
Reading

1. Chapter 13, Java.
2. The web page at <http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html> summarizes the main new features of Java 1.5 and provides links to additional information. The one-page (or slightly longer) descriptions reached through the links at the top of the page give about the right amount of information for this course.
3. *Optional:* The web page at <http://java.sun.com/developer/technicalArticles/releases/j2se15/> provides brief summaries of Java 1.5 features (at a level appropriate to this course) and also provides links to the JSRs at the end of the page.

Problems

1. Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. The designers of Java chose not to allow multiple inheritance. Therefore, a Java derived class may only have one base class. However, Java programs may contain interfaces (roughly, classes without an implementation) and a class may be declared to implement more than one interface. This question asks you to compare these two language designs.

This question asks you to consider the following kinds of movies. For sanity's sake, the list is non-exhaustive.

Movie, a class describing all kinds of movies

Action, a movie containing lots of explosions

Romance, a movie where romantic interest drives the plot

Comedy, a movie with largely humorous content

Mystery, a who-dunnit movie

Rescue, a hybrid action-romance movie, where the main character attempts to save his or her romantic interest from almost certain doom

Romantic Comedy, a hybrid romance-comedy with large amounts of both humorous and romantic content

Hollywood Blockbuster, an action-romance-comedy-mystery movie designed to please crowds

- (a) Draw a C++ class hierarchy with multiple inheritance for the above set of classes.
- (b) If you were to implement these classes in C++ for some kind movie-genre database, what kind of potential conflicts associated with multiple inheritance might you have to resolve?
- (c) If you were to represent this hierarchy in Java, what interfaces and classes would you use? Write your answer by carefully drawing a class/interface hierarchy, identifying which nodes are classes and which are interfaces. Note that there must be a class for each of the movie genres, but you may use any interfaces you require to preserve the relationships between genres. For example, one way of doing this would be to have the `Comedy` and `RomanticComedy` genres both implement some kind of `IComedy` interface.
- (d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.

2. Adding pointers to Java

Java does not have general pointer types. More specifically, Java has primitive types (Booleans, integers, floating point numbers, ...) and reference types (objects and arrays). If a variable has a primitive type, then the variable is associated with a location, and a value of that type is stored in that location. When a variable of primitive type is assigned a new value, a new value is copied into the location. In contrast, variables of reference types are implemented as pointers. When a reference-type variable is assigned, Java copies a pointer to the appropriate object into the location associated with the variable.

Imagine that you were part of the Java design team and you believe strongly in pointers. You want to add pointers to Java, so that for every type A , there is a type A^* of pointers to values of type A . Gosling is strongly opposed to adding an “address of” operator (like $\&$ in C), but you think there is a useful way of adding pointers without adding address-of.

One way of designing a pointer type for Java is to consider A^* equivalent to the following class:

```
class A* {
    private A data=null;
    public void assign(A x) {data=x;};
    public A deref(){return data;}
    A*(){};
};
```

Intuitively, a pointer is an object with two methods, one assigning a value to the pointer and the other dereferencing a pointer to get the object it points to. One pointer, p , can be assigned the object reached by another, q , by writing $p.assign(q.deref())$. The constructor A^* does not do anything because the initialization clause sets every new pointer using the null reference.

- (a) If A is a reference type, do A^* objects seem like pointers to you? More specifically, suppose A is a Java class with method m that has a side effect on the object and consider the following code:

```
A x = new A(...);
A* p = new A*();
p.assign(x);
(p.deref()).m();
```

Here, pointer p points to the object named by x and p is used to invoke a method. Does this modify the object named by x ? Answer in one or two sentences.

- (b) What if A is a primitive type, such as `int`? Do A^* objects seem like pointers to you? (*Hint*: Think about the code in part (a).) Answer in one or two sentences.
- (c) If $A <: B$, should $A^* <: B^*$? Answer this question by completing the following chart and explaining the relationship:

<i>Method</i>	<i>Type</i>
$A^*::assign$?
$B^*::assign$?
$A^*::deref$?
$B^*::deref$?

- (d) Can you generalize the issue discussed in part (c) to Java generics? More specifically, What might happen if you had a pointer generic? Based on the `Ptr` example, do you think it is correct to assume that for every generic class `Template`, if $A <: B$ then `Template(A) <: Template(B)`? Explain briefly.

3. Java Bytecode Analysis

Java programs are compiled into bytecode, a simple machine language that runs on the Java Virtual Machine. Note that it is possible that bytecode could be written by hand, or that compiled bytecode get corrupted when transmitted over the network. So, when a class is loaded by the JVM, it is first examined by the bytecode verifier. The verifier performs static analysis of the class to ensure that a program will not cause an unchecked runtime type error.

One kind of bytecode error that the verifier should catch is use of an uninitialized variable. Here is a code fragment in Java and some corresponding bytecode. We have added comments to the bytecode to help you figure out the effects of the instructions.

```
// Java Source Code
Point p = new Point(3);
p.Print();

// Compiled Bytecode
1: new #1 <Class Point>           // allocate space for Point
2: dup                           // duplicate top entry on stack
3: iconst 3                       // push integer 3 onto stack
4: invokespecial #4 <Method Point(int)> // invoke constructor, which pops
                                   // its argument (3) and one of
                                   // the pointers to p off stack
5: invokevirtual #5 <Method void Print()> // invoke print method, popping
                                   // the other pointer to p.
```

The first line of the Java source allocates space for a new `Point` object and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can be allowed only if the object has been initialized. It is easy to verify from this Java source that `p` is initialized before it is used.

Checking that objects are initialized before use in the bytecode is more difficult. The Java implementation creates objects in several steps: First, space is allocated for the object. Second, arguments to the constructor are evaluated and pushed onto the stack. Finally, the constructor is invoked. In the bytecode for this example, the memory for `p` is allocated in line 1, but the constructor isn't invoked until line 4. If several objects are passed as arguments to the constructor, there could be an even longer code fragment between allocation and initialization of an object, possibly allocating multiple new objects, duplicating pointers, and taking conditional branches.

To account for pointer duplication, some form of aliasing analysis is needed in the bytecode verifier. This problem will consider a simplified form of initialize-before-use bytecode verification that keeps track of pointer aliasing by keeping track of the line number at which an object was first created. When a pointer to an uninitialized object is copied, the alias analysis algorithm copies the line number where the object was created, so that both pointers can be recognized as aliases for a single object. Of course, if an instruction creating a new object is inside a loop, then there may be many different uninitialized objects created at the same line number. However, the bytecode verifier does not need to work for this case, since Java compilers do not generate code like this. We won't consider any cases with conditional branches in this problem.

Let's examine the contents of stack and any associated line numbers for alias detection after execution of each of the bytecode instructions from above. Note that we draw the stack growing downwards in this diagram:

After line:	Stack	line # where created	initialized
1	Point p	1	no
2	Point p alias for p	1 1	no no
3	Point p alias for p int 3	1 1 3	no no yes
4	Point p	1	yes

By using line numbers as object identifiers associated with each pointer on the stack, we keep track of the fact that both pointers on the stack point to the same place after line 2. So when the constructor invoked on line 4 initializes the alias for `p`, we recognize that `p` is initialized as well. Thus the `Print()` method is applied to a properly initialized `Point` object, and this example code fragment passes our simple initialize-before-use verifier.

(a) Consider the following bytecode:

```

1: new #1 <Class Point>
2: new #1 <Class Point>
3: dup
4: iconst 3
5: invokespecial #4 <Method Point(int)>
6: invokevirtual #5 <Method void Print()>

```

When line 5 is reached, there will be more than one uninitialized `Point` objects on the stack. Use the static verification method described above to figure out which one gets initialized, and whether the subsequent invocation of the `Print()` method occurs on an initialized `Point`.

You should draw the state of the stack after each instruction, using the original line number associated with any pointer to detect aliases.

(b) So far we have only considered bytecode operations that use the operand stack. For this problem we introduce two more bytecode instructions, `load` and `store`, which allow values to be moved between the operand stack and local variables.

`store x` removes a variable from the top of the stack and stores it into local variable `x`.
`load x` loads the value from local variable `x` and places it on the top of the stack.

In the following table, we have begun applying the initialize-before-use verification procedure described above to the code fragment in the left column. Note that we are maintaining information about aliasing and initialization state for local variable `x` as well as the contents of the operand stack. The contents of the stack correspond to that obtained just after the code in the left column is executed.

Code	local variable x		stack	
	line #	init?	line #	init?
1: new #1 <Class Point>	n/a	no	1	no
2: new #1 <Class Point>	n/a	no	1 2	no no
3: store x	2	no	1	no
4: load x				
5: load x				
6: iconst 5				
7: invokespecial #4 <Method Point(int)>				
8: invokevirtual #5 <Method void Print()>				
9: invokevirtual #5 <Method void Print()>				

Continue applying the procedure to fill in the missing information on lines 4-8. Based on the state information you have after instruction 7, is the `Print()` method on line 8 applied to a properly initialized object? What about the `Print()` method on line 9?

4. Java generic wildcard upper and lower bounds

Java generics can be written using type parameters or wildcards, which are like anonymous type parameters. Type parameters can be given an upper bound by writing `<T extends C>`, which means that the type parameter `T` must be a subtype of class `C`. In other words, the type parameter `T` has upper bound `C`. The same upper bound syntax can be used for wildcards, as in `<? extends C>`. For example, an object of `List<? extends Number>` is a list that contains objects which extend the `Number` class. For example, the list could be `List<Float>` or `List<Number>`. Reading an element from such a list is guaranteed to return a `Number`, but writing to the list is not generally allowed.

Wildcards can also have lower bound constraints, written using `super` instead of `extends`. A constraint `<? super C>` means that the wildcard type must be a *supertype* of class `C`. For example, an object of `List<? super Number>` could be a `List<Number>` or `List<Object>`. Reading from such a list returns objects of type `Object`, but any object of type `Number` can be added to the list.

This question asks about generic versions of a simple function (static method) that reads elements from one list and adds them to another. Here is sample non-generic code, in case this is useful reference in looking at the generic code below.

```
public static void addAll_nonGeneric(List src, List dest) {
    for (Object o : src) {
        dest.add(o);
    }
}
List listOfNumbers = new ArrayList();
List listOfIntegers = new ArrayList();
...
addAll_nonGeneric(listOfIntegers, listOfNumbers);
```

- (a) The simplest generic version of the `addAll` method uses an unconstrained type parameter and no wildcards.

```
public static <T> void addAll_0(List<T> src, List<T> dest) {
    for (T o : src) {
        dest.add(o);
    }
}
```

Suppose that we declare

```
List<Number> listOfNumbers = new ArrayList<Number>();
List<Integer> listOfIntegers = new ArrayList<Integer>();
```

and call `addAll_0(listOfIntegers, listOfNumbers)`. Will this call compile or will a type error be reported at compile time? Explain briefly.

- (b) With `listOfIntegers` and `listOfNumbers` defined as in the previous part of this question, will the call `addAll_1(listOfIntegers, listOfNumbers)` compile, where `addAll_1` is defined as follows:

```
public static <T> void addAll_1(List<? extends T> src, List<T> dest) {
    for (T o : src) {
        dest.add(o);
    }
}
```

Explain briefly.

- (c) With `listOfIntegers` and `listOfNumbers` defined as in the previous part of this question, will the call `addAll_2(listOfIntegers, listOfNumbers)` compile, where `addAll_2` is defined as follows:

```
public static <T> void addAll_2(List<T> src, List<? super T> dest) {
    for (T o : src) {
        dest.add(o);
    }
}
```

Explain briefly.

- (d) Suppose we want to change our function so that in addition to adding elements of one list to another list, we also return the last element added. Code for this static method is written below. Fill in each of the blanks with type parameters, wildcards, and/or constraints so that the code will type-check at compile time *and* the result of a call to `addAll_3` will have the best possible type.

```
public static <T> T addAll_3(List< _____ > src, List< _____ > dest) {
    T last;
    for (T o : src) {
        dest.add(o);
        last = o;
    }
    return last;
}
```

- (e) In the code in part d) above, suppose `src` has type `List<R>` and `dest` has type `List<S>`. What subtype/supertype relationships between `R`, `S`, and `T` are needed for the body of the method to be type-correct?
- (f) In your solution to part d) above, how do the constraints you wrote guarantee the subtype/supertype relationships between `R`, `S`, and `T` you listed in part e)?

- (g) Suppose that your friend comes across the following general coding suggestion on the web and asks you to explain. What can you tell your friend to explain the connection between the coding advice and principles of subtyping, showing off your understanding gained from CS242?

Get and Put Principle: use an extends wildcard when you only GET values out of a structure, use a super wildcard when you only PUT values in a structure, and don't use a wildcard when you both get and put.

5. Template expansion versus erasure

Java and C# implement generics in two different ways. Java generics are implemented using an “erasure” technique discussed in class that inserts casts and uses one run-time copy for all instances of the generic. Although C# is similar to Java in some respects, the C# implementation of generics may produce separate copies if several instances of a generic are used in a program. The C# implementation has some general similarities to the C++ implementation of templates.

- (c) Java chose the erasure and cast implementation of generics for several reasons. One was the large number of users running old Java VM's and a huge amount of legacy code. Why does the type erasure and casts implementation support both legacy code and old VM's better than an implementation that uses expansion.

- (b) At the Big Game, a Cal student says, “I think C# made the wrong choice when adding generics to their language: they've invalidated all their old code and VM's. Java does it right.”

The Stanford student coughs and says, “But what if your old code starts adding Strings onto your Vector<int>. This is wrong, but the error is not detected until you use the inserted String. It should be detected when the old code inserts a String into a Vector<int>.”

The Cal student replies “I have a solution to this problem. I just use this special MyVector written below. It will catch errors at run time right when a program tries to add a String into a Vector<int>, instead of waiting until the String is used as an Integer.”

```
class MyVector<T> extends Vector {
    void add (Object a) {
        try {
            T tmp = (T)a;
            super.add(tmp);
        } catch (ClassCastException c) {
            System.out.println("Error Detected");
        }
    }
}
```

The Stanford student rolls his eyes and tells the Cal student that his pathetic scheme would never detect those insertion errors when they happen, even at runtime.

Who is right? Explain why. (Hint: Include an explanation of what happens to T for the add function used in a MyVector<Integer> after Java erasure/casting.)

- (c) Translate the following code into the code that could run on an older VM without generics (i.e. Java 1.4 or earlier). Remember that ints are value types that are not subtypes of Object.

```
MyVector<int> a = new MyVector<int>();
a.add (5);
int j = a.lastElement();
```

Fill in the following code:

```
MyVector a = new MyVector();
```

a.add (_____ 5 _____)

int j = _____ a.lastElement() _____ ;

- (d) Sometimes Java's use of erasure and casting has very bizarre effects. In this case, write the console output of the following program.

```
class Container<T> {
    public T internal;
    public static Container lastInstance;
    Container(T value){
        internal=value;
        lastInstance=this;
    }
}

class Main {
    public static void main(String args[]) {
        Container<String> str = new Container<String>("Happy");
        Container<Integer> myint=new Container<Integer>(31337);
        System.out.println(str.lastInstance.internal.toString());
    }
}
```

Explain why it behaves this way in terms of generics and Java erasure.

- (e) C++ uses the expansion method for its templates instead of Erasure and inserted casts. What does the output of the same program translated into C++ look like.

Write the output and explain why it is different or why it remains similar even with a different template expansion mechanism.

- (f) It is sometimes useful to have a class that inherits from a template parameter like so:

```
class MyClass <T> extends T {
    ...
}
```

Why is this construct ineffective when used with erasure and casting?