Principles of Software Construction:
Objects, Design, and Concurrency

Part 5: Concurrency

Introduction to concurrency

Charlie Garrod Michael Hilton

School of
Computer Science

institute for
SOFTWARE
RESEARCH

Administrivia

- Homework 5 team sign-up deadline tonight
- Midterm exam in class Thursday (02 November)
  - Review session Wednesday, 01 Nov. 7-9 p.m. in HH B103
- Do you want to be a software engineer?

The foundations of the Software Engineering minor

- Core computer science fundamentals
- Building good software
- Organizing a software project
  - Development teams, customers, and users
  - Process, requirements, estimation, management, and methods
- The larger context of software
  - Business, society, policy
- Engineering experience
- Communication skills
  - Written and oral

SE minor requirements

- Prerequisite: 15-214 or 17-214
- Two core courses
  - 17-313 Foundations of SE (fall semesters)
  - 17-413 SE Practicum (spring semesters)
- Three electives
  - Technical
  - Engineering
  - Business or policy
- Software engineering internship + reflection
  - 8+ weeks in an industrial setting, then
  - 17-415

To apply to be a Software Engineering minor

- Email clegoues@cs.cmu.edu
  - Your name, Andrew ID, expected grad date, QPA, and minor/majors
  - Why you want to be a SE minor
  - Proposed schedule of coursework

- Fall applications due by Friday, 10 November 2017
  - Only 15 SE minors accepted per graduating class
- More information at:
  - http://isri.cmu.edu/education/undergrad/

Key concepts from last Thursday

Key design principle: Information hiding

- "When in doubt, leave it out."

Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value
- Mutable objects require careful management of visibility and side effects
  - e.g. `Component.getSize()` returns a mutable `Dimension`
- Document mutability
  - Carefully describe state space

Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
  public Object put(Object key, Object value);

  // Throws ClassCastException if this instance
  // contains any keys or values that are not Strings
  public void save(OutputStream out, String comments);
}
```

Avoid behavior that demands special processing

- Do not return `null` to indicate an empty value
  - e.g., Use an empty `Collection` or array instead
- Do not return `null` to indicate an error
  - Use an exception instead

Throw exceptions only for exceptional behavior

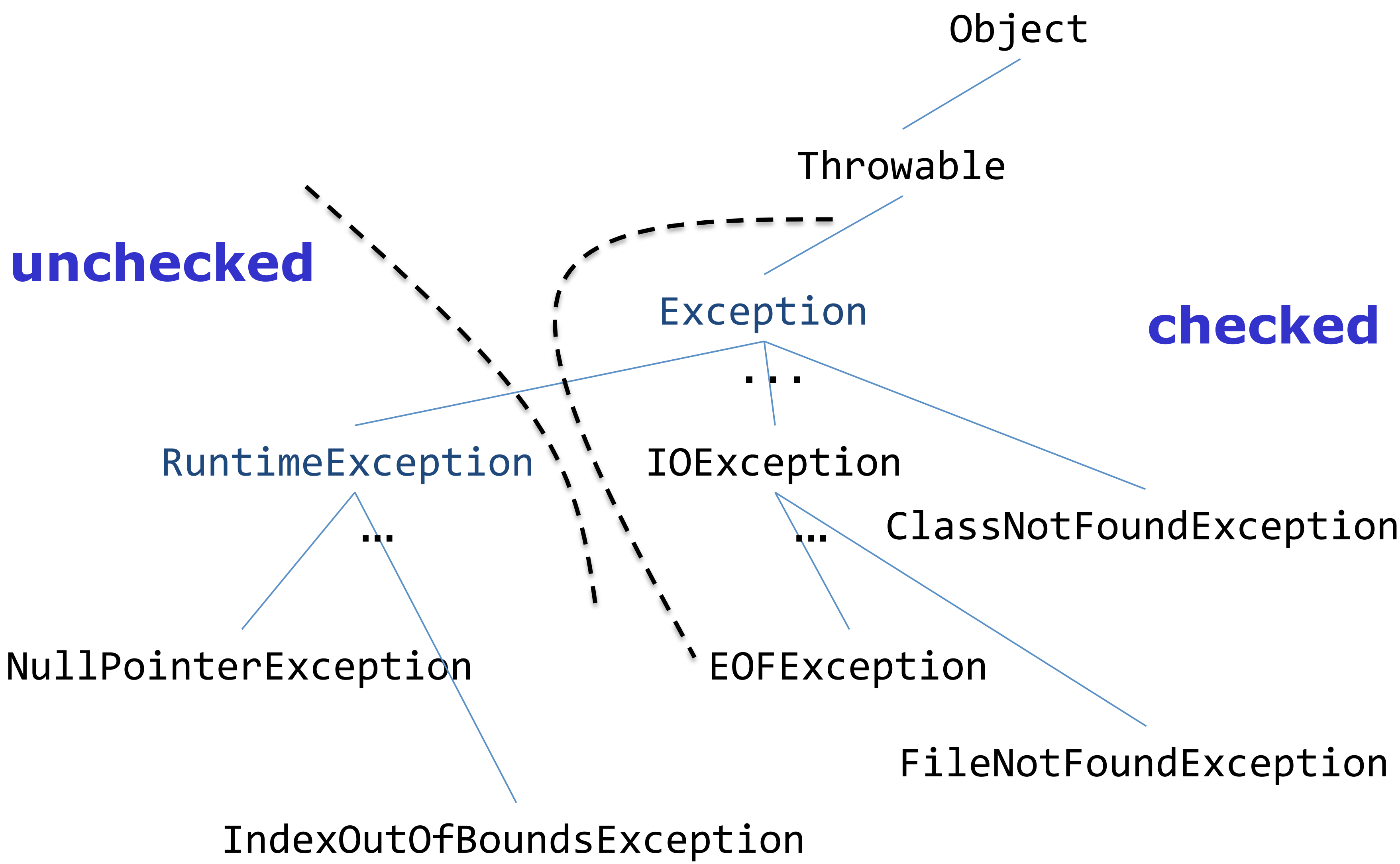
- Do not force client to use exceptions for control flow:

```
private byte[] a = new byte[CHUNK_SIZE];

void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(a, CHUNK_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(a, remaining);
  }
}
```

- Conversely, don't fail silently:

```
ThreadGroup.enumerate(Thread[] list)
```

Context: The exception hierarchy in Java



Avoid checked exceptions, if possible

- Overuse of checked exceptions causes boilerplate code:

```
try {
    Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // This exception can't happen if Foo is Cloneable
    throw new AssertionError(e);
}
```

Don't make the client do anything the module could do

- Carelessly written APIs force clients to write boilerplate code:

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can't happen!
  }
}
```

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  comp
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.ja
        at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
        at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
        at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
        at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
public class Throwable {
  public void printStackTrace(PrintStream s);
  public StackTraceElement[] getStackTrace(); // since 1.4
}

public final class StackTraceElement {
  public String  getFileName();
  public int     getLineNumber();
  public String  getClassName();
  public String  getMethodName();
  public boolean isNativeMethod();
}
```

API design summary

- Accept the fact that you, and others, will make mistakes
  - Use your API as you design it
  - Get feedback from others
  - Hide information to give yourself maximum flexibility later
  - Design for inattentive, hurried users
  - Document religiously

Semester overview

- Introduction to Java and O-O
- Introduction to **design**
  - **Design** goals, principles, patterns
- **Design**ing classes
  - **Design** for change
  - **Design** for reuse
- **Design**ing (sub)systems
  - **Design** for robustness
  - **Design** for change (cont.)
- **Design** case studies
- **Design** for large-scale reuse
- Explicit concurrency

- Crosscutting topics:
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification

Today: Concurrency, motivation and primitives

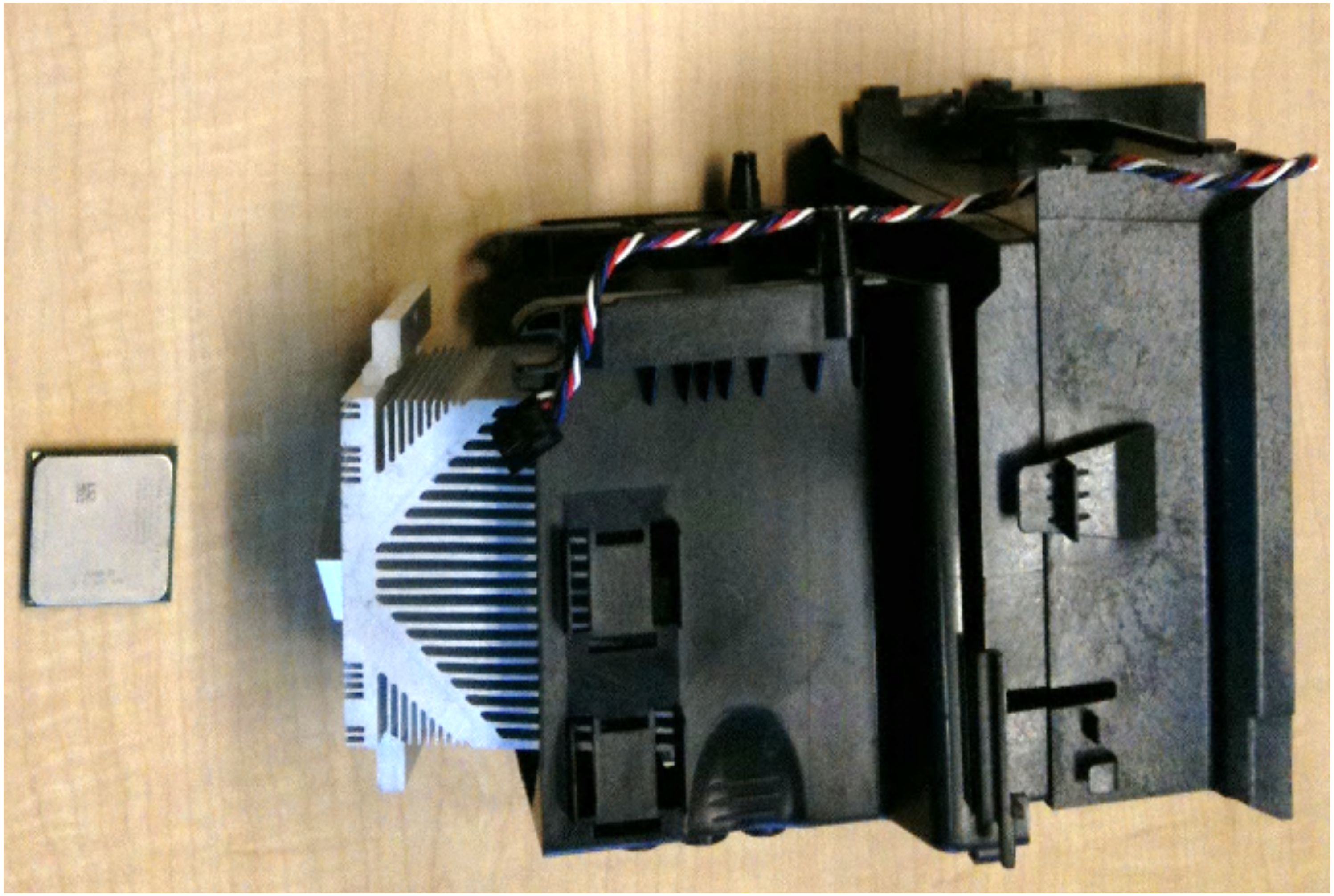
- The backstory
  - Motivation, goals, problems, …
- Basic concurrency in Java
- Coming soon (not today):
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

Power requirements of a CPU

- Approx.: **C**apacitance * **V**oltage^2 * **F**requency
- To increase performance:
  - More transistors, thinner wires
    - More power leakage: increase **V**
  - Increase clock frequency **F**
    - Change electrical state faster: increase **V**
- *Dennard scaling*: As transistors get smaller, power density is approximately constant…
  - …until early 2000s
- Heat output is proportional to power input

One option: fix the symptom

- Dissipate the heat

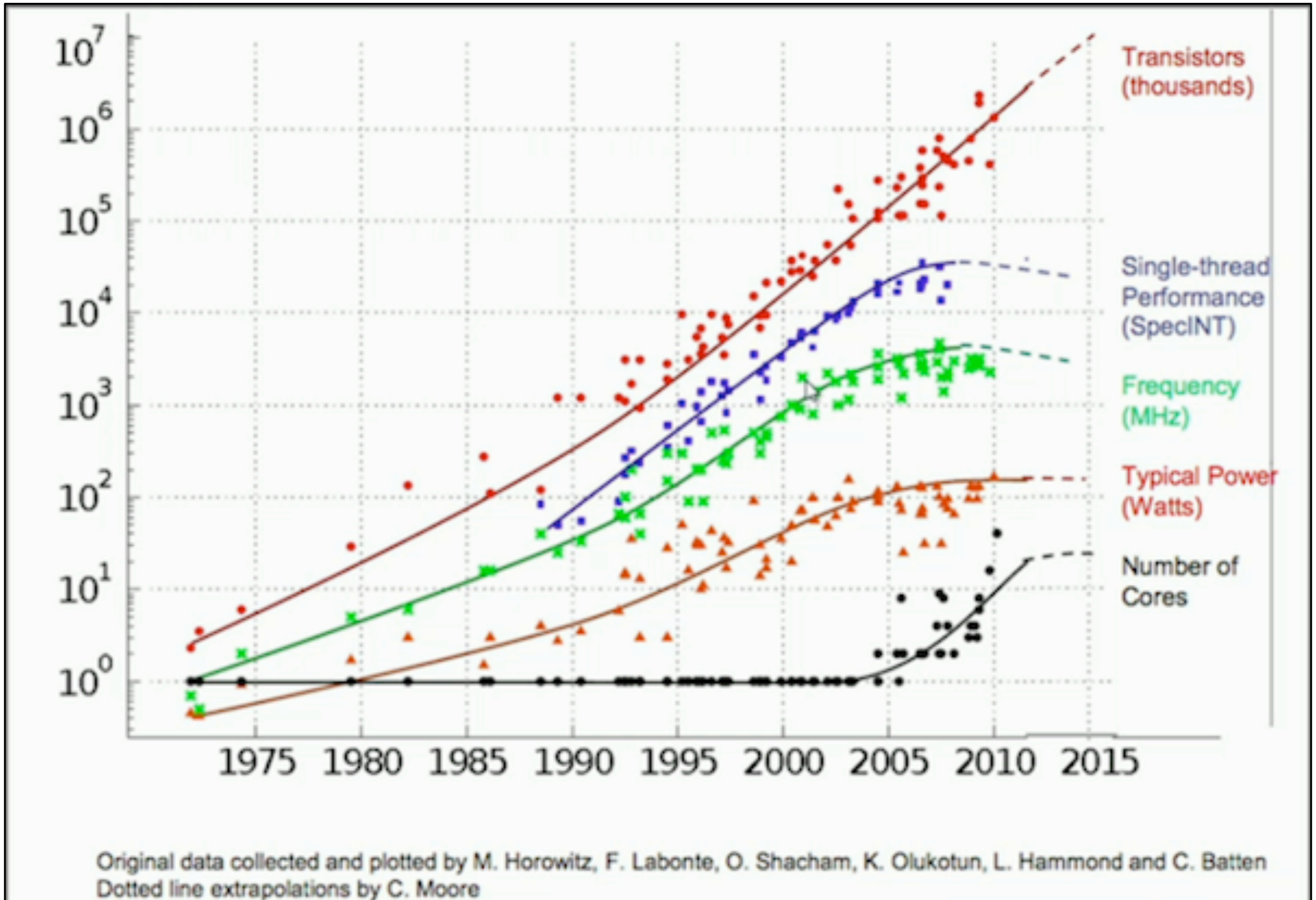


One option: fix the symptom

- Better: Dissipate the heat with liquid nitrogen
  – Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html

Processor characteristics over time



Concurrency then and now

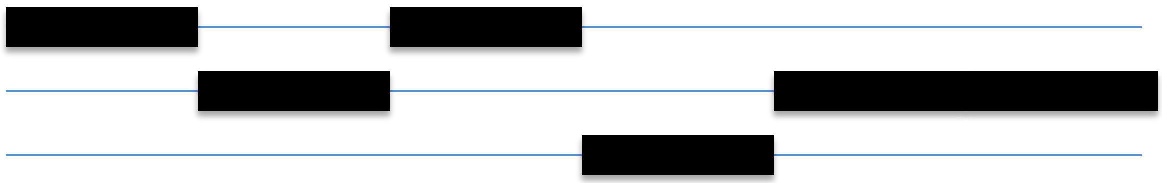
- In past multi-threading just a convenient abstraction
  - GUI design: event dispatch thread
  - Server design: isolate each client's work
  - Workflow design: isolate producers and consumers
- Now: required for scalability and performance

We are all concurrent programmers

- Java is inherently multithreaded
- To utilize modern processors, we must write multithreaded code
- Good news: a lot of it is written for you
  - Excellent libraries exist (`java.util.concurrent`)
- Bad news: you still must understand fundamentals
  - …to use libraries effectively
  - …to debug programs that make use of them

Aside: Concurrency vs. parallelism, visualized

- Concurrency without parallelism:



- Concurrency with parallelism:

Basic concurrency in Java

- An interface representing a task

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

- A class to execute a task in a thread

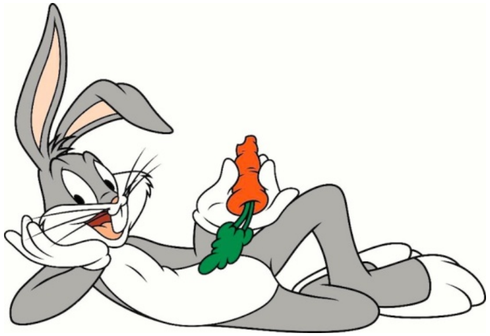
```
public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    …
}
```

Example: Money-grab (1)

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }
    public long balance() {
        return balance;
    }
}
```

Example: Money-grab (2)

```
public static void main(String[] args) throws InterruptedException
   {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() + daffy.balance());
}
```



What went wrong?

- Daffy & Bugs threads had a *race condition* for shared data
  - Transfers did not happen in sequence
- Reads and writes interleaved randomly
  - Random results ensued

Shared mutable state requires concurrency control

- Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: *limit concurrency to achieve safety*

The challenge of concurrency control

- Not enough concurrency control: *safety failure*
  - Incorrect computation
- Too much concurrency control: *liveness failure*
  - Possibly no computation at all (*deadlock* or *livelock*)

An easy fix:

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static synchronized void transferFrom(BankAccount source,
                                          BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }
    public synchronized long balance() {
        return balance;
    }
}
```

Concurrency control with Java's *intrinsic* locks

- `synchronized (lock) { … }`
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock

Concurrency control with Java's *intrinsic* locks

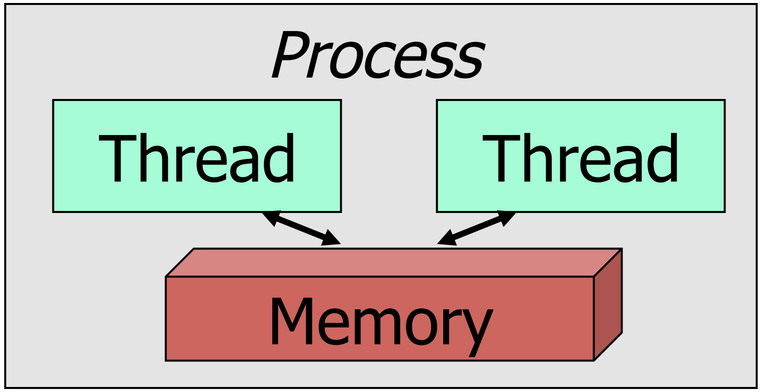
- `synchronized (lock)` { … }
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
- `synchronized` on an instance method
  - Equivalent to `synchronized (this)` { … } for entire method
- `synchronized` on a static method in class Foo
  - Equivalent to `synchronized (Foo.class)` { … } for entire method

Another example: serial number generation

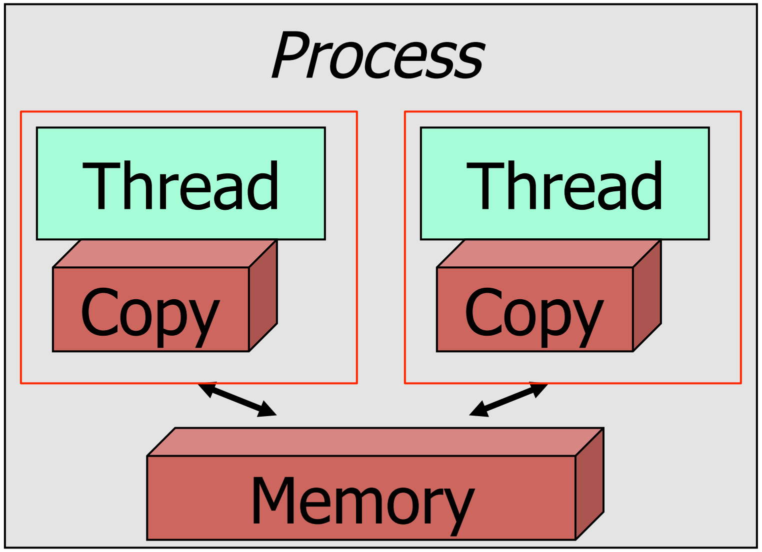
```
public class SerialNumber {
    private static long nextSerialNumber = 0;
    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```

Aside: Hardware abstractions

- Supposedly:
  - Thread state shared in memory



- A (slightly) more accurate view:
  - Separate state stored in registers and caches, even if shared



Atomicity

- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- In Java, integer increment is not atomic

`i++;` is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

Again, the fix is easy

```
public class SerialNumber {
    private static int nextSerialNumber = 0;
    public static synchronized int generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException{
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

i: **00000...00000111**

⋮

i: **00000...00101010**

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

`i:`	00000…00000111

⋮

`i:`	00000…00101010

- In Java:
  - Reading an `int` variable is atomic
  - Writing an `int` variable is atomic
  - Thankfully, `ans:` 00000…00101111 is not possible

institute for SOFTWARE RESEARCH

Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

high bits	low bits

  - Concurrently:
    - Thread A writing high bits and low bits
    - Thread B reading high bits and low bits

Precondition:

```
long i = 10000000000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

ans: **01001...00000000**	(10000000000)
ans: **00000...00101010**	(42)
ans: **01001...00101010**	(10000000042 or ...)

institute for SOFTWARE RESEARCH

Yet another example: cooperative thread termination

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(42);
        stopRequested = true;
    }
}
```

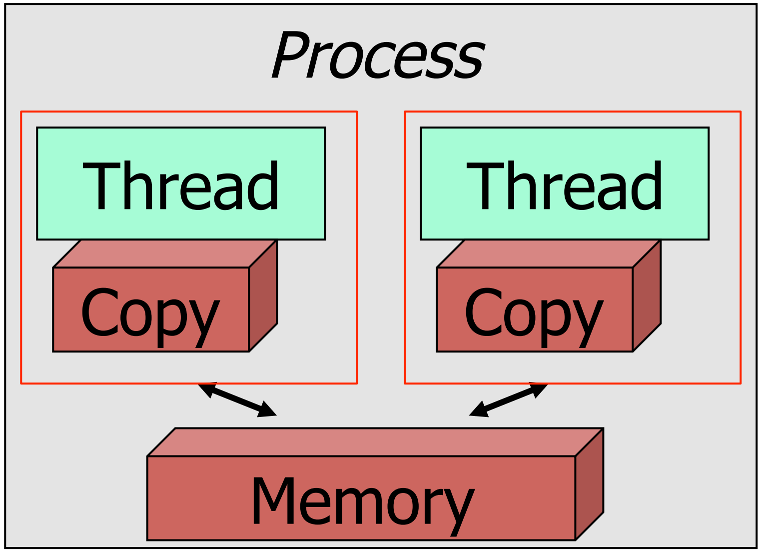
What went wrong?

- In the absence of synchronization, there is no guarantee as to when, if ever, one thread will see changes made by another
- JVMs can and do perform this optimization:

```
while (!done)
    /* do something */ ;
```

becomes:

```
if (!done)
    while (true)
        /* do something */ ;
```



How do you fix it?

```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(42);
        requestStop();
    }
}
```

Summary

- Like it or not, you're a concurrent programmer
- Ideally, avoid shared mutable state
  - If you can't avoid it, synchronize properly
- Even atomic operations require synchronization
  - e.g., `stopRequested = true`
- Some things that look atomic aren't (e.g., `val++`)