

---

# **Data Structures and Algorithms in Java™**

**Sixth Edition**

**Michael T. Goodrich**

Department of Computer Science  
University of California, Irvine

**Roberto Tamassia**

Department of Computer Science  
Brown University

**Michael H. Goldwasser**

Department of Mathematics and Computer Science  
Saint Louis University

---

## **Instructor's Solutions Manual**

**WILEY**

## Chapter

# 2

## Object-Oriented Design

### Hints and Solutions

#### Reinforcement

**R-2.1) Hint** Think of applications that could cause a death if a computer failed.

**R-2.1) Solution** Air traffic control software, computer integrated surgery applications, and flight navigation systems.

**R-2.2) Hint** Consider an application that is expected to change over time, because of changing economics, politics, or technology.

**R-2.3) Hint** Consider the File or Window menus.

**R-2.4) Hint** You can make the change and test the code.

**R-2.4) Solution** The problem is that when a \$5 penalty is assessed, presumably because of an attempt to go over the credit limit, the call `charge(5)` recursively invokes the `PredatoryCreditCard.charge` method; since that fee could again be an attempt at violating the credit limit, it too may fail, leading to an infinite recursion.

**R-2.5) Hint** You can make the change and test the code.

**R-2.5) Solution** The goal is to assess a \$5 charge as a penalty, yet that charge may be refused by the call to `super.charge(5)` if the user is already at or near the credit limit.

**R-2.6) Hint** Your program should output 42, which Douglas Adams considers to be the answer to the ultimate question of life, universe, and everything.

**R-2.6) Solution**

```
public static void main(String[ ] args) {  
    FibonacciProgression fp = new FibonacciProgression(2,2);  
    for (int j=0; j < 7; j++)  
        fp.nextValue(); // ignore the first 7 values  
    System.out.println(fp.nextValue());  
}
```

**R-2.7) Hint** A long value can be no larger than  $2^{63} - 1$ .

**R-2.7) Solution**  $2^{56}$  calls to `nextValue` will end on the value  $2^{63}$ . Since the maximum positive value of a long is  $2^{63} - 1$ ,  $2^{56} - 1$  calls to `nextValue` can be made before a long-integer overflow.

**R-2.8) Hint** Code up an example and see what the compiler says.

**R-2.9) Hint** Think about what happens when a new instance of class `Z` is created and when methods of class `Z` are called.

**R-2.9) Solution** There are two immediate inefficiencies: (1) the chaining of constructors implies a potentially long set of method calls any time an instance of a deep class, `Z`, is created, and (2) the dynamic dispatch algorithm for determining which version of a certain method to use could end up looking through a large number of classes before it finds the right one to use.

**R-2.10) Hint** Think about code reuse.

**R-2.10) Solution** Whenever a large number of classes all extend from a single class, it is likely that you are missing out on potential code reuse from similar methods in different classes. There is likely some factoring of methods into common classes that could be done in this case, which would save programmer time and maintenance time, by eliminating duplicated code.

**R-2.11) Hint** Review the section about casting in an inheritance hierarchy, and recall that an object behaves according to what it actually is, not what it is called.

**R-2.11) Solution**

Read it.

Ship it.

Buy it.

Read it.

Box it.

Read it.

**R-2.12) Hint** Review the definition of inheritance diagram, and begin your drawing with `Object` as the highest box.

**R-2.13) Hint** Casting in an inheritance relationship can only move up or down the hierarchy.

**R-2.13) Solution** No, `d` is referring to a `Equestrian` object that is not not also of type `Racer`. Casting in an inheritance relationship can only move up or down the hierarchy, not “sideways.”

**R-2.14) Hint** You don’t need to declare the array, just show how to use an exception try-catch block to reference it.

**R-2.14) Solution**

```
try {
    System.out.println(array[i]);
}
catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index " + e.getMessage()
        + " out of bounds.");
}
```

**R-2.15) Hint** Reread the section on throwing exceptions.

**R-2.15) Solution**

```
public void makePayment(double amount) {
    if (amount < 0)
        throw new IllegalArgumentException("Amount must be nonnegative");
    balance -= amount;
}
```

---

## Creativity

**C-2.16) Hint** Create a separate class for each major behavior.

**C-2.17) Hint** Try to use variables and conditions that are impossible, but the dependence on their values requires logical reasoning that the compiler writers did not build into their compiler.

**C-2.18) Hint** You will need to maintain some additional state information.

**C-2.18) Solution**

```

private int chargesThisMonth = 0;           // new instance variable

public void processMonth() {
    chargesThisMonth = 0;                     // reset
    ...
}

public boolean charge(double price) {
    boolean isSuccess = super.charge(price); // call inherited method
    if (!isSuccess)
        balance += 5;                         // assess a $5 penalty
    chargesThisMonth++;
    if (chargesThisMonth > 10)
        balance += 1;                         // assess a $1 fee
    return isSuccess;
}

```

**C-2.19) Hint** Keep track of how much has been paid during the current month.

**C-2.20) Hint** Don't forget you can use `getBalance()` as well.

### **C-2.20) Solution**

```

public void processMonth() {
    if (getBalance() > 0) {
        double monthlyFactor = Math.pow(1 + apr, 1.0/12);
        setBalance(monthlyFactor * getBalance());
    }
}

```

**C-2.21) Hint** You need to use the `super` keyword in *B* and *C*.

### **C-2.21) Solution**

```

public class A {
    int x = 1;
    public void setIt(int y) { x = y; }
    public int getIt() { return x; }
}

public class B extends A {
    int x = 2;
    public void setIt (int y) { x = y; }
    public int getIt() { return x; }
    public void superSetIt (int y) { super.x = y; }
    public int superGetIt() { return super.x; }
}

public class C extends B {
    int x = 3;
    public void setIt (int y) { x = y; }
    public int getIt() { return x; }
    public void superSetIt (int y) { super.x = y; }
    public int superGetIt() { return super.x; }
    public void superDuperSetIt(int y) { super.superSetIt(y); }
    public int superDuperGetIt() { return super.superGetIt(); }
    public static void main(String[] args) {
        C c = new C();
        System.out.println("C's is " + c.getIt());
        System.out.println("B's is " + c.superGetIt());
        System.out.println("A's is " + c.superDuperGetIt());
        c.superDuperSetIt(4);
        System.out.println("C's is " + c.getIt());
        System.out.println("B's is " + c.superGetIt());
        System.out.println("A's is " + c.superDuperGetIt());
    }
}

```

**C-2.22) Hint** Recall the rule about inheritance in Java.

**C-2.22) Solution** Inheritance in Java allows specialized classes to be built from generic classes. Because of this progression from generic to specialized in the class hierarchy, there can never be a circular pattern of inheritance. In other words, there cannot be a superclass *A* and derived classes *B* and *C* such that *B* extends *A*, then *C* extends *B*, and finally *A* extends *C*. Such a cycle is impossible because *A* is the generic superclass from which *C* is eventually extended, thus it is impossible from *A* to extend *C*,

for this would mean *A* is extending itself. Therefore, there can never occur a circular relationship which would cause an infinite loop in the dynamic dispatch.

**C-2.23) Hint** Can you determine a missing entry of a Fibonacci sequence if you are given the number immediate before it and after it?

**C-2.23) Solution**

```
protected void advance() {
    current += prev;
    prev = current - prev;
}
```

**C-2.24) Hint** Use the code from the website as a starting point.

**C-2.24) Solution**

```
public class AbsoluteProgression extends Progression {

    protected long prev;

    public AbsoluteProgression() { this(2,200); }

    public AbsoluteProgression(long first, long second) {
        super(first);
        prev = first-second;    // as second = Math.abs(first-prev)
    }

    protected void advance() {
        long next = Math.abs(current-prev);
        prev = current;
        current = next;
    }

}
```

**C-2.25) Hint** Replace each use of type **long** with the generic parameter type *T*.

**C-2.26) Hint** Use the `sqrt` method in the `java.lang.Math` class.

**C-2.27) Hint** Go to the `java.com` website to review the `BigInteger` class.

**C-2.28) Hint** Use three different classes, for each of the actors, and provide methods that perform their various tasks, as well as a simulator engine that performs the periodic operations.

**C-2.29) Hint** If you have not had calculus, you can look up the formula for the first derivative of a polynomial on the Internet.

## Projects

**P-2.30) Hint** You don't have to use GUI constructs; simple text output is sufficient, say, using X's to indicate the values to print for each bar (and printing them sideways).

**P-2.31) Hint** When a fish dies, set its array cell back to **null**.

**P-2.32) Hint** Use random number generation for the strength field.

**P-2.33) Hint** Create a separate class for each major behavior. Find the available books on the Internet, but be sure they have expired copyrights.

**P-2.34) Hint** Lookup the formulas for area and perimeter on the Internet.

**P-2.35) Hint** You need some way of telling when you have seen the same word you have before. Feel free to just search through your array of words to do this here.

**P-2.36) Hint** While not always optimal, you can design your algorithm so that it always returns the largest coin possible until the value of the change is met.