Java Class Types

Class Types

We noted earlier that the `String` type in Java was *not* a primitive (or array) type. It is what Java calls a *class-based* (or *class*) type—this is the third category of types in Java. Class types, are based on *class declarations*. Class declarations allow us to create *objects* that can hold more complex collections of data, along with operations for accessing/modifying that data. For example

```
public class Student {
        private int age;
        private String name;
        private char grade;

        public Student(int theAge, String theName, char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
        }

        public int getAge() { return age;}

        public String getName() { return name;}

        public char getGrade() { return grade;}

        public void setAge(int newAge) { age = newAge;}

        public void setGrade(char grade) { this.grade = grade;}
}
```

The above, grotesquely oversimplified, class declaration specifies the data components (*instance variables*) of a *Student* object, along with a method (called a *constructor*[*]) describing how to create such objects, and several short methods for accessing/modifying the data components (*fields/instance variables*) of a `Student` object. Given such a class declaration, one can write programs that declare (and create) variables of type Student:

```
Student a;
Student b;
a = new Student(18, "Patti Smith", 'A');
b = new Student(20, "Joan Jett", 'B');
```

Or, combining the declaration of the variables and creation (*instantiation*) of the corresponding values (objects):

```
Student a = new Student(18, "Patti Smith", 'A');
Student b = new Student(20, "Joan Jett", 'B');
```

The words `public` and `private` are called *access level modifiers*; they control the extent to which other classes can create, access, or modify objects, their fields, and their methods. Declaring the class `Student` to be `public` allows other classes to create objects of type `Student`. Likewise, declaring the methods `getName, setGrade`, etc., to

[*] Note: The constructor uses the same name as the class

be `public` allows other classes to invoke these methods on any `Student` objects they have created. Declaring the instance variables (`name, age, grade`) to be `private` blocks other classes from directly accessing or modifying those variables. An almost universal rule of thumb in object-oriented design is to allow instance variables to be accessed/modified *only* through the use of class methods. This helps guarantee that the underlying state of an object can't be compromised by users and that the underlying implementation of a class can be changed if needed without compromising the functionality of programs that use the class.

The ability to create new data types through class declarations allows for the construction of robust and reusable code modules and supports the development of larger bodies of code. Class types can be used very much like primitive types: Variables of any class type can be created, passed to (and returned from) methods, used as types of instance variables for even more complex classes, and so on. One can create arrays of variables of any class type:

```
// Create an array to store 3 objects of type Student
Student[] class = new Student[3];

// Create the three Student objects and store them in the array
class[0] = new Student(18, "Patti Smith", 'A');
class[1] = new Student(20, "Joan Jett", 'B');
class[2] = new Student(20, "David Bowie", 'A');
```

Note the use of `new` here, both for the creation of the individual student objects and for the creation of the array. Array and class-based types have more complex *storage requirements* for their values than do primitive types; in Java, that storage is allocated by using the keyword `new` followed by an invocation of the array or class constructor. The two exceptions to this norm are

- The allocation of an array by explicitly listing its values: `int[] scores = {97, 85, 100};`,
- The creation of a String using a String *literal* [†]: `String name = "Zeta";`.

Strings are unique in Java among class-based types in that String values can be specified by String literals; the only other types whose values can be specified by literals are the primitive types.
Strings and arrays in Java have a very similar flavor, but there are some key differences; among them are:

- The i^{th} element of an array x is referenced with syntax $x[i]$; the i^{th} character in a String x is referenced with syntax $x.substring(i, i+1)$.
- Strings are *immutable*: one cannot assign a value to an individual position in a String variable; rather a new String can be constructed by piecing together (*concatenating*) other Strings.
- To get the size (length) of a String x, use $x.length()$ (i.e., invoke the `length` method of the `String` class); to get the size of an array x, use $x.length$ (i.e., access the `length` instance variable of the array x).

The Structure of a Java Program

A Java program consists of a set of class declarations; each class declaration typically describes a type of object that can be created and includes any object data (*instance variables*) and functionality (*class methods*). An executing program consists of a sequence of statements that declare and construct objects and then invoke the methods of the objects in order to access or modify them in some way. These statements are woven together with other statements that control the flow of program execution ("if" statements, looping constructs, and so on). Java itself is not a large language; the set of keywords and symbols in the language is modest. What makes the language powerful and flexible is the ability to add functionality by designing new class types.
Java is designed to be run in many different environments, from stand-alone code on a computer to embedded systems on a wide range of devices. The method for executing Java code that we will focus on is the use of a special method, (always) named *main* that we can include in a Java class declaration. Here's a simple Java program:

[†]A literal is an explicit representation of a value in Java source code, such as `21, 3.14159, 'C', true, "Hi there!"`. The only other literal for class types is `null`, which can be assigned to any variable of non-primitive type.

```
import Student;

public class StudentDemo {

        pubic static void main(String[] args) {
                Student a = new Student(18, "Patti Smith", 'A');
                Student b = new Student(20, "Joan Jett", 'B');

                if( a.grade() == b.grade() )
                        System.out.println("Grades match");
                else
                        System.out.println("Grades don't match");
        }
}
```

The program above consists entirely of a `main` method. The method itself is pretty dull, it merely compares the grades of the two student objects and prints an appropriate message. The first line of the `main` method, called the method *signature*, always has the form `public static void main(String [] args)`[‡]. We'll talk more about the meanings of the keywords *public, static, void*, but, essentially, they indicate that

public the method can be invoked by users of the Student class

static the method can be invoked (called) without reference to a particular object of type Student; that is, the method can be called with the syntax: `Student.main(x)`, where `x` is an array of Strings

void the method does not return a value

Using any text editor, we can create a file that contains the class declaration above, giving the file the name `StudentDemo.java` (always use the name of the class as the name of the file). We *compile* the program (convert it into Java bytecode) by typing `javac StudentDemo.java` in a terminal window. We can then execute the bytecode of the `main` method of the program by typing `java StudentDemo`.
The `StudentDemo` class also includes an `import` statement. This statement ensures that the `Student` class is available for use in the `StudentDemo` class. Here's another example of a class that It's worth noting that we can define a class that consists *only* of a `main` method. For example,

```
public class MathCalcs {

        import java.lang.Math;

        pubic static void main(String[] args) {
                double x = Math.pow(E,PI); // e^pi
                double y = Math.pow(PI,E)/ // pi^e
                if (x > y)      System.out.println("e^pi is greater than pi^e");
                else if (y > x) System.out.println("pi^e is greater than e^pi");
                else            System.out.println("e^pi equals pi^e");
        }
}
```

Note that we import the `Math` class from the `java.lang` *package*. A package is a collection of classes that have been bundled together to provide a family of services. The `java.lang` package is part of the standard Java distribution.

Enumeration Types

Before exploring the properties and uses of class types, let's briefly look at a fourth category of types provided by Java: *enumeration types*. Enumeration types are used to provide families of *named constants*. For example, we could create named constants for the four suits and 13 ranks of a deck of playing cards as follows;

[‡]Well, the name `args` can be replaced by any other legal variable name....

```
public enum Suits { CLUBS, DIAMONDS, HEARTS, SPADES }

public enum Ranks {TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
                JACK, QUEEN, KING, ACE }
```

We could then use these names to create a `Card` class:

```
public class Card {

        private Suit s;
        private Rank r;

        public Card(Suit s, Rank r) {
                this.s = s;
                this.r = r;
        }

        public Suit getSuit() { return s; }
        public Rank getRank() { return r; }
        public void setSuit(Suit s) { this.s = s; }
        public void setRank(Rank r) { this.r = r; }

}
```

This code could then be used to create a deck of cards:

```
Card[] deck = new Card[52];

int i = 0;
for ( Suit s : Suit.values() )
        for ( Rank r : Rank.values() ) {
                Card[i] = new Card( s, r );
                i++;
```

The file BasicCard.java contains an implementation of a simple card class like the one above. Note that the keyword `public` is missing from the two `enum` declarations. This is because a single file can only contain one public class declaration. Which leads us to note that calling enumeration types a "fourth category" is somewhat misleading. An enumeration type is just a special kind of class type that allows us to define classes having fixed numbers of possible values and that can be iterated over using the `for-each` loop construct in Java. In the following more thorough versions of our playing card example, we put each enum declaration in its own file, where we can declare them `public`. More on this later.