Programming in Java

2. Variables and Data Types

Data Types

- In keeping with OOP, all data should be define in terms of objects
  - However, this is impractical from a hardware and processing perspective
  - Especially in 1994 when hardware performance was a processing bottleneck
- Java defines a set of primitive data types
  - These are specifically designed to be used by the stack (more on that in a moment)
  - They are all of a fixed size which is required for them to be on the stack
  - The fixed size is either 32 bits or 64 bits
  - This ensures that data can be moved in no more that two clock cycles
    - *On 32 bit architectures, moving 64 bits can take two clock cycles*
  - Primitive types are intended to allow for fast stack based computation
- There is one primitive type that is not used in computation
  - References to an allocated chunk of heap memory are 32 or 64 bit memory addresses
  - Java does not allow us to directly access these values so we don't corrupt memory

Data Types

- The different primitive data types are stored in different formats in memory
- Based on the type of data and its size
- Java can determine the type of data when looking at its string representation
  - Integer types are a series of digits without a decimal point like 49837
  - Float types are a series of digits with a decimal point or in scientific notation like 1.0 or 3.2E4
  - A character is a single Unicode utf-16 character in single quotes like 'a' or '♣'
  - A string is a series of Unicode utf-16 characters like "Hello world"
    - *Note that the table in the following slide is wrong – chars take values from 0 to 65,535, not 256*
  - 'true' and 'false' are logical values stored as a byte
- When using variables, we have to tell Java what type of data it contains

The Standard Primitive Data Types

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

LearnQuest

Stack Versus Heap Memory

- This is standard architecture
- Usable memory for applications is divided into
  - The stack: Under the control of the OS
  - The heap: Under the control of the user
- The heap starts at the highest available memory address and grows down
- The stack starts at the lowest memory address and grows up
  - Up to the limit of the allocated stack memory
- The white space in the middle is available memory for the heap
- If it goes to zero then
  - When the heap tries to allocate memory an *out of memory* error is generated
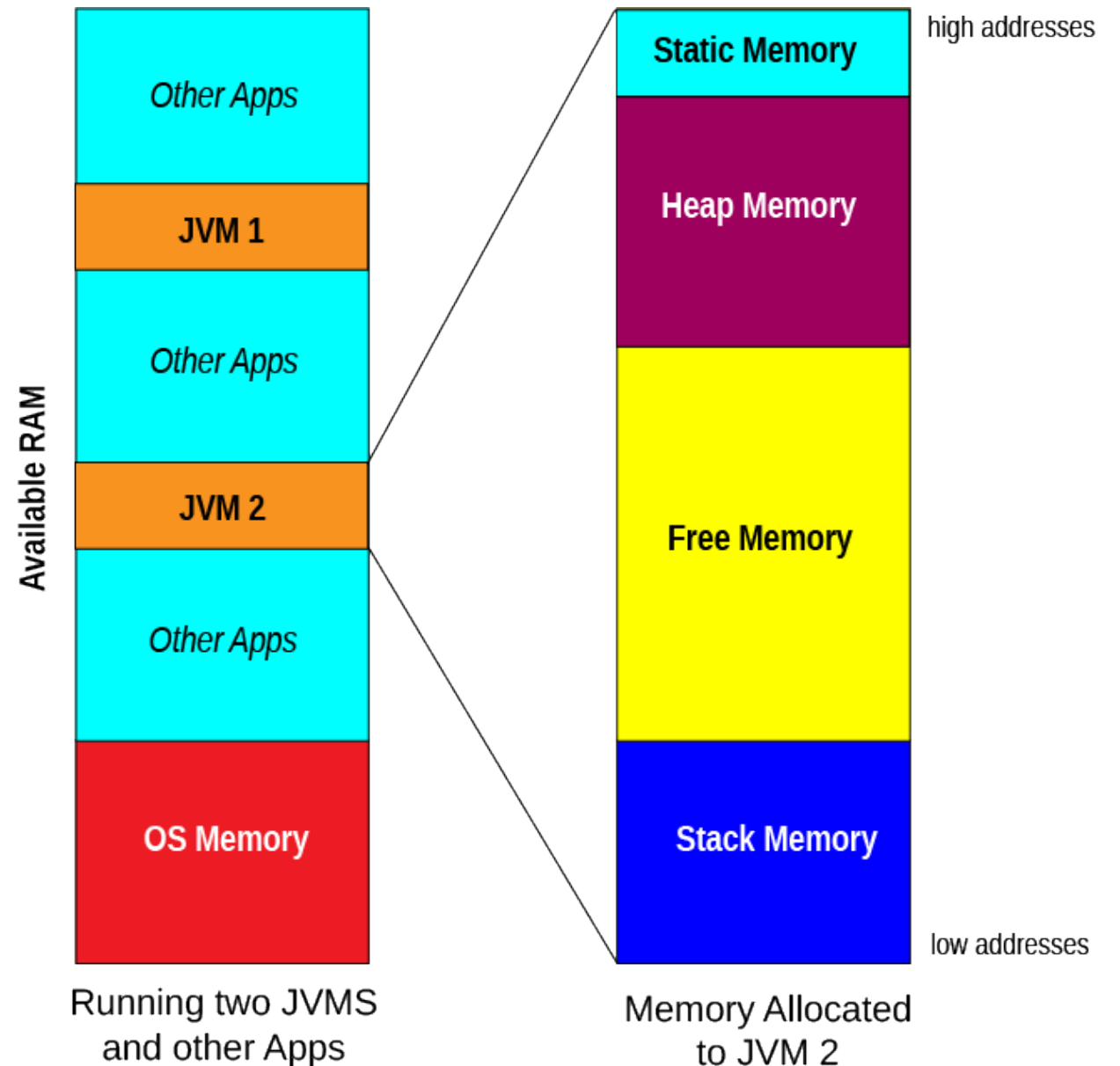 - This usually causes a program to terminate
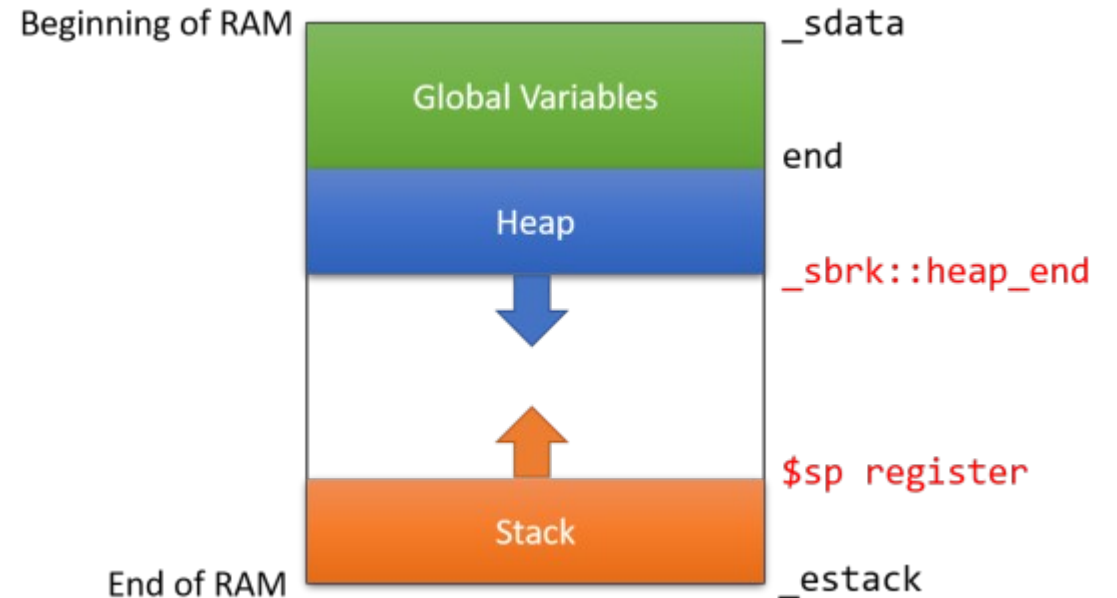
Stack Versus Heap Memory

Stack	Heap
Fixed in size (set by Operating System)	Can shrink/grow in size in real-time
Stack is contiguous memory (i.e., sequential memory addresses)	Heap memory is not contiguous (i.e., not sequential memory addresses)
Memory allocation and release is automatically managed	Memory allocation, use and release is up to the programmer
Memory allocation is fast: only the stack pointer needs to move	Slower than the stack, as space for dynamic variables needs to be found in real-time
Is a First-In-Last-Out (FILO) / Last-In-First-Out (LIFO) system	Heap variables not allocated sequentially, memory can become fragmented
Variables sizes are fixed at compile-time and cannot be resized	Variable sizes can be set at allocation time and can (somewhat) be resized
Variables in stack memory are always in scope (function-based memory allocation)	Heap memory has no scope but pointers to Heap memory do!
Size of variables in stack memory are known at compile time, so variables can have variable names	Size of variables not known until dynamically allocated, so Heap memory can only be accessed with pointers

Image Credit: hhttps://cppbetterexplained.com

Java Memory Management

- Memory in the JVM is handled exactly like an OS handles physical memory

- The size of the stack is fixed when the JVM starts up

  - This can be adjusted by tweaking the JVM parameters

- The static memory is where any data that remains in memory for the duration of the time the JVM is running is located

Available RAM

| Other Apps |
| JVM 1 |
| Other Apps |
| JVM 2 |
| Other Apps |
| OS Memory |

Running two JVMS and other Apps

high addresses

| Static Memory |
| Heap Memory |
| Free Memory |
| Stack Memory |

low addresses

Memory Allocated to JVM 2

LearnQuest

Memory and Data

- There are three kinds of storage types in Java
- Static data
    - Also called permanent data
    - This is data that is initialized when the JVM starts up
    - It remains in memory until the JVM shuts down
    - This includes constants and interned data (we will define that later)
- Automatic data
    - This is data that is managed by the JVM on the stack
    - This data is primarily of local variables created during execution of a method
    - The stack removes these variables from memory when they go out of scope
- Managed data
    - This is data that is created on the heap in the code, usually with the "new" operator
    - It remains on the heap until it can no longer be accessed from the code
    - Inaccessible data is deleted from the heap when the garbage collector runs

Type Safety

- All data values have a type
  - Its type determines how it is stored in memory
  - For a non-typed language, like Python
    - *A variable can be bound to any type of data*
    - *But data values still have types like in Java*
- In a strongly typed language, like Java, variables must have a type
  - The typed variable is allowed to store one specific type of data
  - This is to ensure that potential run time errors are caught at compile time, like trying to add a Boolean and a floating point number
- Literals are strings of characters in a program listing that can be parsed as data
  - Java has a set of rules to assign a type to a literal
  - This ensures that the wrong type of data is not assigned to a variable at run time
  - eg. '123' is a 32 bit integer while '123L' is a 64 bit integer
    - *Tip: Underscores can be added to any numeric literal for readability 898979211 = 898_878_211*
  - Any floating point literal is a double by default unless post-fixed with an 'F' eg. 89.3F

Type Safety and Casting

- In certain cases, some data types can be converted to other data type
  - Integers can be assigned to longs since they are both integer data types
  - Floats can be assigned to doubles since they are both floating point data types
  - These are called *widening* conversion since the target is bigger than the source
  - *Narrowing* conversions where the target is smaller than the source are not allowed
  - Integer types can be assigned to floating point numbers but the will be a loss of precision
- We can override Java's rules by casting or allowing the conversion to take place
  - This may result in data loss or errors at runtime
  - eg.   int k = (int)1.9;   casts a float to an int
  - Casting a float to a non-float causes the fractional part to be truncated
  - Casting a 64 bit value to a 32 bit value may cause a loss of precision
- We can only cast between different numeric types
  - Java has no idea how to cast a String to a boolean or to an int for example

Character Data

- Java was designed as an Internet language
    - At that time UFT-16 (16-bits per character) was the standard language encoding on the Internet
    - C and C++ were using ASCII 8-bit character encoding
    - Java source code and Java data both use UTF-16 encoding
    - It helps to think of the first byte of a Java character as an alphabet and the second to be a letter in the alphabet
        - *ASCII is a subset of UTF-16 where the first byte is zero*
    - However, UTF-16 has been replaced in the Internet world by the variable length encoding UTF-8

Java Syntax

- Statements
  - A statement is a complete unit of execution.
  - In Java, most statements end with a semicolon (;)
- Expressions
  - An expression is any piece of code that produces a value.
  - Expressions can be part of statements.
  - Note that the variable declaration statement shown previously is NOT and expression, it does not produce value
    - *In other words, we can't do the following to get 6*
    - *1 + int = 5*
  - These are often used inside statements:

```java
int x = 5;            // variable declaration statement
x = x + 1;            // assignment statement
System.out.println(x);  // method call statement
```

```java
x + 2            // expression
3 * (y - 1)      // expression
"Hello" + name   // expression
```

```java
int total = x + 2;
```

Java Syntax

- Semicolons (;)
    - Semicolons terminate statements, not blocks.
    - Each statement must end with a semicolon.
    - Java won't process a statement until it finds a semicolon

- Blocks {}
    - A block is a group of statements enclosed in curly braces {}.
    - Used in method bodies, loops, if statements, etc.

```java
if (x > 0) {
    System.out.println("Positive");
    x = x - 1;
}
```

Java Syntax

- Identifiers
  - Identifiers are names for variables, methods, classes, etc.
  - Must follow these rules:
    - *Start with a letter, underscore _, or dollar sign $.*
    - *Subsequent chars can be letters, digits, _, or $.*
    - *Cannot be a Java keyword (e.g., class, public).*
- Utf-16
  - The letters referred to in the above are utf-16 so the following is a valid Java identifier
  - double π = 3.14159;
- Keywords
  - Java reserves words for its own use — you cannot use these as variable names.
  - Examples: int, class, if, else, while, return, new, void, static, etc.
- Whitespace
  - Java ignores whitespace (spaces, tabs, newlines) except to separate tokens.
  - Proper indentation is not required but makes code readable. The following is valid but very hard to read
    - *int a=10;System.out.println(a);*
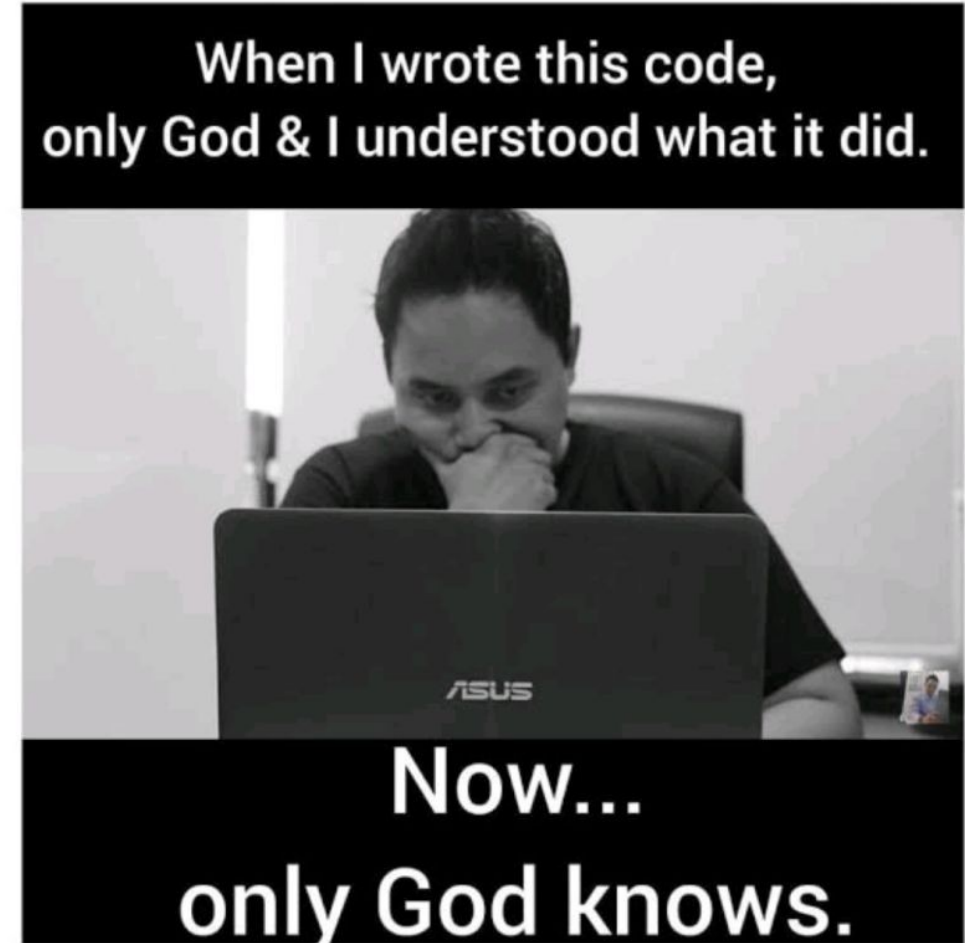
Java Syntax

- Comments
  - Comments document code.
  - Helps other programmers understand your code
  - Or you, when you see it again later

  // single-line comment

  /*
  multi-line comment
  */

When I wrote this code, only God & I understood what it did. Now... only God knows.

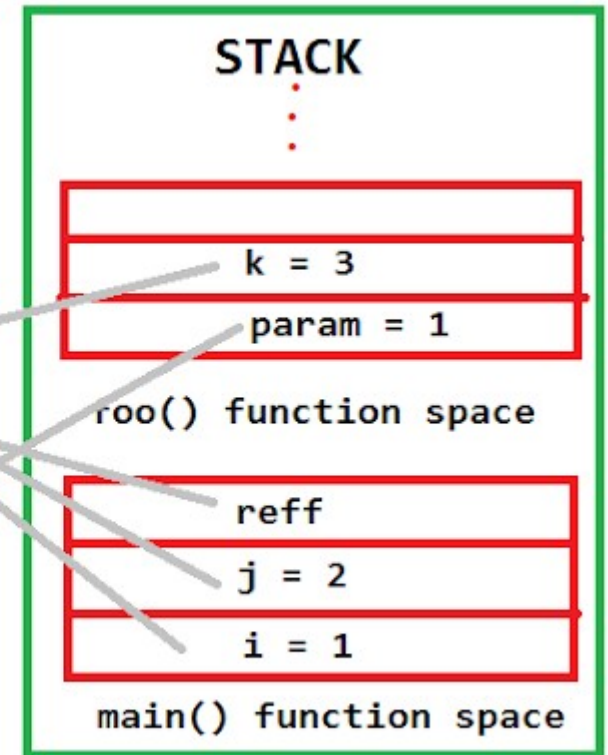**Lab 2-1**

Data Types and Casting

Variable Types

- Data is referenced by variables
- Variables, regardless of what type of data they reference, are one of three types
  - Static
  - Automatic
  - Managed
- Every variable has two properties
- Lexical Scope
  - This just means where in the code a variable can be referenced from
  - Variables can only be used if Java can "see" them
  - The block of code where a variable can be seen is called its lexical scope
- Extent
  - This is the amount of time that a variable is in storage
  - Static variables have infinite extent
  - Automatic and managed variables are created and then destroyed with they go out of scope

Automatic Variables

- Works the same way in most Programming languages

- All local variables in methods are automatic

- "Automatic" means that storage is automatically managed by the stack

- Braces are used by Java to indicate a lexical scope

  - An automatic variable scope is from the time it is declared until the closing brace } in the scope it is declared in

  - We can insert whatever addition scopes we want

  - { } in method bodies are a scope


```
public class Stack_Test {

    public static void main(String[] args) {

        int i=1;
        int j=2;

        Stack_Test reff = new Stack_Test();
        reff.foo(i);
    }

    void foo(int param) {
        int k = 3;
        System.out.println(param);
    }

}
```

STACK

k = 3
param = 1
foo() function space

reff
j = 2
i = 1
main() function space

Demo

Automatic Variables and Scope

Static Variables

- These are defined inside class definitions

  - Essentially these are global variables

  - At the time Java was designed, global variables were considered not proper OO

- Static variables are created when the class is loaded

  - The are also initialized when created

  - Either by using explicit initialization or defaulted to the "natural zero value"
    - *Natural zeros were 0 for numerics, null for references and false for booleans*
    - *Using the defaults is considered poor programming style*

- The variables exist while the JVM is running and are never destroyed

  - The variable is referenced using the *classname.variablename* where *classname* is the class in which it is defined

- The lexical scope of a static variable is determined by the class

  - If the class definition is visible to the some other code, then the variable can be referenced in that code

  - A static variable is also in scope from any method inside the class it is defined in

  - Outside the class, it may need to be declared *public* – more on that later

Demo

Static Variables

Final Variables

- Java does not have a *const* keyword but uses the modifier *final* in different contexts
- When used with a variable, it means that the variable cannot appear on the LHS of an assignment statement
    - The variable must be initialized when it is created
    - This is because memory for the final variable is located in a special permanent memory area
    - Since the value of the variable cannot change, a more efficient storage scheme is used
    - It can be thought of as a literal
    - The value is said to be *interned* in a constant storage pool in the heap static memory
    - When two final variables have the same value, they share the same interned constant as a value

Lab 2-2

Variables