MULTIPLE-CHOICE QUESTIONS ON INHERITANCE AND POLYMORPHISM

Questions 1–10 refer to the BankAccount, SavingsAccount, and CheckingAccount classes defined below:

```
public class BankAccount
{
    private double balance;

    public BankAccount()
    { balance = 0; }

    public BankAccount(double acctBalance)
    { balance = acctBalance; }

    public void deposit(double amount)
    { balance += amount; }

    public void withdraw(double amount)
    { balance -= amount; }

    public double getBalance()
    { return balance; }
}

public class SavingsAccount extends BankAccount
{
    private double interestRate;

    public SavingsAccount()
    { /* implementation not shown */ }

    public SavingsAccount(double acctBalance, double rate)
    { /* implementation not shown */ }

    public void addInterest()    //Add interest to balance
    { /* implementation not shown */ }
}

public class CheckingAccount extends BankAccount
{
    private static final double FEE = 2.0;
    private static final double MIN_BALANCE = 50.0;

    public CheckingAccount(double acctBalance)
    { /* implementation not shown */ }

    /** FEE of $2 deducted if withdrawal leaves balance less
     *  than MIN_BALANCE. Allows for negative balance. */
    public void withdraw(double amount)
    { /* implementation not shown */ }
}
```

5. Which is correct implementation code for the `withdraw` method in the `CheckingAccount` class?

(A)
```
super.withdraw(amount);
if (balance < MIN_BALANCE)
    super.withdraw(FEE);
```

(B)
```
withdraw(amount);
if (balance < MIN_BALANCE)
    withdraw(FEE);
```

(C)
```
super.withdraw(amount);
if (getBalance() < MIN_BALANCE)
    super.withdraw(FEE);
```

(D)
```
withdraw(amount);
if (getBalance() < MIN_BALANCE)
    withdraw(FEE);
```

(E)
```
balance -= amount;
if (balance < MIN_BALANCE)
    balance -= FEE;
```

6. Redefining the `withdraw` method in the `CheckingAccount` class is an example of
   (A) method overloading.
   (B) method overriding.
   (C) downcasting.
   (D) dynamic binding (late binding).
   (E) static binding (early binding).

Use the following for Questions 7–9.
A program to test the `BankAccount`, `SavingsAccount`, and `CheckingAccount` classes ha
these declarations:

```
BankAccount b = new BankAccount(1400);
BankAccount s = new SavingsAccount(1000, 0.04);
BankAccount c = new CheckingAccount(500);
```

7. Which method call will cause an error?
   (A) `b.deposit(200);`
   (B) `s.withdraw(500);`
   (C) `c.withdraw(500);`
   (D) `s.deposit(10000);`
   (E) `s.addInterest();`

8. In order to test polymorphism, which method must be used in the program?
   (A) Either a `SavingsAccount` constructor or a `CheckingAccount` constructor
   (B) `addInterest`
   (C) `deposit`
   (D) `withdraw`
   (E) `getBalance`

11. Consider these class declarations:

```
public class Person
{
    ...
}

public class Teacher extends Person
{
    ...
}
```

Which is a true statement?

I `Teacher` inherits the constructors of `Person`.
II `Teacher` can add new methods and private instance variables.
III `Teacher` can override existing private methods of `Person`.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

12. Which statement about abstract classes and interfaces is *false*?
(A) An interface cannot implement any non-default instance methods, whereas an abstract class can.
(B) A class can implement many interfaces but can have only one superclass.
(C) An unlimited number of unrelated classes can implement the same interface.
(D) It is not possible to construct either an abstract class object or an interface object.
(E) All of the methods in both an abstract class and an interface are public.

Refer to the classes below for Questions 14 and 15.

```
public class ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* implementation of method1 */ }
}

public class ClassB extends ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* different implementation from method1 in ClassA*/ }


    public void method2()
    { /* implementation of method2 */ }
}
```

14. The `method1` method in `ClassB` is an example of
    (A) method overloading.
    (B) method overriding.
    (C) polymorphism.
    (D) information hiding.
    (E) procedural abstraction.

15. Consider the following declarations in a client class.

```
ClassA ob1 = new ClassA();
ClassA ob2 = new ClassB();
```

Which of the following method calls will cause an error?

I `ob1.method2();`

II `ob2.method2();`

III `((ClassB) ob1).method2();`

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) I, II, and III

16. A program that tests these classes has the following declarations and assignments:

```
Solid s1, s2, s3, s4;
s1 = new Solid("blob");
s2 = new Sphere("sphere", 3.8);
s3 = new RectangularPrism("box", 2, 4, 6.5);
s4 = null;
```

How many of the above lines of code are incorrect?

(A) 0
(B) 1
(C) 2
(D) 3
(E) 4

17. Which is *false*?

(A) If a program has several objects declared as type `Solid`, the decision about which `volume` method to call will be resolved at run time.

(B) If the `Solid` class were modified to provide a default implementation for the `volume` method, it would no longer need to be an abstract class.

(C) If the `Sphere` and `RectangularPrism` classes failed to provide an implementation for the `volume` method, they would need to be declared as abstract classes.

(D) The fact that there is no reasonable default implementation for the `volume` method in the `Solid` class suggests that it should be an abstract method.

(E) Since `Solid` is abstract and its subclasses are nonabstract, polymorphism no longer applies when these classes are used in a program.

19. Consider the `Computable` interface below for performing simple calculator operations:

```
public interface Computable
{
    /** Return this Object + y. */
    Object add(Object y);

    /** Return this Object - y. */
    Object subtract(Object y);

    /** Return this Object * y. */
    Object multiply(Object y);
}
```

Which of the following is the *least* suitable class for implementing `Computable`?

```
(A) LargeInteger      //integers with 100 digits or more
(B) Fraction          //implemented with numerator and
                      //denominator of type int
(C) IrrationalNumber  //nonrepeating, nonterminating decimal
(D) Length            //implemented with different units, such
                      //as inches, centimeters, etc.
(E) BankAccount       //implemented with balance
```

Refer to the `Player` interface shown below for Questions 20–23.

```
public interface Player
{
    /** Return an integer that represents a move in a game. */
    int getMove();

    /** Display the status of the game for this Player after
     *  implementing the next move. */
    void updateDisplay();
}
```

20. `HumanPlayer` is a class that implements the `Player` interface. Another class, `SmartPlayer`, is a subclass of `HumanPlayer`. Which statement is *false*?
    (A) `SmartPlayer` automatically implements the `Player` interface.
    (B) `HumanPlayer` must contain implementations of both the `updateDisplay` and `getMove` methods, or be declared as abstract.
    (C) It is not possible to declare a reference of type `Player`.
    (D) The `SmartPlayer` class can override the methods `updateDisplay` and `getMove` of the `HumanPlayer` class.
    (E) A method in a client program can have `Player` as a parameter type.

Consider these declarations for Questions 22 and 23:

```
public class HumanPlayer implements Player
{
    private String name;

    //Constructors not shown ...

    //Code to implement getMove and updateDisplay not shown ...

    public String getName()
    { /* implementation not shown */ }
}

public class ExpertPlayer extends HumanPlayer
{
    private int rating;

    //Constructors not shown ...

    public int compareTo(ExpertPlayer expert)
    { /* implementation not shown */ }
}
```

22. Which code segment in a client program will cause an error?

I
```
Player p1 = new HumanPlayer();
Player p2 = new ExpertPlayer();
int x1 = p1.getMove();
int x2 = p2.getMove();
```

II
```
int x;
Player c1 = new ExpertPlayer(/* correct parameter list */);
Player c2 = new ExpertPlayer(/* correct parameter list */);
if (c1.compareTo(c2) < 0)
    x = c1.getMove();
else
    x = c2.getMove();
```

III
```
int x;
HumanPlayer h1 = new HumanPlayer(/* correct parameter list */);
HumanPlayer h2 = new HumanPlayer(/* correct parameter list */);
if (h1.compareTo(h2) < 0)
    x = h1.getMove();
else
    x = h2.getMove();
```

(A) II only
(B) III only
(C) II and III only
(D) I, II, and III
(E) None

24. Which of the following classes is the least suitable candidate for containing a compareTo method?

(A)
```
public class Point
{
    private double x;
    private double y;

    //various methods follow
        ...
}
```

(B)
```
public class Name
{
    private String firstName;
    private String lastName;

    //various methods follow
        ...
}
```

(C)
```
public class Car
{
    private int modelNumber;
    private int year;
    private double price;

    //various methods follow
        ...
}
```

(D)
```
public class Student
{
    private String name;
    private double gpa;

    //various methods follow
        ...
}
```

(E)
```
public class Employee
{
    private String name;
    private int hireDate;
    private double salary;

    //various methods follow
        ...
}
```

26. Consider the `Orderable` interface and the partial implementation of the `Temperature` class defined below:

```
public interface Orderable
{
    /** Returns -1, 0, or 1 depending on whether the implicit
     *  object is less than, equal to, or greater than other.
     */
    int compareTo (Object other);
}

public class Temperature implements Orderable
{
    private String scale;
    private double degrees;

    //default constructor
    public Temperature ()
    { /* implementation not shown */ }

    //constructor
    public Temperature(String tempScale, double tempDegrees)
    { /* implementation not shown */ }

    public int compareTo(Object obj)
    { /* implementation not shown */ }

    public String toString()
    { /* implementation not shown */ }

    //Other methods are not shown.
}
```

Here is a program that finds the lowest of three temperatures:

```
public class TemperatureMain
{
    /** Find smaller of objects a and b. */
    public static Orderable min(Orderable a, Orderable b)
    {
        if (a.compareTo(b) < 0)
            return a;
        else
            return b;
    }

    /** Find smallest of objects a, b, and c. */
    public static Orderable minThree(Orderable a,
            Orderable b, Orderable c)
    {
        return min(min(a, b), c);
    }

    public static void main(String[] args)
    {
        /* code to test minThree method */
    }
```

27. A certain interface provided by a Java package contains just a single method:

```
public interface SomeName
{
    int method1(Object o);
}
```

A programmer adds some functionality to this interface by adding another abstract method to it, `method2`:

```
public interface SomeName
{
    int method1(Object ob1);
    void method2(Object ob2);
}
```

As a result of this addition, which of the following is true?

(A) A `ClassCastException` will occur if `ob1` and `ob2` are not compatible.

(B) All classes that implement the original `SomeName` interface will need to be rewritten because they no longer implement `SomeName`.

(C) A class that implements the original `SomeName` interface will need to modify its declaration as follows:

```
public class ClassName implements SomeName extends method2
{    ...
```

(D) `SomeName` will need to be changed to an abstract class and provide implementation code for `method2`, so that the original and upgraded versions of `SomeName` are compatible.

(E) Any new class that implements the upgraded version of `SomeName` will not compile.