Questions

For many of the questions below, I did not give you "rules" for answering them. I would like you to answer such questions by answering how you think it *should* be, and checking this is consistent with how it works.

1. A method signature is only defined by method name and parameter types. This implies that we cannot override (or overload) a method by having different return type only. This makes sense. Why? What problem would arise if Java allowed it?

2. What is the output when you run the `Test` class below ?

```
class A {
    String s;

    A(){
       s = "default string";
    }
    public String toString(){
        return "toString() from A returns: " + s;
   }
}

class B extends A {
   public String toString(){
     return "toString() from B returns: " + s;
   }
}

class Test {
   public static void main(String[] args){
      B   b  =  new B();
      A   a  =  b;
      System.out.println( a.toString() );
      System.out.println( b.toString() );
   }
}
```

3. (a) What is the output of the code below, when you run class `B` ?

   Hint: This is a tricky question. To answer it correctly, notice the variables `s` in `A` and `B` have nothing to do with each other. In particular, the output would not change if you were to rename either (or both) the `s` variables within classes `A` or `B`.

   (b) Would the compiler complain if the variable `b` in `main` were declared as type `A`?

   (c) Would the compiler complain the variable `b` in `main` were declared as type `I` ?

```
public interface I {
   public  String  m();
}

public abstract class A implements I {
   private String  s = "A: ";
   public  String  m(String s_arg){
      return s + s_arg;
   }
}

public class B extends A{

    private String  s = "B: ";

    public String m() {
       return s;
    }

    public static void main(String[] args){
        B   b  = new  B();
        System.out.println( b.m( " test ") );
    }
}
```

Note that `A` is indeed an abstract class since it doesn't implement the `m()` method from the interface `I`.

4. Which of the instructions in the code below does the compiler allow or not allow?

```
public  interface  I{
   public  void m(I  other);
}

public  class  A  implements  I {
   public  void m(I  a){
       A  myA = (A) a;
              :
              :
   }
}

public  class B implements I {
   public  void  m(I  x) {

      B  myB = (B) x;
      A  myA = (A) x;
      A    a = myB;
         myA = (A) myB;
           :
  }
}
```

5. Which (if any) of the instructions `(c)-(g)` of `Test` generate compiler errors? [There are no instructions `(a),(b)` because I wanted to avoid confusing with variables `a,b`.]

   What is the output of the program after removing any lines that cause compiler errors?

```
public class A {
   public  int   n  = 3;

   A( ){                       //  constructor
        System.out.println("A");
   }

   public void  foo() { System.out.println( n ); }
}

public class B extends A{
   public  int     n;

   B() { }                                      //  constructor
   B(int n){                                    //  constructor
      System.out.println( "B" );
      this.n = n;
   }
   int  foo(int n) {
           System.out.println(this.n) ;
           return n;
   }
}

public class Test {

    public static void main(String[] args) {
       int n = 2;
       A   a = new A();                          //          (c)
       a.foo();                                  //          (d)
       a.foo( 7 );                               //          (e)
       B   b  =  new B( 11 );                    //          (f)
       n = b.foo( 4 );                           //          (g)
    }
}
```

6. (a) Exactly one of the lines (1)–(4) in the `Test` class has a compiler error. Which one and why?

   (b) Assuming you have commented out the error line identified in (a), what is the output when you run the `Test` class? See lines `(1)-(4)`. Briefly explain why, using the terms "override" or "overload" when appropriate.

```
public class Test {
    public static void main(String[] args) {
        Vehicle  v         = new Vehicle();
                 v.driver  = "Lady Gaga";
        Movable  m         = new Car();
        System.out.println( v.move("Eminem") );      //    (1)
        System.out.println( v.move() );              //    (2)
        System.out.println( m.move("Snoop Dogg") );  //    (3)
        System.out.println( m.move() );              //    (4)
    }
}

public interface Movable {   public String  move();    }

public class Vehicle implements Movable{
    public  String   driver = "Rihanna";
    Vehicle(){}
    Vehicle(String driver){  this.driver = driver;  }
    public String move() {   return "vehicle driven by " + driver; }
    public String move(String driver) { return "vehicle driven by " + driver; }
}

public class Car extends Vehicle implements Movable{
    public String move(){    return "car driven by " + driver;   }
}
```

7. Suppose the classes below are all in the same package. See questions on next page.

```
public class Sharer{
   int    sum;
   String  ID;
   Sharer other;

   public Sharer(String ID, int sum){ this.ID = ID;  this.sum = sum; }

//  give half,  keep half
   void  share(int n){  other.sum += n/2;  this.sum += n - n/2;  }

   public String toString(){ return ID + " " + sum + " "; }
}

class Giver extends Sharer{

   public Giver(String ID, int sum) { super(ID, sum); }

   // give to other
   void share(int n) {   other.sum += n;  this.sum  -= n;  }
}

class Taker extends Sharer{

        public Taker(String ID, int sum) {  super(ID, sum);  }

   //  take from other
       void share(int n) {   other.sum -= n;  this.sum  += n;  }
}

public class Test {

    public static void main(String[] args) {
        Sharer a = new Giver("Geoff", 10);
        Sharer b = new Taker("Tina",  7);
        Sharer c = new Taker("Ted",  15);

        a.other = b;   b.other = c;   c.other = a;
        a.share(2);    b.share(4);    c.share(7);

        System.out.println( a.toString() +  b.toString() + c.toString());
    }
}
```

(a) What is the output when the `Test` class is run?

(b) Suppose we were re-define the visibility of the three fields of `Sharer` to be `private` so that the `Giver` and `Taker` no longer have access to these fields.

Rewrite the methods of `Sharer`, `Giver`, and `Taker` to be consistent with this new definition. You will need to add getters and setters.

8. Let's look at an example of how iterator works. We make a linked list of rectangles and make an iterator for the list.

```
LinkedList<Rectangle>  list = new LinkedList<Rectangle>();
Iterator<Rectangle>    iter = list.iterator();
```

Suppose we add a bunch of rectangle objects to the list and later we want to visit all the rectangles in the list and do something with them. We don't have access to the underlying data structure of the linked list.

Without the iterator, we would do this:

```
for (int i=0;  i < list.size();  i++){
    list.get(i)   //  do something with i-th rectangle
      :
}
```

which you know is inefficient. Using an enhanced for loop, you could do this:

```
for (Rectangle r: list){
   //  do something with rectangle r
}
```

In fact, the enhanced for loop translates into instructions from the `Iterator` interface. How would you rewrite the enhanced for loop using these instructions?

9. The `Object` class has a `clone()` method, but in fact it has an empty implementation `;` and object of the `Object` class is not allowed to invoke this method. (For another class to use this method, it must override it.)

Why does it not make sense to be able to clone an `Object` object? Hint: recall how `clone()` is supposed to be related to `equals(Object)`.

Solutions

1. When a method is invoked in a program, the method name is stated and the argument (if there is one) has a type. If two methods differ only by the return type, then there is no way in general for the compiler (at compile time) or the JVM (at runtime) to know which method should be used. For example, suppose we had two methods defined in the same class:

```
A  m(){ ... };          //   returns type A object
B  m(){ ... };          //   returns type B object
```

   Let `b` be a variable of type `B` and suppose we have an instruction:

```
Object  o = b.m();
```

   It is totally unclear which of the two methods should be used here.

2. The output is

```
toString from B:  default string
toString from B:  default string
```

   Why? The variable `a` references a class `B` object, and so `a.toString()` calls the method `toString()` from class `B`.

3. The output is:

   (a) `A: test`

   Why? The `m()` method from class `A` must be used here since the method `m` in `main` is passed a `String` argument and no such method signature exists in `B`. Thus, class `B` must inherit this method from its superclass `A`.

   Without the hint, you might have been confused as to whether or not inheritance implies that the `m(String ..)` code from A also belongs to class `B`. With the hint, you should see that the code does *not* belong to `B`. Thus, class `A`'s `m` method is used, and this refers to the string `s` from class `A`.

   (b) The object referenced by variable `b` invokes an `m` method with `String` argument. This method is found in `A` so there would be no problem in declaring `b` to be type `A`.

   (c) There is only one `m` method in interface `I` and this does not have a `String` argument. Thus, the compiler would give an error if you declared `b` to be of type `I`.

4.
```
public interface I{
   public void m(I  other);
}

public class A implements I{
   public void m(I  a){
       A  myA = (A) a;  //  Programmer expects type A  argument
              :         //  (or descendent of A).
              :         //  Compiler allows it, since A implements I.
   }
}

public class B implements I {
  public void m(I  b) {

      B  myB = (B) b;    //  Compiler allows it, since B implements I.
      A  myA = (A) b;    //  Compiler allows it, since A implements I.
      A    a = myB;      // compiler error (cannot convert from type B to A)
         myA = (A) myB;  // compiler error (cannot cast from B to A)
           :
  }
}
```

5. The compilation error occurs in (e). The problem is that the `A` class has no `foo` method with an argument, and `a` is declared to be of class `A`.

```
Output:
A             from (c)
3             from (d)
A             from (f)    <---  easy to miss this one (see comment below)
B             from (f)
11            from (g)
```

[**ADDED: Nov. 29**] Every constructor makes an implicit call to its super class constructor `super()`. Thus, the `B(int n)` constructor implicitly calls the constructor `A()`, that is, `super()`. Note that it would do by default so even if there were a constructor `A(int)` with a matching parameter. If there were an `A(int)` constructor, then the **B(int)** constructor would have to include a `super(int)` call (with `int` replaced by a value).

6. (a) The compiler error is in line (3). `m` is of declared type `Movable`, and the `move()` method of the `Movable` interface doesn't have a `String` argument.

   If you think that `m` is of type `Car`, then you are confusing the *type of the variable* `m` with the *type of the object* that `m` is referencing. The variable `m` is declared to be of type `Movable` and so the *compiler* only allows `m` to invoke methods that belong to the interface `Movable`. It doesn't matter that `m` has been initialized to reference some object whose class (`Car`) implements `Movable` and who has a `move(String)` method.

   I suggest you think of the instruction

   ```
   Movable  m         = new Car();
   ```

   as two separate instructions (and see comments for the instructions do):

   ```
   Movable  m;    //  Declare type of variable m.
   m = new Car(); //  Variable m temporarily references a new object.
   ```

   As the program proceeds, `m` might later reference other `Movable` objects and there may be other reference variables the reference that `Car` object.

   (b) (1) prints "vehicle driven by Eminem", since "Eminem" is the parameter passed to the overloaded `Vehicle.move(String)`

   (2) prints "vehicle driven by Lady Gaga", since `Vehicle.move()` uses the `Vehicle.driver` field, which has value "Lady Gaga".

   (4) prints "car driven by Rihanna", since `Car` objects inherit the `Vehicle.driver` field, which is initialized to "Rihanna". `Car.move()` overrides `Vehicle.move()`.

7. (a) `Geoff 1 Tina 13 Ted  18`

   (b)
```
// in the Sharer class

Sharer getOther(){ return  this.other; }
void   setOther(Sharer other){ this.other = other; }
int    getSum(){  return  this.sum;   }
void   setSum(int n){  this.sum  =  n;  }

// in the Giver class

void share(int n) {
   this.getOther().setSum( this.getOther().getSum() + n );
   this.setSum( this.getSum()  - n );  }
}

//  in the Taker class

void share(int n) {
   this.getOther().setSum( this.getOther().getSum() - n);
   this.setSum( this.getSum() + n );
}
```

8. Here is what the enhanced for loop translates into:

```
  while (iter.hasNext() ){
     r = iter.next();
     //   do something with rectangle pointed to by r
}
```

9. Think what the results of the second and third line should be.

```
Object  o = new Object();
o  == o.clone( );
o.equals( o.clone() )
```

   The expression in the second line should evaluate to false. The expression in the third line is supposed to be true for `clone` and `equals` but in the case of the `Object.equals(Object)` method, the third line would be false.