latex input: mmd-article-header Title: Java Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Object Memory Structure

7/16/15

1. C/C++ has a special `sizeof` *operator* that lets you query the size of primitives and even classes
2. Java doesn't have any analogous operator, and that may be because technically, Java doesn't need one because sizes are *well-defined* in the language spec, and you aren't allowed to do pointer arithmetic anyways.
3. But for eg. big data processing with Spark, you start to get curious how much space objects really take and how they're laid out
4. Let's define
    - **Shallow size** -- space occupied by the object alone, not counting objects it references
    - **Deep size** -- space occupied b entire object graph rooted at current object
5. Runtime memory structure of Java objects is not enforced by the VM spec, so memory usage of individual instances of the same class may vary between VMs, so we'll talk specifically about Sun's *HotSpot* JVM

Objects

This is from a 2008 blog post referring to the 32-bit hotspot. I haven't looked into what has changed since then.

1. 2 word header
    - Recall each **word** is 4-bytes = 32-bits = the "architecture size" or whatever that's called
2. 1st has objects identity hash code and flags, eg. lock state and age
3. 2nd has a reference to the object's class
4. Objects are aligned to 8-bytes
5. For an object with no fields, we could stop here
6. Class attributes are aligned in memory to their size: ints to 4-bytes, longs to 8-bytes, etc.
7. Attributes are organized in the following order to cut down on alignment-padding
    1. doubles & longs
    2. ints & floats

    3. shorts & chars
    4. booleans & bytes
    5. references
8. For an object that directly extends Object, we could stop here
    - Now we know why an instance of `java.lang.Boolean` requires *16-bytes*
9. Fields are organized in memory according to the class hierarchy, ie. inherited fields are *not* mixed in with one an other for better alignment
10. There are the rare exceptions to the organization rules to save space in obvious wastage situations

Arrays

1. Extra header field to contain the `length` variable
2. Then come the array elements

Inner classes

1. Non-static inner classes have that "hidden" field holding a reference to the outer class, which also gives it a sneaky 4 byte cost

Further Discussion

1. In the Spark source code, they want to estimate sizes in order to build "memory bounded caches" (source code), which are caches that have a fixed upper-bound on how much heap they are allowed to use, and use LRU to dump keys that can no longer fit.
2. This is because in your Spark program, different machines in your cluster may have different amounts of RAM, and you may cache heterogeneous values and don't want to figure out quota's for each type you're going to store

Refs:

1. Code Instr 2008
2. Madhukar's Blog 2014
3. Github -- `spark.util.SizeEstimator.scala` 2015
4. JavaWorld 2003

Memory Leaks

6/19/15

> A Java heap memory leak occurs when one **maintains object references** that are **no longer needed**.

1. Static analysis and leak detection tools are not so helpful
2. Leaks can lead to `OutOfMemoryErrors` (OOMs)

    1. This is thrown whenever there is *not enough memory* left to allocate the object you called for
3. Often if you get an OOM, you should simply use a JVM flag to increase the size of whatever thing (java heap, perm-gen, etc.) ran OOM
4. Things are even more icky if your using a lot of JNI stuff
5. Given general steps outline for diagnosing leaks
    1. Identify symptoms (e.g. OOM error(s))
    2. Enable verbose garbage collection (pass `-verbosegc` argument)
        1. I think this means whenever a GC occurs, it will dump a bunch of stats to `STDOUT`
    3. Enable profiling of the heap (e.g. using `jmap`) to get detailed info about the type and size of objects
    4. Analyze the trace (e.g. generated by `Java VisualVM`) to find any blocks of objects in the heap that shouldn't be there, and especially look after a point where their utilized space *ought* to have been freed
    5. Look in the *heapdump* to see what classes are using an out-sized proportion of the heap
6. Example: If we continue to add equavalent versions of an object to a `HashMap` but we never *defined* `equals()` and `hashCode()`, a new copy of the object will be added to the `HashMap` every time, i.e. *a leak will occur*

References

- Hunting Memory Leaks in Java

Garbage Collection

6/8/15

Summary

The garbage collector *separates memory into 5 sections of 4 types*. I don't know whether these spaces are "virtual" (i.e. remapped to discontiguous "pages") or not. But I suspect they are not.

1. **Eden** --- where objects go when they're first allocated in the running program
2. **Survivors 1 & 2** a.k.a. "young space" --- objects in Eden are moved here if they survive a minor ("young") GC
3. **Tenured** (a.k.a. **Old**) --- long-lived objects (that have survived [a configurable number of] minor GCs) are moved and then live in here
    - We can tell the JVM to allocate all objects larger than `n` bytes directly into the *old* space.
4. **Permanent** --- this is where the JVM's own objects live (e.g. classes and JITed code). It behaves just like the *tenured* space.

The GC is arranged "generationally" because (according to the "**generational hypothesis**") it is assumed the longer objects live, the longer into the future their life expectancy is. So if we move the older objects into a separate bin, we can do quick, efficient, lucrative "minor GCs" in which we only garbage collect from Eden and the Survivor spaces.

Minor GC is triggered *when Eden becomes full*. It uses the root references to collect the reference set, and moves all live objects from Eden and one survivor space into the other survivor space. I guess this means we're physically moving the object in RAM because we have to update all the objects references to the new location.

Root references for minor GC are from the stack (I think this means entire stacks for all running threads) and old space. HotSpot uses "dirty cards" as an optimization to not have to trace through references from all members of the old space, only the modified ones.

Full GC would *intuitively* be triggered by running out of space in the "tenured" or "permanent" bins, but this is not *necessarily* the case.

References

- Grid Dynamics
- Cubrid Blog
- StOve

Inheritance

Private fields

5/9/14

- `private` fields are not inherited by subclasses.
- Both `protected` *and* `public` fields *are* inherited by subclasses, *no matter* whether they are in the same `package`.
- Instances of subclasses of course *do* ***contain*** the private fields of their superclasses, they just have no access to them.

Useful chart

5/9/14

Modifier	Class	Package	Subclass	World
public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	N

no modifier	**Y**	**Y**	N	N
private	**Y**	N	N	N

Notes on chart

no `modifier` is even stricter than `protected`, and what it *doesn't* allow is `subclass` access, even though it retains `package` access. I find this counterintuitive, personally.

instanceof

1. Requires an object or array on left, and name of a reference type on right
2. Returns `true` iff the left is an *instance of* the right
3. `null instanceof Anything => false`
4. `xx instanceof Anything` implies it is safe to do

```
Anything yy = (Anything) xx;
```

5. That's why `child instanceof Parent` will return **`true`**

Comparable vs. Comparator

Refs: SO, digizol

- Both are **`interfaces`** you can implement
- **Comparable says "I can compare *myself* with another object"**
- **Comparator says "I can compare two *other* objects with each other"**
- **Use `comparable` if it's your class, otw use `comparator`**
- **The associated methods have different names**
  - The `Comparable` method is called `compareTo()`
  - The `Comparator` method is simply called `compare()`

Comparable

- **Says "I can compare *myself* with another object"**
- Allows you to define comparison logic for *your own* types.

Signature

```
java.lang.Comparable: int compareTo(Object o1) {

    case this > o2 => x > 0;  // LaTeX disallowed in code-block
    case this = o2 => x = 0;
    case this < o2 => x < 0;

}
```

How to declare

```
public class MyClass implements Comparable<MyClass> {

    public int compareTo(MyClass o) {
        return this.field - o.field;
    }

}
```

How to use

Simply declare it as in the "How to declare" section above.

Comparator

- **Says "I can compare two *other* objects with each other"**
- Allows *you* to define comparison logic for types you don't control.
- E.g. you could write a new way to compare strings by `extending Comparator`.

Signature

```
java.lang.Comparator: int compare(Object o1, Object o2) {

    case o1 > o2 => x > 0;
    case o1 = o2 => x = 0;
    case o1 < o2 => x < 0;

}
```

How to declare

```
public class MyClassSortByField implements Comparator<MyClass> {

    public int compare(MyClass o1, MyClass o2) {
        o1.getField().compareTo(o2.getField());
    }

}
```

How to use

Make a method like this

```
public static Comparator<Fruit> FruitNameComparator
                    = new Comparator<Fruit>() {

    public int compare(Fruit fruit1, Fruit fruit2) {

      String fruitName1 = fruit1.getFruitName().toUpperCase();
      String fruitName2 = fruit2.getFruitName().toUpperCase();

      //ascending order
      return fruitName1.compareTo(fruitName2);

      //descending order
      //return fruitName2.compareTo(fruitName1);
    }

};
```

And then do this

```
import java.util.Arrays;

Fruit[] fruits = new Fruit[3];

Fruit pineappale = new Fruit("Pineapple", "Pineapple description",70
Fruit apple = new Fruit("Apple", "Apple description",100);
Fruit orange = new Fruit("Orange", "Orange description",80);

fruits[0] = pineappale;
fruits[1] = apple;
fruits[2] = orange;

Arrays.sort(fruits); // ClassCastException

Arrays.sort(fruits, Fruit.FruitNameComparator);  // works
```

Syntax

1. In Java 7+, write numbers in binary with a prefix 0b (e.g. `0b1001` is 9).
2. Also in Java 7+, you can add underscores to number literals (e.g. `1_000_000`).

Asserts

```
assert assertion;
assert assertion : "error message";
```

By default, assertions are not enabled, and don't actually do anything. To enablethem, run `java -ea main.package.MyMainClass`.

Because hitting an assert means the program is *not performing as intended*, there is no plausible way to recover from an `AssertionError`, and you should not attempt to `catch` it.

Generics

1/18/15

Type parameters on generic classes are not known by runtime

Recall from OOP, that unlike in C++, the code

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

prints `true` because **all instances of a generic class have the same run- time class, regardless of their actual type parameters**.

G is *not* a supertype of G

> Source: Oracle Generics Trail

Is the following legal?

```
1||  List<String> ls = new ArrayList<String>();
2||  List<Object> lo = ls;
```

Well consider if next you did the following?

```
3|| lo.add(new Object());
4|| String s = ls.get(0);
```

This would be a terrible mistake, because you can't *assign* an `Object` to a `String`! So that's why `line 2` was actually a *compile time error*.

Introductory motivation for wildcards (<?>)

1. Oracle Generics Trail
2. *Core Java V1 Ed. 9, Section 12.8 "Wildcard Types"*

Given what we learned above---that `G<SuperType>` is *not* a supertype of `G<SubType>`---we know that `G<Object>` is *not* the superclass of all type-parameterized instances of `G`. So if we have the following method

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

we can*not* pass in e.g. a `Collection<String>`. So instead we must use a **wildcard** like so

```
void printCollection(Collection<?> c) {
    ...
}
```

Restrictions on collections of type "unknown" (wildcard)

1. Methods that require *parameters* of type `T` (e.g. `void add(T e)`) can*not* be called because here `T == ?` so the type can't be verified
2. Methods that *return* objects of type `T` (e.g. `T get()`) *can* be called and we can just store them in the global supertype `Object`.

Why we'd write "`Collection<? extends SuperType>`"

If we have an instance `Pair<Superclass> p` then we *cannot* assign `p = new Pair<Subclass>` which is not good. But we can pull this off if we declare `Pair<? extends Superclass> p` instead.

And the same goes for the `printCollection` situation above. If instead of simply calling `toString()` (which *everyone* can do), we call a method `draw()` defined by

objects inheriting from `SuperType`, then we're going to want to allow collections type-parameterized by anything that `extends SuperType`, so instead of writing

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
   }
}
```

we must write

```
public void drawAll(List<? extends Shape> shapes) {
    ...
}
```

Generic Methods

10/11/14

Oracle Docs

1. Introduces its own type parameters
2. The type parameter's scope is limited to the method
3. Method could be static, non-static, *or* class constructor
4. Syntax: appears inside angle brackets in the method declaration before the return type

Why?

1. Say you have objects that have type parameters that must be comparable
2. Now you want to write a static method that operates on them, using those type parameters
3. So you want to note the type-constraint that the parameterized-types of the objects being passed in are Comparable
4. So you'd do

```
public static <K extends Comparable<? super K>, V> boolean
        myMethod(MyType<K, V> a, MyType<K, V> b)
{/*...*/}
```

5. I believe this means that *only now* can you put the line

```
a.key.compareTo(b.key)
```

inside of `myMethod(...)`

6. You invoke this method with

```
StaticClass.<TypeK, TypeV>myMethod(obj1, obj2);
```

An even better example demonstrating how these are used can be found at Oracle Docs2, but basically it confirms the above to be a correct interpretation.

Reflection

5/21/14

> "Reflection" is a language's ability to inspect and dynamically call classes, methods, attributes, etc. at runtime. For example, all objects in Java has the method getClass, which lets you determine its class even if you don't know it at compile time (like if you declared it as Object) - this might seem trivial, but such reflection is not by default possible in less dynamic languages such as C++.

> More advanced uses lets you list and call methods, constructors, etc.

> Reflection is important since it lets you write programs that does not have to "know" everything at compile time, making them more dynamic, since they can be tied together at runtime. The code can be written against known interfaces, but the actual classes to be used can be instantiated using reflection from configuration files. Lots of modern frameworks uses reflection extensively for this very reason.

> Most other modern languages uses reflection as well, and in script languages like Python can be said to be even more tightly integrated, since it matches more naturally with the general programming model for those languages.

> StOve

Notes from a great tutorial

Using Java Reflection you can inspect Java classes at runtime. Inspecting classes is often the first thing you do when using Reflection. From the classes you can obtain information about

- Class Name
- Class Modifies (public, private, synchronized etc.)
- Package Info

- Superclass
- Implemented Interfaces
- Constructors
- Methods
- Fields
- Annotations

You get this from the **`Class` object**, which you get from `Class class = MyObject.class`

- Useful for creating visual IDE features, debugger, test tools
- Drawbacks include slow performance, and allowing you freedom to break incapsulation.

Annotations

5/21/14 & 4/22/16

Oracle Tutorial

- An annotation is metadata about a program that is not part of the program itself
- Info for the compiler -- e.g. detect errors or suppress warnings
- Compile/Deploy-time processing -- generate code for processing XML files
- Runtime processing
- They allow you to add metadata to your classes/methods/variables that can be queried at runtime (some built-in ones are queried at compiletime). This is used by frameworks (e.g. Spring) on initialization to wire things together. It can also be used to provide integrity checks against some semantic meaning of the annotation. Python decorators are said to be inspired by this mechanism.

Basics

- Starts with an `@` character
- Can be `@Title(key = value, key2 = value2)` or `@Title(value)`
- You can stack multiple annotations onto one declaration, but style-wise one should only have one per line

Placement options

Pre Java 8

- Placed before any sort of *declaration*
    - Class, method, field, etc.

Java 8+

- Constructor: `new @Annot MyObject();`
- Type cast: `m = (@Ann A) v;`
- Implements: `class A<T> implements @Annot B<@Annot T> {...}`
- Exception declaration: `void meth() throws @Annot Excep {...}`

Declaring an Annotation

This part of the explanation is not helpful.

```
@interface AnnotationName {
  String annotVar1();
  int annotVar2() default 3;
  String[] vars3();
}
```

Now we can use it like

```
@AnnotationName(
  annotVar1 = "asdf",
  annotVar2 = 44,        // or don't include it bc there's a default
  vars3 = {"a","b","c"} // no semicolon
)
```

Predefined annotation types

@Deprecated

The compiler generates a warning whenever a program uses something with this annotation.

```
/**
 * @deprecated   // put this in the JavaDoc
 * reason for deprecation
 */
@Deprecated
static void meth() { }
```

@Override

Informs the compiler that the following element is meant to override something declared in a superclass. **This annotation is not required, but if you fail to correctly override something when you *said you were trying to*, you'll get a compiler warning.**

The other ones don't look so important

Why would I use annotations?

StOve

Using **reflection** over annotations is how *JUnit* works out your methods are. When you run JUnit, it uses reflection to look through your classes for methods tagged with the `@Test` annotation, and then calls them when running the unit test.

Primitive types

All of the following are big-endian, two's-complement signed, integers

- **byte** --- **1-byte** $[-128, 127]$
  - The addition of two `byte` variables produces an `int` as a result
  - A single `byte` takes up 4-bytes of memory, but a `byte[]` takes up only what it needs
  - When you cast an `int` to a `byte` you're doing a `myInt & 0xFF`
- **short** --- **2-bytes** (rarely used)
- **int** --- **4-byte**
- **long** --- **8-byte**

But this is different

- **char** --- **2-bytes**, **unsigned**
  - represents a character in the *Unicode character set*
  - **Unicode character set** a mapping from numbers to abstract representations (see "Things to Notes about Programming.md" for more)

try/catch/finally

Reference

Evans, Benjamin J; Flanagan, David (2014-10-16). Java in a Nutshell O'Reilly Media. Kindle Edition.

Basics

The `try` block can have zero or more `catch` blocks and zero or one `finally` block, but it must have at least one `catch` *or* `finally`.

If the method does not contain an exception handler that can handle the exception thrown by the `throw` statement, the interpreter stops running the current method and returns to the caller. Now the interpreter starts looking for an exception handler in the

blocks of code of the calling method. If the exception is never caught, it propagates all the way up to the `main()` method of the program. If it is not handled in that method, the Java interpreter prints an error message, prints a stack trace to indicate where the exception occurred, and then exits.

If control leaves the `try` block because of a `return`, `continue`, or `break` statement, the `finally` block is executed before control transfers to its new destination. If there is no local `catch` block to handle the exception, control transfers first to the `finally` block, and then propagates up to the nearest containing `catch` clause that can handle the exception. If a `finally` clause `throws` an exception, that exception replaces any exception that was in the process of being thrown.

Try with resources

You used to have to do it this way

```
InputStream input = null;
try {
    input = new FileInputStream("file.txt");
    int data = input.read();
    // etc.
}
finally {
    if (input != null) {
        input.close();
    }
}
```

Now you can do it this way, which calls `close()` automagically

```
try (FileInputStream input = new FileInputStream("file.txt")) {
    int data = input.read();
    // etc.
}
```

You may want to use *multiple* resources

```
try ( FileInputStream input = new FileInputStream("file.txt");
      BufferedInputStream buffInput = new BufferedInputStream(input)
{
    int data = buffInput.read();
    // etc.
}
```

You can make your own resource work in the `try-with-resources` construct by *implementing* `Autocloseable` and *overriding* `public void close() throws Exception { ... }`

Packages

- These are for putting your objects inside another layer of namespacing
- To put classes into a package, put the name of the package at the top of the source file, *before* any code.
- With no package declaration, your code is in the "default package" (no package name)
- Put files in *package* `com.petski.ethan` in *directory* `com/petski/ethan`
    - If you don't, *it will* ***compile but not run!***

Factory Methods

Why?

1. You can't name constructors
2. A constructor can only return a single type of object

Data types

12/12/14

1. Always use `double` instead of `float`
2. Don't use `char` instead of `String`
    1. `char` uses 16-bit variable-length Unicode.
    2. This means each *code unit* is 16-bits, but some characters require multiple code units.
    3. This is a hassle, and `String` manages all this for you.
3. Don't use `short` unless you need to save the space
4. **Don't use doubles for finance**, use **`BigDecimal`**
5. The reason strings are immutable is so that the compiler can *share* them
    1. Because most of the time you are comparing, not changing, strings
6. For dates, use the `GregorianCalendar` class instead of `Date`

Initialization

Javaworld description

1. Instance and class variables** are **given initial values** of **`0 / 0.0 / false / null`**.
2. **Local variables** are **not given initial values** and **must be initialized explicitly before they are used**. This is true for local object references too, not only primitive types.
3. The Java compiler will not let you use the value of an uninitialized variable

Initialization Blocks

I got this from Stack Overflow

1. **Static initialization block** --- **runs *once***, when the class is *initialized*
2. **Instance initialization block** --- **runs before the constructor** every time you instantiate an object
3. I'm pretty sure you can have as many of either type of block as you want

Example

```
public class InitializationBlocks {
    static int staticVar;
    int instanceVar;

    // 1. Static initialization block
    static {
        staticVar = 5;
    }

    // 2. Instance initialization block
    {
        instanceVar = 7;
    }
}
```

JAR

5/20/14

JAR Wikipedia

- **J**ava **AR**chive (compressed collection of Java files)
- Used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute applicatoin software or libraries on the Java platform.
- Built on the *ZIP* file format

- Allows Java runtimes to efficiently deploy a set of classes and their associated resources
- Optional **manifest file** can add metadata like
    - dependencies on other JARs (`Class-Path: `)
    - digital signatures
    - Which class to call `main()` on to start the application (`Main-Class: `)
    - Versioning (`Specification-Version: "1.2"`)
- **WAR** (**W**eb application **AR**chive) -- a JAR used to deliver Java server stuff and static web pages that together constitute a web application.

Equals and HashCode

5/12/14

SO

Equals

Must be

- reflexive -- `a.equals(a) == true`
- symmetric -- `a.equals(b) iff b.equals(a)`
- transitive -- `a.equals(b) && b.equals(c)` $\implies$ `a.equals(c)`
- consistent -- always returns the same value for the same [unchanged] object

If you *don't* `@Override public boolean equals(Object o){}`, each instance is *equal only to itself*. If this is what you want: *don't override* `equals` (e.g. `Thread`).

Here is an industrial-strength example from *Core Java, Vol. 1, 9th Ed.*

```
class Employee {
    private String name;
    private int salary;
    private Date hireDay;
    ...
    @Override public boolean equals(Object otherObj) {

        /* same reference */
        if (this == otherObj) return true;

        /* comparison with null */
        if (otherObj == null) return false;

        /* classes differ */
        if (getClass() != otherObj.getClass())
            return false;

        /* cast to class */
        Employee other = (Employee) otherObj;

        /* recursively equavalent fields */
        return Object.equals(name, other.name)
            && salary == other.salary
            && Object.equals(hireDay, other.hireDay);

        /* Object.equals works even if one or both args are null,
         * which would *not* work using hireDay.equals(other.hireDa
         */
    }
}
```

This example is for a subclass

```
class Manager extends Employee {
    private int bonus;
    ...
    @Override public boolean equals(Object otherObj) {
        if (!super.equals(otherObj)) return false;
        Manager other = (Manager) otherObj;
        return bonus == other.bonus;
    }
}
```

Hash Code

Rules

- `a.equals(b)` $\Longrightarrow$ `a.hashCode() == b.hashCode()`

Useful things for interview questions

Copy raw array

Arrays.copyOfRange(oldArr, fromInd, toInd)

```
int[] oldArr = {3, 4, 5};
int[] first2 = Arrays.copyOfRange(oldArr, 0, 2);
```

String to char[]

myString.toCharArray()

```
char[] charArr = "My String".toCharArray();
```

Bonus: now we may use a *foreach* loop

```
for (char c : charArr) { ... }
```

Asides

Objects are References

- The following code is *incorrect*

    ```
    class A {
        Obj obj;
        public Obj getObj() {
            return obj;
        }
    }
    ```

    because anyone can *modify* the returned *reference* to `obj`!

    The proper thing to do is

```
public Obj getObj() {
    return obj.clone();
}
```

- If you say

  ```
  private final MutableObj obj;
  ```

  then `obj` will always refer to the same instance of `MutableObj`, however the instance itself is still mutable.

  - This does not apply to `Strings`, which are immutable

Source files

- You can only have *one `public` class in a source file*, but you can have any number of *nonpublic* classes.

`this` keyword

It can be used to access enclosing instances from within a nested class:

```
public class MyClass {
    String name = "asd";
    private class InnerClass {
        String name = "bte";
        public String getOuterName() {
            return MyClass.this.name; // "asd"
        }
    }
}
```

Awesome snippet for dispatching from enum instead of using switch statement

From Stackoverflow

```
enum MyEnum {
    SOME_ENUM_CONSTANT {
        @Override
        public void method() {
            System.out.println("first enum constant behavior!");
        }
    },
    ANOTHER_ENUM_CONSTANT {
        @Override
        public void method() {
            System.out.println("second enum constant behavior!");
        }
    }; // note the semi-colon after the final constant, not just a c
    public abstract void method(); // could also be in an interface
}

void aMethodSomewhere(final MyEnum e) {
    doSomeStuff();
    e.method(); // here is where the switch would be, now it's one l
    doSomeOtherStuff();
}
```

Java from 2,000 feet

This stuff is largely from *Horstmann, Cay S.; Cornell, Gary (2012-11-27). Core Java Volume I-- Fundamentals (9th Edition) (Core Series Book 1). Pearson Education. Kindle Edition.*

Java History

In late 1995, the Java programming language burst onto the Internet scene and gained instant celebrity status. It promised to become the universal glue between users, web servers, databases, etc. Indeed, Java is in a unique position to fulfill this promise, having gained acceptance by all major vendors except for Microsoft. It has great built-in security and safety features. It has built-in support for advanced programming tasks, such as network programming, database connectivity, and multithreading. Since 1995, eight major revisions of the Java Development Kit have been released. Over the course of the last 17 years, the Application Programming Interface (API) has grown from about 200 to over 3,000 classes. The API now spans such diverse areas as user interface construction, database management, internationalization, security, and XML processing.

Java was never just a language...Java is a whole platform, with a huge library , containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.

Java is intended for writing programs that must be reliable. It eliminates situations that are error-prone. The single biggest difference between Java and C/ ++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

Java makes it extremely difficult to outwit its security mechanisms. The bugs found so far have been very technical and few in number.

Unlike C and C++, there are no “implementation-dependent” aspects of the specification. The sizes of the primitive data types are specified , as is the behavior of arithmetic on them. Having a fixed size for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. Strings are saved in a standard Unicode format.

The ease of multithreading is one of the main reasons why Java is such an appealing language for server-side development.

Besides Java "Standard Edition" (SE), there are two other editions: Micro Edition for embedded devices such as cell phones, and Enterprise Edition for server-side processing. I'm only familiar with SE.

Java is successful because its class libraries let you easily do things that were hard before, such as networking and multithreading. The fact that Java reduces pointer errors is a bonus, so programmers seem to be more productive with Java— but these factors are not the source of its success.

Only use C# if you're tied to Windows, because you'll lose the security and platform independence.

in 2007, when Sun announced that future versions of Java will be available under the General Public License (GPL), the same open source license that is used by Linux. There is only one fly in the ointment—patents. Everyone is given a patent grant to use and modify Java, subject to the GPL, but only on desktop and server platforms. If you want to use Java in embedded systems, you need a different license and will likely need to pay royalties. However, these patents will expire within the next decade, and at that point Java will be entirely free.

To our knowledge, no actual Java systems were ever compromised. To keep this in perspective, consider the literally millions of virus attacks in Windows executable files and Word macros. Even 15 years after its creation, Java is far safer than any other commonly available execution platform.

Java vs. C++

- The major difference between Java and C++ lies in *multiple inheritance*, which Java has *replaced with* the simpler concept of *interfaces*, and in the Java

metaclass model.

- The “hot spots” of your code will run just as fast in Java as they would in C++, and in some cases even faster. Java does have some additional overhead over C++. Virtual machine startup time is slow, and Java GUIs are slower than their native counterparts because they are painted in a platform- independent manner. A slow Java program will still run quite a bit better today than those blazingly fast C++ programs did a few years ago.
- In Java, you don't explicitly return an exit code. It simply returns 0 when it ends. You *can* return an exit code to the Operating System though via `System.exit(int)`.
- Java has no unsigned types.
- C++ strings are mutable, Java's are *immutable*
    - Note that boxed primitive types (e.g. `Integer`) are *immutable* as well
- C++ strings can be compared with ==, Java strings must be compared with `"".equals("")`
- In C++, one may redefine a variable inside a nested block. The inner definition then shadows the outer one. Java does not permit this.

```
int n;
if (...) {
    int n; // C++ YES, Java NO
}
```

- C++ allows you to define a copy constructor or copy assignment which actually copies the data. To do this in Java you'd override `Object Object.clone(void)` and then do `MyClass copy = (MyClass) myObj.clone();`.
- In Java there's no way to specify a method as `const` (i.e. access but not mutate)
- Parameters in Java are *always* passed the same way, you can't choose
    - ***Primitives** are passed **by value***
    - ***Objects** are passed **by something similar to reference***
        - It's different though because it's actually passed by, say, **reference value**
        - This means you can't do `void swap(Obj a, Obj b)` because the references in the `swap` method are copies of the original.
        - But if you mutate an object passed in, you're really affecting the caller's object.
- Since Java does garbage collection, there are no destructors

    - However your object might be using something other than heap space, like a file handle, in which case you want to reclaim/recycle it
    - So write a `finalize(void)` method, which will be called by the garbage collector
    - Since you don't know when your object will be GC'd, you don't know when this method will be invoked.
    - If you need to release resources on demand, define a `close(void)` method and call it when you want.
- Like C++, methods have *covariant* return types

```
public class Employee {
    public Employee getBuddy() {...}
}
public class Manager extends Employee {
    public Manager getBuddy() {...}
}
```

In this case, `Manager`'s method successfully *overrides* `Employee`'s