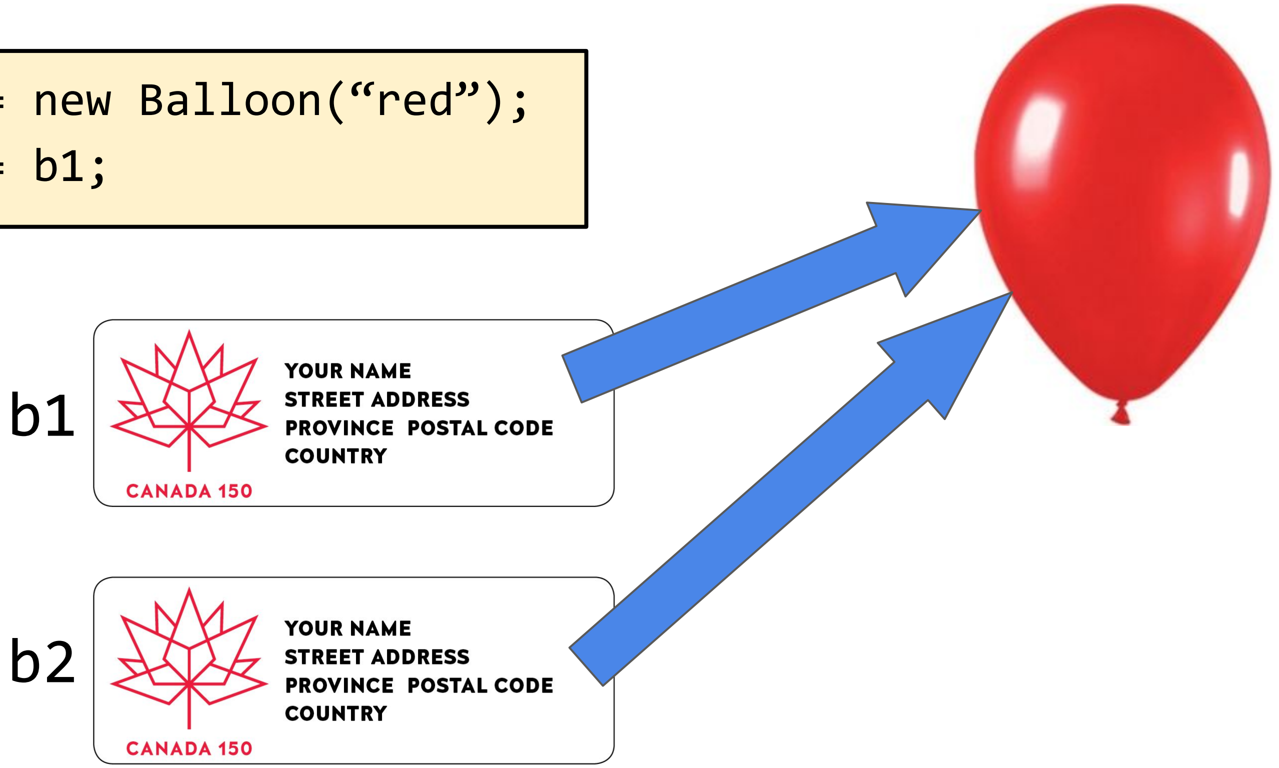
Today’s Outline

- Unit Test: JUnit
- Javadoc
- More OO programming: Inheritance
- UML

Recap + A Couple More Tips

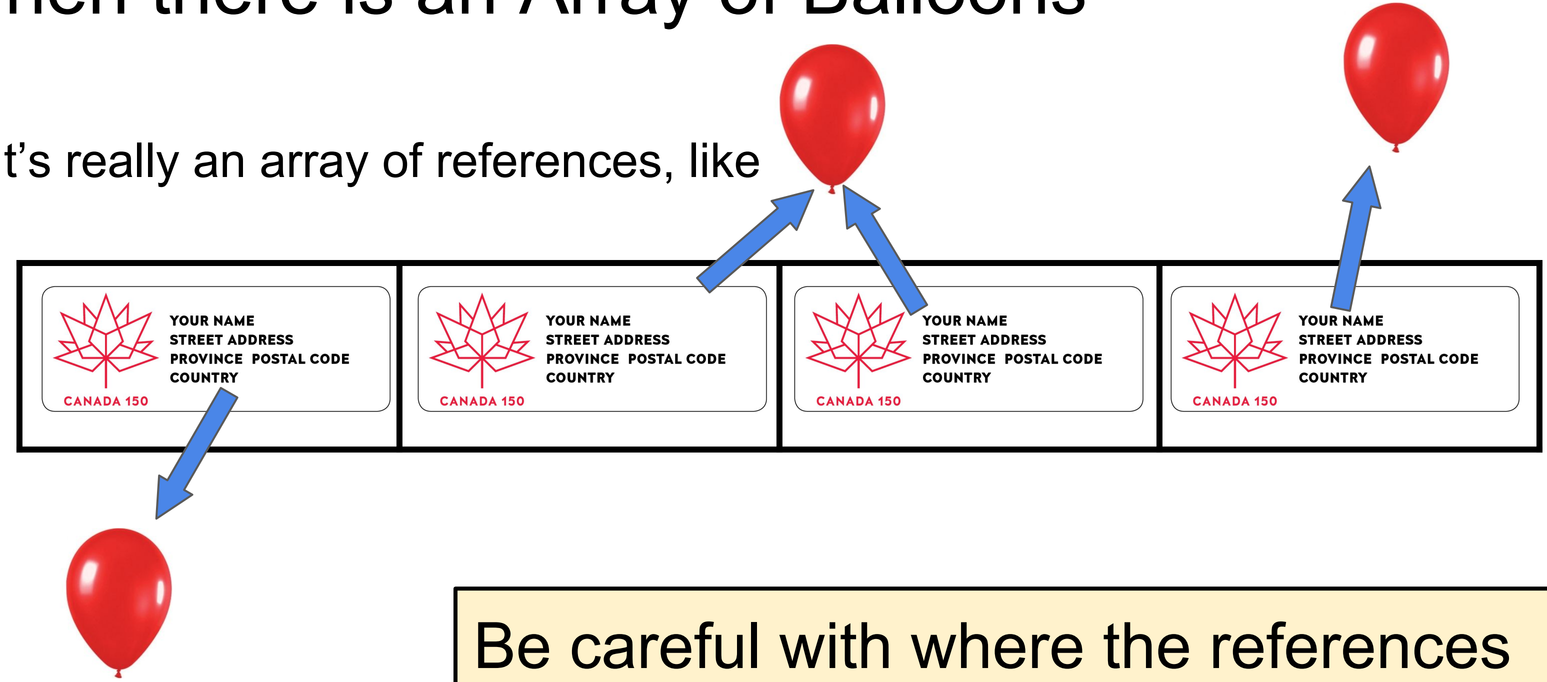
Recap: Reference to an Object is like **Address Label**

```
Balloon b1 = new Balloon(“red”);
Balloon b2 = b1;
```



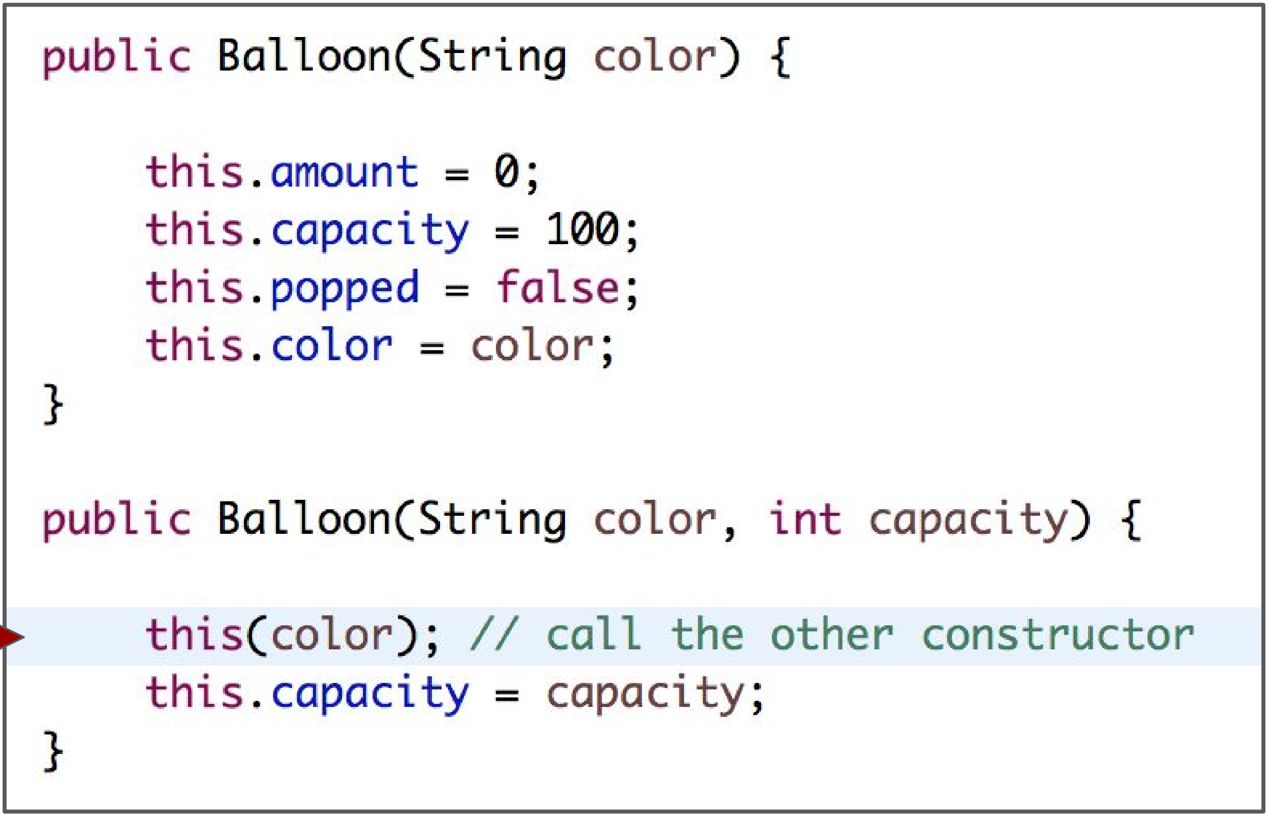
When there is an Array of Balloons

It’s really an array of references, like



Be careful with where the references are pointing to.

Call one constructor from another constructor



```java
public Balloon(String color) {

    this.amount = 0;
    this.capacity = 100;
    this.popped = false;
    this.color = color;
}

public Balloon(String color, int capacity) {

    this(color); // call the other constructor
    this.capacity = capacity;
}
```

keyword static used on variables

- It is a variable which belongs to the **class** and **not to object** (instance)
- Static variables are initialized **only once** , at the start of the execution.
- A **single copy** to be shared by all instances of the class

```
public class Person {
    public static int nPeople = 0; // doesn’t have to be public
    // Constructor
    public Person() {
        nPeople++; // keeps track of # of Person’s constructed
        System.out.println(nPeople + “ objects created”);
    }
}
//… outside the class, static variables can be accessed like
System.out.println(Person.nPeople);
```

Avoid using static variables unless necessary. It is generally bad practice in OO design because it create dependencies between objects and violate encapsulation. Read more: https://stackoverflow.com/questions/7026507/why-are-static-variables-considered-evil

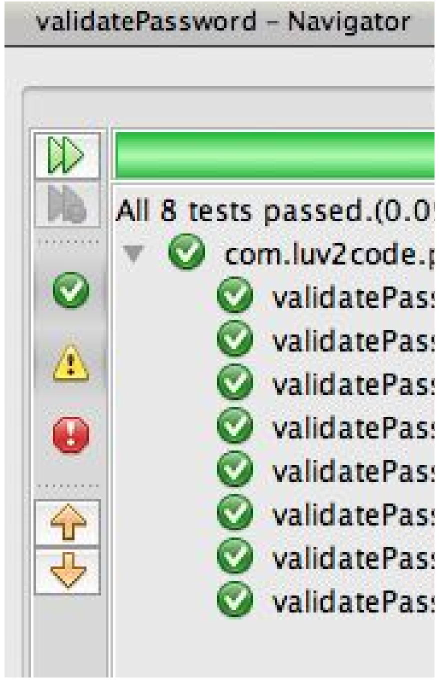
NEXT TOPIC

Unit Testing with JUnit

Why writing a unit testing suite?

- Misconception: It takes too much extra time to write the test suite!
  - Wrong! Empirical research repeatedly show that test suites reduce debugging time more than the amount spent building the test suite.
- Unit Tests allows you to make big changes to code quickly: make the change, then click on “test”. If all green, then you know this module is all good.
- Unit Tests help you understand the **design** of the code you are working on.
  - It forces you to think through the possible inputs and outputs of your code.
  - Test cases should be written even before the implementation.
- Unit test is not really about finding bugs, it’s about designing software components robustly.

Running unit test is very satisfying



validatePassword - Navigator

All 8 tests passed.(0.0

com.luv2code.

validatePas

validatePas

validatePas

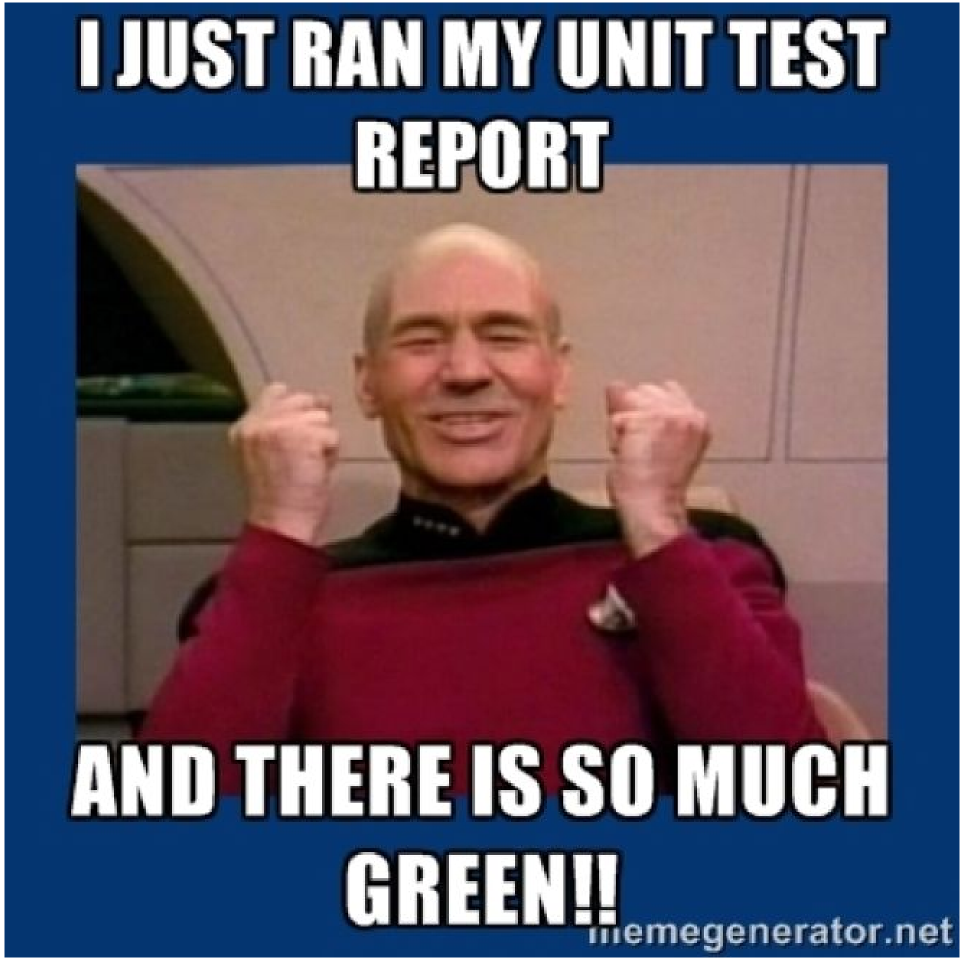
validatePas

validatePas

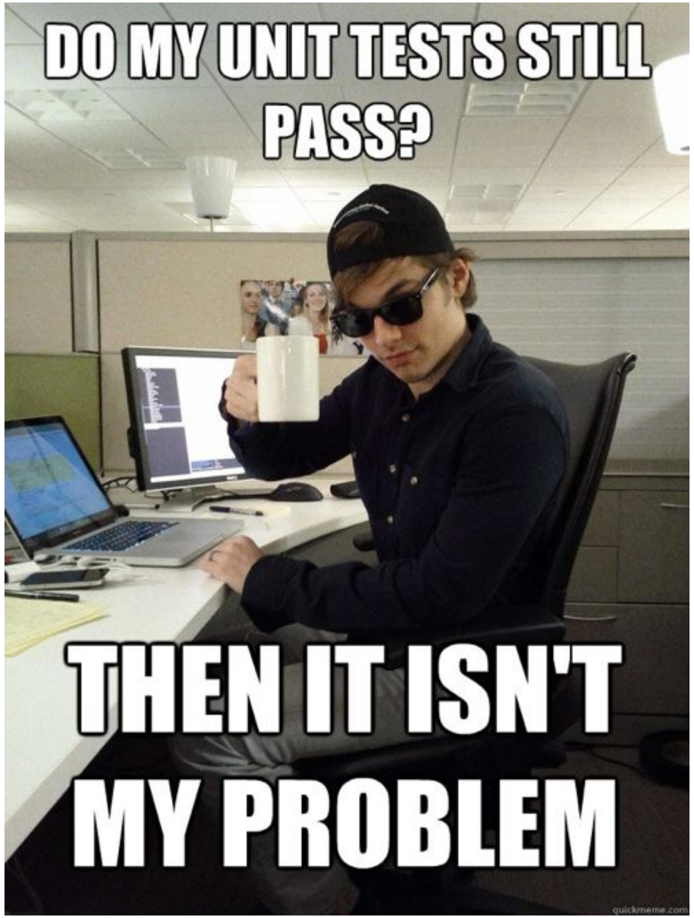
validatePas

validatePas

validatePas



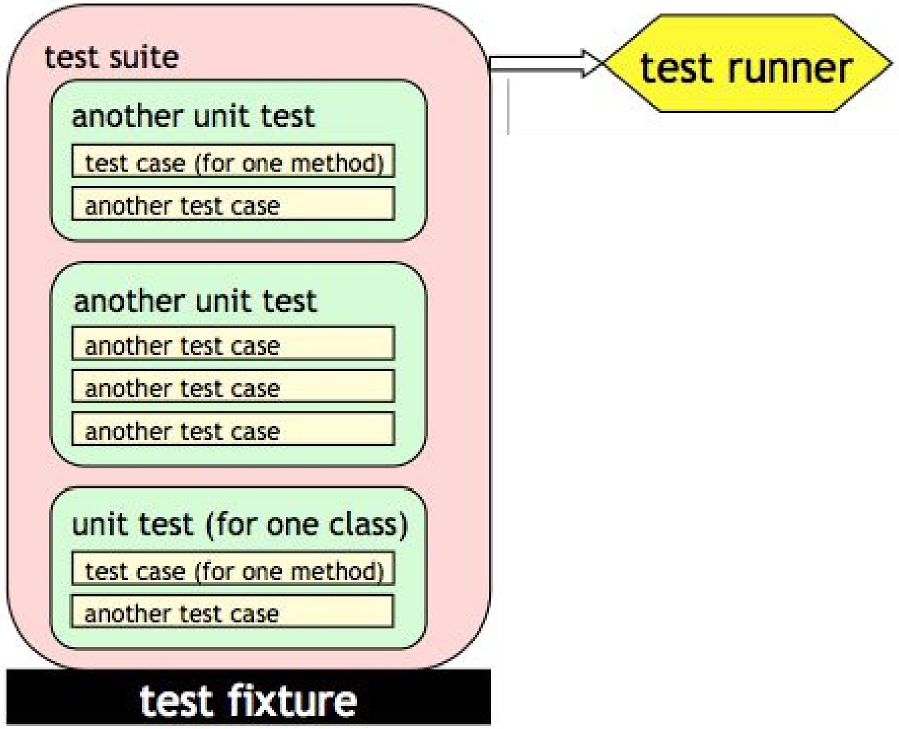
It helps efficiently locate bugs in the software.



JUnit

- JUnit is a framework for writing tests in Java.
- JUnit was written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of Extreme Programming methodology)

JUnit structure



Execution flow

Each method in a Unit Test can be labelled with @Before, @Test, @After

Each Unit Test has many test cases, and for each test case:

- @Before: do some setup/prep work before running the tests, e.g., open a file, open a network connection.
- @Test: run the test case.
- @After: clean up after the tests finish, e.g., closing the file, close the connection.

It’s BTA, BTA, BTA, … , not BTTTTTTTA

DEMO

`w02/src/BalloonTest.java`

Other Tips for Unit Testing

Keep it SIMPLE

Keep it REAL

A UNIT TEST FAILS AFTER A CODE CHANGE
REMOVE THE FAILING TEST
memegenerator.net

Read more on JUnit

http://junit.org/

NEXT TOPIC

Javadoc

a tool for automatically generating documentation from well-commented source code

DEMO

Generate Javadoc in Eclipse: `w02/doc/`

How to write comments for Javadoc

http://www.oracle.com/technetwork/articles/java/index-137868.html

NEXT TOPIC

OO Continued: Inheritance

Basic Syntax for Inheritance

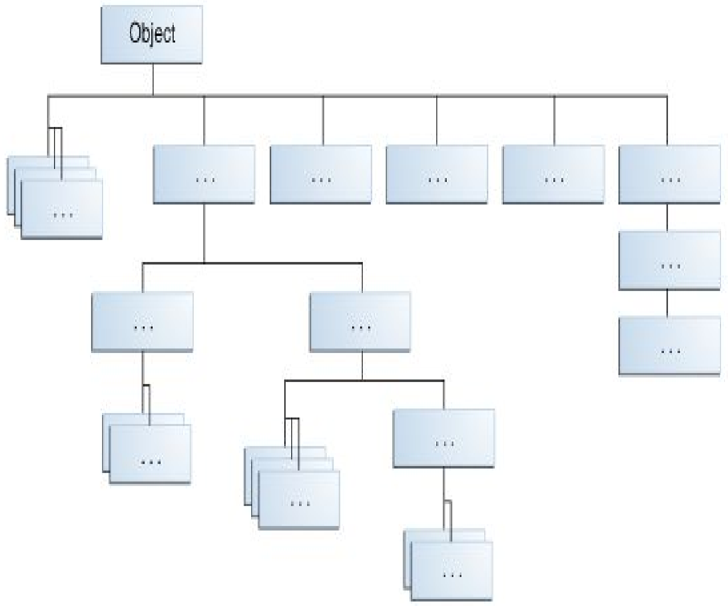
Child Class **IS-A** (subtype of) Parent Class.

E.g., Cat is a subtype of Animal.

```
public class Cat extends Animal {
    ...
}
```

All Classes in Java form a Tree

- All classes form a tree called the **inheritance hierarchy**, with Class **`Object`** at the root.
- Class **`Object`** does not have a parent.
- All other Java classes have one parent. If a class has no parent declared (not extending any class), it is a child of class **`Object`**.
- A parent class can have multiple child classes.



Access Parent class from Child

- Child can access Parent’s variable and method **iff** they are **public** or **protected**.
  - CANNOT access those declared as **private** or **default** in the parent class.
- In a subclass, **super** refers to the parent class.
  - super.variable
  - super.method()
  - super(arguments): parent’s constructor

Constructor in Child class

- First thing to do: call parent’s constructor by **super(args)**.
    - if don’t call, parent’s default constructor with no args will be called, i.e., **super()**.

Overriding and Shadowing

- **Overriding**:
  - a Child class can re-implement a **method** that exists in Parent.
  - child_object.method() will called the re-implemented one in Child.
  - This is often-used technique.
  - If you don’t want a method to be overridden by any child, declare it as **final**.
- **Shadowing**:
  - a Child class re-declares a **variable** that exists in Parent.
  - The parent’s variable gets shadowed by the child’s variable
  - This is bad and confusing, and should almost never be used.

Dynamic Binding

```
class Person {
   …
   public String hello() {
      return “I am a person”;
   }
}


class Student extends Person {
   …
   @Override
   public String hello() {
      return “I am a student”;
   }
}

Student s1 = new Student(“Alice”, 22);
Person p1 = s1;
String x = p1.hello();
// Which hello() will be called?
```

- **p1** is a reference that refers to Student **s1** as a Person.
- Java remembers that **s1** is **created** as a Student.
- When calling **p1.hello()**, Java knows that what **p1** is referring to is a Student, and it will invoke the method **hello()** that is implemented in the **Student** class.
- So **p1.hello()** returns “I am a student”.
- Read more about dynamic binding here: https://stackoverflow.com/questions/19017258/static-vs-dynamic-binding-in-java

DEMO

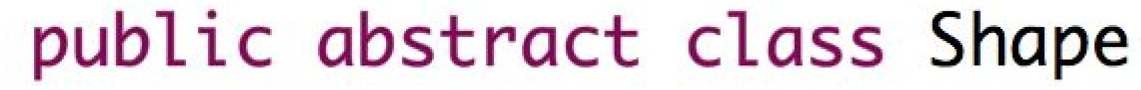
```
w03/src/people/
Person.java
StudentWithoutInheritance.java
Student.java
Doctor.java
PlayWithPeople.java
```

Abstract Class

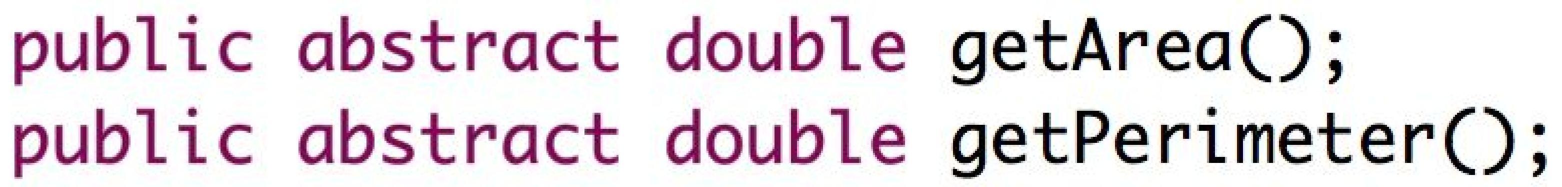
An abstract class:

- contains **abstract** methods that are NOT implemented
- may also contain implemented methods
- may contain variables
- CANNOT be **instantiated**, because it is not “concrete”.
- A class can **extend (inherit from)** an abstract class and implement the abstract methods, then CAN be instantiated.
- A child class of an abstract class can still be abstract.

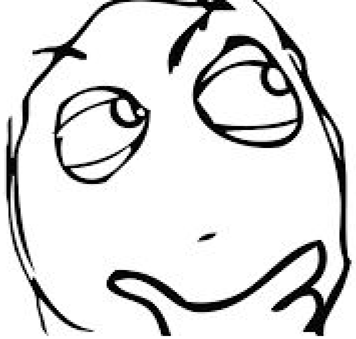
Example



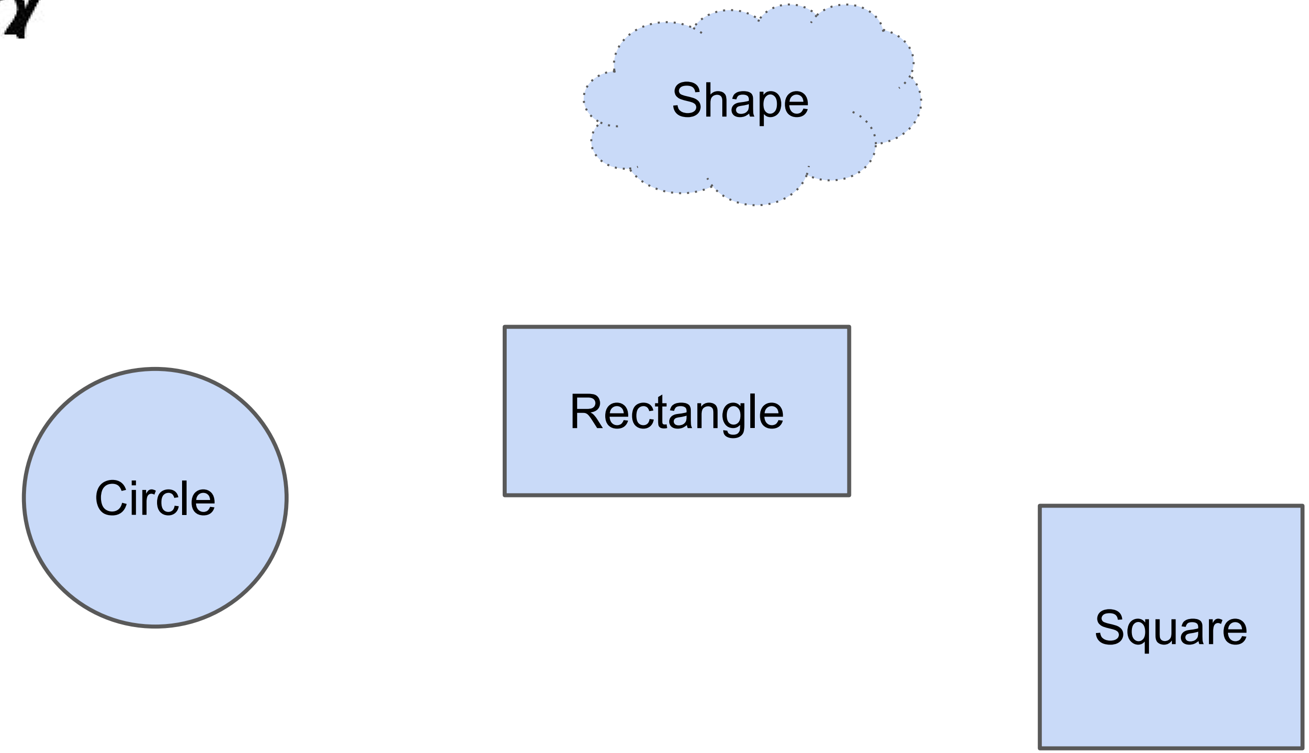
```java
public abstract class Shape
```



```java
    public abstract double getArea();
    public abstract double getPerimeter();
```



What should be the parent/child relationships between these class?



DEMO

```
w03/src/shapes/
Shape.java
Circle.java
Rectangle.java
Square.java
PlayWithShapes.java
```

Liskov Substitution Principle (LSP)

Functions that use references to parent classes must be able to use objects of child classes without knowing it.

In other words, methods in the parent class must make total sense for the child class.

If **Square** is a child of **Rectangle**, then

- **Rectangle**’s **setWidth()** and **setHeight()** methods do NOT make sense for **Square**.
- Violating LSP
- Square should directly inherit from Shape.

Read more: https://stackoverflow.com/questions/56860/what-is-an-example-of-the-liskov-substitution-principle

NEXT TOPIC

UML

UML

- Unified Modeling Language (UML) allows us to express the design of a program before writing any code.
- It is language-independent.
- An extremely expressive language.
- We'll use only a small part of the language, Class Diagrams, to represent basic OO design.

Notation

Data members:

```
name: type
```

Methods:

```
methodName(param1: type1, param2: type2,... ): returnType
```

Visibility:

– private

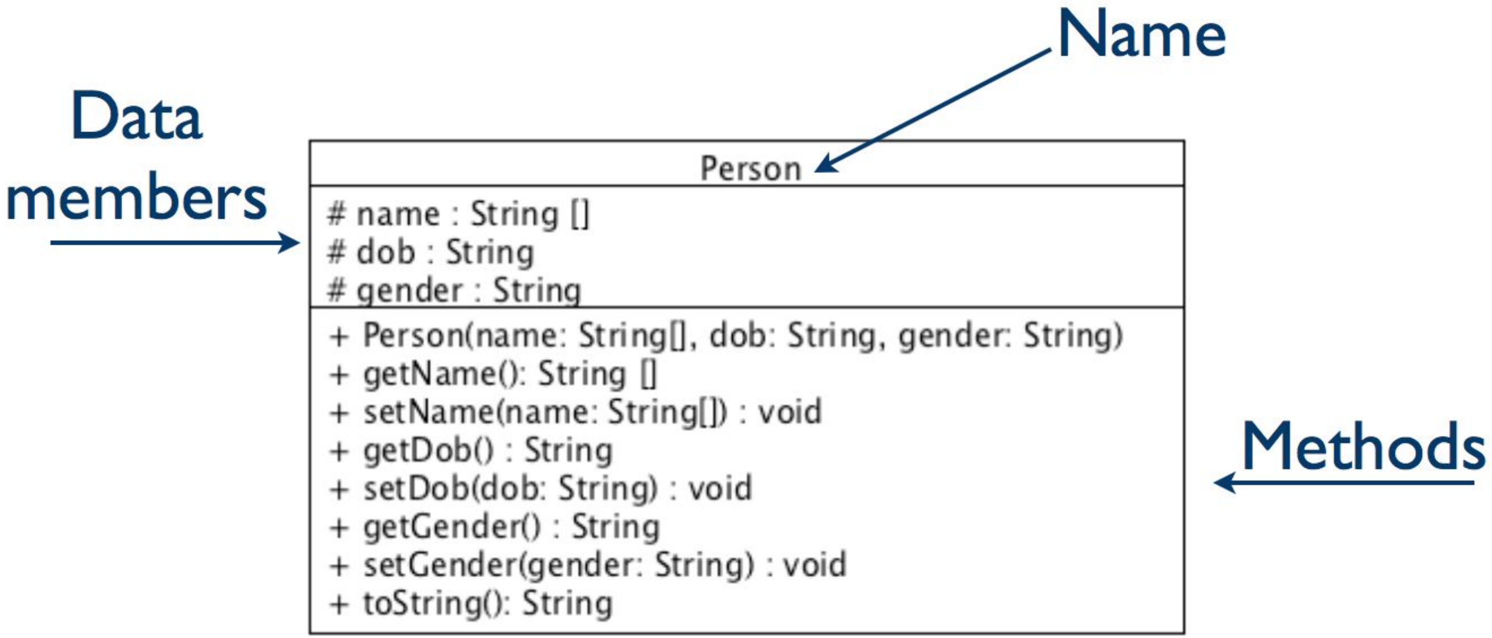
+ public

protected

~ package

Static: <u>underline</u>

Example: Class Person



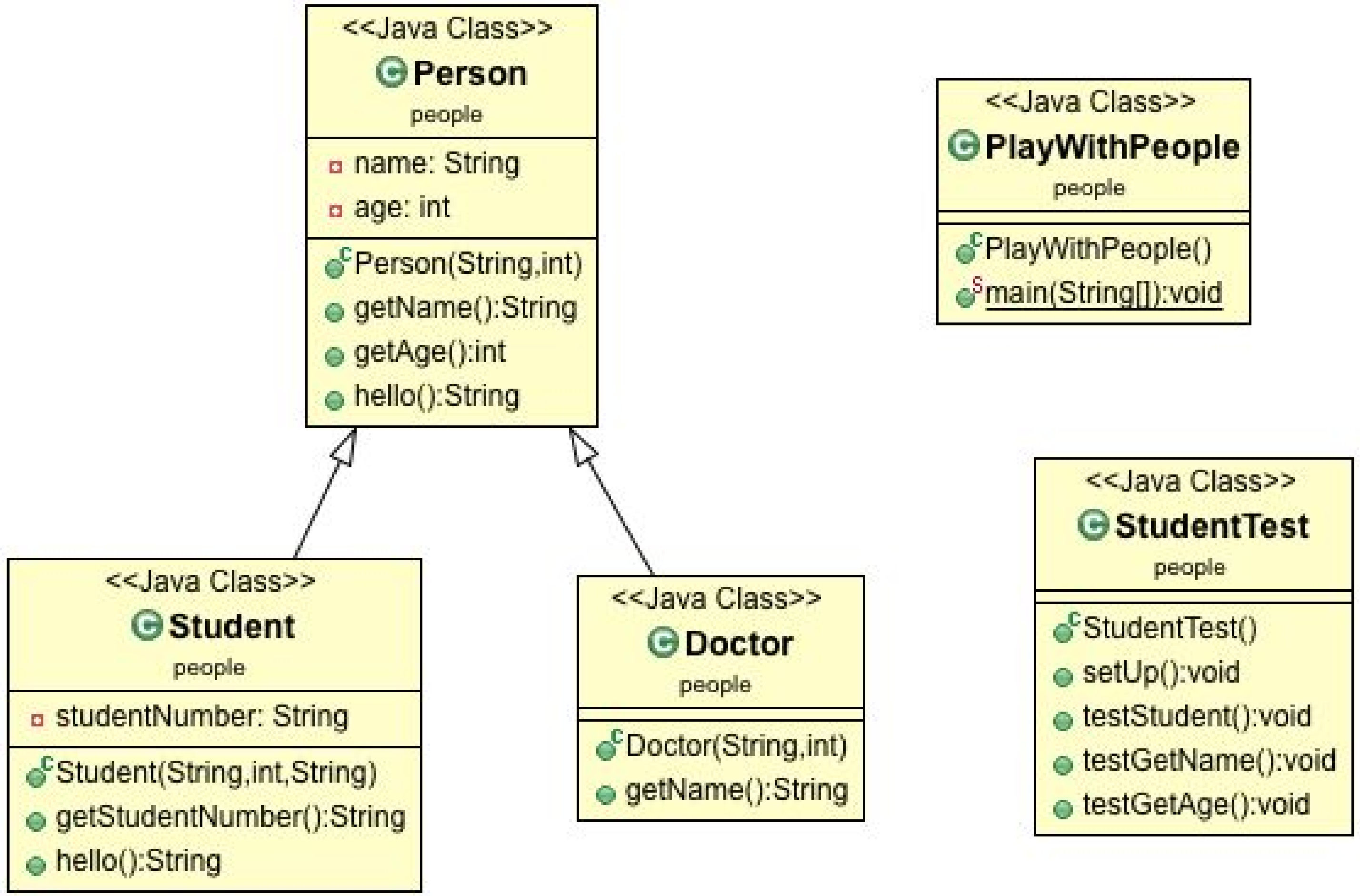
```
Person
-------------------------------------------------
# name : String []
# dob : String
# gender : String
-------------------------------------------------
+ Person(name: String[], dob: String, gender: String)
+ getName(): String []
+ setName(name: String[]) : void
+ getDob() : String
+ setDob(dob: String) : void
+ getGender() : String
+ setGender(gender: String) : void
+ toString(): String
```

DEMO

UML Visualization using ObjectAid

http://www.objectaid.com/installation

and many other UML tools are available for Eclipse.



<<Java Class>>
Person
people
name: String
age: int
Person(String,int)
getName():String
getAge():int
hello():String
<<Java Class>>
Student
people
studentNumber: String
Student(String,int,String)
getStudentNumber():String
hello():String
<<Java Class>>
Doctor
people
Doctor(String,int)
getName():String
<<Java Class>>
PlayWithPeople
people
PlayWithPeople()
main(String[]):void
<<Java Class>>
StudentTest
people
StudentTest()
setUp():void
testStudent():void
testGetName():void
testGetAge():void

References

- JUnit
  - http://junit.org/
- Javadoc
  - http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
- Inheritance
  - http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Arnold’s notes on OO programming
  - http://www.cs.toronto.edu/~arnold/cit/summer2007/ooProgramming.html