Chapter Summary

Inheritance is a feature of Java programs that allows the creation of a parent–child relationship between two types.

The child class of an inheritance relationship (commonly called a subclass) will receive a copy of ("inherit") every field and method from the parent class (superclass). The subclass "extends" the superclass, because it can add new fields and methods to the ones it inherits from the superclass.

A subclass can override a method from the superclass by writing its own version, which will replace the one that was inherited.

Treating objects of different types interchangeably is called polymorphism.

Subclasses can refer to the superclass's constructors or methods using the `super` keyword.

The `Object` class represents the common superclass of all objects and contains behavior that every object should have, such as the `equals` and `toString` methods.

Inheritance provides an "is-a" relationship between two classes. If the two classes are not closely related, inheritance may be a poor design choice and a "has-a" relationship between them (in which one object contains the other as a field) may be better.

An interface is a list of method declarations. An interface specifies method names, parameters, and return types but does not include the bodies of the methods. A class can implement (i.e., promise to implement all of the methods of) an interface.

Interfaces help us achieve polymorphism so that we can treat several different classes in the same way. If two or more classes both implement the same interface, we can use either of them interchangeably and can call any of the interface's methods on them.

An abstract class cannot be instantiated. No objects of the abstract type can be constructed. An abstract class is useful because it can be used as a superclass and can also define abstract behavior for its subclasses to implement.

An abstract class can contain abstract methods, which are declared but do not have bodies. All subclasses of an abstract class must implement the abstract superclass's abstract methods.

Self-Check Problems

Section 9.1: Inheritance Basics

1. What is code reuse? How does inheritance help achieve code reuse?

Code reuse is [illegible] repeatedly. Inheritance allows classes to share properties & methods

2. What is the difference between overloading and overriding a method?

Overriding [illegible] redefines it, overloading is another definition of the same [illegible]

3. Which of the following is the correct syntax to indicate that class `A` is a subclass of `B`?

a. `public class B extends A {`

b. `public class A : super B {`

c. `public A(super B) {`

[✓ d.] `public class A extends B {`

e. `public A implements B {`

4. Consider the following classes:

```
public class Vehicle {...}

public class Car extends Vehicle {...}

public class SUV extends Car {...}
```

Which of the following are legal statements?

[✓ a.] `Vehicle v = new Car();`
[✓ b.] `Vehicle v = new SUV();`
[✓ c.] `Car c = new SUV();`
[✓ d.] `SUV s = new SUV();`
e. `SUV s = new Car();`
f. `Car c = new Vehicle();`

Section 9.2: Interacting with the Superclass

5. Explain the difference between the `this` keyword and the `super` keyword. When should each be used?

Super references the superclass, this represents the object it is called from.

6. For the next three problems, consider the following class:

```
// Represents a university student.
public class Student {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```
public UndergraduateStudent (String name) {
    super (name, 18);
    year = 18;
}
public void setAge(int age) {
    this.setAge = age;
    year++;
}
```

Also consider the following partial implementation of a subclass of `Student` to represent undergraduate students at a university:

```
public class UndergraduateStudent extends Student {
    private int year;
    ...
}
```

Can the code in the `UndergraduateStudent` class access the `name` and `age` fields it inherits from `Student`? Can it call the `setAge` method?

Yes, no

7. Write a constructor for the `UndergraduateStudent` class that accepts a name as a parameter and initializes the `UnderGraduateStudent`'s state with that name, an `age` value of `18`, and a `year` value of `0`.

8. Write a version of the `setAge` method in the `UndergraduateStudent` class that not only sets the `age` but also increments the `year` field's value by one.

9. Consider the following two automobile classes:

```
public class Car {
    public void m1() {
        System.out.println("car 1");
    }

    public void m2() {
        System.out.println("car 2");
    }

    public String toString() {
        return "vroom";
    }
}
public class Truck extends Car {
    public void m1() {
        System.out.println("truck 1");
    }
}
```

Given the following declared variables, what is the output from the following statements?

```
Car mycar = new Car();
Truck mytruck = new Truck();

System.out.println(mycar);
mycar.m1();
mycar.m2();
System.out.println(mytruck);
mytruck.m1();
mytruck.m2();
```

vroom
car 1
car 2
vroom
truck 1
car 2

10. Suppose the Truck code from the previous problem changes to the following:

```
public class Truck extends Car {
    public void m1() {
        System.out.println("truck 1");
    }

    public void m2() {
        super.m1();
    }

    public String toString() {
        return super.toString() + super.toString();
    }
}
```

Using the same variables from the previous problem, what is the output from the following statements?

```
System.out.println(mytruck);        vroomvroom
mytruck.m1();                       truck 1
mytruck.m2();                       car 1
```

Section 9.3: Polymorphism

11. Using the A, B, C, and D classes from this section, what is the output of the following code fragment?

```
public static void main(String[] args) {
    A[] elements = {new B(), new D(), new A(), new C()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        System.out.println(elements[i]);
        elements[i].method1();
        System.out.println();
    }
}
```

```
B2
A
A1

D2
C
C1

A2
A
A1

A2
C
C1
```

12. Assume that the following classes have been defined:

```
public class Flute extends Blue {
    public void method2() {
        System.out.println("flute 2");
    }

    public String toString() {
        return "flute";
    }
}
```

```
public class Blue extends Moo {
    public void method1() {
        System.out.println("blue 1");
    }
}
```

```
public class Shoe extends Flute {
    public void method1() {
        System.out.println("shoe 1");
    }
}
```

```
public class Moo {
    public void method1() {
        System.out.println("moo 1");
    }
```

flute
~~Shoe~~ Shoe 1
flute 2

flute
blue 1
flute 2

moo
moo 1
moo 2

moo
blue 1
moo 2

```
    public void method2() {
        System.out.println("moo 2");
    }

    public String toString() {
        return "moo";
    }
}
```

What is the output produced by the following code fragment?

```
public static void main(String[] args) {
    Moo[] elements = {new Shoe(), new Flute(), new Moo(), new Blue()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

moo 2
blue 1
moo

moo 2
moo 1
moo

13. Using the classes from the previous problem, write the output that is produced by the following code fragment.

```
public static void main(String[] args) {
    Moo[] elements = {new Blue(), new Moo(), new Shoe(), new Flute()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        elements[i].method1();
        System.out.println(elements[i]);
        System.out.println();
    }
}
```

flute 2
shoe 1
flute

flute 2
blue 1
flute

14. Assume that the following classes have been defined:

```
public class Mammal extends SeaCreature {
    public void method1() {
        System.out.println("warm-blooded");
    }
}
```

```
public class SeaCreature {
    public void method1() {
        System.out.println("creature 1");
    }
    public void method2() {
        System.out.println("creature 2");
    }

    public String toString() {
        return "ocean-dwelling";
    }
}
```

```
public class Whale extends Mammal {
    public void method1() {
        System.out.println("spout");
    }

    public String toString() {
        return "BIG!";
    }
}
```

```
public class Squid extends SeaCreature {
    public void method2() {
        System.out.println("tentacles");
    }

    public String toString() {
        return "squid";
    }
}
```

Suqid
Creature 1
tentacles

BIG!
spout
creature 2

ocean-dwelling
Creature 1
creature 2

ocean-dwelling
warm-blooded
creature 2

What output is produced by the following code fragment?

```
public static void main(String[] args) {
    SeaCreature[] elements = {new Squid(), new Whale(),
                              new SeaCreature(), new Mammal()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

15. Using the classes from the previous problem, write the output that is produced by the following code fragment:

```
public static void main(String[] args) {
    SeaCreature[] elements = {new SeaCreature(),
                              new Squid(), new Mammal(), new Whale()};
```

```
        for (int i = 0; i < elements.length; i++) {
            elements[i].method2();
            System.out.println(elements[i]);
            elements[i].method1();
            System.out.println();
        }
    }
```

16. Assume that the following classes have been defined:

```
public class Bay extends Lake {
    public void method1() {
        System.out.print("Bay 1 ");
        super.method2();
    }
    public void method2() {
        System.out.print("Bay 2 ");
    }
}

public class Pond {
    public void method1() {
        System.out.print("Pond 1 ");
    }
    public void method2() {
        System.out.print("Pond 2 ");
    }
    public void method3() {
        System.out.print("Pond 3 ");
    }
}

public class Ocean extends Bay {
    public void method2() {
        System.out.print("Ocean 2 ");
    }
}

public class Lake extends Pond {
    public void method3() {
        System.out.print("Lake 3 ");
        method2();
    }
}
```

What output is produced by the following code fragment?

```
Pond[] ponds = {new Ocean(), new Pond(), new Lake(), new Bay()};
for (Pond p : ponds) {
```

creature 2S
ocean-dwelling
creature 1

vertebres
sai[illegible]
creature 1

creature 2
ocean-dwelling
verm-bloozee

creature 2
ocean-delling
BIG!

Bay 1
Ocean 2
Lake 3

Pond 1
Pond 2
Pond 3

Pond 1
Pond 2
Lake 3

Bay 1
Bay 2
Lake 3

```
        p.method1();
        System.out.println();
        p.method2();
        System.out.println();
        p.method3();
        System.out.println("\n");
    }
```

17. Suppose that the following variables referring to the classes from the previous problem are declared:

```
Pond var1 = new Bay();
Object var2 = new Ocean();
```

None

Which of the following statements produce compiler errors? For the statements that do not produce errors, what is the output of each statement?

```
((Lake) var1).method1();   // Bay 1
((Bay) var1).method1();    // Bay 1
((Pond) var2).method2();   // Ocean 2
((Lake) var2).method2();   // Ocean 2
((Ocean) var2).method3();  // Ocean 3
```

Section 9.4: Inheritance and Design

18. What is the difference between an is-a and a has-a relationship? How do you create a has-a relationship in your code?

Has-a is just a property/feild, is-a is inheritance

19. Imagine a `Rectangle` class with objects that represent two-dimensional rectangles. The `Rectangle` has `width` and `height` fields with appropriate accessors and mutators, as well as `getArea` and `getPerimeter` methods.

You would like to add a `Square` class into your system. Is it a good design to make `Square` a subclass of `Rectangle`? Why or why not? Yes, because a square is a rectagle

20. Imagine that you are going to write a program to play card games. Consider a design with a `Card` class and 52 subclasses, one for each of the unique playing cards (for example, `NineOfSpades` and `JackOfClubs`). Is this a good design? If so, why? If not, why not, and what might be a better design?

No, 52 classes is terrible design. Making each card a insatnce (or even making subclasses for each suit) would be better

21. In Section 9.2 we discussed adding functionality for dividend payments to the `Stock` class. Why was it preferable to create a `DividendStock` class rather than editing the `Stock` class and adding this feature directly to it?

This preserves backward compatability and doesn't effect existing code.

Section 9.5: Interfaces

22. What is the difference between implementing an interface and extending a class?

Implementing an interface doesn't share code

23. Consider the following interface and class:

```
public interface I {
    public void m1();
    public void m2();
}

public class C implements I {
    // code for class C
}
```

What must be true about the code for class `C` in order for that code to compile successfully?

It must contain public methods m1 and m2, which take no args and return void

24. What's wrong with the code for the following interface? What should be changed to make a valid interface for objects that have colors?

```
public interface Colored {
    private Color color;          ~~class~~
    public Color getColor();/*    Method body is illegal
        return color;
    }*/
}
```

I can't do these on paper

25. Modify the `Point` class from Chapter 8 so that it implements the `Colored` interface and `Points` have colors. (You may wish to create a `ColoredPoint` class that extends `Point`.)

26. Declare a method called `getSideCount` in the `Shape` interface that returns the number of sides that the shape has. Implement the method in all shape classes. A circle is defined to have 0 sides.

Section 9.6: Case Study: Financial Class Hierarchy

27. What is an abstract class? How is an abstract class like a normal class, and how does it differ? How is it like an interface?
It has the ability to declare methods & field values/bodies like a class, or empty (abstract) like an interface

28. Consider the following abstract class and its subclass. What state and behavior do you know for sure will be present in the subclass? How do you know?

It will have a private String[] data, public void arrange, are public void getElement (int i) with the body listed

```
public abstract class Ordered {
    private String[] data;
    public void getElement(int i) {
        return data[i];
    }
    public abstract void arrange();
}

public class OrderedByLength extends Ordered {
    ...
}
```

29. Consider writing a program to be used to manage a collection of movies. There are three kinds of movies in the collection: dramas, comedies, and documentaries. The collector would like to keep track of each movie's title, the name of its director, and the year the movie was made. Some operations are to be implemented for all movies, and there will also be special operations for each of the three different kinds of movies. How would you design the class(es) to represent this system of movies?
I would have an abstract class that implements global things, and each classes for class Genre, which extends the whichever [illegible]

Exercises

1. Write the class `Marketer` to accompany the other law firm classes described in this chapter. Marketers make $50,000 ($10,000 more than general employees) and have an additional method called `advertise` that prints `"Act now, while supplies last!"` Make sure to interact with the superclass as appropriate.

2. Write a class `Janitor` to accompany the other law firm classes described in this chapter. Janitors work twice as many hours per week as other employees (80 hours/week), they make $30,000 ($10,000 *less* than general employees), they get half as much vacation as other employees (only 5 days), and they have an additional method `clean` that prints `"Workin' for the man."` Make sure to interact with the superclass as appropriate.