Intro to Java

Table of Contents

- Overview
- Java Features
  - Compilers, Interpreters, etc.
  - Java Virtual Machine
  - Just-in-time compilation
- Hello world!
  - Compiling and running
  - Comments
  - Command Line args
  - Java vs C
  - identifiers
- Data types
- Variables
- Variable classes
- Constants
- Operators
  - Arithmetic
  - Relational
  - Logical
  - Bitwise
  - Other operators
- Mathematical functions
- Control flow
  - Branching
  - Loops
- Operator Precedence
- Wrapper classes
  - Example: Integer wrapper class
  - String parsing
  - Boxing and Unboxing

    - String Comparison

- IO

    - Input
    - Output
    - String Formatting

Overview

- 1991: James Gosling at Sun Microsystems developed first version of Java
- intended for embedded systems (home appliances e.g. washing machines, TVs).

    - complex: various processors make it difficult to make portable, and manufacturers wouldn't want to develop expensive compilers
    - used two-step translation:
        * translate to an intermediate language, *Java byte-code* which is the same for all appliances
        * small, easy-to-write interpreter converts to machine language
        * *Write Once, Run Anywhere*
        * Less low-level facilities

- Oracle now owns Java
- *byte code*: computer-readable program
- *object-oriented programming*: Java is an OOP language

    - objects
    - methods: actions an object can take
    - class: collects objects of the same type

- *Java application program*: class with a `main` method
- *application*: meant to be run by computer, c.f. applet

    - has a `main` method
    - can be invoked from command line using Java interpreter

- *applets*: little Java application;

    - no `main` method
    - program embedded in a web page
    - run by Java-enabled web browser
    - always use a window interface

- Java 13: latest stable verion (3/2020)
  - Java 11: long-term support

Java Features

1. Compiled and interpreted
   - Compiled language (e.g. C)

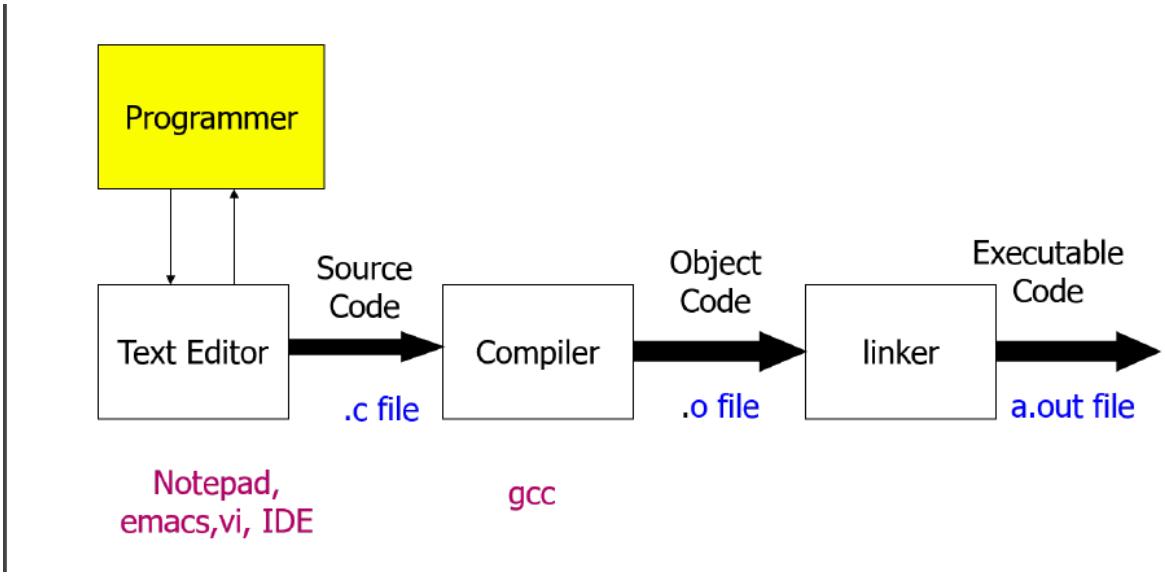

Figure 1: compile

- Java

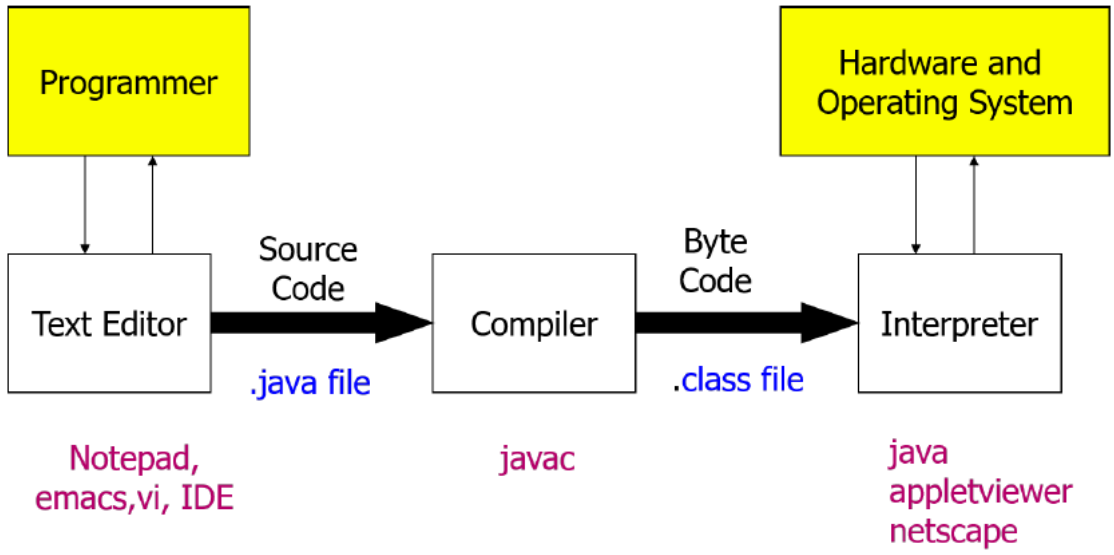

Figure 2: java_compiled_and_interpreted

- Java is compiled to bytecode, then interpreted to machine code
- that bytecode is portable: you can take it to any machine
- porting Java to a new system involves writing a JVM implementation for that system
- most modern implementations of the JVM use **just-in-time** compilation
- older implementations use interpreters that translate and execute bytecode instruction by instruction

2. Platform independent

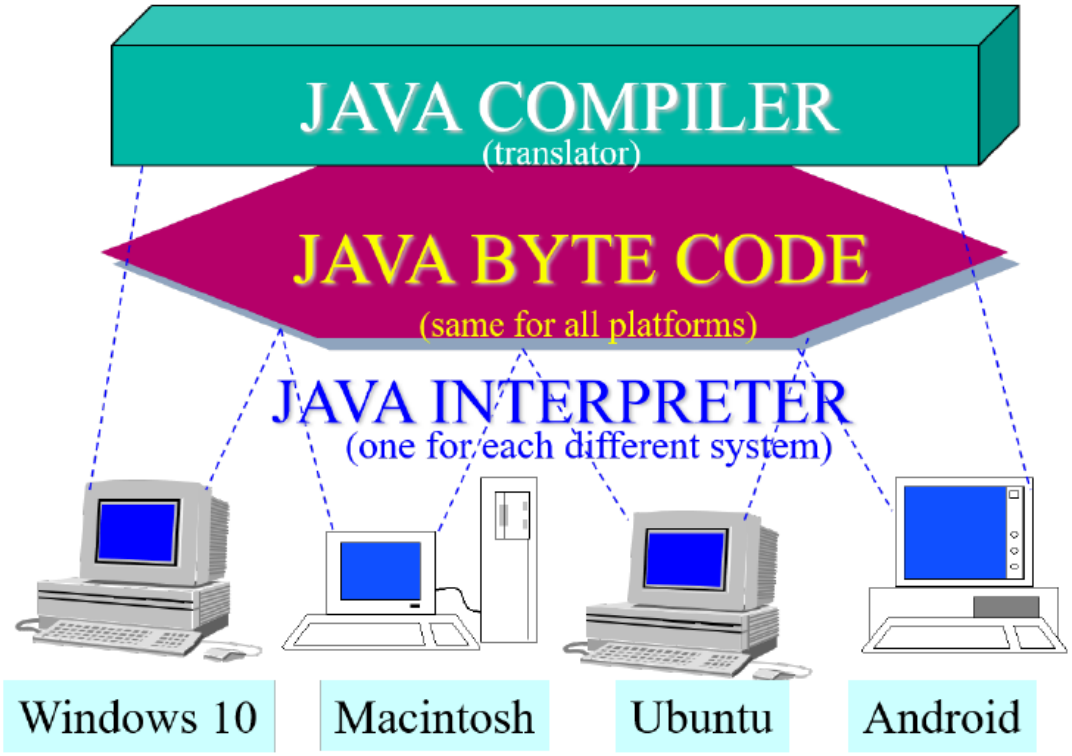

Figure 3: platform_independence

3. Object oriented

Compilers, Interpreters, etc.

Drawn from python interpreter and Absolute Java Ch 1. - **compiler** converts between one language and another - **parser** constructs abstract syntax tree, a tree whose nodes are a syntax element - **semantic analysis**: checks for illegal operations (e.g. 3 args given to a 1 arg function) - analyse AST and modify to syntax for machine code, and produce code in output language - **generator**: walks AST and produces code in output language - disadvantage: compiler translates high level program directly into machine language for particular computer: as different computers have different machine languages, you need a different compiler for each computer type. - **interpreter** performs same operations, except instead of code generation, it loads output in-memory and executes directly on the system

Java Virtual Machine

JVM - JVM lives in RAM - **class loader** loads byte-code from distinct class files together in RAM - byte-code is verified for security breaches - **execution engine** converts bytecode to native machine code via **just-in-time compilation** - running code on JVM makes Java more secure: if a program behaves badly, it does so in context of JVM, instead of directly on your native machine

Just-in-time compilation

JIT compilation - combo of compilation and interpretation - reads chunks of byte-code and compiles to native machine code on the fly - caches compiled chunks, meaning that chunk doesn't need to be recompiled - if the chunk is used very frequently, it can be recompiled with more optimisation to improve performance - has access to dynamic runtime info, which is not available to standard compiler, allowing for some optimisations - performance improvements over pure interpretation Crash course in JIT Compilershttps://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/)

Hello world!

```
1  // HelloWorld.java: Display "Hello World!" on the screen
2  import java.lang.*;             // imports java.lang.* package;
       optional
3  public class HelloWorld{        // name of class must be same as
       filename
4      public static void main(String args[]) {    // standalone program
           must have main defined
5          // args[] contain command-line arguments
6
7          System.out.println("Hello World!");     // out is an object
8          return;                                 // optional; usually
               excluded
9      }
10 }
```

Compiling and running

```
1  # compile
2  $ javac HelloWorld.java
3  # compiler outputs
4  $ ls HelloWorld*
5
6  # run: note absence of extension
7  $ java HelloWorld
```

Comments

- `/**/`: multi-line comments
- `//`: single line comments
- `/** */`: documentation comments

Command Line args

- accessed by `args[]`

Java vs C

- Java: oop language; C: procedural language
- Java:
  - no **`goto`**, `sizeof`, `typedef`
  - no structures, unions
  - no explicit pointer type
  - no preprocessor: (`#define`, `#include`, `#indef`)
  - safe, well-define: memory is managed by VM not programmer

identifiers

- *rules*:
  - must not start with a digit
  - all charactes must in {letters, digits, underscore}
  - can theoretically be of any length
  - are case-sensitive
- *conventions*:
  - `camelCase`:
    - variables, methods, objects: start with lower case, word boundaries uppercase, remaining characters are digits and lower case letters
  - classes: start with upper case letter; otherwise camelCase
- *keywords, reserved words*: cannot be used as identifiers
  - e.g. **`public`**, **`class`**, **`void`**, **`static`**

- *pre-defined identifiers*: defined in libraries required by Java standard packages e.g. `System`, `String`, `println`
  - can be redefined but can be confusing/dangerous

Data types

Java Primitives

Type	Size (bytes)	Values
`boolean`	1	**`false`**, **`true`**
`char`	2	All UTF-16 characters (e.g. 'a', 'ρ', '™')
`byte`	1	-27 to $27-1$ (-128 to 127)
`short`	2	-215 to $215-1$ (-32768 to 32767)
`int`	4	-231 to $231-1$ ($\approx \pm 2 \times 109$)
`long`	8	-263 to $263-1$ ($\approx \pm 1019$)
`float`	4	$\approx \pm 3 \times 1038$ (limited precision)
`double`	8	$\approx \pm 10308$ (limited precision)

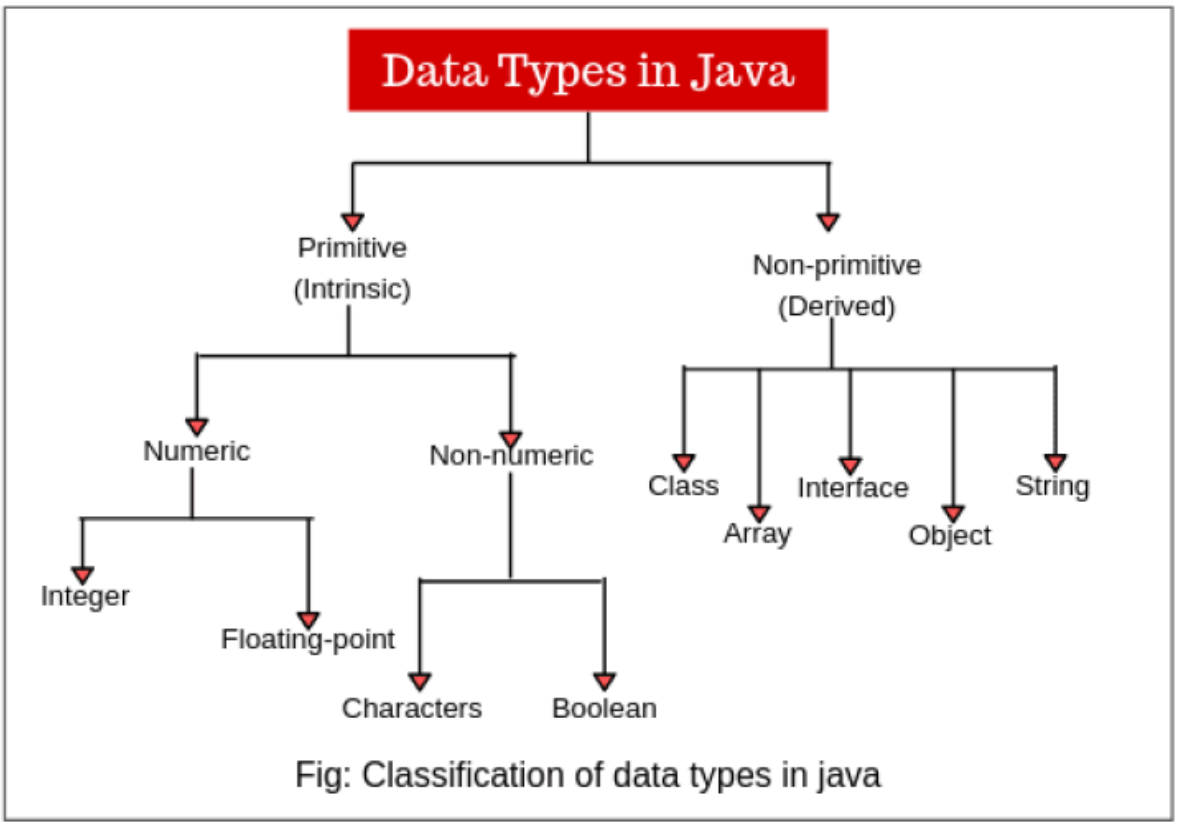

Figure 4: java_data_types

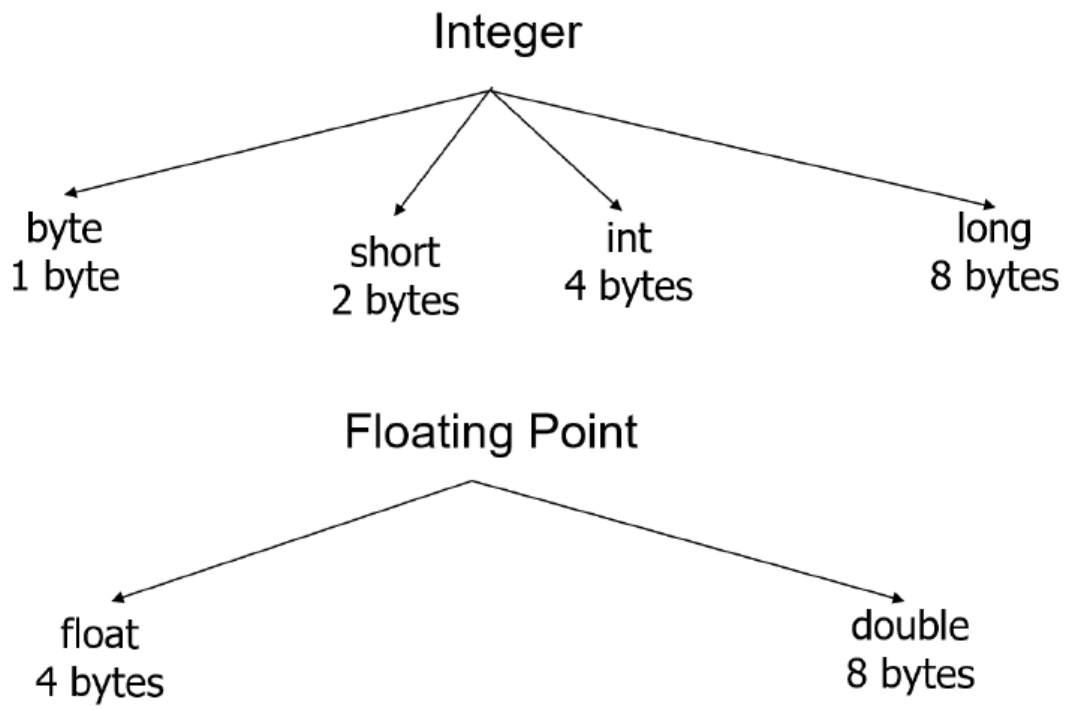

Figure 5: java_numeric_data_types

- characters are enclosed in `' '` not `" "`
- floating point numbers are treated as *double-precision* unless forced by appending `f` or `F` to the number e.g. `float a = 2.3F;`
- `boolean` type: `true`, `false`

Variables

- must be *declared* and *initialised* before use:

```
<type> <variable name> = <initial value>;
```

- **implicit type cast**: a value of any type in the list can be assigned to a variable to its right:

```
byte -> short -> int -> long -> float -> double
char -> int
```

- explicit *type cast* required to assign a value of one type to variable whose type appears to left on above list (e.g. `double` to `int`)

```
int x = 2.99; // invalid assignment
int y = (int)2.99; // valid assignment; x will be 2
```

- `int` variable cannot be assigned to `boolean` variable or vice-versa

Variable classes

1. *instance*
2. *static* (or *class*)
3. *local*: define in a Java method

Constants

- read only values; do not change during execution
- declared with `final` keyword
- `final` variables can be assigned exactly once: this need not be at declaration e.g.

```
final double PI;
...
PI = 3.14159265;
```

- convention: upper case letters with words separated by `_`
- data type need to be explicitly specified

```
final int MAX_LENGTH = 420;
```

Operators

- Java doesn't support operator overloading, except for + in concatenation of `String`s

Arithmetic

Operator	Meaning
+	addition, unary plus
-	subtraction, unary minus
*	multiplication

Operator	Meaning
/	division
%	modulo division

- *mixed-mode arithmetic expression*: if one operand is real and other is integer
  - integer operand converted to real, real arithmetic performed

Relational

Operator	Meaning
<	Is less than
<=	Is less than or equal to
>	
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

- result of relational operator is **`boolean`**

Floating point comparisons

- care needed when checking for equality of two **`float`**s
- use a small `\(\varepsilon\)` value i.e. `Math.abs(float1 - float2)< eps` ### Logical

Operator	Meaning
&&	AND
||	OR
!	NOT

Bitwise

operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
~	one's compliment
<<	shift Left
>>	shift Right
>>>	shift Right with zero fill

Other operators

- (pre/post)-increment: `++`
  - pre-increment: performs addition, returns incremented value
  - post-increment: returns original value, performs addition
- (pre/post)-decrement: `--`
- conditional: `exp1 ? exp2: exp3`
- `a += b` ⟺ `a = a + b`
- `a *= b` ⟺ `a = a * b`

Mathematical functions

- `Math` class in `java.lang` package defines mathematical functions via:

```
1 import java.lang.Math;
2
3 Math.PI;
4 Math.sin(15);
5 Math.toDegrees(Math.PI/2.0);
6 Math.pow(5, 2);
```

Control flow

Branching

- **`if-else`**:

```
if (boolean_expression) {
  // statements
} else if (boolean_expression_2) {
  // statements
} else {
  // otherwise statements
}
```

- **switch**

```
switch (control expression)
{
    case Case_Label_1:
        Statement_Sequence_1
        break; // necessary in most cases, otherwise execution
           falls through to next case
    case Case_Label_2:
        Statement_Sequence_2
        break;
    case Case_Label_<n-1>: // cascading cases: you can join cases
       together like this to produce the same output
    case Case_Label_n:
        Statement_sequence_n
        break;
    default:
        Default_Statement_Sequence
        break;
}
```

- two way decision expression: `expression ? value_true : value_false`

Loops

- **while**

```
while (condition) {
    // statements to execute
}
```

- **do-while**
  - use sparingly. **while** is usually a better approach

```
do {
    // statements to execute
} while (expression)
```

- **for**

```
for (initialise_expr; terminate_expr; update_expr) {
    // statements to execute
}
```

- **break**: exits **while**, **do**, **for** loop
  - exits exactly 1 loop
  - if unlabelled, exits innermost loop
  - loops can be labelled, and then you can specify which loop to break from

```
// break with a label used inside the inner loop to break from the
    outer loop
public class BreakExample {
  public static void main(String args[]) {
      aa: for (int i=1; i <= 3; i++) {
          bb: for (int j=i; j <= 3; j++) {
              if (i ==2 && j ==2) {
                  break aa;
              }
              System.out.println(i + " " + j);
          }
      }
  }
}
```

Output:

```
1 1
1 2
1 3
2 1
```

- **continue**: skips rest of statements in loop
  - kills current iteration of loop

Operator Precedence

Symbol	Definition
. (dot)	Method invocation, member access
++ --	Increment and decrement
- !	Unary negation
(type)	Type casting

Symbol	Definition
* / %	Multiplicative
+ -	Additive
< > <= >=	Relational
== !=	Equality
&&	Boolean and
||	Boolean or
= += *= ...	Assignment

Wrapper classes

- most Java methods expect Objects - so when you need to pass in a primitive e.g. `int`/`double` you need to use a wrapper class to dress up the primitive to behave like an object
- wrapper classes also provide additional functionality to primitives
- use them sparingly - i.e. only when you need to

primitive	wrapper Class
boolean	Boolean
byte	Byte
char	Character
int	Integer
float	Float
double	Double
long	Long
short	Short

Example: Integer wrapper class

```
1 Integer.reverse(10);    // reverses bit sequence of a number
2 // -> 1342177280
3 Integer.rotateLeft(10, 2); // shifts bit sequence
```

```
// -> 40
Integer.signum(-10);    // indicates sign of number
// -> -1
Integer.parseInt("10"); // parses string as integer
// -> 10
```

String parsing

Every wrapper class has a parseXXX method that converts a string into that type

```
int i = Integer.parseInt("10"); // -> 10
double d = Double.parseDouble("10"); // -> 10.0
boolean b = Boolean.parseBoolean("TrUe"); // -> true
```

Boxing and Unboxing

- typically when a primitive is expected, wrapper is automatically **unboxed** to behave like a primitive, and when a class is expected, the primitive is automatically **boxed**

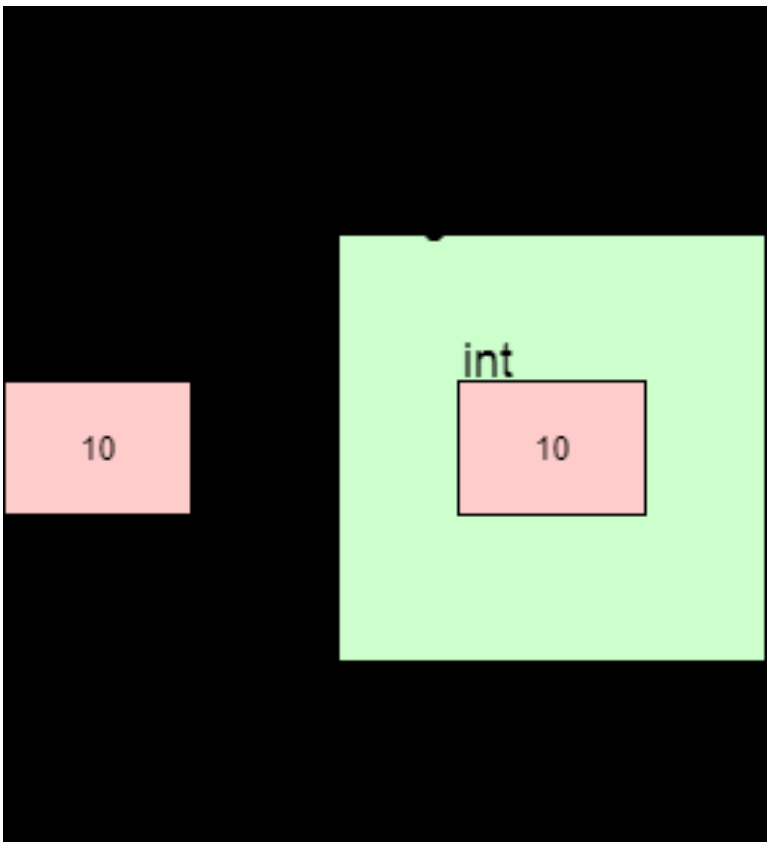

Figure 6: boxing_unboxing

String Comparison

Use `string1.equals(string2)`

IO

Input

```
1  import java.util.Scanner;
2
3  public class Program {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          // read next line of input: NB this is the only one to eat
               newline characters
7          String inputLine = scanner.nextLine();
8          // read next word of input
```

```
9           String input = scanner.next();
10          // read next int
11          int i = scanner.nextInt();
12          // read next double
13          double d = scanner.nextDouble()
14          // read next bool
15          boolean b = scanner.nextBoolean()
16
17      }
```

Use `scanner.hasNextXXX()` to determine if there is input of type `XXX` ready to be read

Output

```
1 System.out.print(...) // outputs without newline character
2 System.out.format(...) // format and print to terminal
3 String.format(...) // returns formatted string
```

String Formatting

- string formatting documentation
- `%`: indicates start of format specifier
- argument index: indexes arguments provided after string to be formatted
  - `<` references previous value
- flags: special characters that can be applied to all formatting
  - `0` pads with zeroes
  - `-` left justify
- width: minimum number of characters a formatted value should occupy
  - by default it is padded with spaces

- precision: number of decimals for a float
- conversion: type of value
  - `d`: integer/decimal
  - `f`: floating point
  - `s`: String

```java
String.format("%3.2f", 4.56789); // min width 3, 2 decimal points, float
// output: 4.57

String.format("%+05d", 10);  // always include a sign, pad with zeroes, min width 5, integer
// output: +0010

String.format("%2$d %<05d %1$d %3$10s", 10, 22, "Hello");
// %2$d: 2nd arg, integer
// %<05d: use previous arg (2nd arg), pad with zeroes, min width 5, integer
// %1$d: use 1st arg, integer
// %3$10s: use 3rd arg, min width 10, string
// output: 22 00022 10 Hello
```