

Subclassing and Dynamic Dispatch

6.170 Lecture 3

Spring 2004

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:

```
class Account {
    String name;
    ArrayList transactions;
    int balance;
    Account (String n) {
        name = n;
        transactions = new ArrayList ();
        balance = 0;
    }
    boolean isAllowed (Trans t) {
        return (balance + t.amount >= 0);
    }
    void post (Trans t) {
        transactions.add (t);
        balance += t.amount;
    }
}
```

and a class representing a transaction:

```
class Trans {
    int amount;
```

```

    Date date;
    Trans (int a, Date d) {
        amount = a;
        date = d;
    }
}

```

2 Extending a Class by Inheritance

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

```

class OverdraftAccount extends Account {
    int creditLimit;
    OverdraftAccount (String n, int cl) {
        super (n);
        creditLimit = cl;
    }
    void bumpUp (int amount) {
        creditLimit += amount;
    }
    boolean isAllowed (Trans t) {
        return (balance + creditLimit + t.amount >= 0);
    }
}

```

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

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

The method `isAllowed` appears again in `OverdraftAccount`, with different code in its body. This is called *overriding*. When the code `acc.isAllowed` is executed, which method actually gets called will depend on whether the object referenced by `acc` is an `Account` object or an `OverdraftAccount` 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:

```

OverdraftAccount oda = new OverdraftAccount ("Zork", 100);
Trans t = new Trans (-50, new Date ());
if (oda.isAllowed (t))
    oda.post (t);

```

In this case, since `oda` is declared to be of type `OverdraftAccount`, and `OverdraftAccount` has no subclasses, we know that the method of `OverdraftAccount` 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 a = new OverdraftAccount ("Zork", 100);
Trans t = new Trans (-50, new Date ());
if (a.isAllowed (t))
    a.post (t);
```

where the variable `a` is declared in the first line above to have the type `Account` rather than the type `OverdraftAccount`. What happens? The code executes exactly as before. What determines which `isAllowed` method gets called is the *runtime* type of the object that `a` refers to — 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 at runtime. But the declared type of a variable *does* affect how the Java compiler checks your code for errors, a process called *static type-checking* (*static* meaning that it happens at compile time, not run time). Consider this code:

```
1.     Account a = new OverdraftAccount ("Zork", 100);
2.     a.bumpUp (400); // BAD CODE! Java compiler rejects it
```

The Java compiler refuses to compile this code, because of the method call in Statement 2. The method `bumpUp` is found in `OverdraftAccount`, but not in `Account`. Although a human reader can tell from looking at the code that `a` is guaranteed to point to an `OverdraftAccount` object, the Java compiler isn't smart enough to draw that kind of inference. Instead, the compiler looks only at the declared type of `a`, `Account`, to decide what methods and fields can be used with `a`. Since `bumpUp` is not available in `Account`, the compiler produces an error.

3 Polymorphism

Now suppose the bank wants to charge a monthly fee to all its accounts. We might write a method (in some other class), implemented something like this:

```
1.     void chargeMonthlyFee (Account a) {
2.         Trans fee = new Trans (-1, new Date ());
3.         if (a.isAllowed (fee))
4.             a.post (fee);
5.         else
6.             sendAngryLetterTo (a.name);
7.     }
```

This code works whether the account passed in is a regular account (in the `Account` class), or a special account (in the `OverdraftAccount` 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 `isAllowed` method is selected for the call at Statement 6 will depend on the runtime type of the object.

```
1.     Account a1 = new Account("Deeb");
2.     Account a2 = new OverdraftAccount("Zork", 100);
3.     chargeFee(a1); // uses Account's isAllowed()
4.     chargeFee(a2); // uses OverdraftAccount's isAllowed()
```

The `chargeFee` method is said to be *polymorphic*, meaning ‘many shapes’, since the same piece of code text can handle different types of account. When the a parameter is passed an object of the class `Account`, the call to the method `isAllowed` will execute code from `Account`. When a is passed an object of class `OverdraftAccount`, it will execute code from `OverdraftAccount`. 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 `OverdraftAccount` object.

4 A Template Method

The protocol we’re using to add transactions to an `Account` — first checking whether the transaction `isAllowed` before doing a `post` — is somewhat clunky and error-prone. Clunky because it’s repetitive; every client that has a transaction to post needs to make this extra check. Error-prone, because we’re relying on myriad places in the code to guarantee an important property of accounts, that the balance can never go below 0 (or the credit limit, in the case of an `OverdraftAccount`). If some client forgets to call `isAllowed` before posting a transaction, this property may no longer be satisfied, and the bank will lose money. This kind of property is called an *invariant*, and we’ll see much more about them in later lectures.

Instead of making the client of the `Account` class call the `isAllowed` method, we could call it inside the `post` method like this:

```
boolean post (Trans t) {
    if (!isAllowed (t)) return false;
    transactions.addElement (t);
    balance += t.amount;
    return true;
}
```

Look at the context this method sits in:

```
class Account {
    boolean post (Trans t) {...}
    boolean isAllowed (Trans t) {...}
}
...
class OverdraftAccount extends Account {
    boolean isAllowed (Trans t) {...}
}
```

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

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

Which `isAllowed` 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 `OverdraftAccount` 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 `OverdraftAccount`, its code will be called for both `Account` and `OverdraftAccount`

objects. Executing `acc.post` when `acc` is an `OverdraftAccount` object will cause the `post` method of `Account` to be executed; inside it, the `isAllowed` method of `OverdraftAccount`, and not `Account`, will be called. So although the `post` method only appears in the code once, it actually behaves differently for `OverdraftAccount` 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.

5 Downcasting

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

```
1.     class Bank {
2.         ArrayList accounts;
3.         ...
4.         void chargeMonthlyFee () {
5.             for (int i = 0; i < accounts.size(); i++) {
6.                 Trans fee = new Trans (-1, new Date ());
7.                 accounts.elementAt (i).post (fee); // BAD CODE! compile error
8.             }
9.         }
10.        ...
11.    }
```

Look at the method `chargeMonthlyFee`, which is used for charging monthly fees to accounts. The method works by looping through the elements of the `accounts` list and attempting to deduct a \$1 fee from each account. (Notice that this bank is unusual: it doesn't hit you when you're down. If deducting the monthly fee would take you below your limit, causing `post` to return `false`, then the bank will just forget the fee.)

Unfortunately, there's a problem in Statement 6. This statement calls `elementAt (i)` to get the *i*th element of `accounts`. The `elementAt` method has this signature:

```
class ArrayList {
    ...
    Object elementAt (int i)
    ...
}
```

It returns an object of class `Object`, the superclass of all classes. So there's no way for the compiler to know whether the expression `accounts.elementAt(i)` will actually evaluate to an `Account` or an `OverdraftAccount` object. If it fails to, the call to `post` in Statement 6 will be made to an object without this method 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 be rejected by the Java compiler, just like it rejected calls to the `bumpUp` method using an `Account` variable.

Instead we have to write this:

```
void chargeMonthlyFee () {
    for (int i = 0; i < accounts.size(); i++) {
        Trans fee = new Trans (-1, new Date ());
        ((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 a = (Account) accounts.elementAt (i);
5.         a.post (fee);
6.     }
7. }
```

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 `a` is guaranteed to be of class `Account` or `OverdraftAccount`, 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.

If you want to avoid risking a `ClassCastException`, you can make the runtime test yourself using the `instanceof` operator:

```
1. if (accounts.elementAt(i) instanceof Account) {
2.     Account a = (Account) accounts.elementAt(i);
3.     a.post (fee);
4. } else
5.     sendAngryLetterTo ("The Programmer");
```

The expression in Statement 1 tests whether the *i*th object in `accounts` belongs to `Account` or one of its subclasses. (Note that part carefully: *or one of its subclasses*. Like downcasting, `instanceof` doesn't just test for membership in `Account`, but also permits `OverdraftAccount` objects or any other subclass of `Account`.) Unlike a downcast, however, `instanceof` simply returns false when the test fails, rather than throwing an exception. In this case, if the test fails, we'll send a complaint to the programmer who allowed a non-`Account` object to sneak into the `accounts` list.

You'll find `instanceof` used most often in implementations of the `equals` method, which has to return false when it's passed an object of the wrong class. In most other cases, however, you shouldn't use it. Good object-oriented code strives to *avoid* `instanceof`.

6 Downcasts are not Typecasts

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.

Typecasts are a different matter. Executing this code

```
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

```
Account a = (Account) accounts.elementAt (i);
```

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

Another common misconception is to think that downcasts can alter the effect of dynamic dispatch. It does no such thing; which method is called depends only on an object's runtime type, which is unaffected by a downcast. This code

```
1.    Object x = "deeb";
2.    System.out.println (x);
3.    String s = (String) x;
4.    System.out.println (s);
```

prints the same string "deeb" twice. The downcast at Statement 3 has no effect. If the code compiles — which it does — we can see from Statement 2 that `System.out.println` accepts an argument of type `Object`, so there is no need to downcast its argument to `String`. And when `System.out.println` runs, its code cannot tell whether a downcast has occurred.

Of course to obtain a string representation of an object, some special code will need to be executed. This code is the method `toString`, which is defined in the `Object` class, and thus implicitly in every other class. This `toString` method, however, simply generates a string representation of the address of the object. It is good practice always to override `toString` in a user-defined class with a method that converts the object to a string that's more useful. Having done this, there's no way to recover the effect of `Object`'s version of the `toString` method; casting to `(Object)`, for example, is simply nonsense and has no effect.

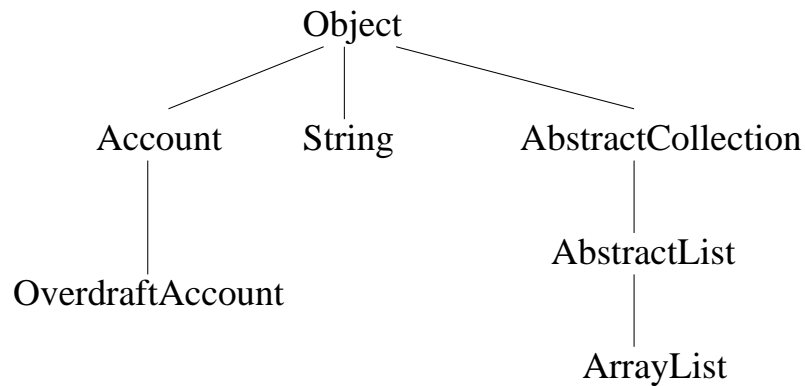
One more confusion worth noting is the relationship between `int` and `Integer`. The type `int` is a *primitive* type; the value of a variable of this type is an integer. The type `Integer` is an *object* type; the value of a variable of this type is a reference to an object — albeit one that represents an integer. These are used differently. The type `int` is used for local variables and integers in arrays; the type `Integer` is used whenever an object is required (such as for the elements of a `ArrayList`).

You can't cast between `int` and `Integer`. To create an `Integer`, you use a constructor, and to extract the primitive integer from an integer object, you call a method, as described in Lecture 2:

```
int i = 5;
Integer obj_i = new Integer (i);
int j = obj_i.intValue();
```

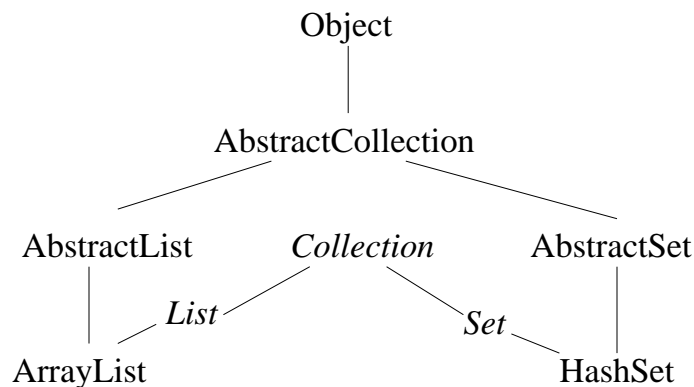
7 Type Hierarchy and Safety

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 `ArrayList` 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 `ArrayList`:



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 .

8 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 at runtime. We noted how downcasts are just tests, and unlike typecasts, have no side effects.