CS112 Lecture: Primitive Types, Operators, Strings

Last revised 2/11/09

Objectives:

1. To explain the fundamental distinction between primitive types and reference types, and to introduce the Java primitive types and Strings
2. To introduce some of the constants and functions of java.lang.Math

Materials:

1. BlueJ BankAccount example to project
2. Dr. Java for demos
3. Table of operator precedence from Bruce book

I. Reference and Value Data Types

A. Java is what we call a *statically-typed* language - by which we mean that every variable is declared to hold only values of a particular data type, and may not be used to hold other types of values.

1. This stands in contrast with *dynamically-typed* languages (such as JavaScript), which allow the same variable to hold values of different types at different times during the execution of a program.

2. Static-typing has two advantages:

a) Efficiency. Dynamically-typed languages require that the actual type of a variable be checked every time the variable is used, because the meaning of an operation may depend on the type of operand it is applied to.

b) Safety. Static typing promotes more robust programs, because the compiler can catch certain errors that would otherwise cause the program to fail at runtime.

B. What are the permissible types for a variable in Java?

1. Basically, they fall into two broad categories:

a) Primitive, or *value* types - the storage allocated for the variable holds its actual value.

b) Reference types - the storage allocated for the variable holds a reference to another region of memory where the entity the variable refers to actually resides.

2. We have been using reference type variables since the beginning of the course.

   a) When we declare an object - as in

      Robot karel;

   or

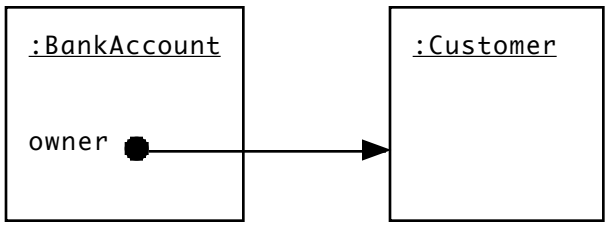
      FilledOval sun;

   What we are doing is declaring a reference type variable which will be used to refer to an object of the appropriate class when that object is created. (That is, what we have been calling an object is actually a variable that refers to an object - but calling it an object is a convenient abbreviation)

   b) Illustration of the meaning of "reference"

   BlueJ Bank Example demo - create a Customer and a BankAccount, then inspect BankAccount and double click owner reference to bring up Customer object.

   The situation that holds here looks something like this:



3. Primitive (value) types

   The astute observer will probably ask the question "if every variable in an object holds a reference to another object, then how do we escape from an OO variant of the 'chicken-and-egg' problem.?" The answer is that Java defines a certain number of "primitive" or "value" types that are - in Java - <u>not</u> objects.

   An illustration of this is the other two fields of the bank account object (account number and currentBalance). BlueJ does not depict them as holding a reference to some other object; instead, it shows them as containing an integer value. This corresponds to the fact that the variable actually holds a binary bit pattern that is meaningful to the CPU.

C. Although the number of different reference types that occur in a program is practically unlimited, since each Class constitutes a distinct type, the set of primitive types is fixed and defined by the language, and the names of these types are *reserved words* - words that can be used for no other purpose.

1. Thus far, we have met three primitive types (one introduced for the first time in this chapter).

   What are they?

   ASK

   `boolean` (used to represent true/false values for various tests)
   `int` (used to integers)
   `double` (used to represent real numbers)

2. There are a total of 8 primitive types in Java - i.e. five more.

   a) Three are integer types.

   (1) The type `int` we are already familiar with uses 32 bits to represent an integer, and thus can represent values in the range -2147483648 to 2147483647.

   (2) Two are smaller than `int` - and thus require less memory:

   `byte` (8 bits)
   `short` (16 bits)

   However, these are rarely used, given the large amounts of memory typically available on a modern computer. [An exception is when a program written in Java interfaces with another system that packs data more compactly]. We will not use them in this course

   (3) One larger than `int` - and thus can represent a much larger range of values

   `long` (64 bits)

   This type is also rarely used - not due so much to economy of memory considerations, but to speed considerations - on a 32-bit CPU, basic arithmetic operations on anything larger than an `int` take more time.

b) There is one additional type used for real numbers.

`float` (32 bits - `double` uses 64)

(1) Historically, this was the first type used for real numbers (even as far back as the 1960's.) The type "double precision" was introduced for cases where greater precision was used at the cost of more memory.

(2) In Java, this type is rarely used. There isn't the same hardware speed issue with 64 bit versus 32 bit real numbers (since special floating point hardware is used), and memory size is usually not an issue either.

c) There is one final type that we will meet more thoroughly later in the course

`char` (used to represent an individual character)

d) As implied by the above, for the most part in this course we will only use the three primitive types we have met thus far (`boolean`, `double`, `int`), plus `char`.

3. Primitive (value) types differ from reference types in a number of ways.

a) The names of primitive types are all lower-case. By convention, the names of reference types begin with an upper-case letter.

b) Value types are <u>not</u> objects.

(1) The operator `new` is <u>not</u> used to create them. (Reference types are always created this way.)

(2) Methods cannot be applied to them. You will <u>never</u> see something like:

```
int i;
i.something();
```

c) Java provides standard ways of writing literals of most of the value types. Examples (for the types we will use):

(1) `boolean - false, true`

(2) `char - 'A'`

(3) `int  - 1`

(4) `double - 1.0` (the decimal point distinguishes from `int`), or scientific notation can be used - e.g. `6.02E23`

d) Java provides various operators that can be applied only to entities of primitive type: Examples:

(1) For `boolean` - boolean operators such as `&&, ||, ^, !`

(2) For the numeric types (`int`, `double`) - arithmetic operators such as `+, -, *, /, %`

Note, though, that the operator `/` has a somewhat different meaning depending on its operand types.

(a) When used with two `int`s, it means integer division. Any remainder is discarded.

DEMONSTRATE with Dr. Java: `2/5`

(b) When used with two `double`s, or a `double` and an `int`, it means real division.

DEMONSTRATE with Dr. Java: `2.0/5.0`

(3) For the numeric types plus `char` - relational operators such as `>, <, >=, <=`

e) The equality comparison operators (== and `!=`) can be applied to both kinds of types, but they mean something different for reference types than they do for value types.

(1) When applied to two entities of value type, they ask "do these two represent the same value?"

(2) When applied to two entities of reference type, they ask "do these two refer to the same object?"

(3) Of course, the answer to the first question will always be yes when the answer to the second question is yes. But the reverse is not true. It is possible to have two different objects that - in some sense - represent the same value. [Example: two different bank accounts with the same balance].

f) Early in the couirse, we saw that rules of precedence are used to resolved ambiguities in expressions that contain several operators - e.g.

```
1 + 2 * 3
```

is interpreted as `1 + (2 * 3)`, rather than as `(1 + 2) * 3`.

Recall that, in conjunction with discussing boolean expressions, we looked at a more complete set of precedence rules that takes into account the additional operators we have just met. Our book introduces these in this chapter - lets look at the table the book presents, which is essentially similar to what we looked at earlier.

PROJECT: Table of operator precedence

D. Just as variables have a type, so expressions have a type.

1. The type of an expression is determined by the types of its operands and operators. The following rules deal with the data types we will use - there are some additional rules that apply to the numeric types we will not discuss.

a) For the arithmetic operators , the type of the result of operation is determined by the types of its operands according to a rule that is sometimes called the *rule of numeric contagion*.

(1) For unary operators, the type of the result is the same as the type of the operand.

(2) For binary operators:

(a) If either operand is of type double, the other is converted to double (if necessary) and the result is of type double.

(b) Otherwise, the result is of type int.

b) For the relational and equality comparison operators

(1) If the two expressions being compared are numbers, they are converted according to the rules of numeric contagion and then compared.

(2) A `char` can only be compared to a `char`.

(3) `booleans` and reference types can only be compared for equality

The result of any comparison, of course, is always `boolean`.

c) Note that the rules are applied step by step during the evaluation of the expression.

Example: What is the value of the following expression

(4 / 5) * 2.0

ASK

Answer: 0.0 !

The division 4 / 5 is done in type int, because both operands are ints. The quotient is 0 and the remainder of 4 is discarded. The int 0 is then converted to double 0.0 and multiplied to still yield 0!

2. A related requirement is that the type of an expression that is assigned to a variable be *assignment compatible* with the variable.

(1) A variable of type `double` may be assigned the result of any numeric computation. If the result is an integer, it is promoted.

(2) A variable of any other type may only be assigned an expression whose value is of the same type.

E. There are times when it is necessary to perform an operation where the needed type conversion is not automatically performed. In this case, one must use an explicit *type cast*.

Example: The following statement is erroneous, because the expression is of type `double` and the variable is of type `int`:

```
double d;
int i = 3 * d;
```

To make this work correctly, we need to use a type cast, as follows:

```
int i = (int) (3 * d);
```

Note: Java requires an explicit cast, because a computation like this might - in principle - result in the loss of information:

The explicit use of a cast means "I know this could result in the loss of information, but in this case I know this won't happen, or I'm prepared to accept it."

II. Mathematical Functions

A. One facility any general purpose programming language must provide is a mechanism for using standard mathematical functions.

B. In Java, for consistency with the style of the language, the standard mathematical functions are made available as class methods of the library class `java.lang.Math`. The package `java.lang` is special in the sense that classes in it do not need to be imported; they are automatically available to every Java program. So one can write `Math.whatever` in a Java program without having to explicitly import the `Math` class.

PROJECT - Javadoc for `java.lang.Math`.

1. Discuss all fields and methods.

2. Demo using Dr. Java

   Note that the various trigonometric functions (and their inverses) expect angles to be specified in radians, not degrees. The `Math` class provides methods `toDegrees()` and `toRadians()` to simplify the conversion

III. The Type String

A. Java defines one special data type that, although it is a reference type, has some of the characteristics of a primitive type. It is the type `String`.

1. A `String` represents a sequence of characters of arbitrary length - possibly as short as zero, and potentially as long as available memory will allow.

2. This type is extremely useful in programs that manipulate textual data - as almost all do.

   a) Because of its usefulness, it would have been nice if it could be a primitive type. However, whereas primitive types use a fixed amount of storage (e.,g., 8, 16, 32, or 64 bits), the amount of memory required by a string varies greatly depending on how long it is. This makes it impossible for it to be a primitive.

   b) Instead, Java treats it as a special reference type - there is no other type like it in the Java language. We will see two ways this is true in a moment.

3. We have already used this type in various places - e.g. when specifying the name of a Customer in the Bank Account example.

B. As is the case with primitive types, the Java compiler recognizes constants of type `String` - any sequence of characters, enclosed in double quotes (").

   1. A pair of quotes with nothing in between (e.g. "") is interpreted as a `String` of length 0.

   2. Certain characters need to be escaped if they need to occur in a string. This includes (among others).

      a) The quote character is represented as \".

      b) The newline character is represented as \n.

      c) The backslash character is represented as \\.

C. The Java compiler recognizes the use of the operator + with strings, as meaning append the two strings. In fact, if either of the operands of + is a string, the other is automatically converted to a string if it is not.

   DEMONSTRATE with Dr. Java

```
"Hello" + "World"  (note no space)
"r" + 2 + "d" + 2
import karelrobot.*;
  Robot karel = new Robot(1,1,Directions.East,0);
  "My name is " + karel;
"x" + "2" + "2"
"x" + 2 + 2 (perhaps a surprising result because of type conversion!)
"x" + (2 + 2)
```

D. However, it is important to remember that `String` is a reference type, and that strings are objects! That means that:

1. Strings can be created using `new` (though usually they do not need to be).

2. Strings can be operated on using method syntax

3. DEMONSTRATE

   `"abc".length()` (`length()` is one of the methods of class String)

4. The relational operators cannot be used, and the equality testing operators have their "reference type" meaning

   DEMONSTRATE

   ```
   "a" < "b"
   ("a" + "b") == "ab"
   ```

   To address the need to compare strings, the class defines two methods that allow one string to be compared to another

   DEMONSTRATE

   ```
   "a".compareTo("b")
   ("a" + "b").equals("ab")
   ```