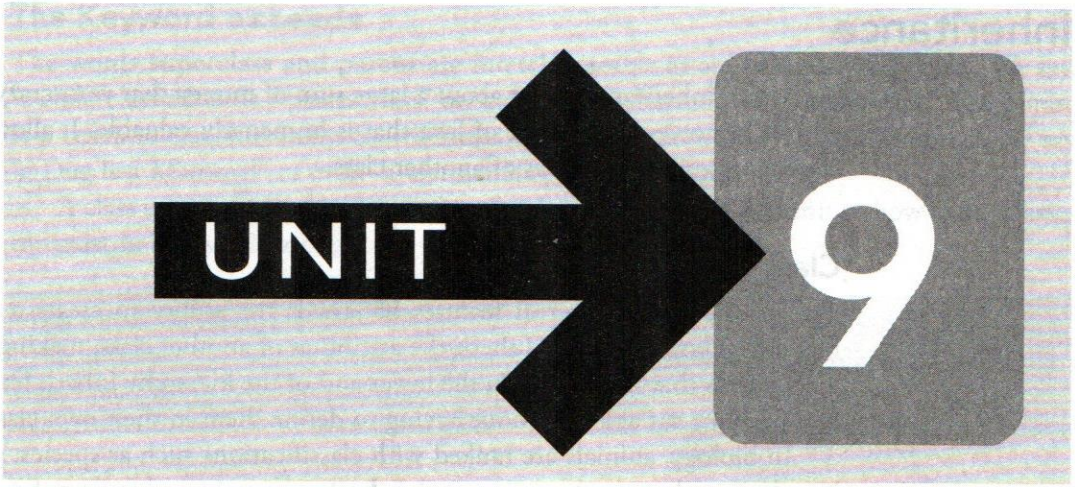


Inheritance

IN THIS UNIT

Summary: In order to take advantage of everything an object-oriented language has to offer, you must learn about its ability to handle inheritance. This unit explains how superclasses and subclasses are related within a class hierarchy. It also explains what polymorphism is and how objects within a class hierarchy may act independently of each other.

Key Ideas

- A class hierarchy describes the relationship between classes.
- Inheritance allows objects to use features that are not defined in their own class.
- A subclass extends a superclass (a child class extends a parent class).
- Child classes do not have direct access to their parent's private instance variables.
- A child class can override the parent class methods by simply redefining the method.
- Polymorphism is what happens when objects from child classes are allowed to act in a different way than objects from their parent classes.
- The `Object` class is the parent of all classes (the mother of all classes).

Inheritance

Sorry, this inheritance is not about a large sum of money that your crazy, rich uncle has left you. It is, however, a feature of Java that is immensely valuable. It allows objects from one class to inherit the features of another class.

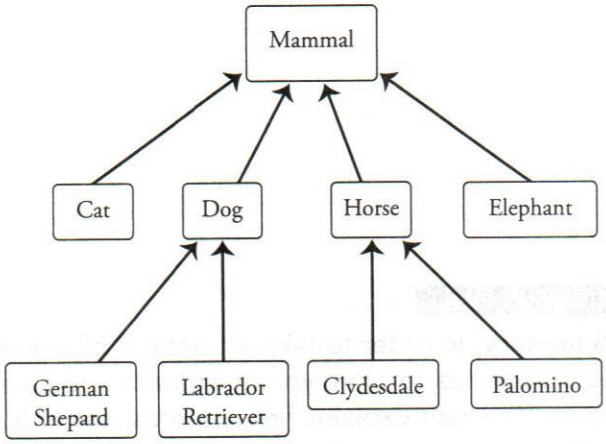
Class Hierarchy

One of the most powerful features of Java is the ability to create a **class hierarchy**. A hierarchy allows a class to inherit the attributes of another class, making the code **reusable**. This means that the classes on the lower end of the hierarchy inherit features (like methods and instance variables) without having to define them in their own class.

In biology, animals are ranked with classifications such as species, genus, family, class, kingdom, and so on. Any given rank **inherits** the traits and features from the rank above it.

Example

Demonstrate the hierarchy of cats, dogs, horses, and elephants. Note that all German shepherds are dogs, and all dogs are mammals, so all German shepherds are mammals.



Parent Versus Child Classes (Superclass Versus Subclass)

So what does this have to do with computer programming? Well, in Java, a class has the ability to extend another class. That is, a **child** class (**subclass**) has the ability to extend a **parent** class (**superclass**). When a child class extends a parent class, it inherits the instance variables and methods of the parent class. This means that even though the child class doesn't define these instance variables or methods on its own, it has them. This is a powerful way to reuse code without having to rewrite it.

Note: The methods from a parent class are inherited and can be used by the child class. However, the child class gets its *own set* of instance variables that are separate from the parent class. The instance variables for the parent and the instance variables for the child are two different sets of instance variables. The child class can only access the private instance variables of the parent by using the accessor and mutator methods from the parent class.

Something for Nothing
Child classes inherit the methods of their parent class. This means that the child objects automatically get these methods without having to define them for themselves.

The Keyword `extends`

The words **superclass** and **parent** are interchangeable as are **subclass** and **child**. We say, "a child class **extends** a parent class" or "a subclass is derived from a superclass." The phrase **is-a** refers to class hierarchy. So when we refer to classes such as Dog or Mammal, we say "a Dog is-a Mammal" or even "a GermanShepherd is-a Mammal."

A class can extend *at most one other class*; however, there is no limit to how many classes can exist in a class hierarchy.

The Keyword `extends`

The keyword `extends` is used to identify that inheritance is taking place. A child class can only extend one parent class.

Example

Simulate class hierarchy and inheritance using the Mammal hierarchy. Each subclass inherits the traits of its superclass and can add additional traits of its own.

```
public class Mammal
{
    // All mammals are warm blooded
    // All mammals have lungs to breathe air
}
```

```
public class Dog extends Mammal
{
    // The Dog Class is a child class of the Mammal Class
    // A Dog is-a Mammal
    // All Dogs are warm blooded (inherited)
    // All Dogs have lungs to breathe air (inherited)
    // Dogs are domesticated (unique trait of a Dog)
}
```

```
public class LabradorRetriever extends Dog
{
    // The LabradorRetriever Class is a direct child class of the Dog Class
    // A LabradorRetriever is-a Dog
    // A LabradorRetriver is-a Mammal
    // LabradorRetrievers are warm blooded (inherited)
    // LabradorRetrievers have lungs to breathe air (inherited)
    // LabradorRetrievers are domesticated (inherited)
    // LabradorRetrievers are cute (unique trait of LabradorRetrievers)
}
```

Be Your Own Class

Child classes are allowed to define additional methods of their own. The total number of methods that the child class has is a combination of both the parent class and child class methods.

Constructors and the Keyword `super`

Constructors in a subclass are not inherited from their superclass. When you make an object from a child class, the child class constructor automatically calls the no-argument constructor of the parent class using the **super()** call (some integrated development environments, or IDEs, display this instruction). However, if you want to call a parent constructor with arguments from the child class constructor, you need to put that in your code explicitly as the first line of code in the child class constructor using the **super(arguments)** instruction. The child is then making a call to a parameterized constructor of the parent class. The actual parameters (arguments) passed in the call to the superclass constructor provide values that the constructor can use to initialize the object's instance variables. Regardless of whether the superclass constructor is called implicitly or explicitly, the process of calling superclass constructors continues until the Object constructor is called. At this point, all of the constructors within the hierarchy execute beginning with the Object constructor.

Example

Demonstrate a child class that has two constructors. The no-parameter constructor makes a call to the parent's no-parameter constructor. The parameterized constructor makes a call to the parent's parameterized constructor. Note: The super() call *must* be the first line of code in each of the child's constructors.

```
public class Mammal
{
   private boolean vertebrate;     // instance variable for a Mammal
   private boolean milkProducer;   // instance variable for a Mammal
   private String hairColor;       // instance variable for a Mammal

   public Mammal()                 // No parameter Mammal constructor
   {
      vertebrate = true;           // mammals are vertebrates
      milkProducer = true;         // mammals produce milk
   }

   public Mammal(String color)     // One parameter Mammal constructor
   {
      vertebrate = true;
      milkProducer = true;
      hairColor = color;           // Assign the hair color
   }

   /**  Assume accessor methods are defined  */
}
```

```
public class Dog extends Mammal
{
   private String name;           // every Dog has a name

   public Dog()
   {
      super();              // This call is made with or without typing it
   }

   public Dog(String hairColor, String nameOfDog)
   {
      super(hairColor);     // the call to the parent must come first!

      name = nameOfDog;     // assign the Dog's name in the Dog constructor
   }

   /**  Assume accessor methods are defined  */
}
```

```
public static void main(String[] args)
{
   Dog myDog1 = new Dog();
   System.out.println(myDog1.getName());          // will not have a name
   System.out.println(myDog1.getHairColor());     // will not have hair color
   System.out.println(myDog1.isVertebrate());     // will be true
   System.out.println(myDog1.isMilkProducer()); // will be true
   System.out.println();

   Dog myDog2 = new Dog("Brown", "Bella");
   System.out.println(myDog2.getName());          // the name is Bella
   System.out.println(myDog2.getHairColor());     // the color is Brown
   System.out.println(myDog2.isVertebrate());     // will be true
   System.out.println(myDog2.isMilkProducer()); // will be true
}
```

```
OUTPUT
null
null
true
true

Bella
Brown
true
true
```

Data Encapsulation Within Class Hierarchy

A subclass can only access or modify the private instance variables of its parent class *by using the accessor and modifier methods* of the parent class.

Children Don't Have Direct Access to Their Parent's `private` Instance Variables
Parent and child classes may both contain instance variables. These instance variables should be declared private. This is data encapsulation. Children have to use the accessor and modifier methods of the parent class to access their parent's private instance variables.

Polymorphism

"Sometimes children want to act in a different way from their parents."

Overriding a Method of the Parent Class

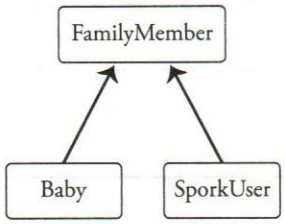
In Java, a child class is allowed to **override** a method of a parent class. This means that even though the child inherited a certain way to do something from its parent, it can do it in a different way if it wants to. This is an example of **polymorphism**.

The way to make a child override a parent method is to redefine a method (signature/method header) and change what is done in the method. When an object from the child class calls the method, it will perform the action that is found in the child class rather than the method of the parent class.

Example

All objects from the `FamilyMember` class drink from a cup and eat with a fork. However, objects from the `Baby` class drink from a cup and eat with their hands. All objects from the `SporkUser` class drink from a cup and eat with a spork. In this case, the eat method of the `FamilyMember` class is overridden by both the `Baby` class and the `SporkUser` class. Therefore, even though the name of the method is the same, the objects from `Baby` class and `SporkUser` class eat in a different way than objects from the `FamilyMember` class. The decision of when to use the overridden method by all objects that extend a superclass is made when the program is compiled and this process is called **static binding**.

Any method that is called must be defined within its own class or its superclass. A subclass is usually designed to have modified (overridden) or additional methods or instance variables. A subclass will inherit all public methods from the superclass; these methods remain public in the subclass.



```
public class FamilyMember // This is the superclass
{
   public String drink()
   {
      return "cup";       // the way that all FamilyMember objects drink
   }

   public String eat()
   {
      return "fork";      // the default way that all FamilyMembers eat
   }
}
```

```
public class Baby extends FamilyMember
{
   public String eat()      // Baby is overriding the eat() method
   {
      return "hands";       // all Baby objects eat with their hands
   }
}
```

```
public class SporkUser extends FamilyMember
{
   public String eat()  // SporkUser is overriding the eat() method
   {
      return "spork";    // all SporkUser objects eat with a spork
   }
}
```

```
public static void main(String[] args)
{
   FamilyMember mom = new FamilyMember();
   Baby junior = new Baby();
   SporkUser auntSue = new SporkUser();

   System.out.println(mom.drink());        // mom drinks with a cup
   System.out.println(junior.drink());     // junior drinks with a cup
   System.out.println(auntSue.drink());    // auntSue drinks with a cup

   System.out.println(mom.eat());          // mom eats with a fork
   System.out.println(junior.eat());       // junior eats with his hands
   System.out.println(auntSue.eat());      // auntSue eats with a spork
}
```

```
OUTPUT
cup
cup
cup
fork
hands
spork
```

Polymorphism
Java uses the term **polymorphism** to describe how different child classes of the same parent class can act differently. This is accomplished by having the child class override the methods of the parent class.

Dynamic Binding

A problem arises when we want to make a list of family members. *To make an array or* `ArrayList` *of people in the family, they all have to be of the same type!* To solve this problem, we make the reference variables all of the *parent* type.

Example

Create a FamilyMember `ArrayList` to demonstrate dynamic binding. Make the reference variables all the same type (the name of the parent class) and make the objects from the child classes. Allow each object to eat and drink the way they were designed to eat and drink. The decision of how each `FamilyMember` object eats is decided while the program is running (known as **run-time**) and this process is called **dynamic binding**. It's pretty cool!

```
public static void main(String[] args)
{
    FamilyMember mom = new FamilyMember();
    FamilyMember junior = new Baby();
    FamilyMember auntSue = new SporkUser();

    // Make a list of family members
    ArrayList<FamilyMember> family = new ArrayList<FamilyMember>();

    family.add(mom);
    family.add(junior);
    family.add(auntSue);

    // Demonstrate polymorphism by allowing everyone to eat their way
    for (FamilyMember member : family)
    {
        System.out.println(member.drink()); // everyone drinks with a cup
        System.out.println(member.eat());   // everyone eats their way
        System.out.println();
    }
}
```

```
OUTPUT
cup
fork

cup
hands

cup
spork
```

Reference Variables and Hierarchy

The reference variable *can* be the parent of a child object.

```
FamilyMember somebody = new Baby();    // Legal
```

The reference variable *cannot* be a child of a parent object.

```
Baby somebody = new FamilyMember();    // Illegal:  Type Mismatch error
```

Using `super` from a Child Method

Even if a child class overrides a method from the parent class, an object from a child class can call its parent's method if it wants to; however, it must do it from the overriding method and pass the appropriate parameters. By using the keyword **super**, a child can make a call to the parent method that it has already overridden. We say, "The child can invoke the parent's method if it wants to."

Example

Revise the Baby class so that if a baby is older than three years, it eats with either its hands or a fork; otherwise it eats with its hands. Use the keyword **super** to make a call to the parent's eat method. Also, add an instance variable for the age.

```
public class Baby extends FamilyMember
{
   private int age;          // every Baby has an age

   public Baby(int myAge)
   {
      age = myAge;           // assign the age to the Baby
   }

   public String eat()
   {
      if (age > 3)           // older baby may eat with hands or a fork
         return "hands or a " + super.eat();
      else
         return "hands";
   }
}
```

```
public static void main(String[] args)
{
   Baby youngBaby = new Baby(2);          // youngBaby is 2 years old
   Baby olderBaby = new Baby(4);          // olderBaby is 4 years old

   System.out.println(youngBaby.eat());   // youngBaby uses only hands
   System.out.println(olderBaby.eat());   // olderBaby uses hands or a fork
}
```

```
OUTPUT
hands
hands or a fork
```

Beware When Using a Parent Reference Variable for a Child Object

A child object has the ability to perform all the public methods of its parent class. However, a parent does not have the ability to do what the child does if the child action is original. Also, if a child object has a parent reference variable, the child *does not* have the ability to perform any of the unique methods that are defined in its own class.

Example

Demonstrate that a parent reference variable cannot perform any of the methods that are unique to a child class. A `FamilyMember` object does not know how to throwTantrum. Only a `Baby` object knows how to throwTantrum. Note: Even though `babyCousin` is a `Baby` object, the `FamilyMember` reference variable prevents it from being able to throwTantrum.

```
public class Baby extends FamilyMember
{
   public String throwTantrum()      // A Baby's impressive new ability
   {
      return "scream loudly";
   }

   /*  all other implementation is not shown  */
}
```

```
public static void main(String[] args)
{
   Baby junior = new Baby();              // reference is a Baby
   FamilyMember babyCousin = new Baby(); // reference is a FamilyMember

   System.out.println(junior.throwTantrum());       // legal
   System.out.println(babyCousin.throwTantrum());   // illegal
}
```

```
OUTPUT
COMPILE TIME ERROR:  the method, throwTantrum is undefined for FamilyMember
```

Downcasting

The problem described in the previous example can be solved using a technique called **downcasting**. This is similar to—but not exactly the same as—casting an `int` to a `double`. A parent object reference variable is cast to the object type of the object that it references.

Downcasting

You can downcast a parent reference variable to a child object as long as the object it is referring to is that child.

```
FamilyMember somebody = new Baby();  // Parent reference and child object
((Baby) somebody).uniqueMethodForBaby(); // Correct way to downcast
```

Example 1

Demonstrate downcasting by allowing the parent reference variable to perform a method that is unique to a child class. The parent reference variable is cast as a child and must refer to the child object. Pay close attention to the parentheses when casting.

```
public static void main(String[] args)
{
    Baby junior = new Baby();              // reference is a Baby
    FamilyMember babyCousin = new Baby(); // reference is a FamilyMember

    System.out.println(junior.throwTantrum());                  // legal
    System.out.println(((Baby) babyCousin).throwTantrum());   // legal
}
```

```
OUTPUT
scream loudly
scream loudly
```

Example 2

You *are not allowed* to cast a parent reference object to a child if the object is a parent. In this example, mom is a `FamilyMember` reference variable that refers to a `FamilyMember` object. Attempting to cast mom as a `Baby` results in an error.

```
FamilyMember mom = new FamilyMember();
((Baby) mom).eat(); // ClassCastException:FamilyMember cannot be cast to Baby
```

The `Object` Class

The **Object class** is the mother of all classes. It is part of the `java.lang` package. Every class, even the ones you write, is a descendent of the Object class. The extends is *implied*, so you don't have to write, "extends Object", in your own classes. The Object class does not have any instance variables, but it does have several methods that every descendent inherits.

Two of these methods, the **toString** and the **equals**, are tested on the AP Computer Science A Exam and are included on the Java Quick Reference sheet. Subclasses of `Object` often override the `equals` and `toString` methods with class-specific implementation. You must be able to implement the `toString` method; however, you only need to know how to use the equals method (like we do when we compare `String` objects).

You will receive a Java Quick Reference sheet to use on the Multiple-choice and Free-response sections, which lists the `String` class methods that may be included in the exam. Make sure you are familiar with it before you take the exam.

Object Class	
`boolean equals(Object other)`	
`String toString()`	

Overriding the `toString` Method

The **toString** method is inherited from the Object class and was created to describe the object in some way, typically by referencing its instance variables. The Object class's `toString` method doesn't give you anything valuable (it returns the hex value of the object's address). Most of the time, when creating your own classes, you will override the `toString` method so it returns a String that describes the object in a meaningful way.

Example

We overrode the `toString` method for the Circle class in Unit 5 so the method gives a description of the Circle object that calls it. The String that is returned should include the radius of the circle and its area.

```
public class Circle
{
   /*  All other implementation not shown  */

   public String toString()              // Override the toString method
   {
      return "The circle has radius " + radius + " and area " +
             getArea();
   }
}
```

```
public static void main(String[] args)
{
   Circle circle = new Circle(5);          // Circle with a radius of 5
   System.out.println(circle);             // call the toString() method
}
```

```
OUTPUT
The circle has radius 5.0 and area 78.5
```

Overriding the `toString` Method
All classes extend the Object class. When you design your own classes, *be sure to override the toString() method* of the Object class so that it describes objects from your class in a unique way using its own instance variables.

Casting an `Object`

If a method's parameter list contains an `Object` reference variable, then you will have to downcast the reference before using it. You must make sure that the object that you are casting `is-a` object from the class.

Example

Demonstrate how to handle an `Object` reference parameter. In this example, when the `FamilyMember` object, `uncleDon`, gets passed to the `timeToEat` method, he is sent as a `FamilyMember` reference variable. When he is received by the `timeToEat` method, he is renamed as `hungryMember` and is now an `Object` reference. Finally, he is cast to a `FamilyMember` and is then able to eat.

```
public static void main(String[] args)
{
   FamilyMember uncleDon = new FamilyMember();
   timeToEat(uncleDon);
}

public static void timeToEat(Object hungryMember)
{
   ((FamilyMember) hungryMember).eat();  // casting Object to FamilyMember

   // Even though uncleDon is an Object reference, he can eat like a
   // FamilyMember because he was downcast to a FamilyMember
}
```

```
OUTPUT
fork
```

› Rapid Review

Hierarchy and Inheritance

- One major advantage of an object-oriented language is the ability to create a hierarchy of classes that allows objects from one class to inherit the features from another class.
- A class hierarchy is formed when one class extends another class.
- A class that extends another class is called the child or subclass.
- The class that is extended is called the parent or superclass.
- A class can only extend one other class.
- A class hierarchy can extend as many times as you want. That is, a child can extend a parent, then the parent can extend another parent, and so on.
- When a class hierarchy gets extended to more than just one parent and one child, the lower ends of the hierarchy inherit the features of each of the parent classes above it.
- The phrase "is-a" helps describe what parent classes a child class belongs to.
- If you want a child class to modify a parent class's method, then you override the parent class's method. To do this, you simply redefine the method in the child class (using the exact same method declaration and replace the code with what you want the child to do).
- The word "super" can be used to refer to either a parent's constructor or methods.
- To purposefully call the no-argument super constructor, you must put super() as the first line of code in the child's constructor. Other instructions are placed after the super() call.
- To call the parameterized super constructor, you must put super(arguments) as the first line of code in the child's parameterized constructor. Other instructions are placed after the super() call.
- If you write a method that overrides the parent class's method, you can still call the original parent's method if you want to. Example: `super.parentMethod()`.
- A reference variable can be a parent class type if the object being created is its child class type.
- A reference variable cannot be a child class type if the object being created is its parent class type.
- A reference variable that is of a parent data type can be cast to a child data type as long as the object that it is referencing is a child object of that type.

Polymorphism

- Polymorphism is when more than one child class extends the same parent class and each child object is allowed to act in a different way than its parent and even its siblings.
- If several child classes extend the same parent class and each child class overrides the same method of the parent class in its own unique way, then each child will act differently for that method.
- Static binding is the process of deciding which method the child reference variable will perform during compile-time.
- Dynamic binding is the process of deciding which method the parent reference will perform during run-time.
- The `Object` class is the mother of all classes in Java since it is the root of the class hierarchy and every class has `Object` as a superclass.
- The Object class contains two methods that are tested on the AP Computer Science A exam: equals and toString.
- The public boolean `equals(Object other)` method returns true if the object reference variable that calls it is referencing the same exact object as other. It returns false if the object reference variables are not referencing the same object.

- It is common to override the `equals` method when you want to determine if two objects from a class are the same.
- The public String `toString()` method returns a String representation of the object.
- It is very common to override the `toString()` method when generating your own classes.
- Downcasting is casting a reference variable of a parent class to one of its child classes.
- Downcasting is commonly used when an `Object` is in a parameter list or `ArrayList`.
- A major problem with downcasting is that the compiler will not catch any errors, and if the cast is made incorrectly, a run-time error will be thrown.

› Review Questions

Basic Level

1. Assuming all classes are defined in a manner appropriate to their names, which of the following statements will cause a compile-time error?

(A) `Object obj = new Lunch();`
(B) `Lunch lunch = new Lunch();`
(C) `Lunch sandwich = new Sandwich();`
(D) `Lunch lunch = new Object();`
(E) `Sandwich sandwich = new Sandwich();`

Questions 2–5 refer to the following classes.

```
public class Student
{
   private int gradYear;
   private double gpa;

    /*  other implementation not shown  */
}

public class UnderClassman extends Student
{
   private int homeRoomNum;
   private String counselor;

    /*  other implementation not shown  */
}

public class Freshman extends UnderClassman
{
   private int middleSchoolCode;

    /*  other implementation not shown  */
}
```

2. Which of the following is true with respect to the classes defined above?

I. `Freshman` is a subclass of `UnderClassman` and `UnderClassman` is a subclass of `Student`
II. `Student` is a superclass of both `UnderClassman` and `Freshman`
III. `UnderClassman` is a superclass of `Freshman` and a subclass of `Student`

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

3. Which of the following lists of instance data contains only variables that are directly accessible to a `Freshman` class object?

   (A) `middleSchoolCode`
   (B) `gradYear, gpa`
   (C) `homeRoomNum, counselor`
   (D) `middleSchoolCode, homeRoomNum, counselor`
   (E) `middleSchoolCode, homeRoomNum, counselor, gradYear, gpa`

4. Assume that all three classes contain parameterized constructors with the following declarations and that all variables are logically named.

```
public Student (int gradYear, double gpa)

public UnderClassman (int gradYear, double gpa, int homeRoomNum, String counselor)

public Freshman (int gradYear, double gpa, int homeRoomNum, String counselor,
      int middleSchoolCode)
```

   Which of the following calls to `super` would appear in the constructor for the `Freshman` class?

   (A) `super(gradYear, gpa, homeRoomNum, counselor, middleSchoolCode);`
   (B) `super(gradYear, gpa, homeRoomNum, counselor);`
   (C) `super(homeRoomNum, counselor);`
   (D) `super(gradYear, gpa);`
   (E) `super();`

5. Consider the following method in the `Student` class.

```
public void attendClass(int roomNumber, String subject)
{
   /* implementation not shown */
}
```

   A `Freshman` object must `attendClass` like a `Student` object, but also have the teacher initial his/her homework planner (using the `initialPlanner` method, not shown).

   Which of the following shows a possible implementation of the `Freshman attendClass` method?

   (A)
```
public void attendClass extends Student(int roomNumber, String subject)
{
   super.attendClass(int roomNumber, String subject);
   initialPlanner();
}
```

   (B)
```
public void attendClass extends Student(int roomNumber, String subject)
{
   super.attendClass();
   initialPlanner();
}
```

   (C)
```
public void attendClass(int roomNumber, String  subject)
{
   super(int roomNumber, String subject);
   initialPlanner();
}
```

(D)
```
public void attendClass(int roomNumber, String  subject)
{
   super(roomNumber, subject);
   initialPlanner();
}
```

(E)
```
public void attendClass(int roomNumber, String  subject)
{
   super.roomNumber = roomNumber;
   super.subject = subject;
   initialPlanner();
}
```

Advanced Level

Questions 6–7 refer to the following classes.

```
public class Club
{
   private ArrayList<String> members;

   public Club() {  }

   public Club(ArrayList<String> theMembers)
   {  members = theMembers;  }

   /* Additional implementation not shown */
}

public class SchoolClub extends Club
{
   private String advisor;

   public SchoolClub(String theAdvisor, ArrayList<String> theMembers)
   {
      /* missing code */
   }

   /* Additional implementation not shown */
}
```

6. Assume `ArrayList<String> members_1` and `ArrayList<String> members_2` have been properly instantiated and initialized with a non-zero number of appropriate elements.

Which of the following declarations is NOT valid?

(A) `Club[] clubs = new SchoolClub[7];`

(B) `Club[] clubs = {new Club(members_1), new Club()};`

(C)
```
SchoolClub[] clubs = {new SchoolClub("Mr. Johnson", members_1),
      new Club(members_2)};
```

(D)
```
SchoolClub[] clubs = {new SchoolClub("Ms. Paymer", members_1),
         new SchoolClub("Mr. Johnson", members_2)};
```

(E) All of the above are valid.

7. Which of the following could replace `/* missing code */` in the `SchoolClub` constructor to ensure that all instance variables are initialized correctly?

(A)
```
super(theMembers);
advisor = theAdvisor;
```
(B)
```
super(new Club(theMembers));
advisor = theAdvisor;
```
(C)
```
this.SchoolClub = new Club(members);
advisor = theAdvisor;
```
(D)
```
advisor = theAdvisor;
```
(E)
```
advisor = theAdvisor;
super(theMembers);
```

Questions 8–9 refer to the following classes.

```
public class Present
{
   private String contents;

   public Present(String theContents)
   {  contents = theContents;  }

   public String getContents()
   {  return contents;  }

   public void setContents(String theContents)
   {  contents = theContents;  }

   public String toString()
   {  return "contains " + getContents();  }
}

public class KidsPresent extends Present
{
   private int age;

   public KidsPresent(String contents, int theAge)
   {
      super(contents);
      age = theAge;
   }

   public int getAge()
   {  return age;  }

   public String toString()
   {  return super.toString() + " for a child age " + getAge();  }
}
```

8. Consider the following code segment.

```
Present p = new KidsPresent("kazoo", 4);
System.out.println(p);
```

What is printed as a result of executing the code segment?

(A) `contains kazoo`
(B) `kazoo`
(C) `contains kazoo for a child age 4`
(D) Nothing is printed. Compile-time error: illegal cast
(E) Nothing is printed. Run-time error: illegal cast

9. Consider the following code segment.

```
Present p = new KidsPresent("kazoo", 4);
p.setContents("blocks");
System.out.println(p);
```

What is printed as a result of executing the code segment?

(A) `contains blocks`
(B) `contains blocks for a child age 4`
(C) `contains kazoo for a child age 4`
(D) Nothing is printed. Compile-time error: there is no `setContents` method in the `KidsPresent` class
(E) Nothing is printed. Run-time error: there is no `setContents` method in the `KidsPresent` class

10. Point3D Class

Consider the following class:

```
public class Point
{
   private int x;
   private int y;

   public Point(int newX, int newY)
   {
      x = newX;
      y = newY;
   }

   public int getX()
   {  return x;  }

   public int getY()
   {  return y;  }
}
```

Write the class `Point3D` that extends this class to represent a three-dimensional point with x, y, and z coordinates. Be sure to use appropriate inheritance. Do not duplicate code.

You will need to write:

a. A constructor that takes three `int` values representing the x, y, and z coordinates
b. Appropriate mutator and accessor methods (setters and getters)