Object-Oriented Programming

CSCI-UA 0470-001
Class 2
Instructor: Randy Shepherd

Java and the JVM

What is Java?

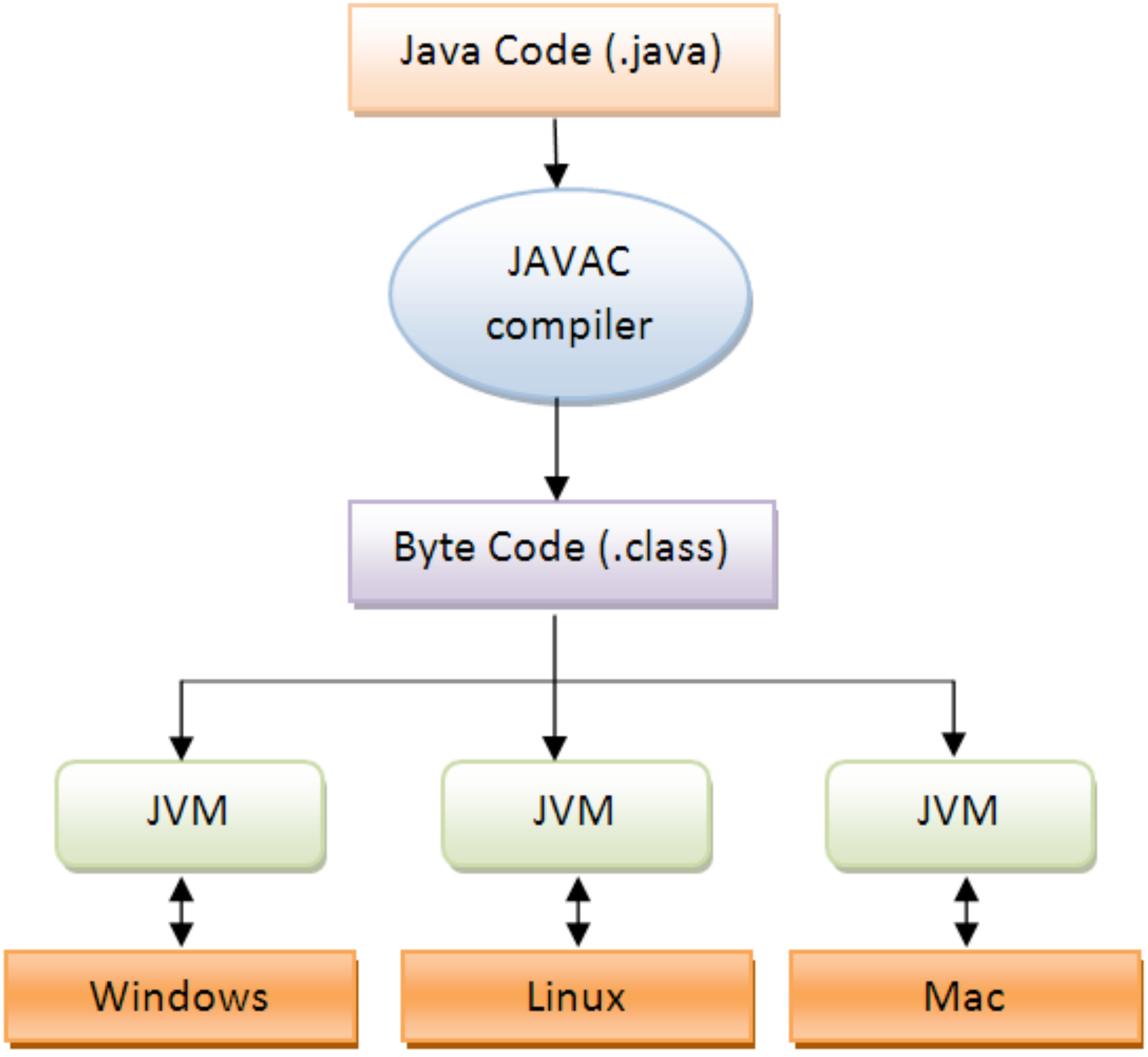
- Programming language
- Standard libraries
- Tools. Compiler, runtime, disassembler…. many others
    - JVM is the runtime. Very sophisticated.

JVM

- In the olden days people would write code in assembly.
- That was painful, programming languages, such as C, evolved to alleviate that pain
- Compilers for a given language, such as C, emit assembly.
- Assembly is specific to hardware platform. Different compilers for each architecture.
- More pain, same code may be compatible with some compilers and not others (or worse).
- The JVM consumes "byte code" and emits platform-correct assembly. Compatibility no longer the programmers problem.

Byte code

- So where does "byte code" come from? The java compiler.
- Java code progression:
  - .java -> javac -> .class -> "byte code" -> Jvm
- .java is the source files in java
- .class is the java byte code
- The JVM does this just in time, which is a combination of interpreter and ahead-of-time compilation.


Java Code (.java)
JAVAC
compiler
Byte Code (.class)
JVM
JVM
JVM
Windows
Linux
Mac

A word about ‘Linking’

- A *dynamic linker* is the part of a system that loads shared libraries needed by an executable when it is executed (at "run time”)

- This is in contrast to a *static linker*. A static linked library is a set of routines, external functions and variables which are resolved at compile-time and copied into a target application by a compiler.

- Dynamic linking reduces resource consumption and is more modular

- Static linking is maybe faster and maybe makes application distribution easier

Classpaths and Classloaders

- Java is dynamically linked.
- The Java classloader loads Java classes into the jvm.
- It does this based on whats in the byte code. This is true of your own code or library code.
- This does not occur until the class is actually used by the program.
- When it has to find a class, it looks on the classpath.
- The classpath is something you can configure when you invoke the jvm and compiler.

OOP

(Very) Basics of OOP

- is a way to organize software as a collection of modules that consist of *data and behavior*
- is a programming approach based on objects and classes

Classes

- Classes have the following characteristics
    - *State* - (or data) are the values that the object has.
    - *Methods* - (or behavior) are the ways in which the object can interact with its data, the actions.

Classes

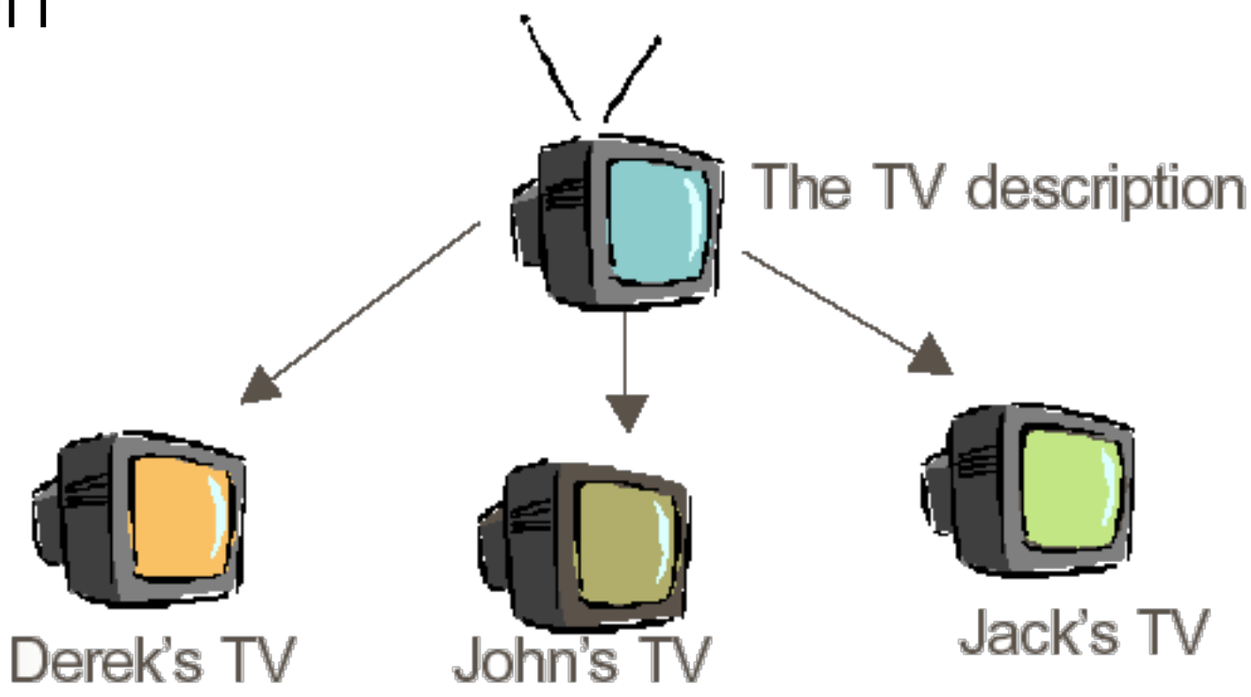
- If we think of a real-world object, such as a television, it will have several features and properties:
    - We do not have to open the case to use it.
    - We have some controls to use it (buttons on the box, or a remote control).
    - We understand the concept of a television without necessarily understanding how it is built and functions.
    - It is complete when we purchase it, with any external requirements well documented.
- Very much how you might describe a class!

Classes

- Similarly, a class should…
  - Provide a well-defined interface
  - Represent a clear concept
  - Be complete and well-documented
  - The code should be robust - and should be tested!

Objects

- An object is an instance of a class.
- You could think of a class as the description of a concept, and an object as the realization of this description



Why OOP?

- allows us to organize our code in a way that reduces complexity

- allows us to isolate *responsibility to a single entity.* "Do one thing and do it well"

- allows us to have code that is reusable so that you "Don't repeat yourself"

- gives us a way to model the primitives in our system.

Principles of OOP

- As we've discussed, object-oriented programming is a programming methodology with supporting primitives (classes, objects and more)

- In practice, a set of design principles and language features

- The structures can change somewhat from language to language, but the principles remain the same.

  …so what are the principles?

The “Pillars” of OOP

- Encapsulation & Abstraction
- Inheritance (or Specialization)
- Polymorphism

Encapsulation & Abstraction

- Encapsulation is the process of putting data and the operations that can be performed on that data into a single container called a class.

- Data is insulated, thus not directly accessible from the outside world. This is known as ‘data hiding’.

- Encapsulation *abstracts* away implementation details from a user of that class.

```java
// A Java class, what access modifiers are allowed?
public class Rectangle {
    // Hide our data from the outside world. Encapsulation!
    // Design decision, should width and length be mutable?
    private int length;
    private int width;

    // Constructor.. design decision, should we have more?
    public Rectangle(int width, int length) {
        this.setWidth(width);
        this.setLength(length);
    }

    // Abstracting the behaviors of rectangle.
    public int getArea() {
        return width * length;
    }

    // By encapsulating and abstracting our draw function
    // our implementation could chage without affecting users!
    public void draw() {
        // ...
    }

    // Accessors and mutators, hiding and protecting our data
    public void setLength(int length) {
        if(length < 1) throw new IllegalArgumentException("Invalid length");
        this.length = length;
    }

    public int getLength() {
        return length;
    }

    public void setWidth(int width) {
        if(width < 1) throw new IllegalArgumentException("Invalid width");
        this.width = width;
    }

    public int getWidth() {
        return width;
    }
}
```

Inheritance (Specialization)

- The ability to have an object or class ‘specialize’ another object, inheriting parent data and behavior.

- subclasses objects form a subset of the set of superclass objects.

- a subclass object can be used whenever a superclass object is expected.

- a superclass is ‘open for extension’, moreover, it allows behavior to be extended without modifying it’s source code.

- gives rise to a potentially complex hierarchies.

```java
// What are the characteristics of an abstract class?
// 1) Define methods which can be used by the inheriting subclass.
// 2) Define abstract methods which the inheriting subclass *must* implement.
// 3) Provide an interface which allows subclasses to be used as Animals (polymorphism).

public abstract class Animal {
    // 'open for extension'
    // adding new types should not require modifying Animal
    public abstract String sound();

    public void vocalize() {
        System.out.println(this.sound());
    }
}

// Cats & Dogs 'specializes' an Animal, but they are still Animals
// Instances of them can be used in case where an Animal instance is expected.

public class Cat extends Animal {
    @Override
    public String sound() {
        return "Meow!";
    }
}

public class Dog extends Animal {
    @Override
    public String sound() {
        return "Woof!";
    }
}
```

Polymorphism

- *poly* means ‘many’ and *morph* means ‘forms’.
- The *provision of a single interface* to entities of different types.
- The ability to execute different implementations of a method based on the *type* of the object at runtime.
- Note that last bullet again, we can encode conditional logic by using polymorphism!
- there are many types of polymorphism, ‘subtype’, ‘ad hoc’, ‘parametric’, we will talk about many of them in this course.

```java
// This class lives somewhere in our codebase, thought we did not write it.
// It is an abstraction and we don't care about its implementation
public class PetStore {
    public static List<Animal> getPets() {
        // ....
    }
}

// Lets pet all the animals in the pet store!

public void petAnimals() {
    for(Animal a : PetStore.getPets()) {
        // 'Leaky abstraction'! maintenance pain!
        // What if the pet store starts selling birds?
        if(a instanceof Dog) {
            System.out.println("Scratch behind ear!");
        } else if (a instanceof Cat) {
            System.out.println("Pet gently on back!");
        } else {
            // not 'type safe'. This is a big problem!!!
            System.out.println("I don't know what you are.");
        }
    }
}

// We could encode  how to pet the animal inside the subclasses of Animal
// the same way we did with vocalize() a couple slides ago.
// This leads to much cleaner code and is future-proof against new types of Animals.
public void petAnimalsWithPolymorphism() {
    for(Animal a : PetStore.getPets()) {
        a.pet();
    }
}
```

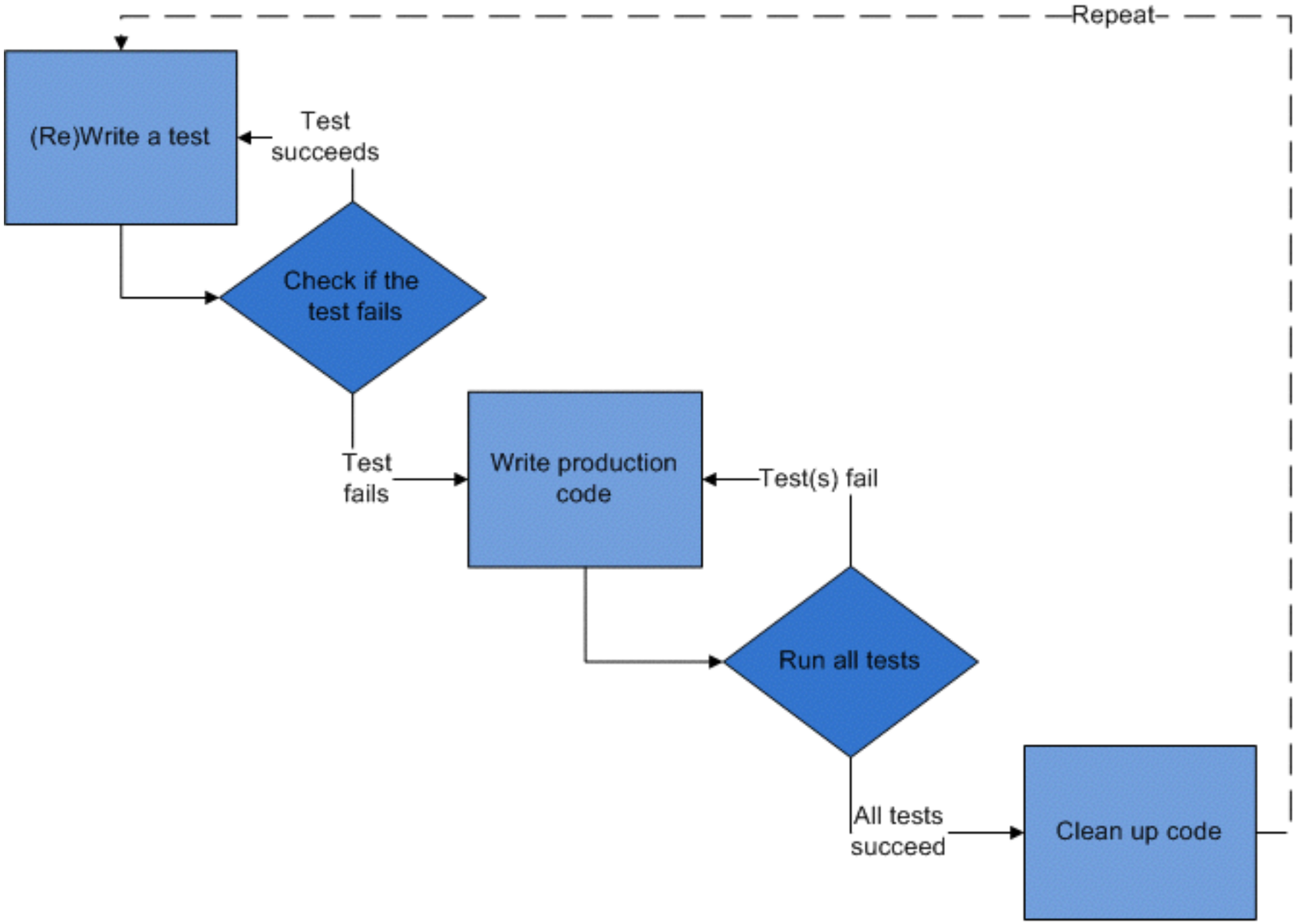
Enough blabbing…

https://github.com/nyu-oop/point-java

Test-driven development

- a software development process that relies on the repetition of a very short development cycle
    - decompose subproblems
    - write a test first that fails for a subproblem
    - write minimal solution
    - refactor the code to acceptable standards
    - move on to next subproblem

Test-driven flow



TDD Benefits

- Better assurance of correctness!
    - Testing end-to-end every time you make a change to a large codebase is very inefficient and not rigorous.
    - Having a test suite catches ‘regressions’ when new code is introduced and breaks some other test.
- Less time to develop!
    - Though there is “more code” the actual time of development is shorter (less time debugging)
- Promotes better design!
    - since TDD requires that the developers think of the software in terms of small units that can be written and tested independently
- Gets you a better grade!

Unit Testing

- A software testing method by which individual *units* of source code are tested *independently*
- Intuitively, one can view a unit as the smallest testable part of an application.
- Unit tests are short code fragments that utilize *assertions* to test conditions in your code.
- We will use JUnit to help us write these tests.

Back to the code..

https://github.com/nyu-oop/point-java