Manifestations of Java Dynamic Linking
- an approximate understanding at source language level -

Sophia Drossopoulou and Susan Eisenbach
Department of Computing, Imperial College, LONDON
[sd,sue]@doc.ic.ac.uk

Abstract

Through dynamic linking, Java supports a novel paradigm for code deployment, which ensures fast program start-up and linking with the most recent version of code. Thus Java dynamic linking, gives support for software evolution, by supporting a piece of code A which uses a piece of code B, to link at run-time with a version of code B which was created *after* A was created.

Dynamic linking involves loading, verification, resolution and preparation of code. Programmers are normally not aware of the dynamic linking process. Nevertheless, in some situations, dynamic linking does manifest itself, and affects program execution and the integrity of the virtual machine. Therefore, there is a need for a description of dynamic linking at the level of the Java source language.

We provide such a description, and demonstrate the process in terms of a sequence of source language examples, in which the effects if dynamic linking are explicit.

1. Introduction

Java supports a novel paradigm for code deployment: Instead of linking a complete program *before* execution, the classes and interfaces making up the program are loaded and linked on demand *during* execution. Classes are verified before the creation of objects. Verification checks subtypes, and may require loading of further classes or interfaces.

The Java linking model is more complex than that usually found in programming languages, but it offers several advantages:

- Start-up is faster, since there is less code to load initially.
- Running programs link every time with the most up-to-date version of any utility.
- Error detection is lazier, since exceptions are only thrown if there is an attempt to execute unsafe code.

These advantages are obtained without compromising type safety. Usually, the Java linking process takes place implicitly, and, as long as "all goes well", it does not manifest itself, and does not affect program evaluation. Thus, dynamic linking is usually transparent to Java programmers.

Nevertheless, it is *not always* transparent to Java programmers, even if the programmers do not use low-level features such as reflection or explicit class loading. During program execution it is possible to encounter load errors or verification errors, and, if the verifier is switched off, type safety may be violated. Therefore, it is necessary for programmers to have an understanding of this mechanism, even if they do not use it explicitly.

The Java Language Specification [5] contains many small program examples that elucidate Java language features. But this is not the case for the dynamic linking process. In fact, Gilad Bracha, one of the authors, has stated that the bulk of the specification discusses constructs in the context of a consistent program, mostly assuming that the compile-time and run-time environments are the same, and only some exceptions are discussed, while the entire semantics should be explicitly parameterized by the environment, as defined by the class loader [1].

Dynamic linking is described only in chapter 12, **Execution**. That chapter, unlike most of the rest of the Specification, contains just a few examples, which demonstrate issues around initialization only. Also, although several papers formalize verification, and the overall dynamic linking process [4,6,8,9,11], to our knowledge, there exists no introduction at the source language level, and no simple examples.

A *complete* understanding of dynamic linking at source language level is impossible: for this one would need to understand the bytecode produced, and how the verifier works.

Nevertheless, we argue that an approximate understanding of dynamic linking at source language level is both possible and advisable. This is the aim of our paper: We explain the Java dynamic linking process in source language terms, and though a sequence of examples, where dynamic linking manifests itself. Most examples demonstrate one feature only.[1] Dynamic linking can be observed either through the trace of class loading, or through loader, verification and resolution exceptions, or through erroneous execution, when the verifier is turned off. All examples have been run both on JDK 1.3 and on JDK1.4.0-Beta2, in verbose mode (`-verbose`). In most examples it makes no difference whether the verifier is on or off; if it does, we state explicitly what state it should be in.

In section 2 we give a brief introduction to the different phases of dynamic linking, and their dependencies on each other. In sections 3, 4, 5, and 6 we give brief descriptions of each of the phases (verification, preparation, resolution, loading) with examples. In section 7, we put the phases together again in a larger example, and demonstrate the dependencies across phases. In section 8 we describe some difference between jdk1.3 and jdk1.4 in what concerns dynamic linking, and in section 9 we give a summary, draw conclusions and describe further work.

2. The phases of dynamic linking - Introduction

Java program execution is in terms of several different kinds of entities. These are:

- loaded (not yet verified) code,
- verified and prepared code,
- the expression being evaluated,
- the contents of the memory, ie the objects created during evaluation.

In accordance with Java terminology, we use the term *type* for interfaces and classes.

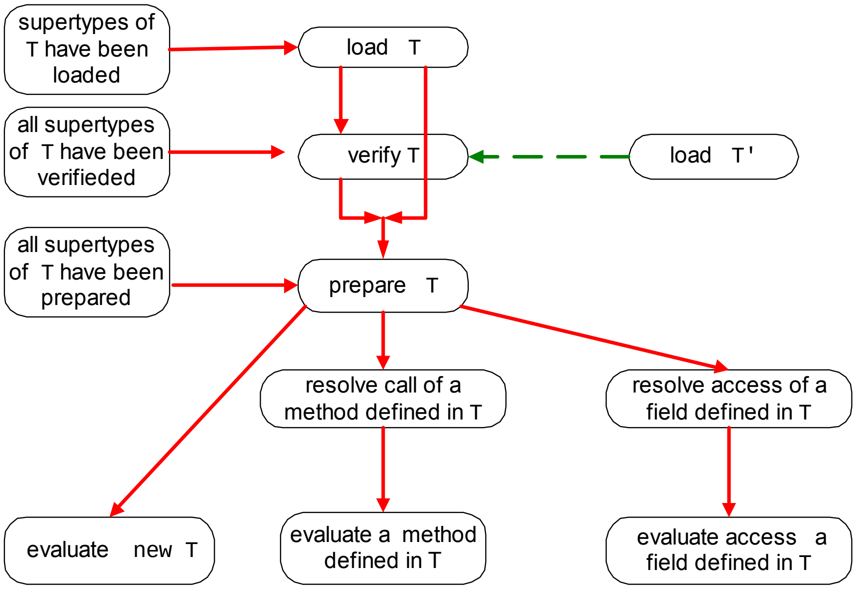
The Java Language Specification [5] distinguishes the following five phases of execution, which are applicable to different parts of a program (some apply to types, some apply to individual expressions):

- Evaluation of an expression is the "ordinary" execution of a Java program; it is unaffected by the linking processes.
- Loading of a type T finds the binary representation of T and turns it into a class object (possibly using several class loaders).
- Verification of a type T checks the format of the bytecode, that the destination of jumps are correct, that there is no stack overflow or underflow, and that the subtype relationships required in T are satisfied.
- Preparation of a type T creates method and field tables to avoid the necessity of searching superclasses at invocation time, as well as initialising static fields.
- Resolution of a symbolic reference (i.e. method call or field access) replaces symbolic references to other classes and interfaces and their fields and methods with direct references.

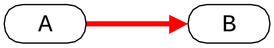
Evaluation is the part of execution that is unrelated to the dynamic linking process, *e.g.* conditionals, assignment *etc.*, and corresponds to these activities as found in most programming languages. The other four phases are directly related to dynamic linking.

The phases depend on each other, and a type needs to have gone through a certain phase in order to be able to go through the next phase. Also, evaluation of method call, field access, and object creation requires resolution of these expressions. These dependencies are demonstrated in the following diagram:

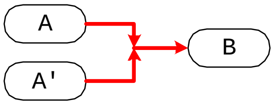
[1] In [4] we give a more extensive sequence of examples. In this paper we give more explanations, and all examples in the current paper run on both jdk1.3 and on jdk1.4, whereas in [4] some examples only run on jdk 1.3.


supertypes of T have been loaded
load T
all supertypes of T have been verifieded
verify T
load T'
all supertypes of T have been prepared
prepare T
resolve call of a method defined in T
resolve access of a field defined in T
evaluate new T
evaluate a method defined in T
evaluate access a field defined in T

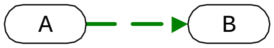
In the above diagram, the red arrows represent requirements, and the green, broken, arrow represents potential requirements, and the "alternative" red arrow represents alternative requirements, *i.e.:*



means B requires A



means B requires A or A'



means B may require A

The diagram shows, that for a type T,

- loading T requires all supertypes of T to have been loaded,
- verifying T requires T to have been loaded, all supertypes of T to have been verified, and may require loading of any further type T', if T' is part of a subtype relation which is required in some method body in T,
- preparing T requires all supertypes of T to have been prepared, and T to have been verified (unless the verifier is turned off),
- evaluation
  - method call of a method defined in T requires resolution of that method, which in its turn requires preparation of T;
  - field access of a field defined in T requires resolution of that field, which in its turn requires preparation of T,
  - creation of an object of class T requires preparation of T.

Thus, execution switches between the different phases for different kinds or terms. Loading, verification and preparation take place consecutively for a particular type, but not necessarily for all types together, nor do they take place without interruption: between loading and verification of a class there may be execution of other parts of a program. Also, verification may be skipped (the "alternative" red arrow), although in such a case, there can be no guarantees of type safety.

2.2. An example

The following example demonstrates the dependencies and interleaving of the phases:

```
class A{   }
class B extends A{   }
class C{
      void m1(A a){ }
```

```
            void m2( ){ m1(new B()); }
        }
        class Test{
            public static void main(String[] args){
                C c;
                c = new C();
                c.m2();                            }
        }
```

Execution of the code from above, in a setting where A, B and C have not been previously loaded, and where loading and verification is lazy, would activate the following linking phases (we distinguish between the case where the verifier is on, and the case where the verifier is off):

	code	linking phases, with verifier on	linking phases, with verifier off
1.	`c=new C();`	load C load A load B verify C prepare C evaluation: create new C object	load C prepare C evaluation: create new C object
2.	`c.m2();`	resolve method `void m2()` from class C evaluation: execute above method with receiver `c`	resolve method `void m2()` from class C evaluation: execute above method with receiver `c`
3.	`new B();`	verify A verify B prepare A prepare B evaluation: create new B object	load A load B prepare A prepare B evaluation: create new B object
4.	`m1(…);`	resolve method `void m1(A)` from class C evaluation: execute above method with receiver `c` and argument `new B`	resolve method `void m1(A)` from class C evaluation: execute above method with receiver `c` and argument `new B`

Namely, in the case where the verifier has been turned on:

1. For the creation of the `new C` object, the class C needs to be prepared, which requires C to be verified. That, in turn, requires verification of the method body for `m2`, where method `m1`, expecting an argument of class A is called with an argument of class B. This requires establishing that B is a subclass of A, which, in turn requires loading of A and B.
2. The method call `c.m2()` requires resolution of the method `void m2()` defined in class `C.`
3. For the creation of the new B object, the class B needs to be prepared, which requires B to be verified, and A to be prepared. Preparation of A also requires verification of A. Verification of classes A and B poses no further requirements, since these classes do not declare any methods.
4. The method call `m1(new B())` requires resolution of the method `void m1(A)` defined in class `C.` Execution of the method requires nothing further, since the method body is empty.

On the other hand, in the case where the verifier has been turned off:

1. For the creation of the new C object, the class C needs to be prepared, which requires C to be loaded
2. The method call `c.m2()` requires resolution of the method `void m2()` defined in class `C.`
3. For the creation of the new B object, the class B needs to be prepared, which in turn requires A, its superclass, to be prepared. This, again, requires loading of A and B.

4. The method call `m1(new B())` requires resolution of the method `void m1(A)` defined in class `C.` Execution of the method requires nothing further, since the method body is empty.

Additional clarification will be obtained from the following sections, where we discuss each phase individually.

3. Loading

Loading is the process of finding the binary of a class or interface of a given name. Classes and interfaces are loaded if they are required for execution, or if they are required in order to establish a subtype relationship required for verification of further classes.

The activity of a loader can be observed through the verbose flag in execution (`-verbose`) and through errors when classes are not found. When loading a class all its superclasses and interfaces are loaded. When loading an interface, its superinterfaces are loaded. Classes are only loaded once. Loading will fail, if the appropriate classes/interfaces cannot be found, the classes have class circularity or are badly formed. Then the errors `NoClassDefFoundError`, `ClassCircularityError` or `ClassFormatError` will be thrown.

Class loaders find the binary form of a class or interface and construct class objects to represent them. The Java Virtual Machine provides a bootstrap class loader. A program is loaded by one or more loaders.

User defined class loaders can be written. Class loaders can delegate the loading of the class or interface to another class loader. Type safe linkage is maintained by using both the loader and the class/interface name to identify a loaded class object. Loading constraints of the form <classname, loader1> = <classname,loader2> are imposed during preparation and resolution. This is needed in order to deal with problems described in [10]; it has been suggested in [8] and formalized in [9]. The examples in this paper do not deal with multiple loaders.

3.1 Manifestations of the loader

When loading a class, all its superclasses will be loaded. In the following example, creating an object of class B, requires B and its superclasses to be loaded:

```
class A{}
class B extends A{}
class Test{
    public static void main(String[] args){
        B b = new B();
    }
}
```

Partial output from executing `Test`:

```
[Loaded Test]
[Loaded B]
[Loaded A]
```

Similar examples can demonstrate that loading a class requires loading its interfaces and all its superinterfaces, *e.g.* those given in [4].

Also, if after compiling all the code, the file `B.class` is removed and `Test` is not recompiled, then execution will throw a `NoClassDefFoundError` exception:

```
[Loaded Test]
...  java.lang.NoClassDefFoundError: B at Test.main
```

Similarly, if after compiling all the code, the file `B.class` is corrupted (*e.g.* by altering it in a normal editor) and `Test` is not recompiled, then execution will throw a `ClassFormatError` exception:

```
[Loaded Test]
...  java.lang.ClassFormatError: B at Test.main
```

Finally, if a circularity is introduced, *e.g.* if after compiling all the code, a file containing

```
class B{}
class A extends B{}
```

is compiled in a separate directory, the resulting file `A.class` is copied into the original directory, and `Test` is not recompiled, then execution will throw a `ClassCircularityError` exception:

```
[Loaded Test]
java.lang.CircularityError:  B
```

4. Verification

> *The lesson is that an implementation that lacks a verifier or fails to use it will not maintain type safety and is, therefore, not a valid implementation.*
>
> [Java Language Specification]

The verifier works on the bytecode representation of a type. It checks that

1. the bytecode is structurally correct,
2. the destination of each goto is a bytecode instruction,
3. the stack will not overflow or underflow,
4. the type rules of Java are respected,

and throws a `VerifyError` or one of its subclasses, if one of the above requirements is not satisfied.

Requirements 1-3 are not directly visible to Java programmers and have more to do with establishing that the bytecode was, indeed, created by a Java compiler. Requirement 4 is visible to Java programmers, when they recompile part of their code without recompiling the remaining, dependent code. Our examples demonstrate how the verifier checks requirement 4.

The bytecode contains some, but not all, type information available in Java source code. It contains signatures of method and field declarations; and method calls and field accesses are enriched with appropriate descriptors. But the bytecode does not contain types of local variables. Thus, the verifier

- checks the inheritance hierarchy,
- checks subtypes for the receiver and arguments of methods,
- does not check subtypes for assignments to local variables.

Furthermore, the verifier is lazy (or optimistic), where subtypes are required for interfaces or between the same type, and so

- does not check if `t` is a subtype of `t`
- does not check if `t1` is a subtype of `t2`, when `t2` is an interface

When the verifier needs to establish that `t1` is a subclass of `t2`, it will check whether this is so, against the existing loaded and prepared code. If either of `t1` or `t2` has not been loaded yet, then the verifier will require the loading of these, and then check the subtype relationship.

Thus, the activity of the verifier can be observed, either by observing which classes are being loaded, or by observing the situations where verification fails, or, where it throws verification errors. The verifier can be turned off (`-noverify`) in which case the integrity of the JVM may be violated.

4.1 Manifestations of the verifier: classes loaded

When the verifier needs to check that `t1` is a subtype of `t2`, it may need to load `t1` and `t2`, if they have not already been loaded.

The following example is similar to the one shown in section 2.2: An object of class `C` is created at `*.` Class `C` contains methods `m1` and `m2`. Within `m2`, at `**`, the method `m1(),` which takes a formal argument of type A`,` is called with an actual argument of type B. The verifier needs to ascertain that B is a subclass of A (even though `m1` is never actually called), and for this, it loads the classes A and B`.` When the example is run with verification off, A and B are not loaded; this demonstrates that it is the verification that causes the loading.

```
class A{ }
class B extends A{ }
class C{
   void m1(A a){ }
   void m2( ){m1(new B()); }   // **
}
class Test{
   public static void main(String[] args){
       System.out.println( 1 );
       C c = new C();         // *
   } }
```

Partial output from executing `Test` with verification on:

```
[Loaded Test]
1
[Loaded C]
[Loaded A]
[Loaded B]
```

Partial output from executing `Test` with verification off:

```
[Loaded Test]
1
[Loaded C]
```

Partial output from executing `Test` with verification on, and with `A.class` removed:

```
[Loaded Test]
1
[Loaded C]
java.lang.NoClassDefFoundError: A
```

4.2 Verification of a class requires verification of its superclasses

In the following example, because of the argument of the method call at `**`, verification of class E requires classes A and B to have been loaded. Because of the subclass relationship, verification of class F requires verification of class E, and because of the argument in the method call at `***`, verification of class F requires classes A and C to have been loaded. Therefore, when the verifier is on, it is called both for E and for F before the creation of the F object at `*`, and in the process it loads classes A, B, and C. Note, that class D is *not* loaded.

```
class A{ }
class B extends A{ }
class C extends A{ }
class D{
   void m1(A a){  }
}
class E
   { void m2( ){ new D().m1(new B());  }  // **
}
class F extends E
   { void m3( ){ new D().m1(new C());  }  // ***
}


class Test{
   public static void main(String[] args){
       F f = new F();  }                    // *
}
```

Partial output from executing `Test`, with the verifier on:

```
[Loaded Test]
[Loaded E]
[Loaded F]
[Loaded A]
[Loaded B]
[Loaded C]
```

Partial output from executing `Test`, with the verifier off:

```
[Loaded Test]
[Loaded E]
[Loaded F]
```

4.3 Verification of a class does not require verification of all classes loaded

Verification of a class does not involve verification of the classes loaded when verifying that class. Here, the creation of an object of class E at * requires previous verification of class E. Because of the return statement at **, verification of class E needs to establish that D is a subclass of C, and so it loads D and C. However, verification of class E does not require verification of class C, and thus does not need to establish that B is a subtype of A, which is required for verification of the return statement at ***. Therefore, the following program does *not* load A and B.

```
class A{}
class B extends A{}
class C {
     A m1(){ return new B(); }         // ***
}
class D extends C{}
class E{
     C m2(){ return new D(); }         // **
}
class Test{
     public static void main(String[] args){
          E e = new E();               // *
     }
}
```

Partial output from executing `Test` with verification on:

```
[Loaded Test]
[Loaded E]
[Loaded C]
[Loaded D]
```

Partial output from executing `Test` with verification off:

```
[Loaded Test]
[Loaded E]
```

4.4 Verifier checks subtypes for arguments and receivers of method calls, for receivers of field access, for return values of method bodies, but does not check subtypes for assignments

The examples in sections 2.2 and 4.1 demonstrate that the verifier checks subtypes for the arguments of method calls.

Due to the differences in information between Java code and bytecode (bytecode does not contain the types of local variables), the verifier does not check subtypes for assignments to local variables, or to formal parameters - in fact, it cannot, since it does not have the information. This is shown in the following example, for the case assignments to local variables. The example could easily be extended to cover assignments to parameters.

The following example also demonstrates that the verifier checks subtypes for the receiver of a field access. The example can easily be extended to demonstrate the same for the receiver of a method call.

The creation of the new C object at line * requires verification of class C. In the body of class C the assignment of an object of class B to a variable of class A at *** does not require establishing that B is a subtype of A. Thus, verification of C does not load A and B.

On the other hand, the creation of the new D object at line * requires verification of class D. The use a of a B object at ****, in a context where an A is expected, does require establishing that B is a subtype of A. Thus, verification of D loads A and B.

```
class A{ int i; }
class B extends A{ }
class C{
    void m(){
       A a; a = new B();        // ***  verif. does not load A, B
    }
}
class D{
    A m2(){
        return new B();         // **** verif. loads A, B
    }
}
class Test{
    public static void main(String[] args){
        System.out.println(1);
        C c = new C();          // *
        System.out.println(2);
        D d = new D();          // **
        System.out.println(3);    }
}
```

Partial output from executing `Test` with verification on:

```
[Loaded Test]
1
[Loaded c]
2
[Loaded D]
[Loaded A]
[Loaded B]
3
```

Partial output from executing `Test` with verification off:

```
[Loaded Test]
1
[Loaded C]
2
[Loaded D]
3
```

4.5 Verifier is optimistic

The verifier is *optimistic*, i.e. does not check whether a type T is a subtype of itself, which would amount to checking existence of T. Also, the verifier does not check whether a class implements an interface.

In the next example at `*`, class `Test` contains a method `m` which returns an object of class A and which has return type A. As we saw in section 4.4, the verifier has to check subtypes for method bodies, and so verification needs to establish that A is a subtype of `A.` But, since the two types are identical, the verifier does not load A to check that subtype relation.

```
class A{   }
class Test{
    A m ()  { return new A(); }         // *
    public static void main(String[] args){}
}
```

Partial output from executing `Test` with verification on (or off):

```
[Loaded Test]
```

In the next example, the creation of the new `D` object at `*` requires verification of class `D`. In class D the method `m1`, with argument type `I`, is called at line `**` with an actual parameter of type class B. Once the verifier establishes that `I` is an interface, it does *not* attempt to establish that B implements `I`, and thus it does *not* load either A or B.

```
interface I{}
class A implements I{}
class B extends A{}
class D{
    void m1(I i){}
    void m2( ){ m1( new B());  }      // **
}
```

```
class Test{
     public static void main(String[] args){
        D d = new D();  }             //*
}
```

Partial output from executing `Test` with verification on:

```
[Loaded Test]
[Loaded D]
[Loaded I]
```

4.6 VerifyError, and the effect of bypassing verification

Verification guarantees wellformedness of the state of the virtual machine during execution. If the verifier is turned off, or fooled (earlier versions of the verifier had some loop-holes), then any part of the memory may be accessed, and the system may be brought to an inconsistent state.

In our example, a field from a superclass is accessed, and then the subclass is changed so as not to be a subclass of the original class. Then, with verification on, a `VerifyError` is thrown. However, with verification off, the program executes without error, and the field access leads to other (and possibly priviledged) parts of memory:

```
class A { int i = 0; }

class B extends A{  }

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

Class B is changed :

```
 class B{ int x = 888; }
```

The modified class B is compiled, and `Test` is not recompiled. Partial output from executing `Test` with verification on:

```
 [Loaded Test]
 [Loaded Test]
 [Loaded B]
...  java.lang.VerifyError: (class: Test, method: main...)
                Incompatible argument to method
```

Partial output from executing `Test` with verification off:

```
[Loaded Test]
[Loaded B]
[Loaded A]
888
```

Thus, with verification off, we ended up accessing the *wrong field*, and *no exception was raised.* This happens because the layout of superclasses is a prefix of the layout of subclasses. When accessing a field defined in a superclass, from an object belonging to a subclass, the offset of the field as defined in the superclass is used - this allows for faster field access, because resolution need only be applied the first time the field access is executed, and from then on, the same offset may be used. However, in the above example, `new B()`, the object from which the field `int i` is accessed, does not belong to a subclass of A, the class containing the field. Therefore, the offset for field `int i` for objects of class A, when applied to class B objects reaches unrelated parts of the memory. In fact, it can reach parts of the memory that do not even belong to the B object - one can easily extend the above example to demonstrate this.

5. Preparation

Preparation consists of determining the object layout, creating method lookup tables, creating class variables and constants and initialising them to their default values. A method lookup table contains enough information to allow the appropriate method to be invoked without having to look at superclasses.

Preparation throws an `OutOfMemory` exception, if there isn't enough memory to create the method lookup table or create class variables and constants.

The IBM Java system informs the user when preparation is occurring when the verbose option is on [12].

6. Resolution

Binary files can contain symbolic references to other classes, fields, methods and interfaces. These symbolic references are fully qualified. For fields and methods these references contain the name of the field or method, appropriate type information and the names of the class or interface where the declaration occurred. Resolution checks that the reference is correct and may replace it with a direct reference.

Resolution will fail if it attempts to resolve:

- a field or method of a given type declared in a certain type, but that type does not contain such a field or method,
- a method from an interface of a certain name, but this name belongs to a class,
- a method or field from a class of a certain name, but the name belongs to an interface.

Failure of resolution usually causes an `IncompatibleClassChangeError` or a subclass, such as one of the following:

- `InstantiationError`
  - change of class to an interface
  - change of an interface to a class
  - change of class to an abstract class
- `NoSuchFieldError`
- `NoSuchMethodError`
- `IllegalAccessError` - caught only if the verifier has been previously run
- `UnsatisfiedLinkError`

6.1 Changing the kinds of types - `InstantiationError`

Whenever a type is used in a setting that expects it to be of a certain kind, but it turns out to be of a different kind, an instance of a subclass of `IncompatibleClassChangeError` is thrown.

Thus, if an attempt is made to create an object of a class which is an interface, an `InstantationError` is thrown. This could happen if a class is compiled, a second class which creates an object of the first class is compiled, the first class is changed to be an interface and the first class is recompiled. The second class is not recompiled. For example, compile class `A` and `Test` :

```
class A{}
class Test{
    public static void main(String[] args){
        A a = new A(); }
}
```

Replace class A by an interface A:

```
interface A{}
```

Recompile A, but not `Test.` Partial output from executing `Test`:

```
[Loaded Test]
[Loaded A]
... java.lang.InstantiationError: A
```

Similar examples can demonstrate that if an attempt is made to create an object of a class which implements an interface that no longer exists as an interface, then an `InstantationError` is thrown. This could happen if an interface were compiled, a class that implements the interface is compiled, a second class, which creates an object of the first class is compiled, the interface is changed to be a class and then is recompiled. The other classes are not recompiled. Such an example is shown on [4].

Also, if an attempt is made to create an object of a class, which is abstract, then an `InstantiationError` is thrown. This could happen if a class were compiled, a second class which creates an object of the first class is compiled, the first class is changed to be abstract and the first class is recompiled. The second class is not recompiled. Such an example is shown on [4].

6.2 Missing members - NoSuchFieldError, NoSuchMethodError

If resolution attempts to access a field from a given type, but the type does not contain the field, then the exception `NoSuchFieldError` is thrown. This could happen if a class with a field were compiled, a second class with methods which access the field is compiled, the field from the first class is removed and the first class is recompiled. The second class is not recompiled. For example, compile class `A` and `Test`:

```
class A{
   int i = 1;
   int j = 2;
   int k = 3;
}
class Test{
   public static void main(String[] args){
        A a = new A();
        System.out.println(0);
        System.out.println("j = " + a.j);
   }
}
```

Change the type of field `j` in class A:

```
class A{
    int i = 1;
    char j = 'c';
    int k = 2;
}
```

Recompile A, but not `Test`. Partial output from executing `Test:`

```
[Loaded Test]
[Loaded A]
0
... java.lang.NoSuchFieldError: j
```

Note, that neither of the fields `i` preceding `j`, or k following `j`, or `String j`, were confused for the field `int j.`

A similar test (*e.g.* given in [4]) can demonstrate how resolution attempting to access a field from a given type, which does not contain that field, throws the exception `NoSuchFieldError.`

7. Putting it all together - a larger example

In the following example we demonstrate several phases, and the dependencies across them

```
class A{
    public void m2( ){ }
}
class B extends A{
    public void m3( ){  A a =  new C();   a.m2(); }
}
class C extends A{ }
class D{
    public void m1( ){ }; }
class Test{
    public static void main(String[] args){
         new D().m1();
         new A().m2();   }
         void g( ){  A a =  new B(); a.m2();    }
}
```

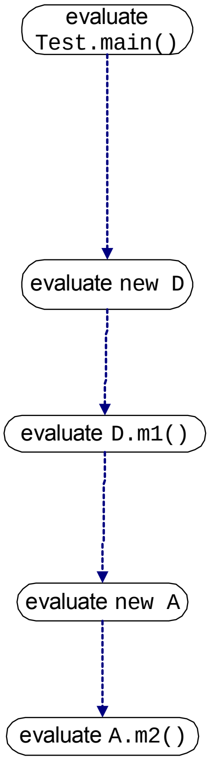
Execution of the program `Test.Main` requires the following evaluation steps to be applied in the following order:

```
Test.main(),
new D(),
D.m1(),
new A(),
A.m2()
```

The above steps are a straightforward reflection of the code, and do not consider the linking phases involved.

The order of execution is shown in the figure in the right, where the blue, broken arrow indicates the order of evaluation, *i.e.*

means *A* followed by *B*



Furthermore, as we discussed earlier, evaluation steps have their own requirements from the linking phases. For example, calling `Test.main()` requires `Test` to have been prepared, which requires `Test` to have been verified. Verification of `Test` requires `Test` to have been loaded, and also requires B to be a subclass of A. Establishing the latter requires A and B to have been loaded (although verifiers may do this by posting constraints instead). Loading B requires A to have been loaded previously.

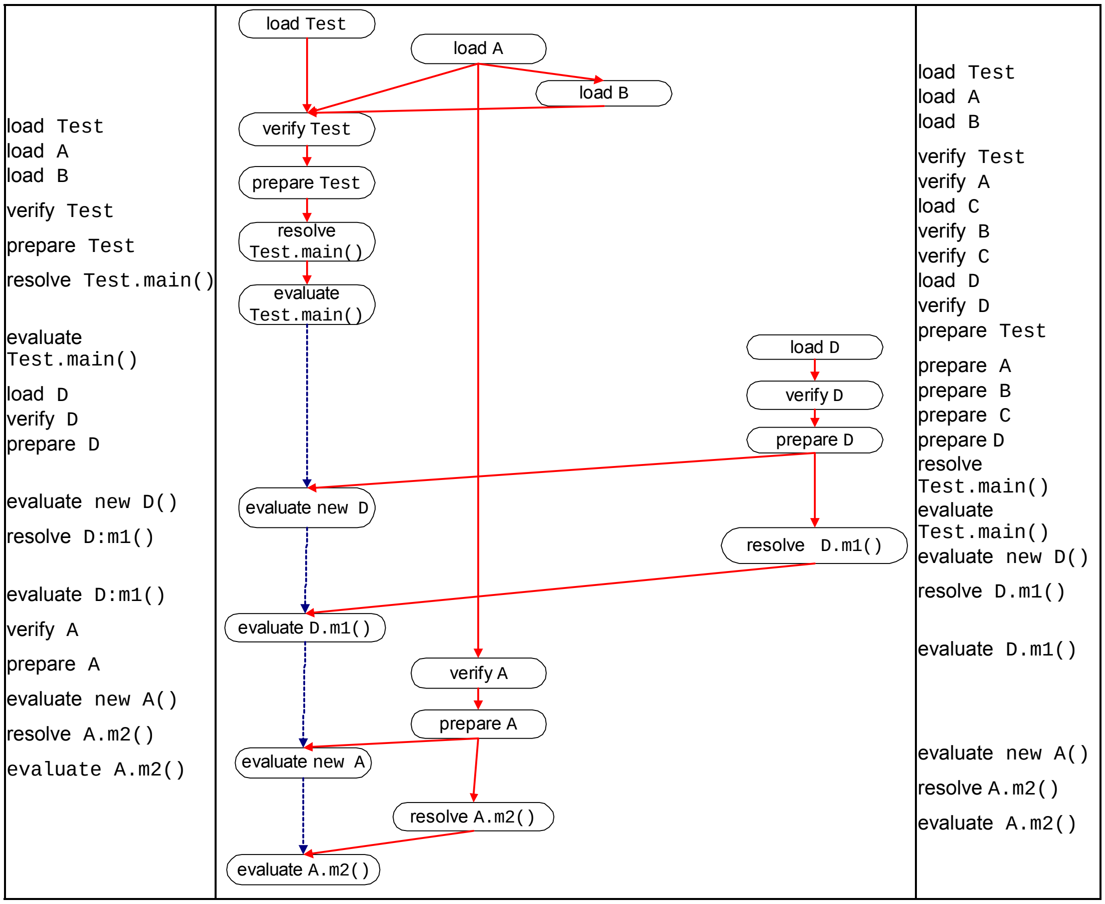
These, and the remaining linking related dependencies are illustrated in the centre of the following figure, where we only consider the case where verification is on. Here again, the blue, broken blue arrows express constraints imposed by the program code, whereas the read arrows express constraints imposed by the dynamic linking process, *i.e.*

means *B* requires *A*

means *A* followed by *B*

The Specification does not totally constrain the execution sequence. In the interrelationship example, class B is loaded for the verification of class `Test` but it need not be verified. Also, class C, although mentioned in class B doesn't need to be loaded if class B has not been verified. Two possible execution sequences are:

Lazy execution	Dependencies across phases	Eager execution



8. Differences between jdk1.3 and jdk1.4

The jdk1.3 and jdk1.4 approaches to dynamic linking are identical. However, some of the examples, *e.g.* several of those appearing in [4] behave differently when using jdk1.3 than when using jdk1.4. This is so, because the jdk1.4 compiler stores different type information in the bytecode representing field access and method call, then the jdk1.3 compiler. Therefore, different subtype relationships need to be checked for bytecode produced by the jdk1.4 compiler, than for bytecode produced by the jdk1.3 compiler.

Namely, the jdk1.3 compiler stores with the field access bytecode, the class where the corresponding field was declared. On the other hand, the jdk1.4 compiler stores with the field access bytecode, the type of the expression whose field is being accessed. Therefore, the type stored with field access bytecode produced by the jdk1.4 compiler is a subtype of (and possibly identical to) the type stored with field access bytecode produced by the jdk1.3 compiler. Similar differences exist for the bytecode for method call.

The following example demonstrates the difference for the case of field access, and its effect on dynamic linking.

The field access on line ** is translated by the jdk 1.3 compiler into bytecode where the receiver is expected to be of class A, or subtype – namely A is the class that contains the field definition of i. Therefore, verification of line ** needs to establish that B is a subclass of A, and so it loads both B and A.

On the other hand, the field access on line ** is translated by the jdk 1.4 compiler into bytecode where the receiver is expected to be of class B, or subtype – namely B is the type of the expression new B(). Therefore, verification of line ** needs to establish that B is a subtype of itself, and, as we showed in section 4.5, this does not need to load the class B (nor A).

```
class A{ int i; }
class B extends A{  }
class C{
    int m1( ){ return new B().i ; }  // **
}
class Test{
    public static void main(String[] args){
         System.out.println( 1 );
         new C();                          // *
         System.out.println( 2 ); }
}
```

Compilation of `Test` with jdk1.3, and partial output from executing `Test` with verifier on:

```
[Loaded Test]
1
[Loaded C]
[Loaded A]
[Loaded B]
2
```

Compilation of `Test` with jdk1.4, and partial output from executing `Test` with verifier on:

```
[Loaded Test]
1
[Loaded C]
2
```

9. Summary

Each dynamic linking phase has some requirements from the previous phases, and in addition it performs some checks.

The requirements from the previous phases determine which other phases will be activated (if they were not already activated at some other part of the program). For example, verification of a class will activate loading of that class, unless the class was already loaded previously.

The checks of the phase determine whether it will complete successfully. For example, if verification requires some subtype relationships that do not hold, then the verifier will throw a `VerifyError`.

We summarize this in the following table:

Phase	Requirements	Possible Exceptions

evaluation of term	Method call and field access require resolution of method/field
	`new` C requires preparation of C
		If language rules broken, then
		`NullPointerEexception,`
		`ArrayStoreException`
		`ArrayIndexOutOfBoundsException,`
		`ArithmeticException` *etc.*, *etc.*,...
		User defined exceptions may be explicitly thrown

loading of T		If a type is not found, or badly formed, then:
		`ClassCircularityError`
		`ClassFormatError`
		`NoClassDefFoundError`

verification of T	T to have been loaded, and T's superclasses verified	If verification not successful, ie subtypes required in T's body do not hold, then: `VerifyError` or subclasses
preparation of T	T to have been loaded or verified (verification required only if verifier is on)	If there is not enough memory for preparation to take place, then `OutOfMemoryException`

resolution of symbolic reference	The reference to belong to an already prepared type	If symbolic reference is invalid, then
		`IncompatibleClassChangeError`
		`IllegalAccessError`
		`InstantiationError`
		`NoSuchFieldError`
		`NoSuchMethodError`
		`UnsatisfiedLinkError`

10. Conclusions and Further Work

Java dynamic linking affects soundness of the virtual machine state and, ultimately, security. Also, its effects can be visible to source language programmers. Therefore, it is important to have a presentation of the mechanisms and dependencies of the phases of Java dynamic linking in terms of the source language. In the current paper we attempt such a presentation.

We have discussed the dependencies across phases, and each phase separately. We have given a series of source language programs, and we have shown the effects of dynamic linking both in terms of the program execution, the loading output in verbose mode, and also in terms of the exceptions throw. We have described the alternatives between lazy and eager strategies, and the compiler versions.

In further work, we want to extend the scope to cover multiple, user defined loaders, and the constraints imposed during resolution. We would also like to give an account of the design space for the tasks and dependencies across the five phases, and give a rationale for the current decisions.

Acknowledgements

In the early stages of this work we collaborated closely with David Wragg. We are grateful to Mark Skipper, Andrew Phillips and Vishnu Kotrajaras for their feedback, which helped us to create a simple presentation. We are very grateful to Gilad Bracha for his constant and critical support.

References

[1.] Gilad Bracha, *Adventure in Computational Theology: Selected Experiences with the Java(tm) Programming Language*, Invited talk at ECOOP Workshop on Formal Techniques for Java Programs, Budapest, June 2001, and private communication.

[2.] Alessandro Coglio and Allen Goldberg, *Type Safety in the JVM: Some Problems in Java 2 SDK 1.2 and Proposed Solutions*, 2001. To appear in Concurrency: Practice and Experience.

[3.] Sophia Drossopoulou, *Towards a model of Java dynamic linking and verification*, Harper, R., (Ed.): (2001) Types in Compilation, Third International Workshop, TIC 2000, Montreal, Canada, September 21, 2000. Revised Selected Papers, LNCS 2071, Springer Verlag, 2001

[4.] Sophia Drossopoulou, Susan Eisenbach, *Observing the Dynamic Linking Process in Java*, Web pages at: http://www-dse.doc.ic.ac.uk/projects/slurp/, 2001

[5.] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *The Java Language Specification*, Addison Wesley, 2nd Ed (31 July, 2000).

[6.] Thomas Jenssen, Daniel LeMetayer, Tommy Thorn. *Security and Dynamic Class Loading in Java*: *A Formalisation*

[7.] Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, Addison Wesley Longman Publishing Co, 2nd Ed (1 July, 1999).

[8.] Sheng Liang, Gilad Bracha, *Dynamic Class Loading in the Java Virtual Machine*, Proc. OOPSLA 1998, October 1998.

[9.] Zhenyu Qian, Allen Goldberg, Alessandro Coglio, *A Formal Specification of Java Class Loading*, Proc. OOPSLA 2000, October 2000.

[10.] Vijay Saraswat, *Java is not type-safe,* Web pages at: http://www.research.att.com/~vj/main.html, 1997.

[11.] Akihito Tozawa and Masami Hagiya, *Careful Analysis of Type Spoofing,* Java Informationstage, Spinger Verlag 1999.

[12.] IBM Java Developer Kits; from http://www-106.ibm.com/developerworks/java/jdk/index.html