J2ME Compression

class files

class file format

constant pool
access flags
class (constant pool index)
superclass (constant pool index)
interfaces (constant pool indices)
fields
methods

constant pool

- table of indexed entries
- eleven types (UTF8 strings, integer, float, class ref, method ref, field ref, etc.)

constant pool entries in a class using String.lastIndexOf(String, int)

```
100     UTF8          "java/lang/String"
101     UTF8          "lastIndexOf"
102     UTF8          "(Ljava/lang/String;I)I"
103     Class         100
104     NameAndType   [ 101, 102 ]
105     MethodRef     [ 103, 104 ]

invokevirtual 105
```

the constant pool is

- verbose (bulk of a class file)
- repeated across class files
- full of class library API references that can't be shorted by obfuscation

strategy

basic idea

We want to collapse all of the application class files into one,
which collapses all of the constant pools into one,
which reduces the number of entries and shrinks the overall application size.

tools

Lots of open-source projects for manipulating class files: BCEL, Clazzer, Cream, Bloat, Soot, etc, etc etc.

Some are interesting (especially Soot) but most are too low-level; manual constant pool fixup is a nightmare.

. . . 2 weeks later . . .

Jasmin

- originally a teaching tool in Jon Meyer's book *The Java Virtual Machine* (1997)
- defacto standard assembler format
- existing Jasmin library converts Jasmin to class files, still works fine
- Soot has an updated class file to Jasmin translator

Jasmin format

Java

```
class Foo
{
  int foo;

  void incFoo()
  {
    ++foo;
  }
}
```

```
class Bar
{
  void static bar(Foo f)
  {
    f.incFoo();
  }
}
```

Jasmin

```
.class Foo
.super java/lang/Object
.field "foo" I
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method
.method incFoo()V
  aload_0
  aload_0
  getfield Foo/foo I
  iconst_1
  iadd
  putfield Foo/foo I
  return
.end method
```

```
.class Bar
.super java/lang/Object
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method
.method static bar(LFoo;)V
  aload_1
  invokevirtual Foo/incFoo()V
  return
.end method
```

a simple transformation

Move a method from one class to another.

```
.class Foo
.super java/lang/Object
.field "foo" I
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method
.method incFoo()V
  aload_0
  aload_0
  getfield Foo/foo I
  iconst_1
  iadd
  putfield Foo/foo I
  return
.end method

.class Bar
.super java/lang/Object
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method
.method static bar(LFoo;)V
  aload_1
  invokevirtual Foo/incFoo()V
  return
.end method
```

```
.class Foo
.super java/lang/Object
.field "foo" I
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method

.class Bar
.super java/lang/Object
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V;
.end method
.method static bar(LFoo;)V
  aload_1
  invokevirtual Bar/incFoo()V
  return
.end method
.method incFoo()V
  aload_0
  aload_0
  getfield Foo/foo I
  iconst_1
  iadd
  putfield Foo/foo I
  return
.end method
```

No changes in the method body, regexp replacement for method invocations, no constant pool fixup.

phases

1. Convert class files to Jasmin files and load them into an in-memory representation.
2. Perform transformations that will allow fields and methods to be moved into a single class.
3. Perform transformations to move fields and methods into a single class.
4. Perform other optimizations (removal of unused fields and methods, etc.)
5. Save the in-memory representation as Jasmin files, convert them back to class files.

phase 2

Perform transformations that will allow fields and methods to be moved into a single class.

createBaseClass();

```
class VindigoBase
{
}
```

insertBaseSuperclassInImmediateSubclasses();

Modify any classes that directly extend Object to extend VindigoBase instead.

before:

```
class Foo {}
class Bar extends Foo {}
class Baz extends MIDlet {}
```

after:

```
class Foo extends VindigoBase {}
class Bar extends Foo {}
class Baz extends MIDlet {}
```

Any constructors that call the default Object constructor are rewritten to call the VindigoBase constructor instead.

```
invokespecial java/lang/Object/<init>()V
```

becomes:

```
invokespecial VindigoBase/<init>()V
```

invokespecial issues

Most instance methods are invoked using the invokevirtual instruction:

```
invokevirtual Foo/bar(Ljava/lang/String;)V
```

Method resolution is dynamic; begins searching for a method of the appropriate descriptor within the object's runtime class, (which may be a subclass of Foo), and continues searching through superclasses until it finds a method with the matching descriptor.

The invokespecial instruction appears in only three contexts:

- calling constructors
- calling superclass methods via super.method()
- calling private instance methods

Constructors are easily recognized because the method descriptor is always named <init>.

For the other two contexts, invokespecial begins method resolution in the class or superclass named in the instruction, rather than in the runtime class of the object. Given superclass Foo and subclass Bar, both of which define method foo():

```
invokespecial Foo/foo()V
```

will invoke the method Foo.foo(), even on objects of type Bar. This is the mechanism 'super' uses.

Java allows private methods to be declared with the same descriptor as public and protected methods in superclasses and subclasses. Invokespecial is used to ensure that the private method is called.

Given a private method foo() in class Foo,

```
invokespecial Foo/foo()V;
```

ensures that the method in class Foo is called, even if the object's runtime class is a subclass of Foo that defines a public method with the same descriptor.

changePrivateInvokespecialToInvokevirtualInAllClasses();

We want to be able to move private methods into the base class. We rename private methods so that their names are guaranteed to be unique, and rewrite all invokespecial instructions to invokevirtual instead.

To generate unique names, prepend method names with the "private$" followed by the fully qualified class name.

```
class Foo
{
  public void foo()              { . . . }
  private int bar()              { . . . }
}
```

becomes:

```
class Foo
{
  public void foo()              { . . . }
  private int private$Foo$bar()  { . . . }
}
```

Any invokespecial instructions are then rewritten from:

```
invokespecial Foo/bar()I;
```

to:

```
invokevirtual Foo/private$Foo$bar()I;
```

Embedding the class name in the method name guarantees globally unique names.

moveBodiesOfInvokespecialMethodsIntoNewMethodsInAllClasses();

Once we have rewritten private invokespecial instructions as invokevirtual, the only remaining uses of invokespecial are for constructors and super calls. The latter present a problem if we want to move methods out of subclasses and into the base class.

To get rid of the invokespecial instruction, we move the body of the method called via invokespecial to a new method, and rewrite the other methods to invoke that:

```
class Foo
{
  void foo() {  . . . }
}


class Bar extends Foo
{
  void foo()
  {
    . . .
    super.foo(); // invokespecial
  }
}
```

```
class Foo
{
  void foo() { special$Foo$foo(); }
  void special$Foo$foo() { . . . }
}

class Bar extends Foo
{
  void foo()
  {
    . . .
    special$Foo$foo(); // invokevirtual
  }
}
```

Now we can use invokevirtual instead of invokespecial, because the invoked method is guaranteed to be globally unique.

Note that at this point, the only remaining invokespecial instructions are for constructors, which will be unaffected by moving the enclosing method.

moveBodiesOfVirtualInstanceMethodsIntoNewMethodsInAllClasses();

Next we deal with methods where the same descriptor exists in superclasses, subclasses, and interfaces. These are problematic if we want to move them to the base class. We need to preserve dynamic method resolution, so they can't simply be renamed.

```
void bar(Foo f) { f.foo(); }
```

If foo() is defined in class Foo and overridden in subclasses of Foo, we can't know which method will actually be called via invokevirtual here. We work around this problem by moving the body of the method into a new method with a unique name.

```
class Foo                { void foo() { ... use a Foo ... } }
class Bar extends Foo    { void foo() { ... use a Bar ... } }
```

becomes:

```
class Foo
{
  void foo()             { virtual$Foo$foo(); }
  void virtual$Foo$foo() { ... use a Foo ... }
}

class Bar extends Foo
{
  void foo()             { virtual$Bar$foo(); }
  void virtual$Bar$foo() { ... use a Bar ... }
}
}
```

The new methods can later be moved into the base class, since their names are globally unique. The virtual method, which has to stay in the subclass, is reduced to a single call to the new method.

moveVirtualMethodDispatchFromSubclasssesToBaseClass();

The previous transformations didn't move methods used virtually, because it would break dynamic method resolution. For subclasses of VindigoBase, we can replace methods in subclasses with a single method in VindigoBase that dispatches based on the runtime type of the object.

Given these classes:

```
class Foo             { void foo() { ... use a Foo ... } }
class Bar extends Foo { void foo() { ... use a Bar ... } }
class Baz extends Bar { void foo() { ... use a Baz ... } }
```

We move the method bodies into uniquely named (non-virtual) methods, while at the same time building up a dispatch method in VindigoBase. The result looks like this:

```
class Foo extends VindigoBase { void virtual$Foo$foo() { ... use a Foo ... } }
class Bar extends Foo         { void virtual$Bar$foo() { ... use a Bar ... } }
class Baz extends Bar         { void virtual$Baz$foo() { ... use a Baz ... } }

class VindigoBase
{
 void foo()
 {
   if (this instanceof Baz)
     virtual$Baz$foo();
   else if (this instanceof Bar)
     virtual$Bar$foo();
   else if (this instanceof Foo)
     virtual$Foo$foo();
 }
}
```

Note that we have to ensure correct order in the dispatcher; subclasses tested before superclasses.

The new methods in the subclasses are no longer used virtually, and can later be moved to the base class.

makeAllClassesPublicAndNonFinal();

The final step before we begin moving methods and fields into the base class is to make all classes, methods and fields public and non-final. This ensures that we can move them without causing access-level problems.

phase 3

Perform transformations to move fields and methods into a single class.

moveStaticFieldsFromAllClassesToBaseClass();

We move all static fields into the base class, and rewrite access instructions accordingly.

```
class Foo
{
  static int foo;
  void setFoo(int x)   { foo = x; }
}

class Bar extends MIDlet
{
  static int bar;
  void setBarFromFoo() { bar = Foo.foo; }
}
```

becomes:

```
class VindigoBase
{
  static int Foo$foo;
  static int Bar$bar;
}

class Foo extends VindigoBase
{
  void setFoo(int x)   { VindigoBase.Foo$foo = x; }
}

class Bar extends MIDlet
{
  void setBarFromFoo() { VindigoBase.Bar$bar = VindigoBase.Foo$foo; }
}
```

moveStaticInitializersFromSubclassesToBaseClass();

We'd like to move static initializers to the base class. In the general case, this isn't a safe change, but we can move static initializers for subclasses of VindigoBase, since it's loaded before any of its subclasses.

We move all static initializers from subclasses to the base class, rename them, and create a static intializer in the base class that calls them.

```
class Foo             { static String s = "abc"; }
class Bar extends Foo { static int n = 3; }
```

In (pseudo-) source, the generated bytecode is equivalent to:

```
class Foo                                       |  class Bar extends Foo
{                                               |  {
  static String s;                              |    static int n;
  static void <clinit>() { Foo.s = "abc"; }     |    static void <clinit>() { Bar.n = 3; }
}                                               |  }
```

Our transformed code looks like this:

```
class VindigoBase
{
  static string Foo$s;
  static int    Bar$n;

  static void Foo$clinit$() { VindigoBase.Foo$s = "abc"; }
  static void Bar$clinit$() { VindigoBase.Bar$n = 3; }

  static void <clinit>()
  {
    Foo$clinit$();
    Bar$clinit$();
  }
}
```

```
class Foo extends VindigoBase {}
class Bar extends Foo {}
```

Note that the order of calls in <clinit>() is significant; we have to call the initializer for a superclass before the initializer for any of its subclasses.

moveStaticMethodsFromAllClassesToBaseClass();

We move all static methods (excluding remaining static initializers) from all classes to the base class. This is a simple transformation, requiring only that we change the method names to ensure uniqueness.

```
class Foo
{
  static void foo()     { ... do something ... }
}

class Bar extends MIDlet
{
  static void bar()     { Foo.foo(); }
}
```

becomes:

```
class VindigoBase
{
  static void Foo$foo() { ... do something ... }
  static void Bar$bar() { VindigoBase.Foo$foo(); }
}

class Foo extends VindigoBase {}
class Bar extends MIDlet {}
```

moveInstanceMethodsFromAllClassesToBaseClass();

We want to move instance methods from all classes to the base class. We make the methods static and insert an explicit first parameter of the correct type.

```
class Bar
{
  Foo f;
  void bar(int x)                    { f.foo(x); }
}
```

becomes:

```
class VindigoBase
{
  static void Bar$bar(Bar self, int x) { VindigoBase.Foo$foo(self.f, x); }
}

class Bar {}
```

This transformation is *much* simpler on bytecode than it appears above.

For an instance method, local variable 0 always contains the implicit 'this' passed to the method. For a static method, local variable 0 always contains the first argument passed to the method (since there is no implicit this.) A typical getter method example:

before

```
.method getFoo()I
  aload_0
  getfield Foo/foo I
  ireturn
.end method
```

after

```
.method static Foo$getFoo(LFoo;)I
  aload_0
  getfield Foo/foo I
  ireturn
.end method
```

Note that only the method name and argument list changes; the method body remains the same because local variable 0 remains the same. At this point, nearly all methods in VindigoBase are static.

phase 4

Perform other optimizations (removal of unused fields and methods, etc.)

inlineSimpleBaseClassStaticMethodsInAllClasses();

We remove method calls that can be replaced with a single instruction. We look for any method used transitively (simply forwards its arguments in the same order to another method):

```
class Foo
{
  static int foo(int a, String b, char c) { . . . do something . . . }
  static int bar(int a, String b, char c) { return foo(a, b, c); }
  static int baz()                        { return bar(1, "2", '3'); }
}
```

In the example above, any calls to bar() can be replaced inline with calls to foo(). In bytecode, this technique can be applied to any method that only load its arguments on the operand stack, executes one of the following instructions, and executes a return instruction:

```
  getfield, putfield, invokevirtual, invokeinterface, invokestatic
```

Simple getters and setters follow this pattern. Given this code:

```
  class Foo
  {
    int foo;

    int getFoo()        { return foo; }
    void setFoo(int f)  { foo = f; }
  }

  class Bar
  {
    void incFoo(Foo f)  { f.setFoo(f.getFoo()+1); }
  }
```

inlineSimpleBaseClassStaticMethodsInAllClasses(); cont'd

After the previous transformations, the code will look like this:

```
class VindigoBase
{
  static int  Foo$getFoo(Foo self)        { return self.foo; }
  static void Foo$setFoo(Foo self, int f) { self.foo = f; }

  static void Bar$incFoo(Bar self, Foo f)
  { VindigoBase.Foo$setFoo(f, VindigoBase.Foo$getFoo(f) + 1); }
}
```

The corresponding Jasmin code:

```
.method static Foo$getFoo(LFoo;)I
  aload_0
  getfield Foo/foo I
  ireturn
.end method

.method static Foo$setFoo(LFoo;I)V
  aload_0
  iload_1
  putfield Foo/foo I
  return
.end method

.method static Bar$incFoo(LBar;LFoo;)V
  aload_1
  aload_1
  invokestatic VindigoBase/Foo$getFoo(LFoo;)I
  iconst_1
  iadd
  invokestatic VindigoBase/Foo$setFoo(LFoo;I)V
  return
.end method
```

becomes:

```
.method static Bar$incFoo(LBar;LFoo;)V
  aload_1
  aload_1
  getfield Foo/foo I
  iconst_1
  iadd
  putfield Foo/foo I
  return
.end method
```

removeUnusedStaticMethodsFromBaseClass();

Search all methods in all classes for:

```
invokestatic VindigoBase/[method-descriptor]
```

If not found, we can safely remove the method.

Note that the current implementation doesn't build a call graph, so it will not detect cycles of unreachable methods, or methods made eligible for removal by the removal of other methods.

removeUnusedStaticFieldsFromBaseClass();

Search all methods in all classes for:

```
getstatic VindigoBase/[field-descriptor]
```

or

```
putstatic VindigoBase/[field-descriptor]
```

If not found, we can safely remove the field. We perform this transformation after removing unused static methods, to ensure that we don't keep fields only referenced in methods that aren't used.

Note that the current implementation doesn't remove fields that are only put.

removeUnusedInstanceFieldsFromAllClasses();

Search all methods in all classes for:

```
getfield [class]/[field-descriptor]
putfield [class]/[field-descriptor]
```

If not found, we can safely remove the field. We perform this transformation after removing unused static methods, to ensure that we don't keep fields only referenced in methods that aren't used.

Note that the current implementation doesn't remove fields that are only put.

removeUnusedClasses();

Search for the class name as an implemented interface, superclass, argument type, or field type. If not found, we can safely remove the class.

Note that the current implementation doesn't find unused cycles of classes.

splitBaseClass();

After all transformations have been applied, the class VindigoBase is large. In some VM environments, the inability to allocate a large enough contiguous block of memory this may prevent it from being loaded. The only solution is to split the class.

Since most of our space saving comes from reducing multiple class constant pools into one, we want to minimize the increase caused by splitting. The current implementation keeps any method that references a javax.* system class in VindigoBase, and split the remaining methods arbitrarily among n extra classes. This avoids duplicating long javax.* constant pool strings (which can't be renamed by the obfuscator) in constant pools.

PostObfuscator

The class PostObfuscator is used to do two post-processing tasks after obfuscation.

regenerate class files from Jasmin files

First, we convert all class files to Jasmin files and back to class files. If the resulting class file is smaller, we use it instead of the original obfuscated class file.

sort constant pool

Second, we sort the constant pool, putting all string and numeric constants first, and then sorting all UTF8 strings lexically. This has two benefits:
- the first 256 constant pool entries can be loaded using an ldc instruction, which takes a one-byte index, rather than an ldc_w instruction, which takes a two byte index
- sorting UTF8 strings makes the LZW compression algorithm used in jar files more efficient

results

VindigoBase now contains

- all static fields from all classes
- all static methods from all classes
- all static initializers from its subclasses
- nearly all instance methods from all classes (some virtual method stubs remain)

other classes contain

- instance variables
- constructors (delegating all work to a method moved into VindigoBase)
- some virtual methods (delegating all work to a method moved into VindigoBase)
- static initializers (in non-subclasses only)

SUDS client numbers

- Vindigo jar:	88900 bytes	41 classes	474 methods
- +obfuscated:	76441 bytes	(same)	(same)
- +compressed:	61025 bytes	39 classes	500 methods (451 VindigoBase, 35 constructors, 14 other)