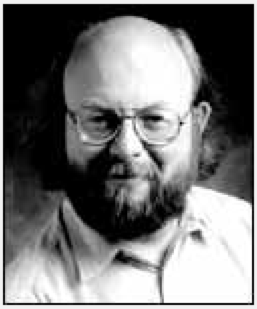
13

Portability and Safety: Java

The Java programming language was designed by James Gosling and others at Sun Microsystems. The language, arising from a project that began in 1990, was originally called *Oak* and was intended for use in a device affectionately referred to as a *set-top box*. The set-top box was intended to be a small computational device, attached to a network of some kind, and placed on top of a television set. There are various features a set-top box might provide. You can imagine some of your own by supposing that a web browser is displayed on your television set and, instead of a keyboard, you click on icons by using some buttons on your remote control. You might want to select a television program or movie or download a small computer simulation that could be executed on the computational device and displayed on your screen. A television advertisement for an automobile might allow you to download an interactive visual tour of the automobile, giving each viewer a personalized simulation of driving the car down the road. Whatever scenario might appeal to you, the computing environment would involve graphics, execution of simple programs, and communication between a remote site and a program executed locally.

At some point in the development of Oak, engineers and managers at Sun Microsystems realized that there was an immediate need for an Internet-browser programming language, a language that could be used to write small applications that could be transmitted over the network and executed under the control of any standard browser on any standard platform. The need for a standard is a result of the intrinsic desire of companies and individuals with web sites to be able to reach as large an audience as possible. In addition to portability, there is also a need for security so that someone downloading a small application can execute the program without fear of computer viruses or other hazards.

The Oak language started as a reimplementation of C++. Although language design was not an end goal of the project, language design became an important focus of the group. Some of the reasons for designing a new language are given in this overly dramatic but still informative quote from “The Java Saga” by David Bank in *Hot Wired* (December 1995):



JAMES GOSLING

James Gosling was the lead engineer and key architect behind the Java programming language and platform. He is now back in Sun's research laboratories, working on software development tools. His first project at Sun was the NeWS window system, distributed for Sun workstations in the 1980s. Before joining Sun, Gosling built a multiprocessor version of UNIX; the original Andrew window system and toolkit; and several compilers and mail systems. He is known to many as the author of the original UNIX 'Emacs.'

A techie with a sense of humor, Gosling is shown in the right-hand photograph about to put a pie in the face of a stagehand wearing a Bill Gates mask, in a picture from his partially politicized web page at http://java.sun.com/people/jag/.

James Gosling received a B.S. in Computer Science from the University of Calgary, Canada, and a Ph.D. in Computer Science from Carnegie-Mellon University for a dissertation entitled, "The Algebraic Manipulation of Constraints."

> Gosling quickly concluded that existing languages weren't up to the job. C++ had become a near-standard for programmers building specialized applications where speed is everything . . . But C++ wasn't reliable enough for what Gosling had in mind. It was fast, but its interfaces were inconsistent, and programs kept on breaking. However, in consumer electronics, reliability is more important than speed. Software interfaces had to be as dependable as a two-pronged plug fitting into an electrical wall socket. "I came to the conclusion that I needed a new programming language," Gosling says.

For a variety of reasons, including a tremendous marketing effort by Sun Microsystems, Java became surprisingly successful a short time after it was released as an Internet communication language in mid-1995.

The main parts of the Java system are

- the Java programming language,

- Java compilers and run-time systems (Java virtual machine),
- an extensive library, including a Java toolkit for graphic display and other applications, and sample Java applets.

Although the library and toolkit helped with early adoption, we will be primarily interested in the programming language, its implementation, and the way language design and implementation considerations influenced each other. Gosling, more modest in real life than the preceding quote might suggest, has this to say about languages that influenced Java: "One of the most important influences on the design of Java was a much earlier language called *Simula*. It is the first OO language I ever used (on a CDC 6400!).... [and] where the concept of a 'class' was invented."

13.1 JAVA LANGUAGE OVERVIEW

13.1.1 Java Language Goals

The Java programming language and execution environment were designed with the following goals in mind:

- *Portability:* It must be easy to transmit programs over the network and have them run correctly in the receiving environment, regardless of the hardware, operating system, or web browser used.
- *Reliability:* Because programs will be run remotely by users who did not write the code, error messages and program crashes should be avoided as much as possible.
- *Safety:* The computing environment receiving a program must be protected from programmer errors and malicious programming.
- *Dynamic Linking:* Programs are distributed in parts, with separate parts loaded into the Java run-time environment as needed.
- *Multithreaded Execution:* For concurrent programs to run on a variety of hardware and operating systems, the language must include explicit support and a standard interface for concurrent programming.
- *Simplicity and Familiarity:* The language should appeal to your average website designer, typically a C programmer or a programmer partially familiar with C/C++.
- *Efficiency:* This is important, but may be secondary to other considerations.

Generally speaking, the reduced emphasis on efficiency gave the Java designers more flexibility than the C++ designers had.

13.1.2 Design Decisions

Some of the design goals and global design decisions are listed in Table 13.1, in which + indicates that a decision contributes to this goal, − indicates that a decision detracts from this goal, and +/− indicates that there are advantages and disadvantages of the decision. Some squares are left blank, indicating that a design decision has little or no effect on the goal. We can see the relative importance of efficiency in the Java design process by looking down the rightmost column. This does not mean that efficiency

Table 13.1. Java design decisions

	Portability	Safety	Simplicity	Efficiency
Interpreted	+	+		−
Type safe	+	+	+/−	+/−
Most values are object	+/−	+/−	+	−
Objects by means of pointers	+		+	−
Garbage collection	+	+	+	−
Concurrency support	+	+		

was sacrificed needlessly, only that relative to other goals, efficiency was not the primary objective.

Interpreted. The initial and most widely used implementations of Java are based on interpreted bytecode. This is discussed in more detail in Section 13.4. In brief, Java programs are compiled to a simplified form of lower-level language. This language, called *Java bytecode*, is the form that is usually used when Java programs are sent across the network as parts of web pages. Java bytecode is executed by an interpreter called the *Java virtual machine* (JVM). One advantage of this architecture is that once a JVM is implemented for a particular hardware and operating system, all Java programs can be run on that platform without change. In addition to portability, interpreted bytecode facilitates safe execution, as commands that violate the semantics of the Java language can be recognized just before they are executed. A good example is array-bounds checking. It is not feasible to tell at compile time whether a program will access arrays out of bounds. However, the JVM does run-time tests to make sure that no Java program accesses memory incorrectly through out-of-bounds array indexing.

Type Safety. There are three levels of type safety in Java. The first is compile-time type checking of Java source code. The Java type checker works like other conventional type checkers (as in Pascal, C++, and so on), preventing compilation of programs that do not conform to the Java type discipline. There is no pointer arithmetic, there are no unchecked type casts, and the language is garbage collected, proving a greater degree of type safety than C++, for example. The second level of type safety is provided by type-checking Java bytecode programs before they are executed. The third level is provided by run-time type checks, such as the array-bounds checks described in the preceding subsection. In addition to safety, the Java type system simplifies the language to some degree by eliminating constructs that would complicate the language semantics or run-time system. There are some efficiency costs associated with run-time checking, however.

Objects and References. In Java, many things are objects but not all things. In particular, values of certain basic types such as integers, Booleans, and strings are not objects. This is a compromise between simplicity and efficiency. In particular, if all integer operations required dynamic method lookup, this would slow down integer arithmetic considerably. A simplifying decision associated with objects is that all objects are accessed by pointer, and pointer assignment is the only form of assignment provided for all objects. This simplifies programs by eliminating some special cases, but in some situations reduces program efficiency.

All parameters to Java methods are passed by value. When the parameter has a reference type (including all objects and arrays), though, it is the reference itself that is copied and passed by value. In effect, this means that values of primitive types are passed by value and objects are passed by reference.

Garbage Collection. As discussed in Subsection 6.2.1, garbage collection is necessary for complete type safety. Garbage collection also simplifies programming by eliminating the need for code that determines whether memory can be deallocated, but has a run-time cost. Java garbage collection takes advantage of concurrency. Specifically, the Java garbage collector is implemented as a low priority background thread, allowing garbage collection to occur during times when it might not affect a user's perception of running speed.

Dynamic Linking. The classes defined and used in a Java program may be loaded into the JVM incrementally, as they are needed by the running program. This shortens the elapsed time between the beginning of transmission of a program across the network and the beginning of program execution, as the program can begin executing before all of the related code is transmitted. Moreover, if a program terminates without needing some classes, these classes never need to be transmitted or loaded into the virtual machine. Dynamic linking does not have a large effect on the language design, other than to require clear interfaces that can be used to check one part of a program under assumptions about the code provided by another part of the program.

Concurrency Support. Java has a concurrency model based on threads, which are independent concurrent processes. This is a significant part of the language, both because the design is substantial and because of the importance of having standardized concurrency primitives as part of the Java language. Clearly, if Java programs relied on operating system specific concurrency mechanisms, Java programs would not be portable across different operating system platforms.

Simplicity. Although Java has grown over the years, and features like reflection and inner classes may not seem "simple," the language is still smaller and simpler in design than most production-quality general-purpose programming languages. One way to see the relative simplicity of Java is to list the C++ features that do not appear in Java. These include the following features:

- *Structures and unions:* Structures are subsumed by objects, and some uses of unions can be replaced with classes that share a common superclass.
- *Functions* can be replaced with static methods.
- *Multiple inheritance* is complex and most cases can be avoided if the simpler interface concept of Java is used.
- `Goto` is not necessary.
- *Operator overloading* is complex and deemed unnecessary; Java functions can be overloaded.
- *Automatic coercions* are complex and deemed unnecessary.
- *Pointers* are the default for objects and are not needed for other types. As a result a separate pointer type is not needed.

Some of these features appear in C++ primarily because of the C++ design goal of backward compatibility with C. Others were omitted from Java after some discussion, because it was decided that complexity of including them was more significant than the functionality they would provide. The most significant omissions are multiple inheritance, automatic conversions, operator overloading, and pointer operations of the forms found in C and C++.

13.2 JAVA CLASSES AND INHERITANCE

13.2.1 Classes and Objects

Java is written in a C++-like syntax so that programming is more accessible to C and C++ programmers. This makes the C++ one-dimensional point class used in Subsection 12.3.1 look similar when translated into Java. Here is an abbreviated version of the class, with the move method omitted:

```
class Point {
    public int getX() { ... }
    protected void setX (int x) { ... }
    private int x;
    Point(int xval) {x = xval;}
};
```

Like other class-based languages, Java classes declare the data and functions associated with all objects created by this class. When a Java object is created, space is allocated to store the data fields of the object and the constructor of the class is called to initialize the data fields. As in C++, the constructor has the same name as the class. Also following C++, the Point class has public, private, and protected components. Although public, private, and protected are keywords of Java, these visibility specifications do not mean exactly the same thing in the two languages, as explained in Subsection 13.2.2.

Java terminology is slightly different from that of Simula, Smalltalk and C++. Here is a brief summary of the most important terms used to discuss Java:

- *Class* and *object* have essentially the same meaning as in other class-based object-oriented languages, *field:* data member
- *Method:* member function, *static member*: analogous to Smalltalk class field or class method, *this:* like C++ this or Smalltalk self, the identifier this in the body of a Java method refers to the object on which this method was invoked
- *Native Method:* method written in another language, such as C
- *Package:* set of classes in shared name space.

We will look at several characteristics of classes and objects in Java, including static fields and methods, overloading, finalize methods, main methods, toString methods used to produce a print representation of an object, and the possibility of defining native methods. We will discuss the Java run-time representation of objects and the implementation of method lookup in Section 13.4 in connection with other aspects of the architecture of the run-time system.

Initialization. Java guarantees that a constructor is called whenever an object is created. Because some interesting issues arise with inheritance, this is discussed in Subsection 13.2.3.

Static Fields and Methods. Java static fields and methods are similar to Smalltalk class variables and class methods. If a field is declared to be static, then there is one field for the entire class, instead of one per object. If a method is declared static, the method may be called without using an object of the class. In particular, static methods may be called before any objects of the class are created. Static methods can access only static fields and other static methods; they cannot refer to this because they are not part of any specific object of the class. Outside a class, a static member is usually accessed with the class name, as in class_name.static_method(args), rather than through an object reference.

Static fields may be initialized with initialization expressions or a *static initialization block*. Both are illustrated in the following code:

```
class . . . {
  /* --- static variable with initial value --- */
  static int x = initial_value;
  /* --- static initialization block         --- */
  static { /* code to be executed once, when class is loaded */
  }
}
```

As indicated in the program comment, the static initialization block of a class is executed once, when the class is loaded. Class loading is discussed in Section 13.4 in connection with the JVM. There are specific rules governing the order of static initialization, when a class contains both initialization expressions and a static initialization block. There are also restrictions on the form of static initialization blocks. For example, a static block cannot raise an exception, as it is not certain that a corresponding handler will be installed at class-loading time.

Overloading. Java overloading is based on the signature of a method, which consists of the method name, the number of parameters, and the type of each parameter. If two methods of a class (whether both declared in the same class, or both inherited, or one declared and one inherited) have the same name but different signatures, then the method name is overloaded. As in other languages, overloading is resolved at compile time.

Garbage Collection and* Finalize*: Because Java is garbage collected, it is not necessary to explicitly free objects. In addition, programmers do not need to worry about dangling references created by premature deallocation of objects. However, garbage collection reclaims only the space used by an object. If an object holds access to another sort of resource, such as a lock on shared data, then this must be freed when the object is no longer accessible. For this reason, Java objects may have finalize methods, which are called under two conditions, by the garbage collector just before the space is reclaimed and by the virtual machine when the virtual machine exits. A useful convention in finalize methods is to call super.finalize, as subsequently illustrated, so that any termination code associated with the superclass is also executed:

```
class . . . {
  . . .
 protected void finalize () {
   super.finalize();
   close(file);
 }
};
```

There is an interesting interaction between finalize methods and the Java exception mechanism. Any uncaught exceptions raised while a finalize method is executed are ignored.

A programming problem associated with finalize methods is that the programmer does not have explicit control over when a finalize method is called. This decision is left to the run-time system. This can create problems if an object holds a lock on a shared resource, for example, as the lock may not be freed until the garbage collector determines that the program needs more space. One solution is to put operations such as freeing all locks or other resources in a method that is explicitly called in the program. This works well, as long as all users of the class know the name of the method and remember to call this method when the object is no longer needed.

Some other interesting aspects of Java objects and classes are main methods used to start program execution, toString methods used to produce a print representation of an object, and the possibility of defining native methods:

- main: A Java application is invoked with the name of the class that drives the application. This class must have a main method, which must be public, static, must return void, and must accept a single argument of type String[]. The main method is called with the program arguments in a string array. Any class with a main method can be invoked directly as if it were a stand-alone application, which can be useful for testing.
- toString: A class may define a toString method, which is called when a conversion to string type is needed, as in printing an object.
- native methods: A native method is one written in another language, such as C. Portability and safety are reduced with native methods: Native code cannot be shipped over the network on demand, and controls incorporated into the JVM are ineffective because the method is not interpreted by the virtual machine. The reasons for using native methods are (1) efficiency of native object code and (2) access to utilities or programs that have already been written in another language.

13.2.2 Packages and Visibility

Java has four visibility distinctions for fields and methods, three corresponding to C++ visibility levels and a fourth arising from packages.

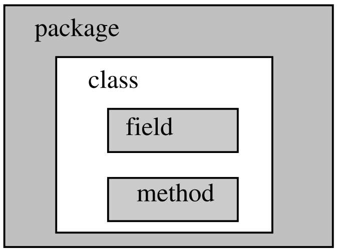


Figure 13.1. Java package and class visibility.

Java *packages* are an encapsulation mechanism similar to C++ name space that allows related declarations to be grouped together, with some declarations hidden from other packages. In a Java program, every field or method belongs to a specific class and every class is part of a package, as shown in Figure 13.1. A class can belong to the unnamed default package, or some other package if specified in the file containing the class.

The visibility distinctions in Java are

- *public:* accessible anywhere the class is visible,
- *protected:* accessible to methods of the class and any subclasses, as well as to other classes in the same package,
- *private:* accessible only in the class itself,
- *package:* accessible only to code in the same package; not visible to subclasses in other packages. Members declared without an access modifier have package visibility.

Put another way, a method can refer to the private members of the class it belongs to, nonprivate members of all classes in the same package, protected members of superclasses (including superclasses in a different package), and public members of all classes in any visible package.

Names declared in another package can be accessed with import, which imports declarations from another package, or with qualified names of the following form, which indicate the package containing the name explicitly:

$$\underbrace{\texttt{java.lang}}_{\text{package}}\,.\,\underbrace{\texttt{String}}_{\text{class}}.\,\underbrace{\texttt{substring()}}_{\text{method}}$$

13.2.3 Inheritance

In Java terminology, a *subclass* inherits from its *superclass*. The Java inheritance mechanism is essentially similar to that of Smalltalk, C++, and other class-based object-oriented languages. The syntax associated with inheritance is similar to C++, with the keyword `extends`, as shown in this example `ColorPoint` class extending the Point class from Subsection 13.2.1:

```
class ColorPoint extends Point {
    // Additional fields and methods
    private Color c;
    protected void setC (Color d) {c = d;}
    public Color getC() {return c;}
    // Define constructor
    ColorPoint(int xval, Color cval) {
        super(xval); // call Point constructor
        c = cval; }   // initialize ColorPoint field
};
```

Method Overriding and Field Hiding. As in other languages, a class inherits all the fields and methods of its superclass, except when a field or method of the same name is declared in the subclass. When a method name in the subclass is the same as a method name in the superclass, the subclass definition *overrides* the superclass method with the same signature. An overriding method must not conflict with the definition that it overrides by having a different return type. An overridden method of the superclass can be accessed with the keyword super. For fields, a field declaration in a subclass hides any superclass field with the same name. A hidden field can be accessed by use of a qualified name (if it is static) or by use of a field access expression that contains a cast to a superclass type or the keyword super.

Constructors. Java guarantees that a constructor is called whenever an object is created. In compiling the constructor of a subclass, the compiler checks to make sure that the superclass constructor is called. This is done in a specific way that programmers generally want to take into consideration. In particular, if the first statement of a subclass constructor is not a call to super, then the call super() is inserted automatically by the compiler. This does not always work well, because, if the superclass does not have a constructor with no arguments, the call super() will not match a declared constructor, and a compiler error results. An exception to this check occurs if one constructor invokes another. In this case, the first constructor does not need to call the superclass constructor, but the second one must. For example, if the constructor declaration `ColorPoint() { ColorPoint(0,blue);}` is added to the preceding ColorPoint class, then this constructor is compiled without inserting a call to the superclass Point constructor.

A slight oddity of Java is that the inheritance conventions for finalize are different from the conventions for constructors. Although a call to the superclass is required for constructors, the compiler does not force a call to the superclass finalize method in a subclass finalize method.

Final Methods and Classes. Java contains an interesting mechanism for restricting subclasses of a class: A method or an entire class can be declared final. If a method is declared final, then the method cannot be overridden in any subclass. If a class is declared final, then the class cannot have any subclasses. The reason for this feature is that a programmer may wish to define the behavior of all objects of a certain type. Because subclasses produce subtypes, as discussed in Section 13.3, this requires some restriction on subclasses. To give an extreme example, the singleton pattern discussed in Section 10.4 shows how to design a class so that only one object of the class can be created. The pattern hides the constructor of the class and makes public only a function that will call the constructor once during program execution. This pattern solves the problem of restricting the number of objects of the class, but only if no subclass overrides the public method with a method that can create more than one object. If a programmer really wants to enforce the singleton pattern, there must be a way to keep other programmers from defining subclasses of the singleton class.

The Java class java.lang.System is another example of a final class. This class is final so that programmers do not override system methods.

In a loose sense, Java final is the opposite of C++ virtual: Java methods can be overridden until they are marked final, whereas C++ member functions can be overridden only if they are virtual. The analogy is not exact, though, as a C++ member function cannot be virtual in one class and nonvirtual in a base class or derived class, because that would violate the requirement that base- and derived-class vtables must have the same layout.

Class *Object.* In principle, every class declared in a Java program extends another class, as a class without an explicit superclass is interpreted as a subclass of the class Object. The class Object is the one class that has no superclasses. Class Object contains the following methods, which can be overridden in derived classes:

- GetClass, which returns the Class object that represents the class of the object. This can be used to discover the fully qualified name of a class, its members, its immediate superclass, and any interfaces that it implements.
- ToString, which returns a String representation of an object.
- equals, which defines a notion of object equality based on value, not reference, comparison.
- hashCode, which returns an integer that can be used to store the object in a hash table.
- clone, which is used to make a duplicate of an object.
- Methods wait, notify, and notifyAll used in concurrent programming.
- finalize, which is run just before an object is destroyed (discussed in Subsection 13.2.1 in connection with garbage collection),

Because all classes inherit the methods of class Object, every object has these methods.

13.2.4 Abstract Classes and Interfaces

The Java language has an *abstract-class* mechanism that is similar to C++. As we discussed in Chapter 12, an abstract class is a class that does not implement all of its methods and therefore cannot have any instances. Java uses the keyword abstract instead of the C++ "=0" syntax, as shown in the following code:

```
abstract class Shape {
  ...
  abstract point center();
  abstract void rotate(degrees d);
  ...
}
```

Java also has a “pure abstract” form of class called an interface. An interface is defined in a manner similar to that of a class, except that all interface members must be constants or abstract methods. An interface has no direct implementation, but classes may be declared to implement an interface. In addition, an interface may be declared as an extension of another, providing a form of interface inheritance.

One reason that Java programmers use interfaces instead of pure abstract classes when a concept is being defined but not implemented is that Java allows a single class to implement several interfaces, whereas a class can have only one superclass. The following interfaces and class illustrate this possibility. The Shape interface identifies some properties of simple geometric shapes, namely, each has a center point and a rotate method. Drawable similarly identifies properties of objects that can be displayed on a screen. If circles are geometric shapes that can be displayed on a screen, then the Circle class can be declared to implement both Shape and Drawable, as subsequently shown.

Interfaces are often used as the type of argument to a method. For example, if the windows system has a method for drawing items on a screen, then the argument type of this method could be Drawable, allowing every object that implements the Drawable interface to be displayed:

```
interface Shape {
  public Point center();
  public void rotate(float degrees);
}
interface Drawable {
  public void setColor(Color c);
  public void draw();
}
class Circle implements Shape, Drawable {
  // does not inherit any implementation
  // but must define Shape, Drawable methods
}
```

Unlike C++ multiple inheritance (discussed in Section 12.5), there is no name clash problem for Java interfaces. More specifically, suppose that the preceding two interfaces Shape and Circle also both define a Size method. If the two Size methods both have the same number of arguments and the same argument types, then class Circle must implement one Size method with this number of arguments and the argument types given in both interfaces. On the other hand, if the two Size methods have a different number of arguments or the arguments can be distinguished by type, then these are considered two different method names and Circle must define an implementation for each one. Because Java method lookup uses the method name and number and types of arguments to select the method code, the two methods with the same name will be treated separately at method lookup time.

13.3 JAVA TYPES AND SUBTYPING

13.3.1 Classification of Types

The Java types are divided into two categories: primitive types and reference types. The eight primitive types are the boolean type and seven numeric types. The seven numeric types are the forms of integers byte, short, int, long, and char and the floating-point types float and double. The three forms of reference types are class types, interface types (subsequently discussed), and array types. There is also a special null type. The values of a reference type are references to objects (which include arrays). All objects, including arrays, support the methods of class Object.

The subtyping relationships between the main families of types are illustrated in Figure 13.2, which includes a Shape interface and Circle and Square classes to show how user-defined classes and interfaces fit into the picture. Other predefined types such as String, ClassLoader, and Thread occupy positions similar to that of Exception in this figure. Object[] is the type of arrays of objects, and similarly for array types Shape[], Circle[], and Square[].

Although it is standard Java terminology to call Object and its subtypes reference types, this may be slightly confusing. Although C++ distinguishes Object from Object *, there are no explicit pointer types in Java. Instead, the difference between a pointer to an object and an object itself is implicit, with pointer dereferencing combined with operations like method invocation and field access. If T is a reference type, then a variable x of type T is a reference to T objects; in C++ the variable x would have type T*. Because there is no explicit way to dereference x to get a value of type T, Java does not have a separate type for objects not referred to by pointer.

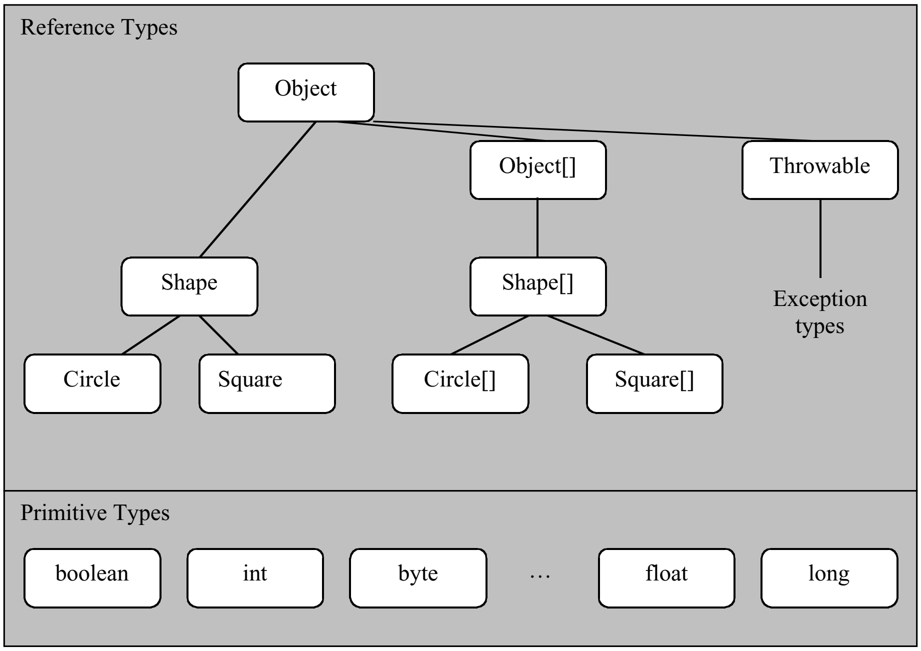


Figure 13.2. Classification of types in Java.

Because every class is a subtype of Object, variables of type Object can refer to objects or arrays of any type.

13.3.2 Subtyping for Classes and Interfaces

Subtyping for classes is determined by the class hierarchy and the interface mechanism. Specifically, if a class A extends class B, then the type of A objects is a subtype of the type of B objects. There is no other way for one class to define a subtype of another, and no private base classes (as in C++) to allow inheritance without subtyping.

A class may be declared to implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified in the interface. This (multiple) interface subtyping allows objects to support (multiple) common behaviors without sharing any common implementation.

Run-Time Type Conversion. Java does not allow unchecked casts. However, objects of a supertype may be cast to a subtype by a mechanism that includes a run-time type test. If you want to make a list class in Java, you make it hold objects of type Object. Objects of any class can then be put onto a list, but, to take them off the list and use them nontrivially, they must be converted back to their original type (or some supertype). In Java, type conversion is checked at run time, raising an exception if the object does not have the designated type.

Implementation. Java uses different bytecode instructions for member lookup by interface and member lookup by class or subclass. In a high-performance compiler, lookup by class could be implemented as in C++, with offset known at compile time. However, lookup by interface cannot, because a class may implement many interfaces and the interfaces may list members in different orders. This is discussed in detail in Subsection 13.4.4.

13.3.3 Arrays, Covariance, and Contravariance

For any type T, Java has an *array* type T[ ] of arrays whose elements have type T. Although array types are grouped with classes and interfaces, it is not possible to inherit from an array type. In Java terminology, array types are final.

Array types are subtypes of Object, and therefore arrays support all of the methods associated with class Object. Like other reference types, an array variable is a pointer to an array and can be null. It is common to create arrays when an array reference is declared, as in

```
Circle[ ] x = new Circle[array_size]
```

However, it is also possible to create array objects “anonymously,” in much the same way that we can create other Java objects. For example,

```
new int[ ] {1,2,3, . . . 10}
```

is an expression that creates an integer array of length 10, with values 1, 2, 3, . . . , 10. Because a variable of type T[ ] can be assigned an array of any length, the length of an array is not part of its static type.

There are some complications surrounding the way that Java array types are placed in the subtype hierarchy. The most significant decision is that if A <: B then the Java type checker also uses the subtyping A[ ] <: B[ ]. This introduces a problem often referred to as the *array covariance problem*. Consider the following class and array declarations:

```
class A {. . .}
class B extends A {. . .}
B[ ] bArray = new B[10]
A[ ] aArray = bArray        // considered OK since B[ ] <: A[ ]
aArray[0] = new A()     // allowed but run-time type error; raises
ArrayStoreException
```

In this code, we have B <: A because class B extends class A. The array reference bArray refers to an array of B objects, initially all null, and the array reference aArray refers to the same array. The declaration A[ ] aArray = bArray is allowed by the Java type checker (although semantically it should not be allowed) because of the Java design decision that if B <: A then B[ ] <: A[ ]. The problem with allowing the A[ ] array reference aArray to refer to an array of B objects is illustrated by the last statement. The assignment aArray[0] = new A() looks perfectly reasonable: aArray is an array with static type A[ ], suggesting that it is acceptable to assign an A object to any location in the array. However, because aArray actually refers to an array of B objects, this assignment would violate the type of bArray. Because this assignment would cause a typing problem, the Java implementation does not allow the assignment to be executed. A run-time test will determine that the value being assigned to an array of B objects is not a B object and the ArrayStoreException exception will be raised.

Although the Java designers considered array covariance advantageous for some specific purposes (writing some binary copy routines), array covariance in Java leads to some confusion and many run-time tests. It does not seem to be a successful language design decision.

13.3.4 Java Exception Class Hierarchy

Java programs may declare, raise, and handle exceptions. The Java exception mechanism has the general features we discussed in Section 8.2. Java exceptions may be the result of a throw statement in a user program or the result of some error condition detected by the virtual machine such as an attempt to index outside the bounds of an array. In Java terminology, an exception is said to be *thrown* from the point where it occurred and is said to be *caught* at the point to which control is transferred. As in other languages, throwing an exception causes the Java implementation to halt every expression, statement, method or constructor invocation, initializer, and field initialization expression that has started but not completed execution. This process continues until a handler is found that matches the class of the exception that is thrown.

One interesting aspect of the Java exception mechanism is the way that it is integrated into the class and type hierarchy. Every Java exception is represented by an instance of the class Throwable or one of its subclasses. An advantage of representing exceptions as objects is that an exception object can be used to carry information from the point at which an exception occurs to the handler that catches it. Subtyping can also be used in handling an exception: A handler matches the class of an exception if it explicitly names the class of the exception that is thrown or names some superclass of the class of the exception.

The Java exception mechanism is designed to work well in multithreaded (concurrent) programs. When an exception is thrown, only the thread in which the throw occurs is affected. The effect of an exception on the concurrent synchronization mechanism is that locks are released as synchronized statements and invocations of synchronized methods complete abruptly. We will return to this topic in Chapter 14.

Java exceptions are caught inside a construct called a *try-finally block*. Here is an example outline, showing a block with two handlers, each identified by the catch keyword. Intuitively, a try-finally block tries to execute a sequence of statements. If the sequence of statements terminates normally, then that is the end result of the block. However, if an exception is raised, it may be caught inside the block. If an exception is raised and caught, then the sequence of statements following the keyword finally will be executed after the exception handler has finished:

```
try {
        ⟨statements⟩
} catch (⟨ex-type1⟩ ⟨identifier1⟩) {
        ⟨statements⟩
} catch (⟨ex-type2⟩ ⟨identifier2⟩) {
        ⟨statements⟩
} finally {
        ⟨statements⟩
}
```

Although it may not be apparent why, there is some complication in the JVM associated with try-finally blocks. Specifically, a significant fraction of the complexity of the Java bytecode verifier is a result of the way finally clauses are implemented as a form of "local subroutine" (called jsr) in the Java bytecode interpreter. We will discuss the JVM in Section 13.4.

The classes of exceptions are shown in Figure 13.3. Every exception is, by definition, an object of some subclass of Throwable. The class Throwable is a direct subclass of Object. Programs can use the preexisting exception classes in throw statements or define additional exception classes. Additional exceptions classes must be subclasses of Throwable or one of its subclasses. To take advantage of the Java platform's compile-time checking for exception handlers, it is typical to define most new exception classes as checked exception classes. These are subclasses of Exception that are not subclasses of RuntimeException.

The phrases checked exceptions and "unchecked exceptions" refer to compile-time checking of the set of exceptions that may be thrown in a Java program. The

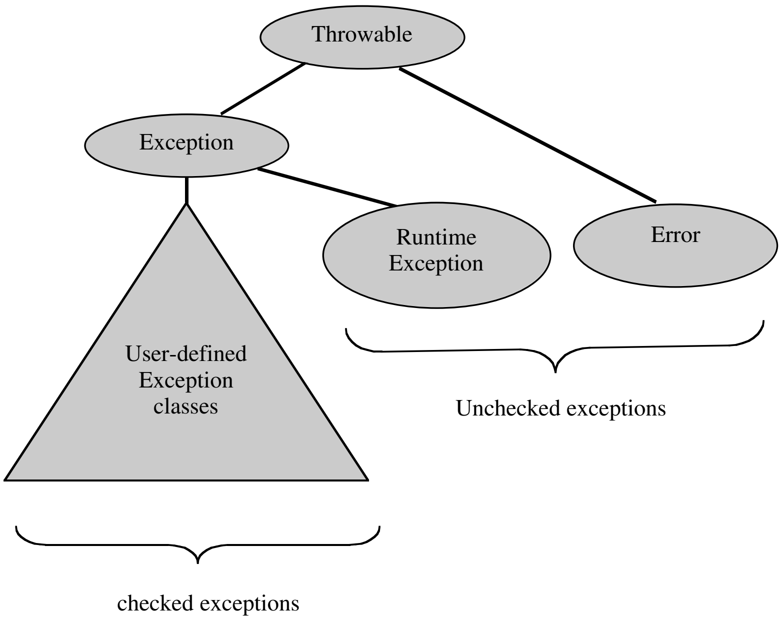


Figure 13.3. Java exception classes.

Java compiler checks, at compile time, that a program contains handlers for every checked exception. We accomplish this by analyzing the checked exceptions that can be thrown during execution of a method or constructor. These checks are based on the declared set of exceptions that a Java method could throw, called the throws clause of the method. For each checked exception that is a possible result of a method call, the throws clause for the method must mention the class of that exception or one of the superclasses of the class of that exception. This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions that are not properly handled.

Because Error and RuntimeException exceptions are generally thrown by the Java run-time system and not by the user code, it is not necessary for a programmer to declare these exceptions in the throws clause of a method. More specifically, the run-time exception classes (RuntimeException and its subclasses) are exempt from compile-time checking because, in the judgment of the designers of the Java programming language, declaring such exceptions would not aid significantly in establishing the correctness of programs. Many of the operations and constructs of the Java programming language can result in run-time exceptions. The information available to a compiler and the level of analysis the compiler performs are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer.

Ordinary programs do not usually recover from exceptions from the class Error or its subclasses. The class Error is a separate subclass of Throwable, distinct from Exception in the class hierarchy, to allow programs to use the idiom

```
} catch (Exception e) {
```

to catch all exceptions from which recovery may be possible without catching errors from which recovery is typically not possible.

13.3.5 Subtype Polymorphism and Generic Programming

In Section 9.4, we discussed some approaches to generic programming based on templates and related mechanisms such as generic modules and packages. The main idea is that we can define a generic data structure, such as a list or binary search tree, so that any type of data can be inserted into or retrieved from the data structure, provided that a few required operations, such as equality test or a linear ordering, are defined and implemented on the data. Generic algorithms such as sorting, applicable to any list or array of data, can be defined similarly by use of templates or related mechanisms. Templates and generic modules are useful in generic programming because they provide a form of polymorphism: The same code may be instantiated and executed on a variety of types of data.

In object-oriented programming, the term polymorphism is often used to refer to a specific kind of polymorphism that we call *subtype polymorphism*. In subtype polymorphism, a single piece of code, typically a set of functions or methods, can be applied to more than one type of argument. However, the language mechanism for allowing this is not through implicit type parameters (as in ML type-inference polymorphism) or explicit type parameters (as in C++ templates or Ada generic packages), but through subtyping. Specifically, in a language with subtyping, if a method m will accept any argument of type A, then m may also be applied to any argument from any subtype of A. This works almost as well as parametric polymorphism (polymorphism based on type parameters), except that typically the compile-time type checker has less information about the types of arguments. This either leads to run-time type checking or some sacrifice in type safety. The general phenomenon is illustrated in the following Java example.

Example 13.1 Java Subtype Polymorphism

Let us consider the problem of defining a general class of stacks. We would like to be able to build stacks of any type of object. We will implement stacks as linked lists of stack nodes, each node containing a stack element.

Because Object is a supertype of all reference types (including all references to objects and arrays), we can let stack cells contain references of type Object. Let us begin with a subsidiary class definition, the class of stack nodes:

```
class Node {
  Object element;   // Any object can be placed in a stack
  Node next;
  Node (Node n, int e) {
    next=n;
    element=e;
  }
}
```

This class is intended to belong to the same package as that of the Stack class. Because there are no access qualifiers here (public, private, or protected) the fields and methods are accessible only to methods in the same package.

Here is the outline of a Stack class in which Node objects are used to store elements of a linked list of stack elements. The methods shown here are empty, which tells whether a stack is empty, push, which adds a new element to the top of a stack, and pop, which removes the top node and returns the object stored in that node. The class is declared public so that it is visible outside the package in which it is defined:

```
public class Stack {
   private Node top=null;           // top of the stack, starts as empty
   public Stack(){
   }
   public boolean empty(){          // determine whether stack is empty
       return top == null;
   }
   public void push(Object val) {   // push element on top of stack
       top = new Node(top,val);
   }
   public Object pop() {            // Remove first node and return value.
       if (top==null) return -1;
       Node temp = top;
       top = top.next;
       return temp.element;
   }
}
```

Most of the code here is entirely straightforward. The important limitation, for the purpose of generic programming, is the type of the node elements and the return type of pop. Because the argument type of push is Object, any object can be pushed onto a stack. However, the return type of pop is also Object, meaning that, when an object is removed from a stack, the static type of the expression removing the object is Object. Therefore, if a program pushes strings onto a stack and then pops them off the stack, the Java type checker will not be able to determine that the objects removed from the stack are strings.

Here is an example of the way that stacks might be used in a Java program:

```
String s = “Hello”;
Stack st = new Stack();
. . .
st.push(s);
. . .
s = (String) st.pop;
```

In this code fragment, a String object is pushed onto a stack. When the String object is popped off the stack, the static type of st.pop() is Object, not string. Therefore, to assign st.pop() to a string variable, it is necessary to cast the Object to String. This cast generates a run-time test. If the top element on the stack is not a string when this assignment is executed, the run-time test will throw an exception and prevent the assignment from occurring.

In comparison, a C++ stack class template may be defined by the following form:

```
template <typename t> class Stack {
   private: Node<t> top;
   public: boolean empty() {. . .};
             void    push (t* x) {. . .}
             t*      pop ( ) {. . .}
};
```

If Java had a template mechanism similar to this, then we could define a generic stack class like this,

```
class Stack<A> {
  public Boolean empty(){. . .}
 public void push(A a) {. . .}
  public A pop() {. . .}
}
```

and use generic stacks as follows:

```
String s = “Hello”;
Stack<String> st = new Stack<String>();
st.push(s);
. . .
s = st.pop();
```

There are several important points of comparison between the two styles illustrated here. In the template-based code, the stack st has type Stack<string> instead of Stack. As a result, st.pop() has static type String. Two advantages are clarity and efficiency. The programmer’s intent is clearer, as the stack st is explicitly declared to be a stack of strings. The efficiency advantage is that there is no need for a cast and run-time test.

There have been several research projects aimed at adding a template mechanism to Java. Although the basic goals of such a mechanism seem clear, there are a number of details that need to be addressed. One issue is implementation. One approach is to simply translate Java with templates into Java without templates. This allows programmers to write code with templates and provides a relatively simple implementation. However, the result of the translation is code that contains many run-time-checked type conversions, so it may not be as efficient as other implementation techniques. An alternative is to compile a generic class such as Stack<t> into a form of class file that has type parameters and load class Stack<String> by instantiating this class file at class-load time. This produces more efficient code, but might involve loading more classes. Because class loading is slow, the overall program may run more slowly in some cases. There is also the problem of instantiating class templates for primitive types such as boolean, int, float, and so on. The first implementation method will not work for primitive types, as these are stored differently from object types. The second method, instantiating class files at load time, can be made to work, although handling primitive types adds some complexity to the mechanism. In either case, there are some additional subtleties in the design of the type-checking mechanism for generic classes that we have not discussed.

13.4 JAVA SYSTEM ARCHITECTURE

13.4.1 Java Virtual Machine

There are several implementations of Java. This section discusses the implementation architecture used in the Sun Java compiler and the JVM. The biggest difference between this architecture and some others is that some optimized implementations generate native code for the underlying hardware instead of interpreted code. However, because the "just-in-time" or (JIT), compilers that generate native code on the fly are significantly more complicated than the standard JVM architecture, we use the JVM architecture as the basis for our discussion and analysis of Java implementation issues.

The Java compiler produces a form of machine code called *bytecode*. The origin of the name bytecode, which is a standard term that predates Java, is that bytecodes are instructions for a *virtual machine* that may be 1 byte long. Many compilers for high-level languages actually produce bytecode instead of native machine code. For example, the influential UCSD Pascal compiler produces a form of bytecode called P-code, which is then interpreted by a virtual machine. Common ML and Smalltalk implementations also generate virtual machine bytecode.

Figure 13.4 shows the main parts of the JVM and some of the steps involved in compiling and executing Java source code. The main parts of the JVM are the class loader, the bytecode verifier, the linker, and the bytecode interpreter.

The top of Figure 13.4 shows a Java source code file A.java compiled to produce a *class file* A.class. The class file is in a specific format, containing bytecode instructions and some auxiliary data structures, including a symbol table called a *constant pool*. The class loader reads the class file and arranges the bytecode instructions in a form suitable for the verifier. The Java bytecode language contains type information, allowing the verifier to check that bytecode is type correct before it is executed by the virtual machine. The linker resolves interfile references, and the bytecode interpreter executes the bytecode instructions.

The box labeled B.class in Figure 13.4 is meant to show that Java classes are loaded incrementally, as needed during program execution, and may be loaded over

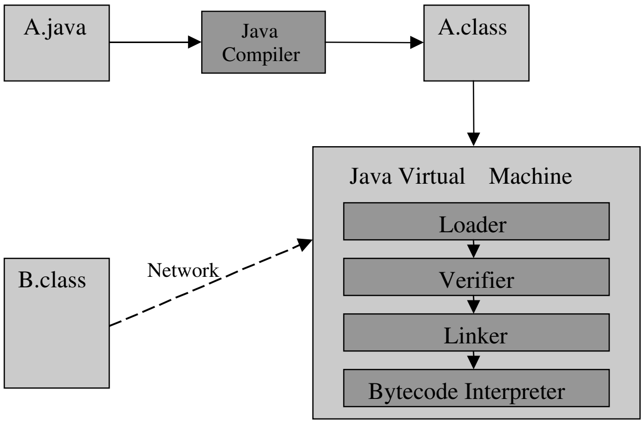


Figure 13.4. JVM.

the network as well as from the local file system. More specifically, if class A has a main method, then class A may be loaded as a main program and executed. If some method of class A refers to class B, then class B will be loaded when this method is called. The class loader then attempts to find the appropriate class file and load it into the virtual machine so that program execution may continue.

The JVM may run several threads (processes) concurrently; it terminates when either all nondaemon threads terminate or some thread runs the exit method of the class Runtime or class System.

13.4.2 Class Loader

The Java run-time system loads classes, as they are needed, searching compiled bytecodes for any class that is referenced but not already loaded. There is a default loading mechanism, which we can replace by defining an alternative ClassLoader object. One reason to define an alternative ClassLoader is to obtain bytecodes from some remote source and load them at run time.

We define a class loader by extending the abstract ClassLoader class and implementing its loadClass method. Here is a possible form of a class-loader definition:

```
class Load_My_Classes extends ClassLoader {
  private Hashtable loaded_classes = new Hashtable();
  public Class loadClass(string name, boolean resolve)
       throws ClassNotFoundException {
     /* Code to try to load class “name” and add to “loaded_classes”.
        If boolean arg “resolve” is true, then invoke “resolveClass”
        method to ensure that all classes referred to by class are
       loaded. Raise exception if class cannot be found and loaded. */
  }
  ...
}
```

When loadClass finds the bytecodes of the class to be loaded, it may invoke the inherited defineClass method, which takes a byte array, a start position, and a number of bytes. This method installs this sequence of bytes as the implementation of the desired class.

The return type of method loadClass is Class. Classes are objects, and we may be able to find the class loader for a given class by invoking the getClassLoader method of the class. If the class has no class loader, the method returns null.

13.4.3 Java Linker, Verifier, and Type Discipline

The linker takes a binary form of class or interface as input and adds it to the run-time state of the JVM.

Each class is verified before it is linked and made available for execution. Java bytecode verification is a specific process that checks to make sure that every class has certain properties:

- Every instruction must have a valid operation code.
- Every branch instruction must branch to the start of some other instruction, rather than the middle of some instruction.
- Every method has a structurally correct signature.
- Every instruction obeys the Java type discipline.

If an error occurs in verification, then an instance of the class VerifyError will be thrown at the point in the Java program that caused the class to be verified.

Linking involves creating the static fields of a class or interface and initializing them to the standard default (typically 0). Names are also resolved, which involves checking symbolic references and replacing them with direct references that can be processed more efficiently if the reference is used repeatedly.

Classes are not explicitly unloaded. If no instance of class is reachable, the class may be removed as part of garbage collection.

13.4.4 Bytecode Interpreter and Method Lookup

The bytecode interpreter executes the Java bytecode and performs run-time tests such as array-bound tests to make sure that every array access is within the declared bounds of the array. The basic run-time architecture of the bytecode interpreter is similar to the simple machine architecture outlined in Section 7.1, with a program counter, instruction area, stack, and heap.

A running Java program consists of one or more sequential threads. When a new thread is created, it is given a program counter and a stack. The program counter indicates the next instruction to execute. The Java stack is composed of activation records, each storing the local variables, parameters, return value, and intermediate calculations for a Java method invocation. The JVM has no registers to hold intermediate data values; these are stored on the Java stack. This approach was taken by the Java designers to keep the JVM instruction set compact and to facilitate implementation on a variety of underlying machine architectures. When an object or array is created, it is stored on the heap. All threads running within a single JVM share the same heap.

A Java activation record has three parts: local variable area, operand stack, and data area. The sizes of the local variable area and operand stack, which are measured in words, depend on the needs of the method and are determined at compile time. The local variable area of a Java activation record is organized as an array of words, with different types of data occupying different numbers of words. The operand stack is used for intermediate calculations and passing parameters to other methods. Instead of accessing data from memory by using a memory address, most Java bytecode instructions operate by pushing, popping, or replacing values from the top of this stack-within-a-stack. An add instruction with two operands, for example, pops its two operands off the operand stack and pushes their sum onto the operand stack. This kind of architecture leads to shorter instruction codes because an instruction with several operands needs to identify only the operation – it is not necessary to give an address or register number for the operands because the operands are always the top few data items on the stack. The data area of an activation record stores data used to support constant pool resolution (which we discuss in the next paragraph), normal method return, and exception dispatch.

Compiled Java bytecode instructions are stored in a file format that includes a data structure called the constant pool. The constant pool is a table of symbolic names, such as class names, field names, and methods names. When a bytecode instruction refers to a field, for example, the reference will actually be a number, representing an index into the constant pool. In the instruction to get a field, for example, the instruction may contain the number 27, indicating the 27th symbolic name in the constant pool. This approach stores symbolic names from the source code in the bytecode file, but saves space by storing each symbolic name only once.

When executing programs, the bytecode interpreter makes the following run-time tests to prevent type errors and preserve the integrity of the run-time system:

- All casts are checked to make sure they are type safe.
- All array references are checked to make sure the array index is within the array bounds.
- References are tested to make sure they are not null before they are dereferenced.

In addition, the absence of pointer arithmetic (see Section 5.2) and the use of automatic garbage collection (programmers cannot explicitly free allocated memory) contribute to type-safe execution of Java programs.

When instructions that access a field or method are executed, there is often a search for the appropriate location in an object template or method lookup table, by use of the symbolic name of the field or method from the constant pool. This is inefficient if access through the constant pool occurs frequently. Search through the constant pool is also an execution bottleneck for concurrent programs, as this may require a lock on the constant pool and locking the constant pool restricts execution of other threads.

The JVM implementation optimizes the search for object fields and methods by modifying bytecode during program execution. More specifically, bytecode instructions that refer to the constant pool are modified to an equivalent so-called *quick* bytecodes, which refers to the absolute address of a field or method. For example, the bytecode instruction

```
getfield #18 <Field Obj var>
```

pushes the value of an object field onto the operand stack. (The object whose field is retrieved is the object that is on top of the operand stack before the operation is performed.) When the getfield instruction is executed, the constant pool is searched for the symbolic field name that matches the string stored at position #18 in the constant pool. If this field is found, and it is located 6 bytes below the first location of the object, then the preceding instruction is replaced with the following quick version,

```
getfield_quick 6
```

which uses the calculated offset of the field from the top of the object. If the program passes through this instruction again, as when the instruction appears inside a loop, for example, the quick instruction is used. You do not need to understand all of the parts of the getfield instruction – the important issue here is the way that modifying a bytecode instruction the first time it is executed can make the program run more quickly if this instruction is executed again.

Finding a Virtual Method by Class

There are four different bytecodes for invoking a method:

- invokevirtual: used when a superclass of the object is known at compile time,
- invokeinterface: used when only an interface of the object is known at compile time,
- invokestatic: used to invoke static methods,
- invokespecial: used in some special cases that will not be discussed.

In simplest terms, invokevirtual is used in situations that resemble C++ virtual function calls, as Java is a statically typed programming language and all subclasses of a class can be implemented with the same method table (vtable) order. Invokeinterface similarly corresponds to Smalltalk method lookup, as an interface implemented by the class of an object does not determine the relative position of the method in a method table. We will look at invokevirtual before considering properties of invokeinterface.

Like getfield, invokevirtual involves searching for a symbolic name the first time the instruction is executed. More specifically, suppose we have a Java source code declaring an object reference x and invoking a method of x:

```
Object x;
...
x.equals(“test”);
```

For concreteness, and because this affects the way the code is compiled, let us assume that x is the first local variable in this block. The call x.equals("test") is compiled into something like the following sequence of bytecodes, in which the phrases to the right of ";" are comments:

```
aload_1      ; push local variable 1 (which is 'x') onto the operand stack
ldc "test" ; push the string "test" onto the operand stack
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
```

The string java/lang/Object/equals(Ljava/lang/Object;)Z is called a method specification. This string is not literally located in the bytecode, but is stored in the constant pool, as previously described.

When invokevirtual is executed, the virtual machine looks at method specification and related information in the class file and determines how many arguments the method requires. For our example equals method, there is one argument. The argument and the object reference, in this case the number 1 indicating the first local variable in the current scope, are popped off the operand stack. Using the information stored in the object reference (local variable x), the virtual machine retrieves the Java class for the object, searches the list of methods defined by that class and then its superclasses, looking for a method matching the method specification. When the correct method is located, the method is called in the same way that function calls are executed in most block-structured programming languages.

***Bytecode Rewriting for** Invokevirtual.* Although this method lookup process will find and correctly call a method, it does not take advantage of the static type information computed by the Java compiler at compile time. More specifically, the Java compiler can arrange every subclass method table in the same way as the superclass table, just as the C++ compiler does. By doing this, the compiler can guarantee that each method is located at the same relative position in all method tables associated with all subclasses of any class. Therefore, once method lookup is used to find a method in a method table, the offset of this method will be the same for all future executions of this instruction.

Figure 13.5 shows how bytecode can be modified so that on subsequent executions of an invokevirtual instruction, the correct method can be found immediately within the method table (called "mtable" in the illustration) of the object.

The part of Figure 13.5 above the dotted line shows the invokevirtual command in the instruction stream, followed by a pointer to the method specification in the constant pool. After the method is found, the instruction stream can be modified as shown below the dotted line. In the modified instruction stream, invokevirtual is replaced with the "quick" version invokevirtual_quick and the method specification pointer is replaced with the offset of the method in the method table. When invokevirtual_quick is executed, the virtual machine will use the object reference on the operand stack to locate the class and method table and then use the mtable offset in the instruction stream to find the appropriate method without further search.

It is important to realize that if the method invocation is in a loop, for example, then different objects may be on the operand stack for different executions of the

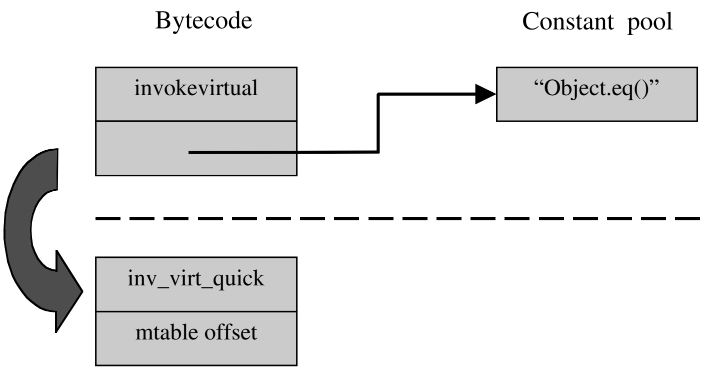


Figure 13.5. Optimizing invokevirtual by rewriting bytecode.

method call. As a result, the class of the object may change from one call to the next. However, the offset of the method in the method table will be the same on each call, as each class is a subclass of the static type of the object reference in the Java source code.

A short summary of this optimization is that the first time a method call instruction is executed, Smalltalk-style method search is used to find the position of the method in a method table. After the search succeeds, the code is changed so that if this method call instruction is executed again, C++-style lookup will find the method quickly.

Finding a Virtual Method by Interface

The situation is similar for finding a method when an interface of the object is known at compile time. The difference is that the optimization involving invokevirtual_quick will not work correctly. Therefore, another technique is used.

Suppose a compiled Java program contains a method declaration

```
void add2(Incrementable x) { x.inc(); x.inc(); }
```

where Incrementable is an interface guaranteeing that the argument x will have an inc method when add2 is called at run time. There may be several classes that implement the Incrementable interface. For example, there could be classes IntCounter and FloatCounter of the following form:

```
interface Incrementable {
   public void inc();
}
class IntCounter implements Incrementable {
   public void add(int);
   public void inc();
   public int value();
}
class FloatCounter implements Incrementable {
   public void inc();
```

```
    public void add(float);
    public float value();
}
```

Because the classes implementing Incrementable might be declared in different packages, neither referring to the other, there is no reason to believe that the method inc is located at the same offset in the two method tables. (In fact, it is easy to construct a set of interfaces and classes for which there is no way of making all classes that implement each interface compatible in this way.)

When the add2 method is compiled, the Java compiler generates code of the following form (with method specification from the constant pool written in the code for clarity):

```
aload_1 ; push local variable 1 (i.e. x) onto the stack
invokeinterface package/inc()Z 1
```

When invokeinterface is executed, the class of the object is found and the method table for this class is searched to find the method with the given method specification. When the method is located, the method is called as for invokevirtual.

Because the offset of the method in the method table may be different on the next execution of this instruction, it is not correct to rewrite invokeinterface to invokevirtual_quick. However, there is some possibility that the next execution of this instruction will be for an object from the same class, so it seems wasteful to discard the offset and other information. Figure 13.6 shows the bytecode modification used for invokeinterface.

The part of Figure 13.6 above the dotted line shows the initial instruction sequence, with the invokeinterface instruction followed by a pointer to the method specification in the constant pool and an unused empty location. After execution of invokeinterface, the instruction is replaced with invokeinterface_quick and a pointer to the class and method is stored below the method specification. When

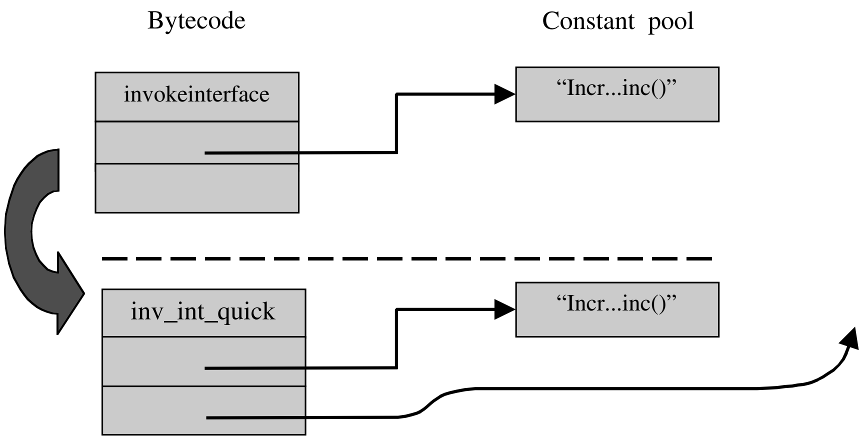


Figure 13.6. Optimizing invokeinterface by rewriting bytecode

invokinterface_quick is executed on another pass through this instruction sequence, the class of the object operand is compared with the cached class. If the object is from the same class, then the same method is used. If the object is from a different class, then, because the method specification is still available in the instruction sequence, the same search process can be followed as if the instruction were invokeinterface again.

Efficiency Summary. In Java, the method call overhead is high the first time an instruction is executed and potentially much lower on subsequent executions. On the first lookup that uses invokevirtual there are two indirections (one to look up the constant table of the class this method is in and one to look up the method table of the class of the object), two array indexes (in the constant table and the method list), and the cost of rewriting some bytecode. On subsequent calls, there is one indirection and one array index. For invokeinterface, there is a search through the method table the first time an instruction is executed. On subsequent calls, there is either one array index or another search through a method table if the class is changed.

13.5 SECURITY FEATURES

Computer security involves preventing unauthorized use of computational resources. Some computer security topics are network security, which provides methods for preserving the secrecy and integrity of information transmitted on the network, and operating system security, which is concerned with preventing an attacker from gaining access to a system. In this section, we look at ways that programming languages can help and hinder our efforts to build secure systems, with particular attention to how Java is designed to improve the security of systems that use Java.

One way that attackers may try to gain access to a system is to send data over the network to a program that processes network input. For example, many attacks against the e-mail program Sendmail have been devised over the years. These attacks typically take advantage of some error or oversight in a particular implementation of Sendmail. Sendmail is a target because this program accepts network connections and runs with superuser privileges. In Subsection 13.5.1 we will look at buffer overflow attacks, which are an important example of the kind of attack that is often mounted against network programs.

Because Java code is sometimes transmitted over the network, a computer user may run code that comes from another site, such as a web page. *Mobile code,* a general term for code transmitted over the network before it is executed, provides another way for an attacker to try to gain access to a system or cause damage to a system. The two main mechanisms for managing risks raised by mobile code are called *sandboxing* and *code signing*. The idea behind sandboxing, which was invented before the development of Java, is to give a program a restricted execution environment. This restricted environment, called a sandbox, may contain certain functions (analogous to certain toys you might give a child to play with in a sandbox), but the program cannot call functions or perform actions that are outside the sandbox.

Code signing is used to determine which person or company produced a class file. With digital signatures, it is possible for the producer of code to sign it in a way that any recipient can verify the signature. The JVM uses digital signatures to identify the producer of a file that contains Java bytecode. The user of a computer system can specify a set of trusted producers and allow code from these producers to execute on their computer. With the Java security manager, it is also possible to assign different access permissions to different code producers.

After looking at buffer overflow, we will discuss some of the main features of the Java sandbox. If you want to learn more about the Java security model, you can read one of the books on Java security or find additional information on the web.

13.5.1 Buffer Overflow Attack

In *a buffer overflow attack*, an attacker sends network messages that cause a program to read data into a buffer in memory. If the program does not check the number of bytes placed in the buffer, then the attacker may be able to write past the allocated memory. By writing over the return address that is stored in the activation record of a function, the attacker may cause the program to misbehave and "return" by jumping to instructions chosen by the attacker. This is a powerful form of attack. The vast majority of security advisories issued by the Computer Emergency Response Team (CERT) over the past decade or two have been security problems related to buffer overflow.

Java code is not vulnerable to buffer overflow attacks because the JVM checks array bounds at run time. This prevents a function from writing more data into an array than the array can hold. To show how important this feature is for Java security, we look at how a buffer overflow attack may be designed against C code.

Here is a very simple C function that illustrates the principle, taken from an article called "Smashing the Stack for Fun and Profit," written by someone called Aleph One (Phrack 49(14), 1996):

```
void f (char *str) {
char buffer[16];
    ...
    strcpy(buffer,str);
}
```

This function copies a character string, passed by pointer, into the array buffer on the run-time stack. The C function strcpy() copies the contents of *str into buffer[ ] until a null character is found in the string. Because the function does not check the length of the string, careful C programmers would consider the use of strcpy() a coding error. The similar function strncpy(), which has an additional parameter n and copies at most n characters, should be used instead.

If the function f is called from this main program,

```
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    large_string[255]=0;
    f(large_string);
}
```

then the call to strcpy in function f will write over locations on the stack. Because buffer[ ] is 16 bytes long and large_string is 256 bytes, the copy will write 240 bytes beyond the end of the array. This will change the return address of f, which is sitting in the activation record for f a few memory locations away from buffer[ ]. Because large_string contains the character 'A', with hex character value 0x41, the return address of f is set to 0x41414141. Because this address is outside of the process address space, a segmentation fault will occur when function f tries to "return" to location 0x41414141. However, if we change 'A' to a value that is the address of a meaningful instruction, the buffer overflow will cause function f to jump to that location.

Several steps are needed to turn this simple idea into an attack. First, the attacker needs to choose a function that reads data that the attacker can control. Next, the attacker must figure out what data will cause the function to misbehave in a way that is of interest to the attacker. On some computer architectures, an attacker can fill a buffer with code the attacker wants to execute and then cause the function to jump into the buffer. This allows the attacker to execute any instructions.

13.5.2 The Java Sandbox

The JVM executes Java bytecode in a restricted environment called the Java sandbox. The term sandbox is metaphorical – there is no literal data structure or code called the sandbox. The word sandbox is used to indicate that, when bytecode is executed, some operations that can be written in the Java language might not be allowed to proceed, just the way that, when a child plays in a sandbox, an adult supervisor may give the child only those toys that the supervisor considers safe.

The JVM restricts the operation of a Java bytecode program by using four interrelated mechanisms: the class loader, the Java bytecode verifier, run-time checks performed by the JVM, and the actions of the security manager. Although the class loader, bytecode verifier, and virtual machine run-time checks perform general functions related to type correctness and preserving the integrity of the run-time system, the security manager is a specific object designed specifically for security functions. We take a quick look at how each of these mechanisms contributes to Java security. If you are interested in learning more, you may want to read *Securing Java* by Gary McGraw and Ed Felten (Wiley, 1999) or other books on Java security or the JVM.

Class Loader

The class loader, discussed in Subsection 13.4.2, contributes to the Java sandbox in three ways:

- The class-loader architecture separates trusted class libraries from untrusted packages by making it possible to load each with different class loaders.
- The class-loader architecture provides separate name spaces for classes loaded by different class loaders.

- The class loader places code into categories (called *protection domains*) that let the security manager restrict the actions that specific code will be allowed to take.

The Bytecode Verifier and Virtual Machine Run-Time Tests

The Java bytecode verifier checks Java bytecode programs before they are executed, as discussed in Subsection 13.4.3. Complimentary run-time checks, such as array-bounds checks, are performed by the JVM. Together, these checks provide the following guarantees:

- *No Stack Overflow or Underflow:* The verifier examines the way that bytecode manipulates data on the operand stack and guarantees that no method will overflow the operand stack allocated to it.
- *All Methods are Called with Parameters of the Correct Type:* This type-correctness guarantee prevents the kind of type-confusion attacks discussed in Subsection 13.5.3.
- *No Illegal Data Conversions (Casts) Occur:* For example, treating an integer as a pointer is not allowed. This is also essential to prevent the kind of type-confusion attacks discussed in Subsection 13.5.3.
- *Private, Public, Protected, and Default Accesses must be Legal:* In other words, no improper access to restricted classes, interfaces, variables, and methods will be allowed.

The Security Manager

The security manager is a single Java object that answers at run time. The job of the security manager is to keep track of which code is allowed to do which dangerous operations. The security manager does its job by examining the protection domain associated with a class. Each protection domain has two attributes – a signer and a location. The signer is the person or organization that signed the code before it was loaded into the virtual machine. This will be null if the code is not signed by anyone. The location is the URL where the Java classes reside. These attributes are used to determine which operations the code is allowed to perform. A standard security manager will disallow most operations when they are requested by untrusted code and may allow trusted code to do whatever it wants.

A running virtual machine can have only one security manager installed at a time. In addition, once a security manager has been installed, it cannot be uninstalled without restarting the virtual machine. Java-enabled applications such as web browsers install a security manager as part of their initialization, thus locking in the security manager before any potentially untrusted code has a chance to run.

The security manager does not observe the operation of other running bytecode. Instead, the security manager uses the system policy to answer questions about access permissions. It is up to library code that has access to important resources to consult the security manager. In outline, the standard Java library uses the security manager according to the following steps:

- If a Java program makes a call to a potentially dangerous operation in the Java API, the associated Java API code asks the security manager whether the operation should be allowed.
- If the operation is permitted, the call to the security manager returns normally

(without throwing an exception) and the Java API performs the requested operation.

- If the operation is not permitted, the security manager throws a SecurityException. This exception propagates back to the Java program.

One limitation of this mechanism is that a hostile applet or other untrusted code may catch a security exception by using a try-finally block. Therefore, although the security manager may prevent an action by raising an exception, this exception might not be noticed by the user of a system under attack.

13.5.3 Security and Type Safety

Type safety is the cornerstone of Java security, in the sense that all of the sandbox mechanisms rely on type safety. To understand the importance of type safety, we discuss some of the ways that type errors can allow code to perform arbitrary and potentially dangerous actions. We begin with a C example, as C has many type loopholes. It is then shown how similar problems could occur in Java code if it were possible to create two pointers with different types that point to the same object.

One way for a C/C++ program to call an arbitrary function is through a function pointer. For example, suppose a C/C++ program contains lines of the following form:

```
int (*fp)()  /* variable "fp" is a function pointer                   */
...
fp = addr;  /* assign fp an address stored in an integer variable   */
(*fp)(n);   /* call the function at this address                    */
```

Using casts, code like this can call a function at virtually any address. Therefore, if a program may cast integers to function pointers, it is impossible to tell in advance which functions the program calls.

Similar type confusion between object types would also allow a program to perform dangerous actions. Here is an example from *Securing Java* by McGraw and Felten. Suppose that it is possible to create a Java program with two pointers to the same object o, one pointer with type T and one with type U. Let us further suppose that classes T and U are defined as follows:

```
class T {
    SecurityManager x;
}
class U {
    MyObject x;
}
```

where MyObject is some class that we can write in whatever way we want. Here is a program that uses the two pointers to object o to change the security manager, even though the fields of the security manager are declared private and should not be subject to alteration by arbitrary code:

```
T t = (the pointer to object o with type T);
U u = (the pointer to object o with type U);
t.x = System.getSecurity(); // the Security Manager
MyObject m = u.x;
```

The first two lines of this code segment create two references to object `o` with types `T` and `U`, respectively. The next line assigns the security manager to the field `x` of object `o`. This assignment is allowed because the type of field `x` is `SecurityManager`. The final line then gives us a reference `m` to the security manager, where the type of `m` is `MyObject`, a class that we can declare in any way we wish. By changing the fields of `m`, the program can change the private fields of the security manager.

Although this example shows how type confusion can be used to corrupt the security manager, the basic tactic may be used to corrupt almost any part of the running Java system.

An early type-confusion attack on Java was made possible by a buggy class loader and linker. The incorrect system aborted class loading if an exception was raised but did not remove the class name and associated class methods. More specifically, the first step in class loading was to add the class name and constructor to the table of defined references, then load the byte code for the class. If an exception was raised during the second step, the first step was not undone. This made it possible to then successfully load another class with a different interface and to use the first class name for objects of the second class.

13.6 JAVA SUMMARY

The Java programming language and execution environment were designed with portability, reliability, and safety in mind. These goals took higher priority than efficiency, although many engineers and implementers have worked hard to make Java implementations run efficiently.

Java programs are compiled from Java source code (in the Java programming language) to Java bytecode, an interpreted language that is executed on a simple stack-based virtual machine. The run-time environment provides dynamic linking, allowing classes to be loaded into the virtual machine as they are needed. The Java language contains specific concurrency primitives (discussed in Chapter 14), and the virtual machine runs multiple threads simultaneously.

Java is a modern general-purpose object-oriented language that is used in many academic, educational, and commercial projects. We studied the object-oriented features of Java, the Java Virtual Machine and implementation of method lookup, and Java security features.

Objects and Classes

A Java object has fields, which are data members of the object, and methods, which are member functions. All objects are allocated on the run-time heap (not the run-time stack), accessible through pointers, and garbage collected.

Every Java object is an instance of a class. Classes can have static (class) fields that are initialized when the class is loaded into the virtual machine. A constructor of the class is used to create and initalize objects, and objects may have a finalize method that is called when the garbage collector reclaims the space occupied by the object.

Dynamic Lookup

Method lookup in Java is more efficient than in Smalltalk, because the static type system gives the compiler more information, but often is less efficient than in C++. When instructions that access a field or method are executed the first time, there is a search for the appropriate location in an object template or method lookup table. After the first execution of a reference inside a program, the virtual machine optimizes the search for object fields and methods by modifying bytecode. Bytecode instructions that refer to the constant pool are modified to an equivalent so-called *quick* bytecode, which refers to the absolute address of a field or method.

Encapsulation

Java has four visibility distinctions for fields and methods:

- *Public:* accessible anywhere the class is visible.
- *Protected:* accessible to methods of the class and any subclasses as well as to other classes in the same package.
- *Private:* accessible only in the class itself.
- *Package:* accessible only to code in the same package; not visible to subclasses in other packages. Members declared without an access modifier have package visibility.

Names declared in another package can be accessed by import or by qualified names.

Java type checking, applied when source code is compiled and again when bytecode is loaded into the virtual machine, together with run-time tests performed by the bytecode interpreter, guarantees that private fields are truly private and similarly for other visibility levels.

Inheritance

In Java terminology, a *subclass* inherits from its *superclass*. The Java inheritance mechanism is essentially similar to that of Smalltalk, C++, and other class-based object-oriented languages. Java provides single inheritance for classes and multiple inheritance for interfaces.

Java individual methods of a class or an entire class can be declared final. If a method is declared final, then the method cannot be overridden in any subclass. If a class is declared final, then the class cannot have any subclasses. This gives the designer of a class the ability to fix the implementation of a method (or the entire class) by keeping designers of subclass from overriding the method.

In principle, every class declared in a Java program inherits from another class. The class `Object` is the one class that has no superclasses.

Subtyping

Subtyping for classes is determined by the class hierarchy and the interface mechanism. Specifically, if class A extends class B, then the type of A objects is a subtype of the type of B objects. There is no other way for one class to define a subtype of another, and no private base classes (as in C++) to allow inheritance without subtyping.

A class may be declared to implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified in the interface. This (multiple) interface subtyping allows objects to support (multiple) common behaviors without sharing any common implementation.

The Java types are divided into two categories, called primitive types and reference types. The primitive types include boolean and seven numeric types. The three forms of reference types are class types, interface types, and array types. All class, interface, and array types are considered subtypes of Object. Java exception types are subclasses of Throwable, which is a subtype of Object.

Virtual Machine Architecture

The main parts of the Java Virtual Machine are the class loader, the bytecode verifier, the linker, and the bytecode interpreter. A Java source code file is compiled to produce a *class file*. Class files are in a specific format that contains bytecode instructions and some auxiliary data structures, including a symbol table called a *constant pool*. The class loader reads the class file and arranges the bytecode instructions in a form suitable for the verifier. The Java bytecode language contains type information, allowing the verifier to checks that bytecode is type correct before it is executed by the virtual machine. The linker resolves interfile references, and the bytecode interpreter executes the bytecode instructions.

When instructions that access a field or method are executed, there is a search for the appropriate location, based on the symbolic name of the field or method in the constant pool. The JVM implementation optimizes this search by modifying bytecode during program execution. Bytecode instructions that initially refer to the constant pool are modified to an equivalent so-called quick bytecode, which refers to the absolute address after the name has been resolved. Different degrees of optimization are possible for method invocation when the class is known and method invocation when the interface is guaranteed to be the same for all executions of the instruction.

Security

Java security is provided by type-correct compilation and execution and by access control through the security manager. The JVM restricts the operation of a Java bytecode program by using four interrelated mechanisms: the class loader, the Java bytecode verifier, run-time checks performed by the JVM, and the actions of the security manager. Whereas the class loader, bytecode verifier, and virtual machine run-time checks perform general functions related to type correctness and preserving the integrity of the run-time system, the security manager is a specific object designed specifically for security functions. The techniques used to restrict execution of a bytecode program are collectively called the Java sandbox.

Code signing is used to determine which person or company produced a class file. The security manager uses the producer of a class and the URL from which it was obtained to decide whether it can perform potentially hazardous operations such as creating, opening, or writing into a file.

Without type safety, Java programs could be susceptible to the buffer overflow attacks and could carry out the type-confusion attacks we discussed in Section 13.5.

EXERCISES

13.1 Initializing Static Fields

Why do static fields of a class have to be initialized when the class is loaded? Why can't we initialize static fields when the program starts? Give an example of what goes wrong if, instead of static fields being initialized too early, they are initialized too late.

13.2 Java final and finalize

Java has keywords final and finalize.

(a) Describe one situation in which you would want to mark a class final, and another in which you would want a final method but not a final class.

(b) Describe the similarity and differences between Java final and the C++ use of nonvirtual in similar situations.

(c) Why is Java finalize (the other keyword!) a useful feature?

13.3 Subtyping and Exceptions

In Java, a method that can throw an exception (other than from a subclass of Error or RuntimeException) must either catch the exception or specify the types of possible exceptions with a throws clause in the method declaration. For example, a method declaration might have the form

```
public void f(int x) throws Exception1, Exception2
```

meaning that a call to f may either terminate normally or raise one of the listed exceptions (without catching them internally).

Assuming that the type of the method f in B is a subtype of the method f in A, class declarations of the following form are type correct in principle:

```
class A {
  ...
  public Returntype1 f(Argtype1 x)...
}
class B extends A {
  ...
  public Returntype2 f(Argtype2 x)...
}
```

This example of function subtyping Argtype2 → Returntype2 <: Argtype1 → Returntype1 requires Returntype2 <: Returntype1 and Argtype1 <: Argtype2.

Suppose we keep the argument and return types the same, but vary the set of exceptions, as in the following code:

```
class A {
  ...
  public Returntype f(Argtype x) throws Exception1, Exception2, ...
}
class B extends A {
  ...
  public Returntype f(Argtype x) throws Exception1, Exception2, ...
}
```

(a) What relation between the two sets of exceptions is required for the subclass B to be a subtype of class A? Do the sets have to be the same? Or would it be alright for one to be a subset of the other? Explain briefly. In this part, do not worry about subtyping of exception types – we are concerned with only the *sets* of types.

(b) Now suppose that we allow for the possibility that the exceptions in one set could be subtypes of exceptions in the other set. What is the relation we require for class B to be a subtype of class A?

13.4 Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. In contrast, a Java derived class may only have one base class but may implement more than one interface. This question asks you to compare these two language designs.

(a) Draw a C++ class hierarchy with multiple inheritance using the following classes:

Pizza, for a class containing all kinds of pizza,
Meat, for pizza that has meat topping,
Vet, for pizza that has vegetable topping,
Sausage, for pizza that has sausage topping,
Ham, for pizza that has ham topping,
Pineapple, for pizza that has pineapple topping,
Mushroom, for pizza that has mushroom topping,
Hawaiian, for pizza that has ham and pineapple topping.

For simplicity, treat sausage and ham as meats and pineapple and mushroom as vegetables.

(b) If you were to implement these classes in C++ for some kind of pizza-manufacturing robot, what kind of potential conflicts associated with multiple inheritance might you have to resolve?

(c) If you were to represent this hierarchy in Java, which would you define as interfaces and which as classes? Write your answer by carefully redrawing your picture, identifying which are classes and which are interfaces. If your program creates objects of each type, you may need to add some additional classes. Include these in your drawing.

(d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.

13.5 Array Covariance in Java

As discussed in this chapter, Java array types are covariant with respect to the types of array elements (i.e., if B <: A, then B[] <: A[]). This can be useful for creating functions that operate on many types of arrays. For example, the following function takes in an array and swaps the first two elements in the array.

```
1:   public swapper (Object[] swappee){
2:       if (swappee.length > 1){
3:         Object temp = swappee[0];
4:         swappee[0] = swappee[1];
5:         swappee[1] = temp;
6:       }
7:   }
```

This function can be used to swap the first two elements of an array of objects of any type. The function works as is and does not produce any type errors at compile time or run time.

(a) Suppose a is declared by Shape[] a to be an array of shapes, where Shape is some class. Explain why the principle if B <: A, then B[ ] <: A[ ] allows the type checker to accept the call swapper(a) at compile time.

(b) Suppose that Shape[] a is as in part (a). Explain why the call swapper(a) and execution of the body of swapper will not cause a type error or exception at run time.

(c) Java may insert run-time checks to determine that all the objects are of the correct type. What run-time checks are inserted in the compiled code for the swapper function and where? List the line number(s) and the check that occur on that line.

(d) A friend of yours is aghast at the design of Java array subtyping. In his brilliance, he suggests that Java arrays should follow contravariance instead of covariance (i.e., if B <: A, then A[ ] <: B[ ]). He states that this would eliminate the need for run-time type checks. Write three lines or fewer of code that will compile fine under your friend's new type system, but will cause a run-time type error (assuming no run-time type tests accompany his rule). You may assume you have two classes, A and B, that B is a subtype of A, and that B contains a method, foo, not found in A. Here are two declarations that you can assume before your three lines of code:

```
B b[];
A a[] = new A[10];
```

(e) Your friend, now discouraged about his first idea, decides that covariance in Java is all right after all. However, he thinks that he can get rid of the need for run-time type tests through sophisticated compile-time analysis. Explain in a sentence or two why he will not be able to succeed. You may write a few lines of code similar to those in part (d) if it helps you make your point clearly.

13.6 Java Bytecode Analysis

One property of a Java program that is checked by the verifier is that each object must be properly initialized before it is used. This property is fairly difficult to check. One relatively simple part of the analysis, however, is to guarantee that each subclass constructor must call the superclass constructor. The reason for this check is to guarantee that the inherited parts of every object will be initialized properly. If we were designing our own bytecode verifier, there are two ways we might consider designing this check:

(i) The verifier can analyze the bytecode program to make sure that on every execution of a subclass constructor, there is some call to a superclass constructor.
(ii) The verifier can check that the first few bytecode instructions of a subclass constructor contain a call to the superclass constructor, before any loop or jump inside the subclass constructor.

In design (i), the verifier should accept every bytecode program that satisfies this condition and reject every bytecode program that allows some subclass constructor to complete without calling the superclass constructor. In design (ii), some subclass constructors that would be acceptable according to condition (i) will be rejected by the bytecode verifier. However, it may be possible to design the Java source code compiler so that every correct Java source code program is compiled to bytecode that meets the condition described in design (ii).

(a) If you were writing a Java compiler and another person on your team was writing the bytecode verifier, which design would you prefer? Explain briefly.

(b) If you were writing a Java compiler and your manager told you that the standard verifier used design (ii) instead of (i), could you still write a decent compiler? Explain briefly.

(c) If you were writing a bytecode verifier and your manager offered to double your salary if you satisfied design condition (i) instead of (ii) but would fire you if you failed, would you accept the offer? Explain in one sentence.

13.7 Exceptions, Memory Management, and Concurrency

This question asks you to compare properties of exceptions in C++ and Java.

(a) In C++, objects may be reside in the activation records that are deallocated when an exception is thrown. As these activation records are deallocated, all of the destructors of these stack objects are called. Explain why this is a useful language mechanism.

(b) In Java, objects are allocated on the heap instead of the stack. However, an activation record that is deallocated when an exception is raised may contain a pointer to an object. If you were designing Java, would you try to call the finalize methods of objects that are accessible in this way? Why or why not?

(c) Briefly explain one programming situation in which the C++ treatment of objects and exceptions is more convenient than Java and one situation in which Java is more convenient than C++.

(d) In languages that allow programs to contain multiple threads, several threads may be created between the point where an exception handler is established and the point where an exception is thrown. In the spirit of trying to abort any computation that is started between these two points, a programming language might try to abort all such threads when an exception is raised. In other words, there are two possible language designs:

- raising an exception affects only the current thread, or
- raising an exception aborts all threads that were started between the point where the exception handler is established and the point where the exception is thrown.

Which design would be easier to implement? Explain briefly.

(e) Briefly explain one programming situation in which you would like raising an exception to abort all threads that were started between the point where the exception handler is established and the point where the exception is thrown. Can you think of a programming situation in which you would prefer not to have these threads terminated?

13.8 Adding Pointers to Java

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

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

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

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

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

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

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

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

(b) What if A is a primitive type, such as int? Do A* objects seem like pointers to you? [*Hint:* Think about the code in part (a).] Answer in one or two sentences.

(c) If A <: B, should A* <: B*? Answer this question by comparing the type of A*::assign with the type of B*::assign and comparing the type of A*::deref with the type of B*::deref.

(d) If Java had templates, then you could define a pointer template `Ptr`, with `Ptr⟨ A ⟩` defined as `A*` above. One of the issues that arises in adding templates to Java is subtyping for templates. From the `Ptr` example, do you think it is correct to assume that, for every class template `Template`, if `A <: B` then `Template ⟨ A ⟩ <: Template ⟨ B ⟩`? Explain briefly.

13.9 Stack Inspection

One component of the Java security mechanism is called *stack inspection.* This problem asks you some general questions about activation records and the run-time stack then asks about an implementation of stack inspection that is similar to the one used in Netscape 3.0. Some of this problem is based on the book *Securing Java*, by Gary McGraw and Ed Felten (Wiley, 1999).

Parts of this problem ask about the following functions, written in a Java-like pseudocode. In the stack used in this problem, activation records contain the usual data (local variables, arguments, control and access links, etc.) plus a *privilege flag*. The privilege flag is part of our security implementation and will be subsequently discussed. For now, SetPrivilegeFlag() sets the privilege flag for the current activation record:

```
void url.open(string url) {
    int urlType = GetUrlType(url); // Gets the type of URL

    SetPrivilegeFlag();

    if (urlType == LOCAL_FILE)
       file.open(url);
}
void file.open(string filename) {
    if (CheckPrivileges())
    {
       // Open the file
    }
    else
    {
       throw SecurityException;
    }
}
void foo() {
    try {
       url.open("confidential.data");
    } catch (SecurityException) {
       System.out.println("Curses, foiled again!\n");
    }
    // Send file contents to evil competitor corporation
}
```

(a) Assume that the URL confidential.data is indeed of type LOCAL_FILE and that sys.main calls foo(). Fill in the missing data in the following illustration of the activation records on the run-time stack just before the call to CheckPrivileges().

For convenience, ignore the activation records created by calls to GetUrlType() and SetPrivilegeFlag(). (They would have been destroyed by this point anyway.)

	Activation Records		*Closures*	*Compiled Code*
(1)	Principal	SYSTEM		
(2)	Principal	UNTRUSTED		
(3)	control link	(2)		
	access link	(1)		
	url.open	●	⟨( ), ● ⟩	
(4)	control link	(3)		code for url.open
	access link	(3)		
	file.open	●	⟨( ), ● ⟩	
(5)	control link	(4)		code for file.open
	access link	(2)		
	foo	●	⟨( ), ● ⟩	
(6) sys.main	control link	(5)		code for foo
	access link	(1)		
	privilege flag	NOT SET		
(7) foo	control link	( )		
	access link	( )		
	privilege flag			
(8) url.open	control link	( )		
	access link	( )		
	privilege flag			
	url	“ ________ ”		
(9) file.open	control link	( )		
	access link	( )		
	privilege flag			
	url	“ ________ ”		

(b) As part of stack inspection, each activation record is classified as either SYSTEM or UNTRUSTED. Functions that come from system packages are marked SYSTEM. All other functions (including user code and functions coming across the network) are marked UNTRUSTED. UNTRUSTED activation records are not allowed to set the privilege flag.

Effectively, every package has a global variable Principal that indicates whether the package is SYSTEM or UNTRUSTED. Packages that come across the network have this variable set to UNTRUSTED automatically on transfer. Activation records are classified as SYSTEM or UNTRUSTED based on the value of Principal, which is determined according to static scoping rules.

List all activation records (by number) that are marked SYSTEM and list all activation records (by number) that are marked UNTRUSTED.

(c) CheckPrivileges() uses a dynamic scoping approach to decide whether the function corresponding to the current activation record is allowed to perform privileged operations. The algorithm looks at all activation records on the stack, from most recent on up, until one of the following occurs:

- It finds an activation record with the privilege flag set. In this case it returns TRUE.

- It finds an activation record marked `UNTRUSTED`. In this case it returns `FALSE`. (Remember that it is not possible to set the privilege flag of an untrusted activation record.)
- It runs out of activation records to look at. In this case it returns `FALSE`.

What will `CheckPrivileges()` return for the preceding stack [resulting from the call to `foo()` from `sys.main`]? Please answer True or False.

(d) Is there a security problem in this code? (That is, will something undesirable or "evil" occur when this code is run?)

(e) Suppose that `CheckPrivileges()` returned `FALSE` and thus a `SecurityException` was thrown. Which activation records from part (a) will be popped off the stack before the handler is found? List the numbers of the records.