CS131 - Week 5

UCLA Spring 2019
TA: Kimmo Karkkainen

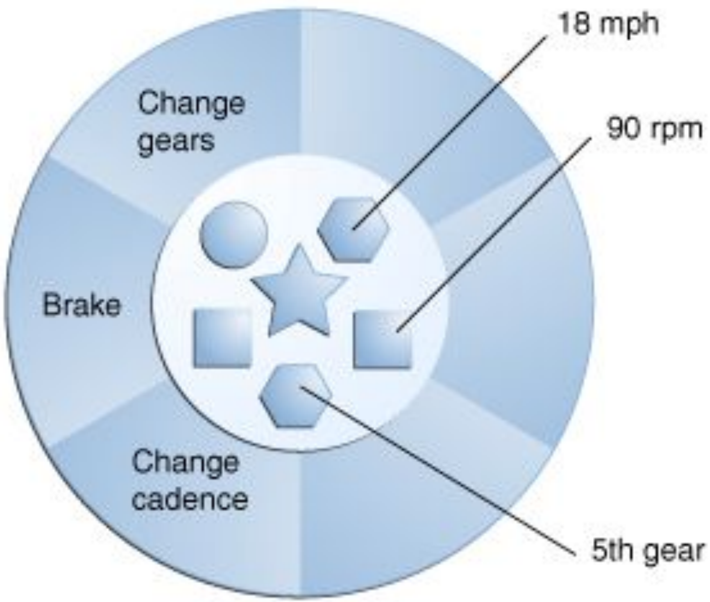
Today

- OOP & Java
- Multithreading and Java Memory Model
- Homework #3

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP)

- Main concept is objects
  - Objects have methods and fields
  - E.g. Bicycle object:
- Encapsulation of related methods/fields
- Example languages e.g. Java, C++, C#, Python, PHP, JavaScript, Ruby, Objective-C, Swift, Scala, Common Lisp, and Smalltalk
  - I.e. Most of the popular languages today



Classes

- Template for an object
    - Object is an *instance* of a class
    - E.g. We can have multiple Bicycle objects that function the same way, but can be moving at different speeds etc
- All objects created using the same class will have the same methods/fields

Objects - Benefits

Objects - Benefits

- Modularity
  - Splitting code into objects can help keep different parts of code separated
- Information-hiding
  - Objects should only interact by using each other's public methods
  - Internal implementation hidden -> easy to change later
- Code reuse
  - Objects easy to reuse in other programs
- Pluggability and debugging ease
  - We can replace an object with a different one as long as they have the same type
    - E.g. An object logging into a file vs an object logging into stdout

Alan Kay's definition of OOP

- Everything is an object
    - Numbers, classes, functions, ...
- Objects communicate by sending/receiving messages
    - Think of biological cells communicating
- Objects have their own memory
- Every object is an instance of some class
- All objects of the same type can receive the same messages

Some of these do not apply to most of the modern OOP languages!

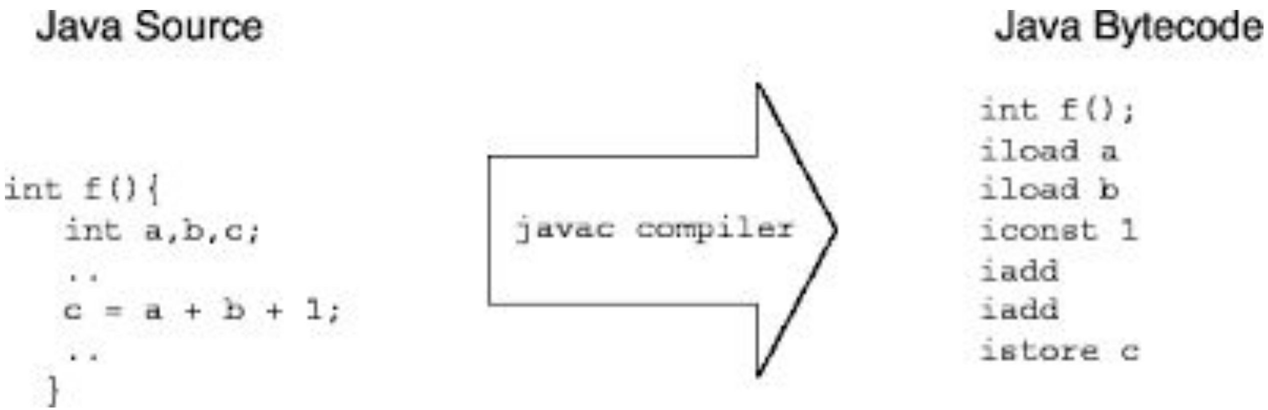
Java

Java Introduction

- General-purpose, object-oriented language
- One of the most popular programming languages
- Code compiled into bytecode and runs on a virtual machine
  - What are the pros/cons of this?
- Popular IDEs are Eclipse, IntelliJ IDEA
  - Eclipse the most popular option, free and open source
  - IDEA free for students, expensive for commercial use
  - **You can use any text editor for your homework**

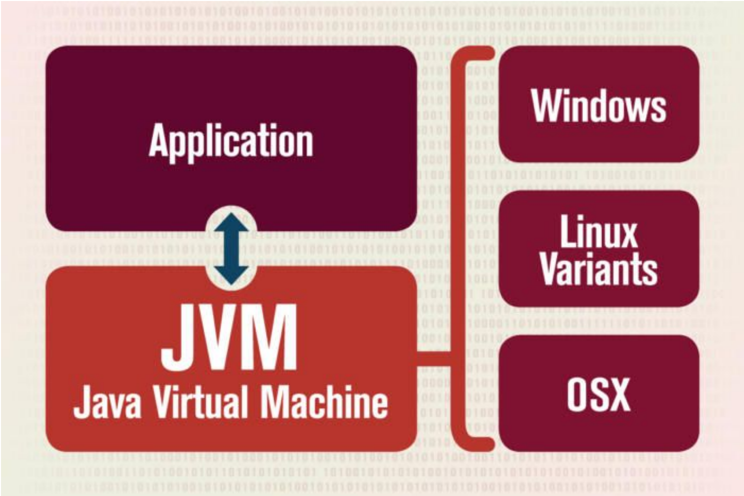
Java Bytecode

- A compromise between compiled and interpreted code:
  - Platform independence
    - Compiled code runs on one specific platform (OS & CPU architecture)
  - Performance
    - Interpreted code is difficult to optimize



Java Virtual Machine (JVM)

- Runs bytecode generated by a Java compiler
- Provides separation of code and operating system / hardware
    - Write once, run everywhere
- Multiple JVM implementations
    - Performance
        - Just-in-time compilation (JIT)
        - Garbage collection
    - Security
    - Support for different CPU architectures
    - Support for different operating systems
- Reference implementation (OpenJDK) provided by Oracle
    - Usually the best choice



Files

- MyClass.java = Code for MyClass
- MyClass.class = Bytecode for MyClass (Compiled from MyClass.java)
- Foo.jar = Java Archive file; ZIP archive
    - Could contain your compiled application with all the images and other resources
    - In your homework, you are provided a jar file containing the necessary code files
        - You could extract the contents with *unzip Foo.jar* or *jar xf Foo.jar*

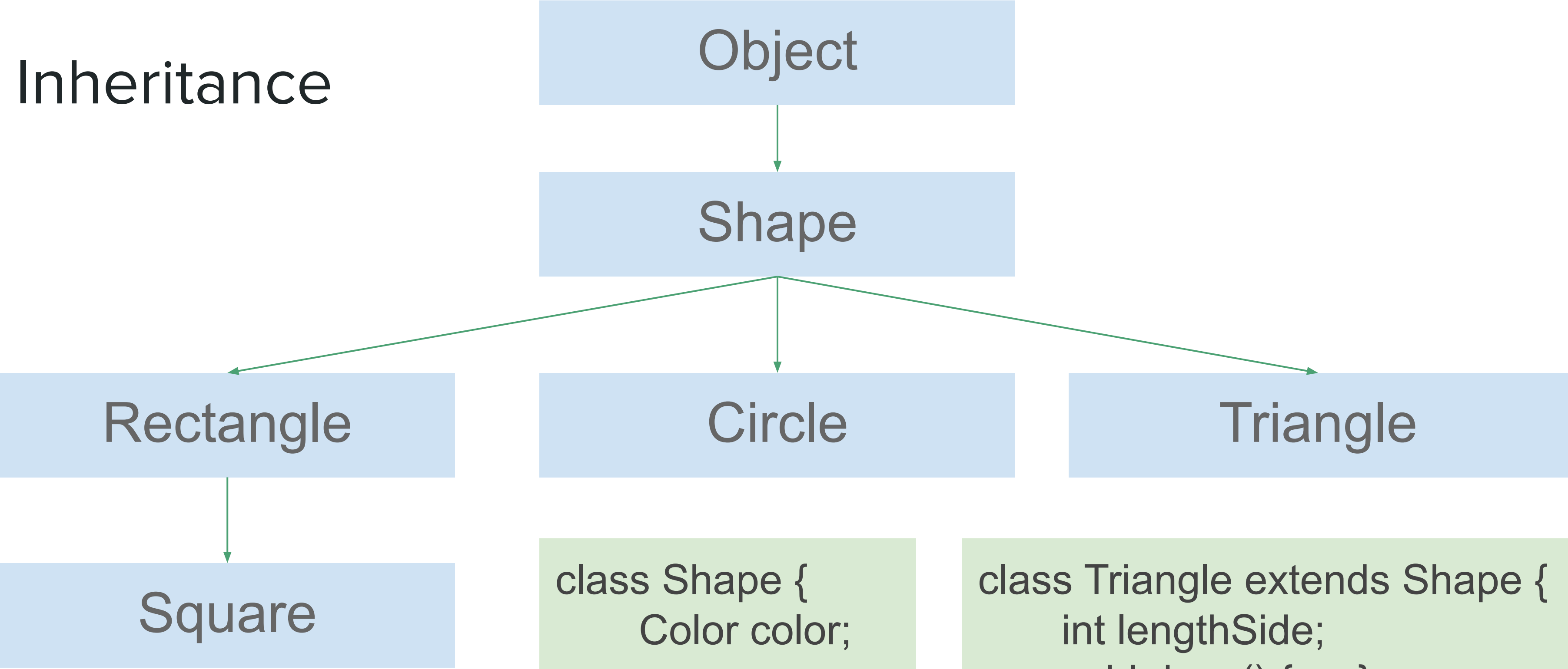
How to run code

- Insert the code for HelloWorld class inside a HelloWorld.java file
- Compile with **javac HelloWorld.java**
    - This generates a HelloWorld.class file, containing the bytecode
    - If your file references other classes, they will be compiled also
- Run your code with **java HelloWorld**
    - Note, the parameter is your class name, not the file name

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello, World");
        }
}
```

```
sh-3.2$ javac HelloWorld.java
sh-3.2$ java HelloWorld
Hello, World
```

Inheritance




```
class Shape {
        Color color;
        ...
}
```

```
class Triangle extends Shape {
        int lengthSide;
        void draw() { … }
        ...
}
```

Inheritance

```
class Shape {
	void draw() { /* do nothing */ }
}

class Rectangle extends Shape {
	void draw() { /* draw a rectangle */ }
}
class Circle extends Shape {
	void draw() { /* draw a circle */ }
}
class Triangle extends Shape {
	void draw() { /* draw a triangle */ }
}
```

```
Triangle a = new Triangle();
a.draw(); /* draws a triangle */

Shape b = a;
b.draw(); /* draws a triangle */

b = new Circle();
b.draw(); /* draws a circle */
```

Inheritance - Questions

Which of the following statements are allowed?

```
Square a = new Square();

Shape b = a;
```

```
Shape a = new Shape();

Square b = a;
```

```
Shape a = new Square();

Square b = a;
```

Inheritance - Questions

Which of the following statements are allowed?

```
Square a = new Square();

Shape b = a;
```

Allowed!

```
Shape a = new Shape();

Square b = a;
```

Forbidden:
Shape does not have the same methods/fields as Square

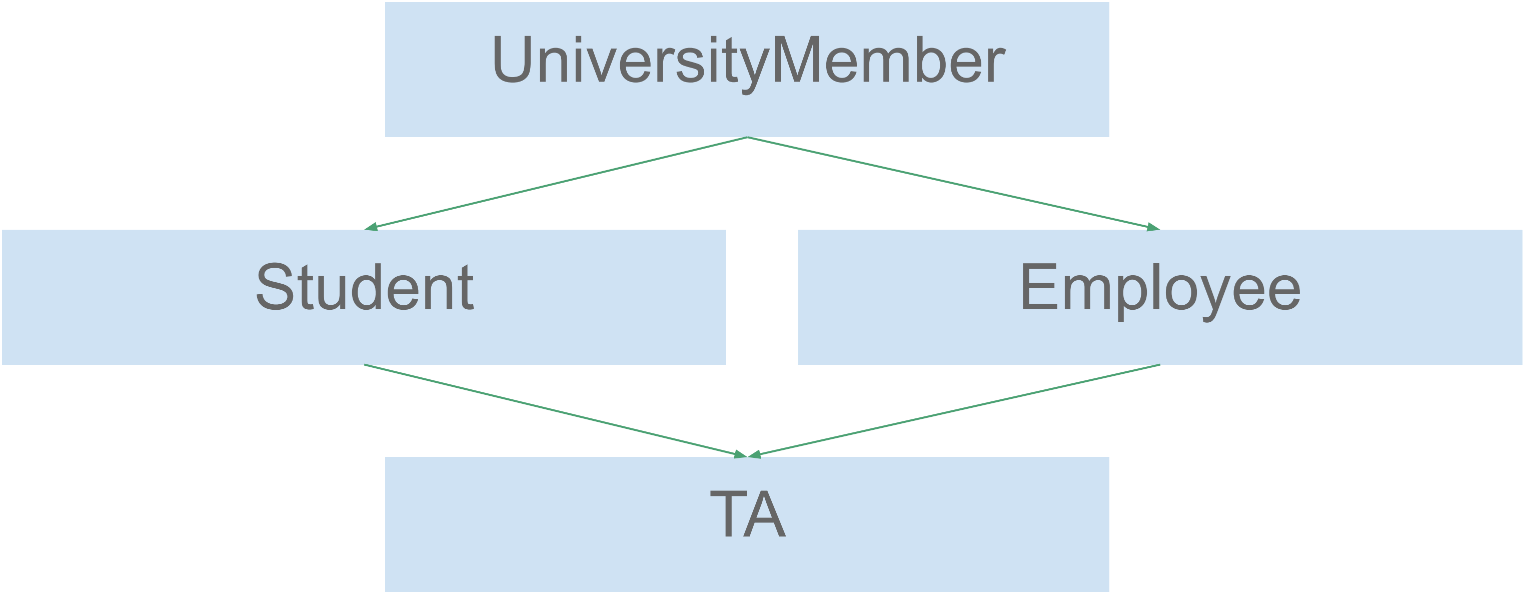
```
Shape a = new Square();

Square b = a;
```

Forbidden:
use *(Square)a* to cast the value before assignment

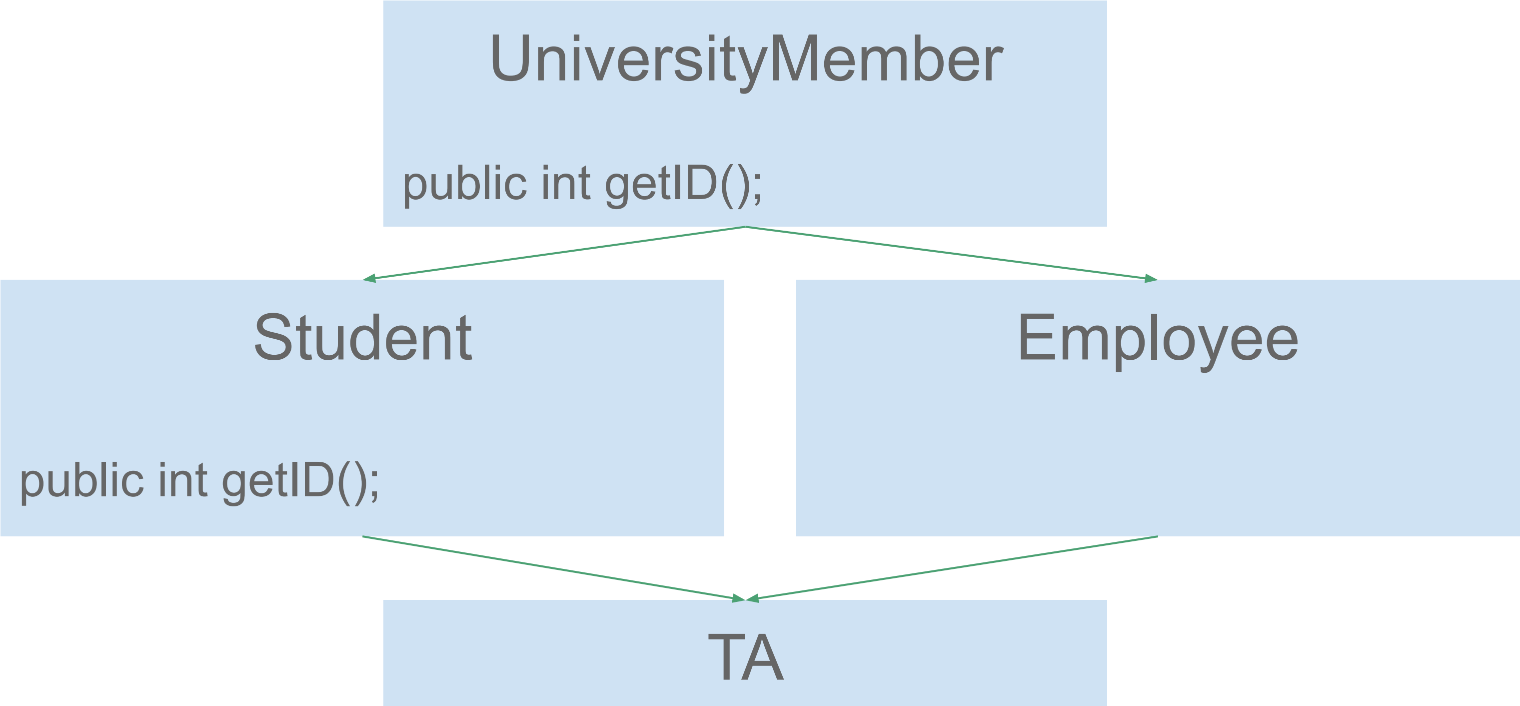
Multiple Inheritance

- Unlike C++, multiple inheritance is **forbidden** in Java, why?



Multiple Inheritance

- Unlike C++, multiple inheritance is **forbidden** in Java, why?



Interface

- Defines what a class must be able to do, not how to do it
- Interface can not be instantiated, must create a class that implements that interface
- One class can implement multiple interfaces

```
interface Vehicle {
	public int currentSpeed;

	public void increaseSpeed();
	public void decreaseSpeed();
	public void turnLeft();
	public void turnRight();
}
```

```
class Car implements Vehicle {
	public void increaseSpeed() {
		pressGasPedal();
	}

	public void decreaseSpeed() {
		pressBrakePedal();
	}

	… rest of the implementations ...
}
```

Abstract Classes

- What are abstract classes?

Abstract Classes

- Abstract classes are a combination of a class and an interface
  - Can’t create an object using an abstract class
  - Can define some parts of the class, while leaving other implementations for children
- Classes can extend only one abstract or normal class

```
abstract class Shape {
	abstract void draw();
	void setColor(Color c) { /* set color */ }
}
```

Generics

- What are generics?

Generics

- Define a class that can be instantiated to handle a specific type
    - Type must be specified when creating an object

```
class Box<T> {
        private T t;
        public T get() { return this.t; }
        public void set(T t1) { this.t=t1; }
}
```

```
var box = new Box<String>();

gs.set(“Hello”);
String value = gs.get();
```

Benefits:

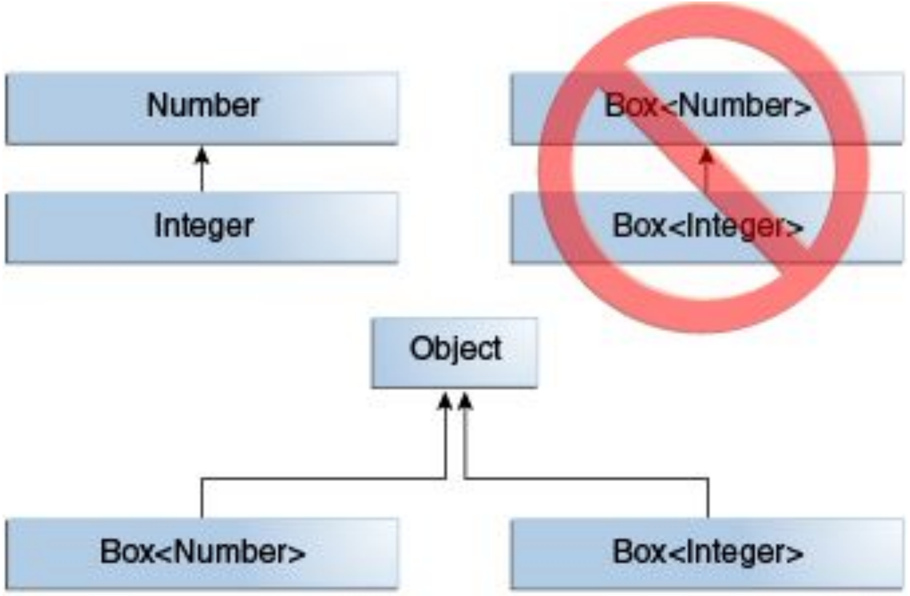
- Avoid casting (E.g. getting elements from *ArrayList<String>*)
- Compile-time type checks

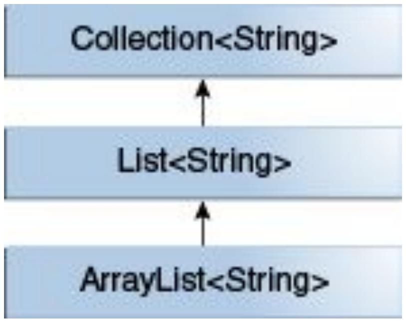
Generics

Question 5 in midterm:

In OCaml, “int list” is a subtype of “’a list”. However, in Java, “List<Integer> is not a subtype of “List<Object>”. Explain this seeming discrepancy, and give some other List type that “List<Integer>” *is* a subtype of in Java.

Generics - Type hierarchy





Access Modifiers

- Controlling who can access object's methods/fields

Access Levels

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

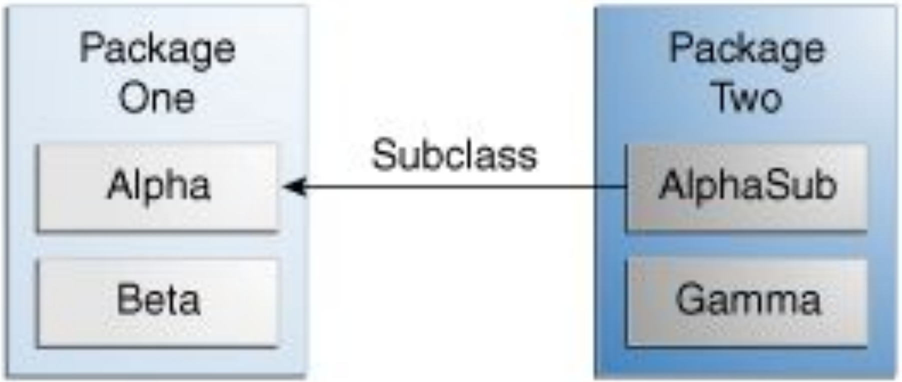
Access Modifiers

- Who can see methods in *Alpha* objects:

```
class Alpha {
        public void myMethod() { ... }
}
```

Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N



Access Modifiers

- In general, best to start with *private* and make fields/methods more visible only when it is necessary
    - Easier to change functionality afterwards if other classes do not depend on it
- Classes have only two access modifiers: public or no modifier (= package private)

Access Modifiers

Are the following implementations allowed?

Why or why not?

```
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;
        }
}
```

```
class Square implements Rectangle {
        private float side;
        private float area() {
                return side * side;
        }
        private float perimeter() {
                return 4 * side;
        }
}
```

Access Modifiers

Are the following implementations allowed?

Why or why not?

```
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;
        }
}
```

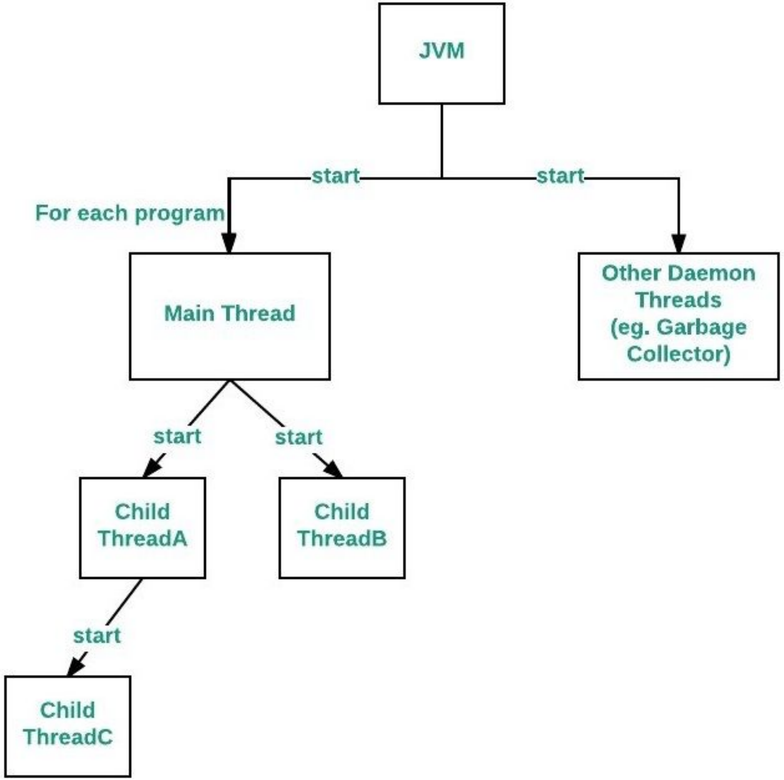
```
class Square implements Rectangle {
        private float side;
        private float area() {
                return side * side;
        }
        private float perimeter() {
                return 4 * side;
        }
}
```

Threading
&
Java Memory Model

Concurrency & Threads

- Concurrent programs can use either processes or threads
    - Processes have their own memory space, threads share the memory space within one process
- Needed to perform tasks faster or to have lower latency
    - In scientific computing, you might use multiple threads to perform a complex calculation fast
    - Web servers might get a lot of simple requests -> handle multiple tasks simultaneously

Threads



JVM
start
start
For each program
Main Thread
Other Daemon Threads (eg. Garbage Collector)
start
start
Child ThreadA
Child ThreadB
start
Child ThreadC

Threads

- Two ways to create a thread in Java:
  - Extend *Thread* class
  - Implement *Runnable* interface

Extending *Thread* class

```
public class MyThread extends Thread {
        public void run() {
                System.out.println("MyThread - START ");
                // Do some heavy processing here
                System.out.println("MyThread - END ");
        }
}

// In your main method:
Thread t1 = new MyThread();
Thread t2 = new MyThread();
t1.start(); // Start executing thread 1
t2.start(); // Start executing thread 2
t1.join(); // Wait for thread 1 to finish
t2.join(); // Wait for thread 2 to finish
```

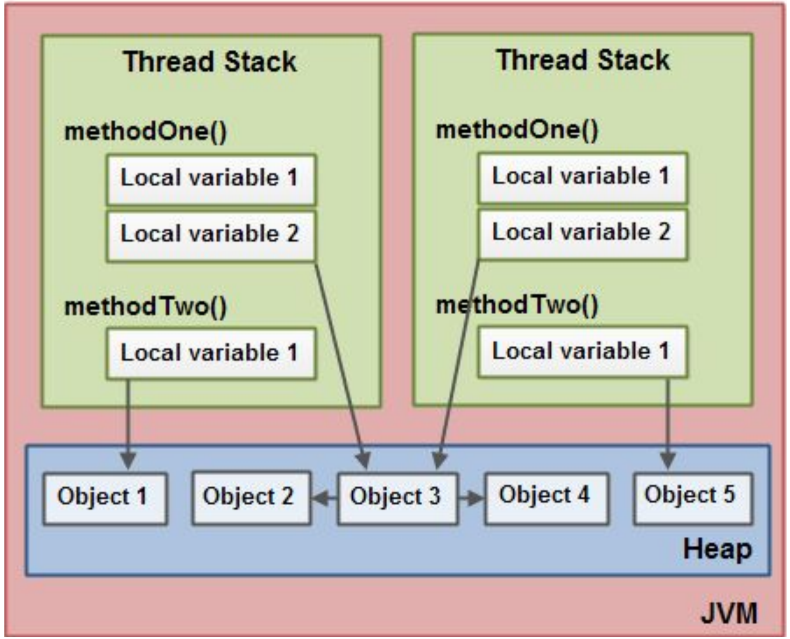
Implementing *Runnable* interface

```
public class MyRunnable implements Runnable {
	public void run() {
		System.out.println("MyRunnable - START ");
		// Do some heavy processing here
		System.out.println("MyRunnable - END ");
	}
}

// In your main method:
Thread t1 = new Thread(new MyRunnable());
Thread t2 = new Thread(new MyRunnable());
t1.start(); // Start executing thread 1
t2.start(); // Start executing thread 2
t1.join(); // Wait for thread 1 to finish
t2.join(); // Wait for thread 2 to finish
```

Java Memory Model

- Defines how threads interact through memory
  - I.e. How multithreaded programs can behave in different situations



Java Memory Model

- "As-if-serial" semantics used within one thread
  - Compiler can change your code in any way as long as the result of execution is the same
  - E.g. **y = 1; x = 2;** vs **x = 2; y = 1;**
- Reasoning across multiple threads more challenging -> needs input from the programmer
  - Java provides multiple ways to set constraints on the order of execution

Problems with Concurrency

Problems with Concurrency - Data Race

- What can happen when we run these threads simultaneously?

Thread 1:	**Thread 2:**
`counter+=1;`	`counter+=1;`

Problems with Concurrency - Data Race

- What can happen when we run these threads simultaneously?

Thread 1:

```
counter+=1;
```

Thread 2:

```
counter+=1;
```

→

Thread 1:

```
tmp1 = counter + 1;
counter = tmp1;
```

Thread 2:

```
tmp2 = counter + 1;
counter = tmp2;
```

Problems with Threading - What is the value of cnt?

tmp1 = cnt + 1;	
cnt = tmp1;	
	tmp2 = cnt + 1;
	cnt = tmp1;

cnt <- cnt + 2

	tmp2 = cnt + 1;
	cnt = tmp2;
tmp1 = cnt + 1;	
cnt = tmp1;	

cnt <- cnt + 2

tmp1 = cnt + 1;	
	tmp2 = cnt + 1;
cnt = tmp1;	
	cnt = tmp2;

cnt <- cnt + 1

tmp1 = cnt + 1;	
	tmp2 = cnt + 1;
	cnt = tmp2;
cnt = tmp1;	

cnt <- cnt + 1

	tmp2 = cnt + 1;
tmp1 = cnt + 1;	
	cnt = tmp2;
cnt = tmp1;	

cnt <- cnt + 1

	tmp2 = cnt + 1;
tmp1 = cnt + 1;	
cnt = tmp1;	
	cnt = tmp2;

cnt <- cnt + 1

Dependencies across threads

- What will the following threads print?
    - Assume **x=y=false** initially

Thread A:

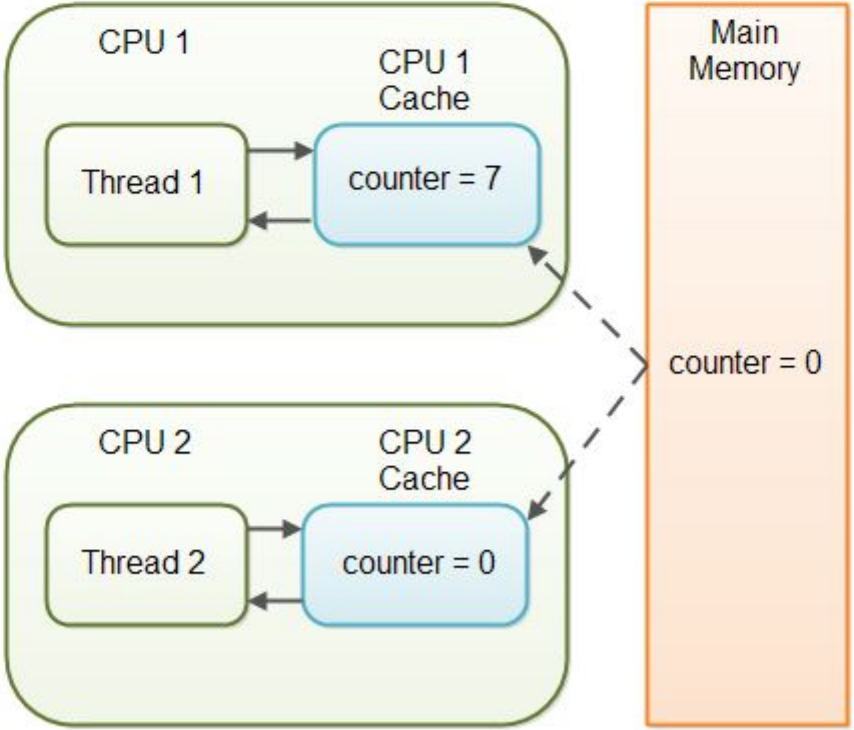
```
if (x) {
    System.out.println("Thread A");
}
y = true;
```

Thread B:

```
if (y) {
    System.out.println("Thread B");
}
x = true;
```

CPU Caches

- Multiple levels of caches
  - Each CPU/core can have their own cached values
- Even if everything happens in the expected order, results can be incorrect



Loops

- What can go wrong? Assume *done=false* initially

Thread 1:

```
x = 5;
done = true;
```

Thread 2:

```
while (!done) { }
System.out.println(x)
```

More concurrency problems

- Read You Don’t Know Jack about Shared Variables or Memory Models
  - Link is in the homework too

Thread Synchronization

Synchronized keyword

- Each object has one lock
- Exclusive access
    - Only one thread can enter any synchronized method in one object at once
- *Happens-before* relationship
    - Everything that one thread did while in a synchronized block will be visible to the next thread entering a synchronized block
- A thread can call any other synchronized methods while it holds the lock

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Synchronized keyword

- *Synchronized* can also be used for smaller blocks of code
- Avoid blocking other threads when it is not necessary

```
public class SynchronizedCounter {
    private int c = 0;

    public void incrementAndWork() {
        … computation here ….
        synchronized(this) {
            c++;
        }
        … computation here ….
    }
}
```

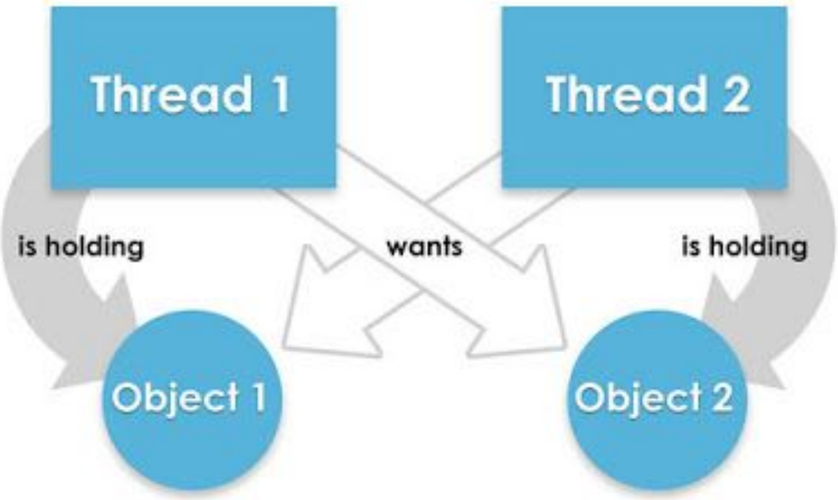
Synchronized keyword

- *Synchronized* block can use any object as the lock

```
public class MyClass {
        private int c1 = 0;
        private int c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();
        public void inc1() {
                synchronized(lock1) {
                        c1++;
                }
        }
        public void inc2() {
                synchronized(lock2) {
                        c2++;
                }
        }
}
```

Deadlock

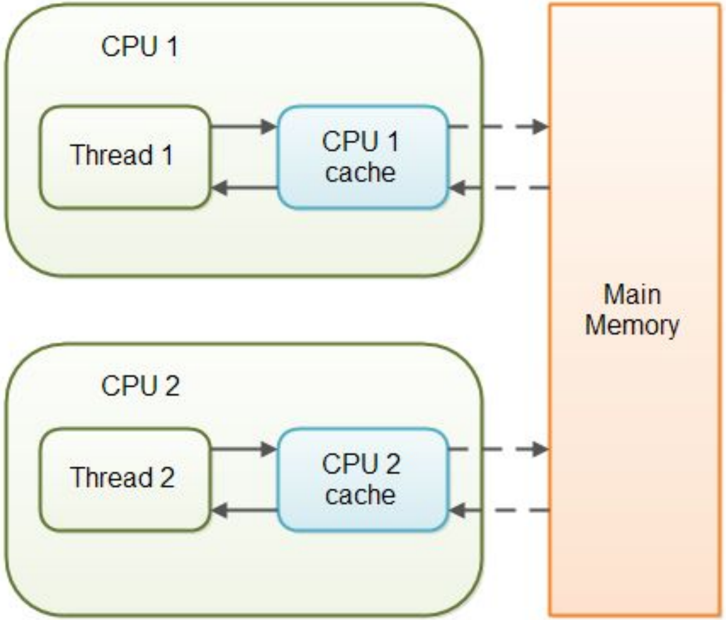
- Using *synchronized* might lead to deadlocks:



Volatile

- Defining variables *volatile* guarantees that other threads will see the changes immediately
    - Uses CPU memory barriers
- Without *volatile*, there is no guarantee that threads are not using their locally cached versions of variables

```
public class SharedObject {
        public volatile int counter = 0;
}
```



Volatile

- Additional guarantees:
    - Volatile access can not be reordered relative to other reads/writes
    - If two threads access the same volatile variable, the second thread is guaranteed to see the same state as the first thread

```
public class MyClass {
        private int years;
        private int months
        private volatile int days;

        public int totalDays() {
                int total = this.days;
                total += months * 30;
                total += years * 365;
                return total;
        }

        public void update(int years, int months, int days){
                this.years = years;
                this.months = months;
                this.days = days;
        }
}
```

Volatile

- Note: Volatile does not prevent our earlier problem where two threads tried to perform *counter++* simultaneously!
  - No locks used -> Reads and writes can still happen simultaneously
- Can fix the other problem though, if we define *done* as volatile:

Thread 1:

```
x = 5;
done = true;
```

Thread 2:

```
while (!done) { }
System.out.println(x)
```

java.util.concurrent.atomic

- *Atomic* package provides data types with atomic operations
- *Atomic* = Other threads do not see intermediate states, only the final state
- For example, AtomicInteger can be used to perform *cnt++* as an atomic operation:

```
AtomicInteger cnt = new AtomicInteger(5);
cnt.incrementAndGet();
```

java.util.concurrent.atomic

- *AtomicIntegerArray* provides an array with atomic/volatile operations
- Calling *get/set* on individual elements is **volatile**
- Calling *incrementAndGet* and similar methods is **atomic**

Locks (java.util.concurrent.locks)

- ReentrantLock
  - One thread at a time holds a lock and can acquire it multiple times (similar to *synchronized*)
- ReentrantReadWriteLock
  - Multiple reads can happen simultaneously, as long as nobody is writing to a variable
  - Only one write at a time

```
ReentrantLock lock = new ReentrantLock();

// In a thread:
lock.lock();
<do something>
lock.unlock(); // Must unlock after use!
```

```
ReentrantReadWriteLock rwl = new
ReentrantReadWriteLock();

// In a thread:
rwl.writeLock().lock();
<write>
rwl.writeLock.unlock()
rwl.readLock().lock();
<read>
rwl.readLock.unlock()
```

Semaphore (java.util.concurrent.Semaphore)

- Allows a limited number of threads to access a resource at the same time
  - E.g. Allow only a few threads to perform computation at once to make sure they don't slow each others down too much

```
Semaphore s = new Semaphore(5); // 5 resources available

// In a thread:
s.acquire();

<do something>

s.release();
```

Fair vs Non-Fair Locks

- Locks and semaphores provide a *fair* and *non-fair* mode
  - Defined when creating the lock
- Fair mode guarantees that the locks are given to the longest-waiting thread
- By default, everything is **non-fair**

VarHandle

- Allows different synchronization levels for regular variables
  - Create a *handle* for a variable and perform reads/writes using that handle
- Different methods:
  - *set(newValue)* - set value like using a typical variable
  - *setVolatile(newValue)* - set value like using a volatile variable
  - *getAndAdd(value)* - Atomically add to the current value and get the old value
  - *releaseFence() / acquireFence()* - Prevent reordering loads/stores
  - + Many more
- Recommended reading: Using JDK 9 Memory Order Modes

Homework #3 (Due next Monday)

Thread safety vs. Performance

- In HW #3, you'll compare different synchronization techniques
- What's the best compromise between reliability and performance?
    - Some techniques are 100% safe but slow, while others are faster but not safe

Background

- We have an array containing integer values between *0..maxval*:

Pos	**0**	**1**	**2**	**...**	**n-1**	**n**
Value	5	98	75	...	84	113

- Only one operation allowed: *swap(i,j)*
  - This operation decreases *i*th value by 1 and increases *j*th value by 1
  - E.g. *swap(0,1)* would update the first two values to 4 and 99 respectively

Background

- We want to call *swap(i,j)* millions of times efficiently
- How can we make it fast and reliable?

Checking for synchronization problems

- The only efficient way to check the correctness is to check:
  - Sum of all the values should be the same as in the beginning
  - Value at each location is between 0..maxval
- These checks can only show that there was a synchronization problem, not that everything worked correctly! Why?

Data Structure - State.java

- Your solutions will implement interface *State*:

```
interface State {
	int size();
	byte[] current();
	boolean swap(int i, int j);
}
```

NullState.java - Dummy implementation

```
class NullState implements State {
    private byte[] value;
    NullState(byte[] v, byte maxval) { value = v; }
    public int size() { return value.length; }
    public byte[] current() { return value; }

    public boolean swap(int i, int j) { return true; }
}
```

Note that this solution passes our sanity checks!

SynchronizedState.java - Safe but inefficient

```
class SynchronizedState implements State {
	private byte[] value;
	private byte maxval;
	SynchronizedState(byte[] v) { value = v; maxval = 127; }
	SynchronizedState(byte[] v, byte m) { value = v; maxval = m; }
	public int size() { return value.length; }
	public byte[] current() { return value; }
	public synchronized boolean swap(int i, int j) {
		if (value[i] <= 0 || value[j] >= maxval) {
			return false;
		}
		value[i]--;
		value[j]++;
		return true;
	}
}
```

SwapTest.java

- Contains test code for one thread:
    - Runs *state.swap(a,b)* with random values a and b as many times as specified

```
class SwapTest implements Runnable {
        private int nTransitions;
        private State state;
        SwapTest(int n, State s) { … }

         public void run() {  ...  }
}
```

- ***Runnable*** interface defines that a ***Thread*** object can run this code
    - Must have ***run()*** method

UnsafeMemory.java

Contains the main method of the code:

1. Parse command line parameters
   - (model name, # threads, # swaps, initial values)
2. Initialize the state object
3. Create and start threads (by running *SwapTest* objects in multiple threads)
4. Wait for threads to finish (keeping track of time)
5. Verify that the state is consistent (sum hasn't changed, values within bounds)

Task #1

- Implement an *UnsynchronizedState* class
  - Similar to *SynchronizedState.java*, except without the *synchronized* keyword
- In theory, this should be faster than synchronized
  - In practice, this approach can be problematic. If you run into problems, consider what can prevent the execution from finishing. (Hint: The program is trying to do *N* **successful** swaps)

Task #2

- Implement *GetNSet*, which is a compromise between *synchronized* and *unsynchronized* state classes
- You should use *AtomicIntegerArray* class, which provides **volatile** access to array elements
    - Use only *get()/set()* methods

Task #3

- Design and implement class *BetterSafe*, which is faster than the synchronized class while providing perfect thread safety
    - Performance difference might be very insignificant on latest Java versions
- You should consider these packages:
    - java.util.concurrent (e.g. Semaphores)
    - java.util.concurrent.atomic (e.g. AtomicInteger, AtomicIntegerArray)
    - java.util.concurrent.locks (e.g. ReentrantLock, ReadWriteReentrantLock)
    - java.lang.invoke.VarHandle

Task #4

- Integrate all the state classes into one program *UnsafeMemory*

```
if (args[0].equals("Null"))
        s = new NullState(stateArg, maxval);
else if (args[0].equals("Synchronized"))
        s = new SynchronizedState(stateArg, maxval);
/* Add your object initializations here */
else
        throw new Exception(args[0]);
```

Task #5 & #6

- Measure and characterize the performance and reliability of each class
- Compare these measurements
- Use OpenJDK 9 **and** 11
    - Both are available on SEASnet, see instructions in homework
    - New version might have optimizations that improve the performance
- Make sure you test on SEASnet! Results depend on hardware

Report

- Write a **2-3 page** report discussing:
    - Pros/cons of the four packages that you were given for BetterSafe implementation
    - Why your solution is faster than *Synchronized* and why it is still 100% reliable
    - Discuss any problems you had to overcome to do your measurements properly
    - Explain why your class is free of data races, or if it isn't (due to a bug), show how to reproduce the problem (i.e. how to run the program in a way that it is likely to fail)

Submission

- Your submission should have a **.jar** file containing all the **.java** files
    - **Not .*class*! Submissions without .java files will not be graded**
    - Creating a jar file: *jar cvf jmmplus.jar *.java*
- Include the report as a separate file *report.pdf*
    - Do **not** include your name or student id

Questions?