Inheritance, Abstract Classes & Polymorphism

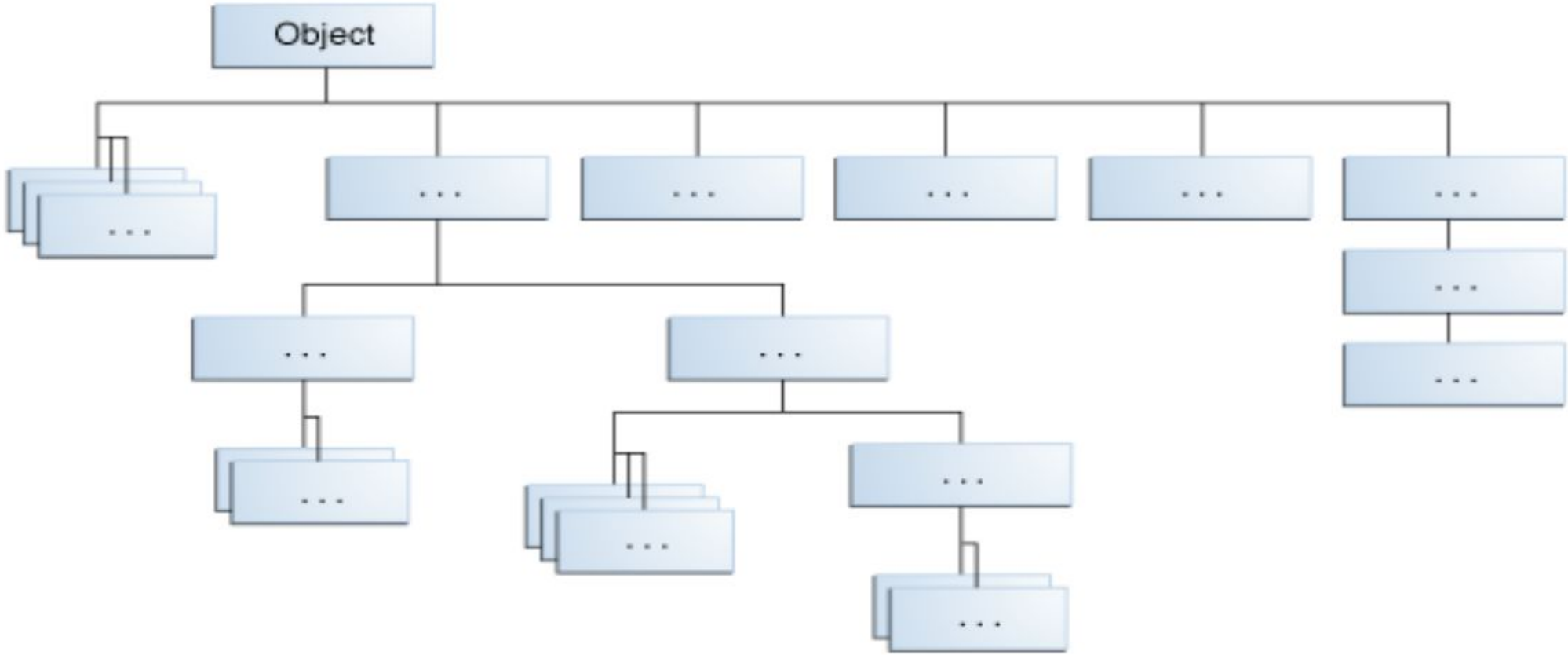
Inheritance

- classes can be derived from other classes, thereby inheriting fields and methods from those classes.
- A class that is derived from another class is called a **subclass** (also a derived class, extended class, or child class).
- The class from which the subclass is derived is called a **superclass** (also a base class or a parent class).
- Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Inheritance

- Classes can be derived from classes that are derived from classes that are derived from classes, and so on
  - ultimately derived from the topmost class, Object.
  - Such a class is said to be descended from all the classes in the inheritance chain stretching back to Object.
- Inheritance allows reuse of the fields and methods of the existing class without having to write (and debug!) them yourself.
- A subclass inherits all the members (fields, methods, and nested classes) from its superclass.
  - Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Example time!



All Classes in the Java Platform are Descendants of Object

Overriding methods

- The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.
- The overriding method has the same name, number and type of parameters, and return type as the method that it overrides.
- An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a covariant return type.

Overriding methods

- When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass.
- If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error.

Hiding Methods

- if a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.
- The distinction between hiding a static method and overriding an instance method has important implications:
  - The version of the overridden instance method that gets invoked is the one in the subclass.
  - The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Polymorphism

- The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages.
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class
- Btw, lets also look at overloading also…..

Example time!

You don’t want people to change class and behavior

- Give them a ‘Final’ warning!
- Don’t inherit my class
- Don’t change this method

Abstract Class & Abstract Methods

- An abstract class is a class that is declared abstract—it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.
- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon)
- The subclass usually provides implementations for all of the abstract methods in its parent class.
  - However, if it does not, then the subclass must also be declared abstract.

Abstract Class & Abstract Methods

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself *must* be declared `abstract`, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

Abstract Classes vs Interfaces

- Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.
- But you can declare fields that are not static and final, and define public, protected, and private concrete methods.
- With interfaces, all fields are automatically public, static, and final, and all methods that you declare are public.
  - You can have protected and private methods in Abstract classes
- You can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which to use? (Use interfaces if)

- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

Which to use? (Use Abstract Classes if)

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields.
  - This enables you to define methods that can access and modify the state of the object to which they belong.

Example time!

Thanks!