SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: About Java

We'll look a bit at some things you should remember or learn about Java, as the case may be.

Java in the World of Programming Languages

Java is a powerful and popular programming language, but is just one of many. In fact, we have a whole course about programming languages that you should take next time it's offered.

Some languages, like C and C++, are *compiled languages*.

- Compilers are CPU specific: chip dependent codes and registers, etc.
- Have to recompile to transport to a different computer architecture
- Data types are chip dependent
  - Short $\leq$ Int $\leq$ Long
  - But the size is not specified in language
  - Wrong assumption about size will cause problems!

Java does things differently.

- Java compiles to *bytecodes*: an intermediate "machine like language"
- Then the Java runtime system interprets the bytecodes and runs the machine-language for the target CPU
- Primitive type sizes are specified and known
- You should never have to re-compile! "Write once, run everywhere"
- The price: the code will be run less efficiently (slower)

Data Types

In digital computers, everything is processed in *binary*.

- Instructions, programs

- Primitive data: integers, floats, characters
- Sound and images
- Reference data: hold a memory address
- Objects are complex data structures
  - Include data and methods (code)
  - Referenced by a reference (pointer) data type

Primitive Types

Java defines several *primitive types*. This means Java is not a "pure" object-oriented language, but it does help with efficiency.

- Primitive data is stored directly into memory locations
  - Storage requirements are known
  - Storage is allocated for data immediately
  - The value in the right hand side of an assignment statement is copied to the variable on the left hand side. Only the left hand side variable changes.

  Let's consider this code fragment (will draw in class)

```
int a = 6;
int b = 3;
a = b;
```

- Integral: `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits)
- Floating point: `float` (32 bits), `double` (64 bits)
- Other: `char` (16 bits, Unicode) , `boolean` (1 bit?)

What does all this mean about the values that can be stored in Java's primitive types?

- How many different values can be stored in one bit?

  2 values: 0 and 1
- How many different values can be stored in two bits?

  4 values: 00, 01, 10, and 11

- How many different values can be stored in three bits?

  8 values: 000, 001, 010, 011, 100, 101, 110, and 111

- How many different values can be stored in N bits?

  2^N

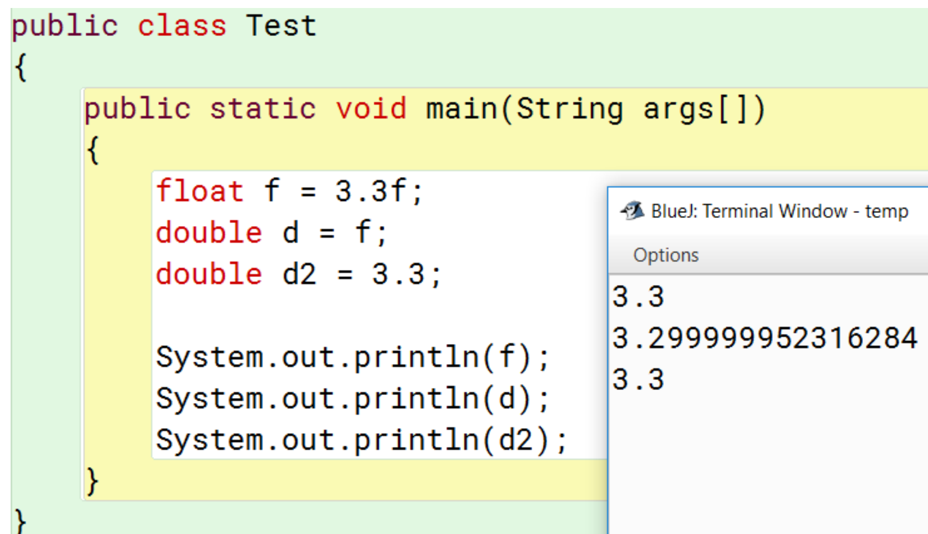
This means the range of values that can be stored by the common integral-like data types is as follows:

Data Type	Range	Number of Bytes (Bits)
`char`	0 to 65,535	2 (16)
`byte`	-128 to 127	1 (8)
`short`	-32,768 to 32,767	2 (16)
`int`	-2,147,483,648 to 2,147,483,647	4 (32)
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 (64)

Floating-point data introduces some complications.

The `float` type uses 4 bytes of storage and offers 6-7 digits of precision. `double` uses 8 bytes, which increases precision to 15-16 digits.

It is important to be aware that most real numbers cannot be stored precisely in floating-point representations. Consider this code and output:



So.. be aware that Java will "round" values to the nearest storable floating-point value. Be careful when comparing floating point numbers for equality with == or !=.

Two terms to keep straight here are the *precision* and the *accuracy* of floating-point representations:

If a particular floating-point representation of π prints with 10 digits of precision as 3.133333333, be aware that this only has 2 digits of accuracy. So just because Java prints out digits to a given level of precision, don't assume they're all accurate digits!

The `char` datatype is pretty straightforward, with each character represented by a numeric code

- ASCII codes are 8 bits for the standard English alphabet, numerals, and punctuation
- Unicode are 16 bits for international characters
- In either case, can be compared with relational operators based on their numeric codes, or *ordinal* values

Of course, the `boolean` data type only can take on the values `false` and `true`. Please note the advice in the style guide about avoiding the use of == and `!` in conditions that simply need to check if a `boolean` value is `true` or `false`.

Reference Types

Java variables can also hold *references* to objects.

- A reference variable will store the *address* of the object containing the data
  - 32 bits in Java
  - Implementation is hidden
  - Points to an object instance
- All object variables are reference types

  ```
  Car myCar  = new Car("Ford", "Escape", 2013);
  Integer numOne = new Integer(4);
  ```

- Arrays are built into the language but like objects are accessed through reference types

  ```
  int[] examScores;
  ```

  This declaration tells the compiler that this variable will hold an array of `int`.

  ```
  examScores = new int[5];
  ```

  This allocates an array with enough memory for 5 `int` elements, assigns the array data's memory reference to the `examScores` variable, and initializes all elements to their *default value*, in this case, to 0.

  Here are Java's default values that are used to initialize array elements when an array is constructed.

  - Object (reference) types – `null`
  - Integral types – 0
  - Floating point types – 0.0
  - `boolean` – `false`
  - `char` - '
    u0000' (the null character)

- Arrays can be initialized on declaration, if you know all of the values that are going to be stored:

```
int[] labScores = { 100, 98, 56, 100, 100 };
```

  This results in the same array as if you did the following:

```
int[] labScores = new int[5];
labScores[0] = 100;
labScores[1] = 98;
labScores[2] = 56;
labScores[3] = 100;
labScores[4] = 100;
```

- Arrays are "sort of" objects, since they do have a *public constant* called `length`
- When we construct an array of objects, it is really an array of references to objects – no objects are created or stored unless/until you put them there
- Once we have the objects in the array, we can call methods on them just like you normally would (assume a hypothetical class `Account` has the given constructor for the account holder name and initial balance, and a `deposit` method):

```
Account bob = new Account("Bob Bobberton", 35);
bob.deposit(50);
Account[] accounts = new Account[3];
accounts[0] = new Account("Joe Smith", 0);
accounts[0].deposit(25);
accounts[1] = bob;
accounts[1].deposit(40);
```