A First Taste of Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	1
Lecture URL	https://youtu.be/ULpvhw2hmSg
Notion URL	https://21f1003586.notion.site/A-First-Taste-of-Java-c8303394228e4344969df9d13f8c4bd0
Week #	2

Getting Started

- In Python

```
print('hello, world')
```

- In C ...

```
#include <stdio.h>
main() {
  printf("hello, world\n");
}
```

- And then there is Java ...

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

Why so complicated?

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

- All code in Java lies within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility
- How does the program start?
  - Fix a function name that will be called by default
  - From C, the convention is to call this function `main()`
- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameters is an array of strings
    - Command line arguments
  - No output, hence the `main` function has the return type `void`
- Visibility
  - Function has to be available to run from outside the class
  - Modifier `public`
- Availability
  - Functions defined inside classes are attached to the objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects, *belongs to the class*
- The actual operation
  - `System` is a public class
  - `out` is a ***stream*** object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`
    - Adds a newline character `\n` at the end of the statement
- Punctuation `{`, `}`, `;` to delimit the blocks, statements
  - Unlike layout and indentation in Python

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`
- Java programs are usually interpreted on ***Java Virtual Machine (JVM)***
  - JVM provides a uniform execution environment across OSes
  - Semantics of Java is defined in terms of JVM, OS-independent
  - It came with a slogan “Write once, run anywhere”
- `javac` compiles into JVM ***bytecode***
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`
- Note:
  - `javac` requires file extension `.java`
  - `java` should not be provided file extension `.class`

    - `javac` automatically follows dependencies and compiles all the classes required
        - Sufficient to trigger compilation for class containing `main()`

Summary

- The syntax of Java is comparatively heavy
- Many modifiers — unavoidable overhead of object-oriented design
    - Visibility: `public` and `private`
    - Availability: all functions live inside objects, need to allow `static` definitions
    - Will see more modifiers as we go along
- Functions and variable types have to be declared in advance
- Java compiles into code for a virtual machine
    - JVM ensures uniform semantics across operating systems
    - Code is guaranteed to be portable

Basic Datatypes in Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	2
Lecture URL	https://youtu.be/IwTdVsLxz04
Notion URL	https://21f1003586.notion.site/Basic-Datatypes-in-Java-3cfdcb7953c14d4890ccc104b8264b26
Week #	2

Scalar Types

- In an object-oriented langauge, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has 8 primitive datatypes
  - `int, long, short, byte`
  - `float, double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
int	4
long	8
short	2
byte	1
float	4
double	8
char	2
boolean	1

- 2-byte `char` for Unicode

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
float y;
char c;
boolean b1;
```

- The assignment statement works as usual

```
int x, y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)
  - Double quotes mark string

```
char c, d;
c = ‘x’;
d = ‘\u03C0’;  // Greek pi, unicode
```

- Boolean constants are `true, false`

```
boolean b1, b2;
b1 = false;
b2 = true;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
float y;
```

  - Use this judiciously to retain readability
- Initialize at the time of declaration

```
int x = 10;
float y = 5.7;
```

- Modifier `final` marks as constant

```
final float pi = 3.1415927f;
pi = 22/7;  // Flagged as error
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+, -, *, /, %`
- No separate integer division operator `//`

- When both arguments are integer, `/` is integer division

  ```
  float f = 22/7;  // Value is 3.0
  ```

  - Note implicit conversion from `int` to `float`
- No exponentiation operator, use `Math.pow()`
- `Math.pow(a, n)` returns a^n
- Special operators for incrementin and decrementing integers

  ```
  int a = 0, b = 10;
  a++;  // Same as a = a + 1
  b–;  // Same as b = b - 1
  ```

- Shortcut for updating a variable

  ```
  int a = 0, b = 10;
  a += 7;  // Same as a = a + 7
  b *= 12;  // Same as b = b * 12
  ```

Strings

- `String` is a built in class

  ```
  String s, t;
  ```

- String constants enclosed in double quotes

  ```
  String s = "Hello", t = "world";
  ```

- `+` is overloaded for string concatenation

  ```
  String s = "Hello";
  String t = "world";
  String u = s + " " + t;  // "Hello world"
  ```

- String are not arrays of characters
  - We cannot write the following

    ```
    s[3] = 'p';
    s[4] = '!';
    ```

- Instead, invoke method substring in class `String`

  ```
  s = s.substring(0, 3) + "p!";
  ```

- If we change a `String`, we get a new object
  - Strings are immutable
  - After the update, `s` points to a new `String`
  - Java does automatic garbage collection

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  or

  ```
  int[] a = new int[100];
  ```

- `a.length` gives the size of `a`
  - Note, for `String`, it is a method `s.length()`
- Array indices run from `0` to `a.length - 1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`
- For example

  ```
  int[] a;
  int n;
  n = 10;
  ```

`a = new int[n];`

`n = 20;`

`a = new int[n];`

`a = {2, 3, 5, 7, 11};`

Summary

- Java allows scalar types, which are not objects
  - `int, long, short, byte, float, double, char, boolean`
- Declarations can include initializations
- Strings and arrays are objects

Control Flow in Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	3
Lecture URL	https://youtu.be/0RHDEourhdo
Notion URL	https://21f1003586.notion.site/Control-Flow-in-Java-3a68f0e8e0fb49a9babfea919410dc68
Week #	2

Control flow

- Program layout
  - Statements end with semi-colon `;`
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional Loop
  - `while (condition) { ... }`
  - `do { ... } while (condition);`
- Iteration
  - Two kinds of `for`
- Multiway branching — `switch`

Conditional Execution

- `if (c) { ... } else { ... }`
  - `else` is optional
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- No `elif`, unlike python
  - Indentation is not forced

    - Just align `else if`
    - Nested `if` is a single statement, no separate braces required
- No surprises
- Aside: No `def` for function definition

```
public class MyClass {
  ...
  public static int sign(int v) {
    if (v < 0)
      return(-1);
    else if (v > 0)
      return(1);
    else
      return(0);
  }
}
```

Conditional Loops

- `while (c) { ... }`
    - Condition must be in parentheses
    - If body is a single statement, braces are not needed

```
public class MyClass {
  ...
  public static int sumupto(int n) {
    int sum = 0;
    while (n > 0) {
      sum += n;
      n--;
    }
    return(sum);
  }
}
```

- `do { ... } while (c);`
    - Condition is checked at the end of the loop
    - At one iteration, even if the condition is false
    - Useful for interactive user-input

```
do {
  read input;
} while (input-condition);
```

```
public class MyClass {
  ...
  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;
    do {
      sum += i;
      i++;
    } while (i <= n);
    return(sum);
  }
}
```

Iteration

- `for` loop is inherited from C language
- `for (init; cond; upd) { ... }`
    - `init` is initialization
    - `cond` is terminating condition
    - `upd` is loop update

- Intended use is

  `for (i = 0; i < n; ++i) { ... }`

- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

- However, not a good style to write `for` instead of `while`
- Can define loop variable within loop
  - The scope of `i` is local to the loop
  - An instance of more general local scoping allowed in Java

Sample code

```
public class MyClass {
  ...
  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for(i = 0; i < n; ++i) {
      sum += a[i];
    }

    return(sum);
  }
}
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

- Again `for`, different syntax

```
for (type x : a) {
  do something with x;
}
```

- It appears that loop variable must be declared in local scope for this version of `for`

Sample code

```
public class MyClass {
  ...
  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for(int v : a) {
      sum += v;
    }

    return(sum);
  }
}
```

Multiway branching

- `switch` selects between different options
- The default behaviour of `switch` is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch

    - `break` available for loops as well
- Options have to be constants
    - Cannot use conditional expressions
- Aside: here return type is `void`
    - Non-void return type requires an appropriate return value

Sample code

```
public static void printsign(int v) {
  switch(v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Summary

- Program layout: semi-colons, braces
- Conditional execution: `if, else`
- Conditional loops: `while, do-while`
- Iteration: two kinds of `for`
    - Local declaration of loop variable
- Multiway branching: `switch`
    - `break` to avoid falling through

Defining Classes and Objects in Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	4
Lecture URL	https://youtu.be/XB-PcJJpKXg
Notion URL	https://21f1003586.notion.site/Defining-Classes-and-Objects-in-Java-5064c2d02789467897f97abe6e60c846
Week #	2

Defining a class

- Definition block using `class`, with class name
  - Modifier `public` to indicate visibility
  - Java allows `public` to be omitted
  - Default visibility is public to `package`
    - Packages are administrative units of code
    - All classes defined in the same directory form part of the same package
- Instance variable
  - Each concrete object of type `Date` will have local copies of `date, month, year`
  - These are marked `private`
  - Can also have `public` instance variable, but breaks encapsulation

```
public class Date {
  private int day, month, year;
  ...
}
```

Creating Object

- Declare type using class name
- `new` creates a new object

    - How do we set the instance variables?
- We can add methods to update values
    - `this` is a a reference to current object

```
public class Date {
  private int day, month, year;

  public void setDate(int d, int m, int y) {
    this.day = d;
    this.month = m;
    this.year = y;
  }
}

public void useDate() {
  Date d;
  d = new Date();
  ...
}
```

    - We can omit `this` if reference is unambiguous
- What if we want to check the values?
    - Methods to read and report values
- Accessor and Mutator methods

```
public class Date {
  ...
  public int getDay() {
    return(day);
  }

  public int getMonth() {
    return(month);
  }

  public int getYear() {
    return(year);
  }
}
```

Initializing objects

- Would be good to set up an object when we create it
    - Combine `new Date()` and `setDate()`
- Constructors — special functions called when an object is created
    - Function with the same name as the class
    - `d = new Date(13, 8, 2015);`
- Constructors with different signatures
    - `d = new Date(13, 8);` sets `year` to `2021`
    - Java allows function overloading — same name, different signatures
        - Python: default (optional) arguments, no overloading

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y) {
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m) {
    day = d;
    month = m;
    year = 2021;
  }
}
```

Constructors ...

- A later constructor can call an earlier one using `this` keyword
- If no constructor is defined, Java provides a default constructor with empty arguments
  - `new Date()` would implicitly invoke this
  - Sets instance variables to sensible defaults
  - For instance, `int` variables set to `0`
  - Only valid if no constructor is defined
  - Otherwise, we need an explicit constructor without arguments

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y) {
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m) {
    this(d, m, 2021);
  }
}
```

Copy constructors

- Create a new object from an existing one
- Copy constructor takes an object of the same type as an argument
  - Copies the instance variables
  - Use object name to disambiguate which instance variables we are talking about
  - Note that private instance variables of argument are visible
- Shallow copy vs Deep copy
  - Want new object to be disjoint from the old one
  - If instance variable are objects, we may end up aliasing rather than copying

```
public class Date {
  private int day, month, year;

  public Date(Date d) {
    this.day = d.day;
    this.month = d.month;
    this.year = d.year;
  }

  public void useDate() {
    Date d1, d2;
    d1 = new Date(12, 4, 1954);
    d2 = new Date(d1);
  }
}
```

Summary

- A class defines a type
- Typically, instance variables are private, available through accessor and mutator methods
- We declare variables using the class name as type
- Use `new` to create an object
- Constructor is called implicitly to set up an object
  - Multiple constructors — overloading
  - Re-use — one constructor can call another

- Default constructor — if none is defined
- Copy constructor — make a copy of an existing object

Basic Input and Output in Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	5
Lecture URL	https://youtu.be/zvtvQcUhFxo
Notion URL	https://21f1003586.notion.site/Basic-Input-and-Output-in-Java-9629c4bcf2de4676bf073f0997d4f87d
Week #	2

Interacting with a Java program

- We have seen how to print data
  - `System.out.println(”Hello, world”);`
- But how do we read data?

Reading input

- Simplest to use is the `Console` class
  - Functionality quite similar to Python `input()`
- Defined within `System`
  - Two methods, `readLine` and `readPassword`
  - `readPassword` does not echo characters on the screen
  - `readLine` returns a string (like Python `input()`)
  - `readPassword` returns an array of `char`

```
Console cons = System.console();
String username = cons.readLine("Username: ");
char[] password = cons.readPassword("Password: ");
```

- A more general `Scanner` class
  - Allows more granular reading of the input
  - Read a full line, or read an integer ...

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
int age = in.nextInt();
```

Generating Output

- `System.out.println(arg)` prints `arg` and goes to a new line
  - Implicitly converts argument to a string
- `System.out.print(arg)` is similar, but does not advance to a new line
- `System.out.printf(arg)` generates formatted output
  - Same conventions as `printf` in C language