Object-Oriented Design — **Lecture 4**
CSU 370 Fall 2007 (Pucella) — **Tuesday, Sep 18, 2007**

The Java Type System

By now, you have seen a fair amount of Java. Time to study in more depth the foundations of the language, because this will become important as we study more advanced object-oriented design concepts.

Programs essentially manipulate *values*. Values are the entities in the programming language that can be assigned to variables and passed as arguments to methods. In Java, every value belongs to one of several possible *types*. (Conversely, you can think of types as sets of values.) There are four important kinds of types.[1] These are:

- **primitive types**: types corresponding to primitive values, including `int`, `short`, `long`, `byte`, `char`, `float`, `double`, and `boolean`.
- **null type**: the type of the value `null`
- **array types**: types of arrays of values
- **class types**: types of objects

Primitive types are the types of primitive values such as integers, floating point numbers, characters, and booleans. The four types of integers differ in how big an integer they allow you to express. (See the textbook in Chapter 1 for a table of the exact range of values permitted by each of these types.) Similarly, the two types of floating point numbers `float` and `double` differ in the range of floating point numbers they let you express. Java will implicit let you treat a small integer as a larger integer without complaining. In particular, it will let you store a `short` in a variable declared `long`, because every short integer fits into the space reserved for a large integer. Similarly, you can pass a `byte` to a method expecting an `int`. However, if the kind of conversion you attempt to do has the potential to lose information, such as storing a `long` integer into a variable expecting a `short` integer, you have to explicitly *cast* the integer to the shorter type, like this:

```
short foo;
long bar = 1000;
foo = (long) bar;
```

[1]Actually, there is a fifth kind of type, interface types, but we will not see those until later.

Casting will come up later under a different guise. Intuitively, casting is the way Java lets you say "please convert a value of this type into a value of this other type." If the cast does not make sense, Java will complain with a runtime error (for instance, converting a `boolean` value into an `int`.) Another situation where casting is useful is converting a `float` value to an `int`, by truncating the decimal part. (This truncation can lose information, and so a cast is required.)

Array Types

An array is a data structure that allows constant-time access to the elements in the array. Arrays have a fixed size.

An array of values of type `T` has type `T[]`. Any type `T` can be used. A new array of size 10, containing values of type `T[]` is created using `new T[10]`. If `a` is an array of size n, the elements of the arrays can be accessed using `a[0]`, ..., `a[n-1]`. (Note that indexing starts with 0.)

Initially, an array is created with all elements initialized to either 0 (in case of integer or float arrays), `false` (in case of Boolean arrays), or `null` (in case of object arrays). To specify an initialized array, one can use the special syntax, such as:

```
int[] a = {0,1,1,2,3,5,8,13}
```

This creates an array of size 8, initialized with the supplied values.

Arrays are in some sense objects, and they do have fields. A useful field is `length`, which returns the length of the array. Thus, in the above example, `a.length` evaluates to 8.

A useful operation on arrays is iterating over all the elements of the array. A `for` loop can be used for that. (We will see other ways of iterating later in the course.) There are two flavors of `for` loops that can be used. The first is C-like. Suppose that `sa` is a `String` array, initialized with some strings, and suppose that we wanted to print out all the strings in the array, one per line.

```
for (int i = 0; i<sa.length; i++)
  System.out.println sa[i];
```

The `for` loop specifies three things: a declaration for the variable that will hold the current index when iterating through the array, along with an initial value (here, `int i = 0`), a test for when to continue iterating (here, we continue iterating as long as as `i < sa.length`, that is, as long as the index has not reached the end of the array), and how to update the index at the end of every iteration (here, we increment the index by one, `i++`).

A cleaner way to express this `for` loop, one that does not require us to worry about indices into the array (which always opens the door to either going past the end of the array, or stopping short before the array ends) is to use the second form, called a `for-each` loop:

```
for (String s : sa)
  System.out.println s;
```

The idea is that variable `s`, of type `String`, repeatedly gets bound to each element of array `sa` in turn.

Class Types and Subclassing

Class types are the types of objects. Consider the following class:

```
public class Person {
  private String name;
  private String nuid;
  public Person (String n, String id) {
    this.name = n;
    this.nuid = id;
  }
  public String getName () {
    return this.name;
  }
  public String getNUId () {
    return this.nuid;
  }
}
```

An object of this class has type `Person`. Class types are just types, and therefore we can create, say, arrays of objects; an array of `Person`s, then, has type `Person[]`, as expected.

The most interesting aspects of classes is that it is possible to define new classes that extend a given class. These classes are called *subclasses*, and the class that they extend is called their *superclass*. A subclass is defined as follows:

```
public class Student extends Person {
  public void registerForCourse (String course) {
      // some implementation for registration
  }
}
```

This subclass extends the `Person` class with a single method for registering for courses. (Note the `void` indicating that the method does not actually return a result; `void` is not a type, it is an annotation on the method.)

Another subclass would be:

```
public class Faculty extends Person {
  public void makeAppointmentWithDean (String time) {
   // some implementation for appointment making
  }
}
```

The main feature of subclassing is that a *subclass provides all the public methods and public fields of its superclass*. That's the main thing Java cares about when doing type checking. It knows, therefore, when it manipulates an object of type `Student`, that that object has a method `getName` that can be invoked.

For those of you that follow at home, let me emphasize that subclassing is *not* inheritance. We will see inheritance later in the course. Of course, subclassing and inheritance are related. As we will see inheritance is a code reuse mechanism that lets you reuse code easily when defining subclasses. But subclassing makes sense even when you do not have inheritance. (Indeed, some languages have subclassing but no inheritance, at least, not inheritance like Java implements.) Subclassing is a property of classes, and is properly part of the type system of Java. Subclassing is used by Java to determine what methods can possibly be invoked on an object, and to return an error at compile-time when an object does not supply a given method.

Suppose there is a method expecting an argument of type `Person`:

```
public void doSomething (Person p) {
  // do something to p
}
```

then because of subclassing it is possible to pass it an object of type `Student`, that is,

```
  // suppose obj is an object of the class defining method doSomething
  Student s = new Student ();
  obj.doSomething (s);
```

During execution of the method, you will only be able to access the methods of the object specified in the `Person` class. That's because the only thing Java knows statically is that argument `p` in the body of the method has type `Person`.[2] However, Java, at runtime, actually knows that the object passed to the method and bound to `p` is an object of the `Student` class, and it is possible to recover that information.

Downcasting is an operation that takes an object and attempts to view it as an object of another class. This is only possible if the object actually is an object of that class. (If it is not, then an exception is thrown—an error is reported.) If the cast succeeds, then the object is viewed as an object of the cast class. Thus, in the above method, we could do this:

[2]Why would it be unsafe to allow the method to access, say, methods defined in the `Student` class?

```
public void doSomething (Person p) {
  Student stu = (Student) p;
  stu.registerForCourse ("CSU 370");
}
```

Of course, if you invoke `doSomething` with either a `Person` object or a `Faculty` object, an error is reported. Happily, there is a way to check whether an object can be safely viewed as an object of another class.

```
public void doSomething (Person p) {
  Student stu;
  if (p instanceof Student) {
    stu = (Student) p;
    stu.registerForCourse ("CSU 370");
  }
}
```

The operation *object* `instanceof` *class* returns true if *object* is an object of the given *class*, and false otherwise.

Subtyping

Subclassing is a relation between classes, and it tells you when you can pass an object of a given class to a method of another class, or assign an object of a given class to a variable of another class.

This relation between classes induces a *relation between types*, called subtyping. It generalizes subclassing as follows. Suppose that v is a value of type `S` (not necessarily an object, perhaps an array), that `S` is a subtype of `t`, and method `m` expects an argument of type `T`, then it is safe to invoke method `m` with v. (By safe, I mean simply that you will not attempt to apply an operation to the value that is undefined for that value.)

We say `S` is a subtype of `T` if one of the following conditions holds:

(1) `S` and `T` are the same type

(2) `S` and `T` are both class types, and `S` is a direct or indirect subclass of `T`

(3) `S` and `T` are both array types, where `S`=`S'[]` and `T`=`T'[]` and `S`' is a subtype of `T`'

(4) `S` is not a primitive type and `T` is the type `Object`

(5) `S` is the null type, and `T` is not a primitive type

(We will add more rules to this when we see interface types.)

Rule 1 says that subtyping is a reflexive relation. Rule 2 tells us, for instance, that `Student` is a subtype of `Person`. Rule 3 tells us, for instance, that `Student[]` is a subtype of `Person[]`. Rule 4 tells us, for instance, that `int[]` is a subtype of `Object`. (More on the `Object` type in the next lecture.)