This lecture is about dynamic dispatch: how a call o.m() may actually invoke the code of different methods, all with the same name m, depending on the runtime type of the receiver object o.

To explain how this happens, we show how one class can be defined as a subclass of another, and can override some of its methods. This is called subclassing or inheritance, and it is a central feature of object oriented languages. At the same time, it’s arguably a rather dangerous and overused feature, for reasons that we’ll discuss later when we look at subtyping.

The notion of dynamic dispatch is broader than the notion of inheritance. You can write a Java program with no inheritance in which dynamic dispatch still occurs. This is because Java has built-in support for specifications. You can declare an object to satisfy a kind of partial specification called an interface, and you can provide many implementations of the same interface. This is actually a more important and fundamental notion than subclassing, and it will be discussed in our lecture on specification.

1 Bank Transaction Code

Here is the code for a class representing a bank account:

```java
class Account {
    String name;
    Vector transv;
    int balance;
    Account (String n) {
        transv = new Vector ();
        balance = 0;
        name = n;
    }
    boolean checkTrans (Trans t) {
        return (balance + t.amount >= 0);
    }
    void post (Trans t) {
        transv.add (t);
        balance += t.amount;
    }
}
```

and a class representing a transaction:

```java
class Trans {
    int amount;
}
Date date;
...
}

2 Extending a Class by Inheritance

Suppose we want to implement a new kind of account that allows overdrafts. We might call it AccountPlus, and code it like this:

class AccountPlus extends Account {
    int creditLimit;
    AccountPlus (String n, int c) {
        super (n);
        creditLimit = c;
    }
    boolean checkTrans (Trans t) {
        return (balance + creditLimit + t.amount >= 0);
    }
}

The keyword extends indicates that the implementation of AccountPlus extends the implementation of Account by adding some new features. AccountPlus is said to inherit features from Account; AccountPlus is a subclass of Account, and Account is a superclass of AccountPlus.

There is a new field, creditLimit. Because a new AccountPlus object needs to have the new field initialized, AccountPlus must have its own constructor; this actually calls the constructor of Account (see Java text for details of this slightly strange syntax). All the other methods and fields of Account are implicitly present in AccountPlus.

The method checkTrans appears again in AccountPlus, with different code in its body. This is called overriding. When the code acc.checkTrans is executed, which method actually gets called will depend on whether the object referenced by acc is an Account object or an AccountPlus object. The method call is said to be dynamically resolved.

At runtime, each object has a type, equal to the class whose constructor created it. A variable that appears in the code also has a type, given by its declaration at compile-time. At runtime, a variable can refer to an object whose type is not the variable’s type; it is sufficient that the object type be the type of a subclass of the variable type. (For now, by the way, we’re using the term ‘type’ to mean classification by class name, to distinguish it from the term ‘class’ which usually carries the connotation of the code in the class too. Later in the course, we’ll be more precise about what type means.)

Sometimes, it will be clear in the code what type an object will have at runtime:

AccountPlus acc = new AccountPlus ("Zork", 100);
Trans t = new Trans (100, new Date ());
if (acc.checkTrans (t))
    acc.post (t);

In this case, since acc is declared to be of type AccountPlus, and AccountPlus has no subclasses, we know that the method of AccountPlus will be called. But it’s not in general the type declaration in the program text that determines which method gets called. Suppose we wrote this instead:
Account acc = new AccountPlus ("Zork", 100);
Trans t = new Trans (100, new Date ());
if (acc.checkTrans (t))
    acc.post (t);

where the variable acc is declared in the first line above to have the type Account rather than
the type AccountPlus. What happens? The code executes exactly as before. What determines
which checkTrans method gets called is the runtime type of acc – that is, the type of the class
that provided the constructor used to create it. In general, how variables are declared has no effect
whatsoever on the behavior of the program, if it executes successfully without class cast errors
(more on that below).

Now suppose we want to handle a collection of accounts. We might have a Bank class, implement­
ment something like this:

1. class Bank {
2.     Account [] accounts;
3.     ...
4.     void chargeMonthlyFee () {
5.         for (int i = 0; i < accounts.length; i++) {
6.             Trans fee = new Trans (-1, new Date ());
7.             if (accounts[i].checkTrans (fee))
8.                 accounts[i].post (fee);
9.         }
10.    }

A Bank object holds an array of Account objects. An array is an object just like a Vector, but
it can’t grow or shrink dynamically.

Look at the method chargeMonthlyFee used for charging monthly fees to accounts. This bank
is unusual: it doesn’t hit you when you’re down. If deducting the monthly fee would take you
below your limit, it won’t do it.

The method works whether the accounts in the array are regular accounts (in the Account
class), or special accounts (in the AccountPlus class). The reason is that the declared type given
in the code says only that the object at runtime will belong to that class or one of its subclasses.
But at runtime, which checkTrans method is selected for the call at Statement 6 will depend on
the runtime type of the object.

This code is said to be polymorphic, meaning ‘many shapes’, since the same piece of code text
can handle different types of account. If the accounts array contains two objects of the class
Account, and a third object of class AccountPlus, the first and second time round the loop the
call to the method checkTrans will execute code from Account, but the third time round, it will
execute code from AccountPlus. The call to post will always call the same code, since this method
has only one body, although sometimes it will be called for an Account object, and sometimes an
AccountPlus object.

3 A Template Method

Instead of making the client of the Account class call the checkTrans method, we could call it
inside the post method like this:
boolean post (Trans t) {
    if (!checkTrans (t)) return false;
    transv.addElement (t);
    balance += t.amount;
    return true;
}

Look at the context this method sits in:

class Account {
    boolean post (Trans t) {...}
    boolean checkTrans (Trans t) {...}
}

class AccountPlus extends Account {
    boolean checkTrans (Trans t) {...}
}

Now suppose we have some code that calls post on an object of AccountPlus:

Account a = new AccountPlus ("Zork", 100);
a.post (new Trans (-50, new Date ()));
System.out.println (a.balance);

Which checkTrans method gets called inside post? If the method from Account is called, it will return False, ignoring the credit limit, and the print statement will print 0 as the balance. If the method from AccountPlus is called, it will return true, the posting will occur, and the balance will print as -50.

The answer depends on the runtime type of the receiver. Although post belongs to the class Account, since there is no post method in AccountPlus, its code will be called for both Account and AccountPlus objects. Executing acc.post when acc is an AccountPlus object will cause the post method of Account to be executed; inside it, the checkTrans method of AccountPlus, and not Account, will be called. So although the post method only appears in the code once, it actually behaves differently for AccountPlus and Account objects.

This idiom is often used in implementations of ‘frameworks’. A framework supplies a collection of classes that the programmer tailors to her own purpose by extension – by adding new subclasses. The superclass may have a method that defines the skeleton of an algorithm, but actually leaves most of the computation to methods that it calls that are defined in subclasses, by the programmer who extends the framework. Such a method is called a template: it lets the programmer redefine steps of an algorithm without changing its overall structure.

4 Downcasting

Arrays aren’t very convenient to program with, since they can’t grow or shrink. Suppose we implement Bank with a vector of accounts instead:

// bad code!
1. class Bank {

2. Vector accounts;

3.     void chargeMonthlyFee () {
4.         for (int i = 0; i < accounts.size(); i++) {
5.             Trans fee = new Trans (-1, new Date ());
6.             if (accounts.elementAt (i).checkTrans (fee))
7.                 accounts.elementAt (i).post (fee);
8.         }
9.     }

The class Vector is provided as part of the standard Java library. Unlike arrays, vectors are not part of the language itself. So there’s no special syntax to access a vector element: you have to call a method (here, elementAt (i) to get the ith element). Also, when you declare a Vector, you can’t say what it’s a vector of. The elementAt method has this signature:

class Vector {
    ...
    Object elementAt (int i)
    ...
}

It returns an object of class Object, the superclass of all classes. So there’s no way to know that the expression accounts.elementAt(i) will actually evaluate to an Account or an AccountPlus object. If it fails to, the calls on Statements 6 and 7 in the Bank class code to checktrans and post will be made to objects without these methods defined. Java is a safe language, which means that certain kinds of runtime errors cannot occur, and calling a non-existent method is one of them. For this reason, the code above will actually be rejected by the Java compiler.

Instead we have to write this:

void chargeMonthlyFee () {
    for (int i = 0; i { accounts.size(); i++) {
        Trans fee = new Trans (-1, new Date ());
        if (((Account) accounts.elementAt (i)).checkTrans (fee)) {
            ((Account) accounts.elementAt (i)).post (fee);
        }
    }
}

or better:

1.     void chargeMonthlyFee () {
2.         for (int i = 0; i < accounts.size(); i++) {
3.             Trans fee = new Trans (-1, new Date ());
4.             Account acc = (Account) accounts.elementAt (i);
5.             if (acc.checkTrans (fee)) {
6.                 acc.post (fee);
7.             }
8.         }
9.     }
The (Account) on Statement 4 is called a downcast. At runtime, it checks that the object returned by the expression belongs to Account or one of its subclasses. If it does, execution continues normally; if it does not, the program is terminated with a ClassCastException. (We'll talk about exceptions in a later lecture.)

For now, it's important just to understand that if execution continues at the next line, the object bound to acc is guaranteed to be of class Account or AccountPlus, and must therefore have the post method. So the Java compiler will accept this code, since the presence of the downcast ensures that there will be no attempt to call a method that does not exist.

Students are often confused about downcasts, and think that some kind of conversion is taking place. This is not true. The downcast is simply a test; no change to the object occurs.

5 Downcasts are not Typecasts

Typecasts are a different matter. Executing this code

```java
1. double d = 1.23;
2. int i = (int) d;
3. System.out.println (d);
4. System.out.println (i);
```

causes the following to be printed

```
1.23
1
```

The phrase (int) in Statement 2 is a typecast or coercion; it ensures the type safety of the program by actually converting the double created at Statement 1 to an integer, so that d and i have different values. No such thing happens with a downcast; if the statement

```java
Account acc = (Account) accounts.elementAt (i);
```

completes successfully, the object referenced by acc after the statement is the same object, unmodified, as the object returned by the expression on the right-hand side.

6 Type Hierarchy and Safety

Types can be arranged in a hierarchy. Here is such a hierarchy showing some of the types we have discussed:
All these types correspond to classes. The root of the tree, `Object` is a superclass, directly or indirectly, of every other class. You can see that `Vector` is actually positioned quite deep in the tree: its code is built by inheritance from the classes `AbstractCollection` and `AbstractList` which provide skeletal implementations of collections and lists respectively.

Not every type is a class, though. Java has specification types, called `interfaces`, that do not correspond to executable code. An interface is just a collection of method signatures. A class that satisfies the specification of an interface is said to `implement` it; this is indicated in the text of the class by the keyword `implements`. A variable can be declared to have an interface type, and interfaces thus contribute to the type hierarchy. Here is a fragment of the type hierarchy that shows some interfaces implemented by `Vector`:

```
Object
    \|-- AbstractCollection
        \|-- AbstractList
            \|-- Collection
                \|-- AbstractSet
                    \|-- List
                        \|-- Vector
                    \|-- Set
                        \|-- HashSet
```

The interface names are italicized to distinguish them from the names of classes.

Because the runtime type of an object is given by the constructor that created it, and because interfaces have no code, it follows that the runtime type of an object is always a class. The declared type of a variable can be a class or an interface. We’ll say that a type (interface or class) `T` is a subtype of a type `T'` if there is a path going up in the type hierarchy from `T` to `T'`. The edges in the path may be `extends` or `implements` edges.

Given this background, we can now state the key type safety property of Java. Java is said to be a `statically typed` language. What this means is that the types that appear in declarations in the program text tell you something about what will happen when the program runs:

**Static typing:** If a variable of (declared) type `T` holds a reference to an object of (runtime) type `T'`, then `T'` is a subtype of `T`.

And we can now explain downcasts like this. In the assignment

```
T x = e;
```
the expression e must evaluate to an object that is a subtype of T, otherwise this guarantee cannot be maintained. So if the compiler is unable to determine that this is true, we must insert a downcast like this

\[
T \ x = (T) \ e;
\]

so that now the test performed by the downcast guarantees the typing property. If the cast fails (that is, e evaluates to an object of the wrong type), the assignment is aborted; if it succeeds, the expression e must have evaluated to an object of an appropriate type – that is, a subtype of T.

7 Conclusion

We have distinguished between the declared type of a variable, and the constructed type of an object, and we have seen how the code for a method is chosen according to the constructed type of the receiver object. In polymorphic code, this type cannot be predicted at compile time, and a call that appears syntactically once in the code may cause different methods to be invoked at runtime. For this reason, the dispatching mechanism is said to be ‘dynamic’.

We’ve seen how downcasts allow polymorphic code to be type checked at compile time, by introducing runtime tests when the compiler cannot statically determine the type that an expression will evaluate to an runtime. We noted how downcasts are just tests, and unlike typecasts, have no side effects.