Advanced Issues in Object Oriented Programming
- implementation questions, part iii -
Java Dynamic Linking
Thank you to Alex Buckley

Usually, Java linking is transparent to programmer: As long as it "goes well", it does not manifest itself. But

- Program execution may encounter load or verification errors, and throw exception.
- If the verifier is switched off or fooled, then type safety may be violated.
- Verbose execution exposes linking process.

We give an informal introduction to this mechanism, and discuss examples; we do *not* attempt a formal system.

The phases of Java execution

Java distinguishes following five phases of execution:

- Evaluation of an expression is the "ordinary" execution, unaffected by inking,
- Loading of a type T finds binary representation of T (T.class)
- Verification of a type T checks format of bytecode, the destination of instructions, stack/local -overflow or underlfow, and types as required in field access or method calls in T.
- Preparation of a type T creates method and field tables, and initializes static fields.
- Resolution of a symbolic reference (ie method call or field access) replaces symbolic references to other classes and interfaces and their fields and methods with direct references.

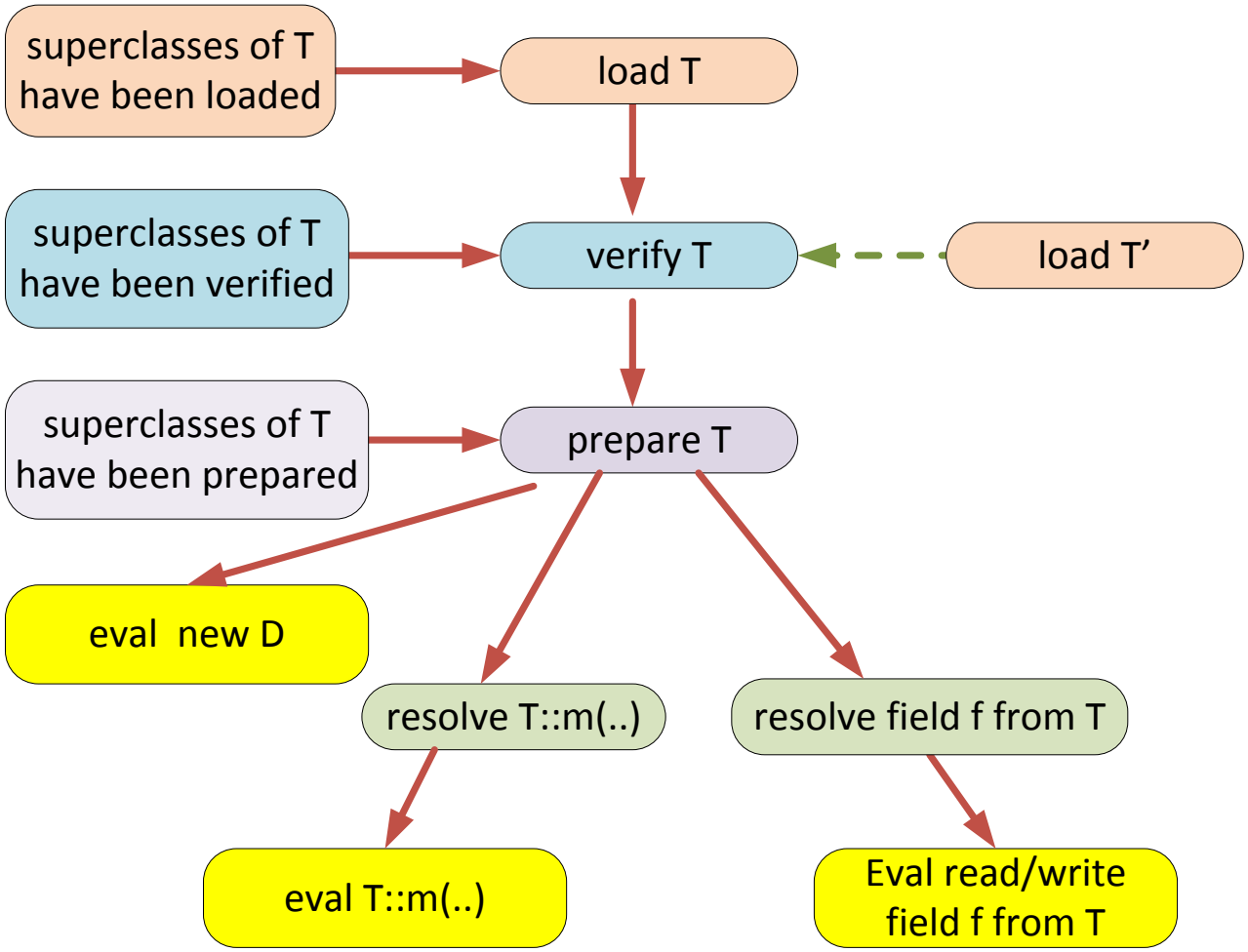
Note: phases applicable to different kinds of entities, “type” stands for class or interface.

Granularity and poss. Exceptions of the Phases

phase	granularity	poss. exception
Loading	type	`ClassCirculrityErr, NoClassDefFoundErr, ...`
verification	all method bodies of class	`VerifyError`
preparation	class	
resolution	Symbolic reference	`IncompatibleClassChangeError, IllegalAccessError, NoSuchFieldError, ...`
evaluation	term	`NullPointrExcept, ArrayStreExcept, ArrIndxOutOfBndsExcept, ...`

Dependencies across phases

Types go through phases consecutively, but not uninterruptedly. A phase for one type may require other phase for another type:



In the above diagram,

- red arrows represent requirements,
- green, broken, arrow represents *potential* requirements.
- The verification phase may be switched off.

Thus, execution switches between the different phases for different kinds or terms.

Notice that resolution of a method or field defined in class T may take place when executing code from a different class T'.

First example, putting it all together

Example 1

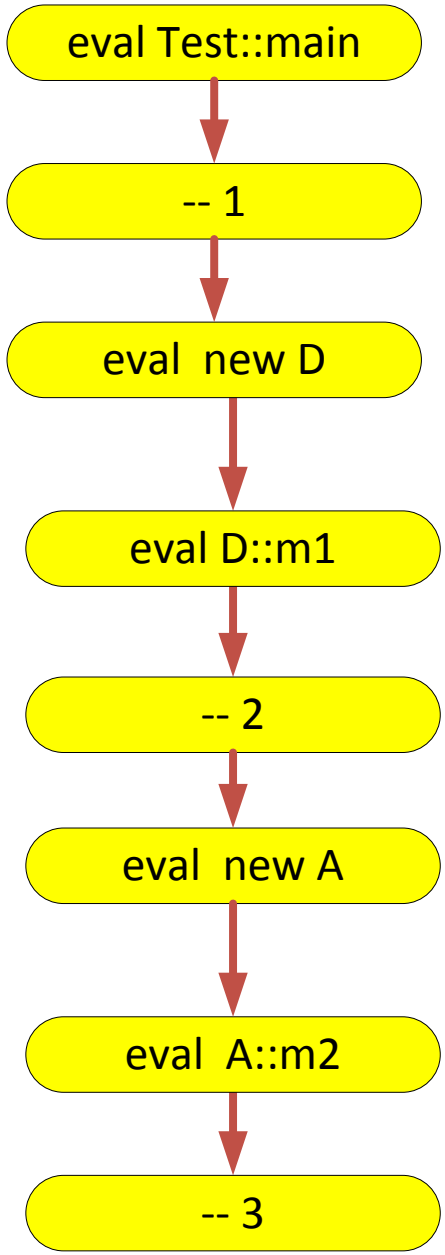
```
class Test{
   A f;
   public static void main(String[] args)
      {  System.out.println(" -- 1 "); new D().m1();
         System.out.println(" -- 2 "); new A().m2();
         System.out.println(" -- 3 "); }
   void g( ){ this.f = new B(); }
}
class A{
   public void m2( ){ }
}
class B extends A{
   public void m3( ){ Test t=new Test(); t.f= new C();}
}
class C extends A{ }
class D{  public void m1( ){ }  }
```

Execution of `Test.main()` requires:

call `Test.main()`,
followed by `-- 1`,
followed by `new D()`,
followed by `D::m1()`,
followed by `-- 2`,
followed by `new A()`,
followed by `A::m2()`,
followed by `-- 3`.

Above steps are a straightforward reflection of the Java code, and do not consider linking

The notation `ClassC::meth()` denotes method `meth`, which takes no arguments, and is defined in class `ClassC`.



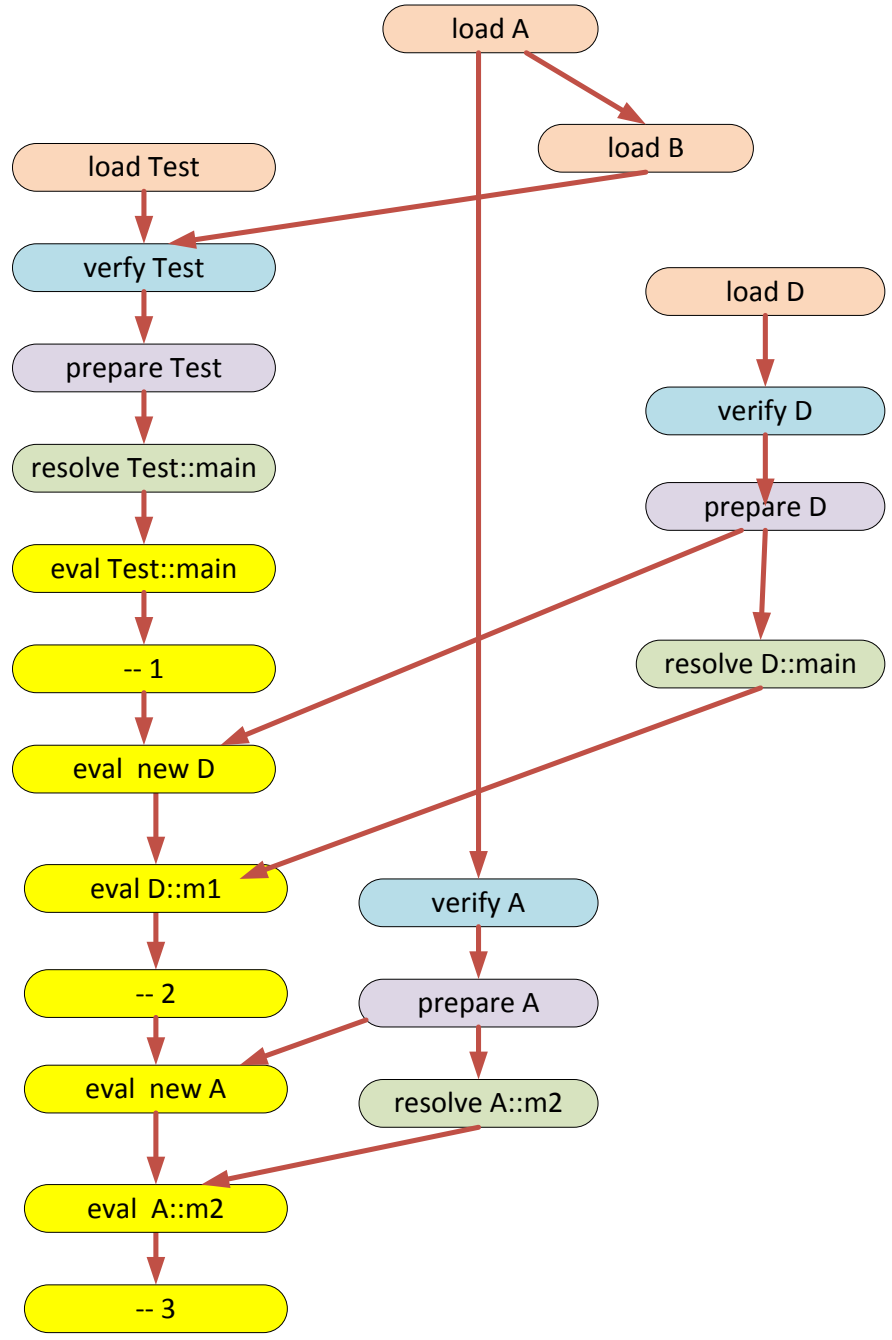
Considering linking phases requirements (verifier on):

Call `Test.main()` expects `Test` previously prepared; that expects `Test` previously verified.

Verification of `Test` requires `Test` to have been loaded, and `B` to have been established to be a subclass of `A`. The latter requires `A` and `B` to have been loaded.

Loading `B` triggers loading of `A`.

Note: `B` loaded for verification of `Test`, but not verified. Also, `C`, mentioned in `B`, but is not loaded.

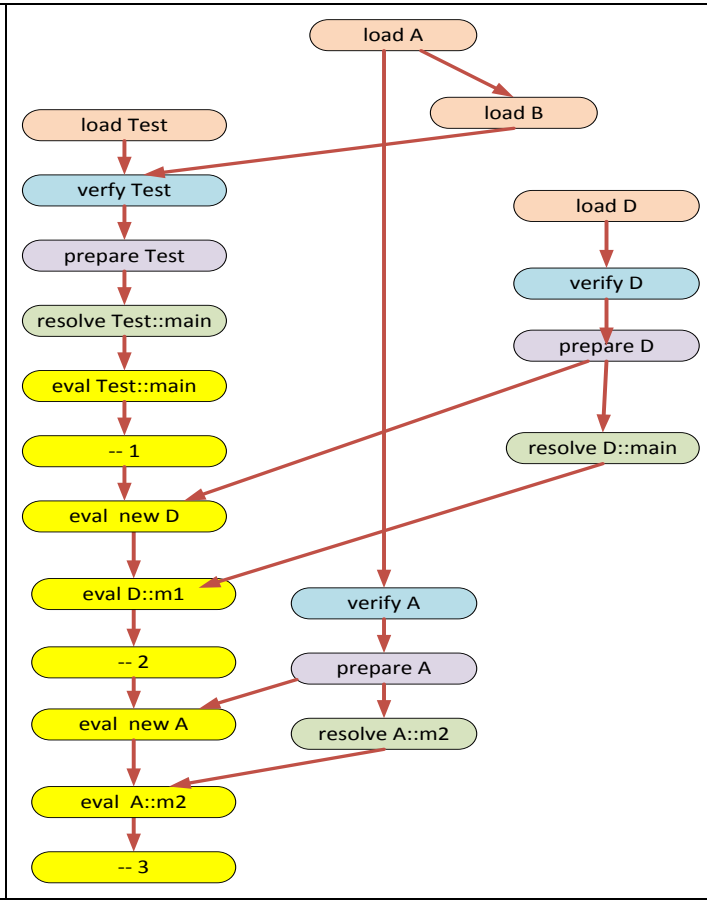


The laziest execution (adopted by SUN implementations) would be:

```
load Test, load A, load B, verify Test, prepare Test, resolve
Test.main(), evaluate Test.main(), load D, verify D, evaluate new
D(), resolve D.m1(), evaluate D.m1(), verify A, prepare A,
evaluate new A(), resolve A.m2(), evaluate A.m2().
```

Indeed, verbose execution with *verifier on* produces:

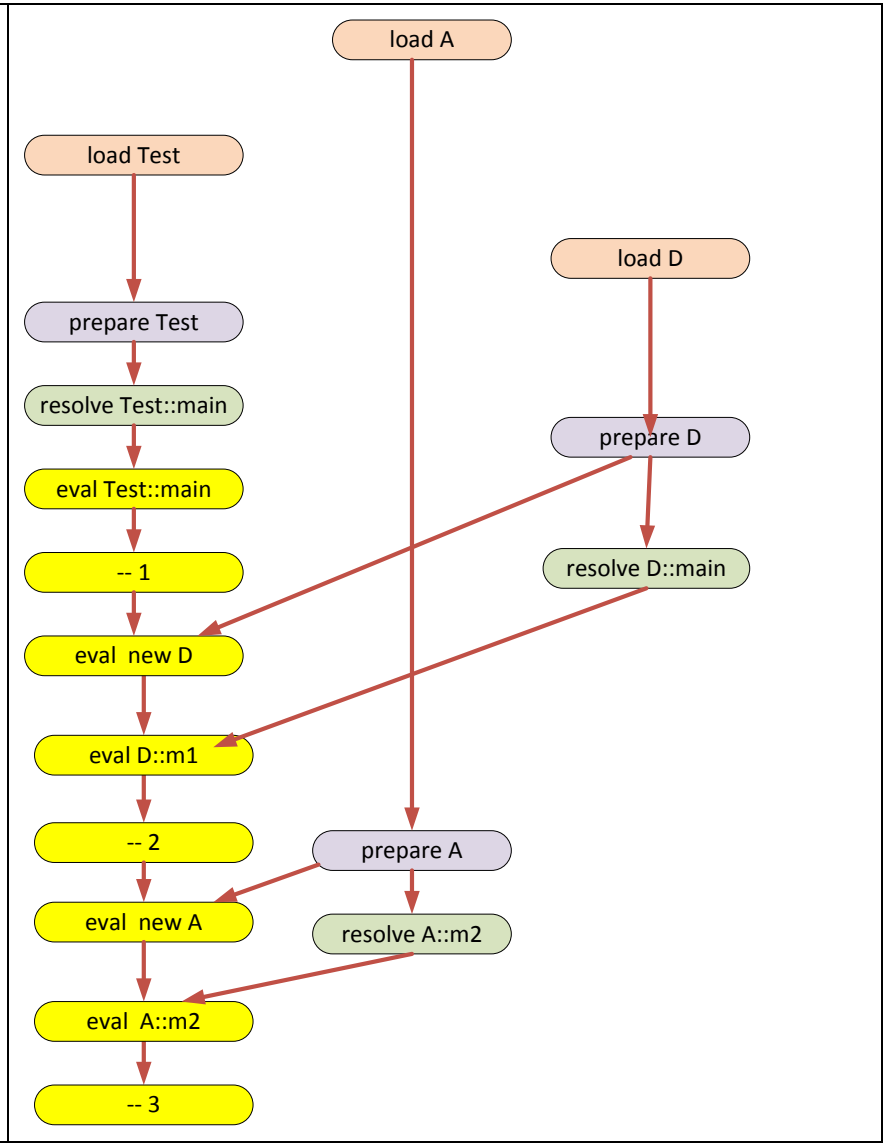
```
...
[Loaded Test]
[Loaded A]
[Loaded B]
-- 1
[Loaded D]
-- 2
-- 3
```



Java allows any other execution that satisfies the constraints. We do not discuss this here.

On the other hand, verbose execution with *verifier off* produces:

```
…
   …
[Loaded Test]
-- 1
[Loaded D]
-- 2
[Loaded A]
-- 3
```



Verifier cheks and manifestation

Verifier manifests itself through loading, and through verification errors when it fails.

Verifier checks

- subtypes for arguments of method calls
- subtypes for assignment to fields
- subtypes for return values of method bodies

Verifier does not check

- subtypes for assignment to local variables (why?)
- subtypes involving interfaces (why?)
- subtypes between the same type (why?)

Verifier does not check the existence of methods or fields

The verifier also checks (but not as “visibly”):

- subtypes for receivers of method calls
- subtypes for receivers in field access

Example 2

```
class A1{}
class B1 extends A1{ }
class C1{
    void m1(A1 a){ }
    void m2(B1 b){ this.m1(b); }
                // **** verif. loads
}
class A2{}
class B2 extends A2{ A2 f;  }
class C2{
   void m (B2 b){ b.f = new B2(); }
                // **** verif. loads
}
class A3{}
class B3 extends A3{ }
class C3{
   A3 m ( ){ return new B3(); }
                // **** verif. loads
}
```

```
class A4{}
class B4 extends A4{ }  class D4 extends A4{ }
class C4{
     void m2( ){ A4 a; a = new D4(); a = new B4(); }
                  // **** verif. loads
}

class A5{}
class C5{
     A5 m ( ){ return new A5(); }
                  // **** verif. loads
}
class Test{
    public static void main(String[] args){
        System.out.println(1);  new C1();
        System.out.println(2);  new C2();
        System.out.println(3);  new C3();
        System.out.println(4);  new C4();
        System.out.println(5);  new C5();
        System.out.println(6);    }
}
```

Partial output from executing `Test` with verification on:

```
[Loaded Test]
1
[Loaded C1]
[Loaded A1]
[Loaded B1]
2
[Loaded C2]
[Loaded A2]
[Loaded B2]
3
[Loaded C3]
[Loaded A3]
[Loaded B3]
4
[Loaded C4]
5
[Loaded C5]
 6
```

Partial output from executing `Test` with verification off:

```
[Loaded Test]
1
[Loaded C1]



2
[Loaded C2]



3
[Loaded C3]



4
[Loaded C4]
5
[Loaded C5]
  6
```

The above example demonstrates that the verifier checks

-
-
-

And that the verifier does not check

-
-

.

Example 3 Compile

```
class A  { int f;  }
class C{  void m (A a){ a.f = 23; }  }
class Test{
    public static void main(String[] args){
        System.out.println(1);  new C ();
        System.out.println(2);  }
}
```

Now modify `A`, as

```
class A  {    }
```

and compile `A` but not the rest.

Partial output - verify. on:	Partial output – verify. off:

`[Loaded Test]`
`1`
`[Loaded c]`
`2`
	`[Loaded Test]`
	`1`
	`[Loaded c]`
	`2`

The above example demonstrates

What will happen if I attempt to access the non existing field? In other words, compile

```
class A{ int f;  }
class C{  void m (A a){
          System.out.println("   m:1");
          a.f = 23;
          System.out.println("   m:2"); }  }
class Test{
    public static void main(String[] args){
        System.out.println(1);
        C c = new C();
        System.out.println(2);
        A a= new A();
        System.out.println(3);
        c.m(a);
        System.out.println(4); }    }
```

Now modify A, as

```
class A  {    }
```

and compile `A` but not the rest.

Partial output - verify. on:	Partial output – verify. off:
[Loaded Test] 1 [Loaded C] 2 [Loaded A] 3 m:1 java.lang.NoSuchFieldError	[Loaded Test] 1 [Loaded C] 2 [Loaded A] 3 m:1 java.lang.NoSuchFieldError

The above example demonstrates

Resolution and its manifestation

Bytecode may contain symbolic references, e.g:

```
getfield  <int fa> from A
putfield  <int fb> from B
invokevirtual  <int m(B,A)> from A .
```

For fields/methods these contain name of field or method, type information and name of type where declaration occurred.

Resolution checks that the reference is correct and may replace it with a direct reference. It fails, 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

Example 4 Compile following classes

```
class A{
  int i = 10;  int j = 20;   int k = 30; }

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

change class A

```
class A{
  int i = 10;  char j = 'c';    int k = 30;  }
```

Do not recompile Test. Partial output from executing Test:

```
[Loaded Test]
[Loaded A]
 --- 1
… java.lang.NoSuchFieldError: j
```

The above example demonstrates .

Bypassing verification

Example 5 Compile following classes

```
class A { int i = 33; }

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() ); }
}
```

Change class `B`:

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

Recompile `B` but not the rest.

Partial output from executing `Test` with verification on:

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

The above example demonstrates .

Summary and Notes

- Intricate interplay between responsibilities of the phases.
- Even more interesting when several class loaders involved (class A may be loaded differently by different loaders)
- C# is similar but slightly different.
- Drossopoulou, Lagorio, Eisenbach ESOP'03: "Flexible Models for Dynamic Linking" nondet. model incl. Java and C#
- **`java -verbose xxx`** (verbose execution, verifier on)
- **`java -Xverify:none xx`** (verbose execution, verifier off)
- Different versions Java compiler slightly different behaviour wrt. overloading/shadowing resolution