Chapter 6

Class Design

Understanding Inheritance

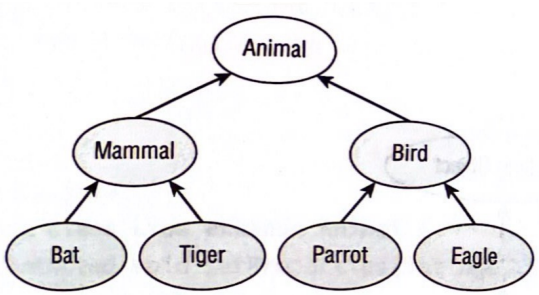
Declaring a Subclass

Make sure the superclass is not marked ~~final~~

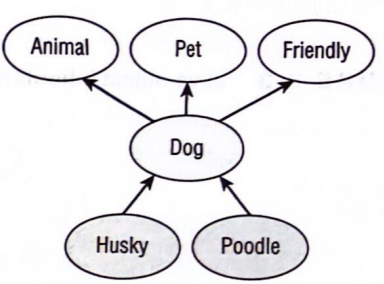
Class Modifiers

Modifier	Description	Chapter covered
`final`	The class may not be extended.	Chapter 6
`abstract`	The class is abstract, may contain `abstract` methods, and requires a concrete subclass to instantiate.	Chapter 6
`sealed`	The class may only be extended by a specific list of classes.	Chapter 7
`non-sealed`	A subclass of a sealed class permits potentially unnamed subclasses.	Chapter 7
`static`	Used for `static` nested classes defined within a class.	Chapter 7

Single vs. Multiple Inheritance



Single Inheritance



Multiple Inheritance

Java doesn't support multiple inheritance

Inheriting Object

In Java, all classes inherit from a single class:

java.lang.Object

Creating Classes

Applying Class Access Modifiers

- Can have many classes with package access
- Trying to declare a top-level class with protected or private will lead to a compiler error

```
//Bear.java
class Bird { }
class Bear { }
class Fish { }
```

```
//ClownFish.java
protected class ClownFish { }
//BlueTang.java
private class BlueTang { }
```

Accessing the this Reference

- The this reference refers to the current instance of the class & can be used to access any member of the class, including inherited members
- Cannot be used in a static method or static initializer block

```
public void setColor(String color) {
    this.color = color;
}
```

Calling the super Reference

The super reference is similar to the this reference, except that it excludes any members found in the current class

```
public int getSpeed() {
    return super.speed;
}
```

if speed isn't in the superclass, this code doesn't compile

Declaring Constructors

Creating a Constructor

- Declaring multiple constructors with different signatures
  → constructor overloading
- Instantiation : `new Turtle(15);`

↓

Java allocates memory for the new obj.

↓

Looks for a constructor with a matching signature

↓

Calls it

The Default Constructor

If we don't include any constructors in the class, Java will create one without any parameters

Calling Overloaded Constructor with this()

```
public class Hamster {
  private String color;
  private int weight;
  public Hamster(int weight, String color){
    //Something
  }
  public Hamster (int weight) {
    //Something
  }
}
```

```
public Hamster (int weight) {
  Hamster (weight, "brown"); //Error
}
```

```
public Hamster(int weight) {
    new Hamster(weight, "brown"); // Extra object
}
```

```
public Hamster(int weight) {
    this(weight, "brown");
}
```

When using this(), Java calls another constructor on the same instance of the class

```
public Hamster(int weight) {
    System.out.println("chew");
    this(weight, "brown"); // Error
}
```

must be the 1st statement in the constructor

(there can be only one call to this() in any constr.)

```
public class Gopher {
  public Gopher(int dugHoles) {
    this(5); //Error
  }
}
```

```
public class Gopher {
  public Gopher() {
    this(5); //Error
  }
  public Gopher(int dugHoles) {
    this(); //Error
  }
}
```

cyclic constructor calls

Calling Parent Constructors with super()

```
public class Zebra extends Animal {
  public Zebra(int age) {
    super(age); // Refers to constructor in Animal
  }
  public Zebra() {
    this(4);
  }
}
```

Rules of calling super() are similar to rules of calling this()

```
public class Animal {
  private int age;
  public Animal(int age) {
    super();
    this.age = age;
  }}
```

refers to constructor in java.lang.Object

The Java compiler automatically inserts a call to `super()` if you don't explicitly call `this()` or `super()` as the first line of a constructor

```
public class Donkey { }
```

=

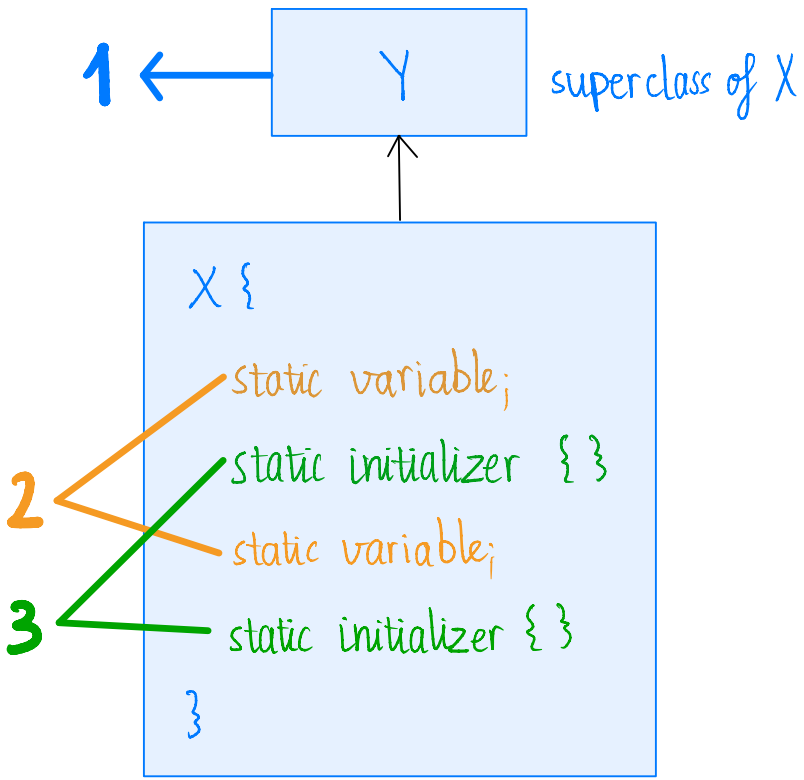
```
public class Donkey {
  public Donkey() {
    super();
  }
}
```

```
public class Mammal {
  public Mammal(int age) { }
}
public class Seal extends Mammal { } //Error
public class Elephant extends Mammal {
  public Elephant() { } // Error
}
```

Initializing Objects

Initializing Classes

The order of initialization for class X



2, 3: in the order in which they appear in the class

```
public class Animal {
   static { System.out.print("A"); }
}
public class Hippo extends Animal {
  public static void main(String[] grass){
    System.out.print("C");
    new Hippo();
    new Hippo();
    new Hippo();
  }
  static { System.out.print("B"); }
}
```

Prints ABC exactly once. Although the main() creates 3 instances, the class is loaded only once.

Initializing final Fields

By the time the constructor completes, all final instance variables must be assigned a value exactly once.

```
public class MouseHouse {
   private final int volume;
   private final String type;
   {
      this.volume = 10;
   }
   public MouseHouse(String type) {
      this.type = type;
   }
   public MouseHouse() { // Error
      this.volume = 2;        // Error
   }
}
```

Initilializing Instances

1. Initialize class X if it has not been previously initialized.
2. If there is a superclass Y of X, then initialize the instance of Y first.
3. Process all instance variable declarations in the order in which they appear in the class.
4. Process all instance initializers in the order in which they appear in the class.
5. Initialize the constructor, including any overloaded constructors referenced with `this()`.

```
public class ZooTickets {
  private String name = "BestZoo";
  { System.out.print(name + "-"); }
  private static int COUNT = 0;
  static { System.out.print(COUNT + "-"); }
  static {
    COUNT += 10;
    System.out.print(COUNT + "-");
  }
  public ZooTickets() { System.out.print("z-"); }
  public static void main(String... patrons) {
    new ZooTickets();
  } } // Output: 0-10-BestZoo-z-
```

```
public class Cuttlefish {
  private String name = "swimmy";
  { System.out.println(name); }
  private static int count = 0;
  static { System.out.println(count); }
  { count ++; System.out.println(count); }
  public Cuttlefish() {
    System.out.println("Constructor");
  }
  public static void main(String[] args){
    System.out.println("Ready");
    new Cuttlefish();
  }}
```

0 → Ready → swimmy → 1 → Constructor

```
1:  class GiraffeFamily {
2:     static { System.out.print("A"); }
3:     { System.out.print("B"); }
4:
5:     public GiraffeFamily(String name) {
6:        this(1);
7:        System.out.print("C");
8:     }
9:
10:    public GiraffeFamily() {
11:       System.out.print("D");
12:    }
13:
14:    public GiraffeFamily(int stripes) {
15:       System.out.print("E");
16:    }
17: }
18: public class Okapi extends GiraffeFamily {
19:    static { System.out.print("F"); }
20:
21:    public Okapi(int stripes) {
22:       super("sugar");
23:       System.out.print("G");
24:    }
25:    { System.out.print("H"); }
26:
27:    public static void main(String[] grass) {
28:       new Okapi(1);
29:       System.out.println();
30:       new Okapi(2);
31:    }
32: }
```

Output:

AFBECHG

BECHG

Inheriting Members

Overriding a Method

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method. (new unchecked exceptions are fine)
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*.

```
public class Camel {
   public int getNumberOfHumps() {
      return 1;
   }}
public class BactrianCamel extends Camel {
   private int getNumberOfHumps() { //Error, rule#2
      return 2;
   }}
```

Because of polymorphism, the overridden method replaces the method on all calls

```
public class Reptile {
   protected void sleep() throws IOException {}
   protected void hide() {}
   protected void exitShell() throws
                     FileNotFoundException {}
}
public class GalapagosTortoise extends Reptile {
   public void sleep() throws
                     FileNotFoundException {}
   public void hide() throws
                     FileNotFoundException {}//Rule #3
   public void exitShell() throws
                     IOException {} //Rule #3
}
```

```
public class Rhino {
  protected CharSequence getName() {
    return "rhino";
  }
  protected String getColor() {
    return "grey, black, or white";
  }}
public class JavanRhino extends Rhino {
  public String getName() {
    return "javan rhino";
  }
  public CharSequence getColor() { // Error, rule #4
    return "grey";
  }}
```

```
public class Fish {
  public void swim() { }
}
public class Shark extends Fish {
  @Override
  public void swim(int speed) { } //Error
}
```

Redeclaring private Methods

```
public class Beetle {
  private String getSize() {
    return "5";
  }}
public class RhinocerosBeetle extends Beetle {
  private int getSize() {//unrelated to the above
    return 5;
  }}
```

Hiding static Methods

A hidden method occurs when a child class defines a static method with the same name & signature as an inherited static method defined in a parent class

5th rule: The method defined in the child class must be marked as static if it is marked as static in a parent class

```
public class Bear {
  public static void sneeze() {}
  public void hibernate() {}
  public static void laugh() {}
}
public class SunBear extends Bear {
  public void sneeze() {}
  public static void hibernate() {}
  protected static void laugh() {}
}
```

Hiding Variables

A hidden variable occurs when a child class defines a variable with the same name as an inherited variable defined in a parent class

```
class Carnivore {
  protected boolean hasFur = false;
}
public class Meerkat extends Carnivore {
  protected boolean hasFur = true;
  public static void main(String[] args) {
    Meerkat m = new Meerkat();
    Carnivore c = m;
    System.out.println(m.hasFur); //true
    System.out.println(c.hasFur); //false
  }
}
```

Writing final Methods

final methods cannot be overridden / hidden

```
public class Bird {
  public final boolean hasFeathers() {
    return true;
  }
  public final static void flyAway() {}
}
public class Penguin extends Bird {
  public final boolean hasFeathers() {
    return false;
  }
  public final static void flyAway() {}
}
```

Creating Abstract Classes

Introducing Abstract Classes

- Declared with the abstract modifier
- Cannot be instantiated directly
- May contain abstract methods

4 rules:

- Only instance methods can be marked `abstract` within a class, not variables, constructors, or `static` methods.
- An abstract method can only be declared in an abstract class.
- A non-abstract class that extends an abstract class must implement all inherited abstract methods.
- Overriding an abstract method follows the existing rules for overriding methods that you learned about earlier in the chapter.

Declaring Abstract Methods

- Declared without a body (no ~~{}~~)
- Includes a semicolon (;)
- An abstract class can include all of the same members as a non-abstract class
- An abstract class is not required to include any abstract methods

Creating a Concrete Class

The first concrete subclass that extends an abstract class is required to implement all inherited abstract methods

```
public abstract class Mammal {
  abstract void showHorn();
  abstract void eatLeaf();
}
public abstract class Rhino extends Mammal {
  void showHorn() { }
}
public class BlackRhino extends Rhino {
  void eatLeaf() { }
}
```

Creating Constructors in Abstract Classes

```
abstract class Mammal {
  abstract CharSequence chew();
  public Mammal() {
    System.out.println(chew()); //yummy!
  }
}
public class Platypus extends Mammal {
  String chew() { return "yummy!"; }
  public static void main(String[] args) {
    new Platypus();
  }
}
```

Default constructor in Platypus → call super() in Mammal

Spotting Invalid Declarations

abstract & final Modifiers

```
public abstract final class Tortoise { // Error
  public abstract final void walk(); // Error
}
```

abstract & private Modifiers

```
public abstract class Whale {
  private abstract void sing(); // Error
}
```

abstract & static Modifiers

```
abstract class Hippopotamus {
  abstract static void swim(); //Error
}
```

Creating Immutable Objects

Declaring an Immutable Class

1. Mark the class as `final` or make all of the constructors `private`.
2. Mark all the instance variables `private and final`.
3. Don't define any setter methods.
4. Don't allow referenced mutable objects to be modified.
5. Use a constructor to set all properties of the object, making a copy if needed.

```
public final class Animal {
  private final ArrayList<String> foods;
  public Animal() {
    this.foods = new ArrayList<String>();
    this.foods.add("Apples");
  }
  public List<String> getFoods() { //mutable obj
    return foods;
  }
}
```

```
public final class Animal {
  private final ArrayList<String> foods;
  public Animal() {
    this.foods = new ArrayList<String>();
    this.foods.add("Apples");
  }
  public int getFoodsCount() {
    return foods.size();
  }
  public String getFoodsItem(int index) {
    return foods.get(index);
  }
}
```

Or, make a copy like this:

```
public List<String> getFoods(){
  return new ArrayList<String>(this.foods);
}
```

Performing a Defensive Copy

```
public final class Animal {
  private final ArrayList<String> foods;
  public Animal(ArrayList<String> foods) {
    if(foods == null || foods.size() == 0)
      throw new RuntimeException("foods is required");
    // A caller can keep a reference to foods & modify
    // this.foods = foods;
    // Defensive copy
    this.foods = new ArrayList<String>(foods);
  }
}
```

Review Questions

1/ E ✓

2/ A B F ✓

3/ B C ✓

4/ F ✓

5/ E ✓

6/ D E ✓

7/ ~~G~~ **A** (overridden method replaces on all calls)

8/ D ✓

9/ B E ✓

10/ A C ✓

11/ C ✓

12/ ~~B~~ **C** (Beaver must declare a constr)

13/ A G ✓

14/ B E F ✓

15/ C ✓

16/ ~~G~~ **D** (inherit static variable)

17/ ~~B~~ C ~~E~~ F

18/ D F ~~G~~ (not covariant)

19/ ~~E~~ **F** (error on line 9)

20/ E ✓

21/ B G ✓

22/ D ✓

23/ B ✓

24/ ~~B~~ **C**

25/ B C ✓

26/ ~~B~~ **D** (no access to method of child)

69,2%