CS 2103: Class 2

Jacob Whitehill

Project 1: walkthrough

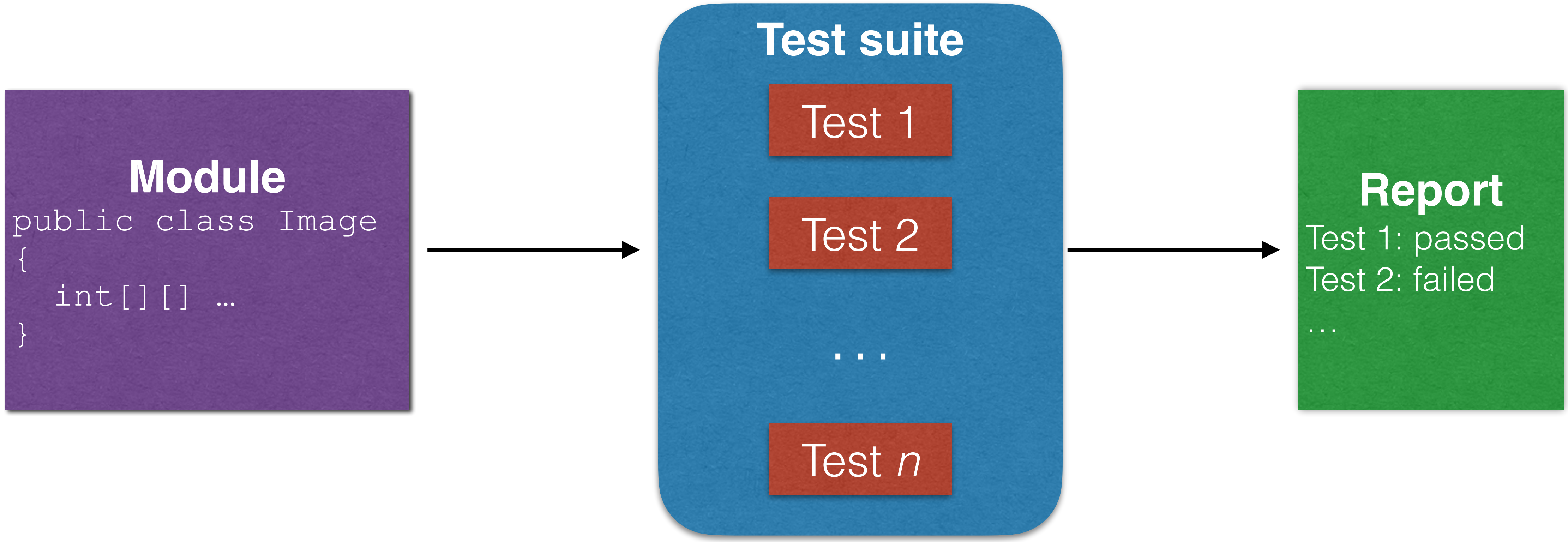
Software testing

Software testing

- When developing new software, it is very often useful *first* to write tests of how the code *should* perform once it has been written.
    - Defining the test cases up-front can help clarify how the method should operate.

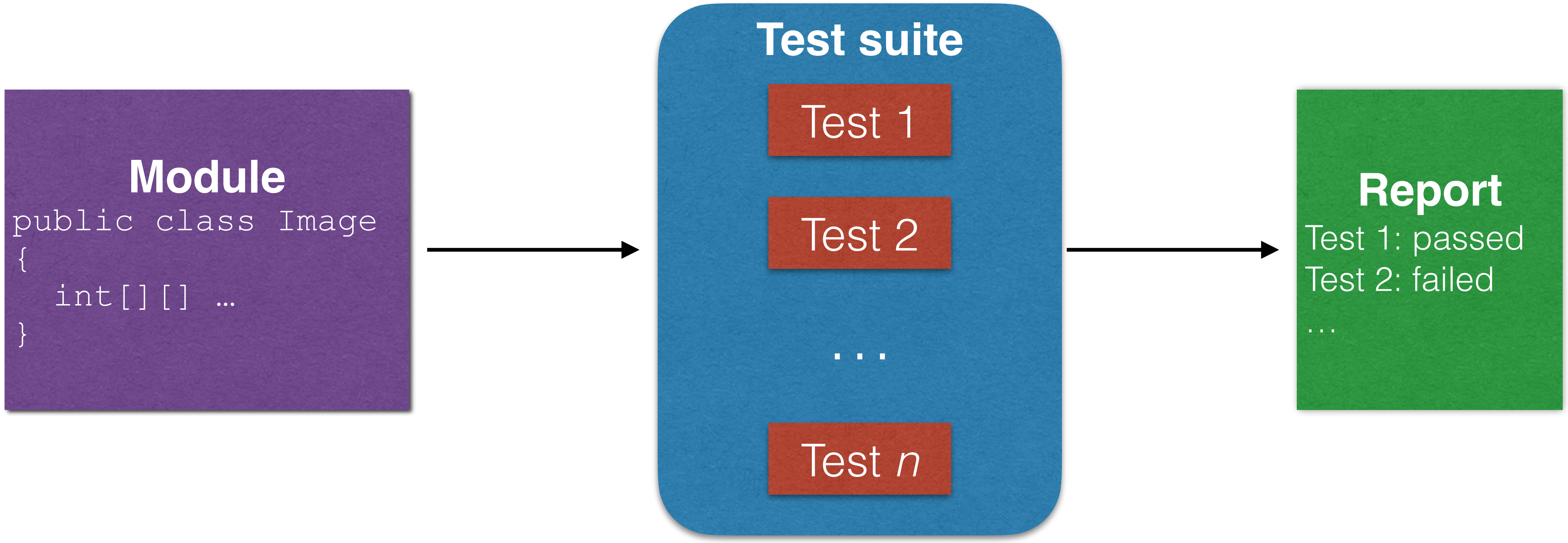
Software testing

- A group of tests designed to test a particular module (e.g., a class) is sometimes called a **test suite**.



Software testing

- If **all** tests in the suite pass, then the program **passes** the test suite.
- If **any** of the tests in the suite fail, then the program **fails** the test suite.



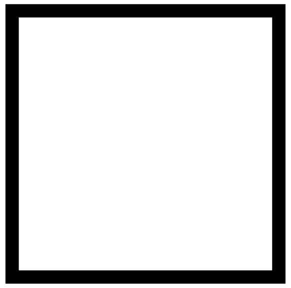
Black-box and white-box testing

- **Black-box testing**: the author of the test does not assume anything about the particular *implementation* of the module. The test can verify correctness only by examining the module's input/output behavior.

Black-box and white-box testing

- **Black-box testing**: the author of the test does not assume anything about the particular *implementation* of the module. The test can verify correctness only by examining the module's input/output behavior.

- **White-box testing**: the test has access to the internal implementation of the program (e.g., private state variables) and possibly the code itself (for static analysis and theorem proving).



Key properties of good test suites

- **False alarm (FA)**: a *fully correct* program fails at least one test in the suite.

- *Zero* false alarm rate.

  - None of the tests in the suite should flag an error if the code being tested is actually correct.

Key properties of good test suites

- **Miss/False Negative (FN)**: a *buggy* program passes all tests in the suite.
- *Low* (close to 0%) false negative rate.
    - We want to minimize the probability that a piece of buggy code “slips through the cracks”.

Key properties of good test suites

- It's usually **impossible** to reach a 0% FN rate for black-box testing. Consider:

```
/**
 * Returns the sum of two numbers.
 */
int sum (int a, int b) {  // correct implementation
  return a+b;
}
```

Key properties of good test suites

- It's usually **impossible** to achieve for black-box testing. Consider:

```
/**
 * Returns the sum of two numbers.
 */
int sum (int a, int b) {  // buggy implementation
  if (a == 123512325) {
    return a + b - 1;
  } else {
    return a + b;
  }
}
```

Key properties of good test suites

- This bug would be virtually impossible to catch with black-box testing:

```
/**
 * Returns the sum of two numbers.
 */
int sum (int a, int b) {  // buggy implementation
  if (a == 123512325) {
    return a + b - 1;
  } else {
    return a + b;
  }
}
```

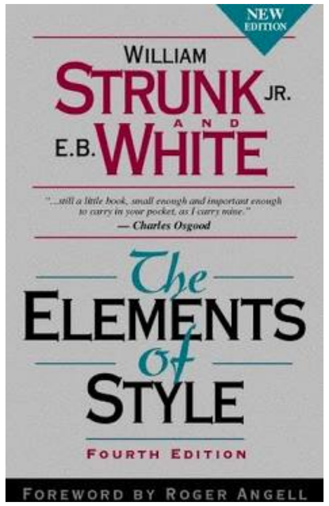
Other important key properties

- Fast (within the context of the application):
  - Otherwise, it may be too cumbersome to execute often.

junit

- In CS 2103, we will use junit-5.8.1 to facilitate **unit testing**.
- **Unit tests** verify the correctness of specific (typically small) units of code (e.g., a method).
- With junit, unit tests are identified using **annotations** (meta-information in your .java files).
    - See `PianoTester.java` in `Project1.zip`.

Elements of Good Programming Style



Elements of good programming style

1. Be consistent — using the same syntax and structure makes it easier to read your code.

2. Don't do too much at once — decompose large methods or classes into smaller ones (<= 50 lines).

3. Avoid redundancy — factor out common code.

4. Keep it simple & intuitive — some program designs are easier to understand than others.

Be consistent

Be consistent

```
public class Paint {
        private Color[][] _pixels;
        private boolean IsDirty;
        private void setColor(int x,int y, Color TheColor)
        {
                _pixels[y][x] = TheColor;
        }
  private Color getColor (int x, int y) {
    return _pixels[y][x];
  }
        private boolean HasBeenModified () {
                return IsDirty;
        }
        private void setAllPixels (Color color) {
                for (int y = 0; y < _pixels.length; y++) {
                for (int x = 0; x < _pixels[y].length; x++)
                {
                        _pixels[y][x] = color;
                }
                }
        }
}
```

Be consistent

- Inconsistent code is harder to read and maintain.
- Code that looks sloppy will be treated with skepticism.
- Other programmers will scrutinize it and expect bugs, even if there are none.
- To avoid these problems, companies often set coding style guides and standards that must be followed.

Avoid redundancy: Program decomposition & Code refactoring

Program decomposition and code refactoring

- A key goal in software design is to eliminate **redundancy** in code.

Redundancy: example

```
void resetAccount (State state) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        if (account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }
        if (balance < 0 || state.mustPayAll()) {
                payBalance(account, balance);
                state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Confirmation");
        }
}

void redeemGiftCard (State state, float giftCardAmount) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        account.setBalance(balance + giftCardAmount);

        if (balance < 0 || state.mustPayAll()) {
                if (askUser(name, "Pay balance?")) {
                        payBalance(account, balance);
                        state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Thanks");
                }
        }
}
```

Program decomposition and code factoring

- Why redundancy is bad:

Program decomposition and code factoring

- Why redundancy is bad:
  - Hard to understand (more code to read).
  - More effort to maintain (since more code has to be updated).
  - Higher chance of bugs (when some code is updated but not the other “copies”).

Reducing redundancy redundancy

- Different programming paradigms (imperative, functional, object-oriented) offer different ways of reducing code redundancy.

- Common to all three paradigms is decomposing a long program into **methods/functions** that can be called from various parts of the program.

- A method is a block of code with a defined purpose and input/output relationship.

Refactoring

- During the evolution of a program, it is common to reorganize code to reduce program redundancy by "factoring out” common code.
- This process is known as **refactoring**.

Refactoring: example 1

- ```
  void method1 () {
    a();
    b();
    c();
  }
  ```

- ```
  void method2() {
    b();
    c();
    d();
  }
  ```

Refactoring: example 1

- ```
  void method1 () {
      a();
      b();
      c();
  }
  ```

- ```
  void method2() {
      b();
      c();
      d();
  }
  ```

Refactoring: example 1

- ```
  void method1 () {
     a();
     e();
  }
  ```

- ```
  void method2() {
     e();
     d();
  }
  ```

- ```
  void e () {
     b();
     c();
  }
  ```
  ```

What are different ways of creating helper methods to “factor out” the common code below?

```
void resetAccount (State state) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        if (account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }
        if (balance < 0 || state.mustPayAll()) {
                payBalance(account, balance);
                state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Confirmation");
        }
}

void redeemGiftCard (State state, float giftCardAmount) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        account.setBalance(balance + giftCardAmount);

        if (balance < 0 || state.mustPayAll()) {
                if (askUser(name, "Pay balance?")) {
                        payBalance(account, balance);
                        state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Thanks");
                }
        }
}
```

Refactoring with helper methods: considerations

- Does each helper method have a cohesive definition, or does it “glue” together random parts?
- Is the refactored code easier or harder to read than before?
- Is the amount of code reduced?

Why Java?

Java

- Java is a compiled, “mid-level” language that runs on a virtual machine.
- “High”, “low” and “mid”-level languages refer to the level of abstraction.
- More abstract than C:
  - E.g., can’t manipulate memory directly using pointers.
- Less abstract than Python:
  - E.g., can’t just call “`range`” to create a list of numbers; need to manually construct an array.

Compilation/Translation

- Before a programming language such as C/C++ can be executed by the physical CPU, it must be compiled into something the CPU can understand.

- The native language of a CPU is its **assembly language**.

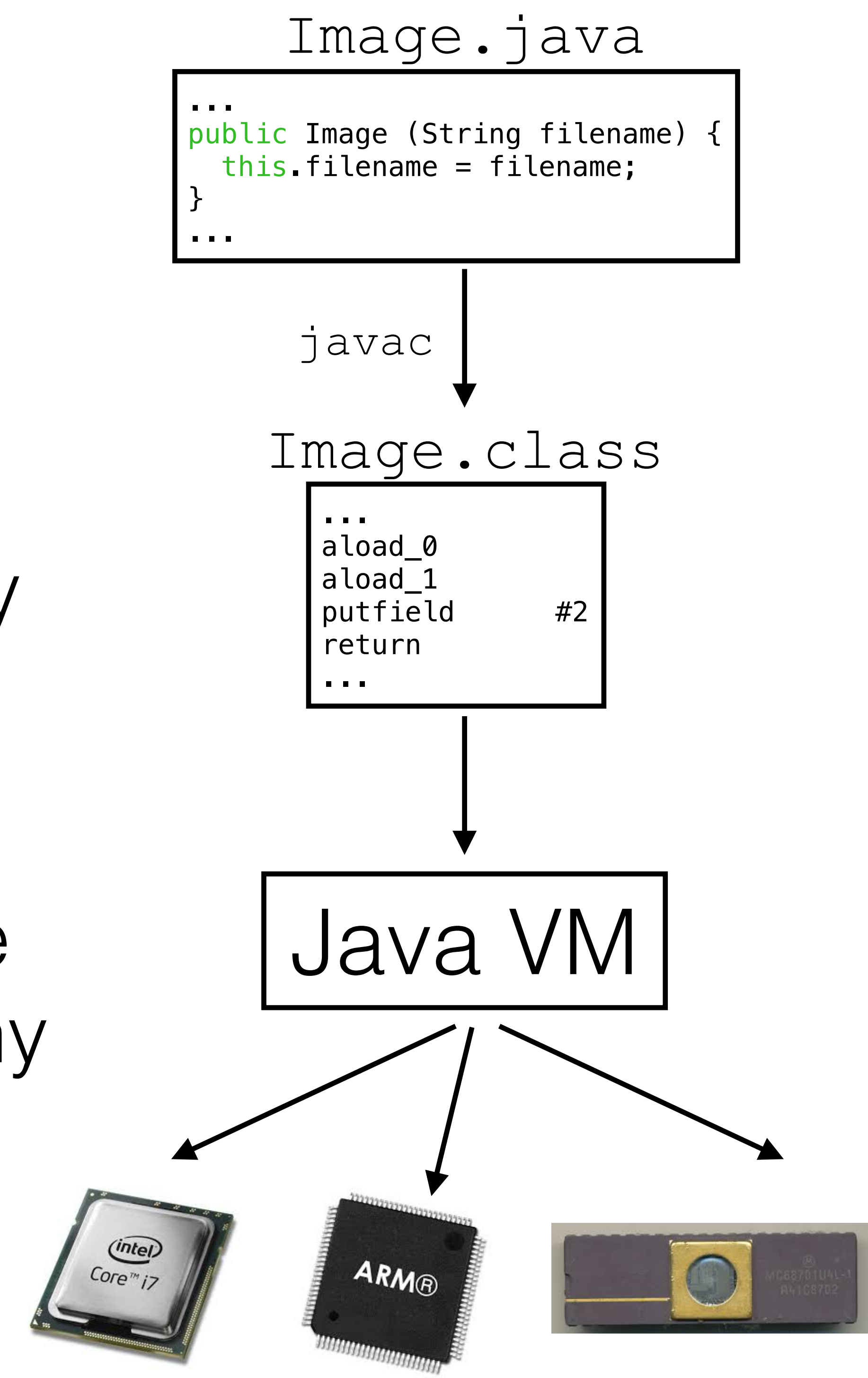
Java VM

- Java code is **not** compiled into assembly language instructions that can be directly executed on the host CPU (e.g., Intel i7, ARM).

- Instead Java runs on a **virtual machine** (VM).

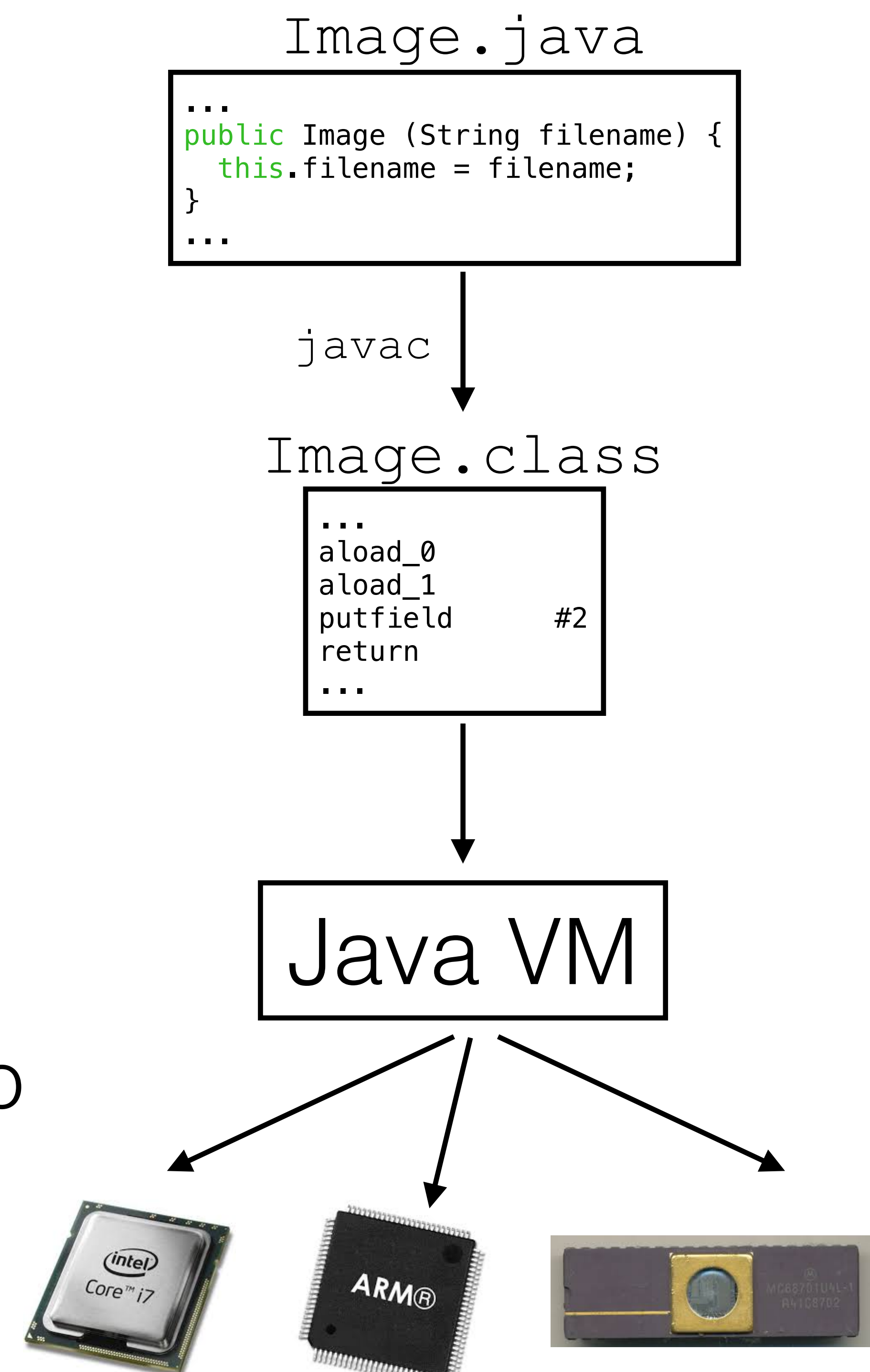
Java VM

- The `javac` compiler compiles Java source (.java) into bytecode (.class).
- These bytecode files are the "native language" of the Java VM.
- Java VM implementations exist for many operating systems and hardware platforms.
- This makes Java very portable because the same .class files can be run on many devices without being recompiled.



Java VM

- The VM has implications for how to manage memory in Java:
  - Once an object is no longer needed, it is automatically deallocated.
  - It is impossible to make certain kinds of mistakes that ubiquitous in C/C++.
  - The programmer does not have to keep track of which memory blocks to “free”.



Why Java?

- Java is arguably **more secure** than some languages (e.g., C) because of features such as:
  - Type checking
  - Array-bounds checking

Why Java?

- Java is usually **slower** than C:
  - Java runs on a VM;
    C runs directly on underlying CPU.
  - Java implements run-time security features;
    C just assumes everything is fine.

Why Java?

- Java is particularly well-suited for:
  - Enterprise computing.
  - Mobile app development (specifically Android).
  - (Some) scientific simulations.

Enterprise computing

- Enterprise computing applications typically involve **complex business logic**; they include:
  - Large-scale billing systems for healthcare, insurance, etc.
  - Online banking platforms
  - Stocks & options trading systems

Mobile app development

- Java offers (fairly) high performance, security, and **portability**:

    - Compiled Java apps can run on many different hardware platforms.

Java in 2020

- The software landscape is changing:
  - **Server-side**: Node.js is being used for more and more large-scale web applications.
  - **Client-side**: Javascript+HTML5 is increasingly powerful, and highly portable. Google now promotes a new language, Kotlin.
  - **Scientific computing**: Python is very popular.