



**Khoury
College**

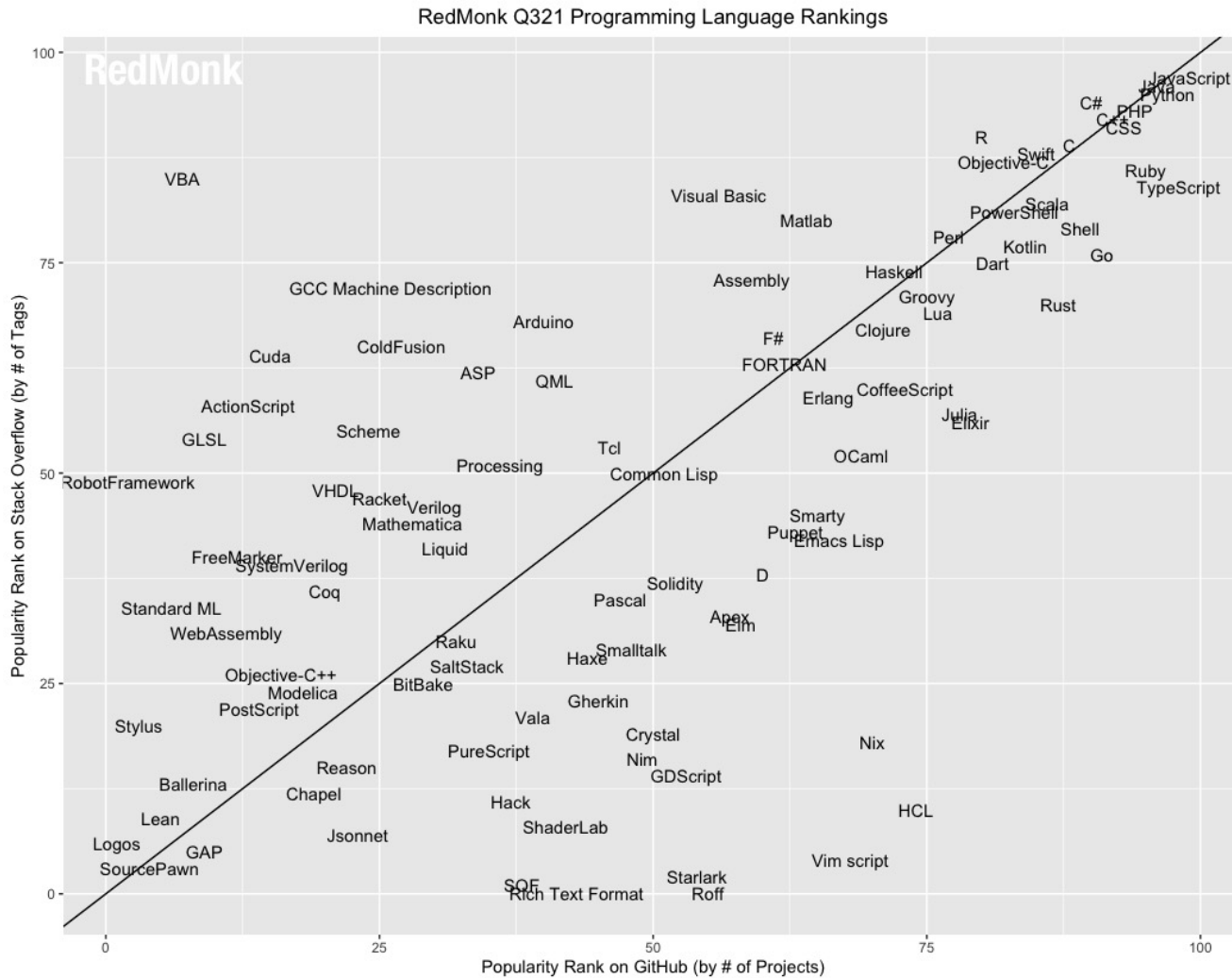
Lecture 1

CS5004 – Object-Oriented Design
Spring 2022

Why Java?





















- The answer is simple: Java is one of the most popular programming languages today.
- Java programmers are and will be in high demand.
- Let's take a look at some important indices from
 - RedMonk: we extract language rankings from GitHub and Stack Overflow, and combine them for a ranking that attempts to reflect both code (GitHub) and discussion (Stack Overflow) traction. The idea is to offer a statistically valid representation of current usage.
 - TOIBE: The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings.

Why Java?



Why Java?

Source: TOIBE

| Jan 2022 | Jan 2021 | Change | Programming Language | Ratings | Change |
|----------|----------|--------|--|---------|--------|
| 1 | 3 | ▲ |  Python | 13.58% | +1.86% |
| 2 | 1 | ▼ |  C | 12.44% | -4.94% |
| 3 | 2 | ▼ |  Java | 10.66% | -1.30% |
| 4 | 4 | |  C++ | 8.29% | +0.73% |
| 5 | 5 | |  C# | 5.68% | +1.73% |
| 6 | 6 | |  Visual Basic | 4.74% | +0.90% |
| 7 | 7 | |  JavaScript | 2.09% | -0.11% |
| 8 | 11 | ▲ |  Assembly language | 1.85% | +0.21% |
| 9 | 12 | ▲ |  SQL | 1.80% | +0.19% |
| 10 | 13 | ▲ |  Swift | 1.41% | -0.02% |
| 11 | 8 | ▼ |  PHP | 1.40% | -0.60% |
| 12 | 9 | ▼ |  R | 1.25% | -0.65% |
| 13 | 14 | ▲ |  Go | 1.04% | -0.37% |
| 14 | 19 | ▲▲ |  Delphi/Object Pascal | 0.99% | +0.20% |
| 15 | 20 | ▲▲ |  Classic Visual Basic | 0.98% | +0.19% |
| 16 | 16 | |  MATLAB | 0.96% | -0.19% |
| 17 | 10 | ▼▼ |  Groovy | 0.94% | -0.90% |
| 18 | 15 | ▼ |  Ruby | 0.88% | -0.43% |
| 19 | 30 | ▲▲ |  Fortran | 0.77% | +0.31% |
| 20 | 17 | ▼ |  Perl | 0.71% | -0.31% |

Computer Language Levels

- *High-level language*: A language that people can read, write, and understand
 - A program written in a high-level language must be translated into a language that can be understood by a computer before it can be run
- *Machine language*: A language that a computer can understand
- *Low-level language*: Machine language or any language similar to machine language (e.g. Assembly)
- *Compiler*: A program that translates a high-level language program into an equivalent low-level language program
 - This translation process is called *compiling*

Introduction To Java

- Most people are familiar with Java as a language for Internet applications
- We will study Java as a general-purpose programming language
 - The syntax of expressions and assignments will be similar to C/C++

Origins of the Java Language

- Created by Sun Microsystems team led by James Gosling in 1991 (now owned by Oracle)
- Originally designed for programming home appliances
 - Difficult task because appliances are controlled by a wide variety of computer processors
 - Team developed a two-step translation process to simplify the task of compiler writing for each class of appliances

Origins of the Java Language

- Significance of Java translation process
 - Writing a compiler for each type of appliance processor would have been very costly
 - Instead, developed intermediate language that is independent of the target processors : Java *byte-code*
 - Therefore, only a small, easy to write program was needed to translate byte-code into the machine code for each processor

Byte-Code and the Java Virtual Machine

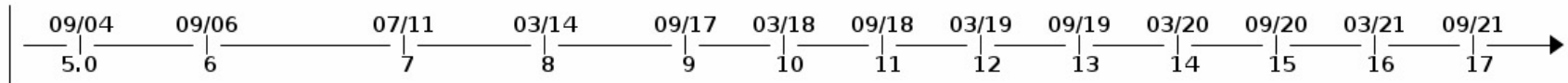
- The compilers for most programming languages translate high-level programs directly into the machine language for a particular computer
 - Since different computers have different machine languages, a different compiler is needed for each one
- In contrast, the Java compiler translates Java programs into *byte-code*, a machine language for a fictitious computer called the *Java Virtual Machine* (JVM)
 - Once compiled to *byte-code*, a Java program can be used on any computer, making it very portable

Byte-Code and the Java Virtual Machine

- *Interpreter*: The program that translates a program written in Java byte-code into the machine language for a particular computer when a Java program is executed
 - The interpreter translates and immediately executes each byte-code instruction, one after another
 - Translating byte-code into machine code is relatively easy compared to the initial compilation step
- Most Java programs today run using a Just-In-Time or JIT compiler which compiles a section of byte-code at a time into machine code

Java versions

- Java versions are released every 6 months.
 - For example, Java 18 is scheduled for March 2022, Java 19 for September 2022 and so on.
- In the past, Java release cycles were much longer:



For more information, check out the following link. This is where the timeline is coming from.

<https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features>

- We will focus on Java 11.

A Sample Java Application Program

Display 1.1 A Sample Java Program

```
1 public class FirstProgram
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello reader.");
6         System.out.println("Welcome to Java.");
7
8         System.out.println("Let's demonstrate a simple calculation.");
9         int answer;
10        answer = 2 + 2;
11        System.out.println("2 plus 2 is " + answer);
12    }
}
```

Annotations:

- ← Name of class (program) (points to `FirstProgram`)
- ← The main method (points to `main`)

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

Objects and Methods

- Java is an *object-oriented programming (OOP)* language
 - Programming methodology that views a program as consisting of *objects* that interact with one another by means of actions (called *methods*)
 - Objects of the same kind are said to have the same *type* or be in the same *class*

Terminology Comparisons

- Other high-level languages have constructs called procedures, methods, functions, and/or subprograms
 - These types of constructs are called *methods* in Java
 - All programming constructs in Java, including *methods*, are part of a *class*

Java Application Programs

- A Java *application program* or Java program is a class with a method named **main**
 - When a Java application program is run, the *run-time system* automatically invokes the method named **main**
 - All Java application programs start with the **main** method

System.out.println

- Java programs work by having things called *objects* perform actions
 - **System.out**: an object used for sending output to the screen
- The actions performed by an object are called *methods*
 - **println**: the method or action that the **System.out** object performs

System.out.println

- *Invoking or calling* a method: When an object performs an action using a method
 - Also called *sending a message* to the object
 - Method invocation syntax (in order): an object, a dot (period), the method name, and a pair of parentheses
 - Arguments: Zero or more pieces of information needed by the method that are placed inside the parentheses

```
System.out.println("This is an argument");
```

Variable declarations

- Variable declarations in Java are similar to those in other programming languages
 - Simply give the type of the variable followed by its name and a semicolon

```
int answer;
```

Using = and +

- In Java, the equal sign (=) is used as the *assignment operator*
 - The variable on the left side of the assignment operator is *assigned the value* of the expression on the right side of the assignment operator

```
answer = 2 + 2;
```

- In Java, the plus sign (+) can be used to denote addition (as above) or *concatenation*
 - Using +, two strings can be connected together

```
System.out.println("2 plus 2 is " + answer);
```

Compiling a Java Program or Class

- Each class definition must be in a file whose name is the same as the class name followed by **.java**
 - The class **FirstProgram** must be in a file named **FirstProgram.java**
- Each class is compiled with the command **javac** followed by the name of the file in which the class resides
 - javac FirstProgram.java**
 - The result is a byte-code program whose filename is the same as the class name followed by **.class**
FirstProgram.class

Running a Java Program

- A Java program can be given the *run command* (**java**) after all its classes have been compiled
 - Only run the class that contains the **main** method (the system will automatically load and run the other classes, if any)
 - The **main** method begins with the line:
public static void main(String[] args)
 - Follow the run command by the name of the class only (no **.java** or **.class** extension)
java FirstProgram

Syntax and Semantics

- *Syntax*: The arrangement of words and punctuations that are legal in a language, the *grammar rules* of a language
- *Semantics*: The meaning of things written while following the syntax rules of a language

Tip: Error Messages

- *Bug*: A mistake in a program
 - The process of eliminating bugs is called *debugging*
- *Syntax error*: A grammatical mistake in a program
 - The compiler can detect these errors, and will output an error message saying what it thinks the error is, and where it thinks the error is

Tip: Error Messages

- *Run-time error:* An error that is not detected until a program is run
 - The compiler cannot detect these errors: an error message is not generated after compilation, but after execution
- *Logic error:* A mistake in the underlying algorithm for a program
 - *The compiler cannot detect these errors, and no error message is generated after compilation or execution, but the program does not do what it is supposed to do*

Identifiers

- *Identifier*: The name of a variable or other item (class, method, object, etc.) defined in a program
 - A Java identifier must not start with a digit, and all the characters must be letters, digits, or the underscore symbol
 - Java identifiers can theoretically be of any length
 - Java is a case-sensitive language: **Rate**, **rate**, and **RATE** are the names of three different variables

Identifiers

- Keywords and Reserved words: Identifiers that have a predefined meaning in Java

- Do not use them to name anything else

`public` `class` `void` `static`

- Predefined identifiers: Identifiers that are defined in libraries required by the Java language standard

- Although they can be redefined, this could be confusing and dangerous if doing so would change their standard meaning

`System` `String` `println`

Naming Conventions

- Start the names of variables, classes, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters

`topSpeed` `bankRate1` `timeOfArrival`

- Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

`FirstProgram` `MyClass` `String`

Variable Declarations

- Every variable in a Java program must be *declared* before it is used
 - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
 - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
 - Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace {)
 - In Java, we have *primitive types* and *reference types*.
 - Some common primitive types are

```
int numberOfBeans;  
double oneWeight, totalWeight;
```

Primitive Types

Display 1.2 **Primitive Types**

| TYPE NAME | KIND OF VALUE | MEMORY USED | SIZE RANGE |
|----------------------|---|-------------|--|
| <code>boolean</code> | <code>true</code> or <code>false</code> | 1 byte | not applicable |
| <code>char</code> | single character (Unicode) | 2 bytes | all Unicode characters |
| <code>byte</code> | integer | 1 byte | −128 to 127 |
| <code>short</code> | integer | 2 bytes | −32768 to 32767 |
| <code>int</code> | integer | 4 bytes | −2147483648 to 2147483647 |
| <code>long</code> | integer | 8 bytes | −9223372036854775808 to 9223372036854775807 |
| <code>float</code> | floating-point number | 4 bytes | $-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$ |
| <code>double</code> | floating-point number | 8 bytes | $\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$ |

Assignment Statements With Primitive Types

- In Java, the assignment statement is used to change the value of a variable
 - The equal sign (=) is used as the assignment operator
 - An assignment statement consists of a variable on the left side of the operator, and an *expression* on the right side of the operator

`Variable = Expression;`

- An *expression* consists of a variable, number, or mix of variables, numbers, operators, and/or method invocations

```
temperature = 98.6;  
count = numberOfBeans;
```

Assignment Statements With Primitive Types

- When an assignment statement is executed, the expression is first evaluated, and then the variable on the left-hand side of the equal sign is set equal to the value of the expression

`distance = rate * time;`

- Note that a variable can occur on both sides of the assignment operator

`count = count + 2;`

- The assignment operator is automatically executed from right-to-left, so assignment statements can be chained

`number2 = number1 = 3;`

Tip: Initialize Variables

- A variable that has been declared but that has not yet been given a value by some means is said to be *uninitialized*
- In certain cases an uninitialized variable is given a default value
 - It is best not to rely on this
 - Explicitly initialized variables have the added benefit of improving program clarity

Tip: Initialize Variables

- The declaration of a variable can be combined with its initialization via an assignment statement

```
int count = 0;
```

```
double distance = 55 * .5;
```

```
char grade = 'A';
```

- Note that some variables can be initialized and others can remain uninitialized in the same declaration

```
int initialCount = 50, finalCount;
```

Shorthand Assignment Statements

- Shorthand assignment notation combines the *assignment operator* (=) and an *arithmetic operator*
- It is used to change the value of a variable by adding, subtracting, multiplying, or dividing by a specified value
- The general form is

Variable Op = Expression

which is equivalent to

Variable = Variable Op (Expression)

- The **Expression** can be another variable, a constant, or a more complicated expression
- Some examples of what **Op** can be are +, -, *, /, or %

Shorthand Assignment Statements

| Example: | Equivalent To: |
|---|---|
| <code>count += 2;</code> | <code>count = count + 2;</code> |
| <code>sum -= discount;</code> | <code>sum = sum - discount;</code> |
| <code>bonus *= 2;</code> | <code>bonus = bonus * 2;</code> |
| <code>time /= rushFactor;</code> | <code>time = time / rushFactor;</code> |
| <code>change %= 100;</code> | <code>change = change % 100;</code> |
| <code>amount *= count1 + count2;</code> | <code>amount = amount * (count1 + count2);</code> |

Assignment Compatibility

- In general, the value of one type cannot be stored in a variable of another type

```
int intVariable = 2.99; //Illegal
```

- The above example results in a type mismatch because a **double** value cannot be stored in an **int** variable

- However, there are exceptions to this

```
double doubleVariable = 2;
```

- For example, an **int** value can be stored in a **double** type

Assignment Compatibility

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it

byte→short→int→long→float→double
char _____↑

- Note that as you move down the list from left to right, the range of allowed values for the types becomes larger
- An explicit *type cast* is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., **double** to **int**)
- Note that in Java an **int** cannot be assigned to a variable of type **boolean**, nor can a **boolean** be assigned to a variable of type **int**

Literals

- A literal (aka constant) is a specific value of a type
 - Literals of an integer type may not be written with a decimal point (e.g., **10**). In Java ≥ 7 , Ints can have underscore to improve readability. E.g. 22_22_223
 - Literals of a floating-point type can be written in ordinary decimal fraction form (e.g., **367000.0** or **0.000589**)
 - Literals of a floating-point type can also be written in *scientific (or floating-point) notation* (e.g., **3.67e5** or **5.89e-4**)
 - Note that the number before the **e** may contain a decimal point, but the number after the **e** may not

Literals

- Literals of type **char** are expressed by placing a single character in single quotes (e.g., '**Z**')
- Literals for strings of characters are enclosed by double quotes (e.g., "**Welcome to Java**")
- There are only two **boolean** type literals, **true** and **false**
 - Note that they must be spelled with all lowercase letters

Arithmetic Operators and Expressions

- As in most languages, *expressions* can be formed in Java using variables, constants, and arithmetic operators
 - These operators are **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), and **%** (modulo, remainder)
 - An expression can be used anyplace it is legal to use a value of the type produced by the expression

Arithmetic Operators and Expressions

- If an arithmetic operator is combined with **int** operands, then the resulting type is **int**
- If an arithmetic operator is combined with one or two **double** operands, then the resulting type is **double**
- If different types are combined in an expression, then the resulting type is the right-most type on the following list that is found within the expression

byte→**short**→**int**→**long**→**float**→**double**
char _____↑

- Exception: If the type produced should be **byte** or **short** (according to the rules above), then the type produced will actually be an **int**

Parentheses and Precedence Rules

- An expression can be *fully parenthesized* in order to specify exactly what subexpressions are combined with each operator
- If some or all of the parentheses in an expression are omitted, Java will follow *precedence* rules to determine, in effect, where to place them
 - However, it's best (and sometimes necessary) to include them

Precedence Rules

Display 1.3 Precedence Rules

Highest Precedence

First: the unary operators: $+$, $-$, $++$, $--$, and $!$

Second: the binary arithmetic operators: $*$, $/$, and $\%$

Third: the binary arithmetic operators: $+$ and $-$

Lowest Precedence

Precedence and Associativity Rules

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

base + rate * hours is evaluated as

base + (rate * hours)

- When two operations have equal precedence, the order of operations is determined by *associativity* rules

Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left (right associative)

`+-+rate` is evaluated as `+ (- (+rate))`

- Binary operators of equal precedence are grouped left-to-right (left associative)

`base + rate + hours` is evaluated as

`(base + rate) + hours`

- Exception: Assignment operators is grouped right-to-left (right associative)

`n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3) ;`

Pitfall: Round-Off Errors in Floating-Point Numbers

- Floating point numbers are only approximate quantities
 - Mathematically, the floating-point number $1.0/3.0$ is equal to $0.3333333 \dots$
 - A computer has a finite amount of storage space
 - It may store $1.0/3.0$ as something like 0.3333333333 , which is slightly smaller than one-third
 - Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type
 $15.0/2$ evaluates to 7.5
- When both operands are integer types, division results in an integer type
 - Any fractional part is discarded
 - The number is not rounded $15/2$ evaluates to 7
- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

The % Operator

- The % operator is used with operands of type `int` to recover the information lost after performing integer division
 - `15/2` evaluates to the quotient `7`
 - `15%2` evaluates to the remainder `1`
- The % operator can be used to count by 2's, 3's, or any other number
 - To count by twos, perform the operation `number % 2`, and when the result is `0`, `number` is even

Type Casting

- A *type cast* takes a value of one type and produces a value of another type with an "equivalent" value
 - If **n** and **m** are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed
`double ans = n / (double)m;`
 - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
 - Note also that the type and value of the variable to be cast does not change

More Details About Type Casting

- When type casting from a floating-point to an integer type, the number is truncated, not rounded
 - `(int)2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*

```
double d = 5;
```

- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast

```
int i = 5.5; // Illegal
```

```
int i = (int)5.5 // Correct
```

Increment and Decrement Operators

- The *increment operator* (**++**) adds one to the value of a variable
 - If **n** is equal to **2**, then **n++** or **++n** will change the value of **n** to **3**
- The *decrement operator* (**--**) subtracts one from the value of a variable
 - If **n** is equal to **4**, then **n--** or **--n** will change the value of **n** to **3**

Increment and Decrement Operators

- When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
 - If **n** is equal to **2**, then **2* (++n)** evaluates to **6**
- When either operator follows its variable, and is part of an expression, then the expression is evaluated using the original value of the variable, and only then is the variable value changed
 - If **n** is equal to **2**, then **2* (n++)** evaluates to **4**