CS131

Saketh Kasibatla

Today

HW2

Intro to Java, object oriented languages

inheritance

concurrency, java memory model

This Class

homework 2 is due **tonight at 23:55**

homework 3 is due Monday Oct. 30 at 23:55

This homework will be graded with automated scripts

- not compiling → no credit
- function signatures must match, or no credit
- Look at the tests to figure out what function type signatures should be
- your code should behave exactly according to the spec
- if developing locally, test your homework out on SEASnet before submitting
- check Piazza for clarifications

Questions about HW2

Install/use Java

install JDK @ http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

eclipse if you prefer an IDE @ https://www.eclipse.org/

text editor + terminal

- javac Main.java
- java Main

Java Resources

Java tutorial @ https://docs.oracle.com/javase/tutorial/

Javadoc @ https://docs.oracle.com/javase/7/docs/api/

Effective Java @ https://www.amazon.com/Effective-Java-2nd-Joshua-Bloch/dp/0321356683

- good chapter on concurrency

What is an OO language?

According to Alan Kay:

http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented

everything is an object (java doesn’t follow this)

objects communicate by sending and receiving messages

objects have their own memory

every object is an instance of a class (which must be an object)

the class holds shared behavior for its instances

Java

statically typed

- designed with type safety in mind
- code only accesses memory locations it is authorized to access, and only in well defined ways
- is not actually type safe

Java Virtual Machine

- compile java code to bytecode
- bytecode runs on the virtual machine

garbage collected

- don’t have to free memory allocated on the heap

no pointers

Java

Everything's in a class, even the main method

```
public class Main {
  public static void main(String[] args) {
    System.out.println(“hello, world!”);
  }
}
```

Primitives

values of these types are **not** objects

byte, short, int, long

- signed integer types of size 8, 16, 32, 64 bits respectively
- 1, 2, 42, 360L

float, double

- IEEE floating point numbers of size 32, 64 bits respectively
- 1. , 32., 3.14

boolean

- true, false

char

- single 16 bit unicode character
- ‘a’, ‘b’

Example 2

sum all integers from 1 to 10

Arrays

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

hold many values of one type

```
int[] a = new int[42];
System.out.println(a.length); // 42
```

Example 3

vector product

Objects

an object *encapsulates* data

an object has:

- identity
- state (instance variables)
- behavior (methods)
- objects are instances of classes

Class vs. Object

class:

- describes common properties of objects
  - instance variables
  - methods
- you write classes

object:

- properties for a single instance
  - holds values for instance variables
- part of the data and program execution (only exists at runtime)

Example

bank account

Information Hiding

Outside world can’t access object’s implementation except via public methods

a client can’t change an Account’s balance without calling deposit

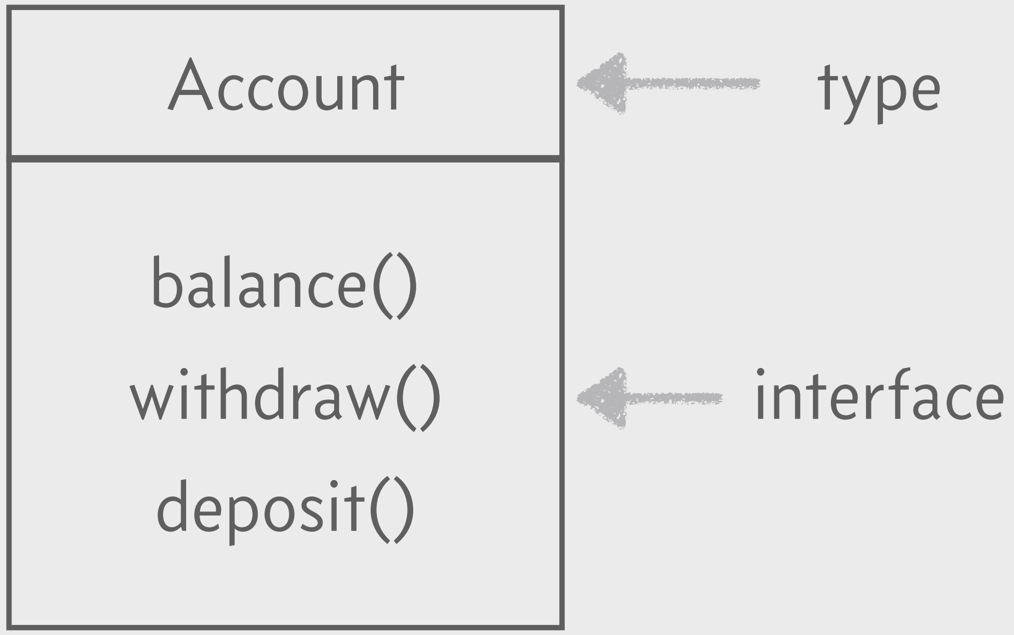
public - anything can see/use this variable/method

private - only instances of the same class can see it

protected - instances of the same class or any subclass can see it

no descriptor - instances of classes from the same package can see it

Type and Interface



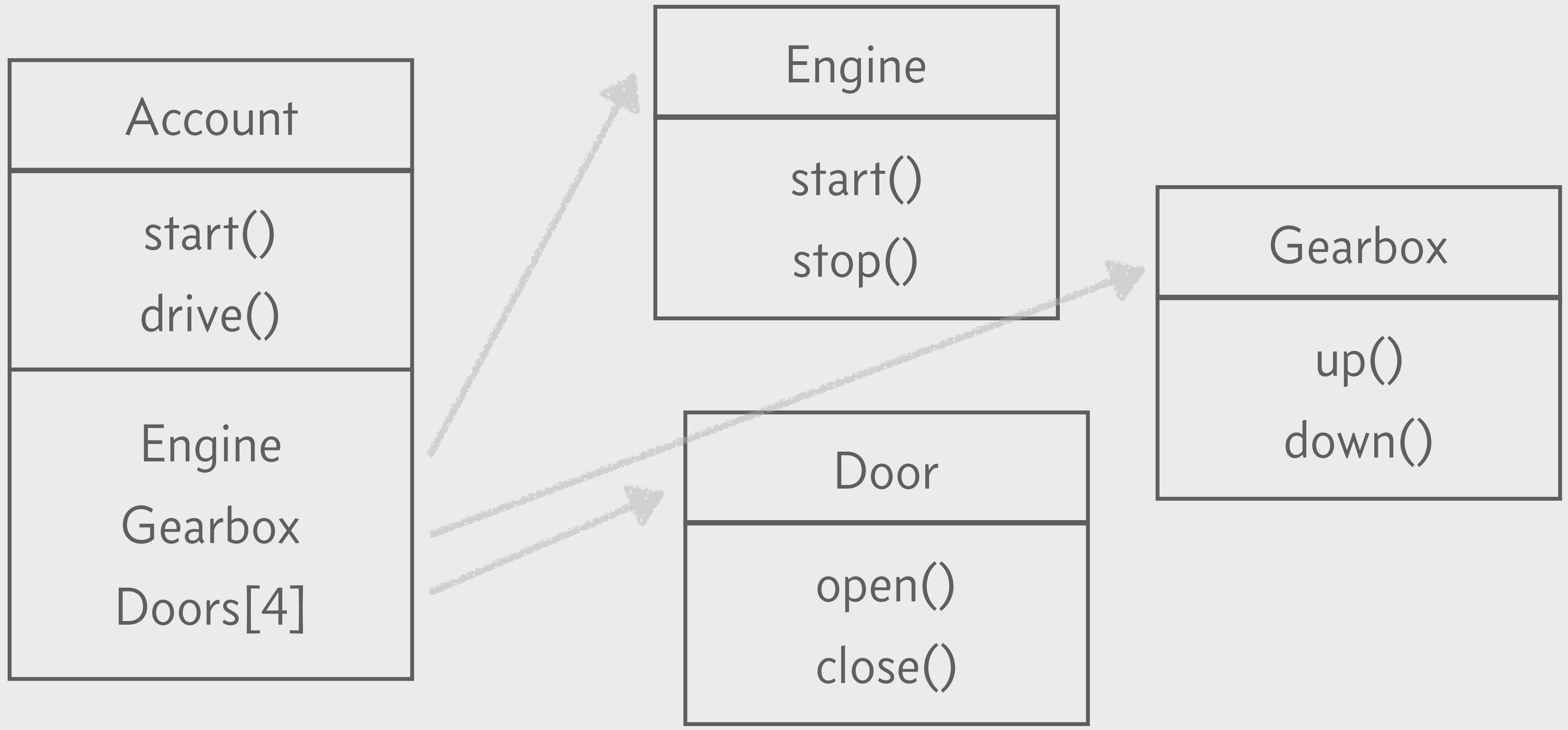
objects of type Account can receive messages according to Account’s interface

```
Account a = new Account(42);
a.balance();
```

Aggregation and Decomposition

Objects have other objects as instance variables

Reuse existing implementations



Inheritance

tool to create a new class based on a previous class

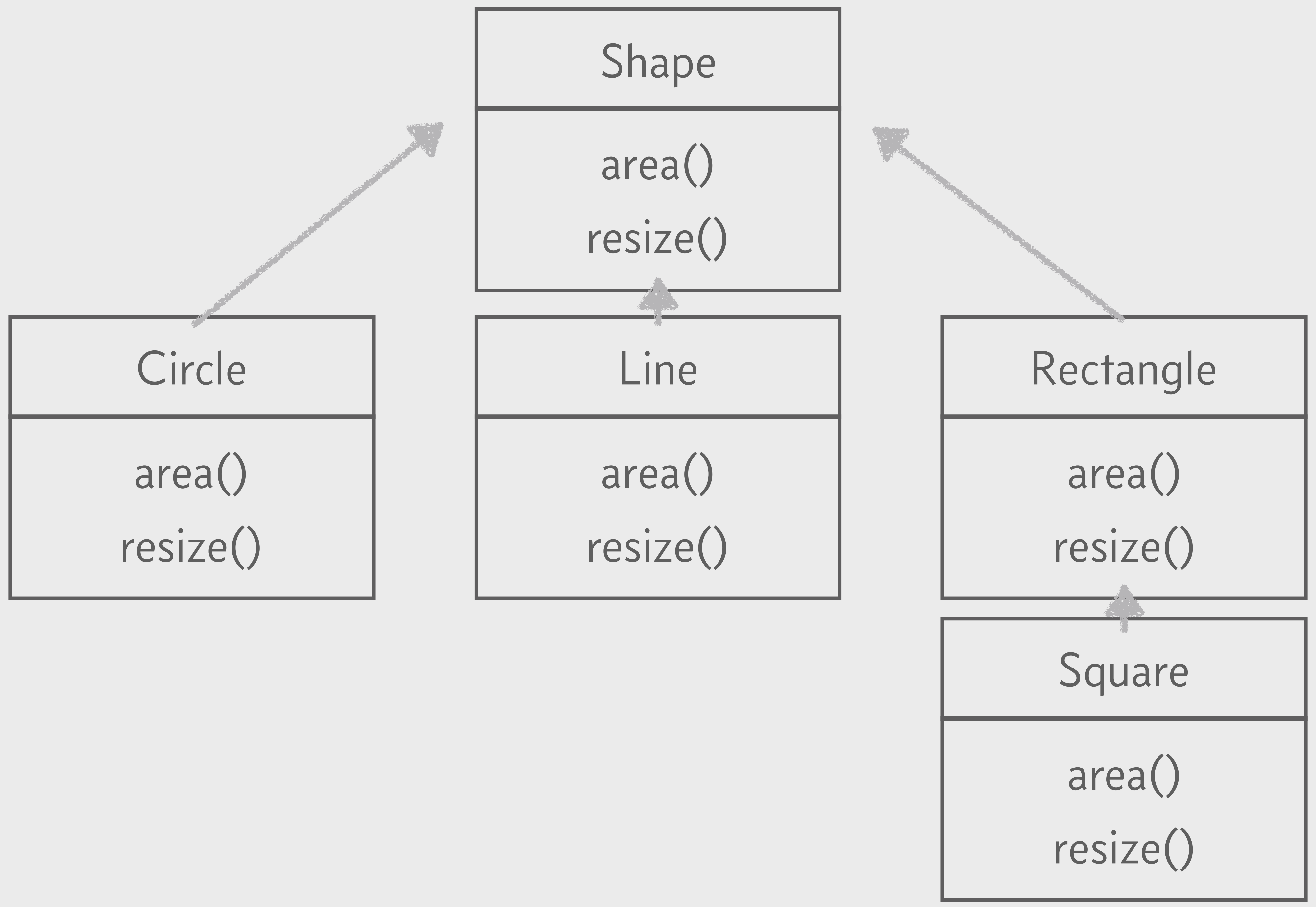
subclass matches the interface of a superclass

```
class Subclass extends Superclass {}
```

used to:

- specialize superclass
- extend superclass functionality (add methods)

Inheritance



Example 5

```
type point =
  Cartesian of (float * float)
| Polar of (float * float)

let negate p = match p with
  Cartesian (x,y) -> Cartesian (-.x, -.y)
| Polar (rho, theta) -> Polar (rho, theta+.180.0)

let toPolar p = match p with
  Polar _ -> p
| Cartesian (x,y) -> Polar(sqrt(x*.x +. y*.y), atan (y /. x))

toPolar (Polar (5., 6.))
toPolar (Cartesian (5., 6.))
negate (Polar (5., 6.))
negate (Cartesian (5., 6.))
```

Let’s write this in Java

Subtyping

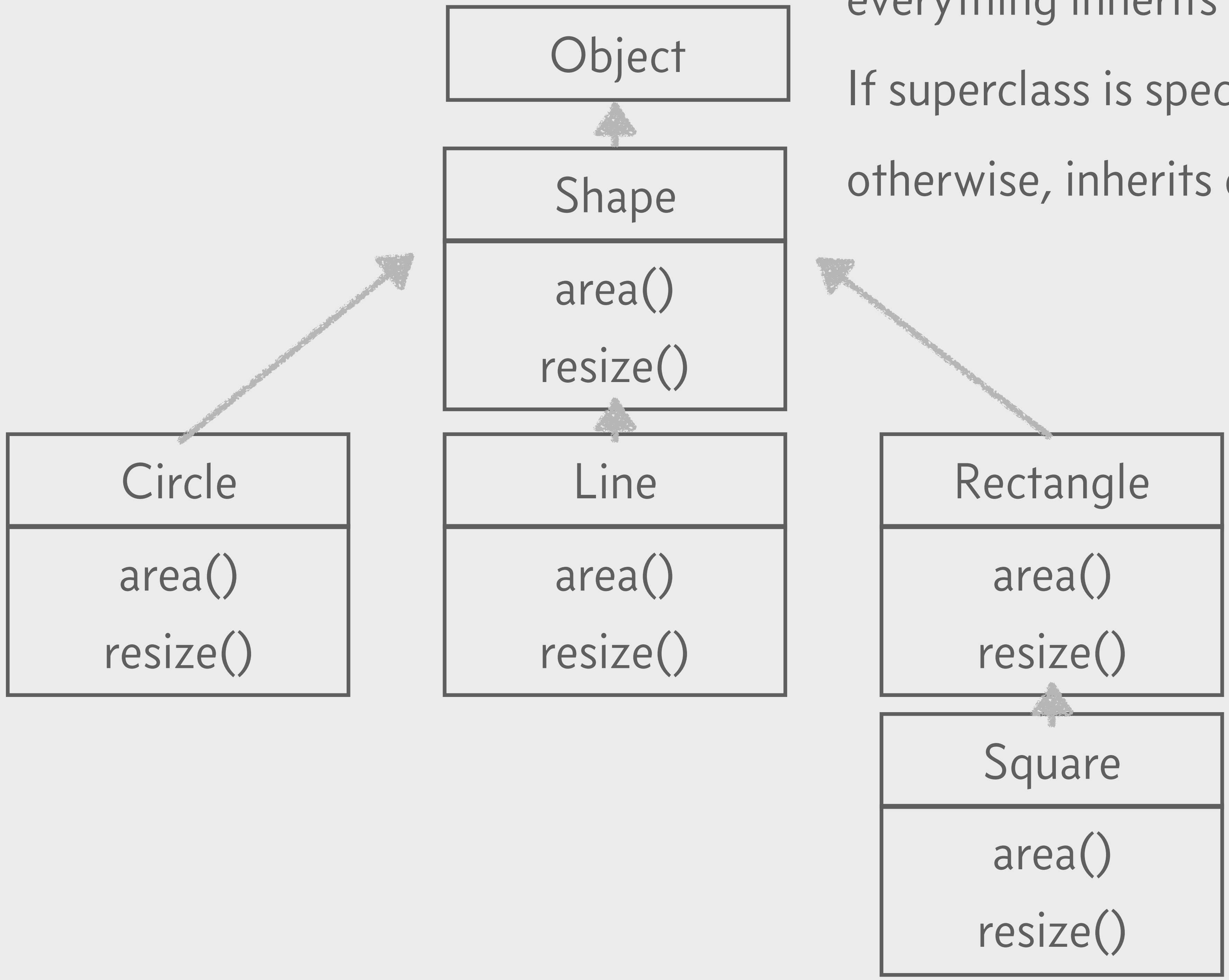
A is a subtype of B → everywhere B can be used, A can be used

subtypes strictly add functionality

3 ways to get subtyping in java

- interface A extends interface B, interface C
- class A implements interface B, interface C
- class A extends class B (can only extend one class)
    - gives interface and implementation

Inheritance



everything inherits from object

If superclass is specified, inherits from superclass

otherwise, inherits directly

Specialization

```
public class Rectangle implements Shape {
  //…
  int area() {
    return this.width * this.height;
  }
  //…
}

public class Square extends Rectangle {
  //…
  int area() {
    return this.width * this.width;
  }
  //…
}
```

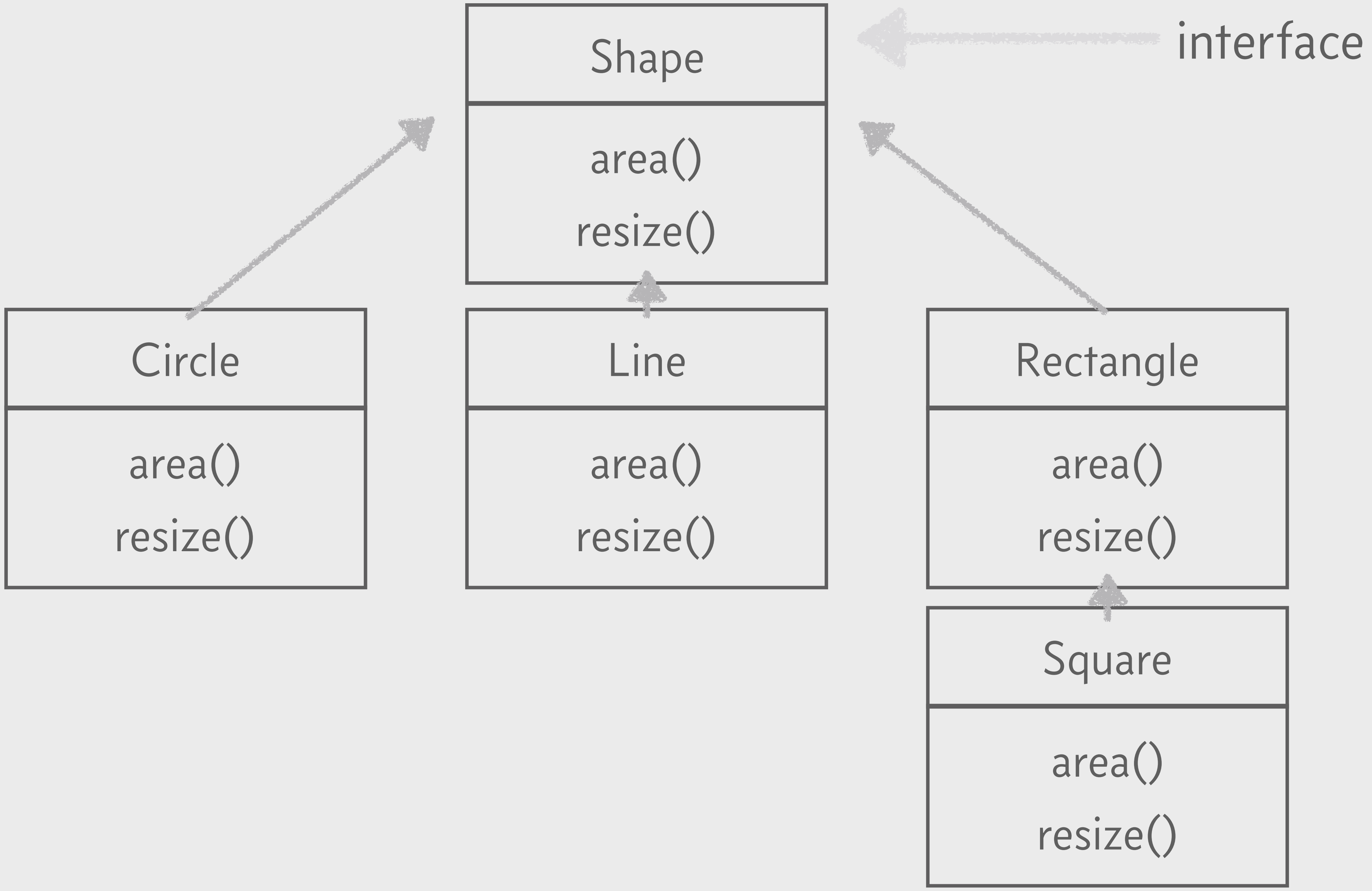
Interfaces

An interface is a collection of constants and method declarations

Has no instance variables or method bodies

```
public interface Runnable {
  public abstract void run();
}
```

Interfaces


Shape
area()
resize()
interface
Circle
area()
resize()
Line
area()
resize()
Rectangle
area()
resize()
Square
area()
resize()

Interfaces

```
public interface Shape {
  double PI = 3.14;
  void draw();  // automatically public, abstract
  void resize();
}

public class Rectangle implements Shape {
  //…
  int area() {
    return this.width * this.height;
  }
  //…
}

public class Square extends Rectangle {
  //…
  int area() {
    return this.width * this.width;
  }
  //…
}
```

← interface

← inheritance

Questions on inheritance/ subtyping

```
// Is Square a subtype of Rectangle?
interface Rectangle {
  float area();
  float perimeter();
}

class Square implements Rectangle {
  private float side;
  public float area() {
    return side * side;
  }

  public float perimeter() {
    return 4 * side;
  }
}
```

Questions on inheritance/ subtyping

how can InstrumentedSet.addAll() return the wrong value?

```
class Set {
  public boolean add(int element) {...}
  public boolean addAll(List<Int> elements) {...}
}

class InstrumentedSet extends Set {
  protected int numberOfAdditions = 0;
  public boolean add(int element) {
    numberOfAdditions++;
    return super.add(element)
  }

  public booelan addAll(List<Int> elements) {
    numberOfAdditions += elements.size();
    return super.addAll(elements);
  }
}
```

Questions on inheritance/ subtyping

What’s a better way to write this code?

```
class Subscription {
  public int SubscriptionPeriod() {
    if (this instanceof PremiumSubscription) {
      return 30;
    } else {
      return 10;
    }
  }
}

class PremiumSubscription extends Subscription { ... }
class RegularSubscription extends Subscription { ... }
```

Memory Model

```
Set s = new ListSet();
// s is on the stack
   the new object, call it o, is on the heap
   s is a pointer to the new object

s.add("hi");
// dereferences s and updates the object o
   to contain the "hi" value

Set s2 = s;
// s2 now points to o
// s and s2 are *aliases*

s2.remove("hi");
// now o no longer contains “hi"
```

Memory Model

all objects are references, allocated on heap

we have garbage collection, so we don’t need to free any pointers

Parameter Passing

pass by value, but most values are objects, which are stored as pointers

```
public class Example7 {
  public static int plus(int a, int b) {
    a += b;
    return a;
  }

  public static void main(String[] args) {
    int x = 3;
    int y = 4;
    int z = plus(x,y);

    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
  }
}
```

Parameter Passing

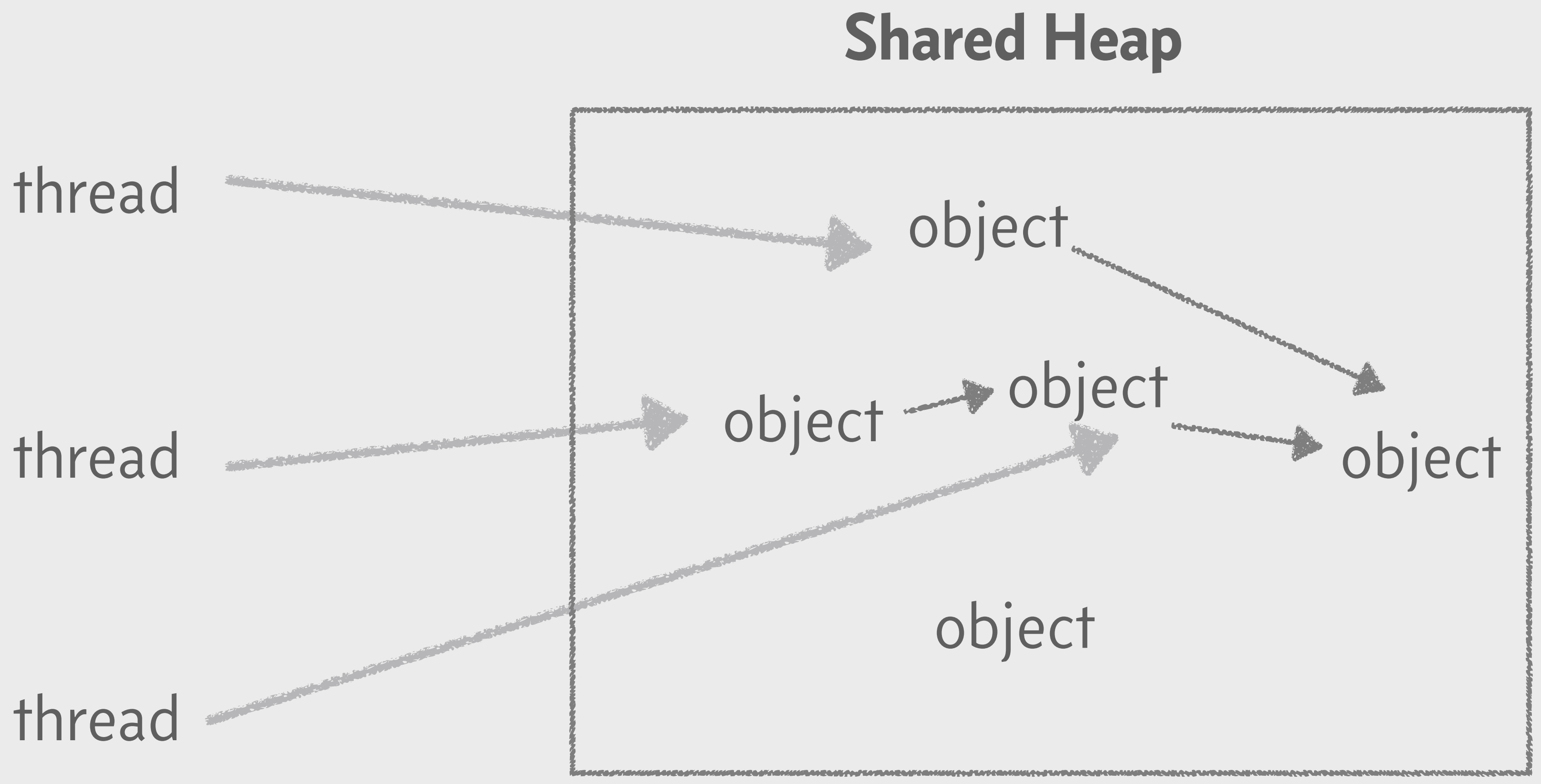
```java
class Integer {
  public int i;
  Integer(int val) { this.i = val; }

  public static Integer plus(Integer a, Integer b) {
    a = new Integer(a.i + b.i);
    return a;
  }
}

public class Example8 {
  public static void main(String[] args) {
    Integer x = new Integer(3);
    Integer y = new Integer(4);
    Integer z = Integer.plus(x,y);

    System.out.println(x.i); // what does this print?
    System.out.println(y.i); // what does this print?
    System.out.println(z.i); // what does this print?
  }
}
```

Parameter Passing

```
class Integer {
  public int i;
  Integer(int val) { this.i = val; }

  public static Integer plus(Integer a, Integer b) {
    a.i += b.i;
    return a;
  }
}

public class Example9 {
  public static void main(String[] args) {
    Integer x = new Integer(3);
    Integer y = new Integer(4);
    Integer z = Integer.plus(x,y);

    System.out.println(x.i); // what does this print?
    System.out.println(y.i); // what does this print?
    System.out.println(z.i); // what does this print?
  }
}
```

Concurrency

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)



Threads

implemented in the Thread class

either subclass the Thread class and override the run method

```
class MyThread extends Thread {
  public void run() {…}
}
(new MyThread()).start()
```

or implement Runnable

```
class MyRunnable implements Runnable {
  public void run() {…}
}
(new Thread(new MyRunnable())).start()
```

Let’s Try using a BankAccount with different threads

What went wrong?

Threads' executions can be interleaved. (Sequential Consistency)

- a: check that balance is OK
- b: check that balance is OK
- a: subtract from balance (new balance: 300)
- b: subtract from balance (new balance: -400)

even worse, the java compiler can sometimes reorder reads for optimization

- http://www.cs.umd.edu/~pugh/java/memoryModel/

DRF Guarantee

For programmers, the JMM claims that correctly synchronised (data race free) programs only have sequentially consistent behaviours, which means that reasoning based on interleaved semantics is valid in this case

http://homepages.inf.ed.ac.uk/da/papers/drfformalization/drfformalization.pdf

Synchronized

Synchronized keyword is used to make sure operations are atomic

```
synchronized int balance() {…}
synchronized int withdraw(int amount) {…}
synchronized int deposit(int amount) {…}
```

no two threads can run withdraw at the same time.

also establishes a ‘happens before’ relationship between methods

```
account.balance()
account.withdraw(500)
```

call to balance will always happen before call to withdraw

Synchronized

Synchronized can also be used as a block

```
synchronized(object) {…}
```

need some object to be used as a lock

```
synchronized int balance() {…}
```

is equivalent to

```
int balance() {
  synchronized(this) {…}
}
```

same lock → mutual exclusion, happens-before

Volatile

keyword to describe a variable which will be modified by multiple threads

```
volatile int balance;
```

The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"

Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

A primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized)

Example 11

AtomicIntegerArray

java.util.concurrent.AtomicIntegerArray

similar classes for other types too

All methods in AtomicIntegerArray are atomic

these include functions you will use in the HW

- array.get(i)
- array.set(i)

VarHandles

Generic version of AtomicInteger

uses reflection to allow atomic access to variables of any type

interesting methods:

- get()
- compareAndSet()

More Examples

thread 1:

```
x = _;
done = true;
```

thread 2:

```
while (!done) {}
_ = x
```

what’s wrong here?

how can we fix it?