

**Question 4–5:** (Solution, p 5) Say you are to prove the following using our axiomatic scheme.

```
{n = 2k and k ≥ 0}
i := 0;
j := n;
while j > 1 loop
    j := j / 2;
    i := i + 1;
end loop;
{n = 2i}
```

What is the loop invariant that will allow you to write the proof? Do not write the proof, just the invariant. Partial credit will be given for partially correct invariants.

**Question 4–6:** (Solution, p 5) Distinguish proofs of total correctness against proofs of partial correctness.

**Question 5–1:** (Solution, p 5)

Consider the Java program at right.

- If Java used dynamic variable scoping, what would it output?
- If Java used static variable scoping, what would it output?
- Does Java use static or dynamic scoping?

```
static class Test {
    static int a = 1;

    int f() { return a; }

    int main() {
        int a = 2;
        System.out.println(f());
    }
}
```

**Question 5–2:** (Solution, p 5)

Consider the Ada program at right.

- If Ada used statically scoped variables, what would this program print?
- If it used dynamically scoped variables, what would it print?
- Which does Ada use?

```
with Text_IO; use Text_IO;
procedure Test is
    X : Integer;

    procedure A is begin
        X := 3;
    end A;

    procedure B is
        X : Integer;
    begin
        X := 4;
        A;
        Put_Line(Integer'Image(X));
    end B;
begin
    X := 2;
    B;
    Put_Line(Integer'Image(X));
end Test;
```

**Question 5–3:** (Solution, p 5) Many languages (such as Java, Ada, and ML) do static type checking, while others (such as PostScript, Scheme, and Smalltalk) use dynamic type checking.

- Name and explain an advantage of static type checking over dynamic type checking.
- Name and explain an advantage of dynamic type checking over static type checking.

## 2 Questions

---

**Question 5–4:** (Solution, p 5) Recall from class that FORTRAN uses static variable allocation, while most other languages (including Ada and Java) use stack-dynamic variable allocation.

- a. Name and explain an advantage of static variable allocation over stack-dynamic variable allocation.
- b. Name and explain an advantage of stack-dynamic variable allocation over static allocation.

**Question 5–5:** (Solution, p 5) Name and explain a reason why generics (such as generic packages in Ada or templates in C++) are a useful feature for a programming language to include.

**Question 5–6:** (Solution, p 5) Explain why generics make more sense as a feature in statically typed languages than in dynamically typed languages.

**Question 8–1:** (Solution, p 6) Fully parenthesize the following expression to illustrate the order in which Smalltalk will evaluate it.

2 + 3 factorial \* 4 gcd: 12 - 4

**Question 8–2:** (Solution, p 6) Suppose we have an Account class in Smalltalk, with an instance variable named `balance`.

- a. Give the definition of an instance method `withdraw:`, which reduces the balance by the given amount. If the withdrawn amount exceeds the balance, the balance should drop to 0.
- b. Using this `withdraw:` method, give the definition of an instance method `deductPenalty`, which should deduct 10 from the balance if the current balance is below 500.

**Question 8–3:** (Solution, p 6) Say you want to add a keyword instance method into Smalltalk’s built-in Integer class called “`max:`”, which returns the maximum value between the Integer instance to which the method is applied and the argument. For example, the code “`3 max: 5`” should return 5, while “`6 max: 2`” would return 6. Write the definition of this method to be added to the Integer class.

**Question 8–4:** (Solution, p 6) Name *two* significant and distinct differences between the object-oriented features of Java and Smalltalk. Do not address other differences, like their radically different syntax.

**Question 8–5:** (Solution, p 6) Describe the process that Smalltalk goes through in looking up a method when a message is sent to an object *A*. (Be sure to allow for the fact that the method may have been inherited by a superclass.)

**Question 8–6:** (Solution, p 6) How does a Smalltalk system store class methods internally?

**Question 9–1:** (Solution, p 6) Compilers for object-oriented languages must support *polymorphism*, where instances of a class may masquerade as instances of a superclass. For example, if Container is a parent class of Suitcase, then a Suitcase object reference can be placed into a Container variable.

```
Container bag = new Suitcase(100);
```

What can the compiler do to support polymorphism? (Virtual methods are not part of this question.)

**Question 9–2:** (Solution, p 6) Explain why virtual methods are problematic for a compiler of object-oriented languages, and explain how the VMT addresses the problem.

**Question 9–3:** (Solution, p 7) Suppose the following two class definitions.

```
class Clothing {
    boolean worn;
}

class Hat extends Clothing {
    int size;
}
```

Diagram the contents of a Hat class instance record.

**Question 9–4:** (Solution, p 7) Explain how the VMT works in generating compiled code for object-oriented languages.

**Question 9–5:** (Solution, p 7)

Consider the following Java class declaration, for which a compiler generates the VMT and CIR format given at right.

```
class A extends Object {
    int a;
    int b;

    int f() {
        return a;
    }

    int g() {
        return f() + b;
    }
}
```

*A\_VMT*

0	Object_VMT
4	A_f
8	A_g

*A's CIR format*

0	A_VMT
4	a variable
8	b variable

Suppose the compiler were to translate A's g method as follows.

```
A_g: movl 4(%esp), %edx # load this into edx
      pushl %edx       # call f (pushing this onto stack first)
      call A_f
      popl %edx        # pop this back into edx
      addl 8(%edx), %eax # add b onto what f returns
      ret
```

Why is this translation wrong?

**Question 10–1:** (Solution, p 7) Define the “diamond problem” with multiple inheritance.

**Question 10–2:** (Solution, p 7) C++ distinguishes *regular inheritance* from *virtual inheritance*. Explain this distinction.

**Question 10–3:** (Solution, p 7) Suppose we have the following sequence of declarations in C++.

```
class A { };
class B : public A { };
class C : public A { };
class D : public B, C { };
```

Note that the following would be legal in C++.

```
B *b = new D();
```

Why would C++ prohibit the following statement?

```
A *a = new D();
```

#### 4 Questions

---

**Question 10–4:** (Solution, p 7) Describe a problem with multiple inheritance that Java's concept of *interface* avoids.

**Solution 4–5:** (Question, p 1)  $n = j \cdot 2^i$  and  $j \geq 1$

**Solution 4–6:** (Question, p 1) A proof of total correctness includes arguments that the program terminates. A partial-correctness proof may omit this fact - it proves that *if* the program terminates, then it reaches its conclusion correctly.

**Solution 5–1:** (Question, p 1) **a.** 2  
**b.** 1  
**c.** Java uses static scoping.

**Solution 5–2:** (Question, p 1) **a.** 4, 3  
**b.** 3, 2  
**c.** Ada uses static scoping

**Solution 5–3:** (Question, p 1)

- a.** Programmer errors are detected at compile-time, so that debugging happens more reliably and faster. Program execution is faster, as type-checking for each operation is expensive.
- b.** It allows more flexibility to the programmer, so that variables can mean different things at different times.  
 The code tends to be simpler, as it does not require code to associate types with variables.

**Solution 5–4:** (Question, p 2)

- a.** It is marginally more efficient because the computer can use direct loads instead of indirect loads. It allows for variables within functions that can remember values from previous calls to the function.
- b.** It is much more conducive to recursion, because each recursive call gets its own set of variables. It is more space-efficient, as space is only allocated for the functions currently on the call stack.

**Solution 5–5:** (Question, p 2) A generic allows the programmer to write subprograms in which a type can vary. For example, you could write a `Sort` subprogram that works with an array of any type of data by using a generic. Without generics, in a statically typed language, you would need a separate `sort` subprogram for each type of array you might want to sort.

(A more minor, but still legitimate, advantage is that generics allow the programmer to write a generic subprogram that takes a constant as a parameter, and this constant could be compiled into by the compiler into individual instances of the generic subprogram. For example, we could have a generic function that multiplies a number by a generic parameter.

```
generic
  Mult : Integer
function Multiply(X : Integer) return Integer is begin
  return Mult * X;
end Multiply;

function Double is new Multiply(2);
function Triple is new Multiply(3);
```

The compiler will be able to optimize the `Double` routine to do a left-shift instead of a multiplication.)

**Solution 5–6:** (Question, p 2) The primary benefit of generics is to allow a programmer to write code that can accommodate a range of types. This is already possible in dynamically typed languages, since code can be ambiguous about type, and this is resolved at run-time.

## 6 Solutions

---

**Solution 8-1:** (Question, p 2)

```
(( 2 + ( 3 factorial )) * 4 ) gcd: ( 12 - 4 )
```

**Solution 8-2:** (Question, p 2)

```
withdraw: aNumber
    balance := balance - aNumber.
    balance < 0 ifTrue: [ balance := 0 ].
    ^ self

deductPenalty
    balance < 500 ifTrue: [ self withdraw: 10 ].
    ^ self
```

**Solution 8-3:** (Question, p 2)

```
max: other
    self > other ifTrue: [ ^ self ] ifFalse: [ ^ other ]

or:

max: other
    ^ self > other ifTrue: [ self ] ifFalse: [ other ]
```

**Solution 8-4:** (Question, p 2)

1. Everything's an object in Smalltalk, while Java has "primitive types," which are not objects.
2. Java provides data protection, while Smalltalk does not.
3. Java is statically typed, and Smalltalk is dynamically typed.
4. Smalltalk classes inherit their superclass's class methods, while Java classes do not.

**Solution 8-5:** (Question, p 2) Each object has a reference to the object representing its class. Suppose  $A$ 's reference is to  $B$ . The interpreter will go to  $B$  from  $A$  and ask  $B$  whether its method dictionary contains the requested method. If not, then from  $B$  it gets the class object representing  $B$ 's superclass, and it asks it whether it contains the requested method. It continues this up the hierarchy, stopping when the requested method is found (or if it exhausts the hierarchy with no success).

**Solution 8-6:** (Question, p 2) When you define a class in Smalltalk, the system creates two objects: a Class object  $A$  representing the class, and a Metaclass object  $B$ . Smalltalk remembers  $B$  as being the class of  $A$ . Any class methods are stored in the method dictionary held by  $B$  (and any instance methods are stored in the method dictionary held by  $A$ ). In this way, all methods are treated identically, whether they are instance methods or class methods: In all cases, to retrieve the method for a receiver, the system goes to the class of the receiving object and looks the method up in the method dictionary found there.

**Solution 9-1:** (Question, p 2) The compiler can ensure that the Suitcase CIR matches with the Container CIR, by maintaining the same CIR format at its beginning. Thus, any Container instance variables would appear in the Suitcase CIR before Suitcase-specific instance variables. If the compiler does this placement, then to support polymorphism it can simply copy the Suitcase CIR address into the Container variable.

**Solution 9-2:** (Question, p 2) When the program says to call a virtual method on an object, the compiler cannot be sure of the exact location of the method, because the object's actual class may be a subclass that overrides the method. To resolve this, the compiler generates a VMT for each class tabulating the virtual methods, and it allocates the first four bytes of every object to point to the VMT of its actual class. For each call to a virtual method, the compiler generates code to look into the object's VMT to determine where to find the beginning of the method.

**Solution 9–3:** (Question, p 3)

*Hat's CIR format*

0	Hat_VMT
4	worn value
8	size value

[It's significant that `size` comes after `worn` in this: since every `Hat` is a `Clothing` too, it must follow the `Clothing` CIR format at the beginning.]

**Solution 9–4:** (Question, p 3) The compiler generates a VMT for each class defined by the program, and it places a pointer to the VMT in each instance of the class. In the compiler-generated code for calling a virtual method on an instance, the CPU looks into the VMT for that instance and enters the method whose address is found within that VMT at the index associated with the method.

**Solution 9–5:** (Question, p 3) Because the `f` method in `A` is virtual, a subclass of `A` could override it. If the subclass doesn't also override `g`, then code calling the `g` method on an instance of this subclass would use `A`'s compiled `g` method. As translated here the computer would enter `A`'s `f` method, but the proper behavior would be to enter the subclass's `f` method.

**Solution 10–1:** (Question, p 3) A class can inherit from two classes, each of which inherits from a single shared class, as in the following class declarations.

```
class A { };
class B : public A { };
class C : public A { };
class D : public B, public C { };
```

The question of how `D` instances should incorporate the information defined by `A` is the diamond problem.

**Solution 10–2:** (Question, p 3) Normally, when a C++ class `D` has two parent classes that both extend a common class `A`, each `D` object includes two `A` sub-objects. This is *regular inheritance*. With *virtual inheritance*, each `D` object would include only one `A` sub-object.

**Solution 10–3:** (Question, p 3) This statement asks the C++ compiler to make a refer to the `A` sub-object of a newly created `D` object. Each `D` object has two `A` sub-objects - one by virtue of extending the `B` class, another by virtue of extending the `C` class. Because it is ambiguous which sub-object to select, C++ says that the compiler should reject such a statement.

**Solution 10–4:** (Question, p 4) Multiple inheritance leads to the possibility of multiply inheriting an instance variable, as in the diamond problem. Approaches for resolving such multiple definitions differ. Java prohibits instance variables in interfaces, so although a class may implement multiple interfaces, this cannot lead to the same problem as encountered via multiple inheritance.