A Class for Modeling an Automobile

```java
public class Automobile {
    private String make;
    private String model;
    private int year;
    private int mileage;
    private String plateNumber;
    private int numSeats;
    private boolean isSUV;

    public Automobile(String make, String model, int year, int numSeats, boolean isSUV) {
        this(make, model, year);
        this.mileage = 0;
        this.plateNumber = "unknown";
    }

    public Automobile(String make, String model, int year) {
        this(make, model, year, 5, false);
    }

    // continued...
```
A Class for Modeling an Automobile (cont.)

public String getMake() {
    return this.make;
}

public String getModel() {
    return this.model;
}

public int getYear() {
    return this.year;
}

public int getMileage() {
    return this.mileage;
}

public String getPlateNumber() {
    return this.plateNumber;
}

public int getNumSeats() {
    return this.numSeats;
}

public boolean isSUV() {
    return this.isSUV;
}

// continued...

A Class for Modeling an Automobile (cont.)

public void setMileage(int newMileage) {
    if (newMileage < this.mileage) {
        throw new IllegalArgumentException();
    }
    this.mileage = newMileage;
}

public void setPlateNumber(String plate) {
    this.plateNumber = plate;
}

public String toString() {
    String str = this.make + " " + this.model;
    str += "( " + this.numSeats + " seats)";
    return str;
}

• There are no mutators for the other fields. Why not?
Modeling a Related Class

• What if we now want to write a class to represent a taxi?

• The Taxi class will have the same fields and methods as the Automobile class.

• It will also have its own fields and methods:
  - taxiID, getID, setID
  - fareTotal, getFareTotal, addFare
  - numFares, getNumFares, getAverageFare
  - resetFareInfo

• We may also want the Taxi versions of some of the Automobile methods to behave differently. Examples:
  - we may want the toString method to include values from different fields
  - we may want the getNumSeats method to return only the number of seats available for passengers

Inheritance

• To avoid redefining all of the Automobile fields and methods, we specify that the Taxi class extends the Automobile class:
  public class Taxi extends Automobile {

• The Taxi class will inherit the fields and methods of the Automobile class.
  • it doesn't have to redefine them
A Class for Modeling a Taxi

public class Taxi extends Automobile {
    // We don't need to include the fields
    // from Automobile!
    private String taxiID;
    private double fareTotal;
    private int numFares;
    // constructor goes here...
    // we don't need to include the methods
    // from Automobile!
    public String getID() {
        return this.taxiID;
    }
    public void addFare(double fare) {
        if (fare < 0) {
            throw new IllegalArgumentException();
        }
        this.fareTotal += fare;
        this.numFares++;
    }
    ...
}

Using Inherited Methods

• Because Taxi extends Automobile, we can invoke a method
defined in the Automobile class on a Taxi object.
  • example:
    Taxi t = new Taxi(…);
    t.setMileage(25000);

• This works even though there is no setMileage method
defined in the Taxi class!
  • Taxi inherits it from Automobile
Overriding an Inherited Method

- A subclass can override an inherited method, replacing it with its own version.

- To override a method, the new method must have the same:
  - return type
  - name
  - number and types of parameters

- Example: our Taxi class can define its own toString method:
  ```java
  public String toString() {
      return "Taxi (id = " + this.taxiID + ")";
  }
  ```
  - it overrides the toString method inherited from Automobile

Rethinking Our Design

- What if we also want to be able to capture information about other types of vehicles?
  - motorcycles
  - trucks

- The classes for these other vehicles should not inherit from Automobile. Why not?

- Solution: define a Vehicle class
  - fields and methods common to all vehicles are defined there
  - leave automobile-specific state and behavior in Automobile
    - everything else is inherited from Vehicle
  - define Motorcycle and Truck classes that also inherit from Vehicle
A Class for Modeling a Vehicle

```java
public class Vehicle {
    private String make;
    private String model;
    private int year;
    private int mileage;
    private String plateNumber;
    private int numWheels;  // this was not in Automobile

    public Vehicle(String make, String model, int year,
                       int numWheels) {
        this.make = make;
        this.model = model;
        if (year < 1900) {
            throw new IllegalArgumentException();
        }
        this.year = year;
        this.numWheels = numWheels;
        this.mileage = 0;
        this.plateNumber = "unknown";
    }

    public String getMake() {
        return this.make;
    }

    // etc.
}
```

Revised Automobile Class

```java
public class Automobile extends Vehicle {
    // make, model, etc. are now inherited from Vehicle
    // The following are specific to automobiles,
    // so we leave them here.
    private int numSeats;
    private boolean isSUV;

    // constructor goes here...

    // getMake(), etc. are now inherited from Vehicle
    // The following are specific to automobiles,
    // so we leave them here.
    public int getNumSeats() {
        return this.numSeats;
    }

    public boolean isSUV() {
        return this.isSUV;
    }

    ...
}
```
Inheritance Hierarchy

- Inheritance leads classes to be organized in a hierarchy:

  ![Inheritance Diagram]

- A class in Java inherits directly from at most one class.
- However, a class can inherit indirectly from a class higher up in the hierarchy.
  - example: Taxi inherits indirectly from Vehicle

Terminology

- When class C extends class D (directly or indirectly):
  - class D is known as a superclass or base class of C
    - super – comes above it in the hierarchy
  - class C is known as a subclass or derived class of D
    - sub – comes below it in the hierarchy

  - Examples:
    - Automobile is a superclass of Taxi and Limousine
    - Taxi is a subclass of Automobile and Vehicle
Deciding Where to Define a Method

- Assume we only care about the number of axles in truck vehicles.
- Thus, we define the `getNumAxles` method in the `Truck` class, rather than in the `Vehicle` class.
  ```java
  public int getNumAxles() {
      return this.getNumWheels() / 2;
  }
  ```
  - it will be inherited by subclasses of `Truck`
  - it won’t be available to non-truck subclasses of `Vehicle`
- We override this method in the `TractorTrailer` class, because tractor trailers have four wheels on all but the front axle:
  ```java
  public int getNumAxles() {
      int numBackWheels = this.getNumWheels() - 2;
      return 1 + numBackWheels/4;
  }
  ```

What is Accessible From a Superclass?

- A subclass has direct access to the `public` fields and methods of a superclass.
- A subclass does not have direct access to the `private` fields and methods of a superclass.
- Example: we can think of an `Automobile` object as follows:

```
make
model
year
mileage
plateNumber
numWheels
numSeats
isSUV
```

- fields defined in `Automobile`. They can be accessed directly by methods in `Automobile`.
- private fields inherited from `Vehicle`. They cannot be accessed directly by methods in `Automobile`. 
What is Accessible From a Superclass? (cont.)

• Example: now that make and model are defined in Vehicle, we're no longer able to access them directly in the Automobile version of toString:

```java
public String toString() {
    String str = this.make + " " + this.model;
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

• Instead, we need to make method calls to access the inherited fields:

```java
public String toString() {
    String str = this.getMake() + " " +
                 this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

What is Accessible From a Superclass? (cont.)

• Faulty approach: redefine the inherited fields in the subclass

```java
public class Vehicle {
    private String make;
    private String model;
    ...
}
```

```java
public class Automobile extends Vehicle {
    private String make; // NOT a good idea!
    private String model;
    ...
}
```

• You should NOT do this!
Writing a Constructor for a Subclass

- Another example of illegally accessing inherited private fields:
  
  ```java
  public Automobile(String make, String model, int year, int numSeats, boolean isSUV) {
    this.make = make;
    this.model = model;
  }
  ...
  ```
  
- To initialize inherited fields, a constructor should invoke a constructor from the superclass.
  
  ```java
  public Automobile(String make, String model, int year, int numSeats, boolean isSUV) {
    super(make, model, year, 4); // 4 is for numWheels
    this.numSeats = numSeats;
    this.isSUV = isSUV;
  }
  ```
  
  - use the keyword `super` followed by appropriate parameters for the superclass constructor
  - must be done as the very first line of the constructor

Writing a Constructor for a Subclass (cont.)

- If a subclass constructor doesn't explicitly invoke a superclass constructor, the compiler tries to insert a call to the superclass constructor with no parameters.
  
  - example: this constructor won't compile:
    ```java
    public Taxi(String make, String model, int year, String ID) {
      this.taxiID = ID;
    }
    ```
    
    - the compiler attempts to insert the following call:
      ```java
      super();
      ```
    - there isn't an `Automobile` constructor with no parameters
The Object Class

- If a class doesn't explicitly extend another class, it implicitly extends a special class called Object.
- Thus, the object class is at the top of the class hierarchy.
  - all classes are subclasses of this class
  - the default toString and equals methods are defined in this class

Inheritance in the Java API

```java
java.lang.Object
  - java.awt.geom.RectangleShape
    - java.awt.geom.Rectangle2D
    - java.awt.Rectangle

All Implemented Interfaces:
  Shape, Serializable, Cloneable

Direct Known Subclasses:
  DefaultRect
```
More Examples of Method Overriding

- Vehicle inherits the fields and methods of Object.
- The inherited toString method isn't very helpful.
- We define a Vehicle version that overrides the inherited one:
  ```java
  public String toString() { // Vehicle version
    String str = this.make + " " + this.model;
    return str;
  }
  ```
- When toString is invoked on a Vehicle object, the vehicle version is executed:
  ```java
  Vehicle v = new Vehicle("Radio Flyer", "Classic Tricycle", 2002, 3);
  System.out.println(v);
  ```
  outputs: Radio Flyer Classic Tricycle

More Examples of Method Overriding (cont.)

- The Automobile class inherits the Vehicle version of toString.
- If we didn't define a toString() method in Automobile, the inherited version would be used.
- The Automobile version overrides the Vehicle version so that the number of seats can be included in the string:
  ```java
  public String toString() {
    String str = this.getMake() + " " + this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
  }
  ```
Invoking an Overridden Method

- When a subclass overrides an inherited method, we can invoke the inherited version by using the keyword `super`.

- Example: the `Automobile` version of `toString()` begins with the same fields as the `Vehicle` version:

```java
public String toString() {
    String str = this.getMake() + " " +
                 this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

- Instead of calling the accessor methods, we can do this:

```java
public String toString() {
    String str = super.toString();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

Another Example of Inheritance

- A square is a special type of rectangle.
  - But the width and height must be the same

- Assume that we also want `Square` objects to have a field for the unit of measurement (e.g., "cm").

- `Square` objects should mostly behave like `Rectangle` objects:

```java
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();
Square sq = new Square(40, "cm");
int area2 = sq.area();
```

- But there may be differences as well:

```java
System.out.println(r);  ➞ output: 20 x 30
System.out.println(sq); ➞ output: square with 40-cm sides
```
Another Example of Inheritance (cont.)

```java
public class Rectangle {
    private int width;
    private int height;
    ...
    public Rectangle(int initWidth, int initHeight) {
        ...
    }
    public int getWidth() {
        ...
    }
    ... // other methods
}

public class Square extends Rectangle {
    private String unit; // inherits other fields
    public Square(int side, String unit) {
        super(side, side);
        this.unit = unit;
    }
    public String toString() {  // overrides
        String s = "square with ";
        s += this.getWidth() + "-";
        s += this.unit + " sides";
        return s;
    }
    // inherits other methods
}
```

Another Example of Inheritance (cont.)

```java
public class Rectangle {
    private int width;
    private int height;
    ...
    public Rectangle(int initWidth, int initHeight) {
        ...
    }
    public int getWidth() {
        ...
    }
    ... // other methods
}

public class Square extends Rectangle {
    private String unit; // inherits other fields
    public Square(int side, String unit) {
        super(side, side);
        this.unit = unit;
    }
    public String toString() {  // overrides
        String s = "square with ";
        s += this.getWidth() + "-";
        s += this.unit + " sides";
        return s;
    }
    // inherits other methods
}
```
Another Example of Method Overriding

• The Rectangle class has the following mutator method:
  
  ```java
  public void setWidth(int w) {
    if (w <= 0) {
      throw new IllegalArgumentException();
    }
    this.width = w;
  }
  ```

• The Square class inherits it. Why should we override it?

Which of these works?

A. // Square version, which overrides
   // the version inherited from Rectangle
   public void setWidth(int w) {
     this.width = w;
     this.height = w;
   }

B. // Square version, which overrides
   // the version inherited from Rectangle
   public void setWidth(int w) {
     this.setWidth(w);
     this.setHeight(w);
   }

C. either version would work

D. neither version would work
Accessing Methods from the Superclass

• The solution: use super to access the inherited version of the method – the one we are overriding:

```java
// Square version
public void setWidth(int w) {
    super.setWidth(w);  // call the Rectangle version
    super.setHeight(w);
}
```

• Only use super if you want to call a method from the superclass that has been overridden.

• If the method has not been overridden, use this as usual.

Accessing Methods from the Superclass

• We need to override all of the inherited mutators:

```java
// Square versions
public void setWidth(int w) {
    super.setWidth(w);
    super.setHeight(w);
}
public void setHeight(int h) {
    super.setWidth(h);
    super.setHeight(h);
}
public void grow(int dw, int dh) {
    if (dw != dh) {
        throw new IllegalArgumentException();
    }
    super.setWidth(this.getWidth() + dw);
    super.setHeight(this.getHeight() + dh);
}
```

`setWidth()` and `getHeight()` are not overridden, so we use this.
is-a Relationships

- We use inheritance to capture is-a relationships.
  - an automobile is a vehicle
  - a taxi is an automobile
  - a tractor trailer is a truck

```
Object
    ↓
Vehicle
    ▼
Motorcycle   Automobile   Truck
        ▼            ▼
Limousine   Taxi         MovingVan   TractorTrailer
```

has-a Relationships

- Another type of relationship is a has-a relationship.
  - one type of object "owns" another type of object
  - example: a driver has a vehicle

- Inheritance should not be used to capture has-a relationships.
  - it does not make sense to make the Driver class a subclass of Vehicle

- Instead, we give the "owner" object a field that refers to the "owned" object:

  ```java
  public class Driver {
      String name;
      String ID;
      Vehicle v;
      ...
  ```
Polymorphism

• We've been using reference variables like this:
  \[\text{Automobile } a = \text{new Automobile("Ford", "Model T", ...);}\]
  • variable \(a\) is declared to be of type \text{Automobile}
  • it holds a reference to an \text{Automobile} object

• In addition, a reference variable of type \(T\) can hold a reference to an object from a \textit{subclass} of \(T\):
  \[\text{Automobile } a = \text{new Taxi("Ford", "Tempo", ...);}\]
  • this works because \text{Taxi} is a subclass of \text{Automobile}
  • a taxi is an automobile!

• The name for this feature of Java is \textit{polymorphism}.
  • from the Greek for “many forms”
  • the same code can be used with objects of different types!

Polymorphism and Collections of Objects

• Polymorphism is useful when we have a collection of objects of different but related types.

• Example:
  • let's say that a company has a collection of vehicles of different types
  • we can store all of them in an array of type \text{vehicle}:
  \[
  \text{Vehicle[]} \ \text{fleet} = \text{new Vehicle}[5]; \\
  \text{fleet}[0] = \text{new Automobile("Honda", "Civic", ...);} \\
  \text{fleet}[1] = \text{new Motorcycle("Harley", ...);} \\
  \text{fleet}[2] = \text{new TractorTrailer("Mack", ...);} \\
  \text{fleet}[3] = \text{new Taxi("Ford", ...);} \\
  \text{fleet}[4] = \text{new Truck("Dodge", ...);}
  \]
Processing a Collection of Objects

- We can determine the average age of the vehicles in the company's fleet by doing the following:

```java
int totalAge = 0;
for (int i = 0; i < fleet.length; i++) {
    int age = CURRENT_YEAR - fleet[i].getYear();
    totalAge += age;
}
double averageAge = (double)totalAge / fleet.length;
```

- We can invoke `getYear()` on each object in the array, regardless of its type.
  - their classes are all subclasses of `Vehicle`
  - thus, they must all have a `getYear()` method

Practice with Polymorphism

- Which of these assignments would be allowed?

```java
Vehicle v1 = new Motorcycle(...);
TractorTrailer t1 = new Truck(...);
Truck t2 = new MovingVan(...);
Taxi t3 = new Automobile(...);
Vehicle v2 = new TractorTrailer(...);
MovingVan m1 = new TractorTrailer(...);
```
Declared Type vs. Actual Type

- An object's declared type may not match its actual type:
  - declared type: type specified when declaring a variable
  - actual type: type specified when creating an object

- Recall this client code:
  ```java
  Vehicle[] fleet = new Vehicle[5];
  fleet[0] = new Automobile("Honda", "Civic", 2005);
  fleet[1] = new Motorcycle("Harley", ...);
  fleet[2] = new TractorTrailer("Mack", ...);
  ```

- Here are the types:

<table>
<thead>
<tr>
<th>object</th>
<th>declared type</th>
<th>actual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>fleet[0]</td>
<td>Vehicle</td>
<td>Automobile</td>
</tr>
<tr>
<td>fleet[1]</td>
<td>Vehicle</td>
<td>Motorcycle</td>
</tr>
<tr>
<td>fleet[2]</td>
<td>Vehicle</td>
<td>TractorTrailer</td>
</tr>
</tbody>
</table>

- The compiler uses the declared type of an object to determine if a method call is valid.
  - starts at the declared type, and goes up the inheritance hierarchy as needed looking for a version of the method
  - if it can't find a version, the method call will not compile

- Example: the following would not work:
  ```java
  Vehicle[] fleet = new Vehicle[5];
  ...
  fleet[2] = new TractorTrailer("Mack", ...);
  ...
  System.out.println(fleet[2].getNumAxles());
  ```
  - the declared type of fleet[2] is Vehicle
  - there's no getNumAxles() method defined in or inherited by Vehicle

Determining if a Method Call is Valid
Determining if a Method Call is Valid (cont.)

- In such cases, we can use casting to create a reference with the necessary declared type:
  ```java
  Vehicle[] fleet = new Vehicle[5];
  ...
  fleet[2] = new TractorTrailer("Mack", ...);
  ...
  TractorTrailer t = (TractorTrailer) fleet[2];
  ```

- The following will work:
  ```java
  System.out.println(t.getNumAxles());
  ```
  - the declared type of `t` is `TractorTrailer`
  - there is a `getNumAxles()` method defined in `TractorTrailer`, so the compiler is happy

Determining Which Method to Execute

- Truck also has a `getNumAxles` method, so this would be another way to handle the previous problem:
  ```java
  Vehicle[] fleet = new Vehicle[5];
  ...
  fleet[2] = new TractorTrailer("Mack", ...);
  ...
  Truck t2 = (Truck) fleet[2];
  System.out.println(t2.getNumAxles());
  ```

- The object represented by `t2` has:
  - a declared type of ______________
  - an actual type of ______________

- Both Truck and TractorTrailer have a `getNumAxles`. Which version will be executed?

- More generally, how does the interpreter decide which version of a method should be used?
Dynamic Binding

- At runtime, the Java interpreter selects the version of a method that is appropriate to the actual type of the object.
  - starts at the actual type, and goes up the inheritance hierarchy as needed until it finds a version of the method
  - known as dynamic binding

- Given the code from the previous slide
  ```java
  Vehicle[] fleet = new Vehicle[5]
  fleet[2] = new TractorTrailer("Mack", ...);
  Truck t2 = (Truck) fleet[2];
  System.out.println(t2.getNumAxles());
  ```
  the TractorTrailer version of getNumAxles would be run
  - TractorTrailer is the actual type of t2, and that class has its own version of getNumAxles

Dynamic Binding (cont.)

- Another example:
  ```java
  public static void printFleet(Vehicle[] fleet) {
      for (int i = 0; i < fleet.length; i++) {
          System.out.println(fleet[i]);
      }
  }
  ```
  - the toString() method is implicitly invoked on each element of the array when we go to print it.
  - the appropriate version is selected by dynamic binding
  - note: the selection must happen at runtime, because the actual types of the objects may not be known when the code is compiled
Dynamic Binding (cont.)

• Recall our initialization of the array:
  
  ```java
  Vehicle[] fleet = new Vehicle[5];
  fleet[0] = new Automobile("Honda", "Civic", ...);
  fleet[1] = new Motorcycle("Harley", ...);
  fleet[2] = new TractorTrailer("Mack", ...);
  ...
  ```

  • System.out.println(fleet[0]); will invoke the
    Automobile version of the toString() method.

  • Motorcycle does not define its own toString() method,
    so System.out.println(fleet[1]); will invoke the Vehicle
    version of toString(), which is inherited by Motorcycle.

  • TractorTrailer does not define its own toString() but Truck does, so System.out.println(fleet[2]);
    will invoke the Truck version of toString(), which is inherited
    by TractorTrailer.

Dynamic Binding (cont.)

• Dynamic binding also applies to method calls on the
called object that occur within other methods.

  • Example: the Truck class has the following toString method:
    
    ```java
    public String toString() {
        String str = this.getMake() + " " +
        this.getModel();
        str = str + ", capacity = " + this.capacity;
        str = str + ", " + this.getNumAxles() + " axles";
        return str;
    }
    ```

    • The TractorTrailer class inherits it and does not override it.

    • When toString is called on a TractorTrailer object:
      • this Truck version of toString() will run
      • the TractorTrailer version of getNumAxles() will run when the code above is executed
The Power of Polymorphism

• Recall our printFleet method:
  public static void printFleet(Vehicle[] fleet) {
    for (int i = 0; i < fleet.length; i++) {
      System.out.println(fleet[i]);
    }
  }
  • polymorphism allows this method to use a single println statement to print the appropriate info. for any kind of vehicle.

• Without polymorphism, we would need a large if-else-if:
  if (fleet[i] is an Automobile) {
    print the appropriate info for Automobiles
  } else if (fleet[i] is a Truck) {
    print the appropriate info for Trucks
  } else if ...

  • Polymorphism allows us to easily write code that works for more than one type of object.

Polymorphism and the Object Class

• The object class is a superclass of every other class.

• Thus, we can use an object variable to store a reference to any object.

  Object o1 = "Hello World";
  Object o2 = new Temperature(20, 'C');
  Object o3 = new Taxi("Ford", "Tempo", 2000, "T253");
Summary and Extra Practice

• To determine if a method call is valid:
  • start at the *declared* type
  • go up the hierarchy as needed to see if you can find the specified method in the declared type or a superclass
  • if you don't find it, the method call is *not* valid

• Given the following:
  TractorTrailer t1 = new TractorTrailer(...);
  Vehicle v = new Truck(...);
  MovingVan m = new MovingVan(...);
  Truck t2 = new TractorTrailer(...);

• Which of the following are valid?
  v.getNumAxles()
  m.getNumAxles()
  t1.getMake()
  t1.isSleeper()
  t2.isSleeper()

Summary and Extra Practice (cont.)

• To determine which version of a method will run (dynamic binding):
  • start at the *actual* type
  • go up the hierarchy as needed until you find the method
  • the first version you encounter is the one that will run

• Given the following:
  TractorTrailer t1 = new TractorTrailer(...);
  Vehicle v = new Truck(...);
  MovingVan m = new MovingVan(...);
  Truck t2 = new TractorTrailer(...);

• Which version of the method will run?
  m.getNumAxles()
  t1.getNumAxles()
  t2.getNumAxles()
  v.getMake()
  t2.getMake()
public class E extends G {
    public void method2() {
        System.out.print("E 2 ");
        this.method1();
    }
    public void method3() {
        System.out.print("E 3 ");
        this.method1();
    }
}
public class F extends G {
    public void method2() {
        System.out.print("F 2 ");
    }
}
public class G {
    public void method1() {
        System.out.print("G 1 ");
    }
    public void method2() {
        System.out.print("G 2 ");
    }
}
public class H extends E {
    public void method1() {
        System.out.print("H 1 ");
    }
}

More Practice (cont.)

• Which of these would compile and which would not?
  E e1 = new E();
  E e2 = new H();
  E e3 = new G();
  E e4 = new F();
  G g1 = new H();
  G g2 = new F();
  H h1 = new H();

• To answer these questions, draw the inheritance hierarchy:
Here are the classes again...

```java
public class E extends G {
    public void method2() {
        System.out.print("E 2 ");
        this.method1();
    }
    public void method3() {
        System.out.print("E 3 ");
        this.method1();
    }
}
public class F extends G {
    public void method2() {
        System.out.print("F 2 ");
    }
}
public class G {
    public void method1() {
        System.out.print("G 1 ");
    }
    public void method2() {
        System.out.print("G 2 ");
    }
}
public class H extends E {
    public void method1() {
        System.out.print("H 1 ");
    }
}
```

More Practice (cont.)

```java
E e1 = new E();
G g1 = new H();
G g2 = new F();
```

- Which of the following would compile and which would not? For the ones that would compile, what is the output?

```java
e1.method1();
e1.method2();
e1.method3();
g1.method1();
g1.method2();
g1.method3();
g2.method1();
g2.method2();
g2.method3();
```