The Java Object Model

Topics in this section include:

- Java *classes*, including the use of access rights, inheritance, method definitions, constructors, and instance versus class members
- Java packages.
- Java *interfaces*, and why are they are so important
- Java *dynamic loading*, and how application classes are loaded on demand

Introduction

A Java program consists of a set of class definitions, optionally grouped into packages. Each class encapsulates state and behavior appropriate to whatever the class models in the real world and may dictate access privileges of its members. In this chapter, you will learn how Java supports the primary features of an object-oriented programming system: encapsulation, data hiding, polymorphism, and inheritance. We assume a knowledge of the object-oriented paradigm.

Classes

Minimally, a class defines a collection of state variables, as well as the functionality for working with these variables. Classes are like C `struct` or Pascal `record` definitions that allow function definitions within them. The syntax for defining a class is straightforward:

```
class ClassName {
  variables
  methods
}
```

For example, an `Employee` class might resemble:

```
class Employee {
  String name;
  ...
}
```

Typically, Java classes are defined one per `.java` file so `Employee` would be in `Employee.java` and compiled to `Employee.class`. When a class references another class, the Java compiler finds the associated class definition to perform type checking and so on. For example, `Employee` references `String`, which the

compiler finds in file `String.class` stored in the standard Java class library.

Variable Definitions

Variable definitions are of the form:

```
modifiers TypeName name;
```

where *modifiers* include access specifiers (discussed in a section below) and `static` if the variable is a class variable.

You may also initialize Java variables to any constant or nonconstant object (except forward references to members, because they will not yet have been initialized):

```
// Java code
class T {
  int i = 0; // instance variable initialization
  static int version = 3; // class variable init
  String id = "x432";     // initialize object variable
}
```

The Java language has a static block initializer for initializing such things as static arrays or for determining initialization values that cannot be calculated until runtime:

```
class TT {
  static int[] a = new int[100];
  static {
    // init all elements to 2
    for (int i=0; i<100; i++) a[i]=2;
  }
}
```

Static variables are initialized and static initializers are executed exactly once--upon class loading.

Method Definitions

In Java, methods must be defined within the class definition. The general syntax for methods is:

```
modifiers ReturnTypeName methodName(argument-list) {
  body
}
```

where *`ReturnTypeName`* can be a primitive type, a class, or an array. The *modifiers* include access specifiers (discussed in a section below) and `static` if

the method is a class method. For example, to return an array of strings, the syntax is:

```
String[] foo() {...}
```

All methods must specify a return type, even if it is `void` (indicating no return value).

Some of the most common methods are so-called getter/setter accessor methods that provide controlled access to the state of an object. For example, consider the following simple employee record definition:

```
class Employee {
  String name = null;

  void setName(String n) {
    name = n;
  }

  String getName() {
    return name;
  }
}
```

The Java language has no special ordering requirements for method definitions. For example, method `setName` could call `getName` as far as Java is concerned as in:

```
void setName(String n) {
  String old = getName();
  if ( !old.equals(n) ) { // same?
    name = n; // only set if different
  }
}
```

Java also allows you to reuse method names (called method overloading). You may reuse any method name in the same class as long as each formal argument list is different. For example,

```
class Employee {
  void giveBonus(StockOptions opt) {...}
  void giveBonus(float cash) {...}
}
```

Constructors

The `Employee` class definition above initializes the `name` instance variable to `null` so that references such as

```
Employee e = new Employee();
```

start out without a useful value for `name`--you must call method `setName` explicitly. How do you set the value of `name` during object construction by passing in a value? By specifying a *constructor* with an argument.

Java constructors are methods that have the same name as the enclosing class and no return type specifier. In general they also are `public`, which we will discuss later. The formal argument list of the constructor must match the actual argument list used in any `new` expression. To be able to execute:

```
Employee e = new Employee("Jim");
```

you must specify a constructor in `Employee`:

```
class Employee {
  String name = null;

  public Employee(String n) {
    name = n;
  }
  ...
}
```

You can overload constructors just like regular methods. For example, extending the `Employee` definition to include salary information might induce you to create a constructor to set both instance variables:

```
class Employee {
  String name = null;
    float salary = 0.0;

  public Employee(String n) {
    name = n;
  }
  public Employee(String n, float sal) {
    name = n;
    salary = sal;
  }
  ...
}
```

If you specify a constructor with arguments it is a good idea to specify a default (sometimes called *copy*) constructor--a constructor with no arguments (one is created for you automatically if you do not specify one). Using expression "`new Employee()`" calls the default constructor:

```
class Employee {
  public Employee() {
    name = "a useful default name";
  }
```

```
  ...
}
```

Class Inheritance

Java supports the single inheritance model, which means that a new Java class can be designed that incorporates, or inherits, state variables and functionality from **one** existing class. The existing class is called the *superclass* and the new class is called the *subclass*. Specify inheritance with the `extends` keyword. For example, to define a manager as it differs from an employee:

you would write the following:

```
class Manager extends Employee {
  ...
}
```

All classes without explicit superclasses are considered direct subclasses of class `Object`. Therefore, all classes directly or indirectly inherit from `Object`.

The `Manager` subclass inherits all of the state and behavior of the `Employee` superclass (except for `private` members of `Employee` as we describe later). Subclasses may also add data or method members:

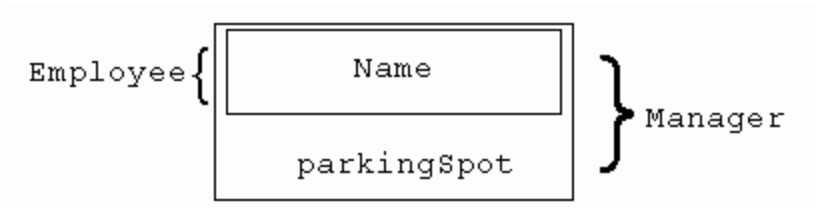
```
class Manager extends Employee {
  int parkingSpot;
  void giveRaiseTo(Employee e) {...}
}
```

To illustrate what a `Manager` object would look like in memory, consider the following `Employee` definition:

```
class Employee {
  String name = null;

  public Employee(String n) {
    name = n;
  }
}
```

A `Manager` object contains both a name which it inherits from `Employee` and a parking spot:



Subclasses may *override* methods defined in the superclass by simply redefining them. For example, to redefine the `getName` method in `Manager`, you could do the following:

```
class Manager {
  ...
  String getName() {
    return "manager " + name;
  }
}
```

The behavior of a `Manager` would now be different than a generic `Employee` with respect to `getName`.

Object Type Equivalence

Recall that class inheritance is an identity relationship in that a subclass is a specialization of the superclass. You can say that *a manager is an employee*. In Java, this implies that an `Employee` reference can refer to any kind/specialization of `Employee`:

```
Employee m = new Manager();
```

The reverse is not true, however. It is not the case that *an employee is a manager*. The compiler will produce an error message if you type in the following:

```
// can't convert Employee to Manager!
Manager m = new Employee();
```

If you use a cast to fool the compiler, the Java virtual machine will cause an exception at runtime. You cannot make Java do something that is not "type safe":

```
// will generate runtime ClassCastException
Manager m = (Manager) new Employee();
```

The flexibility of object-oriented code relies on the type equivalence or "automatic conversion" of references to subclasses. Methods will very commonly define generic formal argument types, but callers will pass specific subclasses as actual arguments. For example, the following method accepts any kind of `Employee`:

```
class Company {
  void promote(Employee e) {...}
  ...
}
```

The callers of method `promote` can pass an `Employee` object, a `Manager` object, or any other object whose type is a subclass of `Employee`. The compiler compares

actual to formal arguments in the light of type equivalence just like the assignment shown above. Hence, passing a `Dog` object, for example, would make the compiler generate an error message about the non-equivalence.

Scope Override Keywords

Inheritance introduces several scoping issues related to member variables and methods. In one of the above examples, `Manager` overrode `Employee`'s `getName` method to change its behavior. What if you wanted to *refine* the behavior instead of replace it totally? You cannot simply call `getName` as you are already in `getName` (an infinite recursion loop would occur). The question more precisely then is how can you access the `getName` method in the super class. Prefix the method call with "`super.`", which informs the compiler you would like the inherited version of `getName` not the `getName` defined in the current class. For example,

```
class Manager {
  ...
  String getName() {
    return "manager " + super.getName();
  }
}
```

The `getName` method in this class now tacks on the string `manager` to whatever the superclass defined `getName` to return.

What if you define a variable member with the same name as a variable inherited from the superclass? How can you access both of them? Use `this.variable` to access the variable in the current class and `super.variable` to access the variable inherited from the superclass.

A more common variation of this problem arises when you define the argument of a method to have the same name as a member variable. Again, the `this` keyword provides a scope override to allow unambiguous references:

```
class Employee {
  String name = null;
  void setName(String name) {
    // member var = argument
    this.name = name;
  }
}
```

Constructors and Inheritance

Constructors are not automatically inherited by subclasses. In other words, if

`Manager` does not specify a constructor, you cannot do the following:

```
Manager m = new Manager("John");
```

even if its superclass, `Employee`, defines a constructor with a `String` argument. You could define a constructor in `Manager` that performs the appropriate initialization, but what if you wanted to define the `Manager` constructor as a refinement of the `Employee` constructor? Use the scope override keyword `super` like a method call. For example,

```
class Employee {
  String name = null;
  public Employee(String n) {
    name = n;
  }
}

class Manager extends Employee {
  int parkingSpot;
  Manager(String n) {
    super(n);  // call constructor Employee(String)
    parkingSpot = -1;
  }
}
```

Note that any call to `super()` must be the *first* statement of the Java constructor. A call to `super()` is inserted by default if one is not specified explicitly.

Also, the first statement may be a call to another constructor within the same class with the help of the `this` keyword. Typically you will write a general constructor and a bunch of small ones that refer to the general constructor. This technique is one way to provide default values for constructor arguments. For example, consider the following definition:

```
public class Circle {
  float area = 0, radius = 0;
  ...
  public Circle(float radius) {
    this.radius = radius;
    area = 3.14159 * radius * radius;
    ...
  }
  public Circle() {
    this(2.0);
  }
  ...
}
```

The first constructor performs certain "standard" initializations, as is common for

many classes. Note that the second constructor (the no-argument constructor) contains a single line of code. In this context, `this()` represents an invocation of another constructor, namely, a constructor whose parameter list matches the invocation arguments. In this case, the invocation matches the first constructor definition. With this functionality, it's easily to provide a variety of constructors with different signatures that indirectly establish default values for state variables.

Order of Initialization and Constructor Calls

When an object of class `T` is instantiated, the following order of events is observed:

1. An implied or explicit call to the superclass constructor of `T` is called, thereby initializing its portion of the object.
2. The member variables of `T` are initialized in the order they are defined.
3. The remainder of the constructor for `T` is executed (the portion after an explicit reference to the superclass constructor).

Using the `Manager` class above as an example, the order of initialization for:

```
Manager m = new Manager("Terence");
```

would be:

1. The constructor for class `Manager` is called.
2. The constructor `Employee("Terence")` is called.
3. The constructor for class `Object` (`Employee`'s superclass) is called.
4. The member variables of `Object` are initialized.
5. The constructor for `Object` is executed.
6. The member variables of `Employee` are initialized.
7. The constructor for `Employee` is executed.
8. The member variables of `Manager` are initialized.
9. The constructor for `Manager` is executed.

Polymorphism (Late Binding)

Object-oriented code is flexible because the implementations of objects are

normally hidden from users of the objects. Also, a single statement can refer to groups of classes (related by inheritance) rather than a single class type. Consequently, the addition of another class in that group should not force changes to any statements referring to that group. For example, the following statement asks an employee to go on vacation:

```
Employee e = some kind of Employee object;
e.goOnVacation();
```

Imagine that this code was originally written when only `Manager` and `Developer` existed as subclasses (kinds of `Employee`s). The code would still work (even without recompiling it) if you added another subclass, say `Marketeer`, because the statement refers generally to any `Employee`.

How is it possible that the `e.goOnVacation()` expression could invoke code that did not exist when it was compiled? More precisely, how could that code invoke, for example, method `Marketeer.goOnVacation` when the compiler was not aware of class `Marketeer`? The answer of course is that method calls are not bound to exact implementations by the compiler--method calls are bound to implementations at run-time.

Method invocation (or *message send*) expressions such as `o.m(args)` are executed as follows:

1. Ask `o` to return its type (class name); call it `T`.
2. Load `T.class` if `T` is not already defined.
3. Ask `T` to find an implementation for method `m`. If `T` does not define an implementation, `T` checks its superclass, and its superclass until an implementation is found.
4. Invoke method `m` with the argument list, `args`, and also pass `o` to the method, which will become the `this` value for method `m`.

Consider two subclasses of `Employee`, `Manager` and `Developer`:

```
class Manager extends Employee {
  void goOnVacation() {
    X
  }
  ...
}

class Developer extends Employee {
  void goOnVacation() {
```

```
    Y
  }
  ...
}
```

To illustrate how the target object (the object receiving the message/method call) type dictates which method implementation to execute, examine the following code snippet:

```
Manager m = new Manager();
Developer d = new Developer();
Employee jim = m;
Employee frank = d;
jim.goOnVacation();   // executes X, this==jim
frank.goOnVacation(); // executes Y, this==frank
```

Variables `jim` and `frank` have type `Employee` as far as the compiler is concerned--you could be referring to any subclass of `Employee`. It's not until run-time, when the exact type of the object is known (that is, which subclass). The exact method can then be determined.

In summary, polymorphism refers to the multiple forms that an `Employee` can take on (its subclasses) and late binding refers to the mechanism for achieving the flexibility.

Access Rights to Class Members

Data hiding is an important feature of object-oriented programming that prevents other programmers (or yourself) from accessing implementation details of a class. In this way, data hiding limits or prevents "change propogation" that arises when users of a class rely on its implementation rather than its interface.

The two most common class member access modifiers are `public` and `private`. Any member modified with `public` implies that any method in any class may access that member. Conversely, any member modified with `private` implies that only methods of the current class may refer to that member (even subclasses are denied access).

In general, it is up to the programmer to determine the appropriate level of visibility, but there are a few reasonable guidelines:

- constructors are `public` so other classes can instantiate them. A class with only `private` constructors cannot usually be instantiated (except via static intializers/methods).
- data members should be `private`, forcing access through public getter/setter

methods like `getName` and `setName` from the `Employee` class above. The state of an object is almost always part of the implementation, which you want to prevent other programmers from relying upon.

While packages are discussed below, for completeness, we discuss the remaining visibility levels. For the moment, consider packages to be logical collections of classes such as the Java AWT package that groups the GUI widgets etc...

Any nonmodified member of a class is visible to any method of any class within the same package, but no methods in classes outside the package may refer to those members. For example, this makes sense that a `Window` class could access the non-private members of the `Button` class if they belonged to the same package.

A `protected` member of a class has the same visibility as a nonmodified member except that all subclasses, even outside the package, may access it.

One final complication and our access picture is complete. Classes may also be modified with `public` (or have no modifier). The `public` classes within a package are visible to classes outside the package, but nonmodified classes are not. We mention public classes versus members here because nonmodified, `protected`, and `public` members are only visible (according to the rules we laid out above) if the enclosing class is `public`. We explore this notion in more detail in the packages section below.

To summarize, in order of least to most protective, here are the Java member access specifiers:

- A `public` member is visible to any other class. Classes outside the package can see the member only if its defining class is `public`.
- A `protected` member is visible to any class within the package and to any subclass even if the subclass is defined in another package as long as the superclass is `public`.
- A member without a specifier is considered *"package friendly"* and can be seen by any class in the package.
- A `private` member is visible only within its defining class.

  Another way to look at visibility within a single package is that a member of a class is visible throughout the package when the member is `public`, `protected`, or unmodified.

Abstract Classes

An abstract class is a class that doesn't provide implementations for all of its declared methods. Abstract classes cannot be instantiated; instead, you must subclass them and provide implementations for any methods that have none.

The Java language has explicit abstract classes, for example,

```
// Java code
abstract class Shape {
  float angle = 0.0;
  public abstract void draw();
  public void rotate(float degrees) {
    angle = degrees;
    draw();
  }
}
```

All subclasses of `Shape` inherit the abstract method `draw` as well as `rotate`, which depends on `draw`:

```
// Java code
class Box extends Shape {
  public void draw() {
    // code to draw box at angle
  }
}
```

Abstract classes cannot be instantiated; you can only inherit from them. Abstract classes typically are used to describe the generic behavior and partial implementation of a set of related classes. For example, all shapes could be rotated by simply changing the angle and redrawing as we show in method `rotate` above.

When you provide no implementation and no data members in an abstract class, that class is probably better reformulated as an *interface*, which specifies only the behavior of an object.

Interfaces

Class inheritance is used to define a new class, `T`, as an extension or specialization of a previous class, `S`. Further, `S` describes `T`'s intrinsic identity thereby implying that all subclasses of `S` are related. For example, if class `Employee` is a generalization of both `Manager` and `President`, then `Manager` and `President` must be related. Any common behavior such as `paySalary` is generalized and defined in `Employee`.

An interface, on the other hand, defines a skill associated with a particular class `T`

that cannot be generalized to all related classes (i.e., other subclasses of `T`'s superclass `S`). We say that interface inheritance groups classes by common behavior and class inheritance groups classes by common identity.

Objects Implement Roles

A convenient way to illustrate the difference between class inheritance and interface inheritance is to consider roles object may take on. The role of "The Terminator" may be described via:

```
interface Terminator {
  public void doRandomViolence();
}
```

This role may be "played" by any number of objects such as:

```
class Arnold extends Actor implements Terminator {
  public void doRandomViolence() {...}
  ...
}
```

and

```
class Dilbert extends Geek implements Terminator {
  public void doRandomViolence() {...}
  ...
}
```

`Arnold` and `Dilbert` have totally different identities (`Actor` versus `Geek`, respectively), but both may play the role of "The Terminator."

A single object may play multiple roles; e.g., `Dilbert` is also able to play the role of an engineer (it is not true that all geeks are engineers so we cannot rely on class `Geek` having the necessary engineer interface):

```
class Dilbert extends Geek
    implements Terminator, Engineer {
  public void doRandomViolence() {...}
  public void borePeople() {...}
}
```

where `Engineer` is defined as:

```
interface Engineer {
  public void borePeople();
}
```

The important concept to grasp here is that interfaces allow you to refer to an object not by its specific class name, but by its behavior. For example,

```
playMovie (Terminator badGuy) {
  badGuy.doRandomViolence();
}
```

An interface allows you to perform similar operations on objects of totally unrelated classes. Unrelated objects may share common behavior by having them implement an `interface`. You could use class inheritance instead, but only if the objects were related.

Interfaces are like classes that have nothing but abstract methods and constants (if any). That is, interfaces have no inherent state or behavior, but rather define a protocol for accessing the behavior of objects that implement them.

Interface Example 1

As a concrete example, consider the interface `Runnable` that indicates a class may be run in a thread. Clearly, the set of classes that can be run in threads are related by a skill rather than identity. For example, a scientific computation and an animation are totally unrelated by identity, but both have the skill that they can be run in a thread.

Interface `Runnable` prescribes that a class define method `run`:

```
interface Runnable {
  public abstract void run();
}
```

The definition of the computation might look like:

```
class ComputePrimes implements Runnable {
  public void run() {
    ...
  }
}
```

And the animation might be:

```
class Animation implements Runnable {
  public void run() {
    ...
  }
}
```

Note that we could define a subclass of `Animation` (a class related by identity) that would also be `Runnable`:

```
class CartoonAnimation extends Animation
                       implements Runnable {
```

```
  ...
}
```

Interface Example 2

Java provides a wonderful mechanism for separating the abstract notion of a collection of data from a data structure implementation. During the design process, collections of data should be thought of in the abstract such as `List`, `Set`, `Queue`, etc.

It is only at the programming stage that actual data structure implementations should be chosen for each collection. Naturally, appropriate data structures are chosen according to memory constraints and how the collections will be accessed (e.g., how fast can data be stored, retrieved, sorted, etc.). In Java, we separate the abstract notion of a collection of data from its data structure implementation using interfaces. For example, consider a list interface versus a linked list implementation. A simple `List` might have the following behavior:

```
interface List {
  public void add(Object o);
  public boolean includes(Object o);
  public Object elementAt(int index);
  public int length();
}
```

whereas a linked list (that can actually play the role of a `List` collection) would be a class that implements interface `List`:

```
class LList implements List {
  public void add(Object o) {...}
  public boolean includes(Object o) {...}
  public Object elementAt(int index) {...}
  public int length() {...}
}
```

Such a list could be used as follows:

```
List list = new LList();
list.add("Frank");
list.add("Zappa");
list.add(new Integer(4));
if ( list.includes("Frank") ) {
  System.out.println("contains Frank");
}
```

The key observation here is that a `LList` implementation was created, but was accessed via the collection interface `List`.

Interface Example 3

Consider the predefined Java interface `Enumeration`:

```
interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

Any data structure can return an `Enumeration` of its elements, which should be viewed as a particular "perspective" of the data stored within. To walk any enumeration, the following generic method can be used:

```
void walk(Enumeration e) {
  while ( e.hasMoreElements() ) {
    System.out.println(e.nextElement());
  }
}
```

The `java.util.Vector` class has a method called `elements` that returns an object (of class `VectorEnumerator`) that implements the `Enumeration` behavior. For example:

```
Vector v = ...;
walk(v.elements());     // sends in a VectorEnumerator
```

This same method, `walk`, will work for any `Enumeration` of any data structure. For example, `Hashtable` and `Dictionary` also can return an `Enumeration`:

```
Hashtable h = ... ;
walk(h.elements());
Dictionary d = ... ;
walk(d.elements());
```

Interface Benefits

In summary, Java interfaces provide several benefits that allow you to streamline your programming:

- Interfaces allow you to work with objects by using the interface behavior that they implement, rather than by their class definition.
- You can send the same message to two unrelated objects that implement the same interface. In C++, the two objects must be related (i.e., inherit from the same parent). The Java language thus provides a great framework for working with distributed objects, even when you do not have the class definition for the target objects!

Interface Implementation Versus Class Inheritance

Interface implementation specifies that a class must explicitly define the methods of an interface and, hence, define a portion of the object's behavior. Class inheritance is used to inherit the state and behavior of the superclass. You may consider interface implementation to be a restricted subset of class inheritance.

Interface implementation is, in fact, a subset of class inheritance in terms of functionality, except for one important distinction: interface implementation forms another completely independent hierarchy among your classes. In the Java language, grouping by behavior specification is clearly distinct from the grouping of related objects.

Interfaces are like classes that have nothing but abstract methods and constants (if any). That is, interfaces have no inherent state or behavior, but rather define a protocol for accessing the behavior of objects that implement them.

Packages

Java has a *packages* feature (similar to Ada) that lets you group together related classes and decide which classes will be visible to classes in other packages. For example, consider the following two incomplete Java compilation units, one called `List.java`:

```
// Define a new package called LinkedList
package LinkedList;

public class List {...}
  // class is visible outside of package

class ListElement {...}
  // class is not visible outside of package
```

and the other called `A.java`:

```
// bring in all classes from LinkedList.
import LinkedList.*;

// define class A in the default, unnamed package
public class A {...}
```

that implement and use a linked list.

The important points to note in this example are:

- Class `A` in file `A.java` must inform the compiler that it will be using symbols from

package `LinkedList`.

- Class `List` is visible to classes outside package `LinkedList`. For example, class `A` is able to make `List` objects.

- We have hidden the wrapper for list elements, class `ListElement`, within package `LinkedList`. Class `A` may not access `ListElement` because class `A` is in a different package (specifically, the default unnamed package).

  We have demonstrated class visibility and now consider member visibility as we flesh out `List.java`:

```
// Define a new package called LinkedList
package LinkedList;

// public class List is visible outside of package
public class List {

  public void add() {...}
    // visible outside of class and package

  void internal() {...}
    // not visible outside of class
}

// class ListElement is not visible outside of package
class ListElement {

  public void getNext() {...}
    // getNext() is not visible out of package

}
```

  and `A.java`:

```
// bring in all classes from LinkedList
import LinkedList.*;

public class A {
  // define class A in the default package.

  public static void main(String[] args) {
    List b = new List(); // access to type List OK;
                         // LinkedList.List visible
    // ok, add() is public in List
    b.add();
```

```
        // INVALID! internal() not public
        b.internal();

        // INVALID!
        // LinkedList.ListElement not visible

        ListElement c = new ListElement();

        // INVALID! fields of ListElement
        // also not visible
        c.getNext();
    }
}
```

Package Hierarchy and Directory Hierarchy

The Java runtime system normally insists that the directory structure of your class library match that of the language package hierarchy. For example, class

```
LinkedList.List
```

must exist in file

```
LinkedList/List.class
```

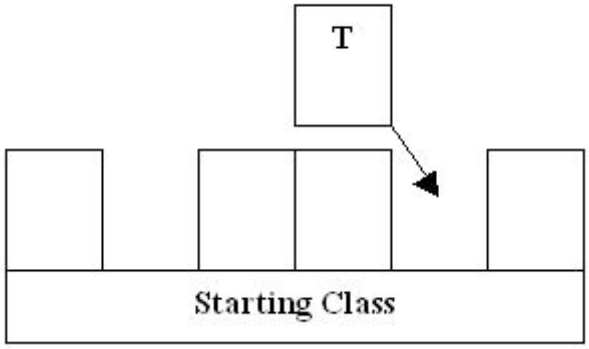
starting from one of the directories listed in your `CLASSPATH` environment variable. Your `CLASSPATH` should include `'.'`, the current directory.

Class files in the default package should reside in the current directory or in one of the directories specified in your `CLASSPATH` environment variable.

Dynamic Loading

A Java program is not linked together all at once. The runtime environment first loads only the starting class, followed by other classes as they are needed. You may think of a Java application as a program made up of nothing but DLLs (dynamically loadable libraries).

The first time a class, say `T`, is instantiated, the Java interpreter will recognize that the `T.class` file has not been loaded. The class loader will search for and load the class definition and byte code for the class's methods. (Your `CLASSPATH` environment variable determines where the runtime system looks for class files.)



Benefits of Dynamic Loading

The main benefits of dynamic loading are:

- Your program loads only the code it needs to run.
- You grab the latest version of each class at runtime, not at link time. Note that this can be a two-edged sword; you may not really want the latest version...
- Java applications can be distributed. They can combine "services" from several sources to form a complete application so you do not end up with a large, hard-to-change (and possibly out-of-date) application.
- It's easier to work in teams-you do not have to relink your entire project after you (or someone else on your team) change a module.

Classes Are Conscious

Java classes are "conscious" in the sense that they know their type at runtime, whereas C `struct`s and Pascal records are blocks of "unconscious" data--a block of memory. For example, if you call a function `f` on some data for which it was not designed to operate, something unexpected will happen at best or it will crash the program. In java, calling a method of a class that has been modified to no longer implement `f`, results in the runtime exception `NoSuchMethodException`.

The class of an object can be determined at runtime via the `instanceof` operator, which returns a `boolean` value, for example,

```
boolean isButton(Object o) {
  return (o instanceof Button);
}
```

The Relationship between C Structures and Java

Classes

[*This section is intended as review for C++ programmers and as a bridge from C to Java for C programmers.*]

Consider how you would build a list manager for arbitrary elements in C. First, you would define operations on a list such as `add` and `get`:

```
void list_add(void *item) {..}
void *list_get(int i) {..}
```

Second, you might choose the implementation data structure such as an array of "void *":

```
static void *data[MAX_ELEMENTS];
static int nelements = 0;
```

To ensure that users of the list manager do not access the data structure explicitly (isolating implementation from interface), we have made the data static, thus making it invisible outside of the file.

Naturally, the problem with this scheme is that only one list manager can be created. To solve this, you would put the data within a struct and then list manager users would each allocate one of the structs:

```
struct List {
  void *data[MAX_ELEMENTS];
  int nelements;
};

void list_add(struct List *list, void *item) {..}
void *list_get(struct List *list, int i) {..}
void list_init(struct List *) {nelements=0;}
```

Notice that now each list function receives the particular list to operate on as the first argument. Also note that an explicit initialization function is required to initially set the fields of a `List` structure.

Unfortunately, this struct-based scheme opens up all of the data elements to the users as they must see the definition of struct `List` to be able to use it.

A Java Implementation

A similar, but simpler and safer, solution to the struct-based scheme, is available to Java programmers: Change the name `struct` to `class`. Place the functions within the class definition. Finally, remove the first argument that indicates which

list to manipulate.

```
class List {
  private  Object data[];
  private  int nelements=0;
  public   void add(Object item) {..}
  public   Object get(int i) {..}
  public   List() {
      data = new Object[MAX_ELEMENTS];
  }
}
```

Notice that we have also changed:

- `void *` to `Object`, which refers to *any* Java object. (But cannot refer to a variable of a primitive type.)
- `list_add()` to `add()` because moving it inside the class definition for `List` means that it is now really `List.add()`! Function names are more general in Java.
- `list_init()` to `List()`, which is called the constructor for class `List`. Because Java cannot allocate any space with a simple field (member) definition, we use the constructor to allocate the space. We can, however, initialize `nelements` with the definition.
- access specifiers have been added to prevent access to `List`'s private data.

Using the List Managers

Now compare how you would use the C list manager versus the Java list manager:

```
/* C code */
#include "List.h"
foo() {
  struct List *mylist = (struct List *)
  calloc(1, sizeof(struct List));
  list_add(mylist, "element 1");
  list_add(mylist, "element 2");
  /* we can write to mylist->data[] directly! */
}
```

In Java, you would write:

```
// Java code
void foo() {
  List mylist = new List();
  mylist.add("element 1");
  mylist.add("element 2");
  // cannot write to mylist.data[] directly!
}
```

The crucial difference between C and Java is in the function call versus method invocation; i.e.,

```
list_add(mylist, "element 1");
```

versus

```
mylist.add("element 1");
```

In C you write function-centric code. In Java, you write data-centric code. That is, "execute this code using this data" versus "here's some data...operate on it". Object-oriented programmers typically go further than this to say "ask the list to add "element 1" to itself." Another common terminology (from Smalltalk) would describe this operation as "send the message add to the object mylist with an argument of "element 1"."

Are Classes Only For Object-Oriented Programming?

The Java version of the list manager described above can be viewed merely as a simpler description than the C version. In fact, classes that are not derived from another class (using inheritance) are really nothing more than fancy struct definitions. All the mumbo-jumbo surrounding object-oriented programming such as polymorphism can be ignored while studying the benefits of encapsulation and data hiding:

- **encapsulation**: the methods that operate on an object are packaged up with that object.
- **data hiding**: portions of the object's data may be hidden from the user, thus, allowing the implementation of that object to be isolated from the user. Changes to the implementation would not affect a user's code as all access to the data element can be controlled through a set of functions.

[*MML: 0.995a*]
[*Version: $Id: //depot/main/src/edu/modules/JavaObjectModel/javaObjectModel.mml#2 $*]