COMP 110

Prasun Dewan[1]

11. More Types

So far, we have looked at a number of predefined primitive types defined by Java, including int, double, byte, short, long, and float. In this section, we study three new predefined types. One of these types, char, is a primitive type, while the remaining two are object types.

char

The computer must often process characters. Most programs accept character input and produce character output. In fact, some programs, such as a spelling checker, perform most of their computations in terms of characters. Characters are the building blocks for the strings we have seen before.

This type defines a variety of characters including the English letters (both lowercase, a..z, and uppercase, A..Z), the decimal digits 0..9; "whitespace" characters such as blank and tab; separators such as comma, semicolon, and newline; and other characters on our keyboard. A character can be represented in a program by enclosing its print representation in single quotes:

```
'A'   'Z'  '1'   '0'  ' '
```

Two consecutive single-quotes denote a special character called the *null* character:

```
''
```

The null character is used to mark the end of a string. It is not useful to print it since Java prints nothing for it.

How do we represent the single-quote character itself? We could enclose it in single-quotes:

```
'''
```

However, Java would match the first two single-quotes as the null character, and think you have an extra single-quote character. So, instead, it defines the following representation for a single-quote:

```
'\''
```

[1] © Copyright Prasun Dewan, 2000.

Table 1. Some Useful Java Escape Sequences

Escape Sequence	Character Denoted
\’	’
\n	new line
`	back space
\\	\
\t	tab
\”	”

Here, instead of enclosing one character within quotes, we have enclosed a two-character escape *sequence*. The first of these characters, \, or *backslash*, is an *escape* character here, telling Java to escape from its normal rules and process the next character in a special way.

Java defines escape sequences to represent either those characters that cannot use the normal representation or those for which the normal representation may not be readable. A literal cannot have a new line character in it, so \n denotes the new line character. A backslash after the first quote denotes special processing, not the backlash character itself, so \\ denotes the backlash character. Typing a backspace after a single-quote would erase the single-quote, so \b denotes the backspace character. We can represent the tab character by entering a tab between quotes:

‵   ‵

but this representation can be mistaken as the space character. So \t denotes a tab character. Similarly, we can represent the double quote character as:

‵″′

but it may be mistaken for two null characters. So \” denotes the double quote character. Table 1 summarizes our discussion.

Java allocates 16 bits for storing a character. As a result, it can support as many as 2^{16} different characters, which is useful since we would like to represent characters of all current languages; and some of them such as Chinese have a large character set. It stores a non-negative integer code for each character. As programmers, we do not have to concern ourselves with the exact integer assigned to each character. However, as discussed later, we need to know something about the relative order in certain subsets of the character set such as the set of lower case letters and the set of digits.

Ordering Characters

Clearly, it makes sense to order Java values that are numbers, but why order characters? In ordinary life, we do order characters, when we learn the alphabet, and more important, when we search directories. It is to support such searches that programming languages order the elements in the character set. The integer code, or *ordinal number*, assigned to a character is its position in this set. We do not need to know the exact ordinal number assigned to a character. It is sufficient to know that:

- The null character, ‘’, is assigned the ordinal number 0.
- The digits are in order.
- The uppercase letters, ‘A’ .. ‘Z’, are in order.
- The lowercase letters, ‘a’ .. ‘z’, are in order.

Letters of other alphabets are in order.

Thus, we know that:

‘1’ > ‘0’	→	true
‘B’ > ‘A’	→	true
‘a’ > ‘b’	→	false
c >= ‘’	→	true

where c is a character variable holding an arbitrary character value.

Based on the information above, we cannot compare elements from different ordered lists. Thus, we cannot say whether:

‘A’ > ‘a’
‘A’ > ‘0’

Converting between Characters and their Ordinal Numbers

Like other programming languages, Java lets you find out the exact ordinal number of a character by casting it as an `int`. Thus:

```
System.out.println ( (int) 'B')
```

will print out the ordinal number of ‘B’. This cast is always safe, since `char` (16 unsigned bits) is a narrower type than `int` (32 bits). Therefore, when context demands `int`s, Java automatically performs the cast. Thus:

```
int i = 'A';
```

and:

```
'B' - 'A'
```

computes the difference between the integer codes of the two characters, returning 1. Java lets you directly perform all the `int` arithmetic operations on characters, and uses their ordinal numbers in the operations. Usually we do not look at absolute values or sums of ordinal number - the differences are more important, as we see below.

You can also convert an ordinal number to a character:

```
char c = (char) intCodeOfB;
```

We had to perform a cast because not all integers are ordinal numbers of characters, just as not all doubles are integers. For instance, the following assignment makes no sense:

```
char c = (char) -1
```

since ordinal numbers are non-negative values. Java simply truncates the 32 bit signed value into a 16 unsigned value, much as it truncates a `double` with a fraction part to an `int` without a fraction. Again, by explicitly casting the value you are telling Java that you know what you are doing and are accepting the consequences of the truncation.

The two-way conversion between characters and ordinal numbers can be very useful. For instance, we can find the predecessor or successor of characters:

```
(char) ('I' - 1)   ==    'H'
(char) ('I' + 1)   ==    'J'
```

We can also convert between uppercase and lower case characters, which is useful, for instance, when processing languages in which case does not matter:

```
(char) ('i' - 'a' + 'A')      ==     'I'
(char) ('I' - 'A' + 'a')      ==     'i'
```

To understand why the above equalities hold, consider an equality we know his true based on the fact that letters are ordered:

```
'i' - 'a'    ==     'I' - 'A'
```

Moving the 'a' to the right, we get:

```
'i'    ==     'I' - 'A' + 'a'
```

Moving the 'A' to the left we get:

```
'i' - 'a' + 'A'    ==     'I'
```

Printing Different Types

The `println` method can be used to print values of arbitrary types:

```
System.out.println(2)                output:     2
System.out.println(2.0)            output:     2.0
System.out.println((int) 2.0)      output:     2
System.out.println('2')            output:     2
System.out.println((int) '2')      output:     50
System.out.println((char) 51)      output:     3
System.out.println(5 > 0)          output:     true
```

`println` is an overloaded method, and different implementations of it are used for the different types of values, much as + is an overloaded operator, and different implementations of it are used to add `int` and `double` values.

Notice the use of the explicit cast:

```
(int) '2'
```

Contrast this with some of the previous that also converted chars to their integer codes:

```
int i = 'A'
 'B' - 'A'
```

In the previous examples, Java automatically does the cast for us since otherwise the program fragments would not be legal. It cannot do so in the `println` case, since a `println` without the cast is legal, as shown above. `println` is an overloaded procedure defined for both character and integer arguments. The cast explicitly indicates which definition should be used.

Strings

To better understand strings, let us return to our discussion of structured objects. Recall these are objects that can be decomposed into simpler components. For example, an instance of `ALoanPair` is a structured object that can be decomposed into the simpler components, carLoan and houseLoan of type `Loan`. We saw two kinds of components: logical components or properties, which are visible outside the class of the object; and physical components or instance variables, which should be visible only inside the class.

`String` is also a structured object: each character in a `String` instance is a logical component of it. How should these components be accessed from outside the class? Like `ABMISpreadsheet`, `String` could define a separate getter method for each of its components:

```
public char getFirstChar();
public char getSecondChar();
…
```

A problem with this approach is that it is not clear when we should stop, that is, how many such getter methods we should define? The number of components defined by the class `String` is *variable*, that is, different instances of it can have different number of components. For instance, "h" has one component, while "hello world" has 11 components. We did not face this problem in the definition of `ALoanPair`, since all instances of this types have the same number of (logical) components.

The solution is based on the observation that we can identify string components through, not their names, but their positions or *indices* in the string. Instead of making the component identifier part of the name of its getter method, `String` makes it a parameter of the getter method. Thus, `String` does not provide different getter methods for different components but a single getter method that takes the component index as an argument and returns a character:

```
public char charAt(int);
```

To be consistent with the naming scheme for getter methods of fixed structures, it would have been better to call this method, `getCharAt`, but the class `String`, like several other predefined Java classes we will see later, was defined before this scheme was established.

`String` indices start from 0 rather than 1. Thus, if:

```
String s = "hello world"
s.charAt(0) == 'h';
s.charAt(1) == 'e'
```

In general, we access the ith character of string, `s`, as:

```
s.charAt(i-1);
```

Not all string indices are legal. An index that is smaller (greater) than the index of its first (last) character is illegal. Thus, both of the following accesses will raise a `StringIndexBounds` exception:

```
s.charAt(11)
s.charAt(-1)
```

The instance function, `length`, returns the number of characters in a string. Thus:

```
"helloworld".length() == 11
"".length() ==    0
```

We can use this function to define the range of legal indices of an arbitrary string `s`:

```
0 .. (s.length() - 1)
```

Sub-Strings

Besides individual characters, we may also wish to retrieve *sub-strings* of a string, that is, sequences of consecutive characters that appear in the string. The Java function:

```
public String substring (int beginIndex, int endIndex)
```

when invoked on a `String`, `s`, returns a new string that consists of the character sequence starting at `beginIndex` and ending at `endIndex - 1`, that is:

```
s.charAt(beginIndex)  ..  s.charAt(endIndex - 1)
```

It raises the `StringIndexOutOfBounds` exception if `beginIndex` is greater than `endIndex`. If they are both equal, it returns the empty string. Thus:

```
"hello world".substring(4, 7) → "o w"
"hello world".substring(4, 4) → ""
"hello world".substring(7, 4) throws StringIndexOutOfBounds
```

While `String` provides getter methods to read string characters and sub strings, it provides no setter method. This is because Java strings are read-only or *immutable*, that is, they cannot change. A separate class, `StringBuffer`, which we will not study in this course, defines mutable strings. The class `String` does, as we have seen before, provide the + operation on strings to create *new* strings from existing strings. Thus:

```
"hello" + "world" == "hello world"
```

Here, we do not change either string; instead, we create a new string that stores the result of appending the second string to the first one.

`String` provides several other useful methods. For example, it provides methods `toUpperCase` and `toLowerCase`, which can be invoked on a string to return another string that has the same letters as the first one, except that they are all in upper case and lower case, respectively. Thus:

```
"Hello World".toLowerCase() == "hello world"
```

We have omitted several other methods provided by `String`. We will introduce them as we need them.

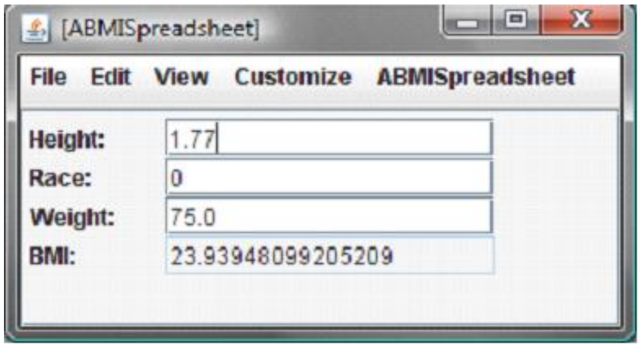
Variable-Size Type vs Variable-Size Instance

A particular string, like all the other objects we have seen so far, has a fixed structure, that is, its size (and, in fact, contents) is fixed during its lifetime. In contrast, as we have seen above, the type String defines a variable structure, that is, different instances of it can have different sizes.

Thus, types can define fixed or variable structure; and a type with a variable structure can have instances with fixed or variable structures. All the primitive types and some of the object types we have seen have fixed structure. `String` (and array types we will study in the next section) define variable structures, but their instances have fixed structure. Later, we will see types with variable structure whose instances also have variable structure. We will refer to these three kinds of types as *fixed*, *variable*, and *dynamic* types, respectively.

Capturing Choices

Often a variable or property must represent a group of alternative choices. For example, we might want to add a property to `ABMISpreadsheet` that represents the race of the person, thereby allowing us to see the relationship between race and bmi values. It is possible to encode choices using the int type, where each choice is mapped to a different int value. For example, the six official races could be mapped to int values between 0 and 5, as shown below:



Height: 1.77
Race: 0
Weight: 75.0
BMI: 23.9394809205209

Figure 2. BMISpreadsheet with Race property of type int

```
public static int AFRICAN_AMERICAN = 0;
public static int AMERICAN_INDIAN = 1;
public static int ASIAN = 2;
public static int NATIVE_HAWAIIAN = 4;
public static int WHITE = 5;
public static int SOME_OTHER_RACE = 6;
```

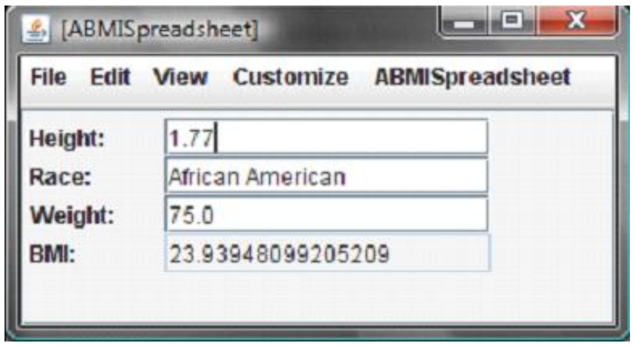
A variable/property that represents one of the choices is simply assigned/set to the int representing the race, as shown below:

```
int race = AFRICAN_AMERICAN;
public int getRace() {
      return race;
}
public void setRace (int newVal) {
      race = newVal;
}
```

Here the variable `race` stores one of the race choices, and is therefore declared to be of type int. The getter and setter directly export this variable as an int property.

Figure 1 shows a display of an instance of the new ABMISpreadsheet class with the additional property. There are two problems with using the int type to represent a set of choices. Perhaps the most obvious problem is that the end-users who need to enter the choices must know the mapping between a race and its int encoding. For instance, in Figure 1, users must know that 0 stands for the race African American. A more subtle and important problem is that it is possible for programmers to make errors in the encoding. For example, we might erroneously map two different races, Native Hawaiian and White, to the same encoding:

```
public static int AFRICAN_AMERICAN = 0;
public static int AMERICAN_INDIAN = 1;
public static int ASIAN = 2;
public static int NATIVE_HAWAIIAN = 4;
public static int WHITE = 4;
public static int SOME_OTHER_RACE = 5;
```



Height: 1.77
Race: African American
Weight: 75.0
BMI: 23.93948099205209

Figure 3. BMISpreadsheet with Race property of type String

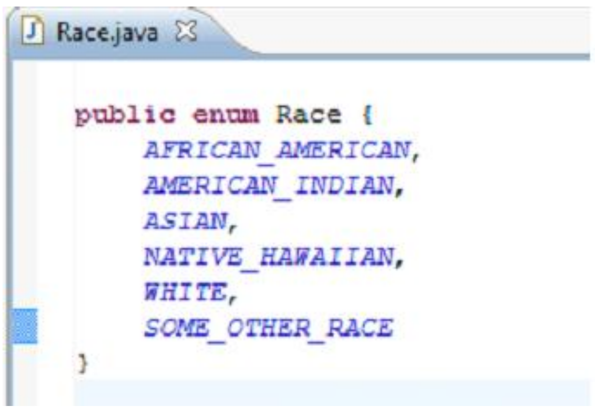
One way to solve the first problem is to represent the choices using their String names, as shown below:

```
private static String AFRICAN_AMERICAN = "African American";
private static String AMERICAN_INDIAN = "American Indian";
private static String ASIAN = "Asian";
private static String NATIVE_HAWAIIAN = "Native Hawaiian";
private static String WHITE = "White";
private static String SOME_OTHER_RACE = "Some Other Race";

String race = AFRICAN_AMERICAN;

public String getRace() {
      return race;
}
public void setRace (String newVal) {
      race = newVal;
}
```

As Figure 3 shows, now the user sees meaningful names for the choices. However, encoding each choice as a String is more space inefficient than representing it as an int, as for each String, all of its characters must be stored in memory. Moreover, it is still possible for users and programmers to make mistakes. For example, a user might enter African_American instead of African American. Similarly, as with the case of ints, two different choices can be accidentally mapped to the same String:

```
private static String NATIVE_HAWAIIAN = "Native Hawaiian";
private static String WHITE = " Native Hawaiian ";
```

This sort of problem often occurs when we copy and paste code. For example, we might create the second named constant above by copying the first one and edit the name of the constant but not its value.



```
public enum Race {
      AFRICAN_AMERICAN,
      AMERICAN_INDIAN,
      ASIAN,
      NATIVE_HAWAIIAN,
      WHITE,
      SOME_OTHER_RACE
}
```

Figure 4. Race enum

Enums

The reason for the errors above was that we manually mapped choices to their encodings. Java allows us to define special types, called enums, that automatically perform this task for us. Figure 4 is an example of an enum type defining the race choices.

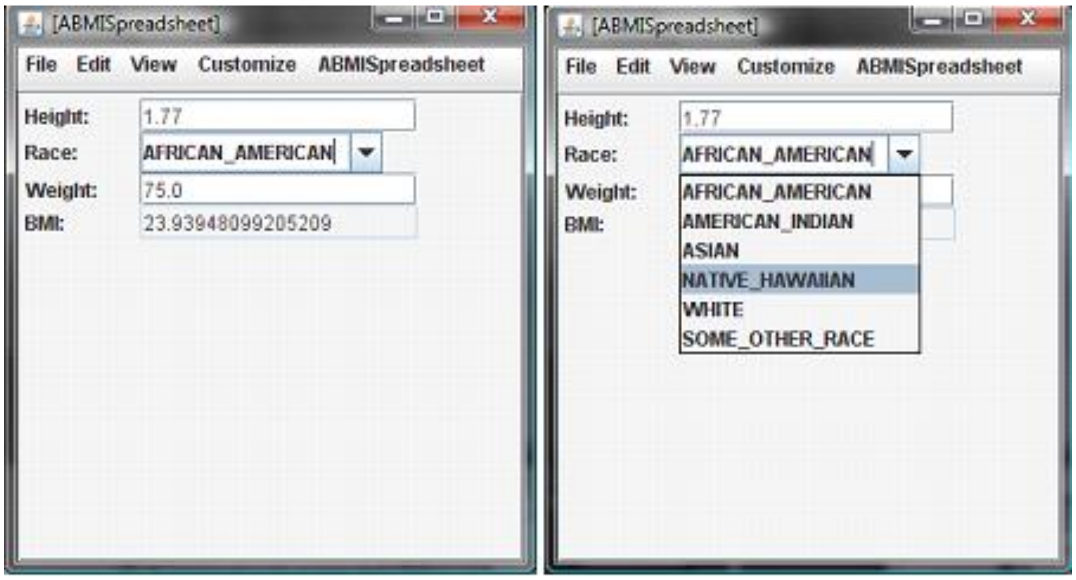
As in the case of a class and interface, the type definition consists of a header and body delimited by curly braces; the header defines the accessibility of the type the kind of type, and its name, and the name of the containing file adds the suffix .java to the name of the type. The keyword **enum** tells Java we are creating an enum type rather than a class or interface. The body of the type definition simply lists identifiers representing the different choices. Each of the choices is called an enum constant. Java automatically maps the enum choices to unique internal values, which are hidden from the programmer. Thus, no encoding errors occur.

The following code illustrates how enum types are used:

```
Race race = Race.AFRICAN_AMERICAN;
public Race getRace() {
      return race;
}
public void setRace (Race newVal) {
      race = newVal;
}
```

As we see above, each enum choice is specified by the name of the enum type (e.g. Race), followed by a period, followed by the identifier representing the choice (e.g. AFRICAN_AMERICAN). The reason for prefixing the enum constant with the type name is that different enum types can use the same identifier to represent a choice. For example, we can use the identifier FLY in the following two enum types:

```
public enum Action = { WALK, RUN, FLY, …}
public enum Insect = {MOSQUITO, FLY, …}
```



Height: 1.77
Race: AFRICAN_AMERICAN
Weight: 75.0
BMI: 23.93948099205209

Figure 5. ObjectEditor display of an enum property called Race

The name FLY, thus, does not uniquely identify an enum choice, while the specification, Action.FLY, does.

As Figure 5 shows ObjectEditor maps an enum property as a combo-box that displays the current value of the property and gives the user a menu of all choices defined by the type of the property. The user can select a menu item to set the property to the corresponding enum constant. For example, in Figure 5, the user selects the NATIVE_HAWAIIAN combo-box menu item to set the Race property to the NATIVE_HAWAIIAN enum constant. ObjectEditor does not display the name of the type of an enum choice as it is usually implied by context and thus there is likely to be no ambiguity.