Chapter 5: Class Design.

Introducing Class Inheritance.

When creating a new class in java, you can define the class to inherit from an existing class. Inheritance is the process by which the new child subclass automatically includes any public or protected primitives, objects, or methods defined in the parent class.

Java supports single inheritance, by which a class may inherit from only one direct parent class. A parent class may have multiple children in this case. Java also support multiple levels of single inheritance.

Java does not support multiple inheritance, in which a child class inherits from multiple parents, as it can lead to complex code and difficult to maintain. Java does allow one exception to the single inheritance rule: classes may implement multiple interfaces. Marking a class as final won't make it possible to apply inheritance; the compiler will throw an error.

In order to apply it, we use the 'extends' keyword.

```
public abstract class ElephantSeal extends Seal {
// Methods and variables defined here
}
```

Where *public* is the access modifier, *abstract* or *final* keyword is optional, *class* keyword required, *ElephantSeal* is the class name, *extends* followed by the parent class name to be inherited.

Applying class access modifiers.

As we know we can apply access modifiers (public, private, protected, default) to both classes and methods. For the top class that is in the file, only public and default package-level modifiers can be applied to it. The protected and private modifiers can be applied to inner classes.

If a class is default / no access modifier, only other classes in the same package can see it and extend it.

Creating Java objects.

In Java, all classes inherit a single class: java.lang.Object. Object is the only class without a parent class. But we don't see this in our code, and that's because at compile time java inserts code. If in your class, you don't extend any other class, java will automatically extend Object:

```
public class Zoo extends java.lang.Object { }
```

Only when the child class is extending a parent, the compile won't add this code, but it will to the parent if it doesn't extend from another parent. Java.lang.Object will always be on top of the tree.

Defining constructors.

We have learned that java creates a default no-argument constructor if no constructor is coded in a class, and that we can refer to inner constructors using the word this().

When inheritance is applied, we can use the super() keyword to refer to the constructor/s in the parent class. As this) keyword, super() needs to be the first line, otherwise it won't compile. For more examples see Animal.java and Zebra.java classes.

Understanding compiler enhancements.
Java automatically creates a no-argument constructor for all classes. If a child inherits a class, with no constructor definition, the java compile will automatically call the parent class' constructor with the super() keyword. The following Java classes definitions are the same, as Java will convert them all to the last one:

```
public class Donkey { }
public class Donkey {
     public Donkey() { } }
public class Donkey {
     public Donkey() {
     super();
} }
```

However, in the case in which the parent class does have a constructor defined, the child class won't compile if no constructor is defined in the child per se, as Java will insert a no-argument constructor during compile time, as well as adding the super() keyword:

```
public class Mammal { public Mammal(int age) {} }
public class Elephant extends Mammal { } // Does not compile.
```

The solution to this problem is to explicitly create a constructor in the child, and call super, passing an int argument.

Reviewing Constructor rules.

1. Frist statement is a call to another constructor within the class [this()] or a constructor in the direct parent class [super()].
2. The super() call may not be used after the first statement of the constructor.
3. If not super() call is declared, Java will insert a no-argument super(() as the first statement of the constructor.
4. If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

When calling a constructor, the parent one will always execute first.

Calling inherited class members.
If the parent class and child class are part of the same package, the child class may also use any default members defined in the parent class. A child class may never access a private member of the parent, at least not through direct reference. For examples, see Fish.java and Shark.java classes.

Super() vs. super

In the same way this() and this are unrelated to Java, super() and super are as well. Super() will explicitly call the parent constructor and may only be used in the first line of a constructor in the child class.

The second super is a keyword used to reference a member defined in a parent class and may be used throughout the child class:

```
public Rabbit(int age) {
      super();
      super.setAge(10);
}
```

Inheriting Methods.

When inheriting classes, there might be conflicts between the methods defined in both parent and child classes. We'll review how Java handles these scenarios.

Overriding a method.

If you may want to define a new version of an existing method of the parent class, in the child, that's what we call overriding a method. We override a method by defining a new one with the same signature and return type as the method in the parent class. See Canine.java and Wolf.java classes for examples.

The compiler will perform the following checks when you override a nonprivate method:

1. The method in the child class must have the same signature (including parameter list) as the method in the parent class. Note: if not, it would be overloading.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or boarder than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.

Redeclaring private methods.

If a member of the parent class is private, it's not accessible to the child. However, Java allows you to declare or redeclare a new version of the same method in the child class, with its unique implementation. This is not overriding in this case and therefore none of the rules are applied. Java allows to do this with the same or modified signature as the method in the parent class. They would end up totally independent, unrelated from each other.

Hiding static methods.

A hidden method occurs when a child class defines a static method with the same name and signature as a static method defined in the parent class. Method hiding is similar but slightly different than overriding. The same rules apply but one rule is added:

1. The method in the child class must have the same signature (including parameter list) as the method in the parent class. Note: if not, it would be overloading.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or boarder than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.
5. The method defined in the child class must be marked as *static* if its marked as *static* in the parent class. If it's not marked *static* in the parent class, it cannot be *static* in the child.

Real life scenario: it is a good practice to avoid hidden methods, as it entitles complications in the code, also makes it hard to read.

Overriding vs. Hiding Methods.
The code behaves differently at runtime.

a) The child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method. In this manner, the parent method is never used unless an explicit call to the parent method is references, using the syntax *super.method().*
b) The parent version of a hidden method is always executed if the call to the method is defined in the parent class.

For this, see examples under Overriding vs. Hiding folder.

Creating final methods.
The rule is very simple: final methods cannot be overridden or hidden. By doing this, we are ensuring the method / behavior in the parent class is always present / executed.

<u>Inheriting Variables.</u>
Java doesn't allow variables in the parent class to be overridden, but instead, hidden.

Hiding variables.
To hide a variable, you only need to define it with the same name in the child as in the parent class. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference. The rule for this is similar as hiding a method. If you are referencing the variable within the parent class, the variable in parent is used, same as if you reference the variable in the child, the one in the child is used. You can also reference the variable in the parent class by using the super keyword. See Rodent.java and Mouse.java classes for examples.

Real life scenario: hiding variables is considered poor coding practice.

Creating Abstract Classes.

An abstract class is marked with the abstract keyword and cannot be instantiated. An abstract method is marked with the same keyword defined inside of an abstract class, for which no implementation is provided. It is extended by the child classes, and another difference from interfaces is that methods can be defined with code implementation in them.

Defining an abstract class.

Some rules to keep in mind. An abstract class is not required to have abstract methods, it may have nonabstract members. However, only an abstract member can be declared in a class like this. When a method is abstract, no code implementation can be included, otherwise it won't compile, only the method signature is allowed.

No abstract class or member can be marked as final, including nonabstract method with code implementation that we want the child classes to extend and override. Finally, a method may not be marked as both abstract and private.

Creating a concrete class.

An abstract class becomes useful when it's extended by a concrete class. A concrete class is the first nonabstract subclass that extends an abstract class and is required to implement all inherited abstract methods. When you see this, make sure it is implemented all abstract method, otherwise it won't compile.

Extending an abstract class.

An abstract class can extend another abstract class, without the need to implement the method of the parent. However, the fist concrete class that does extend the abstract class, does need to implement all abstract methods. See Animal.java, BigCat.java and Lion.java classes for examples.

There is an exception rule: a concrete class is not required to provide an implementation of an abstract method if an intermediate abstract class provides it. See BigCatV2.java and LionV2.java classes for examples.

Implementing Interfaces.

Java doesn't allow multiple inheritance but does allow a class to implement any number of interfaces. An interface is an abstract data type that fines a list of abstract public methods that any class implementing the interface must provide. Can also include constant variables and default methods, and does not provide any code implementation, ever. We use the *interface* keyword in the class signature.

Defining an interface:

```
public abstract interface CanBurrow {
     public static final int MAXIMUM_DEPTH = 2;
     public abstract int getMaximumDepth();
}
```

Where the abstract keyword in both class definition and method signature, as well as the static keyword in that constant above, are assumed. This means that whether or not you provide it, the compiler will insert it for you.

Implementing the interface:

```
public class FieldMouse implementd CanBurrow {
     public int getMaximumDepth() {
          return 10;
     } }
```

Think of an interface as a kind if an abstract class, since they share many rules:

1. Interfaces cannot be instantiated directly.
2. Is not required to have any methods.
3. May not be marked as final.
4. All top-level interfaces are assumed to have public or default access. They are assumed to be abstract whether you use the keyword or not. Making a method private, protected or final will trigger a compiler error.
5. All nondefault methods inside are assumed to be abstract and public in their definition. Making it private, protected or final will trigger a compiler error.

Inheriting an interface.

There are two inheritance rules to keep in mind:

1. An interface that *extends* another interface, as well as an abstract class that *implements* an interface, inherits all the abstract methods as its own abstract methods.
2. The first concrete class that *implements* an interface, or *extends* an abstract class that implements an interface, must provide an implementation for all inherited abstract methods.

When an abstract class implements an interface, it's not required to provide an implementation, but a concrete class is required to do so.

Abstract methods and multiple inheritance.

If a concrete class implements multiple interfaces, with they have method with the same name, as long as it has the same signature, only one will be implemented. If the method signatures are the same but different return type, it won't compile. If they are different return type, and have different parameters, which now will be consider different method signatures, then both will be implemented. Same as if an interface is extending other conflictive interfaces or an abstract method is implementing them, compile error will be thrown.

See Bears, Herbivores, Omnivore classes for more examples.

Interface variables.

Rules:

- Interface variables are assumed to be public, static and file. Making a variable private or protected will throw a compiler error.
- Its value must be defined when it's declared, since it's final.

Default interface methods.
It's a method defined inside the interface, with the *default* keyword, which body implementation is provided. A class that extends an interface with default methods, have the option to override them, but not required to do so. If the class doesn't override the method, default implementation will be used. In this manner, the method definition is concrete, not abstract.

The purpose of these is backward combability. In the case scenario in which a interface is extended by many classes, in case a code change needs to take place, a default method won't break the other classes implementation. Otherwise, all classes that extends the interface would need to make code changes.

```
public interface IsWarmBlooded {
     boolean hasScales();
     public default double getTemperature() { return 10.0; }
}
```

The default keyword is different from the access modifiers shown in Chapter 4. That's why getTemerature() is public. This default method only refers to its special use inside of an interface.
Rules:

- A default method may only be declared within an interface and not within a class or abstract class.
- A default method must be marked with the *default* keyword and a body must be provided.
- It's not assumed to be static, final or abstract, as it may be used or overridden by a class that implements the interface.
- It's assumed to be public. It won't compiled if marked as private or protected.

Default methods and multiple inheritance.
If a class implements two interfaces that have the default methods with the same name and signature, the compiler will throw an error. There is one exception to this rule: of the subclass overrides the duplicate default methods, the code will compile without issue – the ambiguity about which version of the method to call has been removed.

Static interface methods.
Rules:

- A static method is assumed to be public. If marked private or protected it won't compile.
- To reference the static method, a reference to the name of the interface must be used. This happens because static methods are not inherited when subclass implements the interface.

Understanding Polymorphism.
It's the property of an object to take many different forms.
A java object may be accessed using:

- A reference with the same type as the object.
- A reference that is a superclass of the object.

- A reference that defines an interface the object implements, either directly or through a superclass.

A cast s not required if the object is being reassigned to a super type or interface of the object.

Please see *Lemur.java* for examples. In this example, only *Lemur* object is instantiated and then assigned to new objects declared for *HasTail* interface and *Primate* parent class, which is the nature of polymorphism. The new objects only have access to the members defined within them; therefore they cannot access the members unique to *Lemur* class.

Object vs. Reference.

All objects extend from the java.lang.Object class, for which the following example is possible:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

When polymorphism is applied, the object in the Heap memory remains the same, only the reference we are using to access the object is changed, along the ability to what members we can access. Rules:

1) The type of the object determines which properties exist within the object in memory.
2) The type of reference to the object determines which methods and variables are accessible to the java program.

Casting objects.

However, if we want to access all members of an object when a reference doesn't allow us to, can solve that problem by doing an explicit cast:

```
Primate primate = lemur;
Lemur lemur2 = (Lemur)primate;
```

Basic rules to keep in mind for casting objects:

1. Casting an object from a subclass to a superclass doesn't require an explicit cast.
2. Casting an object from a superclass to a subclass requires an explicit cast.
3. The compiler will not allow casts to unrelated types.
4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

Virtual methods.

The most important feature of polymorphism is to support virtual methods. It's a method in which the specific implementation is not determined until runtime. In fact, all non-final, non-static, non-private Java methods are considered virtual, since any of them can be overridden at runtime.

In simpler language, a class can extends another and override one of its methods, but Java may be unaware of that during compiling time. At run time, while making an instance, the implementation is determined when you call the overridden version of the method. See Bird.java and Peacock.java classes under Polymorphism folder for examples.

Polymorphic parameters.

It's the ability to pass instances of a subclass or interface to a method, as a parameter. For example, you can define a method that takes an instance of an interface as a parameter. In this manner, any class that implements the interface can be passed to the method. Since you're casting from a subtype to a supertype, an explicit cast is not required. See Reptile.java, Alligator.java, Crocodile.java and ZooWorker.java as examples.