Object-Oriented Programming

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

What is OOP

- The most popular programming languages developed in the last 30 years are, for the most part, Object-Oriented languages, but this wasn't always the way.
- The prevailing paradigm before that was “Procedural Programming”
- It had functions, but the source files for large programs would be many thousands of lines long!
- This led to software that was extremely difficult to understand and change.

What is OOP

- OOP was a direct response to that condition
- In an OOP language, this one large program will instead be split apart into self contained *objects*, almost like having several mini-programs.
- Each *object* represents a different part of the application. Now each object contains its own data and its own logic, and they communicate and collaborate to execute your program.

What is OOP

- Furthermore, OOP is…
  - a programming method that is used to help organize your code when you have complex programs that require a lot of code.
  - a set of language constructs that help you organize your code according to this method.
- *Thats it.*

Benefits of OOP

- Allows us to organize our code in a way that allows us to isolate responsibility to a single entity, therefore reducing complexity
    - “Do one thing and do it well”
- Allows us to have code that is reusable
    - “Don’t repeat yourself”
- Allows us to model the “primitives” in our system as types.

Nouns & Verbs

- So how do we make these mini-programs?
- Identify the entities in the program. Our nouns.
- Identify the actions in the program. Our verbs.

- For example, lets say we had wanted to write the software for an online shopping application.
  - What types of entities would be in the system? ex ‘Shopping Cart’
  - What kinds of actions would be performed by the actors? ex. ‘checkout’

Classes

- A class represents a '*noun*' in our program
- It has two attributes..
  - *State* - the variables on the class.
  - *Methods* - the ways in which the object can interact with its data, the '*verbs*'.

Classes

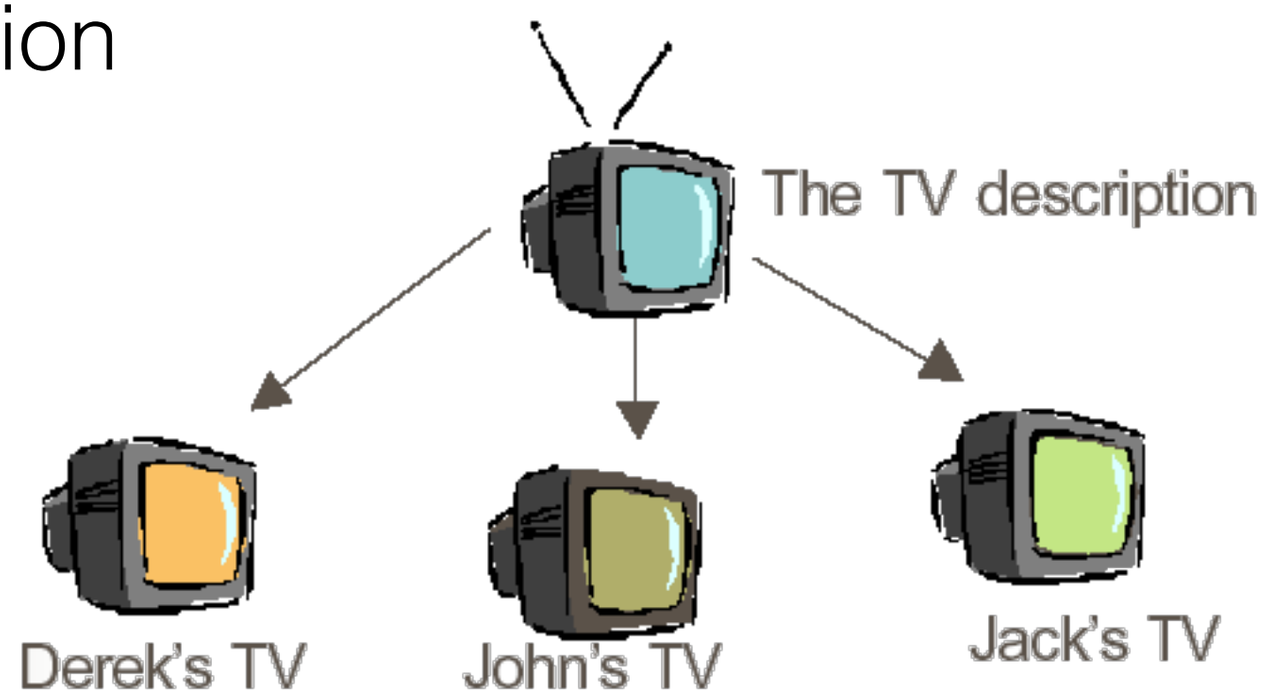
- If we think of a real-world object, such as a television, it will have several features and properties:
    - We do not have to open the case to use it.
    - We have some controls to use it (buttons on the box, or a remote control).
    - We understand the concept of a television without necessarily 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 purpose
  - Represent a clear concept
  - Be complete and well-documented
  - Do one thing and do it well!

Objects

- An object is an instance of a class. What we get when we call **new**
- You could think of a class as the description of a concept, and an object as the realization of this description



Classes & Object Example

```java
public class Rectangle {

    final int length;
    final int width;

    // Constructor.. special kind of function.
    public Rectangle(int w, int l) {
        width = w;
        length = l;
    }

    public int getArea() {
        return width * length;
    }

}
```

```java
public class SomeOtherClass {

    public static void main(String[] args) {
        Rectangle r = new Rectangle(100, 50);
        System.out.println(r.getArea());
    }

}
```

Principles of OOP

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 its 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.

Types and Subtypes

- Classes define a *type*. For example, variable *a* is of type “Bird”

```
Bird a;
```

- Subclasses can define subtypes. For example, “Duck” extends “Bird” so it is considered a *subtype* of “Bird”

```
Duck b;
```

- In the example above, “Duck” is a *subtype* of “Bird”, and “Bird “is a *supertype* of “Duck”.

- Ducks can do anything that Birds can do because Ducks *is-a* Bird in our program.

Is-a Relationship

- If B extends A…
  - type A is a subtype of type B when A's specification implies B's specification.
  - Moreover, any object (or class) that satisfies A's specification also satisfies B's specification, because B's specification is weaker.
  - B *is-a* A because B is a subclass (and therefore subtype) of A

Inheritance & Subtypes

- When you create a subclass you are inheriting instance variables and methods from a superclass

- A subclass can then implement new features that are independent of the superclass, but it still contains all of the methods and fields available to the superclass.

- A subclass can do everything a superclass can do, and maybe more.

- Again, an instance of a subclass *is-a* instance of a superclass too!

Inheritance & Subtypes

- Polymorphism allows us to treat a variable as its supertype. For example:

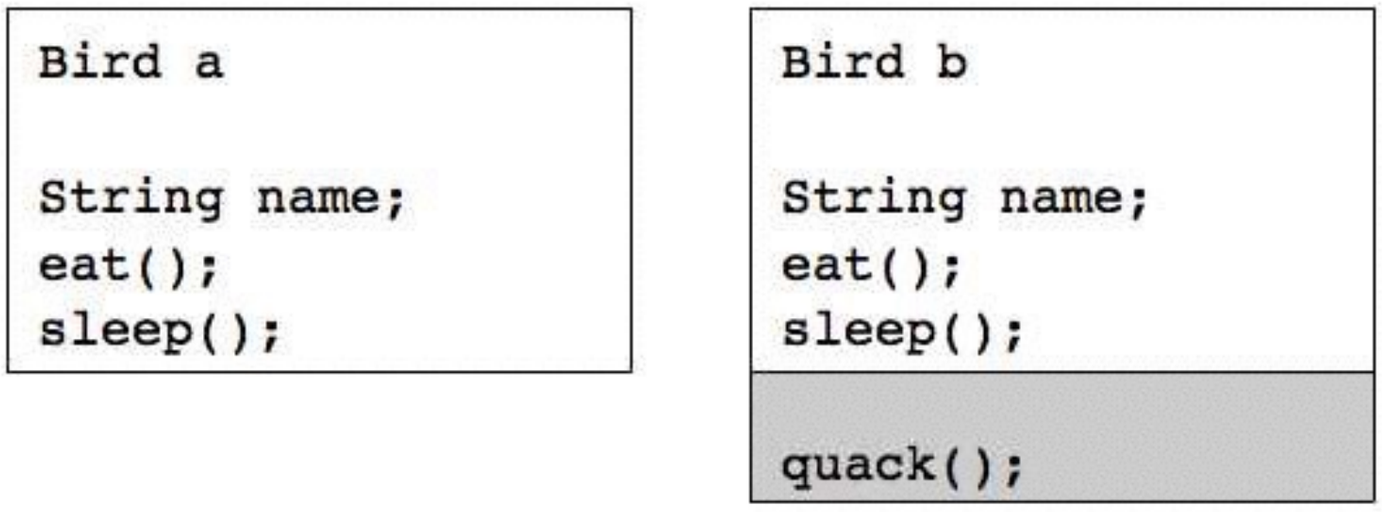
```
Bird a = new Bird();
Bird b = new Duck();
```

- The above example may look wrong, but in our world Ducks extend Birds. *They are Birds* – but with more stuff added to them.

- We can safely call any Bird method on a Duck object and be guaranteed that it will be available.

Inheritance & Subtypes

```
Bird a = new Bird();
Bird b = new Duck();
```



Static Vs Dynamic Type

- This should make sense if you remember that a Duck is-a Bird

```
Bird a = new Bird();
Bird b = new Duck();
Duck c = new Duck();
```

  Java uses the “static” type of the object to determine which methods and instance variables are available on that object.

- The static type is the type declared on the left hand side of the initialization.
  - (not to be confused with static variables or memory)

- The dynamic type is the type declared on the right hand side of the initialization.

```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();
    }
}
```

Dynamic Dispatch

- In the previous example we told Java that each object was of type Animal. This is the “static” type of the object.

- However,some of the *objects* were not of type Animal – they were subtypes of Animal. In this case these two objects had “dynamic” types that are different from their “static types”.

- When you call an overridden method on an object with a static type that differs from its dynamic type, which method gets called?
  - The method on the static type (Animal) or the method on the dynamic type (Dog)?

Dynamic Binding

- The answer is that Java uses the “dynamic” type when determining which method to call. This is called “dynamic binding”

- Java waits until the run time to determine which method to call by checking an object’s static type and calling the appropriate method.

- *We will understand this in very great detail by the end of the semester!*

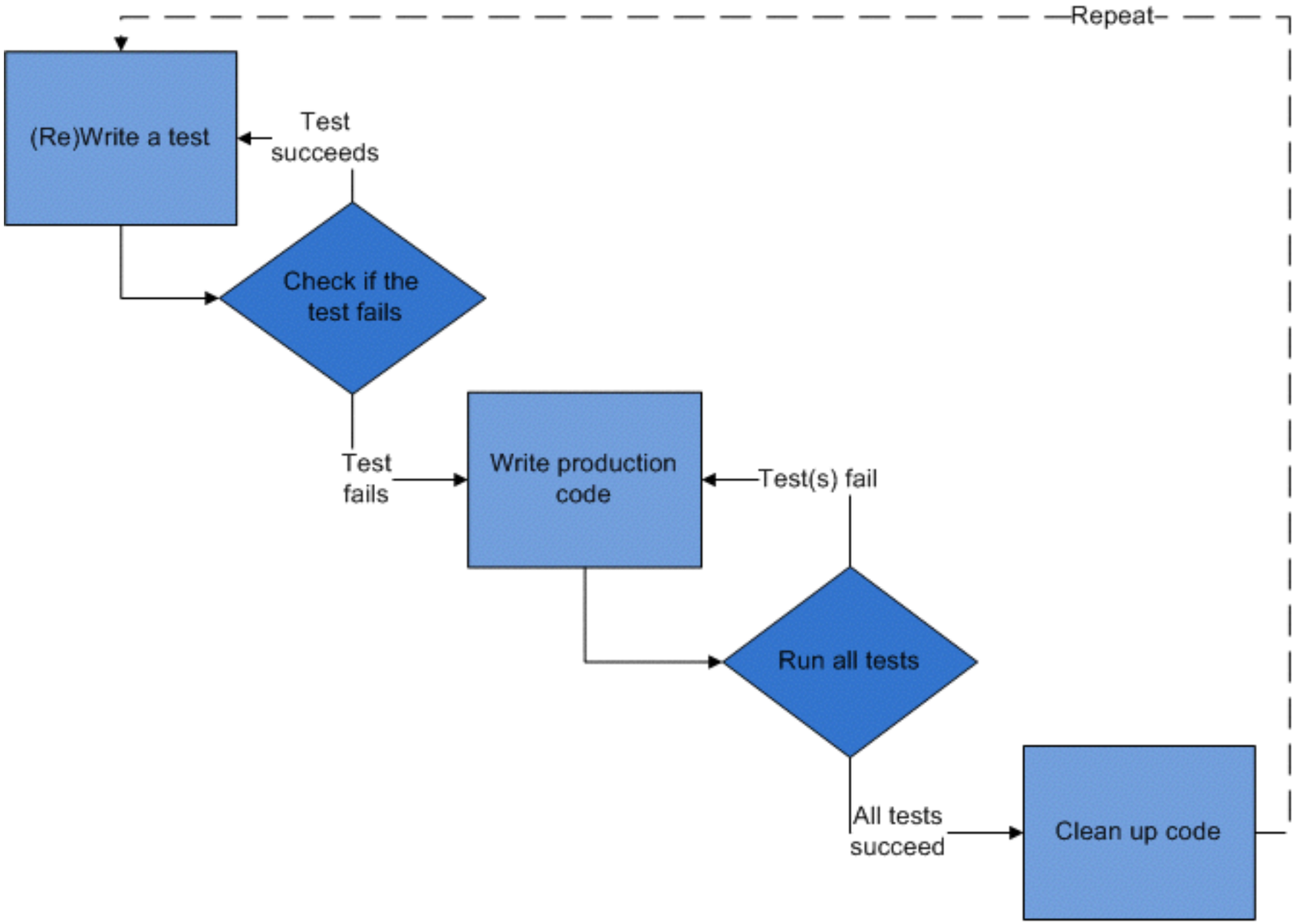
Enough blabbing…

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

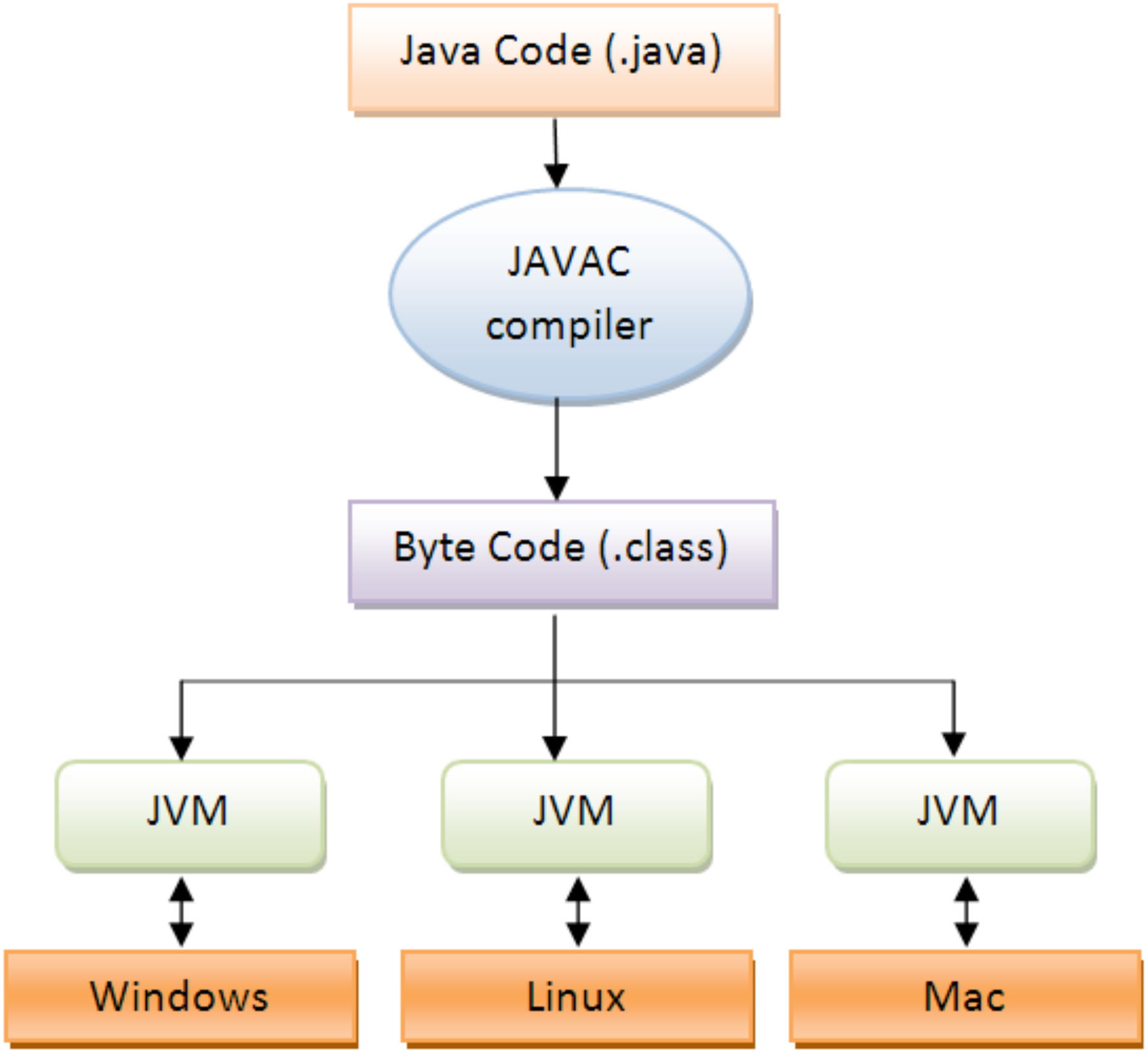
Java and the JVM

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.


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

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.

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. (managed by SBT for us)