Programming Languages and Techniques (CIS1200)

Lecture 22

Java: Objects, Interfaces

Chapters 19 & 20

Announcements

- HW05: GUI programming
  - Due: tomorrow at 11.59pm
- Java Bootcamp / Refresher: Sunday, October 27
  - 1-3pm, Towne 100
  - Will be recorded
  - Look for more details on Ed
- HW06: Pennstagram
  - Java array programming
  - Available soon
  - Due *Thursday*, October 31st at 11.59pm

19: How far along are you in HW05: GUI Programming?

0

Not started yet	0%
Task 0 finished	0%
Working on tasks 1-4	0%
Working on Task 5	0%
Working on Task 6	0%
All done!	0%

Review: Java Core Language

differences between OCaml and Java

Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

Expression form	Example	Type
Variable reference	x	Declared type of variable
Object creation	new Counter ()	Class of the object
Method call	c.inc()	Return type of method
Equality test	x == y	boolean
Assignment	x = 5	don’t use as an expression!!

Type System Organization

	OCaml	**Java**
primitive types (values stored "directly" in the stack)	int, float, char, bool, …	int, float, double, char, boolean, …
structured types (a.k.a. *reference types* — values stored in the heap)	tuples, datatypes, records, functions, arrays (*objects encoded as records of functions*)	objects, arrays (*records, tuples, datatypes, strings, first-class functions are special cases of objects*)
generics	'a list	List<A>
abstract types	module types (signatures)	interfaces, abstract classes, public/private modifiers

Arithmetic & Logical Operators

<table>
<tr><th>OCaml</th><th>Java</th><th></th></tr>
<tr><td>=, ==</td><td>==</td><td>equality test</td></tr>
<tr><td><>, !=</td><td>!=</td><td>inequality</td></tr>
<tr><td>>, >=, <, <=</td><td>>, >=, <, <=</td><td>comparisons</td></tr>
<tr><td>+</td><td>+</td><td>addition</td></tr>
<tr><td>^</td><td>+</td><td>string concatenation</td></tr>
<tr><td></td><td></td><td>subtraction (and unary minus)</td></tr>
<tr><td>*</td><td>*</td><td>multiplication</td></tr>
<tr><td>/</td><td>/</td><td>division</td></tr>
<tr><td>mod</td><td>%</td><td>remainder (modulus)</td></tr>
<tr><td>not</td><td>!</td><td>logical “not”</td></tr>
<tr><td>&&</td><td>&&</td><td>logical “and” (short-circuiting)</td></tr>
<tr><td>||</td><td>||</td><td>logical “or” (short-circuiting)</td></tr>
</table>

Java: Operator Overloading

- The *meaning* of an operator in Java is determined by the *types* of the values it operates on:
  - Integer division

    `4/3` ⇒ `1`
  - Floating point division

    `4.0/3.0` ⇒ `1.3333333333333333`
  - Automatic conversion from int to float

    `4/3.0` ⇒ `1.3333333333333333`

- Method *overloading* is a general mechanism in Java
  - we'll see more of it later

Equality

- like OCaml, Java has two ways of testing reference types for equality:
  - "reference equality"

    o1 == o2
  - "deep equality"

    o1.equals(o2)

every object provides an "equals" method that should "do the right thing" depending on the class of the object

- Normally, you should use == to compare primitive types and ".equals" to compare objects

- Careful: Single-equals (=) means assignment, not equality comparison

Strings

- `String` is a *built in* Java class
- Strings are sequences of (unicode) characters

  `""` `"Java"` `"3 Stooges"` `"富士山"`
- + means String concatenation (overloaded)

  `"3" + " " + "Stooges"` ⇒ `"3 Stooges"`
- Text in a String is immutable (like OCaml)
  - but variables that store strings are not
  - `String x = "OCaml";`
  - `String y = x;`
  - Immutability: can't do anything to `x` so that `y` changes
- The `.equals` method returns true when two strings contain the *same* sequence of characters

Aside: StringBuffers

- `StringBuffer` is a *mutable* Java String
- Alternative to "+" when constructing large strings

```
String s = "Hello";
for (int i=0; i<200; i++) {
  s = s + "!";
}
```

```
StringBuffer sb = new StringBuffer("Hello");
for (int i=0; i<200; i++) {
  sb.append("!");  // modify end of sb
}
String s = sb.toString();  // convert back to String
```

21: What is the value of *ans* at the end of this program?

0

true	0%
false	0%
NullPointerException	0%

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200" ;
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true

This is the preferred method of comparing strings!

21: What is the value of *ans* at the end of this program?

0

true

0%

false

0%

NullPointerException

0%

What is the value of ans at the end of this program?

```
String x1 = "CIS ";
String x2 = "1200";
String x = x1 + x2;
String z = "CIS 1200";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: false

Even though x and z both contain the characters “CIS 1200”, they are stored in two different locations in the heap.

21: What is the value of *ans* at the end of this program?

0

true

0%

false

0%

NullPointerException

0%

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: true(!)
Why? Since strings are immutable, two identical
strings that are known when the program is compiled can be
aliased by the compiler (to save space).

Moral

Always use s1.equals(s2) to compare Strings!

Compare strings with respect to their content, not where they happen to be allocated in memory…

Object Oriented Programming

Preview: The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - dynamic dispatch (“receiver” of method call determines behavior)
  - classes (“templates” for object creation)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)

encapsulated state

- Good for:
  - GUIs
    - complex software systems that include many different implementations of the same “interface” (set of operations) with different behaviors
  - Simulations
    - designs with an explicit correspondence between “objects” in the computer and things in the real world
  - Games

"Objects" in OCaml

```
(* The type of counter objects *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

(* Create a counter "object" *)
let new_counter () : counter =
  let r = {contents = 0} in
  {
    inc = (fun () ->
      r.contents <- r.contents + 1;
      r.contents);
    dec = (fun () ->
      r.contents <- r.contents - 1;
      r.contents)
 }
```

Why is this an object?

- *Encapsulated local state* only visible to the methods of the object
- Object is *defined by what it can do*—local state does not appear in the interface
- There is a way to *construct* new object values that behave similarly

OO terminology

- *Object*: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*
- *Class*: a template for creating objects
- The class of an object specifies…
  - the types and initial values of its local state (fields)
  - the set of operations that can be performed on the object (methods)
  - one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)
- Every (Java) object is an *instance* of some class
  - Instances are created by invoking a constructor with the `new` keyword

OO programming

OCaml (part we've seen)

- Explicitly create objects using a record of higher order functions and hidden state
- Flexibility through ***composition***: objects can only implement one interface

```
type button =
    widget *
    label_controller *
    notifier_controller
```

Java (and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)
- Flexibility through ***extension***: ***Subtyping*** allows related objects to share a common interface

```
class Button extends Widget {
  /* Button is a subtype
     of Widget */

}
```

Objects in Java

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

class declaration

class name

field / instance variable

constructor

methods

object creation and use

```
public class Main {

  public static void
     main (String[] args) {

       Counter c = new Counter();

       System.out.println( c.inc() );

     }
}
```

constructor invocation

method call

Encapsulating local state

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

r is *private*

constructor and methods can refer to r

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

    }
}
```

other parts of the program can only access public members

method call

Encapsulating local state

- *Visibility modifiers* make the state local by controlling access
- Two levels of visibility*:
  - public : accessible from anywhere in the program
  - private : only accessible inside the class
- Design pattern — first cut:
  - Make *all* fields private
  - Make constructors and (non-helper) methods public

*Java offers a couple of other protection levels — "protected" and "package protected". These are not important at this point.

Constructors with Parameters

```
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {

  public static void
      main (String[] args) {

        Counter c = new Counter(3);

        System.out.println( c.inc() );

    }
}
```

constructor invocation

Creating Objects

- *Declare* a variable to hold a `Counter` object
  - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c = new Counter();
```

Creating Objects

- Every Java variable is mutable

```
Counter c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type can also contain the special value "null"

```
Counter c = null;
```

☞ Single = for assignment
Double == for reference equality testing

22: What is the value of *ans* at the end of this program?

0

1	0%
2	0%
3	0%
Program raises NullPointerException	0%

What is the value of ans at the end of this program?

```
Counter x;
x.inc();
int ans = x.inc();
```

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

1. 1
2. 2
3. 3
4. Program raises `NullPointerException`

Answer: Program raises `NullPointerException`

22: What is the value of *ans* at the end of this program?

0

1	0%
2	0%
3	0%
Program raises NullPointerException	0%

What is the value of ans at the end of this program?

```
Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Program raises `NullPointerException`

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

Answer: 3

Interfaces

Working with objects abstractly

“Objects” in OCaml vs. Java

OCaml

```
(* The type of “objects” *)
type point = {
    getX  : unit -> int;
    getY  : unit -> int;
    move  : int*int -> unit;
}

(* Create an "object" with
   hidden state: *)
type position =
  { mutable x: int;
    mutable y: int; }

let new_point () : point =
  let r = {x = 0; y=0} in {
    getX = (fun () -> r.x);
    getY = (fun () -> r.y);
    move = (fun (dx,dy) ->
              r.x <- r.x + dx;
              r.y <- r.y + dy)
}
```

Type is separate from the implementation

Java

```
public class Point {

  private int x;
  private int y;

  public Point () {
    x = 0;
    y = 0;
  }
  public int getX () {
    return x;
  }
  public int getY () {
    return y;
  }
  public void move
         (int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}
```

Class specifies *both* type and implementation of object values

Interfaces

- Give a *type* for an object based on how it can be *used*, not on how it was *constructed*
- Describe a *contract* that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {
  int getX();
  int getY();
  void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!

Implementing the interface

- A class that *implements* an interface provides appropriate definitions for the methods specified in the interface
- The class fulfills the contract implicit in the interface

```
public class Point implements Displaceable {
   private int x, y;
   public Point(int x0, int y0) {
      x = x0;
      y = y0;
   }
   public int getX() { return x; }
   public int getY() { return y; }
   public void move(int dx, int dy) {
      x = x + dx;
      y = y + dy;
   }
}
```

interfaces implemented

methods required to satisfy contract

Another implementation

```
public class Circle implements Displaceable {
  private Point center;
  private int radius;
  public Circle(Point initCenter, int initRadius) {
    center = initCenter;
    radius = initRadius;
  }
  public int getX() { return center.getX(); }
  public int getY() { return center.getY(); }
  public void move(int dx, int dy) {
    center.move(dx, dy);
  }
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center

Yet another implementation

```
public class ColoredPoint implements Displaceable {
  private Point p;
  private Color c;
  public ColoredPoint (int x0, int y0, Color c0) {
     p = new Point(x0,y0);
     c = c0;
  }
  public void move(int dx, int dy) {
     p.move(dx, dy);
  }
  public int getX() { return p.getX(); }

  public int getY() { return p.getY(); }

  public Color getColor() { return c; }
}
```

Flexibility: Classes may contain more methods than interface requires

Interfaces are types

- Can declare variables and method params with interface type

```
void m (Displaceable d) { … }
```

- Can call m with any `Displaceable` argument…

```
obj.m(new Point(3,4));
obj.m(new ColoredPoint(1,2,Color.Black));
```

- … but m can only operate on d according to the interface

```
  d.move(-1,1);
…
… d.getX() …          ⇒ 0
… d.getY() …          ⇒ 3
```

Using interface types

- Variables with interface types can refer, at run time, to objects of any class that implements the interface
- Point and Circle are *subtypes* of Displaceable

```
Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2,3), 1);
d2 = new ColoredPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
d2.move(-2,0);
…
… d0.getX() …      ⇒ -1
… d1.getX() …      ⇒  0
… d2.getX() …      ⇒ -3
```

The class that created the object value determines which move code is executed: *dynamic dispatch*

i.e., run-time

Abstraction

The `Displaceable` interface gives us a single name for all the possible kinds of "moveable things." This allows us to write code that manipulates arbitrary `Displaceable` objects, without caring whether it's dealing with points or circles.

```
public class DoStuff {
  public void moveItALot (Displaceable s) {
    s.move(3,3);
    s.move(100,1000);
    s.move(1000,234651);
  }

  public void dostuff () {
    Displaceable s1 = new Point(5,5);
    Displaceable s2 = new Circle(new Point(0,0),100);
    moveItALot(s1);
    moveItALot(s2);
  }
}
```

Multiple interfaces

- An interface represents a point of view

  ...and there can be multiple valid points of view on a given object

- Example: Geometric objects
  - All can move (are `Displaceable`)
  - Some have Color (are `Colored`)

Colored interface

- Contract for objects that that have a color
  - Circles and Points don't implement Colored
  - ColoredPoints do

```
public interface Colored {
    public Color getColor();
}
```

ColoredPoints

```
public class ColoredPoint
 implements Displaceable, Colored {

   … // previous members

   private Color color;
   public Color getColor() {
     return color;
   }


   …
}
```

“Datatypes” in Java

OCaml

```
type shape =
    | Point of …
    | Circle of …

let draw_shape (s:shape) =
   begin match s with
    | Point … -> …
    | Circle … -> …
   end
```

Java

```
interface Shape {
  void draw();
}
class Point implements Shape {
    …
  public void draw() {
    …
  }
}
class Circle implements Shape {
   …
  public void draw() {
   …
  }
}
```

Recap

- **Object**: A collection of related *fields* (or *instance variables*) and *methods* that operate on those fields
- **Class**: A template for creating objects, specifying
  - types and initial values of fields
  - code for methods
  - optionally, a *constructor* that is run each time a new object is created from the class
- **Interface**: A "signature" for objects, describing a collection of methods that must be provided by classes that *implement* the interface
- **Object Type**: Either a class or an interface (meaning "this object was created from a class that implements this interface")

Static Methods

Java Main Entry Point

```
class MainClass {

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

}
```

- Program starts running at `main`
  - `args` is an array of `Strings` (passed in from the command line)
  - must be public
  - returns `void` (i.e. is a command)

- What does *static* mean?

Static method example

```
public class Max {

  public static int max (int x, int y) {
    if (x > y) {
       return x;
    } else {
       return y;
    }
  }

  public static int max3(int x, int y, int z) {
     return max(max(x,y), z);
  }
}
```

closest analogue of top-level functions in OCaml, but must be a member of some class

Internally (within the same class), call with just the method name

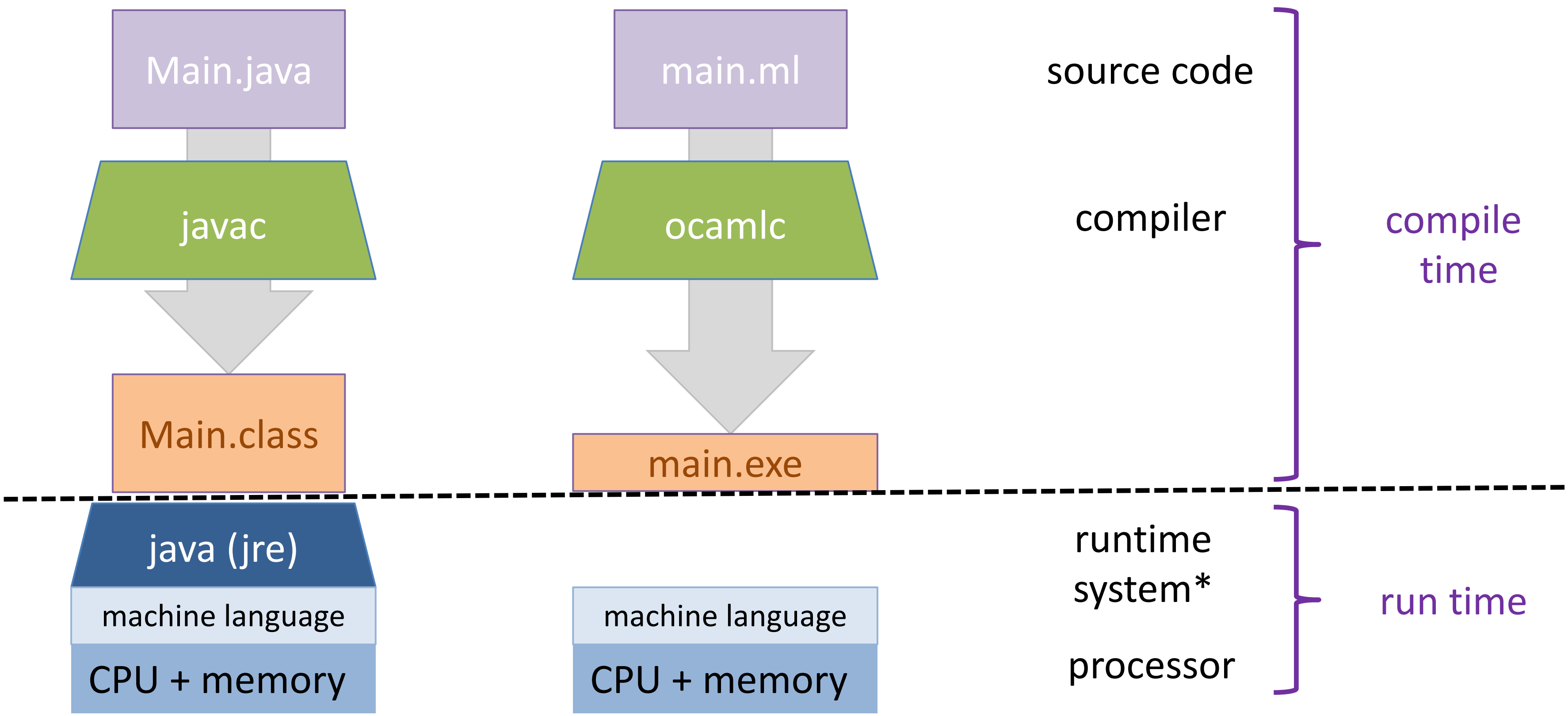
main method must be static; it is invoked to start the program running

```
public class Main {

  public static void main (String[] args) {

       System.out.println(Max.max(3,4));
       return;
    }
}
```

Externally, prefix with name of the class mantra

Static = Decided at *Compile Time*

Dynamic = Decided at *Run Time*



*simplified (e.g., omitting the OS)

Static vs. Dynamic Methods

- Static methods are *independent* of object values
  - Similar to OCaml functions
  - Cannot refer to the local state of objects (fields or normal methods)
- Use static methods for:
  - Non-OO programming
  - Programming with primitive types: Math.sin(60), Integer.toString(3), Boolean.valueOf("true")
  - "`public static void main`"
- "Normal" methods are *dynamic*
  - Need access to the local state of the particular object on which they are invoked
  - We only know at *runtime* which method will get called

```
void moveTwice (Displaceable o) {
   o.move (1,1); o.move(1,1);
}
```

Method call examples

- Calling a (dynamic) method of an object (o) that returns a number:

```
x = o.m() + 5;
```

- Calling a static method of a class (C) that returns a number:

```
x = C.m() + 5;
```

- Calling a method that returns void:

Static
```
C.m();
```
Dynamic
```
o.m();
```

- Calling a static or dynamic method in a method of the same class:

Either
```
m();
```
Static
```
C.m();
```
Dynamic
```
this.m();
```

- Calling (dynamic) methods that return objects:

```
x = o.m().n();
x = o.m().n().x().y().z().a().b().c().d().e();
```

Which **static** method can we add to this class?

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  // A,B, or C here ?

}
```

A.

```
public static int dec () {
   r = r – 1;
   return r;
}
```

B.

```
public static int inc2 () {
   inc();
   return inc();
}
```

C.

```
public static int getInitialVal () {
   return 0;
}
```

Answer: C

Static Fields

Static vs. Dynamic Class Members

```java
public class FancyCounter {
  private int c = 0;
  private static int total = 0;

  public int inc () {
    c += 1;
    total += 1;
    return c;
  }

  public static int getTotal () {
    return total;
  }
}
```

```java
FancyCounter c1 = new FancyCounter();
FancyCounter c2 = new FancyCounter();
int v1 = c1.inc();
int v2 = c2.inc();
int v3 = c1.getTotal();
System.out.println(v1 + " " + v2 + " " + v3);
```

Static Class Members

- Static methods can depend *only* on other static things
  - Static fields and methods, from the same or other classes
- Static methods *can* create *new* objects and use them
  - This is typically how `main` works
- `public static` fields are the "global" state of the program
  - Mutable global state should generally be avoided
  - Immutable global fields are useful for constants

```
public static final double PI = 3.14159265359793238462643383279;
```

Style: naming conventions

Kind	Part-of-speech	Example
class	noun	RacingCar
field / variable	noun	initialSpeed
static final field (constants)	noun	MILES_PER_GALLON
method	verb	shiftGear

- Identifiers consist of alphanumeric characters and _ and cannot start with a digit
- The larger the scope, the more *informative* the name should be
- Conventions are important: variables, methods and classes can have the same name

Why naming conventions matter

```
public class Turtle {
  private Turtle Turtle;
  public Turtle() { }

  public Turtle Turtle (Turtle Turtle) {
    return Turtle;
  }
}
```

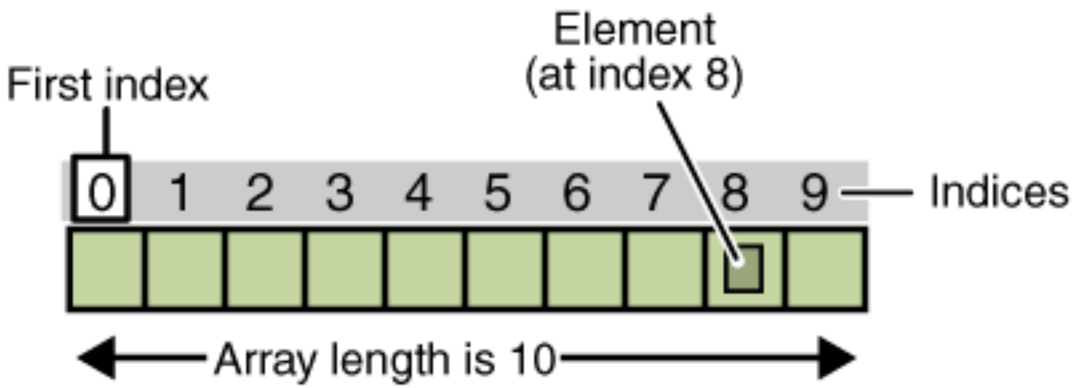
Many more details on good Java style here:
http://www.seas.upenn.edu/~cis1200/current/java_style.shtml

Java Arrays

Working with static methods

Java Arrays: Indexing

- An array is a sequentially ordered collection of values that can be indexed in *constant* time
- Index elements from 0



- Basic array expression forms

`a[i]`	access element of array `a` at index `i`
`a[i] = e`	assign `e` to element of array `a` at index `i`
`a.length`	get the number of elements in `a`

Java Arrays: Creation

- Create an array a of size n with elements of type C, initialized with default values

```
C[] a = new C[n];
```

- Create an array with given initial values

```
C[] a = new C[] { new C(1), new C(2) };
```

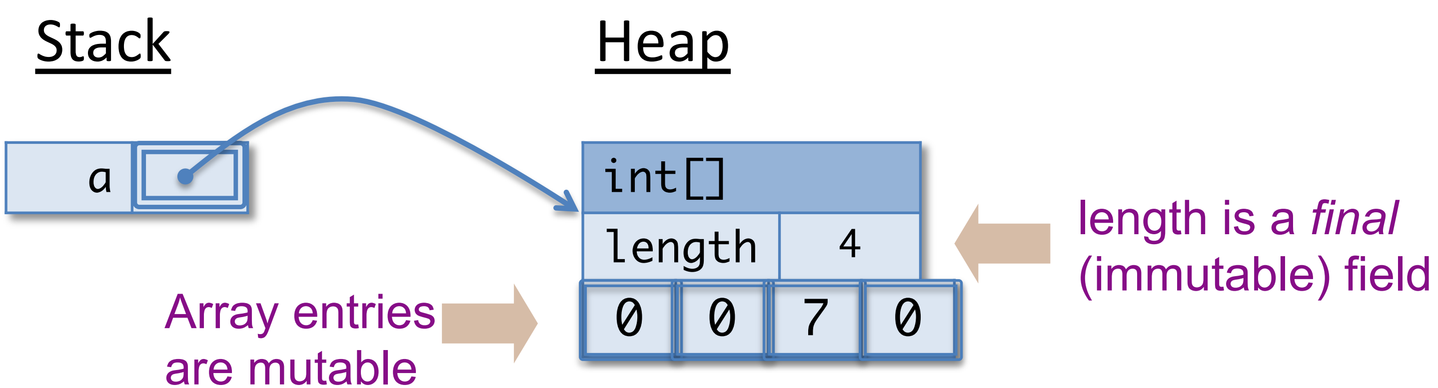
- When initializing a variable can omit `new` keyword and type

```
C[] a = { new C(1), new C(2) };
```

Java Arrays: Java ASM

- Arrays live in the heap; values with array type are mutable references

```
int[] a = new int[4];
a[2] = 7;
```



Java Arrays: Aliasing

- Variables of array type are references and can be aliases

```
int[] a = new int[4];
int[] b = a;
a[2] = 7;
int ans = b[2];
```