**Review Questions**

1.

Which method is invoked in a particular class when a method definition is overridden in several classes that are part of an inheritance hierarchy? Answer the question for the case in which the class has a definition for the method and also for the case where it doesn't.

If the class has a definition for the method, then that method is invoked. If the class does not have a definition for the method, but a class higher up in the hierarchy does, then that is the one invoked. The search for a method to execute moves up thru the hierarchy so the method in the class closest to the class of the object on which the method was called is the one invoked.

2.

Explain how assignments can be made within a class hierarchy and the role of casting in a class hierarchy. What is strong typing? Why is it an important language feature?

The target of the assignment must be of the same type or a super-type (i.e., a class at the same level or higher in the hierarchy) as the value being assigned.
Casting allows the compiler to view an object as being of the sub-class. At run-time the object is checked to see if it is of the specified sub-type. If it is not, a `ClassCastException` is thrown; if it is, operations appropriate to that subtype can be performed on the object.
Strong typing means that compile-time and run-time checks are made to ensure that assignments meet these criteria. It is an important language feature because it prevents some errors.

3.

If Java encounters a method call of the following form:
`superclassVar.methodName()`
where `superclassVar` is a variable of a superclass that references an object whose type is a subclass, what is necessary for this statement to compile? During run time, will method `methodName` from the class that is the type of `superclassVar` always be invoked, or is it possible that a different method `methodName` will be invoked? Explain your answer.

For the expression `superclassVar.methodName()` to compile, `methodName` must be declared in `superclass` or a super class of `superclass`. During run time the `methodName` method in the class of the object referenced by `superclassVar` will be invoked if that class overrides the definition in superclass.

4.

Assume the situation in Question 3, but method `methodName` is not defined in the class that is the type of `superclassVar`, although it is defined in the subclass type. Rewrite the method call so that it will compile.

```
((subclass)superClassVar).methodName()
```

5.

Explain the process of initializing an object that is a subclass type in the subclass constructor. What part of the object must be initialized first? How is this done?

Space is allocated for the object. Then the superclass constructor is invoked, followed by the subclass constructor. Therefore, the first statement of the subclass constructor must be a call to the superclass constructor. Where the superclass has only a default constructor, the compiler inserts a call to the superclass constructor as the first statement of the subclass constructor.

6.

What is default or package visibility?

Members declared within a class with no access modifier (e.g., `public`, `protected`, or `private`) are given package visibility. These members are visible to any other member of the package, but not outside the package.

7.

Indicate what kind of exception each of the following errors would cause. Indicate whether each error is a checked or an unchecked exception.

a. Attempting to create a `Scanner` for a file that does not exist

`FileNotFoundException`

b. Attempting to call a method on a variable that has not been initialized

`NullPointerException`

c. Using `-1` as an array index

`ArrayIndexOutOfBoundsException`

8.

Discuss when abstract classes are used. How do they differ from actual classes and from interfaces?

An abstract class is a class that is declared abstract and has one or more declarations of an abstract method. Abstract classes can contain fields and non-abstract methods. Abstract classes can provide common fields and common methods within a class hierarchy and reserve certain implementation details to sub-classes. Variables of the abstract class type may be declared, but objects of the abstract class cannot. Interfaces are like abstract classes in that they declare abstract methods. Interfaces cannot contain variable fields. Interfaces can define constant fields. Interfaces can provide default implementation for abstract methods and can provide static methods. Variables of the interface type can be declared, but objects cannot.

## 9.

What is the advantage of specifying an ADT as an interface instead of just going ahead and implementing it as a class?

Specifying an ADT as an interface allows for alternative implementations. It makes the abstract data type abstract. Implementing an ADT as an implementing class would no longer be abstract.

## 10.

Define an interface to specify an ADT `Money` that has methods for arithmetic operations (addition, subtraction, multiplication, and division) on real numbers having exactly two digits to the right of the decimal point, as well as methods for representing a `Money` object as a string and as a real number. Also, include methods `equals` and `compareTo` for this ADT.

```
/**
 *
  *  ADT  Money  that  has  methods  for  arithmetic  operations  (addition,
subtraction,
 * multiplication, and division) on real numbers having exactly two digits to
 * the right of the decimal point. Implementing classes should provide a
 * constructor that takes a double value.
 */
public interface Money extends Comparable<Money> {

    /**
     * Add an amount of Money to this Money object and return a new Money
     * object containing the sum.
     * @param other The other Money object
     * @return A new Money object containing the sum.
     */
    Money add(Money other);

    /**
     * Subtract an amount of Money from this Money object and return a
```

```
     * new Money
     * object containing the sum.
     * @param other The other Money object
     * @return A new Money object containing the difference.
     */
    Money sub(Money other);

    /**
     * Multiply this Money object by the factor and return a new Money object
     * containing the resulting product.
     * @param factor The amount the money should be multiplied by
     * @return The resulting product.
     */
    Money mul(double factor);

    /**
     * Divide this Money object by the factor and return a nee Money object
     * containing the resulting quotient.
     * @param factor The amount the money should be divided by
     * @return The resulting quotient.
     */
    Money div(double factor);

    /**
     * Return a string representation of this Money object.The result should
     * be rounded to two decimal places,
     * @return A String representation of the object.
     */
    @Override
    String toString();

    /**
     * Return the double value of this Money object rounded to two decimal
     * places.
     * @return A double value of this object.
     */
    double toDouble();

    /**
     * Determine of two Money objects are equal. Money objects are considered
     * equal if their values are equal after being rounded to two decimal
     * places.
     * @param o The other object
     * @return true if the other is equal to this after being rounded to two
     * decimal places.
     */
    @Override
    boolean equals(Object o);

    /**
     * Compare this Money object to another Money object.
     * @param other The other money object
      * @return -1 if this is less than other, +1 if this is greater than
other
```

```
     * and 0 if this is equal to other.
     */
    @Override
    public int compareTo(Money other);

}
```

## 11.

Answer Review Question 10 for an ADT `Complex` that has methods for arithmetic operations on a complex number (a number with a real and an imaginary part). Assume that the same operations (+, –, *, /) are supported. Also, provide methods `toString` and `equals` for the ADT `Complex`.

```
/**
 * An interface to define the ADT Complex. Complex that has methods
 * for arithmetic operations on a complex number (a number with a real
 * and an imaginary part).
  *  The  operations  (+,  –,  *,  /)  are  supported.  Also,  provides  methods
toString,
 * and equals for the ADT Complex.
 * The implementation must provide a constructor that takes two double values
 * that specify the real and imaginary parts.
 * @author Koffman & Wolfgang
 */
public interface Complex {

    /**
     * Return the real component of this Complex value.
     */
    double re();

    /**
     * Return the imaginary component of this Complex value.
     */
    double im();

    /**
     * Return the magnitude of this Complex value.
     */
    double ro();

    /**
     * Return the angle of this Complex value.
     */
    double theta();

    /**
     * Return the sum of this complex value and the other Complex value.
     */
    Complex add(Complex other);
```

```
    /**
       * Return the difference of this complex value an the other Complex
value.
     */
    Complex sub(Complex other);

    /**
     * Return the product of this complex value an the other Complex value.
     */
    Complex mul(Complex other);

    /**
     * Return the quotient of this complex value and the other Complex value.
     */
    Complex div(Complex other);

    /**
     * Return a String representation of this complex value.
     */
    @Override
    String toString();

    /**
     * Determine if two Complex values are equal.
     * @param o
     */
    @Override
    boolean equals(Object o);
}
```

## 12.

Like a rectangle, a parallelogram has opposite sides that are parallel, but it has a corner angle, *theta*, that is less than 90 degrees. Discuss how you would add parallelograms to the class hierarchy for geometric shapes (see Figure 1.10). Write a definition for class `Parallelogram`.

Define fields to contain the length of the sides and the angle. Also add Parallelogram to the choices in the ComputeAreaAndPerim main method.

```
import java.util.Scanner;

/**
 * Definition of the class Parallelogram
 * @author Koffman & Wolfgang
 */
public class Parallelogram extends AbstrtactShape {
    // Data fields

    /** One side */
    private double a;
```

```
/** The other side */
private double b;
/** The angle between a and b */
private double theta;

/** Default constructor */
public Parallelogram) {
    super("Parallelogram");
}

/**
 * Construct a Parallelogram with the specified values
 *
 * @param a The length of one side
 * @param b The length of the other side
 * @param theta The angle between a and b in radians between 0 and PI/4
 */
public Parallelogram(double a, double b, double theta) {
    super("Parallelogram");
    this.a = a;
    this.b = b;
    this.theta = theta;
}

/**
 * Compute the area
 * @return the area
 */
@Override
public double computeArea() {
    return a*Math.sin(theta) * b;
}

/**
 * Compute the perimeter
 * @return the perimeter
 */
@Override
public double computePerimeter() {
    return 2*(a + b);
}

/**
 * Read the attributes of the paralogram.
 */
@Override
public void readShapeData() {
    Scanner in = new Scanner(System.in);
    System.out.println("Enter the one side of the Parallelogram ");
```

```
        a = in.nextDouble();
        System.out.println("Enter the other side of the Parallelogram");
        b = in.nextDouble();
        System.out.println("Enter the angle between sides a and b "
            + "in degrees");
        double angle = in.nextDouble();
        if (0 < angle && angle <= 90.0) {
            theta = Math.toRadians(angle);
        } else {
            throw new IllegalArgumentException(angle +
                    " must be greater than 0 and less than or " +
                    "equal to 90");
        }
    }


    /**
     * Get the width of the bounding rectangle
     * @return the width
     */
    @Override
    public double getWidth() {
        return b + a*Math.cos(theta);
    }

    /**
     * Get the height of the bounding rectangle
     * @return the height
     */
    @Override
    public double getHeight() {
        return a*Math.sin(theta);
    }
}
```