Chapter 13 A Brief Comparison with Java

Some Java features

Java has many features that my simple B-- language does not have.

For example:

- Java has automatic garbage collection, which makes it a much better language than C or B--. There is a price paid in terms of efficiency, but it is well worth paying given the gains in programmer productivity.
- Java supports the notion of packages, for grouping classes, access control (public, protected, private, package access), and overloading. These features make the lookup of the compile-time environment more complex. However, they generate no additional run-time costs.
- Java has a notion of a base class (Object), that all classes extend. It also has the notion of the class Class, used to represent information about class types, and help in run-time actions related to casts and the instanceof operator.
- Java has static fields and methods. Static fields are not a part of an object, and static methods do not occur in the method table. A reasonable implementation might make them part of the instance of the class Class corresponding to a class type.
- The Object class has a toString() method, that is used to convert an object to a String (for example when performing concatenation or printing), and a getClass() method to let the programmer determine the run-time class of the object. There are many other methods in the Object class, and many classes in the API have hundreds of methods, so method tables can be quite large.
- Java has interface types. Find out what the equivalent is in C++, and you will see why we don't teach C++ in elementary courses.
- Pointers are implicit in Java. All array, class and interface variables are actually pointers to objects. We need to explicitly allocate space for each object using a constructor. The size of an array is not part of the type of the array, but is specified as a parameter to the constructor.
- There is no "&" (address of) operator in Java, so it is not possible to point to variables. There is no equivalent of var parameters. However, we can pass the address of an object or array to a method, and modify the components of the object or array.
- Java has constructors to initialise a class. A constructor for a class always invokes the constructor for the superclass, before performing its own initialisation. The superclass constructor can be indicated explicitly, but if there is no explicit indication, the default constructor with no parameters is invoked.
- At run time, Java initialises classes in a lazy fashion. Static variables are only initialised when the class is first referred to. In fact, even the loading of a class can be lazy.

Overloading

In modern languages such as Java, the requirement for an identifier to be used for at most one declaration in a block is relaxed. We can **overload** a name (have more than one declaration with the same identifier in the same block). However, there is still a requirement that no two declarations in the same block have exactly the same signature. There is no point in allowing multiple declarations with the same signature, because one of them would hide the others.

To cope with overloading, when we search an environment, we specify a signature the declaration must satisfy. The signature depends on the construct we are processing.

- It is usually possible to tell from the syntactic context whether the identifier should correspond to a type, variable, or method, so we can permit overloading of identifiers in different categories. However, there is an ambiguity for member selection, since the left operand can be either a class name (for a static member), or variable name (for an instance member). Java prefers variable names over class names, but this is a design flaw in the language. Overloading between class and variable names should not be permitted.
- Method names can be overloaded. When determining the meaning of an identifier, we take into account the number and types of the actual parameters.

  An invocation matches a declaration if the number of formal and actual parameters agree, and each actual parameter can be "widened" to the type of the formal parameter. "Widening" means performing a type conversion of a value from a subtype to a supertype, or from a more limited primitive type to a more encompassing primitive type (e.g., char → int, int → double).

Out of all the matches, we choose the match that involves the fewest implicit widenings of actual parameters to formal parameters.

Sometimes there is no unique closest match. We then generate an error message.

For example, there could be two applicable declarations

```
double plus( int, double );
double plus( double, int );
```

Both match plus(3, 4), but neither is a closer match than the other.

An important point to note is that **the matching of an invocation to a signature is done at compile time, and depends only on the declared type of variables**, not the runtime values. In general, the compiler cannot determine the type of a run-time value. Code needs to be generated for a class without any knowledge of whether a programmer might later extend the class to form a subclass.

Overloading is a simple convenience to allow the programmer to declare multiple methods with the same name but different signature. It has no relationship with overriding of methods. It adds complexity to name lookup in the compile-time environment, but makes no difference to what happens at run time.

For example, suppose we have a method

```
class A {
     int f( double x ) { … }
     }
class B extends A {
     int f( int x ) { … }
     }
A a = new B();
a.f( 3 );
```

Then the invocation

```
a.f( 3 );
```

matches the declaration

```
int f( double x ) { … }
```

in the declared type A of "a", not the declaration in B, in the actual type of "a".

Access control

When searching an environment, many languages also take into account access control (whether the declaration has public, protected, private or package access). The search method has an additional parameter corresponding to a set of acceptable access rights.

In Java

- When attempting to determine the meaning of a simple identifier, we start by searching the environment with all access flags set.
- When attempting to determine the meaning of an identifier when processing member selection, we first determine the class for the object being selected. We always permit public access. If this object is in the same package, we permit package access. If the object is in a superclass, we permit protected access. If the object is of the same type as the current class, we permit private access.
- When searching an environment, we only match declarations for which the access is permitted.
- When searching the enclosing environment, we search with the same access flags set.
- When searching the environment of the superclass, we clear the flag for private access. We also clear the flag for package access if the superclass is in a different package.

The class Class

The class Object has a getClass() method, that returns an instance of the class Class, representing the run-time type of the object. The Object class (and hence every object) probably has a private field, that refers to the appropriate instance of the class Class. Given that we have a reference to the instance of the class Class, we could eliminate the reference to the method table, because the instance of the class Class probably contains the method table. However, this would increase the level of indirection involved in access to the method table.

The class Class representing a class type probably contains fields for

- The name of the class type.
- A reference to the static field table and method table for the class type.
- A reference to the instance method table for the class type.
- The code for the class type, including the code for static and instance initialisation, and for constructors and methods.
- A reference to the instance of the class Class for the superclass of the class type. This information is used to implement casts involving narrowing, and the instanceof operator.
- References to the instances of the class Class corresponding to the classes referred to by the class type. The code for this class accesses the other classes through these references. It is discovers a null reference, it can make a request to the class loader to load the class referred to, and initialise it.
- Information about the types and names of all fields, methods, etc, declared in the class type. This information is used for “reflection”.

Order of initialisation

We can find out a lot about the implementation of a language by doing little experiments.

The program

```
class A {
     static int classInitA() {
          System.out.println( "Init A static fields" );
          return 0;
          }
     static int y = classInitA();
     int initA() {
          System.out.println( "Init A instance fields" );
          return 0;
          }
     int x = initA();
     A() {
          System.out.println( "Init A default constructor" );
          }
     }

class B extends A {
     static int classInitB() {
          System.out.println( "Init B static fields" );
          return 0;
          }
     static int y = classInitB();
     int initB() {
          System.out.println( "Init B instance fields" );
          return 0;
          }
     int x = initB();
     B() {
          System.out.println( "Init B default constructor" );
          }
     B( int x ) {
          System.out.println( "Init invoked B constructor" );
          }
     }

class Main {
     static int classInitMain() {
          System.out.println( "Init Main static fields" );
          return 0;
          }
     static int y = classInitMain();
     public static void main( String[] arg ) {
          System.out.println( "Invoke main" );
          B b = new B( 3 );
          }
     }
```

generates the output

```
Init Main static fields
Invoke main
Init A static fields
Init B static fields
Init A instance fields
Init A default constructor
Init B instance fields
Init invoked B constructor
```

So we can see that

- Before a field, method, or constructor of a class can be accessed, we initialise the static fields of its superclass(es), then the static fields of the class itself.

- To execute a constructor, we first execute the appropriate constructor(s) for the superclass(es), then initialise the instance fields for the class, then execute the body of the constructor.

Type checking is a compile time action, based on the declared types of variables

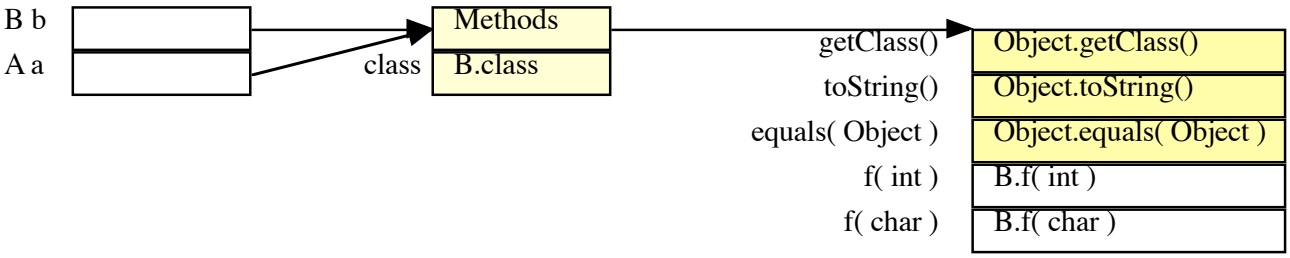
The program

```
class A  {
     String f( int x ) {
          return "A.f( int " + x + " )";
          }
     }

class B extends A  {
     String f( char x ) {
          return "B.f( char '" + x + "' )";
          }
     String f( int x ) {
          return "B.f( int " + x + " )";
          }
     }

class Main {
     public static void main( String[] arg ) {
          B b = new B();
          A a = b;
          System.out.println( "a.f( 'c' ) = " + a.f( 'c' ) );
          System.out.println( "b.f( 'c' ) = " + b.f( 'c' ) );
          }
     }
```

generates the output

```
a.f( 'c' ) = B.f( int 99 )
b.f( 'c' ) = B.f( char 'c' )
```

So you can see that because the declared type of “a” is “A”, when type checking, the Java compiler looks in the class “A” to match “f(char)”, and gets a best match of “f(int)”. At run time, the offset of “f(int)” is used to look up the method table. But the method table used is that of the actual type, “B”. Now “f(int)” happens to be overridden in “B”. So it finds “B.f(int)”.



Methods are overridden, fields are not

The program

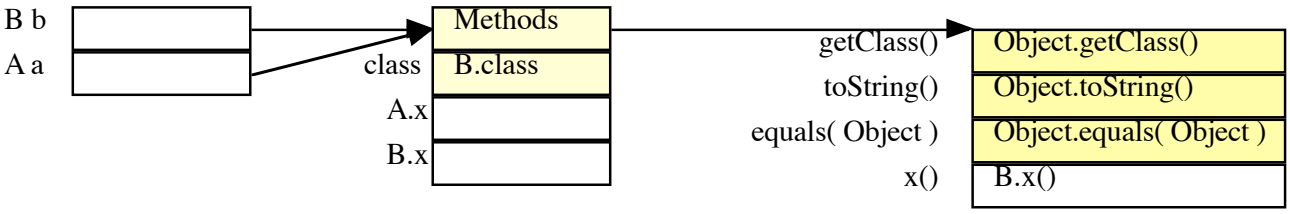
```
class A  {
     String x = "A.x";
     String x() {
          return x;
          }
     }

class B extends A  {
     String x = "B.x";
```

```
     String x() {
          return x;
          }
     }

class Main {
     public static void main( String[] arg ) {
          B b = new B();
          A a = b;
          System.out.println( "a.x = " + a.x );
          System.out.println( "b.x = " + b.x );
          System.out.println( "a.x() = " + a.x() );
          System.out.println( "b.x() = " + b.x() );
          }
     }
```

generates the output

```
a.x = A.x
b.x = B.x
a.x() = B.x
b.x() = B.x
```

There are actually two fields called “x”, one from “A”, and one from “B”. However, there is only one method “x()”. It has the offset of “A.x()” in the method table, but it is the overridden version “B.x()”. Now “B.x()” was compiled as a part of “B”, and so the reference to “x” inside “B.x()” was mapped to the offset of “B.x”.



If we delete the declaration of “x()” in “B”, the program

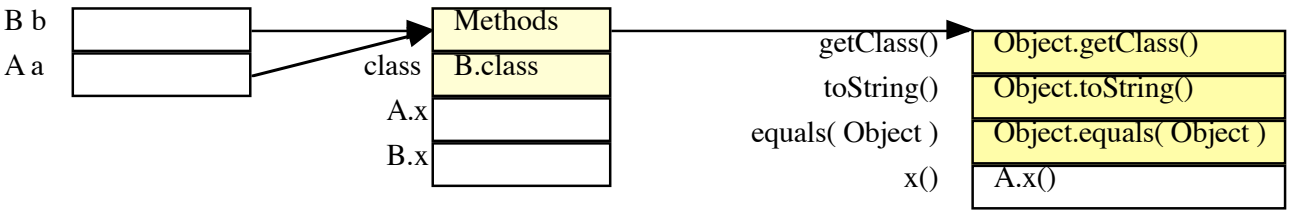
```
class A  {
     String x = "A.x";
     String x() {
          return x;
          }
     }

class B extends A  {
     String x = "B.x";
     }

class Main {
     public static void main( String[] arg ) {
          B b = new B();
          A a = b;
          System.out.println( "a.x = " + a.x );
          System.out.println( "b.x = " + b.x );
          System.out.println( "a.x() = " + a.x() );
          System.out.println( "b.x() = " + b.x() );
          }
     }
```

generates the output

```
a.x = A.x
b.x = B.x
a.x() = A.x
```

```
b.x() = A.x
```

"A.x()" was compiled as a part of "A", and so the reference to "x" inside "A.x()" was mapped to the offset of "A.x".



Private methods are not overridden

If we really look into things, we discover many subtleties. For example, private methods cannot be overridden.

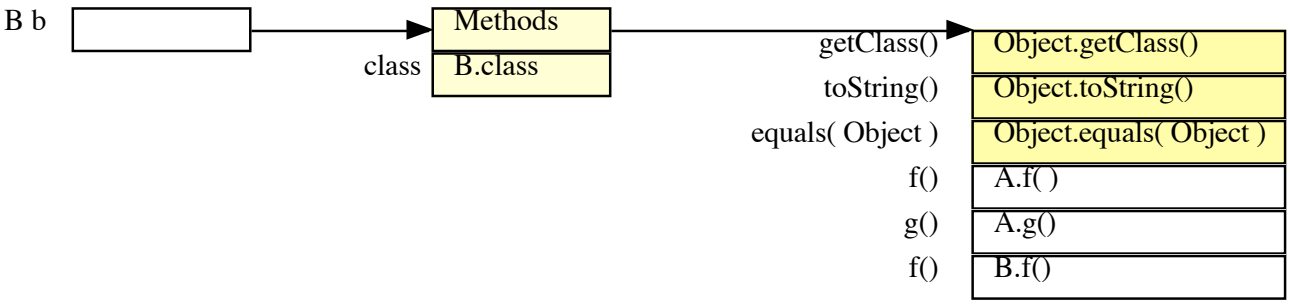
The program

```
class A {
     private String f() {
          return "A.f()";
          }
     public String g() {
          return f();
          }
     }

class B extends A {
     public String f() {
          return "B.f()";
          }
     }

class Main {
     public static void main( String[] args ) {
          B b = new B();
          System.out.println( b.f() );
          System.out.println( b.g() );
          }
     }
```

generates the output

```
B.f()
A.f()
```

Why is this? Well, when the method f() is searched for in the extended environment of B, it is not found, because it is private to the class A, so f() is regarded as not overriding the method f() in B. So the two versions of f() are given different offsets in the method table. When g() executes, it uses the offset of the private method in A. It is not that there is special code in the compiler to check this. It is a simple consequence of the algorithm for checking whether methods are being overridden. The important point is that it shows that having a good model for how the compiler works lets us predict what will happen in unusual situations.



A more complex example

The program

```
class A {

     public static int q = 100, r = 200;

     public int p = 888;

     public A( int p ) { this.p = p; q++; }
     public A() { r++; }

     public String toString() { return "A.toString(): p = " + p; }
     public String e( char c ) { return "A.e( '" + c + "' )"; }
     public String e( double x ) { return "A.e( " + x + " )"; }
     }

class B extends A {

     public static int q = 300;

     public int p = 999;

     public B( int p ) { this.p = p; q++; }

     public String toString() { return "B.toString(): p = " + p; }
     public String e( int i ) { return "B.e( " + i + " )"; }
     public String e( double x ) { return "B.e( " + x + " )"; }
     }
```

```
class Main {
     public static void main( String[] args ) {

          A a1 = new A();
          A a2 = new A( 1000 );
          B b1 = new B( 2000 );
          A b2 = b1;

          System.out.println( "A.q = " + A.q );
          System.out.println( "A.r = " + A.r );
          System.out.println( "B.q = " + B.q );
          System.out.println();

          System.out.println( "a1.p = " + a1.p );
          System.out.println( "a2.p = " + a2.p );
          System.out.println( "b1.p = " + b1.p );
          System.out.println( "b2.p = " + b2.p );
          System.out.println();

          System.out.println( "a1 = " + a1 );
          System.out.println( "a2 = " + a2 );
          System.out.println( "b1 = " + b1 );
          System.out.println( "b2 = " + b2 );
          System.out.println();

          // 'A' is ASCII 65
          System.out.println( "a1.e( 'A' ) = " + a1.e( 'A' ) );
          System.out.println( "b1.e( 'A' ) = " + b1.e( 'A' ) );
          System.out.println( "b2.e( 'A' ) = " + b2.e( 'A' ) );
          System.out.println();

          System.out.println( "a1.e( 65 ) = " + a1.e( 65 ) );
          System.out.println( "b1.e( 65 ) = " + b1.e( 65 ) );
          System.out.println( "b2.e( 65 ) = " + b2.e( 65 ) );
          System.out.println();

          System.out.println( "a1.e( 65.0 ) = " + a1.e( 65.0 ) );
          System.out.println( "b1.e( 65.0 ) = " + b1.e( 65.0 ) );
          System.out.println( "b2.e( 65.0 ) = " + b2.e( 65.0 ) );
          System.out.println();

          }
     }
```

generates the output

```
A.q = 101
A.r = 202
B.q = 301

a1.p = 888
a2.p = 1000
b1.p = 2000
b2.p = 888

a1 = A.toString(): p = 888
a2 = A.toString(): p = 1000
b1 = B.toString(): p = 2000
b2 = B.toString(): p = 2000

a1.e( 'A' ) = A.e( 'A' )
b1.e( 'A' ) = A.e( 'A' )
b2.e( 'A' ) = A.e( 'A' )

a1.e( 65 ) = A.e( 65.0 )
b1.e( 65 ) = B.e( 65 )
b2.e( 65 ) = B.e( 65.0 )

a1.e( 65.0 ) = A.e( 65.0 )
b1.e( 65.0 ) = B.e( 65.0 )
b2.e( 65.0 ) = B.e( 65.0 )
```

The two instances of A share the method table for A.

There is only one instance of B, one pointed to by a variable of type B, one of type A.

There is one copy of the static variables for each, and the method tables do not contain static methods.

The value of A.r is incremented twice, due to the explicit and implicit invocations of new A(). A.q is incremented in the invocation of new A(int). B.q is incremented in the invocation of new B(int).

The object of run-time type B has two fields named p, but the one accessed depends on the declared type of the variable.

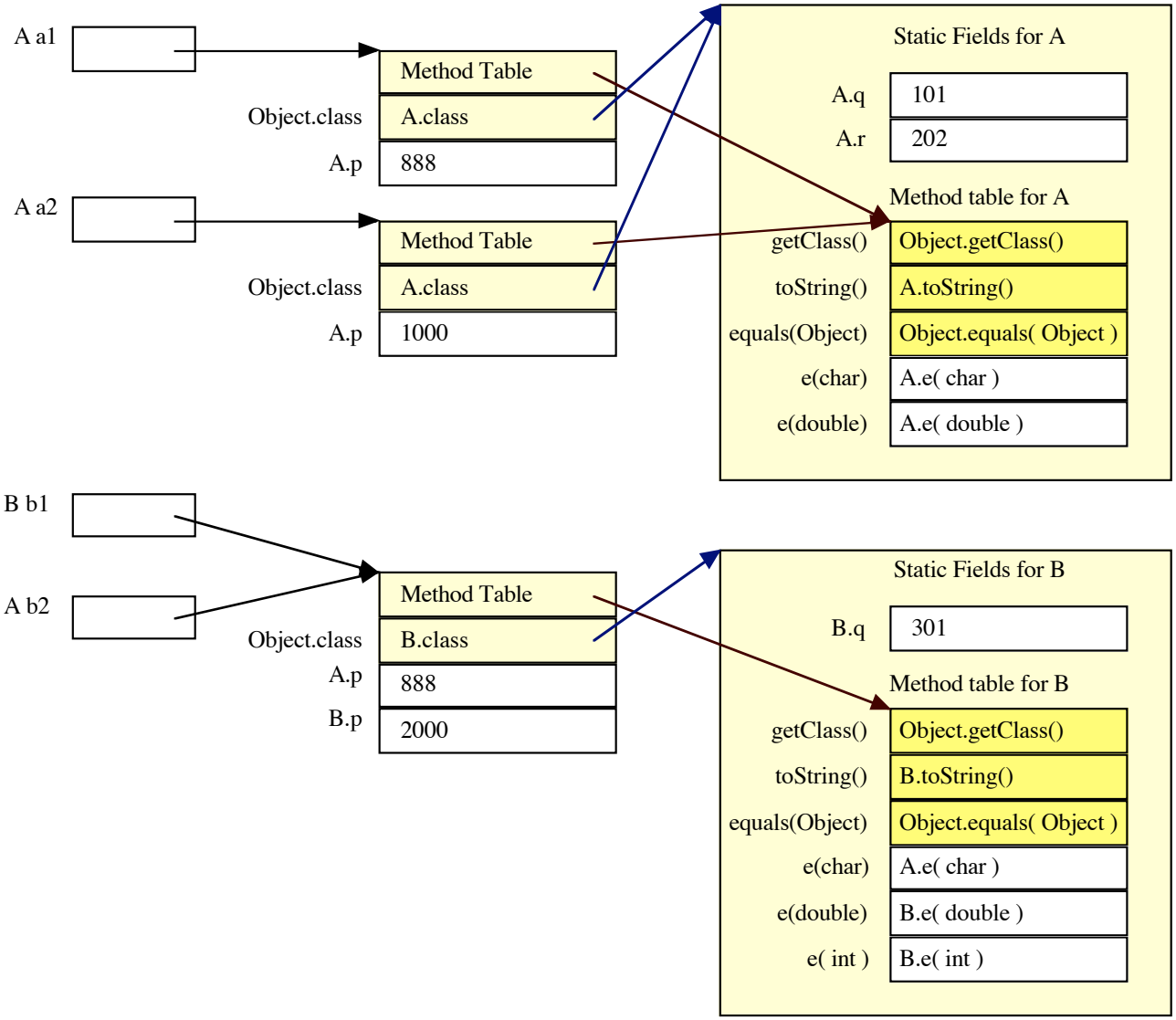
The getClass() method is implicitly overridden for each class.

The toString() method is overridden in both A and B, which affects statements such as

```
System.out.println( "a1 = " + a1 );
```

But the field p accessed by toString() depends on which class toString() is declared in.

An invocation of e(int) when the object is of declared type A invokes e(double), because there is no e(int) declared in A. But e(double) is overridden in B, so b2.e(int) ends up invoking B.e(double).


A a1
Method Table
Object.class
A.class
A.p
888
A a2
Method Table
Object.class
A.class
A.p
1000
Static Fields for A
A.q
101
A.r
202
Method table for A
getClass()
Object.getClass()
toString()
A.toString()
equals(Object)
Object.equals(Object)
e(char)
A.e(char)
e(double)
A.e(double)
B b1
A b2
Method Table
Object.class
B.class
A.p
888
B.p
2000
Static Fields for B
B.q
301
Method table for B
getClass()
Object.getClass()
toString()
B.toString()
equals(Object)
Object.equals(Object)
e(char)
A.e(char)
e(double)
B.e(double)
e(int)
B.e(int)