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 myBalance;

    public BankAccount()
    { myBalance = 0; }

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

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

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

    public double getBalance()
    { return myBalance; }
}

public class SavingsAccount extends BankAccount
{
    private double myInterestRate;

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

    public SavingsAccount(double balance, 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 balance)
    { /* 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 */ }
}
```

1. Of the methods shown, how many different nonconstructor methods can be invoked by a `SavingsAccount` object?
   (A) 1
   (B) 2
   (C) 3
   (D) 4
   (E) 5

2. Which of the following correctly implements the default constructor of the `SavingsAccount` class?

   I
   ```
   myInterestRate = 0;
   super();
   ```

   II
   ```
   super();
   myInterestRate = 0;
   ```

   III
   ```
   super();
   ```

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

3. Which is a correct implementation of the constructor with parameters in the `SavingsAccount` class?

   (A)
   ```
   myBalance = balance;
   myInterestRate = rate;
   ```

   (B)
   ```
   getBalance() =  balance;
   myInterestRate = rate;
   ```

   (C)
   ```
   super();
   myInterestRate = rate;
   ```

   (D)
   ```
   super(balance);
   myInterestRate = rate;
   ```

   (E)
   ```
   super(balance, rate);
   ```

4. Which is a correct implementation of the `CheckingAccount` constructor?

   I
   ```
   super(balance);
   ```

   II
   ```
   super();
   deposit(balance);
   ```

   III
   ```
   deposit(balance);
   ```

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

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

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

(B)
```
withdraw(amount);
if (myBalance < 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)
```
myBalance -= amount;
if (myBalance < MIN_BALANCE)
    myBalance -= 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 h
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 progra
   (A) Either a `SavingsAccount` constructor or a `CheckingAccount` constru
   (B) `addInterest`
   (C) `deposit`
   (D) `withdraw`
   (E) `getBalance`

9. Which of the following will *not* cause a `ClassCastException` to be thrown?
   (A) `((SavingsAccount) b).addInterest();`
   (B) `((CheckingAccount) b).withdraw(200);`
   (C) `((CheckingAccount) c).deposit(800);`
   (D) `((CheckingAccount) s).withdraw(150);`
   (E) `((SavingsAccount) c).addInterest();`

10. A new method is added to the `BankAccount` class.

```
/* Transfer amount from this BankAccount to another BankAccount.
 * Precondition: myBalance > amount */
public void transfer(BankAccount another, double amount)
{
    withdraw(amount);
    another.deposit(amount);
}
```

A program has these declarations:

```
BankAccount b = new BankAccount(650);
SavingsAccount timsSavings = new SavingsAccount(1500, 0.03);
CheckingAccount daynasChecking = new CheckingAccount(2000);
```

Which of the following will transfer money from one account to another without error?

I `b.transfer(timsSavings, 50);`

II `timsSavings.transfer(daynasChecking, 30);`

III `daynasChecking.transfer(b, 55);`

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

11. Consider these class declarations:

```
public class Person
{
    ...
}

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

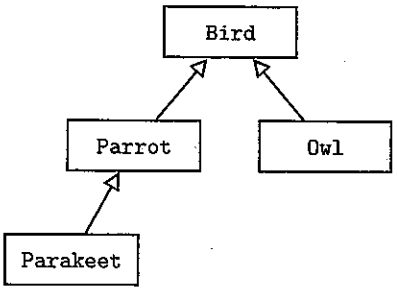
Which is a true statement?

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 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.

13. Consider the following hierarchy of classes:



A program is written to print data about various birds:

```
public class BirdStuff
{
    public static void printName(Bird b)
    { /* implementation not shown */ }

    public static void printBirdCall(Parrot p)
    { /* implementation not shown */ }

    //several more Bird methods

    public static void main(String[] args)
    {
        Bird bird1 = new Bird();
        Bird bird2 = new Parrot();
        Parrot parrot1 = new Parrot();
        Parrot parrot2 = new Parakeet();
        /* more code */
    }
}
```

Assuming that none of the given classes is abstract and all have default constructors, which of the following segments of `/* more code */` will *not* cause an error?

(A)
```
printName(parrot2);
printBirdCall((Parrot) bird2);
```

(B)
```
printName((Parrot) bird1);
printBirdCall(bird2);
```

(C)
```
printName(bird2);
printBirdCall(bird2);
```

(D)
```
printName((Parakeet) parrot1);
printBirdCall(parrot2);
```

(E)
```
printName((Owl) parrot2);
printBirdCall((Parakeet) parrot2);
```

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

ANSWER KEY

1. D
2. C
3. D
4. E
5. C
6. B
7. E
8. D
9. C
10. D
11. E
12. E
13. A
14. B
15. E
16. A
17. E
18. A
19. E
20. C
21. C
22. C
23. E
24. B
25. A
26. B
27. B
28. B

ANSWERS EXPLAINED

1. (D) The methods are `deposit`, `withdraw`, and `getBalance`, all inherited from the `BankAccount` class, plus `addInterest`, which was defined just for the class `SavingsAccount`.
2. (C) Implementation I fails because `super()` *must* be the first line of the implementation whenever it is used in a constructor. Implementation III may appear to be incorrect because it doesn't initialize `myInterestRate`. Since `myInterestRate`, however, is a primitive type—`double`—the compiler will provide a default initialization of `0`, which was required.
3. (D) First, the statement `super(balance)` initializes the inherited private variable `myBalance` as for the `BankAccount` superclass. Then the statement `myInterestRate = rate` initializes `myInterestRate`, which belongs uniquely to the `SavingsAccount` class. Choice E fails because `myInterestRate` does not belong to the `BankAccount` class and therefore cannot be initialized by a super method. Choice A is wrong because the `SavingsAccount` class cannot directly access the private instance variables of its superclass. Choice B assigns a value to an accessor method, which is meaningless. Choice C is incorrect because super invokes the *default* constructor of the superclass. This will cause `myBalance` of the `SavingsAccount` object to be initialized to `0`, rather than `balance`, the parameter value.
4. (E) The constructor must initialize the inherited instance variable `myBalance` to the value of the `balance` parameter. All three segments achieve this. Implementation I does it by invoking `super(balance)`, the constructor in the superclass. Implementation II first initializes `myBalance` to `0` by invoking the *default* constructor of the superclass. Then it calls the inherited `deposit` method of the superclass to add `balance` to the account. Implementation III works because `super()` is automatically called as the first line of the constructor code if there is no explicit call to `super`.

5. (C) First the `withdraw` method of the `BankAccount` superclass is used to withdraw `amount`. A prefix of `super` must be used to invoke this method, which eliminates choices B and D. Then the balance must be tested using the accessor method `getBalance`, which is inherited. You can't test `myBalance` directly since it is private to the `BankAccount` class. This eliminates choices A and E, and provides another reason for eliminating choice B.

6. (B) When a superclass method is redefined in a subclass, the process is called *method overriding*. Which method to call is determined at run time. This is called *dynamic binding* (p. 130). *Method overloading* is two or more methods with different signatures in the same class (p. 91). The compiler recognizes at compile time which method to call. This is *early binding*. The process of *downcasting* is unrelated to these principles (p. 131).

7. (E) The `addInterest` method is defined only in the `SavingsAccount` class. It therefore cannot be invoked by a `BankAccount` object. The error can be fixed by casting `s` to the correct type:

```
((SavingsAccount) s).addInterest();
```

The other method calls do not cause a problem because `withdraw` and `deposit` are both methods of the `BankAccount` class.

8. (D) The `withdraw` method is the only method that has one implementation in the superclass and a *different* implementation in a subclass. Polymorphism is the mechanism of selecting the correct method from the different possibilities in the class hierarchy. Notice that the `deposit` method, for example, is available to objects of all three bank account classes, but it's the *same* code in all three cases. So polymorphism isn't tested.

9. (C) You will get a `ClassCastException` whenever you try to cast an object to a class of which it is not an instance. Choice C is the only statement that doesn't attempt to do this. Look at the other choices: In choice A, `b` is not an instance of `SavingsAccount`. In choice B, `b` is not an instance of `CheckingAccount`. In choice D, `s` is not an instance of `CheckingAccount`. In choice E, `c` is not an instance of `SavingsAccount`.

10. (D) It is OK to use `timsSavings` and `daynasChecking` as parameters since each of these *is-a* `BankAccount` object. It is also OK for `timsSavings` and `daynasChecking` to call the `transfer` method (statements II and III), since they inherit this method from the `BankAccount` superclass.

11. (E) Statement I is false: A subclass must specify its own constructors. Otherwise the default constructor of the superclass will automatically be invoked. Note that statement III is true: It is OK to override private instance methods—they can even be declared public in the subclass implementation. What is *not* OK is to make the access more restrictive, for example, to override a public method and declare it private.

12. (E) All of the methods in an interface are by default public (the `public` keyword isn't needed). An abstract class can have both private and public methods.

13. (A) There are two quick tests you can do to find the answer to this question:

    (1) Test the *is-a* relationship, namely the parameter for `printName` *is-a* `Bird`? and the parameter for `printBirdCall` *is-a* `Parrot`?
    (2) A reference cannot be cast to something it's not an instance of.

Choice A passes both of these tests: `parrot2` *is-a* `Bird`, and `(Parrot) bird2` *is-a* `Parrot`. Also `bird2` is an instance of a `Parrot` (as you can see by looking at the right-hand side of the assignment), so the casting is correct. In choice B `printBirdCall(bird2)` is wrong because `bird2` *is-a* `Bird` and the `printBirdCall` method is expecting a `Parrot`. Therefore `bird2` must be downcast to a `Parrot`. Also, the method call `printName((Parrot) bird1)` fails because `bird1` is an instance of a `Bird` and therefore cannot be cast to a `Parrot`. In choice C, `printName(bird2)` is correct: `bird2` *is-a* `Bird`. However, `printBirdCall(bird2)` fails as already discussed. In choice D, `(Parakeet) parrot1` is an incorrect cast: `parrot1` is an instance of a `Parrot`. Note that `printBirdCall(parrot2)` is OK since `parrot2` *is-a* `Parrot`. In choice E, `(Owl) parrot2` is an incorrect cast: `parrot2` is an instance of `Parakeet`. Note that `printBirdCall((Parakeet) parrot2)` is correct: A `Parakeet` *is-a* `Parrot`, and `parrot2` is an instance of a `Parakeet`.

14. (B) Method overriding occurs whenever a method in a superclass is redefined in a subclass. Method overloading is a method in the same class that has the same name but different parameter types. Polymorphism is when the correct overridden method is called for a particular subclass object during run time. Information hiding is the use of `private` to restrict access. Procedural abstraction is the use of helper methods.

15. (E) All will cause an error!
    I: An object of a superclass does not have access to a new method of its subclass.
    II: `ob2` is declared to be of type `ClassA`, so a compile-time error will occur with a message indicating that there is no `method2` in `ClassA`. Casting `ob2` to `ClassB` would correct the problem.
    III: A `ClassCastException` will be thrown, since `ob1` is of type `ClassA`, and therefore cannot be cast to `ClassB`.

16. (A) The only incorrect line is `s1 = new Solid("blob");`: You can't create an instance of an abstract class. Abstract class references can, however, refer to objects of concrete (nonabstract) subclasses. Thus, the assignments for `s2` and `s3` are OK. Note that an abstract class reference can also be null, so the final assignment, though redundant, is correct.

17. (E) The point of having an abstract method is to postpone until run time the decision about which subclass version to call. This is what polymorphism is—calling the appropriate method at run time based on the type of the object.

18. (A) This is an example of polymorphism: The correct `volume` method is selected at run time. The parameter expected for `printVolume` is a `Solid` reference, which is what it gets in `main()`. The reference `sol` will refer either to a `Sphere` or a `RectangularPrism` object depending on the outcome of the coin flip. Since a `Sphere` is a `Solid` and a `RectangularPrism` is a `Solid`, there will be no type mismatch when these are the actual parameters in the `printVolume` method. (Note: The `Math.random` method is discussed in Chapter 4.)

19. (E) Each of choices A though D represent `Computable` objects: It makes sense to add, subtract, or multiply two large integers, two fractions, two irrational numbers, and two lengths. (One can multiply lengths to get an area, for example.) While it may make sense under certain circumstances to add or subtract two bank accounts, it does not make sense to multiply them!

20. (C) You can *declare a reference* of type `Player`. What you cannot do is *construct*