

The code in these exercises does not do anything useful, since the purpose here is for you to learn to apply the rules of inheritance, casting, polymorphism, etc., without getting “tricked” by your meaningful expectations of what the program should do (based on class, method, and variable names).

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

```
class A {
    String s;

    A(){
        s = "A default string";
    }
    public String toString(){ // "public" needed here to match
                               // signature of Object.toString()

        return "toString from A: " + s;
    }
}

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

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

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

*Hint:* This is a nasty trick question. To get it right, you need to 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 `v` in `main` were declared as type A?  
(c) Would the compiler complain the variable `v` 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 v = new B();
        System.out.println( v.m( " test "));
    }
}
```

3. Java has wrapper classes for primitive types. This allows you treat numbers and characters as objects. This is useful, for example, since you can pass the object as a parameter to a method who expects an `Object` as a parameter.

There is an abstract class `Number` and two of its subclasses are `Integer` and `Double`.

For the following sequence of instructions, specify which is upcasting or downcasting, whether the compiler allows the instruction and (if so) whether there is a runtime error.

```
Number n;  
Integer i;  
  
n = new Integer(3);    // ??  
i = (Integer) n;      // ??  
n = new Double(3.14); // ??  
i = (Integer) n;      // ??
```

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 those lines (if any) that cause compiler errors? *Be sure to mark which instructions produce which output.*

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

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

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

public class B extends A{
    public int    n;

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

## Solutions

1. The output is

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

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

2. 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, `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 `v` invokes an `m` method with `String` argument. This method is found in `A` so there would be no problem in declaring `v` 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 `v` to be of type `I`.

```
3. Number n;           // This is an abstract class. Check out Java API.
Integer i;
n = new Integer(3);   // Upcasting.
i = (Integer) n;     // Downcasting. Compiler allows it.
                    // No error at runtime since n is an Integer.
n = new Double(3.14); // Upcasting.
i = (Integer) n;     // Downcasting. Compiler allows it.
                    // Class exception at runtime since n references a
                    // Double whereas i is declared to be type Integer
```

Note: if the last line were replaced by

```
if (n instanceof Integer) // note keyword "instanceof"
    i = (Integer) n;
```

then the instruction would not be executed at runtime, so there would be no runtime error.

```

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
B           from (f)
11          from (g)

```