
Programming
Fundamentals II
Course Logistics and Introduction

In this course, we focus less on problem solving…

… and more on program design

We run programs as systems of interacting objects

Object-Oriented Programming (OOP)

We will also discuss some of the more advanced concepts

Generics

Concurrency

Here's the OUTLINE

- Java basics:
  - Values, types, variables, expressions, and operators
  - Control structures
  - Collections
- Classes and objects
- Methods
- Inheritance
- Polymorphism
  - Abstract classes and interfaces

- Object design
- Unit testing and debugging
- Exceptions
- Generics
- Asynchronous programming and concurrency

We will use Java in this course… Why?

- Java is a **widely-used enterprise-grade** programming language with an OOP-centric design
  - First appeared in 1995, it is designed by James Gosling and developed by Sun Microsystems (now part of Oracle Corp.)

- The language is relatively simple and well-suited for **illustrating OOP** concepts

- Java syntax derives from C-family languages – transition to C, C++, C#, and others will be easier

Why not C++ or C#?

- C# is also a widely-used enterprise-grade OOP-centric language
- C++ is used a lot where high performance is crucial, e.g., real-time systems, operating systems, and game development

- They are similar on the surface, but Java is conceptually simpler

- Skills learned from Java will help ease the transition to C# or C++ if needed in the future

What we need to learn Java

- An editor (we recommend Atom for simplicity, but you may use a full-blown IDE, e.g., Eclipse, NetBeans, IntelliJ IDEA, or **VS Code**)
- Java Development Kit (JDK)
  - https://www.oracle.com/java/technologies/downloads/#java19
  - JDK 8 or higher is required for this course

What is a JDK?

- Java Development Kit is a package containing tools for developing Java-based software
- JDK contains a Java **compiler**, a **debugger**, Java **Virtual Machine** (JVM), **runtime environment**, **standard libraries**, and all other stuff needed to develop and deploy a Java application

- And what is a JRE?
  - Java Runtime Environment is a package required to run a Java program
  - It includes JVM and libraries, but not development tools
  - **JDK already includes JRE**

Java SE, ME, and EE

- Java Platform, **Standard Edition (Java SE)** is what we use here

- Java Platform, **Micro Edition** (Java ME) is targeted at **embedded** and mobile devices
  - Android devices are not based on Java ME – they use Java, but with their own runtime system and support platform

- Java Platform, **Enterprise Edition** (Java EE) is Java SE **plus an application server specification** designed for developing Enterprise-class Java applications

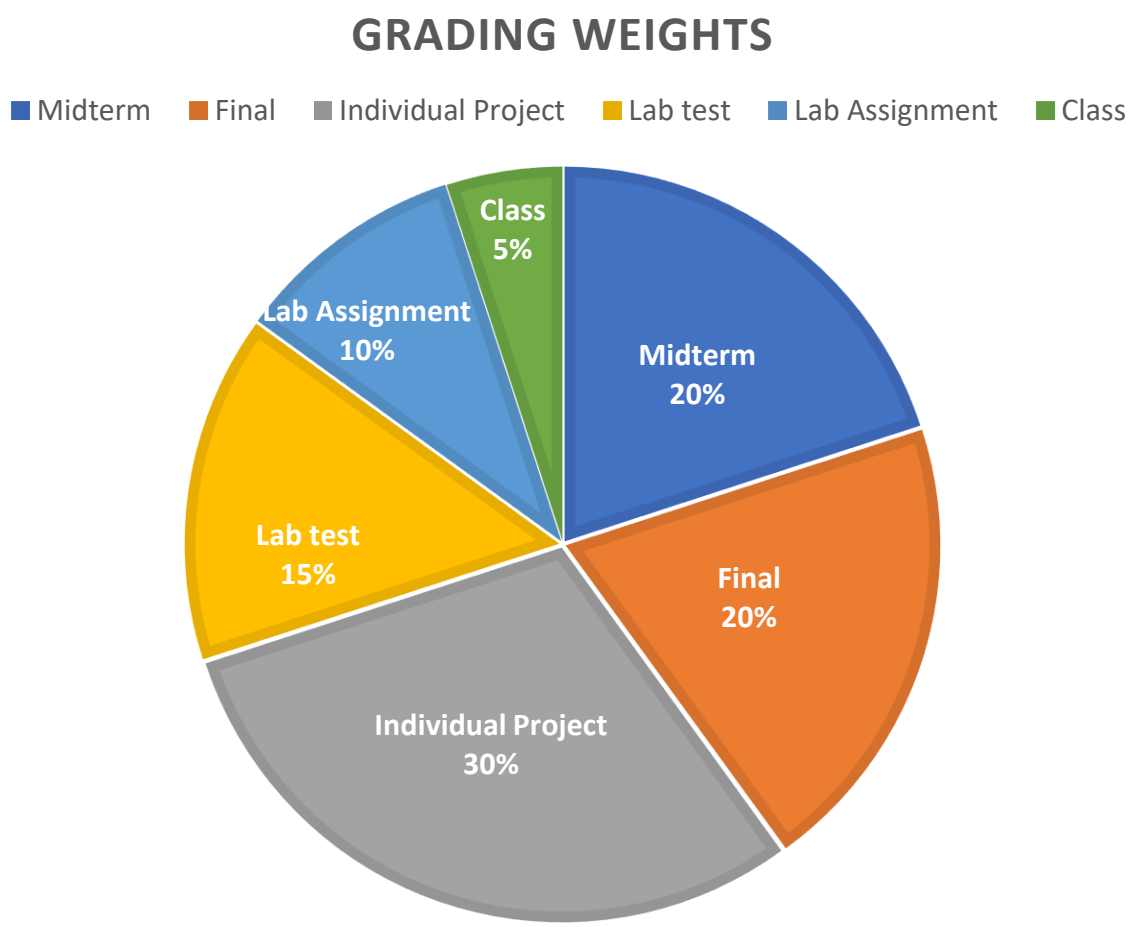
Additional resources

- Java 9 for Programmers, 4th Edition, Paul Deitel and Harvey Deitel
- Java: A Beginner's Guide, 8th Edition, Herbert Schildt

- Drafts of Java lecture notes: https://github.com/Poonna/java-book
  - Corrections and suggestions are welcome if you find any errors or issues

- There are a number of Java textbooks available in Thai language. However, be careful in choosing one for reference as a few of them are outdated, and/or contain inaccurate information and bad programming practices.

Course logistics



Grading Scale

A	80 - 100%
B+	75 - 79%
B	70 - 74%
C+	65 - 69%
C	60 - 64%
D+	55 - 59%
D	50 - 54%
F	< 50%

Code of ethics (EXTREMELY IMPORTANT)

- We take academic honesty extremely seriously.
- Do you own work. Do not cheat, copy, or claim other people's work as your own. This includes all homework, assignments, and exams.
- VIOLATION of the above rule WILL NOT BE TOLERATED and will automatically award you with the F grade for this course.
  - (Yes, homework included. No plagiarism. At all.)

- Cheating is unfair to students who work diligently.
- Besides, you will not gain any skills or understanding by doing so.

Did I say we take academic honesty extremely seriously?

Good. Now you remember.

That said, peer discussion is encouraged

- You are encouraged to discuss homework and assignments with your friends, and to collectively work out solutions to the problems
- You are encouraged to teach and guide your friends
  - Be careful. Guide them, not do it for them! Try to not touch your friends' computers. Just verbally explain to them.
- You should bring your own ideas to the discussion, i.e., not joining the discussion just to get ready-made solutions without any contributions

- The final work – the coding, however, must be done by you
  - No, not just typing code – this includes translating ideas into code

Did I say we take academic honesty extremely seriously?

If you think it’s a joke, stop. Drop the course. Return when you think you’re ready to take it seriously.

Don’t plagiarize. Just don’t.

Well, let’s begin!

Our first Java program

```
class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

What!?

The same code in Python is just one line!

```
print("Hello, world!")
```

Running a Java program

- Contrary to Python, Java code (like the one on the previous page) cannot be run directly

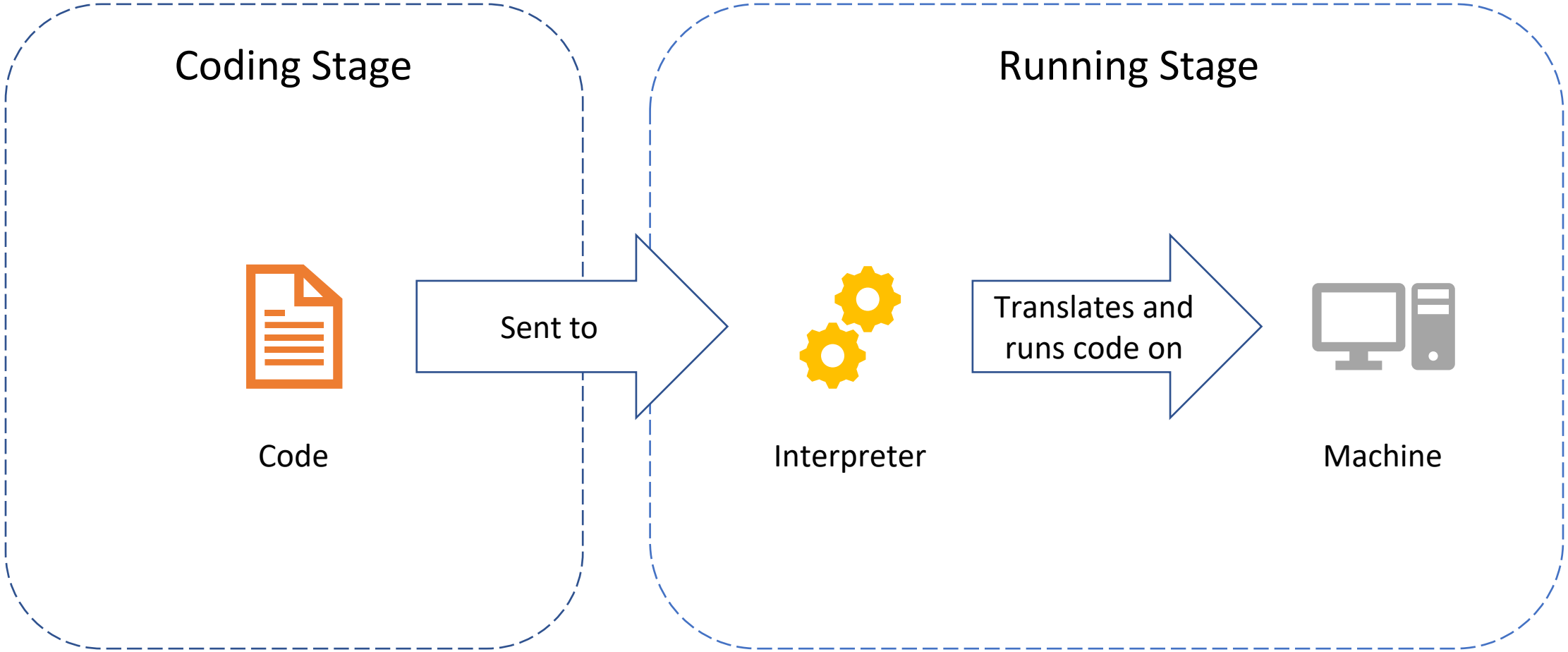
- We need to translate it first

There are 2 models of program translation

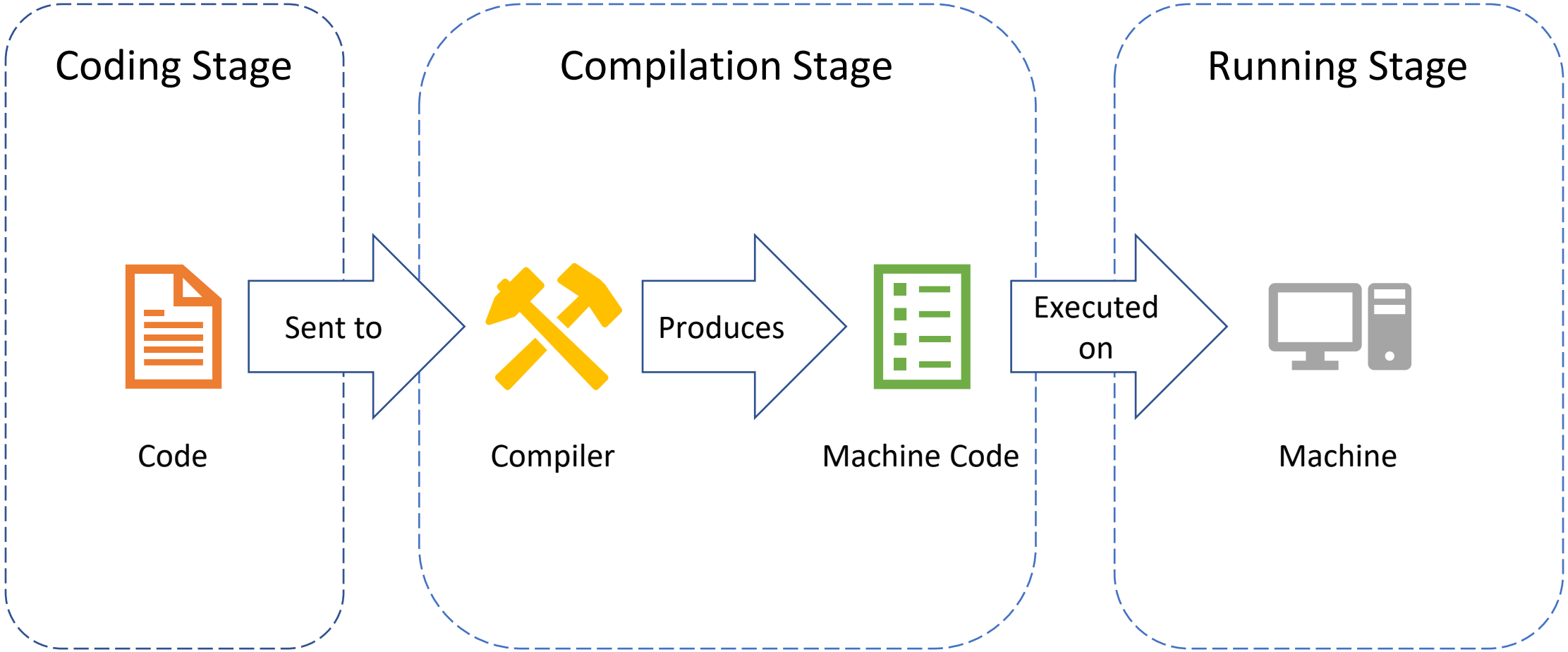
Interpretation

Compilation

Interpreted languages run code directly



Compiled languages need a compilation stage



In the real world, it is not so clear-cut

- Python is generally an interpreted language, but can also be compiled

- Java is a compiled language, but produces Java bytecode instead of machine code
  - The Java bytecode is subsequently translated to machine code by Java Virtual Machine (JVM) at run time
  - The method in which the bytecode is translated is a hybrid of interpretation and compilation
  - There are techniques like Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation that JVM uses to translate bytecode

Back to our code

```
class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Save this code to a file and name it: MyClass.java

Note that Java requires that the file name matches the ***public*** **class** name in the file

* This code does not have a public class – we just want to name the file this way

We then compile the code

- Compiling a Java source file (.java) translates it to a class file (.class) which contains JVM bytecode instructions

- Type this command on the shell/cmd/Terminal prompt:

```
javac MyClass.java
```

- ‘javac’ is the **Java compiler** which will produce the file MyClass.class as its translation result

And run it

- Type this:

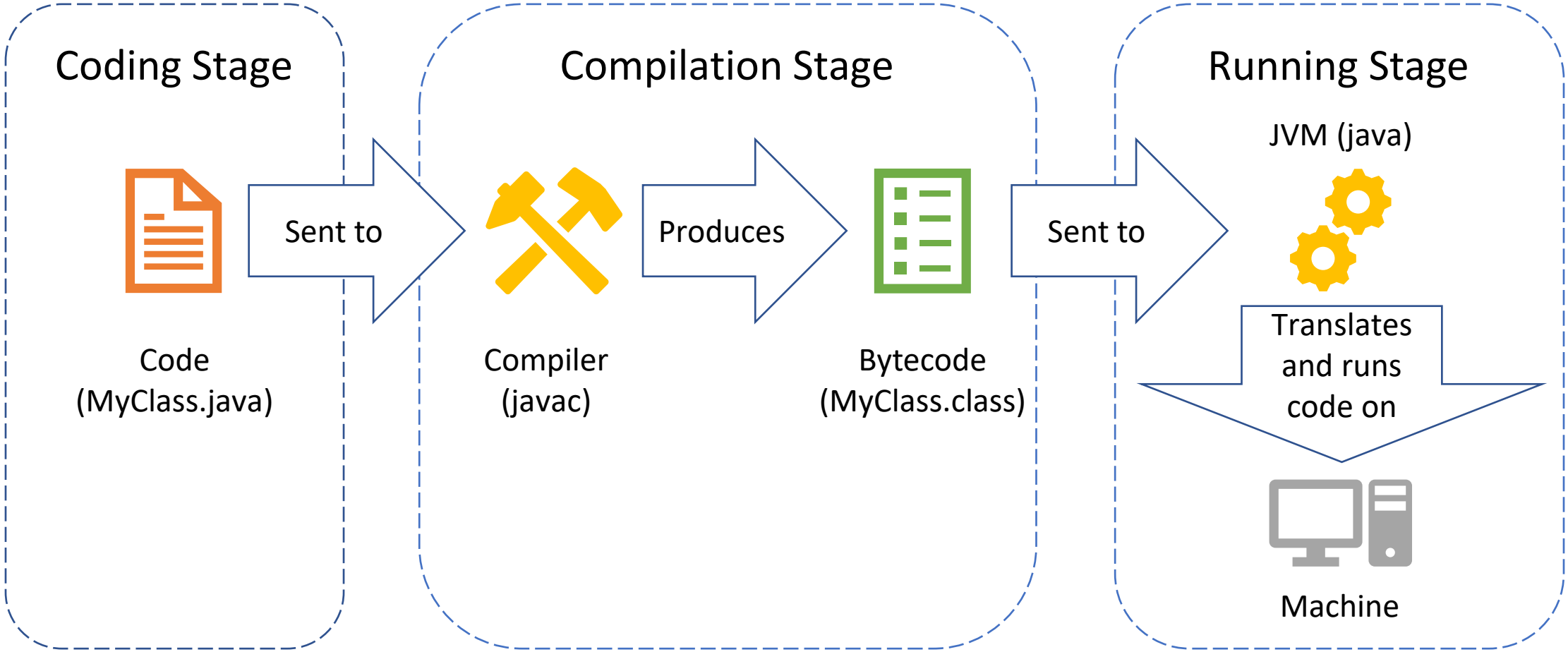
```
java MyClass
```

- The command **‘java’ invokes Java Virtual Machine (JVM)** to translate and run the bytecode on the machine
  - Notice that we use MyClass here, not MyClass.class, because JVM takes class name, not file name

- Here’s the result:

```
Hello, world!
```

Java’s compiled code does not run directly



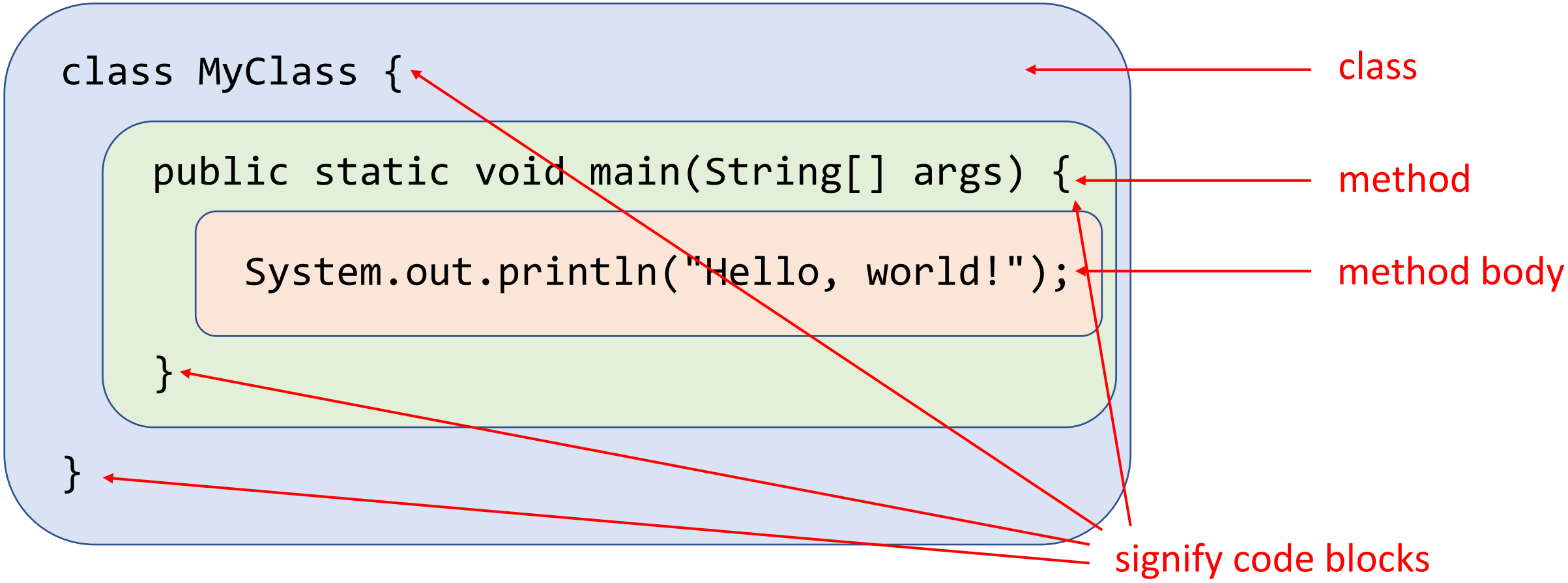
Why does Java need JVM to run code?

- C++ compiler, for example, produces machine code that can run directly on the machine. Why doesn't Java do the same?

- Java's original vision: **"Write Once, Run Anywhere"**

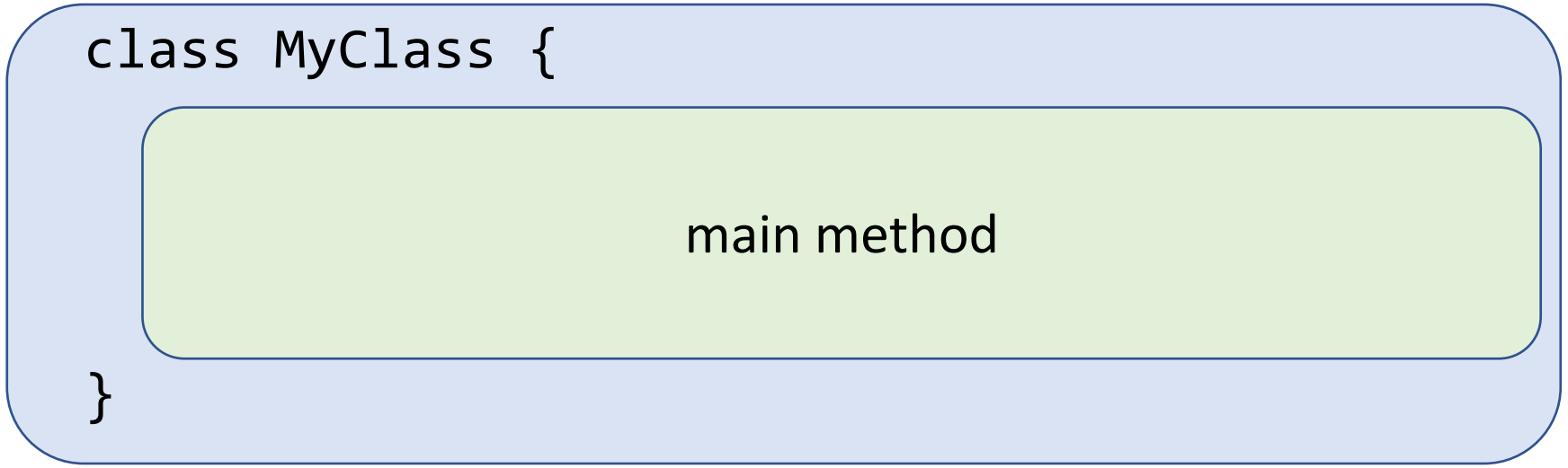
- Java bytecode can (at least in theory) run on any machine, as long as it has JVM installed
  - C++ compiled binary code can run only on similar systems
  - JVM is packaged in JRE and JDK

Now, let's dissect the code



Java programs are built from classes

- A program must have at least one class (it can have more)
- Java code (except imports and comments) must reside within a class



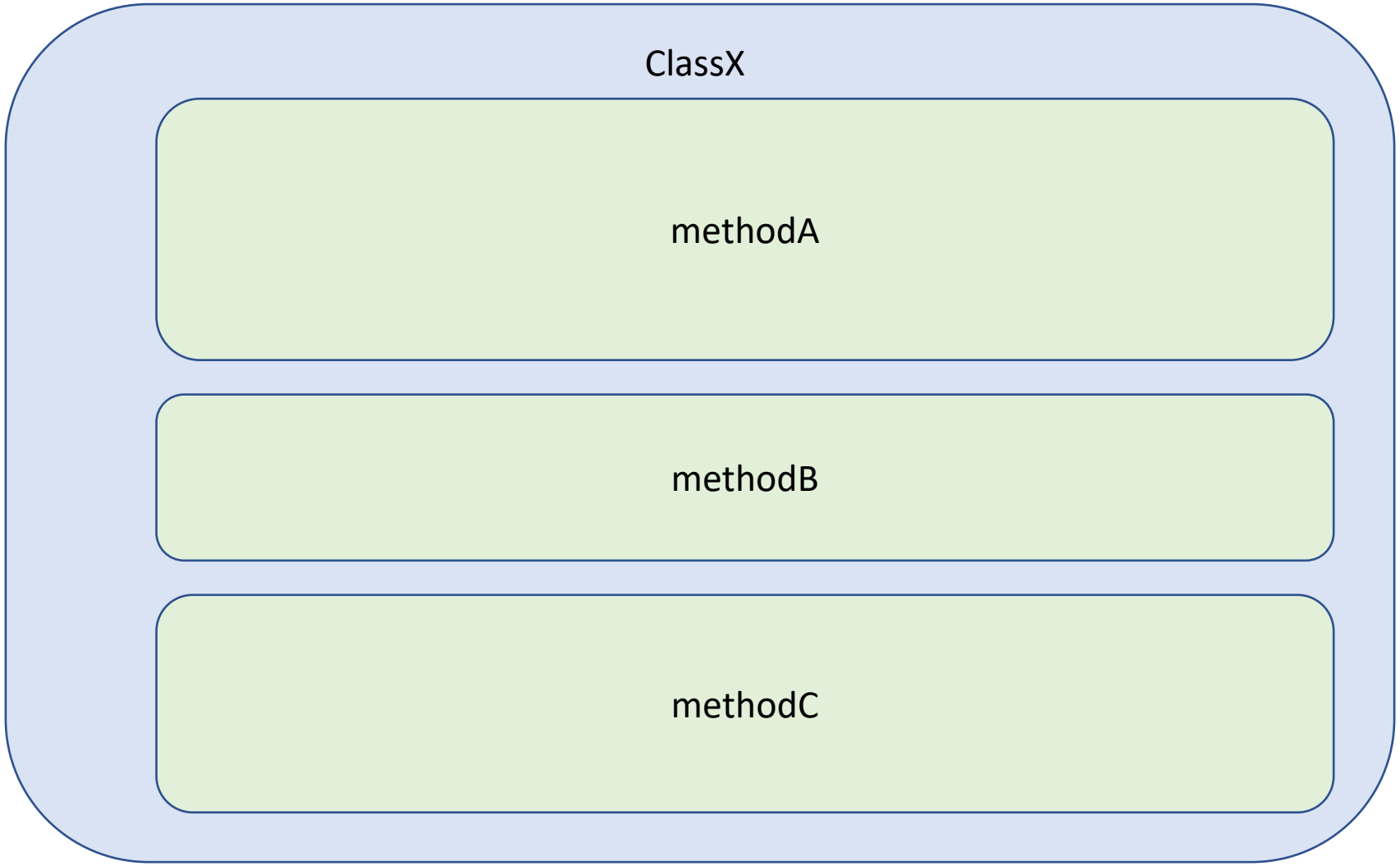
A class specifies methods and attributes

- For our purpose, we will for now look at a class simply as a syntactic container that encloses its methods and attributes (variables)

- More formally, a class is a specification for objects that specifies what the objects will have (their attributes) when they are created and how they will behave (their methods)

- We will discuss the actual class concept in detail later

A class may contain many methods



Java naming convention: **classes**

- Java classes are named using Pascal case
- Pascal case **capitalizes the first** letter of every word
- Underscore (_) is not used
- Conventionally correct:

```
ClassA          X              CarEngine1500      Dog
LineItem        HttpResponse   ThisClassDoesNotBelongHere
```

- Conventionally incorrect:

```
Class_A         CLASSX         Carengine1500      dog
lineItem        HTTPResponse   this_class_belongs_here
```

Convention is not a rule

- Java naming rules:
  - A valid name starts with a letter or dollar sign ($) or underscore (_), followed by zero or more letters , dollar signs, underscores, or digits (0-9)
  - You cannot use spaces in the name
  - You cannot use reserved words as names

- Java naming convention:
  - We will discuss common naming practices as we go
  - It’s not illegal to not follow the convention – it’s just that your code will look out of place
  - So, don’t break the convention unless you have good reasons

Are these names correct?

Name	By Rule	By Convention
`1ClassOnly`	✗	✗
`_Thi$_c14$$_i$_c001_`	○	✗
`Type13Account`	○	○
`MasterController`	○	○
`accessControlList`	○	✗
`HTMLReader`	○	✗
`X`	○	○

Method is a function, just within a class

- You can think of a method as a function that belongs to a class (or objects of that class)

- Now, this declaration looks really wordy:

```
public static void main(String[] args) {
  …
}
```

- We will take it apart next

```
public static void main(String[] args) {
    // Your code here
}
```

- The method name is "`main`"
  - "`main`" method is the **first method that will be executed**

- `(String[] args)` is the parameter specification
  - There is only one parameter here, and its name is "`args`"
  - The type of the parameter is a `String` array (`String[]`)

- Before the method name is the type of the return value
  - **"`void`"** means that the method **does not return a value**

```
public static void main(String[] args) {
    // Your code here
}
```

- "public" means that the method is **accessible to any classes**

- "static" means that the **method belongs to the class** (not the objects)
  - Static methods **are callable without creating objects**
  - "`main`" method **must be static**, as it is the creator of all objects
    - If `main` were not static (in other words, belonging to objects), it could only be invoked (called) through an object
    - **In the beginning there were no objects**, so non-static `main` could not be invoked – therefore, `main` must be static

Don’t worry if some concepts are unclear to you now.
It will become clearer as we progress through the OOP topics.

For now, just remember the structure and go ahead with it.

Inside the main method

```
System.out.println("Hello, world!");
```

- Every statement ends with a semicolon **(;)**
- `println` is a built-in method for printing a string on the screen
  - It takes one parameter
  - It goes to the next line after printing

- Use `print` if you want to stay on the same line

Okay, what is System.out then?

- `System` is a pre-defined class that handles system tasks
- `out` is a static variable in the System class
  - Its type is `PrintStream` class

- Together, `System.out` represents the standard output stream (screen output)
  - And `println` is a method of the `PrintStream` class

- There are also `System.in` that represents the standard input stream (keyboard input) and `System.err` that represents the standard error stream (also screen output , but for error messages)

Again, don’t worry if some concepts are unclear to you now.

Line comments and block comments

- Java (like C) uses `//` for line comments and `/* … */` to encloses multi-line (or block) comments

- There is also another form of comments in Java called Javadoc
  - It's of this form: `/** … */`
  - Used when writing API documentation for your code

Now, with comments

```
/*
 * This is the greatest program ever!
 * And it’s written by me!!!
 */
class MyClass {
    // Program starts here
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // Print it!
    }
}
```

Only one method today

Today, we will concern ourselves only with programs that contain one method: the main method

We focus only on the code in the method body for now

Let’s do some I/O

```
import java.util.Scanner;

class NameAndAge {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Name: ");
    String name = sc.nextLine();

    System.out.print("Age: ");
    int age = sc.nextInt();

    System.out.println(name + " is " + age + " years old.");
  }
}
```

Running result

```
Name: John
Age: 10
John is 10 years old.
```

```
Scanner sc = new Scanner(System.in);

System.out.print("Name: ");
String name = sc.nextLine();

System.out.print("Age: ");
int age = sc.nextInt();

System.out.println(name + " is ”
    + age + " years old.");
```

We use Scanner class to get user input from the keyboard

```
import java.util.Scanner;

class NameAndAge {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Name: ");
    String name = sc.nextLine();

    System.out.print("Age: ");
    int age = sc.nextInt();

    System.out.println(name + " is " + age + " years old.");
  }
}
```

1. We need to import it first – Scanner belongs to package java.util

2. Then we create a Scanner object, specifying System.in as the source

3. We declare variable sc to hold the Scanner object

4. We can then call the methods nextLine and nextInt using the Scanner object sc to obtain user inputs

There are variable declarations in the code

```
class NameAndAge {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Name: ");
    String name = sc.nextLine();

    System.out.print("Age: ");
    int age = sc.nextInt();

    System.out.println(name + " is " + age + " years old.");
  }
}
```

Java **needs type** declarations everywhere!

- This is different from a dynamic language like Python

- Variables must be declared with their associated types
- Methods' specifications include parameters' types and return types

- Note: Java 10 has local variable type inference – you don't need to explicitly specify types for local variables
  - The compiler will infer the correct types from the context
  - Java still needs to know exactly what type a variable is, even if by inference

Variables must be declared before use

- Declaration is of one of the following forms:

```
<type> <name>;
<type> <name> = <initializer>;
```

- Examples:

```
double cash;
String name = "John”;
int dayOfMonth = sc.nextInt();
```

- Variable types are fixed – **they cannot be changed once declared**
  - Java is statically-typed, unlike Python which is dynamically-typed

What’s with the typing?

Static typing (Java and others)

- Type of a variable is fixed
- Type is determined at declaration
- We can only assign a value of a compatible type to a variable

Dynamic typing (Python and others)

- Type of a variable is dynamic
- Type is determined by its current value
- We can assign any value of any type to a variable

Naming convention: **variables**

- Java variables are named using **camel case**
- Camel case capitalizes the first letter of every word after the first
  - **The first word is lower-case**
- Underscore (_) is not used
- Conventionally correct:

```
varA                x                carEngine1500       dog
lineItem            httpResponse     thisVarDoesNotBelongHere
```

- Conventionally incorrect:

```
var_A               Variablex        carengine1500       Dog
LineItem            HTTPResponse     this_var_belongs_here
```

Are these names correct?

Name	By Rule	By Convention
`aVarName`	⭕	⭕
`sum_of_products`	⭕	❌
`http404Handler`	⭕	⭕
`MasterController`	⭕	❌
`$accessControlList`	⭕	❌
`2xValue`	❌	❌
`public`	❌	N/A

Java primitive types: integer types

- `int`
  - Size: 32 bits, Min: `-2_147_483_648`, Max: `2_147_483_647`
  - Example literals: `0`, `1`, `-20`, `1500`, `23_200`

- `byte`, `short`, `long`
  - All are integer types, with word lengths of **8, 16, and 64 bits, respectively**
  - `long` literals: `0L`, `1L`, `-20L`, `1_500L`, `23200L`

- Integer literals with bases
  - `0b0101` (binary), `0764` (octal), `0xAF01` (hexadecimal)

Java primitive types: floating-point types

- `float`
  - Size: 32 bits, Min: $\pm 10^{-38}$, Max: $\pm 10^{38}$, **Significant digits: 6-7**
  - Example literals: `-1.0f`, `0.49f`, `3E8f`, `1.24E-10f`

- `double`
  - Size: 64 bits, Min: $\pm 10^{-308}$, Max: $\pm 10^{308}$, **Significant digits: 15-16**
  - Example literals: `-1.0`, `0.49`, `3E8`, `1.24E-10`

- Floating-points are imprecise – **avoid direct comparison (using ==)**

Java primitive types: boolean and char types

- `boolean`
  - Size: 1 bit
  - Example literals: `false`, `true`

- `char` (a UTF-16 Unicode character)
  - Size: 16 bits, Range: `'\u0000'` – `'\uFFFF'`
  - Example literals: `' '`, `'0'`, `'A'`, `'a'`, `'ก'`, `'+'`, `'\u0040'`

There are also reference types

- Reference types include:
  - Classes
  - Interfaces
  - Arrays
  - Enums

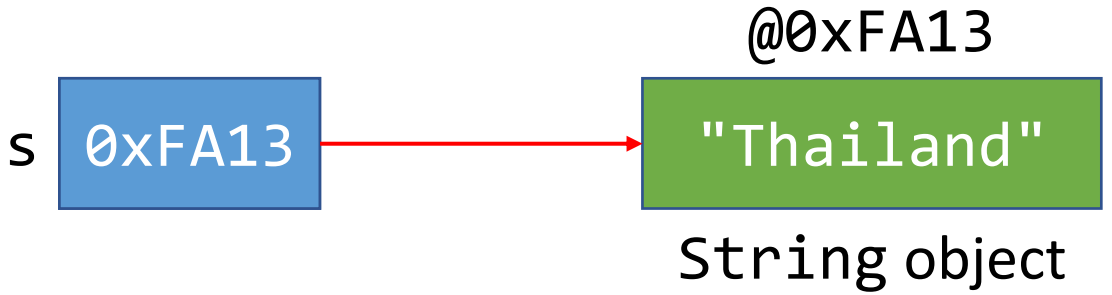
- A value of reference type is **either null or a reference to an object or array**

Primitive types vs reference types

- Variables of primitive types store the values
- Variables of reference types store references to values

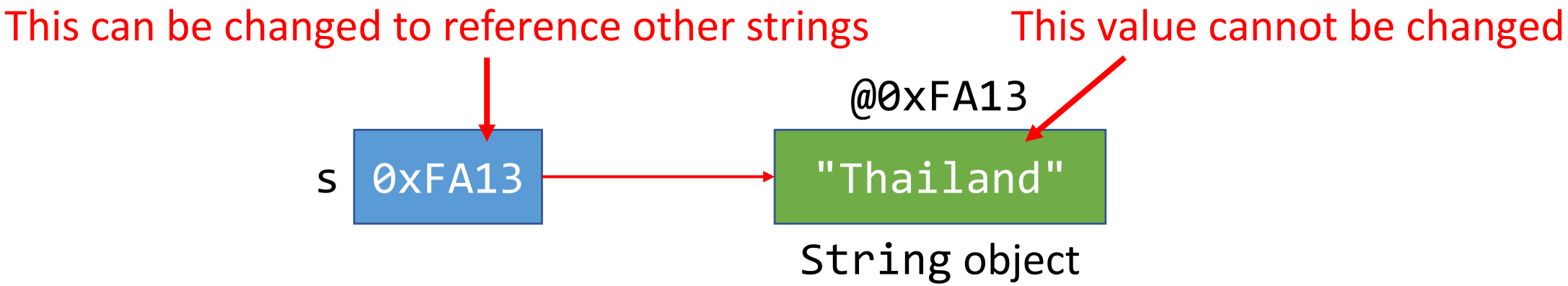
```
int a;
a = 10;
```


```
String s;
s = "Thailand";
```



Java reference types: String

- A string is actually an object of class `String`
- Example literals:
  - `""`, `"123"`, `"Welcome to \"Thailand\""`, `"ฉันรักจาวา (จริง ๆ นะ)"`
- A String object is immutable, but a String variable is not



What operators can I use?

- Usual arithmetic operators are supported
  - +, -, *, /, %
  - There's no power operator
- Like most C-family languages, unary ++ and -- are supported
  - ++ is pre-/post-increment
  - -- is pre-/post-decrement
- We will introduce more along the way

Examples of Java expressions

```
7 + 2 * 11 % 12

5 / 2.0 + 5 / 2

"Hi, " + "friends"

3 + 2 + " equals " + 3 + 2

3 + 2 + " equals " + (3 + 2)
```

```
17 (int)

4.5 (double)

"Hi, friends" (String)

"5 equals 32" (String)

"5 equals 5" (String)
```

You've seen type coercion in action

- When operands' types are different, one of the operand will **automatically be promoted to the larger/wider type (if possible)**
  - Roughly, `byte` → `short` → `int` → `long` → `float` → `double`
  - Conversion from a large `int` to `float`, or a large `long` to `float` or `double` is potentially lossy

- **Narrowing conversion** requires explicit casting
  - For example, from long to int or from double to long
  - `15 / (int)5.5` yields 3 of `int` type

5 / 2.0 + 5 / 2

1. 5 / 2.0
   - 5 is `int` and 2.0 is `double`
   - 5 is promoted to `double`
   - Result is 2.5
2. 5 / 2
   - Both are `int`, so no conversion is needed
   - Result is 2
3. 2.5 + 2
   - 2.5 is `double` and 2 is `int`
   - 2 is promoted to `double`
   - Result is 4.5

3 + 2 + " equals " + 3 + 2

1. 3 + 2
   - Both are `int`, and the result is 5
2. 5 + " equals "
   - 5 is `int` and " equals " is `String`, so 5 is converted to `String`
   - Result is "5 equals "
3. "5 equals " + 3
   - "5 equals " is `String` and 3 is `int`, so 3 is converted to `String`
   - Result is "5 equals 3"
4. "5 equals 3" + 2
   - Likewise, this results in "5 equals 32"

Mixing incompatible types results in type errors

```
int a = 100;
boolean b = true;
String s = "Fix me for ";

// This is OK
s = s + a;          // String coercion can be used in expression

// These produce type errors
String t = 12345; // int cannot be assigned to a String variable
int u = "12345";  // String cannot be assigned to an int variable
int v = a + b;    // boolean cannot be converted to int
int w = a + 500L; // Narrowing conversion needs explicit casting
```

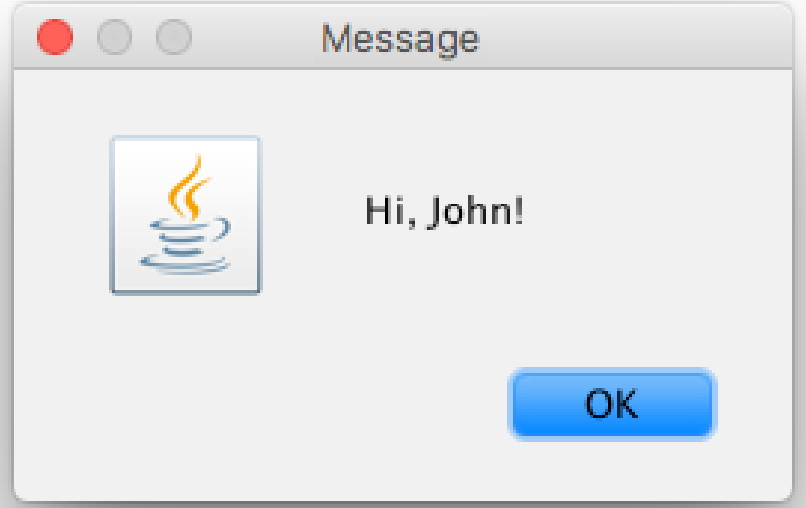
Let’s play with some GUI dialogs

```
import javax.swing.JOptionPane;

class DialogExample {
  public static void main(String[] args) {
    // Read a name
    String name = JOptionPane.showInputDialog(null,
        "Enter your name:");

    // Show a greeting message
    JOptionPane.showMessageDialog(null, "Hi, " + name + "!");
  }
}
```


Input
Enter your name:
John
Cancel
OK


Message
Hi, John!
OK

JOptionPane offers ready-made dialogs

- First, import JOptionPane from the Swing library

```
import javax.swing.JOptionPane;
```

- Use showMessageDialog() to show a message

```
JOptionPane.showMessageDialog(null, "Hi, " + name + "!");
```

- Use showInputDialog() to receive text input (result is a String)

```
String name = JOptionPane.showInputDialog(null, "Enter your name:");
```

- The first parameter is the parent frame
  - "`null`" means default parent – dialogs will show up at the center of the screen

Scanner’s pitfall

```
System.out.print("Number: ");
int number = sc.nextInt();
System.out.print("Name: ");
String name = sc.nextLine();

System.out.println("Result: \"" + name + " got "
    + number + "\"");
```

You expect:

```
Number: 123
Name: John
Result: "John got 123"
```

What you actually get:

```
Number: 123
Name: Result: " got 123"
```

Understanding text input buffer

- When you enter “123”, this is what it actually looks like in the buffer:


```
int number = sc.nextInt();    // Read an integer
String name = sc.nextLine(); // Read until next line
```

These methods **do not** flush the new line

- `nextInt()     // Read next integer`
- `nextDouble() // Read next double`
- `next()        // Read next word (String)`

- Actually, any `nextXXX()` except `nextLine()`

What can you do about it?

- Use `nextLine()` to flush the new line character before reading the next whole-line input
  - Basically, use `nextLine()` twice if the previous input method is not `nextLine()`

```
System.out.print("Number: ");
int number = sc.nextInt();
sc.nextLine(); // Flush the new line character
System.out.print("Name: ");
String name = sc.nextLine();
```

Question: Why does this one work fine?

```
System.out.print("Name: ");
String name = sc.nextLine();

System.out.print("Age: ");
int age = sc.nextInt();

System.out.println(name + " is " + age + " years old.");
```

Summary

- Java is a statically-typed object-oriented language
- Java is a compiled language, but needs JVM to run
- A Java program consists of one or more classes
- A method is a function that belongs to a class or objects
- A Java program starts at the `main` method
- Variables need to be declared before use
- Class naming should follow Pascal case convention, while method and variable naming should follow camel case convention

Summary

- Primitive types include `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`
- Variables of primitive types store the values
- Variables of reference types store references to the values
- Most of the widening type conversion are automatic (implicit)
- Narrowing type conversion needs explicit casting

- Don’t cheat in this course – the consequence is not worth the risk
- Discussion and consulting are allowed, but coding is your job