

# Programming Model

Please note that these slides are based in part on material originally developed by Prof. Kevin Wayne of the CS Dept at Princeton University.

# Java

- What is Java exactly? It's two things, actually:
  - It's the Java Language (current version is Java 9); and
  - It's the *Java Virtual Machine*
    - Many languages run on the JVM:
      - Java, Scala, Groovy, Kotlin, Clojure, JRuby, Jython
      - Plus many languages which have been ported to use the JVM
    - The JVM has a class-loader which, in turn, enables the concept of plugins and servlets.

# Java (continued)

- What is special about Java exactly?
  - For the first time (because of the JVM), programmers could write code and run it on *any* machine!!
  - For the first time (at least as far as I know), it had automatic garbage collection!
  - It's object-oriented (it wasn't the first but it was the first *major* language to be O-O).
  - It introduced an important new paradigm for problems that happen during run-time: exceptions
    - Languages had mechanisms before (e.g. "C" longjump) but exceptions really transformed programming.

# The JVM and its importance

- The JVM is what actually *runs* your program.
- Its input is called *byte-code* and it translates this into architecture-specific instructions.
- It knows about: *int, double, long, float, char, short, byte, array, reference*.
- It knows about nothing else!

# Java and its data types, etc.

Our study of algorithms is based upon implementing them as programs written in the Java programming language. We do so for several reasons:

- Our programs are concise, elegant, and complete descriptions of algorithms.
- You can run the programs to study properties of the algorithms.
- You can put the algorithms immediately to good use in applications.

**Primitive data types and expressions.** A *data type* is a set of values and a set of operations on those values. The following five primitive data types are the basis of the Java language:

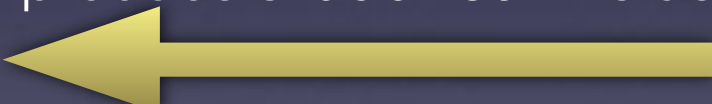
- *Integers*, with arithmetic operations (`int`) : 32-bits
- *Real numbers*, again with arithmetic operations (`double`) : 64-bits
- *Booleans*, the set of values { *true*, *false* } with logical operations (`boolean`): undefined
- *Characters*, the alphanumeric characters and symbols that you type (`char`): 16-bits
- *Bytes*, pure binary data (`byte`): 8 bits

A Java program manipulates *variables* that are named with *identifiers*. Each variable is associated with a data type and stores one of the permissible data-type values. We use *expressions* to apply the operations associated with each type.

term	examples	definition
<i>primitive data type</i>	int double boolean char	a set of values and a set of operations on those values (built in to the Java language)
<i>identifier</i>	a abc Ab\$ a_b ab123 lo hi	a sequence of letters, digits, _, and \$, the first of which is not a digit
<i>variable</i>	[any identifier]	names a data-type value
<i>operator</i>	+ - * /	names a data-type operation
<i>literal</i>	int                    1 0 -42 double                2.0 1.0e-15 3.14 boolean                true false char                   'a' '+' '9' '\n'	source-code representation of a value
<i>expression</i>	int                    lo + (hi - lo)/2 double                1.0e-15 * t boolean                lo <= hi	a literal, a variable, or a sequence of operations on literals and/or variables that produces a value



type	set of values	operators	typical expressions	
			expression	value
int	integers between $-2^{31}$ and $+2^{31}-1$ (32-bit two's complement)	+	5 + 3	8
		-	5 - 3	2
		*	5 * 3	15
		/	5 / 3	1
		%	5 % 3	2
double	double-precision real numbers (64-bit IEEE 754 standard)	+	3.141 - .03	3.111
		-	2.0 - 2.0e-7	1.9999998
		*	100 * .015	1.5
		/	6.02e23 / 2.0	3.01e23
boolean	true or false	&& (and)	true && false	false
		(or)	false    true	true
		! (not)	!false	true
		^ (xor)	true ^ true	false
char	characters (16-bit)	<i>[arithmetic operations, rarely used]</i>		

- **Expressions.** Typical expressions are *infix*. When an expression contains more than one operator, the *precedence order* specifies the order in which they are applied: The operators `*` and `/` (and `%`) have higher precedence than (are applied before) the `+` and `-` operators; among logical operators, `!` is the highest precedence, followed by `&&` and `and` and then `||`. Generally, operators of the same precedence are *left associative* (applied left to right). You can use parentheses to override these rules.
  - **Type conversion.** Numbers are automatically “widened” to a more inclusive type if no information is lost. For example, in the expression `1 + 2.5`, the `1` is widened to the `double` value `1.0` and the expression evaluates to the `double` value `3.5`. A *cast* is a directive to convert a value of one type into a value of another type. For example `(int) 3.7` is `3`. Casting a `double` to an `int` truncates toward zero.
  - **Comparisons.** The following *mixed-type* operators compare two values of the same type and produce a `boolean` value:
    - equal (`==`)
    - *not equal* (`!=`)
    - *less than* (`<`)
    - *less than or equal* (`<=`)
    - *greater than* (`>`)
    - *greater than or equal* (`>=`)
- 
- Please note: this means literally the same



# Primitives vs. Objects

- As well as the five primitives described above, there are also long (64-bit), short (16-bit), and (32-bit) float.
- All of these primitives have a “boxed” object which essentially wraps the primitive value: viz. Integer, Double, Long, Boolean, Char, etc. Boxing/unboxing is, normally, automatic (like widening) according to the context.
- Null pointers: If *x* is an object, you can write `if (x==null)`. If *x* is a primitive, this doesn’t make sense.
- Primitives and objects are all passed into methods by *value* in Java. That’s clear for a primitive—but what does it mean for an object? Objects are really just pointers.

# Primitives vs. Objects (contd.)

- Why aren't objects passed by value?
  - It's easy to pass primitives by value: their lengths are chosen to correspond to the normal word size of the processor (core)\*.
  - But passing objects by value would mean copying—byte by byte: for a large hash table, e.g. this could take a long time!
- What else is different between primitives and objects?
  - primitives are supported by the Java Virtual Machine (JVM)—that's to say there is *byte code* for the various primitive operations such as addition, division, etc.
- In that sense, arrays are like primitives: there is special support for them in the JVM. But in every other sense, they are treated as objects (no copying).

\* back in the 80s, there were machines with 36-bit words. Not since.

# There's one other very important object in Java...

- *String*:
  - Because of its importance, *String* is a **final class** in Java (and Scala, BTW). That's to say you can't subclass it and you can't mutate it at all. In fact, *String* is implemented in Java the same way it would be in any functional programming language.
  - We will be talking about the very important and ubiquitous *String* algorithms towards the end of the semester.

# Which should we use?

- The JVM (and its arithmetic functions) knows little about objects. As far as the JVM is concerned, they are only pointers to an instance of a class (but no parametric (generic) type because that information is erased).
- Otherwise, the JVM only knows about the primitives and arrays of primitives, or objects, e.g. `int[] z = new int[10];`
- So, where performance is important, always use primitives and, usually, arrays of primitives.

# When *isn't* performance important?

```
public class Matrix {  
  
    private final int rows;  
    private final int columns;  
    private final double[][] values;  
  
    public Matrix(final int rows, final int columns, final double[][] values) {  
        super();  
        this.rows = rows;  
        this.columns = columns;  
        this.values = values;  
    }  
  
    public Matrix multiply(final Matrix other) {  
        if (this.columns == other.rows) {  
            double[][] result = new double[rows][other.columns];  
            for (int i = 0; i < this.rows; i++) {  
                for (int j = 0; j < other.columns; j++) {  
                    double x = 0;  
                    for (int k = 0; k < this.columns; k++)  
                        x += this.values[i][k] * other.values[k][j];  
                    result[i][j] = x;  
                }  
            }  
            return new Matrix(this.rows, other.columns, result);  
        }  
        else  
            throw new RuntimeException("incompatible matrices");  
    }  
}
```

This outer loop has  $O(N)$  complexity:  
performance not critical

This inner loop has  $O(N^3)$  complexity

- Let's assume  $N$  is  $\sim 1000$ .  $i$  loop is run 1000 times;  $j$  loop 1 million;  $k$  loop: 1 billion times



# The inner loop

- You nearly always have to worry only about the inner loop because, assuming  $N$  is large, the inner loop swamps the times of the outer loops;
- So, for example, you have plenty of time to unpack the rows and columns of the two multiplicands from a less efficient form such as *List<List<Double>>* into *double[][]*.

# Binary search

- Let's take a quick look at the BinarySearch class and algorithm from the textbook (p47). Note that the name (and signature) of the method is different (*rank* vs. *indexOf*)

*import a Java library (see page 27)*

```
import java.util.Arrays;
```

*code must be in file BinarySearch.java (see page 26)*

```
public class BinarySearch
```

*parameter  
variables*

*static method (see page 22)*

```
{  
    public static int rank(int key, int[] a)
```

*initializing  
declaration statement  
(see page 16)*

```
{  
    int lo = 0;
```

```
    int hi = a.length - 1;
```

```
    while (lo <= hi)
```

```
    {  
        expression (see page 11)  
        int mid = lo + (hi - lo) / 2;
```

```
        if (key < a[mid]) hi = mid - 1;  
        else if (key > a[mid]) lo = mid + 1;
```

```
        else  
            return mid;
```

```
    }
```

```
    return -1;
```

*return statement*

```
}
```

*loop statement  
(see page 15)*

*system calls main()*

*unit test client (see page 26)*

```
public static void main(String[] args)
```

```
{
```

*no return value; just side effects (see page 24)*

```
    int[] whitelist = In.readInts(args[0]);
```

```
    Arrays.sort(whitelist);
```

*call a method in a Java library (see page 27)*

```
    while (!StdIn.isEmpty())
```

*call a method in our standard library;  
need to download code (see page 27)*

```
    {
```

```
        int key = StdIn.readInt();
```

```
        if (rank(key, whitelist) == -1)  
            StdOut.println(key);
```

*call a local method  
(see page 27)*

```
    }
```

```
}
```

*conditional statement  
(see page 15)*

```
}
```

# Sort once; search many

- The “dictionary principle”:
  - In order to have maximum performance of searching, a collection of objects should be sorted\*;
  - This takes time, but it is only sorted once (at the beginning of the main program) while it can be searched almost an infinite number of times (the length of the input file).

\* It's a little bit like taking out insurance against a large number of searches: the cost of sorting is like paying the premium

# A few observations on Object-oriented programming

- Primitives and arrays are all very well and would be just fine if we only ever programmed things like the Newton Approximation (original version)\*.
- But object-oriented programming allows us to create our own data types and instantiate them as objects. We can put whatever combination we like of primitives in those types (which in Java are generally referred to as classes). But we can also include objects. And those objects can contain objects...
- Of course, O-O is much more than that: instance and class methods, instance and class fields, encapsulation, polymorphism, interfaces, abstract classes, inheritance hierarchies, Liskov substitution principle.

\* That's what all programming was like in the early days



# Testing

- One area I disagree strongly with the text book:
  - “A best practice in Java programming is to include a *main()* in every library of static methods that tests the methods in the library.”
  - No! All testing should be done via unit tests. In fact, best practice is to forget that there is such a thing as a static *main* method in Java. Never use one (in today’s world its rare that you are actually creating a *main* because most applications are run as some sort of plugin).
- Unit tests are also *specifications* and *documentation*!
- In fact, all of our assignments will come with unit tests.