

Advanced Java 8 Programming

Virtual Logistics

- Put up a green check ✓ or a red X:
 - Can you **hear** the instructor?
 - Can you **view** the presentation?
- Virtual Manners
 - Please mute your audio until you wish to speak.
 - Unmute at any time and ask questions.
 - Also, chat or "hand up" to ask questions.
 - This class is interactive - please participate!



Housekeeping

- Roster (attendance)
- Class Times
 - Time zone
 - Breaks, lunch
- Silence cell phones and electronics
- Questions welcome at any time



Course Materials

Put up a green check ✓ or a red X:

- **Have you downloaded the course materials?**
 1. Course Manual
 2. Lab Manual
 3. Setup Guide
 4. Setup Files
- Electronic manuals' tables of contents contain hyperlinks for navigation



Course Prerequisites

- Java Programming Skills
- Real-world experience with Core Java

Instructor Welcome

Instructor Introduction:

Name

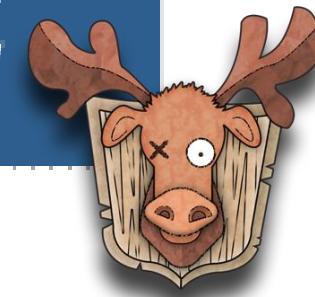
Background

Qualifications / Experience

Something interesting about yourself, hobby, etc.

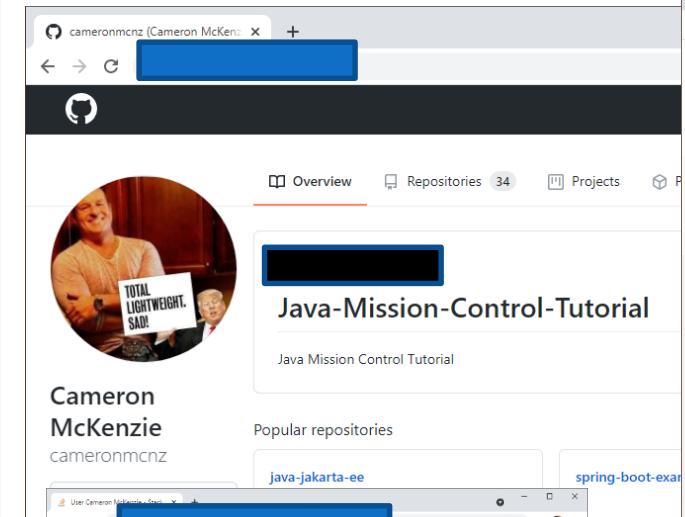


Welcome to the class. I'm Cameron McKenzie



About Me

I'm Cameron McKenzie



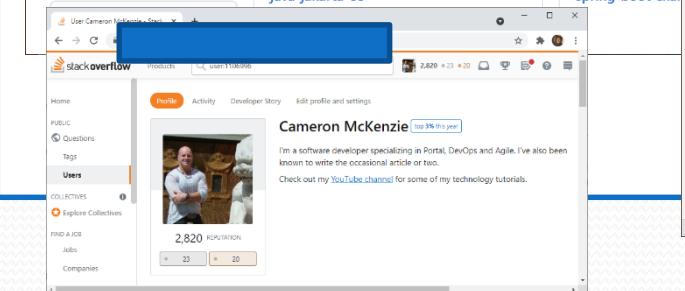
cameronmcnz (Cameron McKenzie) Overview Repositories 34 Projects

 Cameron McKenzie TOTAL LIGHTWEIGHT. SAD!

Java-Mission-Control-Tutorial
Java Mission Control Tutorial

Popular repositories

- java-jakarta-ee
- spring-boot-exam



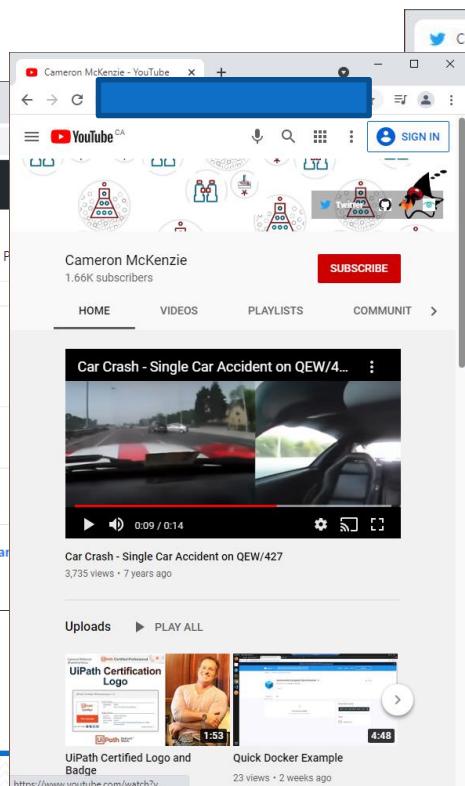
User Cameron McKenzie - Home Questions Tags Users EXPLORE COLLECTIVES FIND A JOB Jobs COMPANIES

Activity Developer Story Edit profile and settings

 Cameron McKenzie top 5% this year

I'm a software developer specializing in Portal, DevOps and Agile. I've also been known to write the occasional article or two. Check out my [YouTube channel](#) for some of my technology tutorials.

2,820 REPUTATION 23 20



Cameron McKenzie - YouTube CA

1.66K subscribers

SUBSCRIBE

HOME VIDEOS PLAYLISTS COMMUNITY

Car Crash - Single Car Accident on QEW/4...

0:09 / 0:14

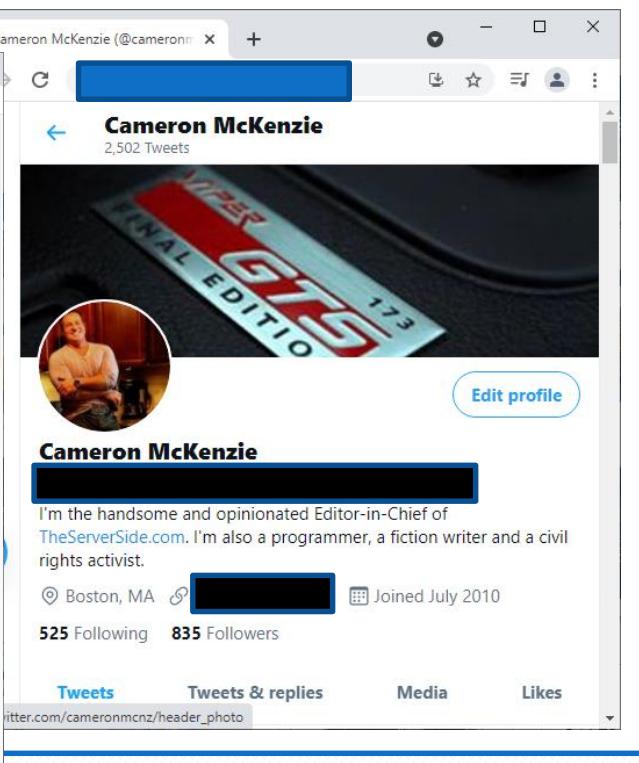
Car Crash - Single Car Accident on QEW/4... 3,735 views • 7 years ago

Uploads PLAY ALL

UIPath Certification Logo 1:53

UIPath Certified Logo and Badge 4:48

Quick Docker Example 23 views • 2 weeks ago



Cameron McKenzie (@cameronmcnz) +

2,502 Tweets

 Cameron McKenzie FINAL EDITION 173

Edit profile

Cameron McKenzie

I'm the handsome and opinionated Editor-in-Chief of [TheServerSide.com](#). I'm also a programmer, a fiction writer and a civil rights activist.

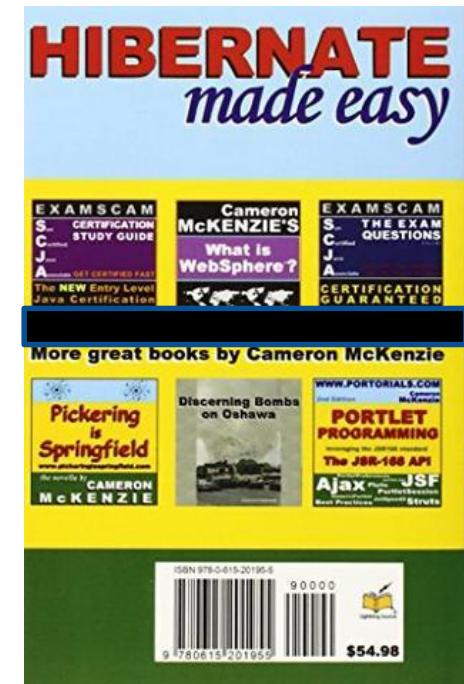
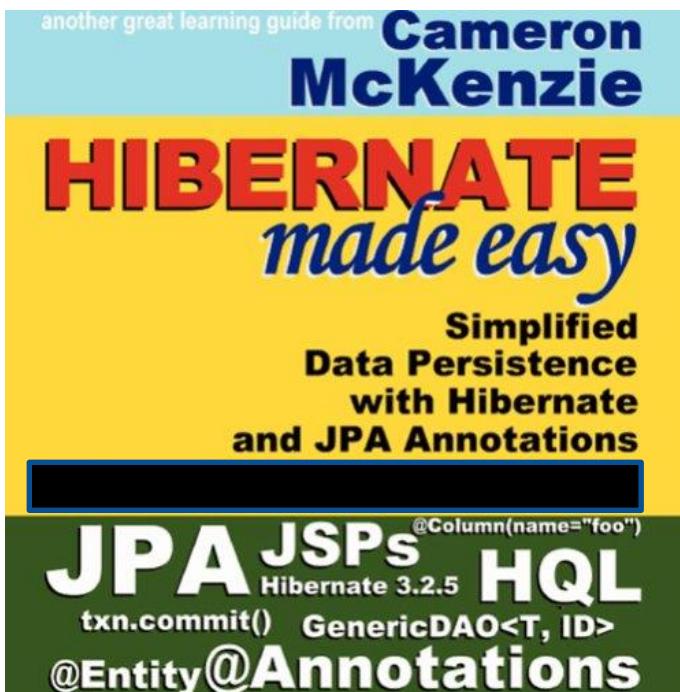
Boston, MA Joined July 2010

525 Following 835 Followers

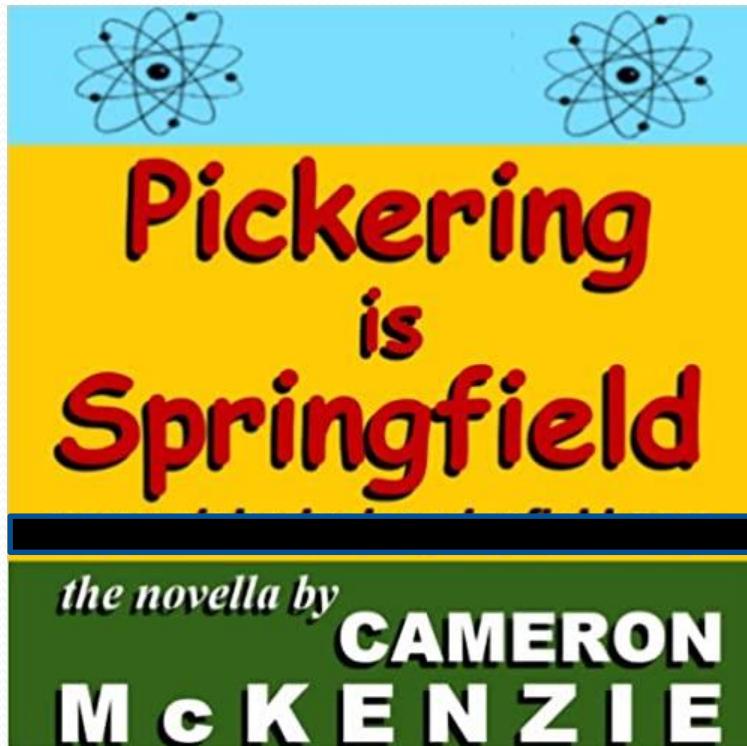
Tweets Tweets & replies Media Likes

https://twitter.com/cameronmcnz/header_photo

A few of my books



Why the horrible colors?



A screenshot of a web browser window displaying a page with the title "Pickering is Springfield" in large, red, stylized letters. The page has a light blue header and footer. In the center, there are two atomic symbols. On the left side, there is a sidebar with links: "Home", "Similarities", "*CanCon*", "CanQuotes", "The Quiz", "Buy Now!", "Learn Java", "tv & radio", and "Bombs Away!". Below these links is a small thumbnail image labeled "Battering Bombs on Shatner". The main content area contains text about Canadians in the US, featuring a photo of Mike Myers. To the right, there is a sidebar with "More Canadians >" and "amazon" links, along with thumbnails for "HARTMAN" and "Saturday Night Live - The...". At the bottom, there is a link "Click here" and a "Privacy Information" link.

Agenda

- Course Introduction
- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL
- Course Wrap-up

Participant Introductions

Name

Company

Department or Team

Projects

Programming background

Top two goals for this course

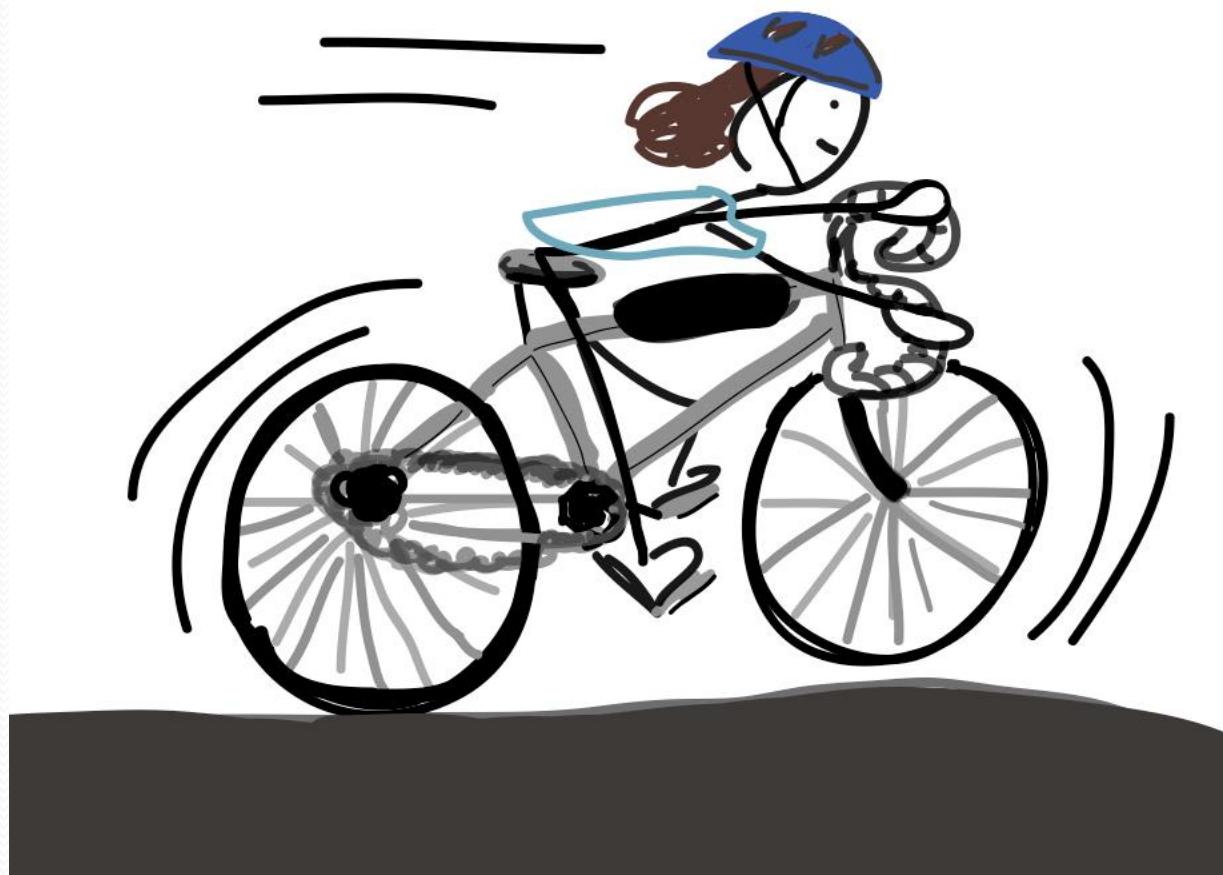


Poll - Hot Topics?

1. Which of the agenda topics are you most interested in?
 - a. JVM (Memory Changes, etc.)
 - b. Packaging Applications (Signing, Verifying JARs)
 - c. Best Practices for Exception Handling
 - d. Threading and Concurrency
 - e. Networking (Sockets)
 - f. Advanced JDBC
 - g. Performance
 - h. Writing Effective Java - best practices
 - i. Data Structures - optimizing for performance and task
 - j. Internationalization
 - k. All Java Security topics

Select your top three topics

Let's Get Started



Java Virtual Machine

Agenda

- **Java Virtual Machine**
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand what the Java Virtual Machine is

Understand where data is in the JVM

Understand where code is in the JVM

Understand memory management within the JVM

Learn about Lambda expressions in JDK 8

What is Java

- Java is an object-oriented, cross-platform programming language
 - Released in 1992, Java has been around for three decades
 - Invented by Sun Microsystems engineer James Gosling
 - Originally named Oak
- The Java trademark is currently owned by Oracle
 - Open source distributions of the JDK are available
 - Google used Java as the foundation of the Android OS
- Java remains one of the most popular programming languages in the world

Benefits of Java

- The original release of Java boasted about these qualities:
 - Platform independence
 - Multithreading support
 - Built in security features
 - Strong networking and distributed computing capabilities
 - Portability
 - Robustness
 - Openness

Uses of Java

- Java has become a popular language for application development
- Java is commonly used in the following spaces:
 - Traditional Enterprise development targeted to servers such as:
 - Tomcat
 - JBOSS
 - Jetty
 - WebSphere
 - Batch Processing
 - Easy Batch, Quartz and Jshell
 - Microservice development
 - Spring Boot and Eclipse Microprofile are popular frameworks
 - Android App development
 - Tool development
 - Open Source tools and technologies such as Hadoop and Jenkins

Version Implications

- 3% of Java in use is still JDK < 1.7_45
 - JDK 1.7_45 100+ Security bugs
 - JDK 1.7 End of Life (EOL) April 2015
- 64% are on Java 8
- 25% are on Java 11
- LTS Java version 17 was just released
- Issues with not upgrading the latest JDK?
 - Class loader formats
 - Method deprecation
 - Security
 - New features
 - Garbage collectors



Plan to upgrade?

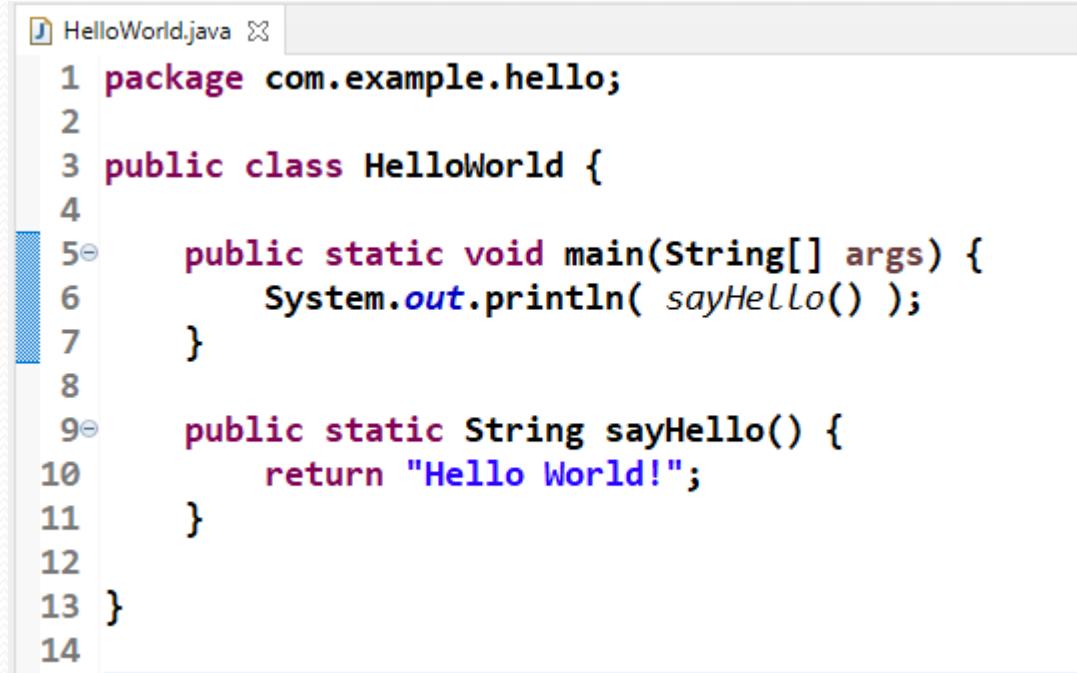
- Organizations should commit to an upgrade schedule
 - Why not take advantage of improved performance, security and an enhanced feature set?
- Are the following objections to a JDK upgrade reasonable?
 - The current setup is good enough
 - The business won't agree on migration
 - Cost is too great
 - The new release cycle is confusing
 - No new features are needed



Java Syntax Overview

- Java syntax was inspired by C++
 - The main method is the entry point of an application
 - Curly braces mark the beginning and end of programming constructs:
 - classes
 - methods
 - loops
 - conditional statements
 - Statements end with a semi-colon;
 - Packages refer to the subfolder in which a Java file is located
 - Java programmers prefer the term 'method' to 'function'

Sample Java Code

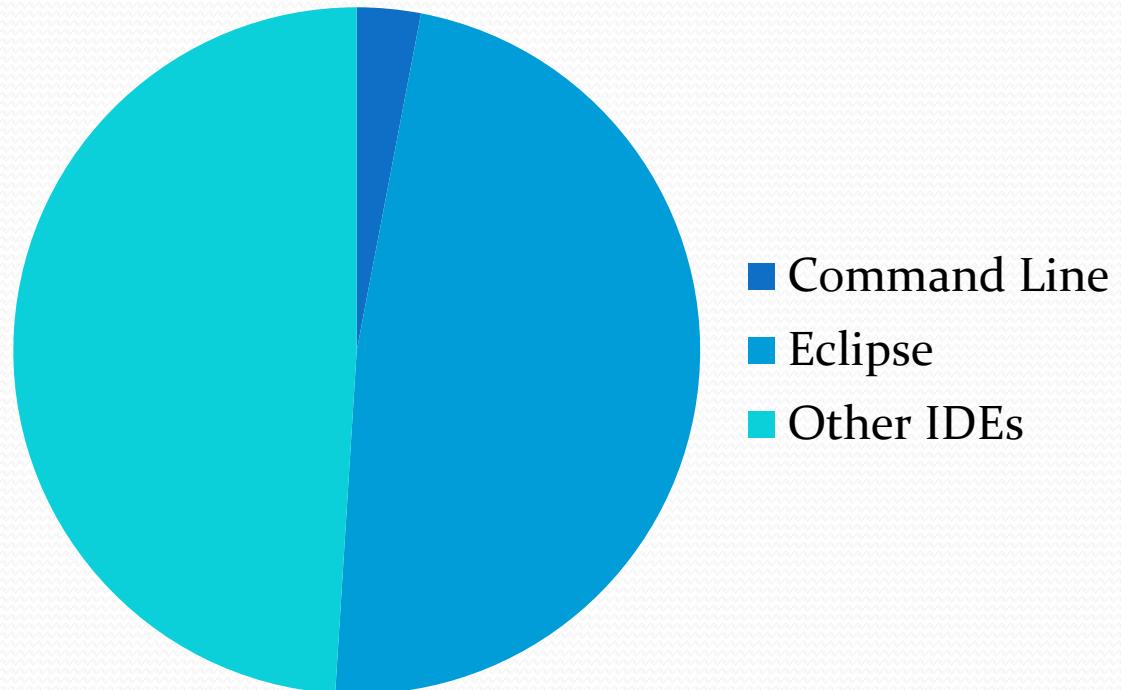


A screenshot of a Java code editor showing a file named `HelloWorld.java`. The code is a simple "Hello World" application. The editor interface includes a title bar with the file name, a toolbar with a single icon, and a code editor area with syntax highlighting.

```
1 package com.example.hello;
2
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         System.out.println( sayHello() );
7     }
8
9     public static String sayHello() {
10        return "Hello World!";
11    }
12
13 }
```

Software Development Tools

- Which IDE do you prefer?
 - 97% of developers use an IDE
 - 48% use Eclipse
- Other IDEs
 - IntelliJ - 43%
 - Netbeans - 5%



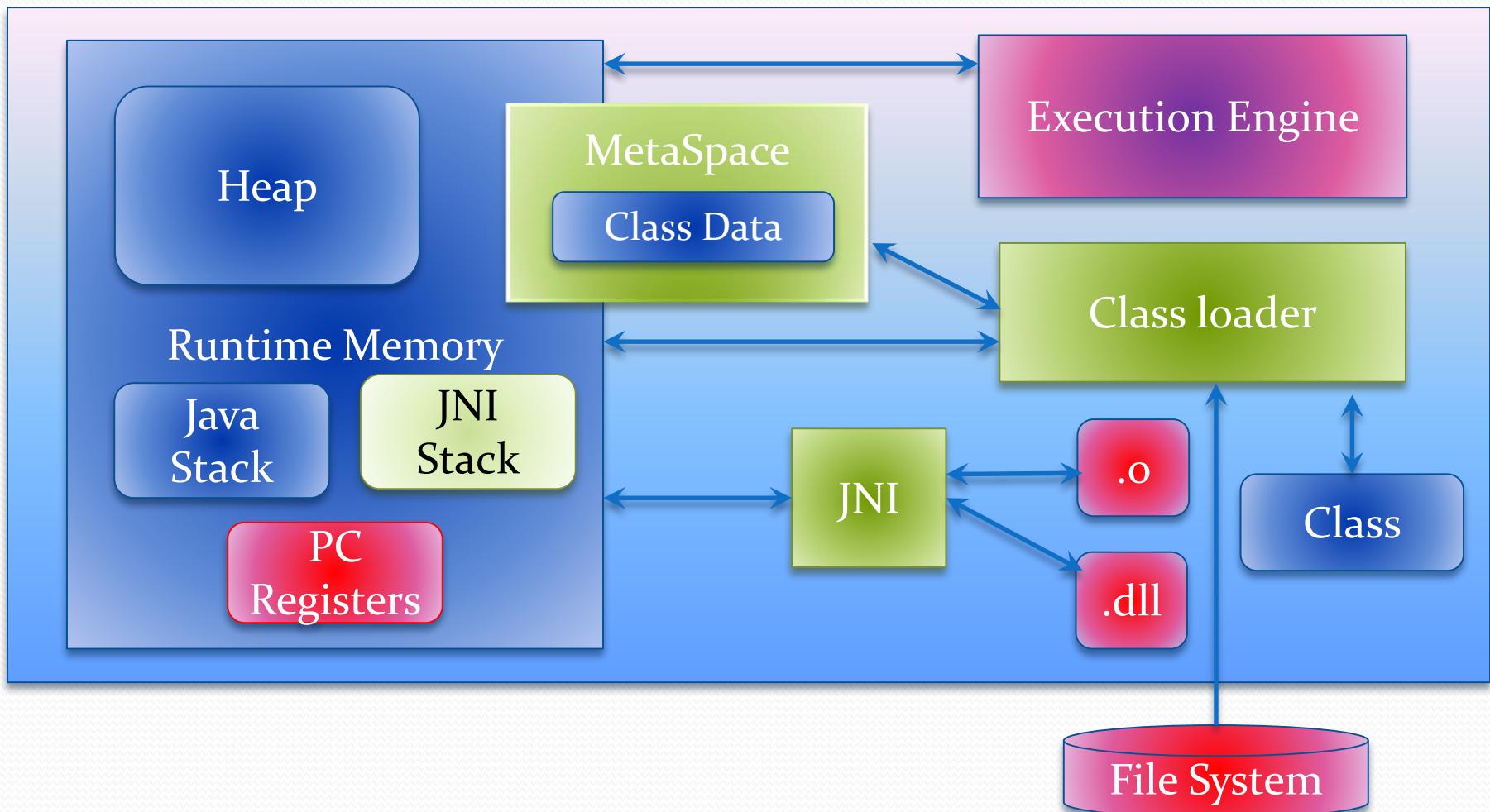
Cross Platform Java

- Java source code is saved in a file with a .java extension
 - Each public Java class is saved in its own .java file
- Java source code is compiled into bytecode
 - Each .java file gets compiled into a corresponding .class file
 - The .class file contains the bytecode
- Many programming languages compile directly into machine code
 - Bytecode is an intermediary compilation format
 - Bytecode requires a Java virtual machine (JVM) to run
- Every platform must have a platform-specific JVM to run bytecode
 - This is what makes Java programs cross-platform
 - Java compiles into platform neutral bytecode
 - Each platform has its own JVM capable of running bytecode

Supported OS

- Linux
- Mac
- Windows
- Solaris
- HPUX
- AIX
- Mainframes (POSIX)
- iOS
- Android

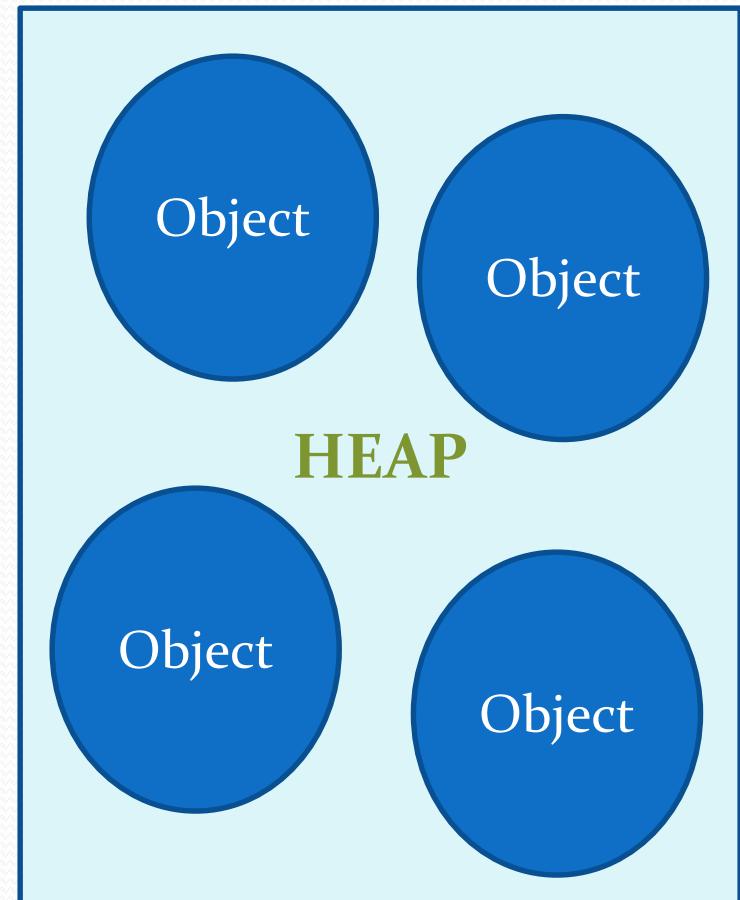
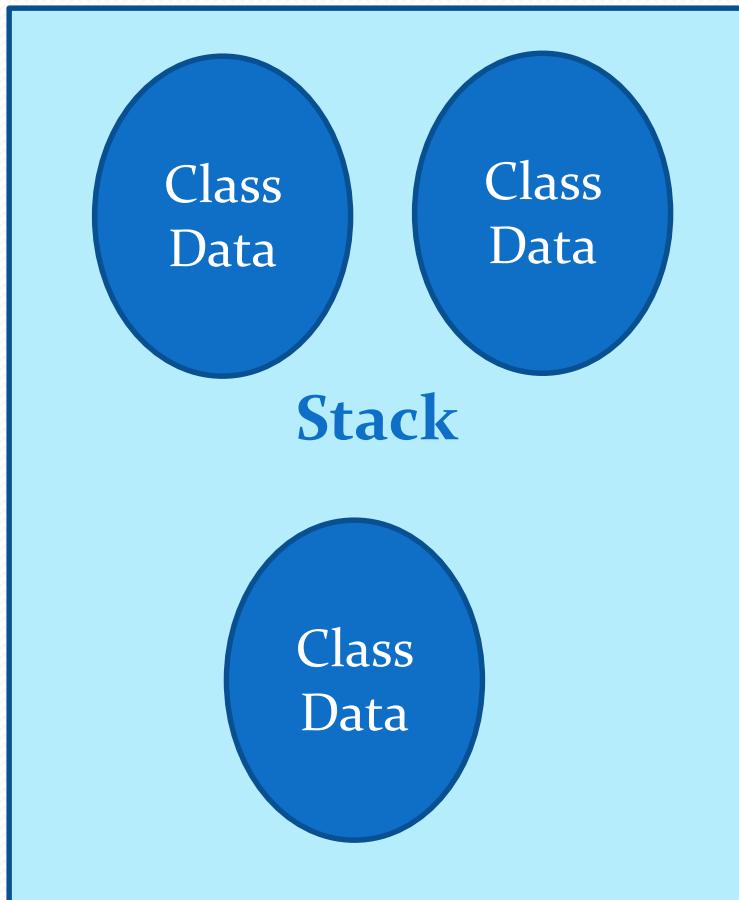
Architecture of the JVM



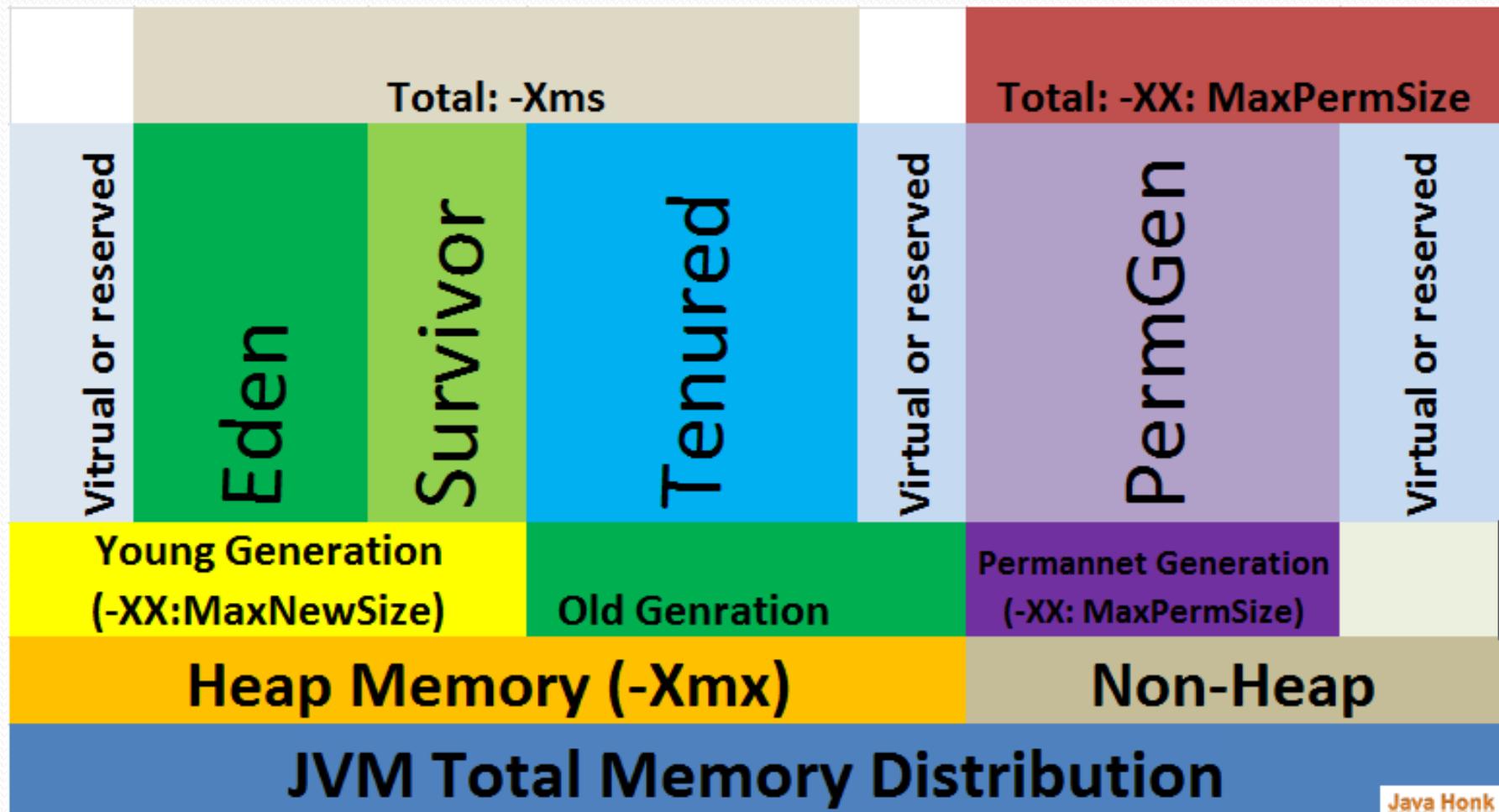
Reference Objects

- Objects in the heap are allocated through a reference
- Reference types
 - class type - an instance defined as a class
 - interface type - an instance defined as an interface
 - array type - an instance defined as an array
 - null - doesn't reference anything
- JVM Specification defines the types and limits but not the implementation or size
- Java also makes use of Soft, Strong and Weak references
 - Strong need to survive
 - Weak are usually transient and can be finalized at any time

Stack vs Heap



Memory Usage Pre Java 8

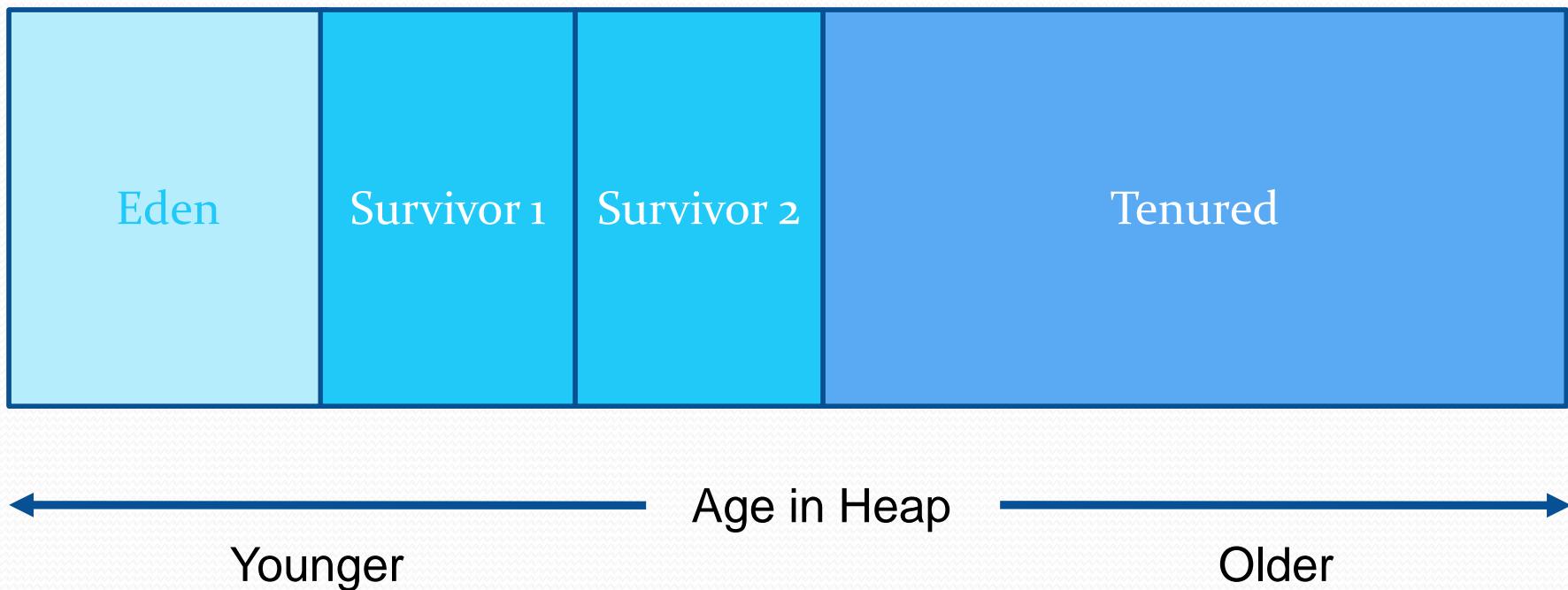


MetaSpace vs. PermGen

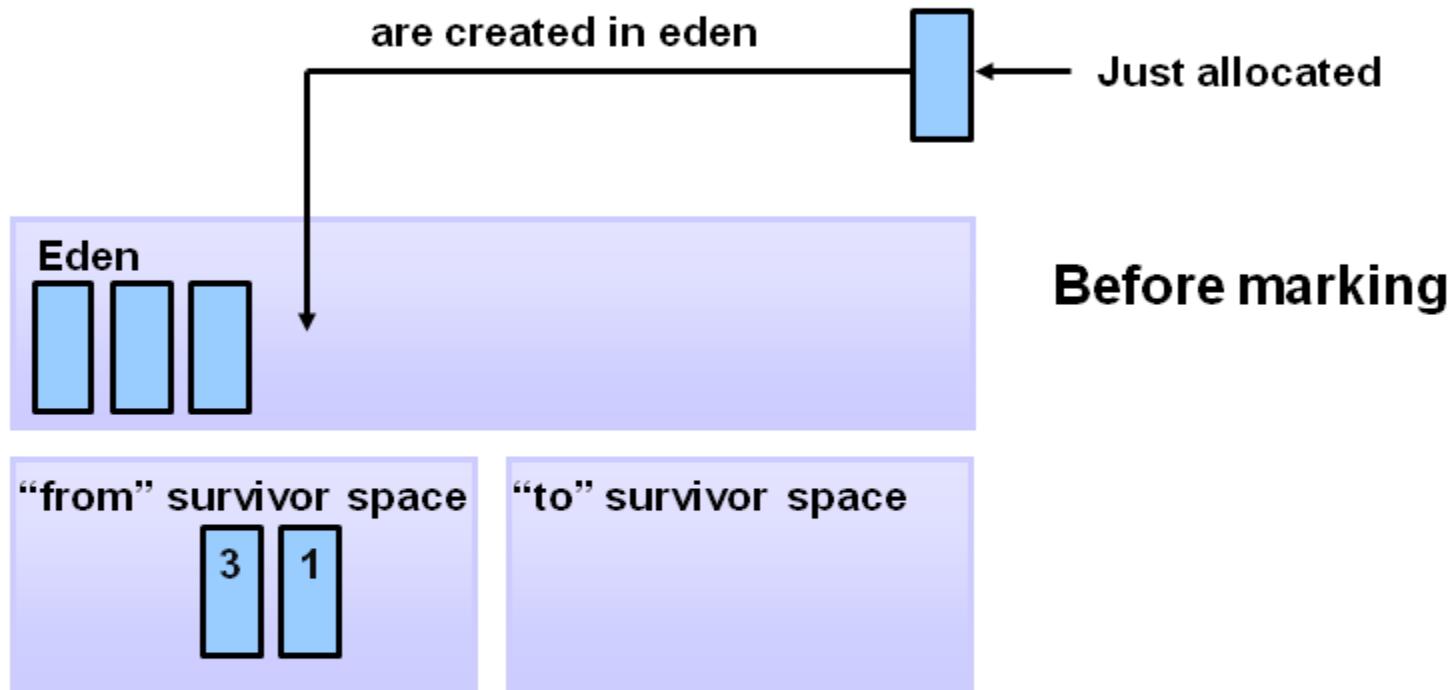
- **PermGen** was where class metadata was loaded in JDKs prior to JDK 8
 - Not movable
 - Allocated at startup
 - Default very low
 - Dynamic class generation is all the rage
 - Gets used up quite quickly
- **MetaSpace** allocated in Native memory
 - Not in the Heap
 - Managed by OS not Java
 - Gives the Heap back some memory to work with
- Biggest gainer
 - Dynamic classes frameworks
 - Multiple class loader environments - like application servers

Memory Usage Java 8 - GC

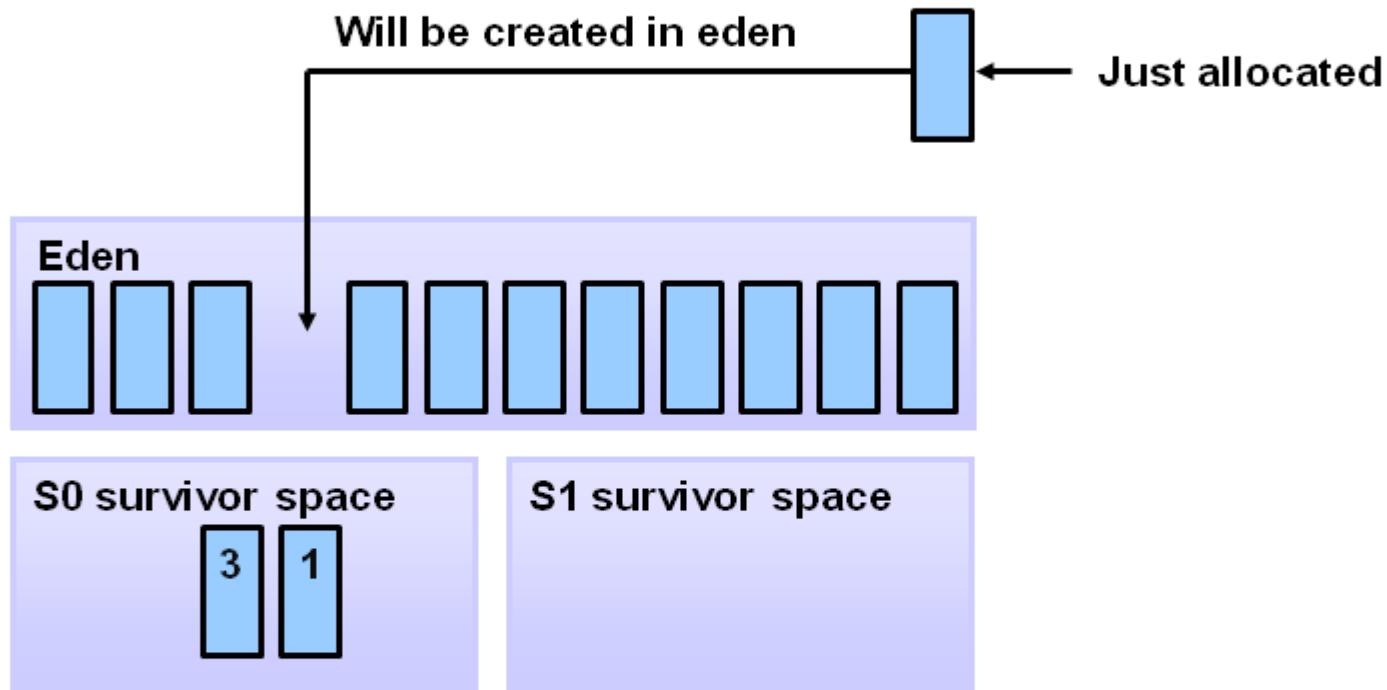
MetaSpace



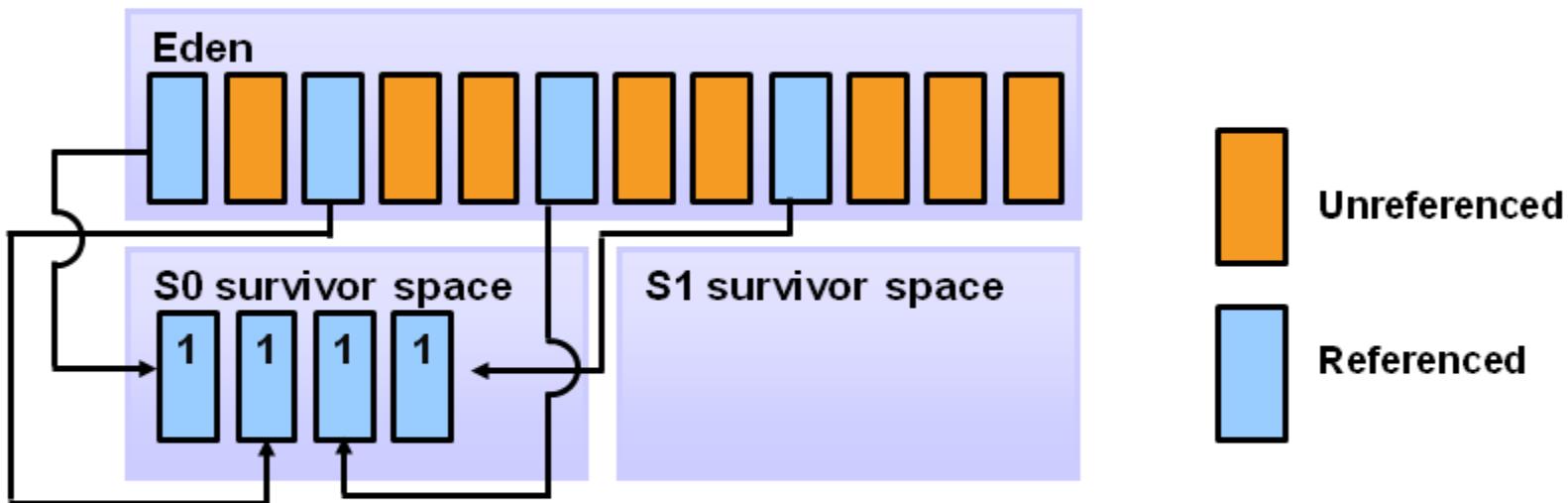
Object Allocation



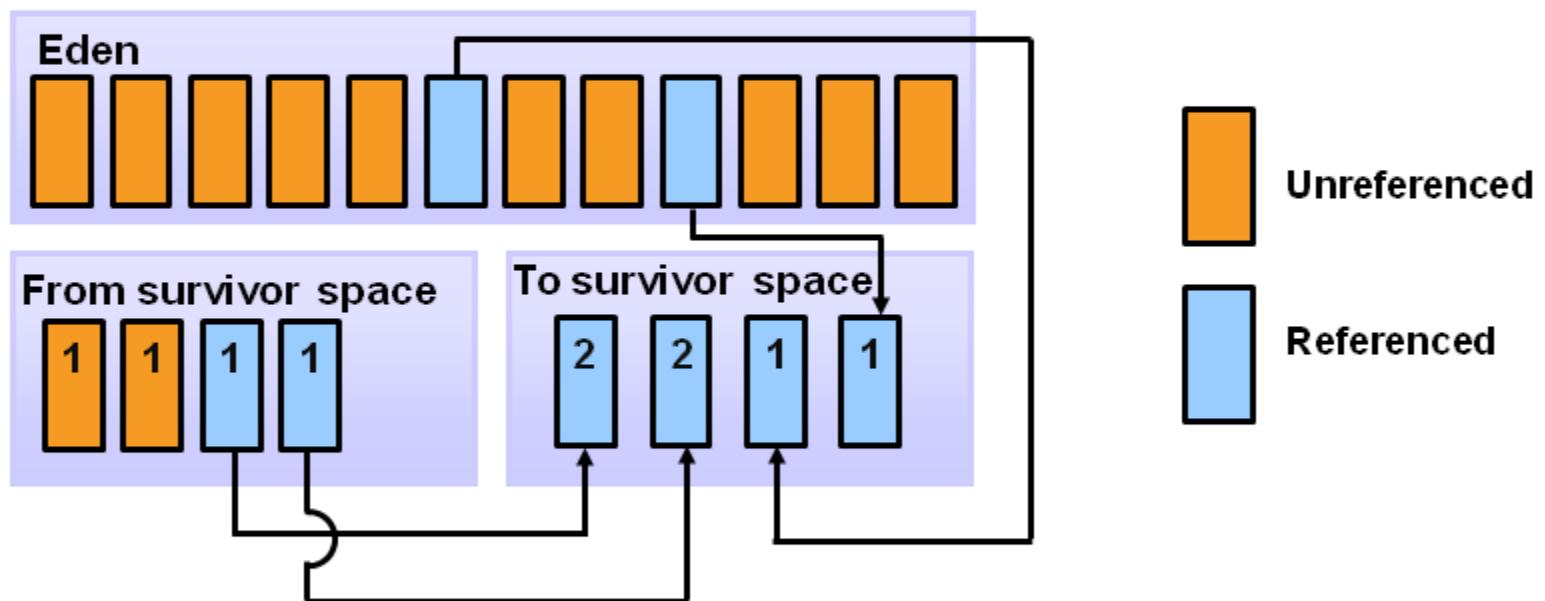
Filling the Eden Space



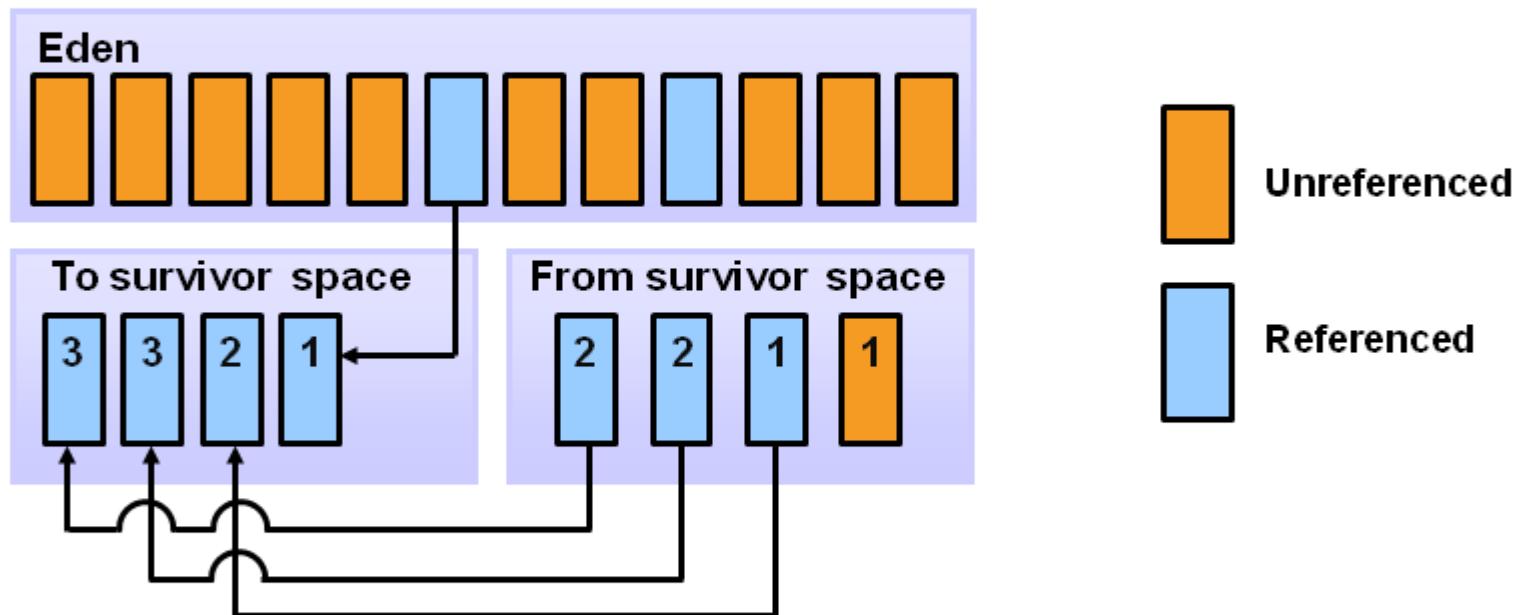
Copying Referenced Objects



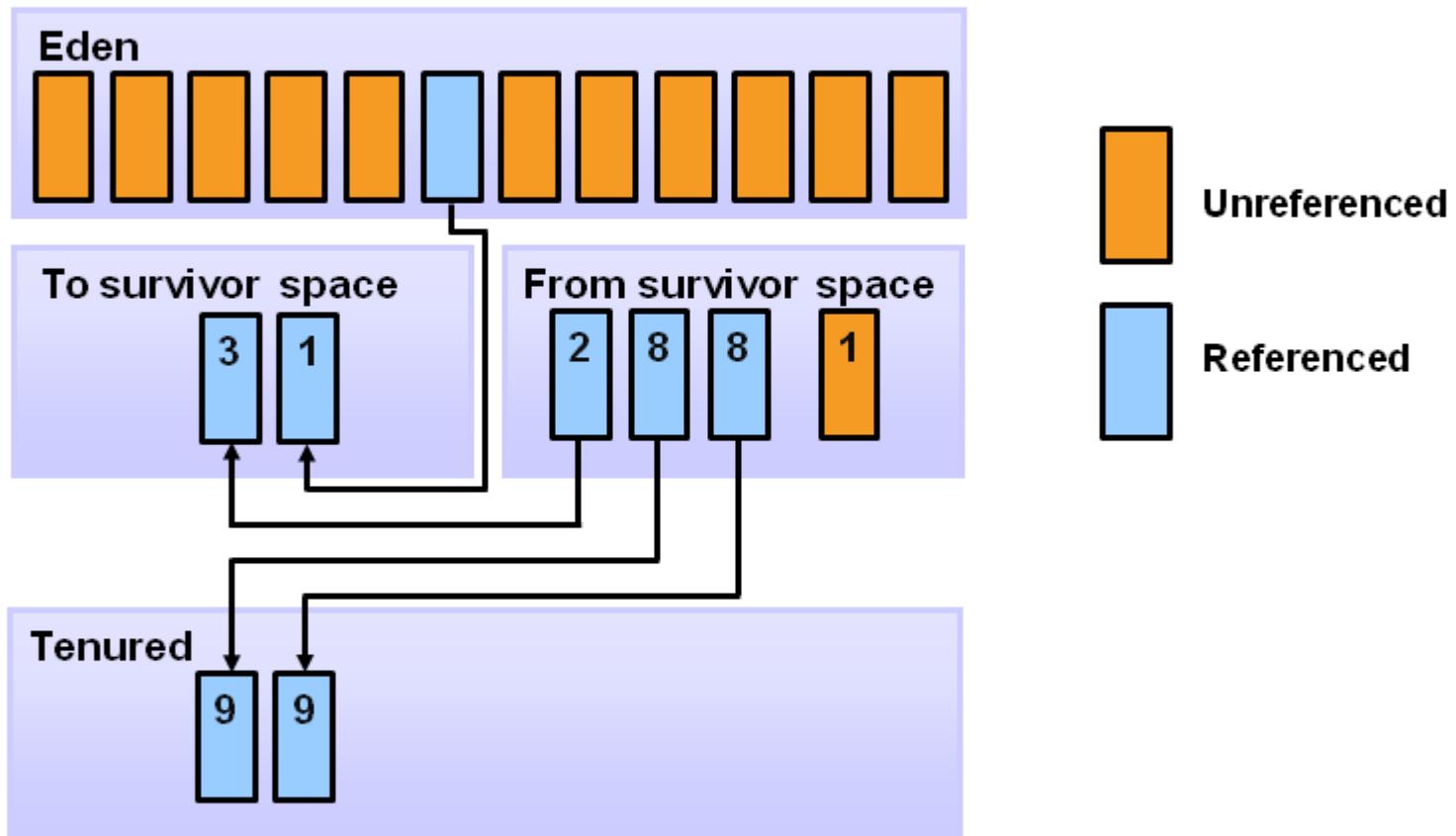
Object Aging



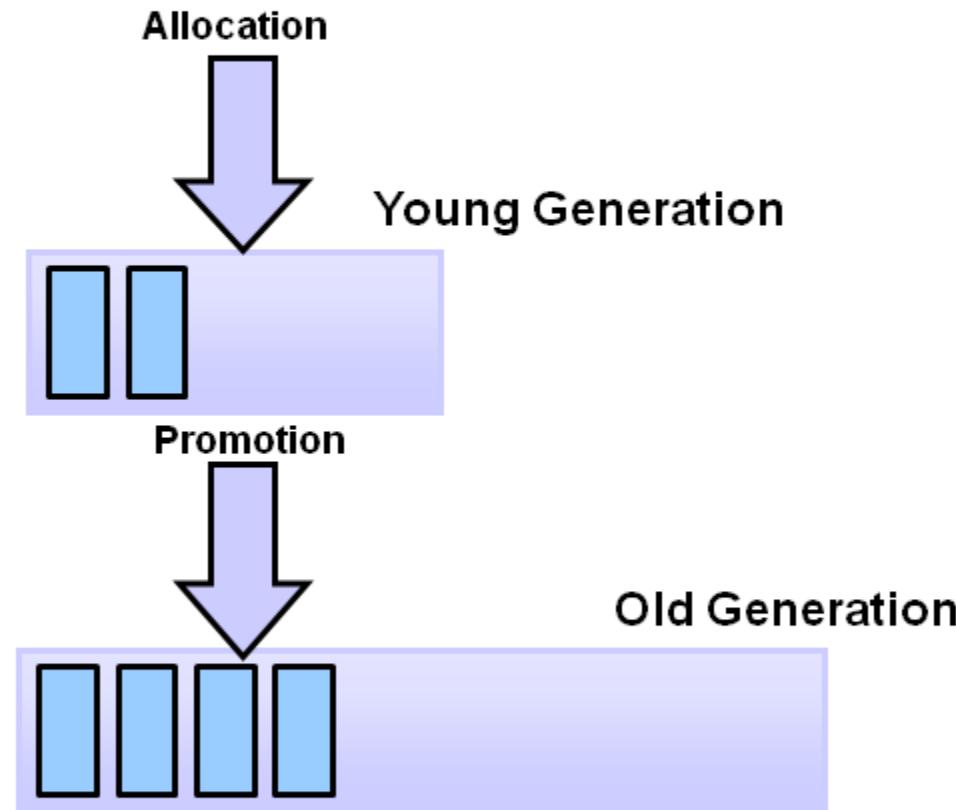
Additional Aging



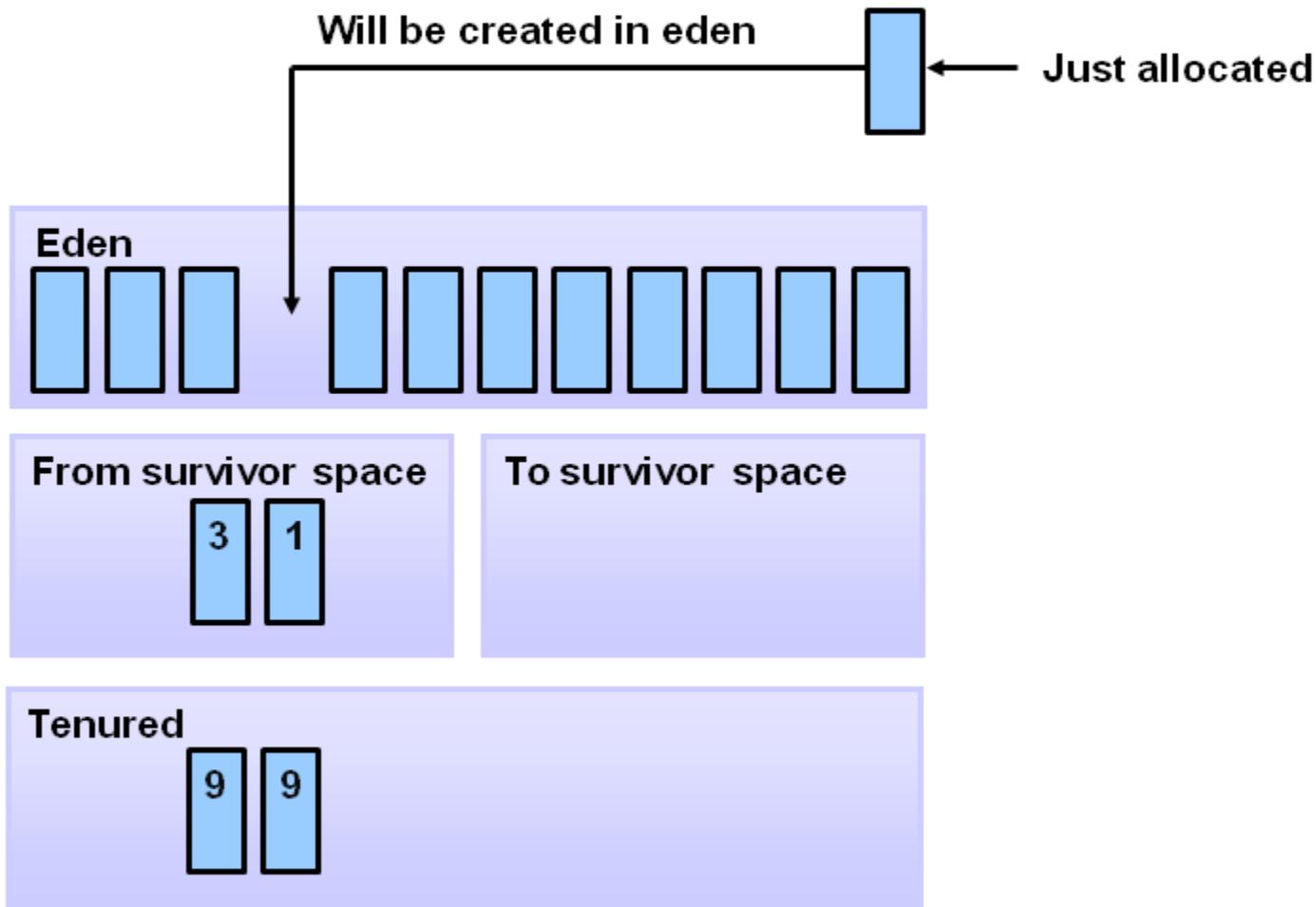
Promotion



Promotion



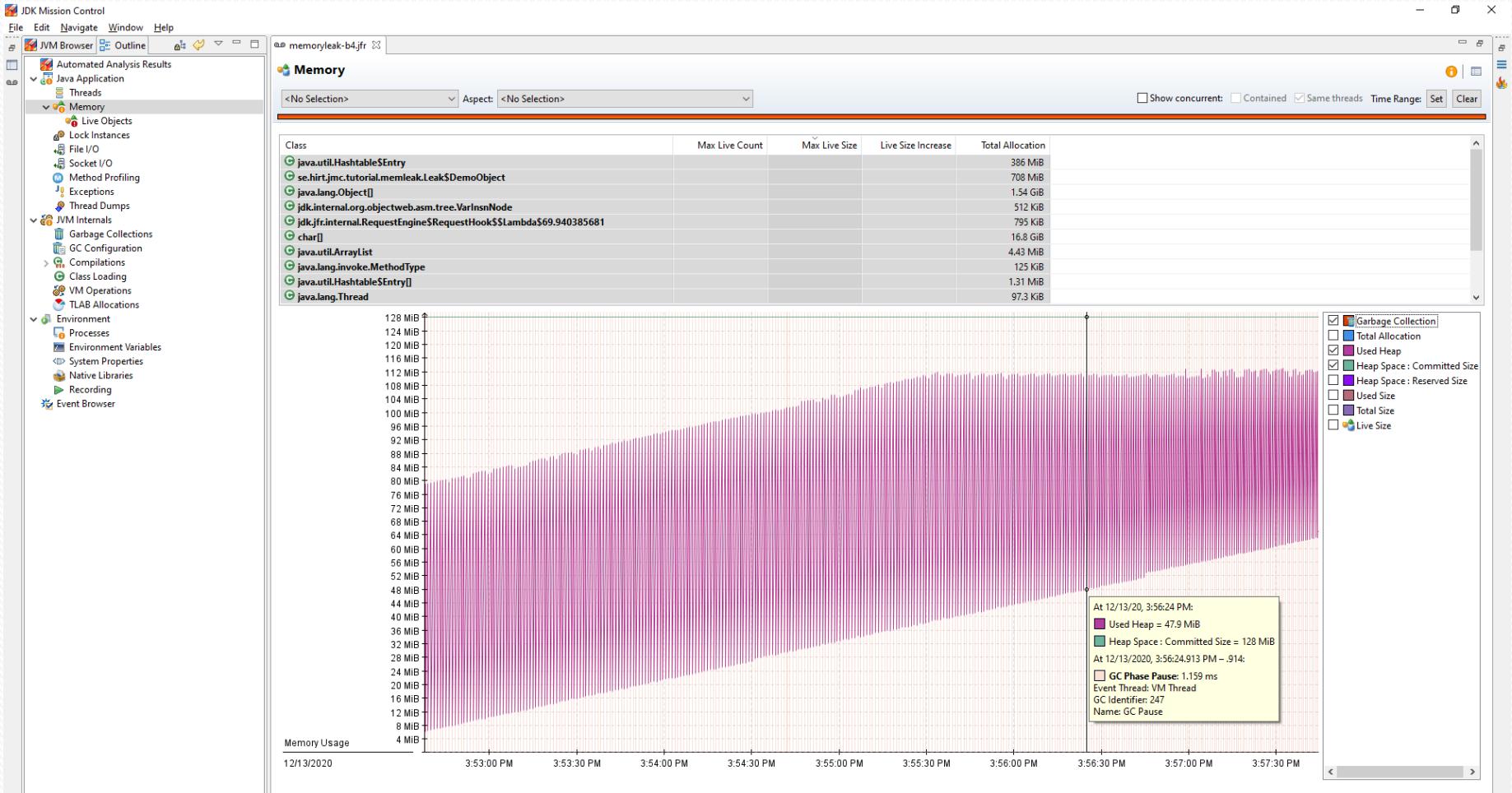
GC Process Summary



Memory Management

- JVM can grow and shrink the heap
 - xms minimum**
 - xmx maximum**
- Bigger heap - more time scanning during GC
- Smaller heap - potential to get an **OutOfMemoryError**
- In JDK 7 or older - dynamic class generation can use **PermGen** but there is a limit
- Using more than one dynamic class generator can create the same class more than one time in PermGen
- Class loaders isolate libraries and code versions but may duplicate memory contents

Profile of a Memory Leak



Review

- What is the significant change to the way Java 1.8 handles memory allocation?



Lambda Expressions

- One of the biggest changes JDK 8 brought was the introduction of Lambda Expressions
 - Less wieldy than a local or anonymous class
 - Defines code as data that can be passed to a method
 - Define instances of single-method classes more compactly
 - Similar to C# or Scala functionality
 - Developers have asked for it for years

What Is a Lambda Expression?

- Essentially a method without a declaration
- Lambda expression consists of a parameter list and a body separated by the ->
 - (**a, b**) -> **a+b** returns the sum of a and b
 - (**a, b**) -> **a-b** returns the difference of b from a
 - (**Boolean a, Boolean b**) -> **Boolean(a.value || b.value)**
OR of a and b returned as a Boolean
- Can be assigned to a Functional interface
 - Runnable t = () -> System.out.println();**
 - Above could be passed to a Thread constructor to run as a runnable object

Lambdas and Class Types

- Lambda expressions simplify the creation of local and anonymous classes
- **Lambda Expressions**
 - Like anonymous classes or local classes but more useful
- Local classes
 - Defined in any block
 - Like a local variable, goes out of scope at end of block
- Anonymous classes
 - Defined when passed as a parameter to a method
- Nested classes
 - Defined in another class
 - Can also have class modifiers

```
1 package com.mcnz.lambdas;
2
3 import java.util.*;
4
5 public class InterfaceExample {
6
7     public static void main(String args[]) {
8
9         Comparator<Integer> comparator = new NumberComparator();
10        int result = comparator.compare(20, 30);
11        System.out.println(result);
12    }
13}
14
15}
16
17 class NumberComparator implements Comparator<Integer> {
18
19     public int compare(Integer a, Integer b) {
20         return a-b;
21     }
22
23}
24
25
```

```
1 package com.mcnz.lambdas;
2
3 import java.util.*;
4
5 public class InnerClassExample {
6
7     public static void main(String args[]) {
8
9         Comparator<Integer> comparator = new NumberComparator() {
10            @Override
11            public int compare(Integer a, Integer b) {
12                return a-b;
13            }
14        };
15
16        int result = comparator.compare(20, 30);
17        System.out.println(result);
18
19    }
20
21 }
22 }
```

InterfaceEx... InnerClassEx... LambdaExampl... CompareArray... MikeTyson.java

```
1 package com.mcnz.lambdas;
2
3 import java.util.*;
4
5 public class LambdaExamples {
6
7     public static void main(String args[]) {
8
9         Comparator<Integer> comparator = (x, y) -> x-y;
10
11     int result = comparator.compare(20, 30);
12     System.out.println(result);
13
14 }
15
16 }
17 }
18 }
```

```
2  
3 import java.util.*;  
4  
5 public class CompareArrayElements {  
6  
7     static Integer[] numbers = {5, 12, 11, 7};  
8     static String[] names = {"adam", "brian", "cj", "dahlia"};  
9  
10    public static void main(String args[]){  
11        Arrays.sort(numbers, new Comparator<Integer>(){  
12            @Override  
13            public int compare(Integer a, Integer b) {  
14                return a-b;  
15            }  
16        });  
17  
18  
19        Arrays.sort(names, new Comparator<String>(){  
20            @Override  
21            public int compare(String a, String b) {  
22                return a.length()-b.length();  
23            }  
24        });  
25  
26        System.out.println(Arrays.toString(numbers));  
27        System.out.println(Arrays.toString(names));  
28  
29    }  
30 }
```

```
1 package com.mcnz.lambdas;
2
3 import java.util.*;
4
5 public class CompareLambda {
6
7     static Integer[] numbers = {5, 12, 11, 7};
8     static String[] names = {"adam", "brian", "cj", "dahlia"};
9
10    public static void main(String args[]) {
11
12        Arrays.sort(numbers, (x,y) -> x-y);
13        Arrays.sort(names, (a,b) -> a.length() - b.length());
14
15        System.out.println(Arrays.toString(numbers));
16        System.out.println(Arrays.toString(names));
17
18
19    }
20
21 }
22
23
```

Traditional Example

```
public class Account {  
    float balance = 10.0;  
    class PaymentReceiver implements ReceivePayment{  
        void creditPayment (float amount) {  
            balance += amount;  
        }  
    }  
    public void process() {  
        PaymentReceiver t = new PaymentReceiver ();  
        t.creditPayment(2.00);  
    }  
}  
  
interface ReceivePayment {  
    void creditPayment(float amount);  
}
```

Lambda Example

```
public class Account {  
    float balance = 10.0;  
    public void process() {  
        ReceivePayment t = (float amount) -> balance += amount;  
        t.creditPayment(2.00);  
    }  
}  
  
interface ReceivePayment {  
    void creditPayment(float amount);  
}
```

Advanced Lambda Examples

- What do you think of these advanced Lambda examples?
 - Is the logic easily deciphered?

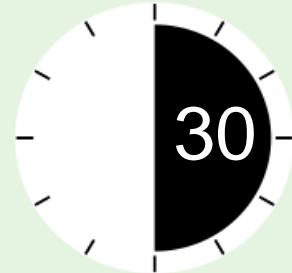
```
Map<String, Integer> jobs = new HashMap<>();  
  
jobs.put("Jr Developer", 40000);  
jobs.put("Sr Developer", 30000);  
jobs.put("Architect", 50000);  
  
jobs.forEach((k,v)->System.out.println("Job : " + k + " Salary : " + v));  
  
jobs.computeIfAbsent("Jr Programmer", s -> s.length()*10000);  
jobs.computeIfPresent("Architect", (k,v) -> k.length()*10000);  
jobs.computeIfPresent("Sr Programmer", (k,v) -> k.length()*10000);  
  
jobs.replaceAll((job, salary)->new Integer(salary.toString().replaceAll("0", "5")));  
  
jobs.forEach((k,v)->System.out.println("Job : " + k + " Salary : " + v));
```

Summary

- Java Virtual Machine Specification defines what Java should do
 - Vendors implement that specification
 - Developers try to build applications that run on them
 - Some vendor decisions directly affect developers
 - When different vendors implement differently, applications might not be portable
-
- Continue to Lab Introduction on next slide



Lab 1 Introduction



- Using Lambda Expressions
- You will use three different ways to define a method for implementing Java logic
 - Nested class
 - Local class and anonymous class
 - Lambda expression
- After the lab, we will have a lab review
 - If virtual, indicate when you are done by a green check

Lab 1 Review - Discussion



- Using Lambda Expressions
- Three different ways to define a method for implementing Java logic
 - Nested class
 - Local class and anonymous class
 - Lambda expression
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

Packaging Applications

Agenda

- Java Virtual Machine
- **Packaging Applications**
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand how Java stores classes and resources

Learn how to ensure the content of a JAR can be trusted

Practice the mechanics of sealing and signing

Java Artifacts

- A Java program may be made up of hundreds of different Java files
 - And a program may use any number of third-party libraries that contain hundreds of Java files
- A Java build gathers all of the required Java class files, resources and libraries and saves them inside a single, compressed file
 - **JAR** - Java ARchive for libraries, stand-alone Java apps and cloud-native apps with embedded application servers
 - **WAR** – Web Application aRchive for traditional Servlet and JSP based apps that are deployed to Java Web application servers
 - **EAR** – Enterprise Application aRchives for Java applications that contain Java web modules and EJB modules
- All of these file types use a standard compression algorithm and can be opened with any standard decompression utility (WinZip, 7-Zip, WinRAR)
- The goal of a Java build is to generate a deployment artifact

JAR Files

- Java Archive is a cross-platform open format to archive:
 - Images
 - Classes
 - Audio content
 - XML
 - Properties
- Based on PKWARE zip archive
- Specific folder for adding Signed content

Structure and Content

- META-INF folder
 - MANIFEST.MF - metadata about the JAR
 - INDEX.LST - location information for packages
 - x.SF - signature file for the JAR - x base file name
 - x.DSA - signature block to go with the .SF file
 - /services
 - Service provider configuration files
- Content is name-value pairs
 - MANIFEST.MF
 - Manifest-Version: 1.6
 - Main-Class: somepackage.SomeClass
 - Class-Path: xyz.jar abc.jar def.jar
 - Sealed: true - all packages in this JAR are sealed
 - Name: com/xyz - package name not sealed
 - Sealed: false

Update the MANIFEST.MF

- Edit the values in a file Manifest.txt
- Use the following JAR command
`jar cfm Somejar.jar Manifest.txt SomeFolder/*.class`
- Content will be updated in the MANIFEST.MF in the META-INF folder

Application Entry Points

- Defined in the MANIFEST.MF

Main-Class : somepkg.SomeClass

- Class must have a `public static void main(String [] args)` method
- What if you want more?
- Create your own custom class loader and define your own mechanism

Versioning

- Per package, assign version numbers
- Name: java/util
- Specification Title: Java Utility Classes
- Specification Version: 1.2
- Specification Vendor: Some Company, Inc.
- Implementation Title: java.util
- Implementation Version: build199
- Implementation Vendor: Some Company, Inc.

Classes JARs and Packages

- Sealing guarantees that all the classes in the JAR come from the same code source
- Specific packages can be excluded
- Version consistency and security
- In the MANIFEST.MF
 - No package specified - all packages are sealed
Sealed : true
- Unseal a specific package
Name : /package/
Sealed : false
- Seal a specific package
Name : /apackage/
Sealed : true

Signing and Versioning

- We will discuss signatures and public-private key pairs more later
- Signed using **jarsigner**
- Signed with a private key
- Public key is put in the JAR along with the signatures
- Certificate is also put in the JAR for verification
- Sign using command
`jarsigner jar-file key-alias`
- The public and private keys need to be in the **keystore**
 - Default **.keystore** in the home directory

JAR Version Improvements

- JDK 6 added **-e** parameter to specify the **Application Entry Point** to override the Main-Class in the MANIFEST.MF
 - Extracted files get the timestamp in the JAR not the extraction timestamp
 - ZIPs now allow more than 64k entries
 - Removed some Windows ZIP limitations
 - Filenames longer than 256 allowed
 - Allowed to have more than 2000 open files
 - JDK 7 defaults to SHA-256
 - JDK 8
 - **Timestamp Authority (TSA)** URL can be specified
 - Restricts signing JARs with insecure algorithms
 - Verify insecure algorithms in signed JARs
- jarsigner -verify -J-Djava.security.debug=jar**

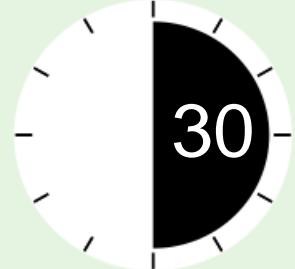
Summary

- JAR files hold classes that can be accessed by the class loader
- META-INF holds important meta information about the classes and their dependencies
- Signing establishes a level of trust between classes and the JVM



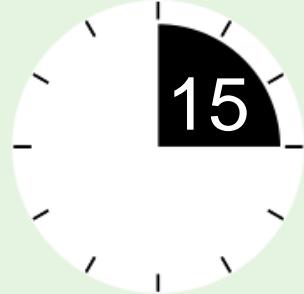
- Continue to Lab Introduction on next slide

Lab 2 Introduction



- Signing and Verifying a JAR
- You will generate a self-signing certificate so that you can:
 - Sign a JAR
 - Seal a JAR
 - Verify that it was signed properly
- After the lab, we will have a lab review
 - If virtual, put up a green check when you are done

Lab 2 Review - Discussion



- **Signing and Verifying a JAR**
- You generated a self-signing certificate to:
 - Sign a JAR
 - Seal a JAR
 - Verify that it was signed properly
- We will discuss the *digital certificate* in detail later
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

Best Practices for Exception Handling

Agenda:

- Java Virtual Machine
- Packaging Applications
- **Best Practices for Exception Handling**
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand how Java processes Exceptions

Discover the difference between the main Exception types

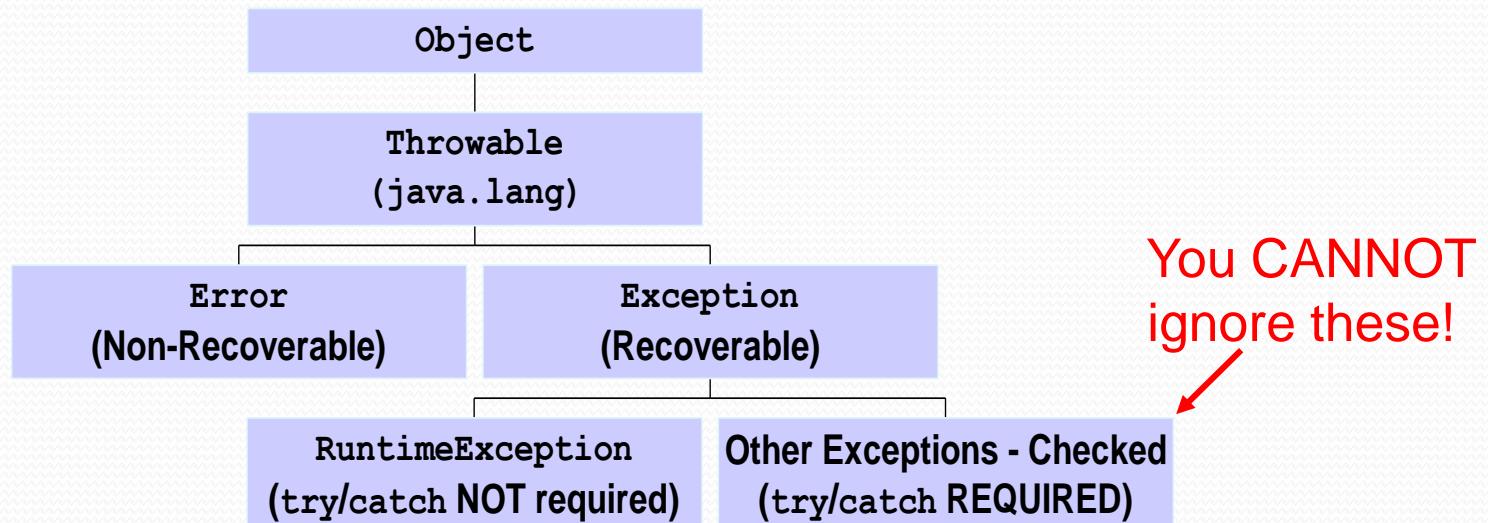
Learn best practices for Exception handling

Learn about what is new in JDK 8 regarding Exceptions

Exceptions

- Signals that some unexpected/abnormal condition has occurred **OR** that an error has been detected in the program
- Exception objects subclass `java.lang.Exception`

Java's Exception Inheritance Hierarchy



Custom Exceptions

- If your application needs a "customized" checked exception, then create your own exception class

- Must subclass **Exception**

```
public class WithdrawException extends Exception
```

- Code the exception

- **Don't forget:** Display or log appropriate message for when the exception occurs (Users need helpful information)

- Example - If users try to withdraw too much money from their bank accounts, then a useful message might be:
 - "Your current balance is _____. Withdrawal rejected."



Exception Keywords

- **throw** - Indicates that an exception is signaled
 - Used in method body
 - Program execution is stopped
 - Resumed at the nearest containing **catch** statement
 - catch statement must be able to handle the specified exception object
- **throws** - Lists possible exceptions that the method can throw
 - Used in method declaration
 - See next slide

Throw and Throws Example

```
public void withdraw(int amt) throws WithdrawException {  
    if (balance > amt){ //Balance can't be 0 or less  
        balance -= amt;  
    }  
    else { // Insufficient Funds  
        throw new WithdrawException(balance, amt);  
    }  
}
```

Exception Keywords (continued)

- **try** - Indicates a block of code to which subsequent **catch** and **finally** blocks apply
 - Code that can throw an exception must be in a try block
- **catch** - Handles exceptions
 - The catch keyword is followed by:
 - Exception type and argument name in parentheses
 - A block of code within curly braces
- **finally** - (Optional) Used for cleanup
 - *finally* code will always be executed before the method ends (regardless of any exceptions that may be *thrown*)

try/catch Example

- Sample code to catch an exception:

```
try {  
    transactions.addSale(type, price);  
    transactions.printReport();  
} catch (WithdrawException e) {  
    System.out.println(e);  
}
```

What happens here?

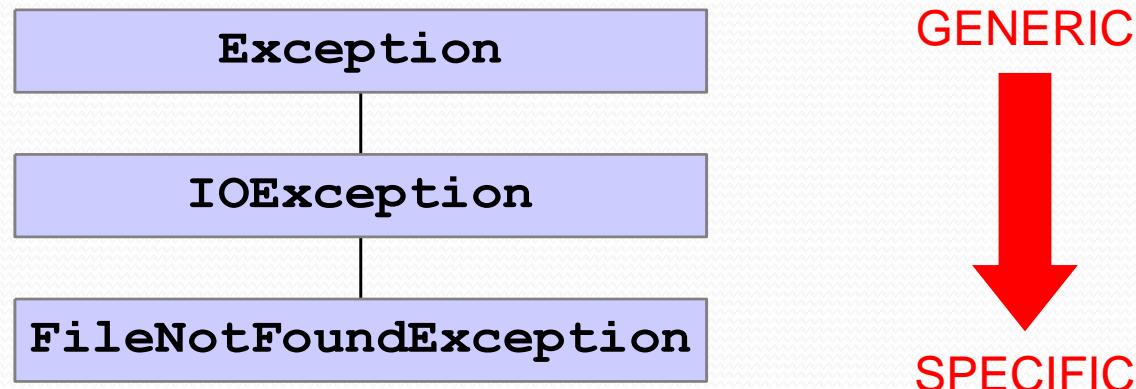
```
import java.io.*;
public class TwistInTaleNestedTryCatch {
    static FileInputStream players, coach;
    public static void main(String args[]) {
        try {
            players = new FileInputStream("players.txt");
            System.out.println("players.txt found");
            try {
                coach.close();
            } catch (IOException e) {
                System.out.println("coach.txt not found");
            }
        } catch (FileNotFoundException fnfe) {
            System.out.println("players.txt not found");
        } catch (NullPointerException ne) {
            System.out.println("NullPointerException");
        }
    }
}
```

Is this allowed?

```
class NoCatchOnlyFinally {  
    public static void main(String args[]) {  
        String name = null;  
        try {  
            System.out.println("Try block : open resource 1");  
            System.out.println("Try block : open resource 2");  
  
            System.out.println("in try : " + name.length());  
            System.out.println("Try block : close resources");  
        } finally {  
            System.out.println("finally : close resources");  
        }  
    }  
}
```

Exception Catching Hierarchy

- When working with multiple **catch** clauses, structure your code to handle the most **specific exceptions** first:



```
try { . . . }
  catch (FileNotFoundException fnfe) { . . . }
  catch (MalformedURLException | SQLException ae) {}
  catch (IOException ioe) { . . . }
  catch (Exception e) { . . . }
finally { . . . }
```

Imagine the file doesn't exist

```
import java.io.*;
public class MultipleExceptions {
    public static void main(String args[]) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("file.txt");
            System.out.println("File Opened");
            fis.read();
            System.out.println("Read File");
        } finally {
            System.out.println("finally");

        } catch (FileNotFoundException fnfe) {
            System.out.println("File not found");
        } catch (IOException ioe) {
            System.out.println("File Closing Exception");
        }
        System.out.println("Next task..");
    }
}
```

Union Exceptions

```
public int getPlayerScore(String playerFile) {  
    try (Scanner contents = new Scanner(new File(playerFile))) {  
        return Integer.parseInt(contents.nextLine());  
    } catch (IOException | NumberFormatException e) {  
        logger.warn("Failed to load score!", e);  
        return 0;  
    }  
}
```

Try With Resources

- Use when a resource needs to be closed
- Behaves like a **finally** with a **close**
- Any resource that implements **java.lang.AutoCloseable** can be used
- Saves having to define the finally block with a check for null and close if not null
- Example:

```
try ( BufferedReader br = new BufferedReader(new FileReader(path)) ) {  
    return br.readLine();  
} catch (IOException e) {  
    logger.severe("Error reading " + path, e);  
}
```

Try With Resources

```
public static void channelsAndBuffers() throws IOException {
    try (RandomAccessFile reader = new RandomAccessFile("C:\\_instaloader\\file.txt", "r")) {
        FileChannel channel = reader.getChannel();
        ByteArrayOutputStream out = new ByteArrayOutputStream() {
            int bufferSize = 1024;
            if (bufferSize > channel.size()) {
                bufferSize = (int) channel.size();
            }
            ByteBuffer buff = ByteBuffer.allocate(bufferSize);
            while (channel.read(buff) > 0) {
                out.write(buff.array(), 0, buff.position());
                buff.clear();
            }
            String fileContent = new String(out.toByteArray(), StandardCharsets.UTF_8);
            System.out.println(fileContent);
        }
    }
}
```

Exception Chaining

- Strategy for maintaining the exception context when rethrowing the exception
- Create new exception passing the original exception
 - Use **exception.getMessage()** to get the wrapped exception
 - **exception.getSuppressed()** added in Java 1.7
 - Allows you to obtain more information about the suppressed exception
 - Useful in **try-with-resources**

Asynchronous Exceptions

- Most exception handling deals with Synchronous Exceptions
- Examples of Asynchronous Exceptions
 - **OutOfMemoryError**
 - **VirtualMachineError**
- How do you handle this?
- Developers of applications don't usually have to deal with these
- JVM developers do
- JNI developers might see these
- Application Server developers will

Be Proactive

- Some libraries don't have good exception handling
- JEE libraries are notorious for throwing **RuntimeExceptions** that you don't have to catch
- That can really wreak havoc with your application
- You aren't supposed to catch **Throwable**, but sometimes you have to
 - When you do, rethrow it, but log it

Best Practices

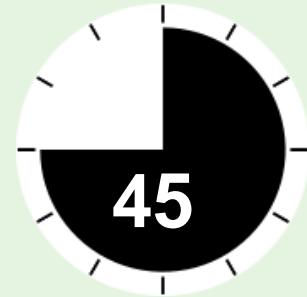
- Never catch an exception and do nothing with it
 - Either log it or rethrow it
- Whenever possible, rethrow a business layer exception
 - Instead of throwing the **SQLException** when querying the DB throw a **CustomerNotFoundException**
- Always catch subclass first
- Throw early, catch late
- Limit logging of exceptions
 - Only log full **stacktrace** of wrapped exceptions *if necessary*
 - Use **log.debug** for printing stacktraces

Summary

- **Throwable** is the superclass for all exceptions and errors
- Exceptions cannot be ignored or swallowed up
- A method's declaration statement should list the possible exceptions that the method might **throw**
- Code that has the possibility of throwing an exception must be in a **try** block if the exception is not in the throws list
- When working with multiple exceptions, **catch** clauses should be structured to handle the most specific exceptions first
- Use **try with resources** when the resource is **AutoCloseable**
- Continue to Lab Introduction on next slide



Lab 3 Introduction



- **Working with Exceptions**
- Experiment with Exception Handling using JDK 7 and 8 syntax
 - Refactor provided code
 - **NumberFormatException**
 - **try-with-resources**
 - Create a custom **Exception** for rethrowing a business layer exception
- After the lab, we will have a lab review
 - If virtual, green check when you are done

Lab 3 Review - Discussion



- **Working with Exceptions**
- Experiment with Exception Handling using JDK 7 and 8 syntax
 - Refactored provided code
 - **NumberFormatException**
 - **try-with-resources**
 - Custom **Exception** for rethrowing a business layer exception
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

Threading and Concurrency

Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- **Threading and Concurrency**
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand the concept of multi-threading in Java

Learn how to create and start a thread

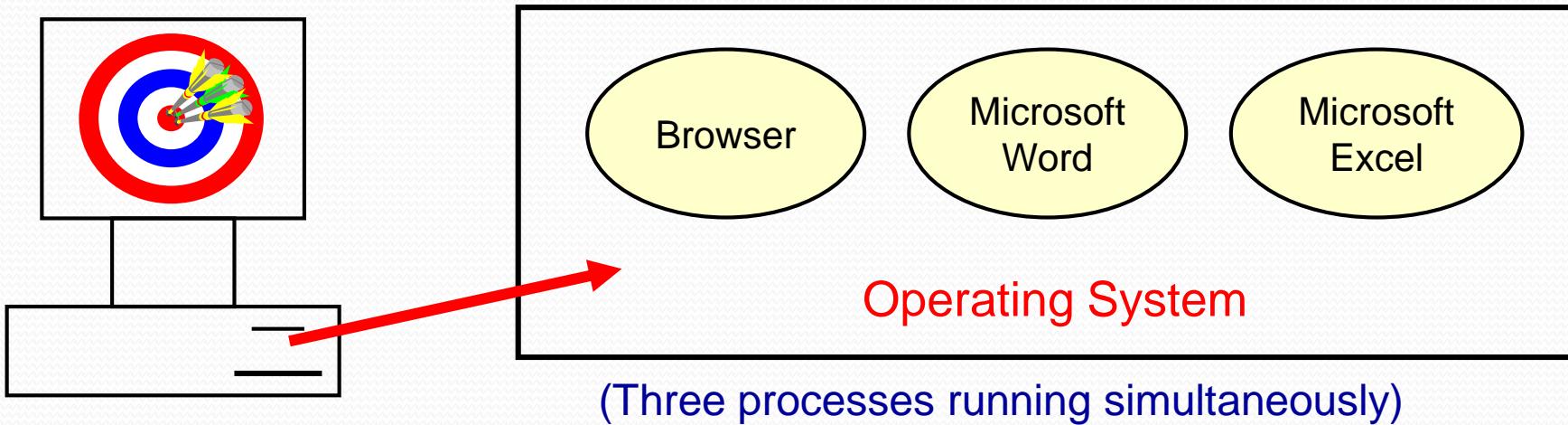
Look at some important thread methods

Review the lifecycle and different states of a thread

Discuss when and when not to use threads

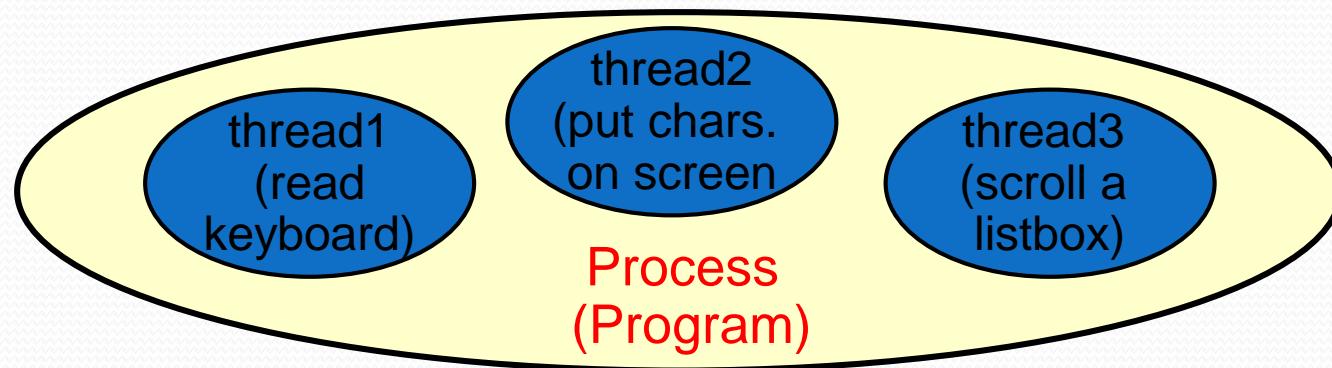
Computer Processes

- Self-contained tasks running on an operating system
- Multi-tasking or multi-processing
 - Processing in more than one executable environment at a time
 - Windows users can start IE and Word at the same time
 - Mac users can run Safari and an editor at the same time



What is a Thread?

- A single, sequential flow of control within a program
 - A single execution path of instructions running within a process
- Multi-threading
 - A program's ability to execute multiple threads simultaneously within the same process



(Three threads running within the same program)

Java Threads

- Two types of threads: User and Daemon
- User threads
 - Programmer creates most of them
 - Example: **main()**
 - Other user threads are often generated from the **main()** thread
- Daemon threads
 - JVM creates most of them
 - Designed as background threads to service other threads
 - Example: Garbage collector thread

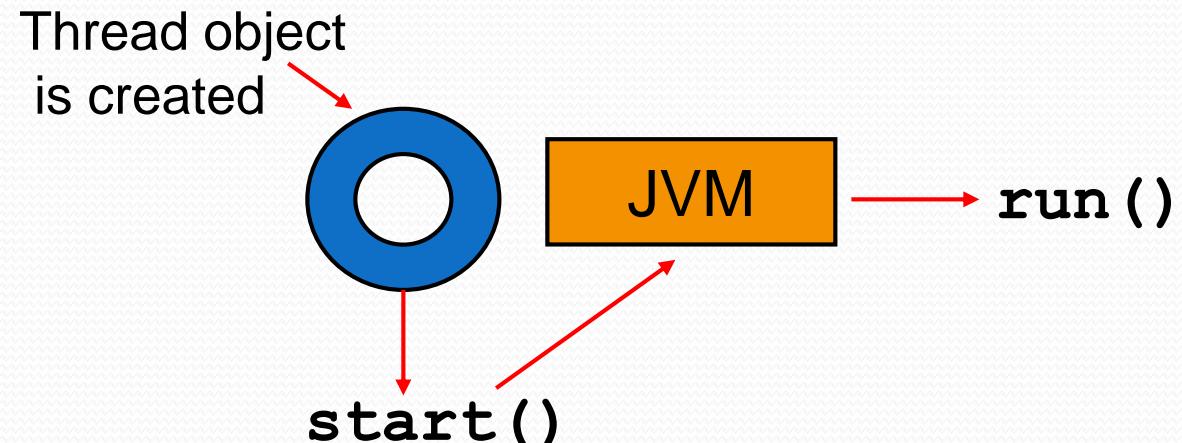
Thread Creation

- Threads must be runnable - Must have a **run()**method
 - **Runnable** interface - Should be implemented by any class whose instances will be executed by a thread
 - Supplies one method to implement:
`public void run();`
- Two ways for a Java class to create a thread:
 - Subclass **java.lang.Thread**
 - Not a viable option if the class must extend some other class
 - **OR**
 - **Thread** class implements the **Runnable** interface
 - Recommended technique

Starting a Thread

Basic steps to starting a thread:

- Create a **Thread** object
 - Pass yourself (**this**) as the target if you want the thread to execute in the current object's **run()** method
- Call the thread's **start()** method
 - **start()** returns immediately
 - Thread execution begins in the runnable object's **run()** method



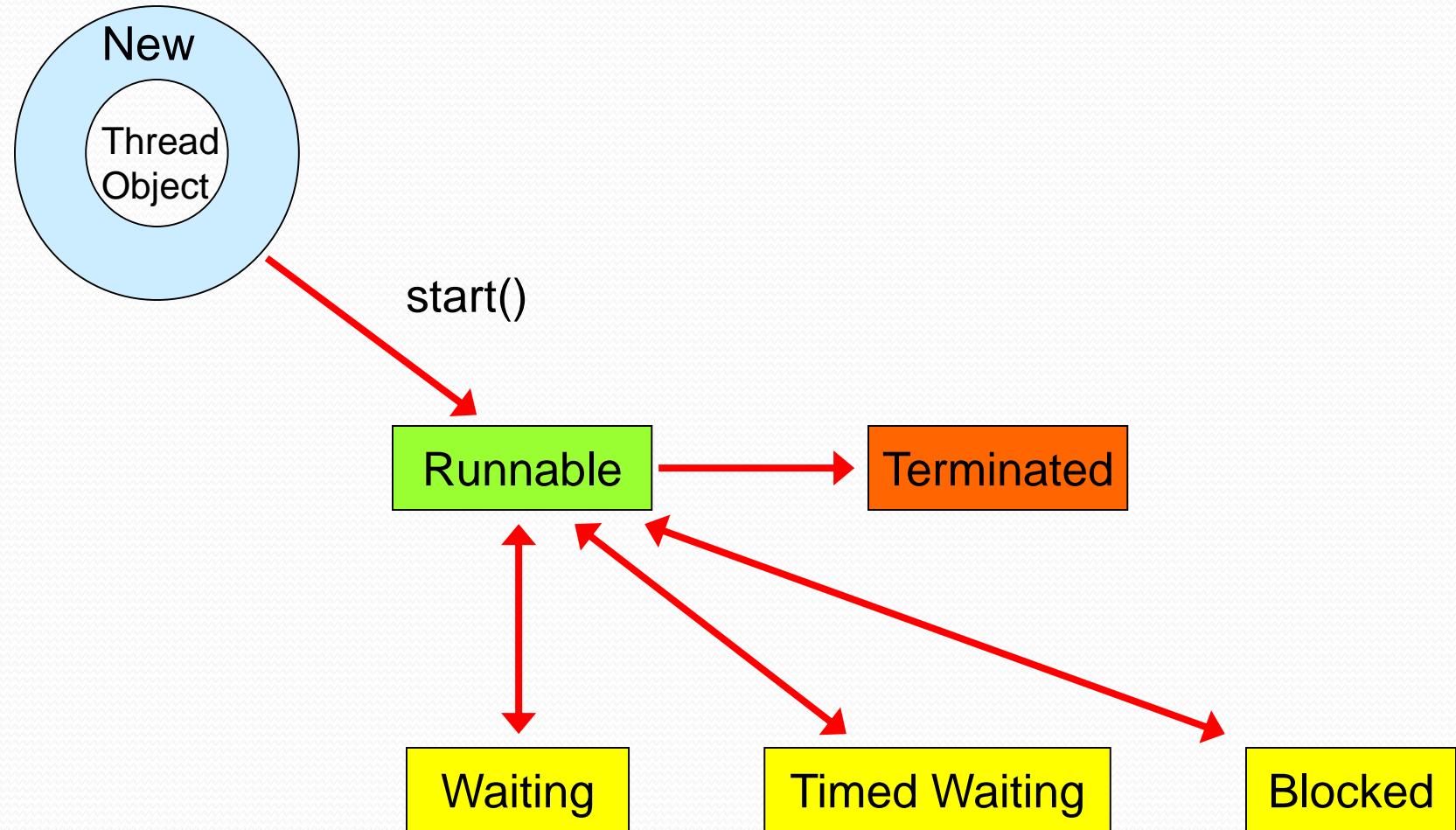
Creating a Simple Thread

```
 . . . // import statements
public class MyThread implements Runnable {
    . . .
public static void main(String[] args){
    . . .
    Thread t = new Thread(obj);
    t.start(); /* t execution will begin in the
object's run() */
}
. . .
public void run() {
    . . . // Code for thread execution
}
}
```

Thread Priority

- Every thread has a priority
 - Higher priority threads are executed in preference to those with lower priority
 - When a new thread is created, its priority is initially set to that of the creating thread
 - Call **setPriority(int newPriority)** to change a thread's priority
- The Thread class defines three constants to represent thread priority:
 - **MAX_PRIORITY**
 - **NORM_PRIORITY**
 - **MIN_PRIORITY**
- Invoke **join()** on a thread to wait for it to terminate

Thread States Diagram



Thread States

- **Thread.State.NEW**
 - After instantiation but BEFORE it is started

```
Thread t = new Thread();
```
- **Thread.State.RUNNABLE**
 - After **t.start()** is called
 - Thread is executed in the JVM, but may be waiting
 - Threads are often in this state
- **Thread.State.BLOCKED**
 - Waiting for a lock
 - Caused by operations such as I/O

Simple Thread Example

```
public class ThreadExample extends Thread {  
    public void run() {  
        while (true){  
            System.out.println("Keep it up ...");  
            try {  
                Thread.sleep(5000);  
            } catch (InterruptedException e) {return;}  
        }  
    }  
    public static void main(String[] args) {  
        ThreadExample t = new ThreadExample();  
        t.start();  
        try {  
            Thread.sleep(25000);  
        } catch (InterruptedException e) { }  
        t.interrupt();  
    }  
}
```

Thread States (continued)

- **Thread.State.WAITING**
 - The thread is waiting indefinitely
 - `wait()` causes thread to go into a waiting state
 - Goes from waiting to dispatchable when another thread calls `notify()` or `notifyAll()`
- **Thread.State.TIMED_WAITING**
 - The thread is waiting for a specific time period
 - `sleep()` causes the thread to cease execution temporarily for the specified amount of time (in milliseconds)
 - Returns to dispatchable when the time expires
- **Thread.State.TERMINATED**
 - The thread has been explicitly stopped
 - Or, `run()` has finished executing

One Thread at a Time

A **synchronized** method:

- Guarantees only one thread at a time executes the method code
- Thread must acquire an object lock prior to execution
- Object lock released upon return from **synchronized** method

An Object's
External Entry Queue

thread1
thread2
thread3
...



Lock

An Object's Internal Wait Queue

synchronized
method1()

synchronized
method2()

Note: Only one thread at a time
can own an object's monitor lock.

Stopping a Thread

- Don't use **stop()**
 - Ugly way to shutdown a thread - What if it is in the middle of a critical operation?
 - Unsafe - Causes the thread to suddenly unlock any object monitors that it has locked
 - Deprecated
- Better alternative
 - Thread terminates when it returns from **run()**
 - Structure code so that **run()** will terminate gracefully
 - Example: Use a **while** loop



Interrupting a Thread

- **public void interrupt()**
 - Invoke interrupt when a thread should stop what it is doing and do something else
 - The program determines how a thread responds to an interrupt
 - Generally, the thread terminates
- Code should support interruption
 - When an **InterruptedException** occurs, catch it and return from the run method
 - Periodically invoke **Thread.interrupted** to respond to interruptions

More Thread Methods

public final void notify()

- "Wakes up" a **single** thread waiting on an object's monitor lock
 - Only one thread is chosen - Choice is arbitrary
 - Thread cannot obtain object lock until current thread releases object lock

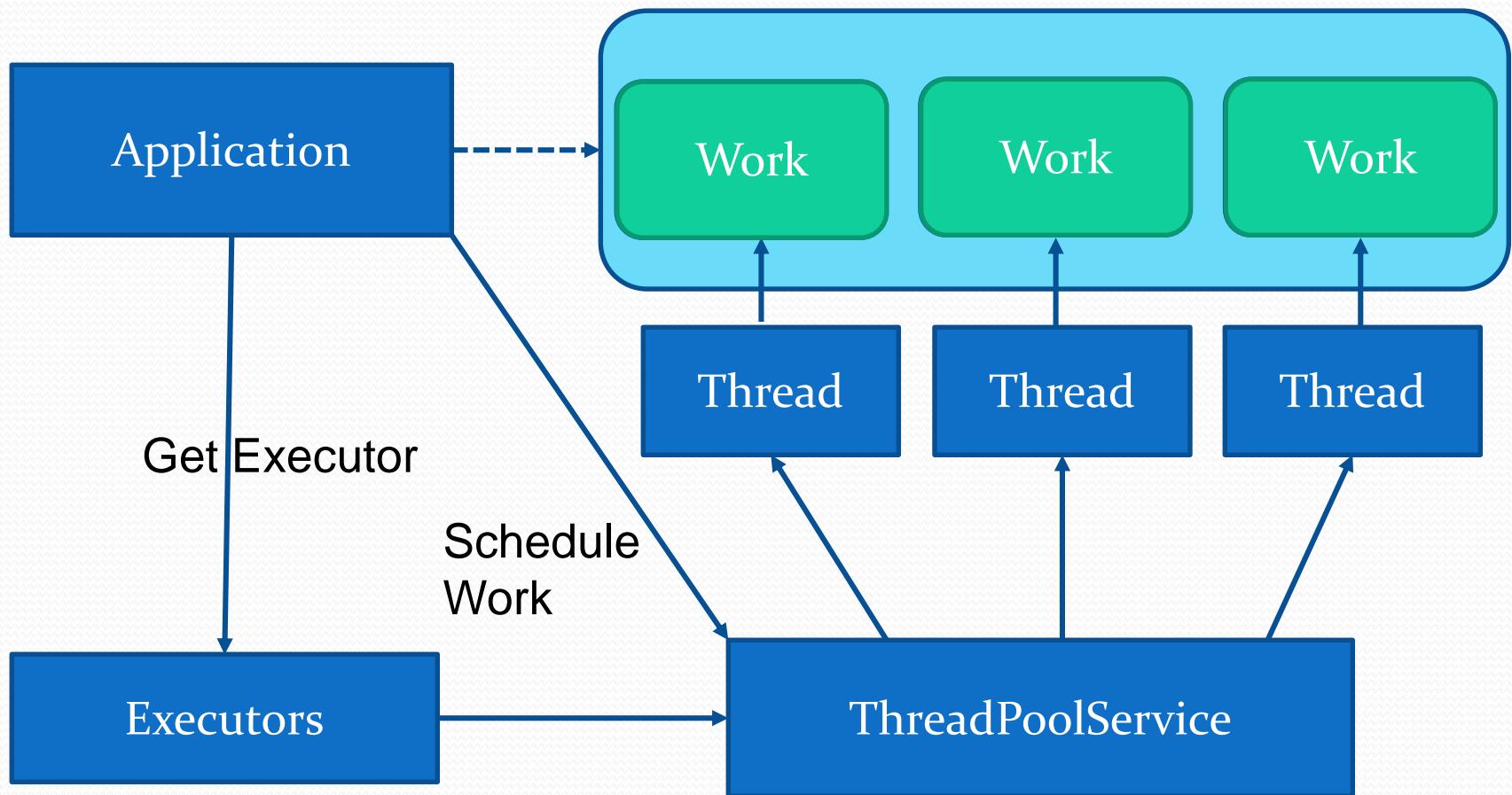
public final void notifyAll()

- Same as **notify()**, except **notifyAll()** wakes up **ALL** the threads waiting on the object's monitor lock
- **wait()** - A thread calling this method waits upon an object's monitor lock
 - Object lock released when thread enters a wait state
 - Can take a time parameter
 - Wait state time can be limited

High-Level Concurrency

- Java 5.0 introduced high-level concurrency features
 - Supports more advanced tasks and large-scale applications
 - Implemented in the new **java.util.concurrent** packages
- Three main interfaces:
 - **Executor** - executes a **Runnable** object
 - **ExecutorService** - extends **Executor**
 - Adds the ability to execute **Callable** objects executor
 - Provides management of executors
 - **ScheduledExecutorService** - extends **ExecutorService** and adds the ability to schedule tasks periodically and/or after a specific delay

Concurrency Overview



Concurrency Examples

- Implementation classes provide functionality
- Use **java.util.concurrent.Executors** static methods to return implementations

```
Executor e = Executors.newThreadPool(5);
```

creates a pool of 5 executors

```
e.execute(...) would use 1 of 5
```

- From

```
Runnable r
```

```
(new Thread(r)).start();
```

- To

```
Runnable r
```

```
e.execute(r);
```

Enhanced Concurrency

- Java 7 and 8 improved concurrency
- **CyclicBarriers** – supports barrier synchronization amongst a static set of threads
- **Phasers** - a reusable, dynamic synchronization barrier with multiple registered parties
- **ForkJoinPool** - allows shared working threads to "steal" work from busy threads
- **RecursiveAction** - forked tasks use this to break up the work into subtasks
- Improved **ConcurrentHashMap** - thread safe and atomicity guaranteed
- **DoubleAdder** and **DoubleAccumulator** - thread safe class to manage a double value
- **LongAdder** and **LongAccumulator** - thread safe class to manage a long value

Phaser

- Register with phaser
- Synchronize - when the last party registered arrives, the phaser advances
 - **arrive** - no waiting - still registered
 - **arriveAndDeregister** - no waiting - no longer registered
 - **arriveAndAwaitAdvance** - waiting - still registered
- Termination
 - **isTerminated** - returns true when terminated
 - Default is true when the number of registered parties becomes zero
- Tiering is used to reduce contention
 - Establish a parent/child relationship between phasers

ForkJoin Library

- Not a replacement for **ExecutorService**
- Adds task breakdown support to an execution context
- A subclass of **RecursiveAction** defines `compute()`
- **RecursiveAction** can contain collection of **RecursiveAction**
- Compute normal flow
 - Fork task into smaller tasks by **actions.add(something)**
 - **invoke(actions)** - process all actions accumulated so far
- **ForkJoinPool** processes the tasks
 - For each **RecursiveAction**, wait for next available Thread
 - Invoke the **compute()** method

ConcurrentHashMap Improved

- Stream support added for JDK 8
- New methods which support multiple **Views** of the same collection as a **Set of Keys**
- New methods accepting functional interfaces from **java.util.Function** package
- New **forEach** methods (key, value and entry)
- New **search** methods (key, value and entry)

Timer

- A **Timer** can schedule tasks for future execution or repeated execution
 - A **TimerTask** can be scheduled for execution by a **Timer**
 - Timer tasks should not take excessive time to complete
- For each **Timer** object there is a single background thread that executes the timer's tasks
- The **Timer** class is thread-safe, which allows multiple threads to share a single Timer object
 - The Java 5 `ScheduledThreadPoolExecutor` is more versatile with regard to time units

Timer Example

```
public static void main(String[] args) {  
    TimerTask task = new TimerTask(){  
        public void run() {  
            System.out.println("Give me five");  
        }  
    };  
    Timer timer = new Timer();  
    timer.schedule(task, 0, 5000);  
    try {  
        Thread.sleep(25000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    timer.cancel();  
}
```

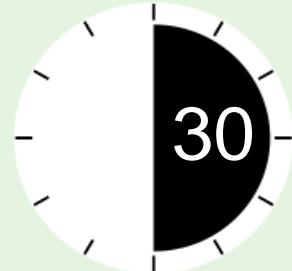
Summary

- A thread is a single, sequential flow of control within a program
- Java supports multi-threading
- A synchronized method guarantees that only one thread at a time can execute the method code
- Avoid the use of deprecated methods



- Continue to Lab Introduction on next slide

Lab 4 Introduction



- **Threads**
- Use threads to invert the colors of an image file and output the image to a new file
 - First, use the old style **Thread**
 - Next, try new way with the **java.util.concurrent** package
 - Then, use the **ThreadPool ExecutorService**
 - Finally, use **ForkJoin** to see how simple it is to create subtasks and allow Java to manage tasks for you
- After the lab, we will have a lab review
 - If virtual, green check when you are done

Lab 4 Review - Discussion



- **Threads**
- Used threads to invert the colors of an image file and output the image to a new file
 - First, old style **Thread**
 - Next, new **java.util.concurrent** package
 - Then, **ThreadPool ExecutorService**
 - Finally, **ForkJoin** to allow Java to manage tasks
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?



Networking

Agenda:

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- **Networking**
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand the concept of networks in Java

Understand stateful vs. stateless network protocols

Learn how to implement a network-aware application

Learn the features of the new network support in JDK 8

What's Important to Programmers?

Application	Application	SMTP, POP, HTTP, FTP, DHCP, LDAP, SSH, TELNET
Presentation		
Session		
Transport	Transport	TCP, UDP, SCTP
Network	Internet	IPv4, IPv6, ICMP
Data Link	Data Link	PPP, Tunneling, NDP, MAC (Ethernet, WIFI), RS-232, ISDN
Physical		
OSI Model	TCP/IP Protocol Stack	

What's Important to Programmers?

Application	Application	SMTP, POP, HTTP, FTP, DHCP, LDAP, SSH, TELNET
Presentation		TCP, UDP, SCTP
Session		IPv4, IPv6, ICMP
Transport	Transport	PPP, Tunneling, NDP, MAC (Ethernet, WIFI), RS-232, ISDN
Network	Internet	
Data Link		
Physical		
OSI Model	TCP/IP Protocol Stack	

Transport Layer Protocols

- **Transmission Control Protocol (TCP)**
- **User (Unreliable?) Datagram Protocol (UDP)**

TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Point-to-Point	Broadcast <i>How can a network connection be connectionless?</i>
Retransmission and windowing	No retransmission or windowing
Segment sequencing	No sequencing
Acknowledgement	No acknowledgement

Socketless Servers (Connectionless?)

- No connection is established
- Data exchanged via **Datagrams**
 - Connected streams are not necessary
- **DatagramPacket** - independent self-contained message
 - Arrival, arrival time and content are not guaranteed
- Packet is built and sent to a host or broadcast
- Packet is received from client
- Normally don't use **InputStream** or **OutputStream**
 - **socket.receive(packet)** waits for a datagram to come in
 - **socket.send(packet)** sends a datagram
 - **socket.close()** when server is done listening

Example - Server Side

```
Socket server = new DatagramSocket( port );
while (true) {
    DatagramPacket packet = new DatagramPacket( buf, buf.length );
    server.receive(packet);
    InetAddress address = packet.getAddress();

    // loop for all data to send
    response = new DatagramPacket(buf, buf.length,
                                  address,packet.getPort());
    server.send(response);
}
```

Example - Client Side

```
Socket socket = new DatagramSocket(host, port);

DatagramPacket packetOut = new DatagramPacket(buf, buf.length);

socket.send(packetOut);

DatagramPacket packetIn = new DatagramPacket(buf, buf.length);

socket.receive(packetIn);

String received =
    new String(packetIn.getData(), 0, packetIn.getLength());
```

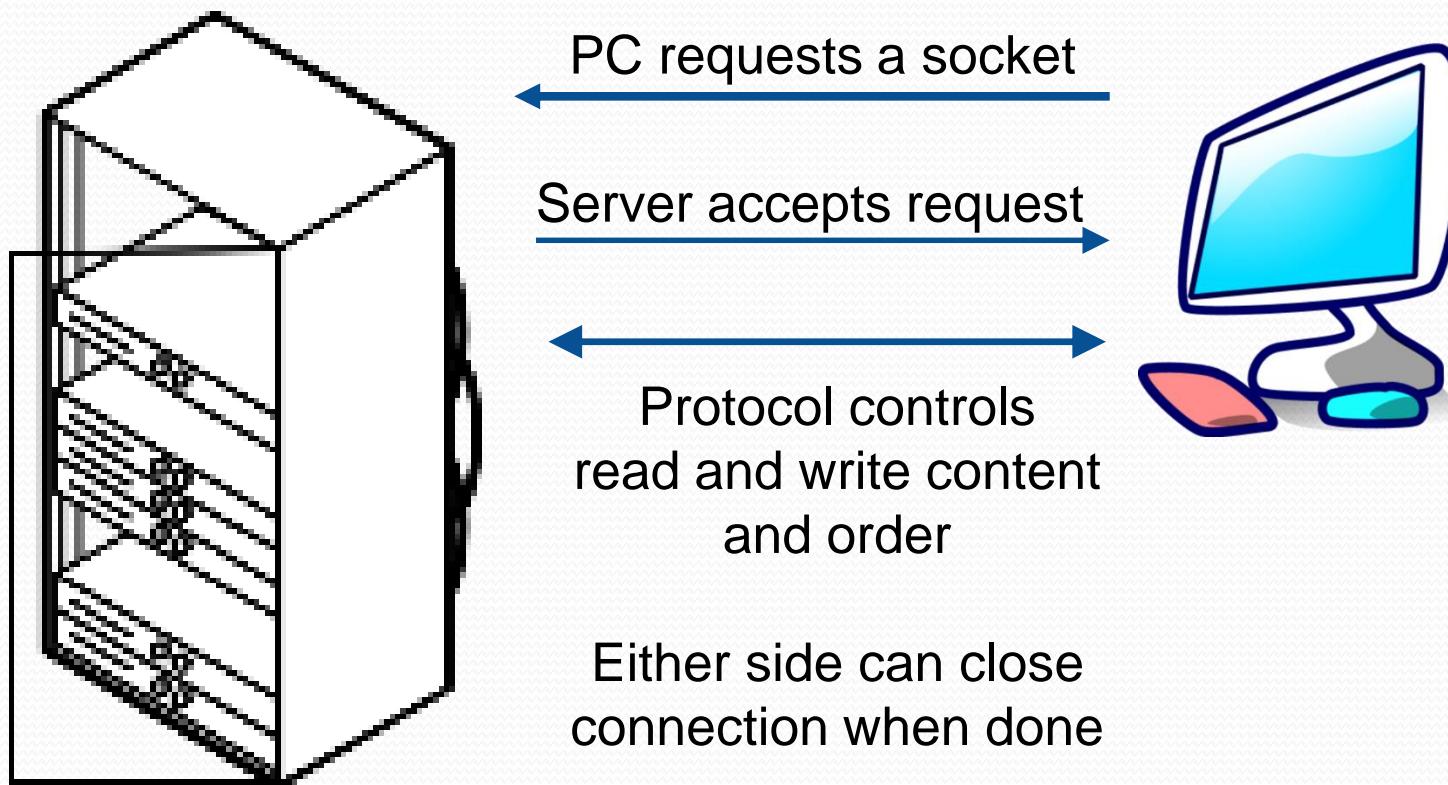
UDP Network Services

- **Domain Name System (DNS)**
- **Network Time Protocol (NTP)**
- IP Tunneling
 - **Layer 2 Tunneling Protocol (L2TP)**
 - **Point-to-Point Protocol (PPP)**
- **Dynamic Host Configuration Protocol (DHCP)**
- Broadcasts to multiple hosts
- Any host can respond if a response is necessary
- Connected UDP is a hybrid - one host to one host

TCP Network Services

- Application Layer to communicate host to host
- **File Transfer Protocol (FTP)**
- **Hypertext Transfer Protocol (HTTP)**
- **Simple Mail Transfer Protocol (SMTP)**
- Use well-known ports like 80, 443, 21, 35, etc.
- Connected
- Guaranteed Delivery

TCP Client/Server Overview



Connection Based Server

- Connections can be short or long duration
 - FTP
 - Long duration connection
 - Multiple requests over the same connection
 - HTTP
 - Short duration connection
 - One request per connection
- Connection is requested by client
- Connection is accepted by server
- Client sends command
- Server usually acknowledges command is valid
- Client sends data if command requires data
- Server responds with data or not as expected

Socket - Client/Server

- Used to exchange information
- Pipe to connect between the client and server
 - Usually bidirectional
- **getInputStream()** the stream to read from
- **getOutputStream()** the stream to write to
- **getChannel()** a connected or unconnected channel to a host on a port
- **close()** terminates the connection and connection cannot be reused

Example - Client Side

```
try {  
    Socket aSocket = new Socket(host, port);  
    PrintWriter out = new  
        PrintWriter(socket.getOutputStream(), true);  
    BufferedReader in = new BufferedReader(new  
        InputStreamReader(socket.getInputStream()));  
    String inputLine = null;  
    while ((inputLine= in.readLine()) != null) {  
        out.println(inputLine);  
    }  
} catch...
```

Example - Server Side

```
try {  
    ServerSocket server = new ServerSocket(80);  
    while (true) {  
        Socket client = server.accept();  
        PrintWriter out = new  
            PrintWriter(socket.getOutputStream(), true);  
        BufferedWriter out = new BufferedWriter(new  
            OutputStreamWriter(socket.getOutputStream()));  
        String[] content = {};  
        for(int i = 0; i < content.length, i++) {  
            out.println(content[i]);  
        }  
    }  
    client.close();  
} catch...
```

Java 7 NIO (New I/O)

- Java NIO is an alternative Java I/O API
 - Encourages non-blocking IO
- Incorporates buffers instead of streams
 - Buffered data is more easily traversable
 - Buffers do not block threads like Streams do
- Buffers and channels are fundamental NIO concepts
 - Data is read from a channel into a buffer
 - Data is written from a buffer to a channel
 - A single thread can manage & monitor multiple channels
- Many different types of channels exist:
 - SocketChannel, FileChannel, DatagramChannel

Java 7 NIO (New I/O)

```
public static void channelsAndBuffers() throws IOException {
    try (RandomAccessFile reader = new RandomAccessFile("C:\\\\instaloader\\\\file.txt", "r")) {
        FileChannel channel = reader.getChannel();

        ByteArrayOutputStream out = new ByteArrayOutputStream() {
            int bufferSize = 1024;
            if (bufferSize > channel.size()) {
                bufferSize = (int) channel.size();
            }

            ByteBuffer buff = ByteBuffer.allocate(bufferSize);

            while (channel.read(buff) > 0) {
                out.write(buff.array(), 0, buff.position());
                buff.clear();
            }

            String fileContent = new String(out.toByteArray(), StandardCharsets.UTF_8);
            System.out.println(fileContent);
        }
    }
}
```

Java 7 NIO (New I/O)

```
public static void channelsAndBuffers() throws IOException {
    try (RandomAccessFile reader = new RandomAccessFile("C:\\\\instaloader\\\\file.txt", "r");
        FileChannel channel = reader.getChannel();) {  
      
    Channel  
  
        ByteArrayOutputStream out = new ByteArrayOutputStream(); {  
  
            int bufferSize = 1024;
            if (bufferSize > channel.size()) {
                bufferSize = (int) channel.size();
            }
              
            Buffer  
            ByteBuffer buff = ByteBuffer.allocate(bufferSize);  
  
            while (channel.read(buff) > 0) {
                out.write(buff.array(), 0, buff.position());
                buff.clear();
            }
            String fileContent = new String(out.toByteArray(), StandardCharsets.UTF_8);
            System.out.println(fileContent);
        }
    }
}
```

NIO ByteBuffers

- **ByteBuffer** is used to buffer input and output for channel IO
 - **get** - getters for various data types
 - **put** - adders of various types
 - **flip** - change from adding mode to read for channel use
 - **array** - get the contents as a byte array

```
int bufferSize = 1024;
if (bufferSize > channel.size()) {
    bufferSize = (int) channel.size();
}

ByteBuffer buff = ByteBuffer.allocate(bufferSize);

while (channel.read(buff) > 0) {
    out.write(buff.array(), 0, buff.position());
    buff.clear();
}
```

NIO Sockets and Channels

- **ServerSocketChannel**
 - **open** - create a channel
 - **channel.bind** - bind the channel to a port
 - **channel.accept** - open a channel to a client
 - **channel.read(aByteBuffer)** - read the data from a client into a buffer
- **SocketChannel**
 - **open** - returns an open channel to communicate over
 - **channel.connect(host, port)** - opens a channel to a listening server on host, port
 - **channel.write(aByteBuffer)** - needs to be in a loop until *hasRemaining* is false

NIO Sockets and SocketServers

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.socket().bind(new InetSocketAddress(8888));

while(true){

    SocketChannel socketChannel = serverSocketChannel.accept();
    //do something with socketChannel...

}
```

NIO Channels

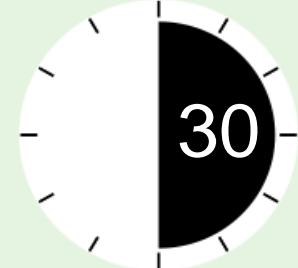
- **DatagramChannel**
 - **open()** - open a connection to a port
 - **socket()** - returns a *DatagramSocket*
 - **socket.bind(port)** - binds the channel to a specific port
 - **channel.receive(aByteBuffer)** - accepts a packet
- **AsynchronousChannel** interface
 - Supports Asynchronous I/O
 - Usually takes a **Future<> operation(...)**
 - Or **void operation(A attachment, CompletionHandler)**
 - Can be cancelled
 - **AsynchronousFileChannel** operates on files
 - **AsynchronousSocketChannel** operates on sockets
 - **AsynchronousServerSocketChannel** for server sockets

Summary

- Sockets are central to Java networking
- Sockets represent the input and output context for communication over network topologies
- **ServerSockets** *listen* for connections
- Sockets *establish* connections
- **DatagramPackets** are useful for protocols that don't need guaranteed delivery
- **Continue to Lab Introduction on next slide**



Lab 5 Introduction



- **Networking**
- You will create a server application that listens for a text string sent by a client. You will also create the client that sends the text string to the server.
 - Demonstrate how easy it is to create a networking application
 - **SocketChannel** and **ServerSocketChannel**
 - **try-with-resources**
 - **ByteBuffer** and **BufferedReader**
- After the lab, we will have a lab review
 - If virtual, green check when done

Lab 5 Review - Discussion



- **Networking**
- Client/Server Application listening for input
 - **SocketChannel** and **ServerSocketChannel**
 - **try-with-resources**
 - **ByteBuffer** and **BufferedReader**
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?



This slide has been intentionally left blank.

Advanced JDBC

Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- **Advanced JDBC**
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand how to access and store content with databases in Java

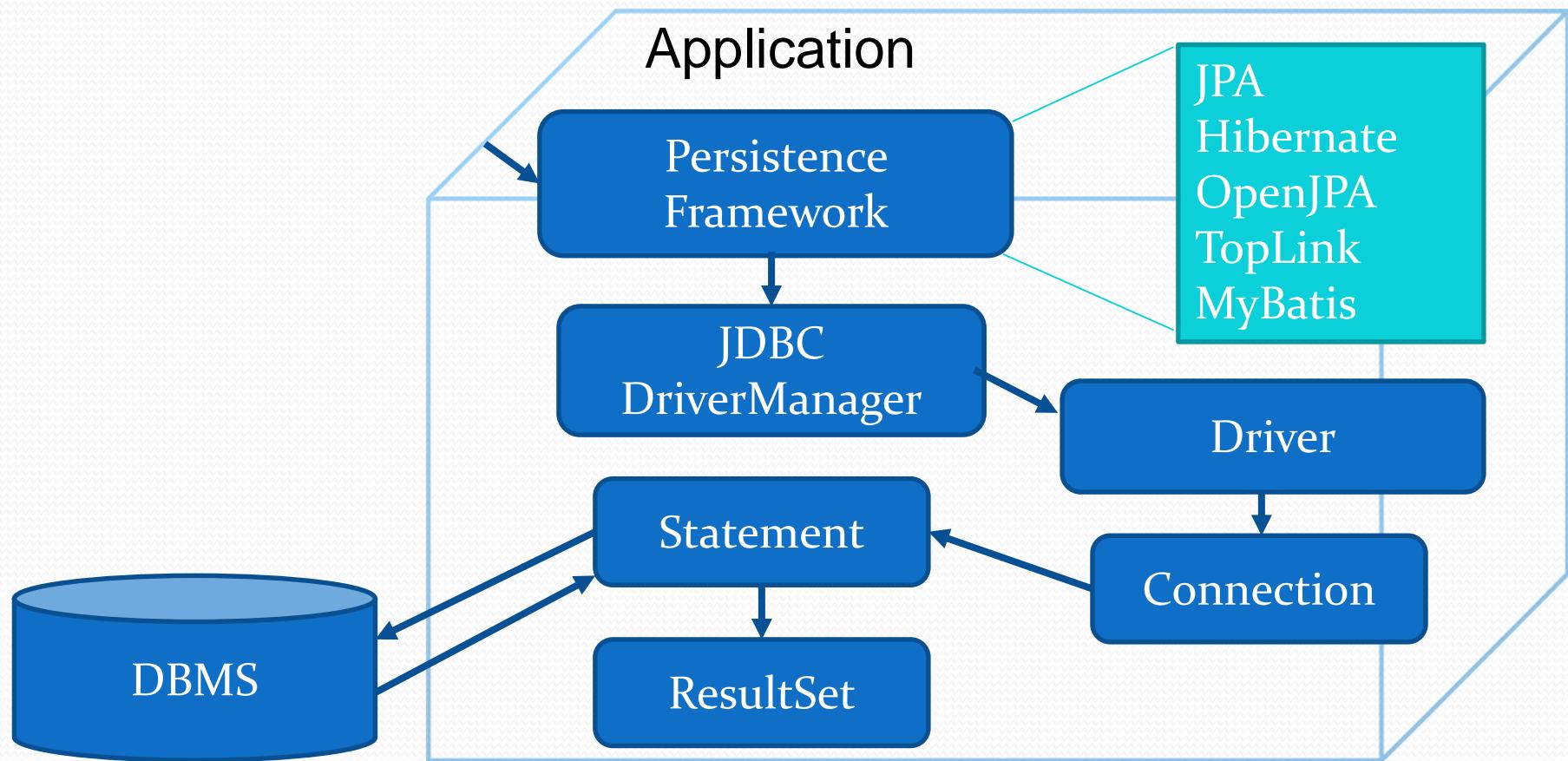
Explore the structure of the Java Data Base Connectivity library

Learn how to make connections to databases

Learn how to create and populate databases

Learn about the new features of JDBC with JDK 8

JDBC Overview



Recent JDBC Enhancements JDK7

- Support of **try-with-resources** statement on **Connection**, **Statement** and **ResultSet**
- **RowSetFactory** interface and **RowSetProvider** class introduced

Recent JDBC Enhancements JDK8

- Addition of **REF_CURSOR** support
- Addition of **java.sql.DriverAction** Interface
- Addition of security check on **deregisterDriver** method in **DriverManager** class
- Addition of the **java.sql.SQLType** Interface
- Addition of the **java.sql.JDBCType** Enum
- Add support for large update counts
- Changes to the existing interfaces
- Rowset 1.2: Lists the enhancements for JDBC **RowSet**

Working with ResultSet

Supports scrollable results sets with update capability

Supports scroll-sensitive or scroll-insensitive attribute

ResultSetMetaData - attributes of data returned

Normal Usage of ResultSet

```
PreparedStatement pstmt =  
    aConnection.prepareStatement("select  
        col1, col2 from atable where col1 = ?");  
pstmt.setString(1, "AVALUE");  
ResultSet prs = pstmt.executeQuery();  
Statement stmt = aConnection.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT col1, col2  
    FROM ATABLE where col1 = 'Avalue'");  
  
// ResultSet is not updatable and is not scrollable
```

Scrollable ResultSet Example

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT col1, col2
    FROM ATABLE");
/* rs will be scrollable, will not show changes made
   by others, and will be updatable */
```

Working with RowSet

- RowSets are JavaBean ResultSets
 - Have properties - getters and setters
 - Fire events to listeners
- Allows scrolling and updating a RowSet even if the DBMS doesn't support it
- Can be connected or disconnected

Types of RowSet Interfaces

- **JdbcRowSet** - only connected RowSet and most like ResultSet
 - **CachedRowSet** - can connect to database, read ResultSet, reconnect and write updates, and detect conflicts
 - **WebRowSet** - like CachedRowSet but supports XML output
 - **JoinRowSet** - adds join capabilities to the CachedRowSet
 - **FilteredRowSet** - adds the ability to filter without a query where clause

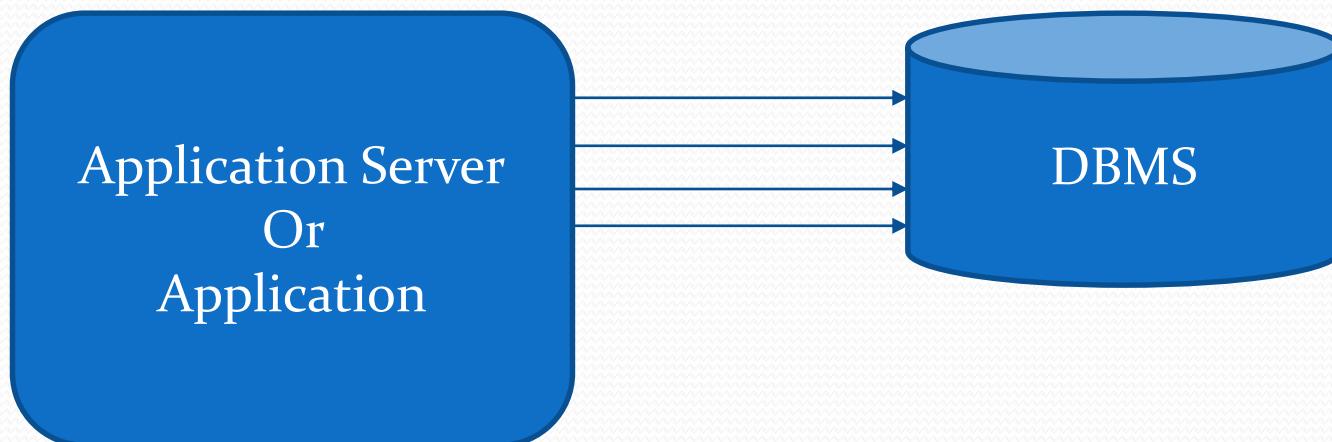
FilteredRowSet Example

```
FilteredRowSet frs = new FilteredRowSetImpl();
frs.setCommand("SELECT * FROM COFFEE_HOUSES");
frs.setUsername(settings.userName);
frs.setPassword(settings.password);
frs.setUrl(settings.urlString);
MyFilter myFilter = new MyFilter(10000, 10999, 1);
frs.setFilter(myFilter);

/* MyFilter extends java.sqlx.rowset.Predicate and
evaluates the conditions */
```

Connection Pooling

- Connections are expensive and limited by DBMS
- Many applications need data but not all at the same time
- Applications usually need connection for a short time
- Application get a connection from the pool
- Frees the connection when done



Resource Management

- **try-with-resources** automatically manages resources

```
try(Connection con = getConnection()) {  
    try (PreparedStatement prep =  
        con.prepareStatement("Update ...")) {  
        ... // do something with the PreparedStatement  
        con.commit();  
    } catch (SQLException e) {  
        //something failed  
        con.rollback();  
        throw new ApplicationException();  
    }  
}
```

Security Concerns

Connection string and authentication

User input: SQL-Injection

Transport security

Applications have to enforce some security

Advanced Types

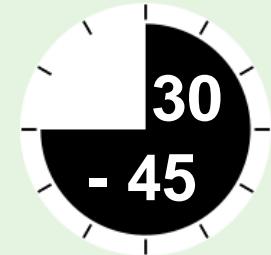
- **BLOB:** Blob interface
- **CLOB:** Clob interface
- **NCLOB:** NClob interface
- **ARRAY:** Array interface
- **XML:** SQLXML interface
- Structured types: Struct interface
- **REF**(structured type): Ref interface
- **ROWID:** RowId interface
- **DISTINCT:** Type to which the base type is mapped.
 - For example, a DISTINCT value based on a SQL NUMERIC type maps to a `java.math.BigDecimal` type because NUMERIC maps to `BigDecimal` in Java
- **DATALINK:** `java.net.URL` object

Summary

- JDBC provides for connections to databases
- Not all JDBC drivers support all features of the API
- JDBC 4 introduced the **ScrollableResultSet** which improves performance when large datasets are being read
- **ResultSetMetaData** and **DatabaseMetaData** allow querying the attributes
- Advanced Data Types are rarely used except BLOB and CLOB
- JDBC is becoming less frequently used directly as persistence frameworks are more common
- **Continue to Lab Introduction on next slide**

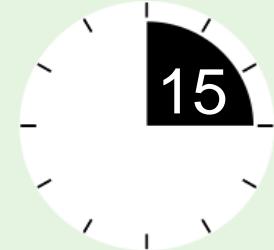


Lab 6 Introduction



- JDBC
- Create a database of books, populate the database and query the content of the database
- Although snippets are provided, use only if you get stuck!
 - Log **SQLException**
 - **SQL Statements** and **execute()** to create a table
 - **PreparedStatement** with ? value substitution to **insert** a book
 - **executeQuery()** to retrieve **ResultSet** of books
 - **FilteredRowSet** to insert 2 more books
 - Implement **Predicate** in an **AuthorFilter** class to search for a book by author
- After the lab, we will have a lab review
 - If virtual, green check when you are done

Lab 6 Review - Discussion



- JDBC
- Did you use the snippets or blaze your own trail?
 - Log **SQLException**
 - **SQL Statements** and **execute()** to create a table
 - **PreparedStatement** with ? value substitution to **insert** book
 - **executeQuery()** to retrieve **ResultSet** of books
 - **FilteredRowSet** to insert 2 more books
 - Implement **Predicate** in an **AuthorFilter** class to search for a book by author
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

Performance

Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- **Performance**
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Understand the many factors that affect performance in Java

Identify some causes of performance problems and solutions for them

Learn about design principles that can influence performance

Process and Issues - Top 10

1. Poor capacity planning
2. Poor infrastructure middleware match
3. Excessive JVM garbage collection
4. Poor integration with external systems
5. Poor SQL tuning
6. Application-specific performance problems
7. Poor middleware tuning
8. Lack of proactive monitoring
9. Saturated hardware
10. Network latency problems

Which issues affect your applications?

What is your purpose?

- When you start gathering metrics, know your purpose:
 - Troubleshooting?
 - Monitoring?
 - Optimization?
 - Tuning?
- Your purpose will impact how you use the tool

Performance Problem?

- Do you really have a performance problem?
- What is your system capable of?
 - What are its limits?
 - How do similar systems perform?
 - What type of request-response cycles do you handle?

TCP Benchmarks

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
1					TPC-E BENCHMARK RESULTS															
2	These results are valid as of date 6/3/2020 12:06:36 PM																			
3																				
4	TPC-E Results - Revision 1.X - Part 1: Active Results																			
5																				
6	Result ID	Short ID	Company	System	Spec. Rev	TpsE	Price/Perf	Total Sys.	Currency	Database	Operating	Server CPU Type	Processors	Cores	Threads	# Front En	Total FE Pr	Total FE C	Total FE Th	
7	1.19E+08	4086	Fujitsu	Fujitsu Server PRIMERGY RX2540 M1	1.14.0	6844.2	85.13	582623	USD	Microsoft	Microsoft	Intel Xeon Platinum 8280 2.70GHz	2	56	112	1	2	56	112	
8	1.18E+08	4082	Fujitsu	FUJITSU Server PRIMERGY RX2540 N1	1.14.0	6606.75	92.85	613391	USD	Microsoft	Microsoft	Intel Xeon Platinum 8180 2.50GHz	2	56	112	1	2	56	112	
9	1.19E+08	4085	Lenovo	Lenovo ThinkSystem SR655	1.14.0	6716.88	99.99	671576	USD	Microsoft	Microsoft	AMD EPYC 7742 2.25GHz	1	64	128	1	2	48	96	
10	1.19E+08	4084	Lenovo	Lenovo ThinkSystem SR650	1.14.0	7012.53	90.99	638052	USD	Microsoft	Microsoft	Intel Xeon Platinum 8280 2.70GHz	2	56	112	1	2	48	96	
11	1.18E+08	4083	Lenovo	Lenovo ThinkSystem SR650	1.14.0	6779.53	92.49	626980	USD	Microsoft	Microsoft	Intel Xeon Platinum 8180 2.50GHz	2	56	112	1	2	44	88	
12	1.17E+08	4081	Lenovo	Lenovo ThinkSystem SR950	1.14.0	11357.28	98.83	1122410	USD	Microsoft	Microsoft	Intel Xeon Platinum 8180 2.50GHz	4	112	224	1	2	44	88	
13	1.17E+08	4080	Lenovo	Lenovo ThinkSystem SR650	1.14.0	6598.36	93.48	616777	USD	Microsoft	Microsoft	Intel Xeon Platinum 8180 2.50GHz	2	56	112	1	2	44	88	
14																				
15																				
16	TPC-E Results - Revision 1.X - Part 2: Historical Results																			
17																				
18	Result ID	Short ID	Company	System	Spec. Rev	TpsE	Price/Perf	Total Sys.	Currency	Database	Operating	Server CPU Type	Total Server	Total Serv	Total Serv	# Front En	Total FE Pr	Total FE C	Total FE Th	
19	1.1E+08	4040	Dell	Dell PowerEdge T710	1.10.0	1074.14	264.32	283914	USD	Microsoft	Microsoft	Intel Xeon X5680 Hex-Core - 3.33 GHz	2	12	24	2	1	4	8	
20	1.1E+08	4034	Dell	Dell PowerEdge R910	1.9.0	1933.96	297.44	575235	USD	Microsoft	Microsoft	Intel Xeon X7560 8-Core - 2.27 GHz	4	32	64	3	2	8	8	
21	1.09E+08	4026	Dell	Dell PowerEdge T610	1.7.0	766.47	273.65	209741	USD	Microsoft	Microsoft	Intel Xeon X5570 Quad-Core - 2.93 GHz	2	8	16	2	2	8	8	
22	1.09E+08	4022	Dell	Dell PowerEdge R905	1.7.0	635.43	403.87	256625	USD	Microsoft	Microsoft	AMD Opteron 8384 - 2.70 GHz	4	16	16	4	2	8	8	
23	1.08E+08	4014	Dell	Dell PowerEdge R900	1.5.1	671.35	500.55	336039	USD	Microsoft	Microsoft	Intel Xeon X7460 Hex-Core - 2.67 GHz	4	24	24	2	2	8	8	
24	1.08E+08	4013	Dell	Dell PowerEdge R900		1.3	451.29	734.25	331357	USD	Microsoft	Microsoft	Intel Xeon X7350 Quad-Core - 2.93 GHz	4	16	16	4	2	8	8
25	1.08E+08	4009	Dell	Dell PowerEdge 2900 III		1.5.0	295.27	694.08	204940	USD	Microsoft	Microsoft	Intel Xeon X5460 Quad-Core - 3.16 GHz	2	8	8	2	2	2	4

Set Targets

- Set realistic performance targets
 - 500 transaction per second
 - Page load times of less than 2 seconds
 - Fail over in less than 5 seconds
 - 99% of transactions happen in less than 40ms
 - Stateless response times of 50ms
 - Stateful response times of 500ms

Latency
Throughput
Responsiveness
Stability
Jitters
What else?

There is always a bottleneck

- Performance is only a problem when you don't meet your targets
- There is always a bottleneck, so long as there is load
 - Solve the CPU issue? Then it's memory
 - Solve the memory issue? Then it's network I/O
 - Solve the network I/O problem? Then it's the processor

The Usual Suspects

- Java performance culprits are well known
 - So are the solutions
- Java performance problems reveal themselves in four ways:
 - CPU overutilization
 - Memory management issues
 - File, network or database I/O operations
 - Threading problems

Metrics to Follow

- Set realistic performance targets
 - CPU utilization
 - System context switching
 - Physical memory utilization
 - Heap consumption over time
 - Network bandwidth used
 - Disk IO latency
 - Database locks

Database Issues

- The database is a common bottleneck
 - Tune your connection pools
 - Monitor the connection wait time
 - Map your ORM closely to the DB
 - Look for the queries that are slowest to perform
 - Cache queries performed repeatedly
 - Use parameterized queries rather than custom SQL

CPU & Memory Issues

- Memory issues manifest themselves in several ways
 - Resource leaks
 - Memory leaks
 - Object allocation
 - Memory churn
- Both memory and CPU issues are attributable to poor programming practices

Practices to Avoid

- Coarse grained communication is best
 - Don't create 'chatty' apps
- Architect the ability to cache data in your apps
- Make it possible to parallelize complex ops
- Keep transactions as short as possible
- Paginate large database requests
- Avoid locking shared resources

Java Programming Tips

- Use the right collection class
- Don't use recursion
- Close all resources
- Avoid autoboxing

Code Optimization

- Compiler does some optimizations
 - Dead code removal
 - Code "in-lining"
- The compiler is smarter than you
 - Many efforts to optimize code just replicate what the compiler already does
 - Not all optimization recommendations remain valid from one JVM release to the next
- Use static code analyzers like **PMD** or **FindBugs** and clean up the code based on the recommendations

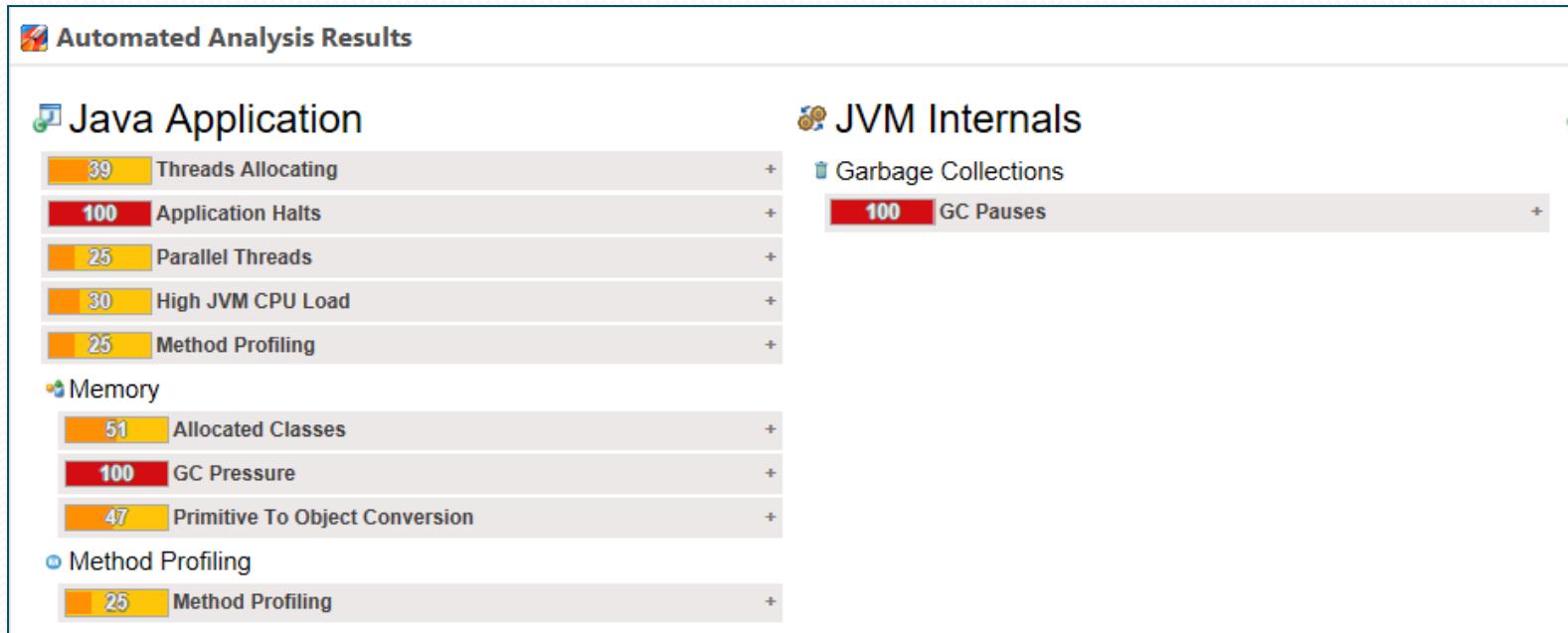
What is Java Flight Recorder?

- Java profiling tool embedded inside the JVM
- Used to capture runtime metrics
 - Threads, memory, processor usage etc
- Designed to run constantly in the background
 - Akin to an airplane's black box
- Provides great utility with a minimal impact

What is Java Mission Control?

- Java Flight Recorder writes its data in binary format
 - Not readable by humans
- Java Mission Control visually presents JFR data
 - Chart
 - Tables
 - Histograms
 - Flame Graphs

Together, JDK Flight Recorder and Java Mission Control produce an incredibly powerful mechanism for profiling, troubleshooting, performance tuning and optimizing your JVM.



Live Performance Dashboard

JDK Mission Control

File Edit Navigate Window Help

[11.0.9] spock-lizard-1.0.jar (15064)

Overview

JMX Data Persistence Settings

Dashboard

- Used Java Heap Memory: Now: 191 MiB Max: 194 MiB
- JVM CPU Usage: Now: 0.393 % Max: 10.3 %
- Live Set + Fragmentation: Now: 17.1 % Max: 17.1 %

Processor

Processor usage over time:

- 9:26:51 AM: JVM CPU Usage (purple), Machine CPU Usage (blue)
- 9:27:05 AM: JVM CPU Usage (purple), Machine CPU Usage (blue)
- 9:27:19 AM: JVM CPU Usage (purple), Machine CPU Usage (blue)
- 9:27:33 AM: JVM CPU Usage (purple), Machine CPU Usage (blue)
- 9:27:47 AM: JVM CPU Usage (purple), Machine CPU Usage (blue)

Processor usage legend:
 JVM CPU Usage
 Machine CPU Usage

Memory

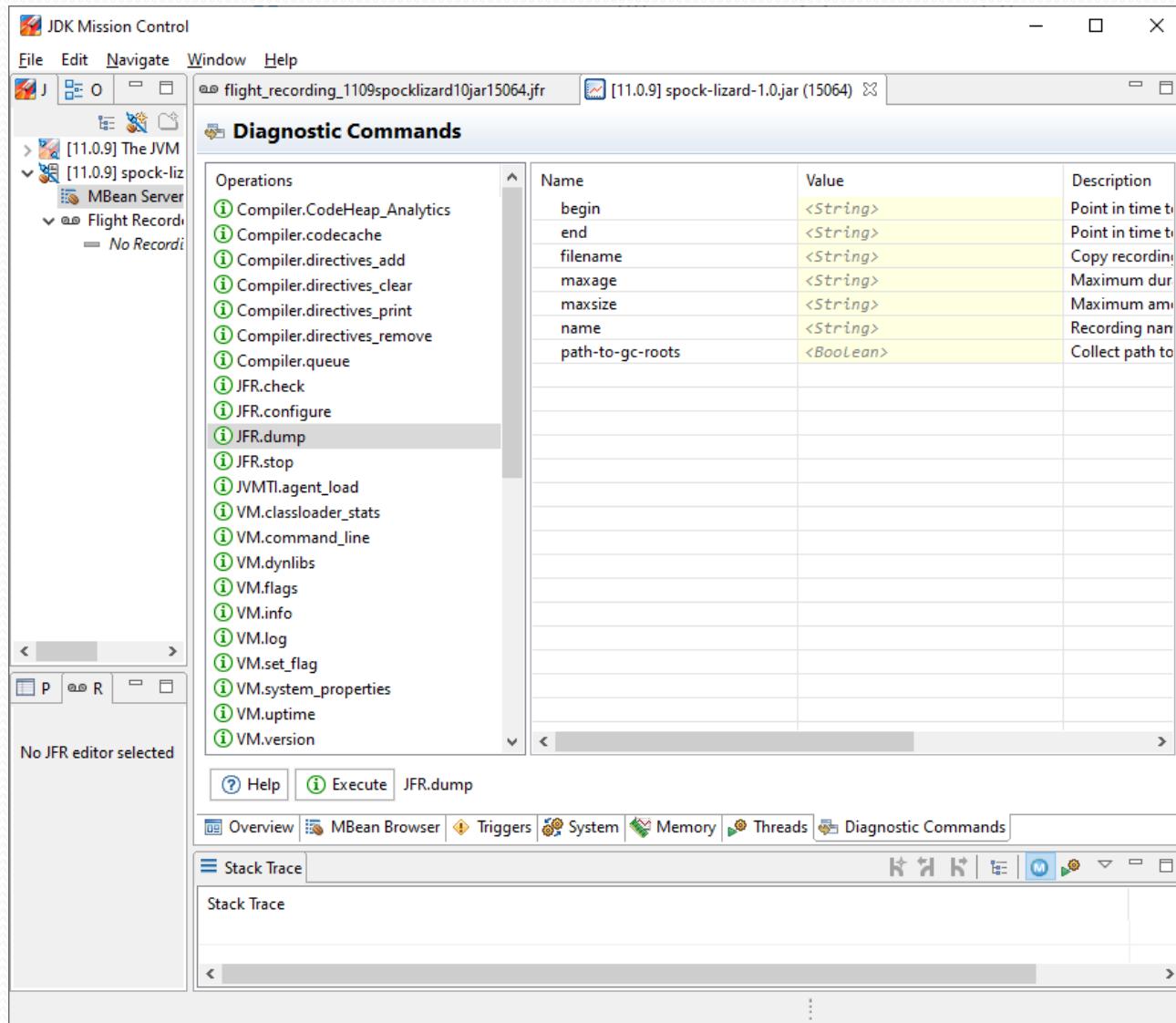
Committed Java Heap: 128 MiB

Overview MBean Browser Triggers System Memory Threads Diagnostic Commands

Stack Trace

Stack Trace

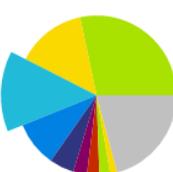
Diagnostic Command Access



JOverflow Heap Analyzer

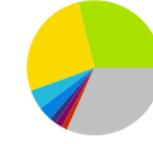
JOverflow

Object Selection	Memory KB	Overhead KB	Objects
All Objects	13,696.60 (100%)	0.00 (0%)	253,356
Boxed Collections	4,074.88 (30%)	5,752.00 (42%)	1,297
Small Collections	913.47 (7%)	712.86 (5%)	4,607
Sparse Small Collections	356.13 (3%)	69.39 (1%)	1,768
Duplicate Strings	345.47 (3%)	241.06 (2%)	5,467
Sparse Large Collections	241.96 (2%)	27.00 (0%)	79
Empty Arrays	137.38 (1%)	137.38 (1%)	1,756
Arrays with One Element	134.23 (1%)	134.19 (1%)	5,727
Sparse Arrays	83.62 (1%)	58.16 (0%)	834
Duplicate Arrays	51.92 (0%)	36.02 (0%)	621
Arrays with Underused Elements	43.31 (0%)	20.82 (0%)	472
Long Zero Tail Arrays	41.78 (0%)	40.39 (0%)	21
Empty Unused Collections	38.88 (0%)	38.88 (0%)	478
Zero Size Arrays	36.89 (0%)	36.89 (0%)	2,361
Vertical Bar Arrays	5.23 (0%)	11.18 (0%)	13



Referrer

Referrer	Memory KiB	Overhead KiB	Objects
{HashMap}	3,807.90 (28%)	0.00 (0%)	159,948
com.mcnz.jfr.jmc.gc.MikeTyson.map	3,637.63 (27%)	0.00 (0%)	8
javax.management.openmbean.CompositeData\$CompositeData	632.24 (5%)	0.00 (0%)	2,541
Object[]	463.13 (3%)	0.00 (0%)	7,647
java.lang.reflect.Method[]	218.25 (2%)	0.00 (0%)	1,552
Unknown GC root	211.74 (2%)	0.00 (0%)	2,688
java.lang.invoke.LambdaForm\$Name[]	193.31 (1%)	0.00 (0%)	4,124
{HashMap}	133.47 (1%)	0.00 (0%)	2,390
java.lang.invoke.MethodType\$ConcurrentWeaver\$MethodType	102.54 (1%)	0.00 (0%)	1
javax.management.openmbean.TabularData\$Table\$Table	99.86 (1%)	0.00 (0%)	154
byte[]	98.52 (1%)	0.00 (0%)	388
TreeMap	94.90 (1%)	0.00 (0%)	3,973
jdk.internal.loader.BuiltinClassLoader\$package	93.09 (1%)	0.00 (0%)	1
LinkedHashMap	90.98 (1%)	0.00 (0%)	2,859
SoftReference.referent	82.62 (1%)	0.00 (0%)	746

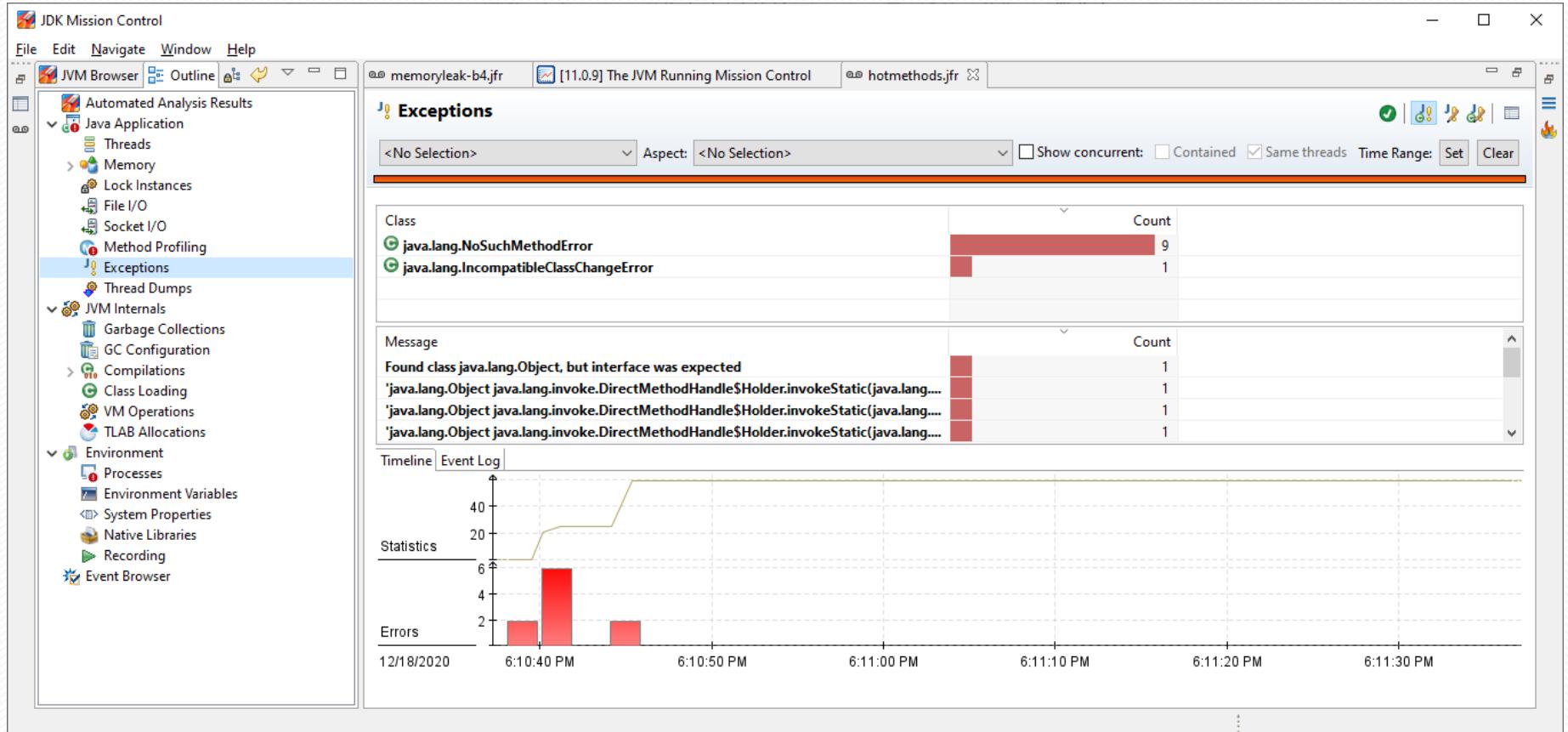


Ancestor Referrer

Name	Memory KB	Overhead KB
{HashMap}	3,941.37 (29%)	0.00 (0%)
com.mcnz.jfr.jmc.gc.MikeTyson.map	3,637.63 (27%)	0.00 (0%)
javax.management.openmbean.CompositeData\$CompositeData	632.24 (5%)	0.00 (0%)
Object[]	463.13 (3%)	0.00 (0%)
N/A	233.08 (2%)	0.00 (0%)
java.lang.reflect.Method[]	218.25 (2%)	0.00 (0%)
java.lang.invoke.LambdaForm\$Name[]	193.31 (1%)	0.00 (0%)
TreeMap	166.82 (1%)	0.00 (0%)
ArrayList	115.78 (1%)	0.00 (0%)
java.lang.invoke.LambdaForm\$Name[]	108.08 (1%)	0.00 (0%)
LinkedHashMap	107.49 (1%)	0.00 (0%)
java.lang.invoke.MethodType\$ConcurrentWeaver\$MethodType	102.54 (1%)	0.00 (0%)
javax.management.openmbean.TabularData\$Table\$Table	99.86 (1%)	0.00 (0%)
byte[]	98.52 (1%)	0.00 (0%)
jdk.internal.loader.BuiltinClassLoader\$package	93.09 (1%)	0.00 (0%)

Ancestor prefix: Update

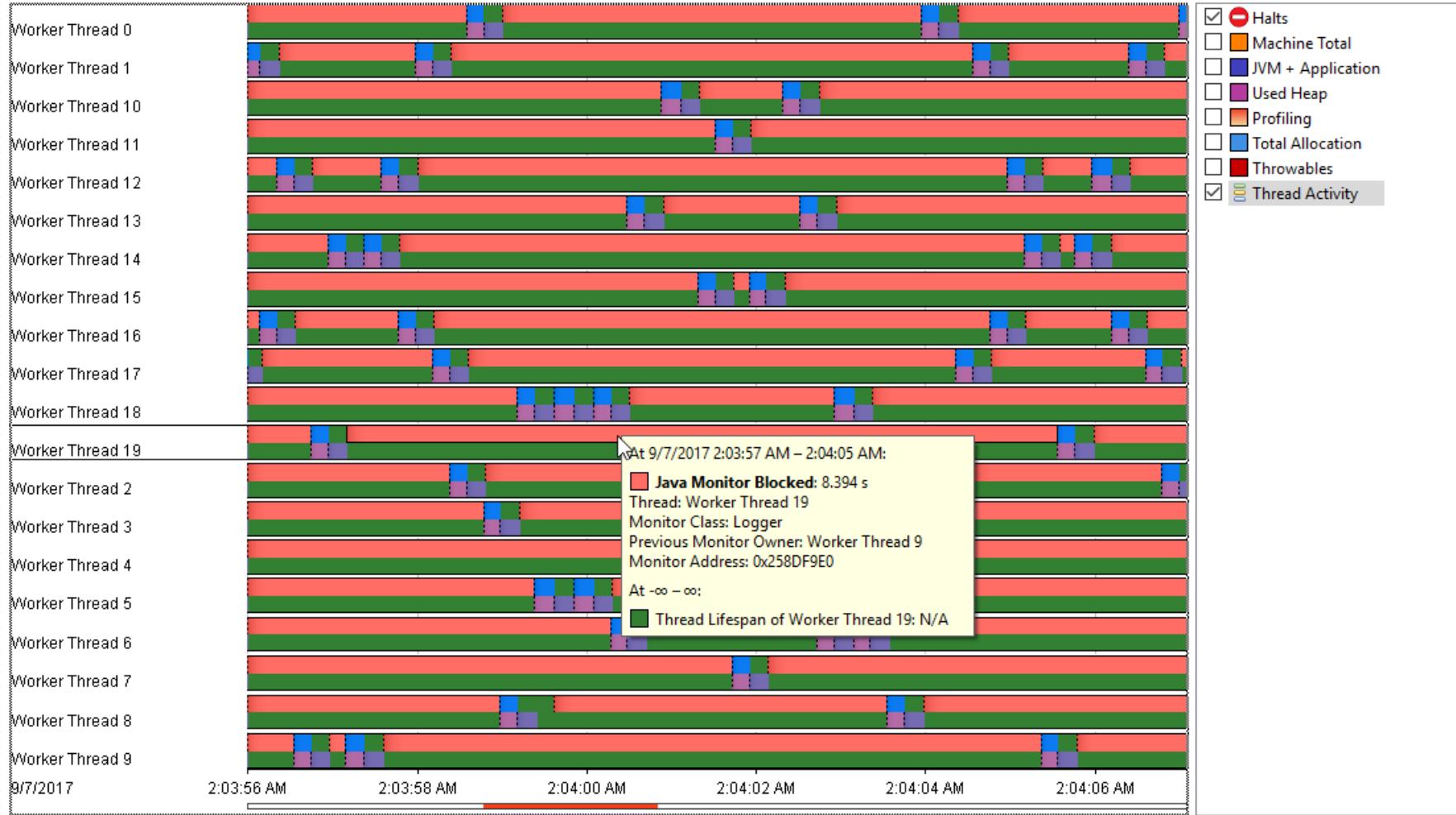
Exception Monitoring



Exception Monitoring

- Exception throwing is expensive
 - Design so that exceptions rarely happen
 - Handle exceptions early
 - Exception cost is relative to exception depth
- Reflection is very convenient but expensive
 - Have to create an instance of a method
 - Call invoke
 - Invoke has to perform validation and security checks
 - GC cleans up an object that wouldn't exist otherwise
 - No option for compiler in-lining

Thread Monitoring

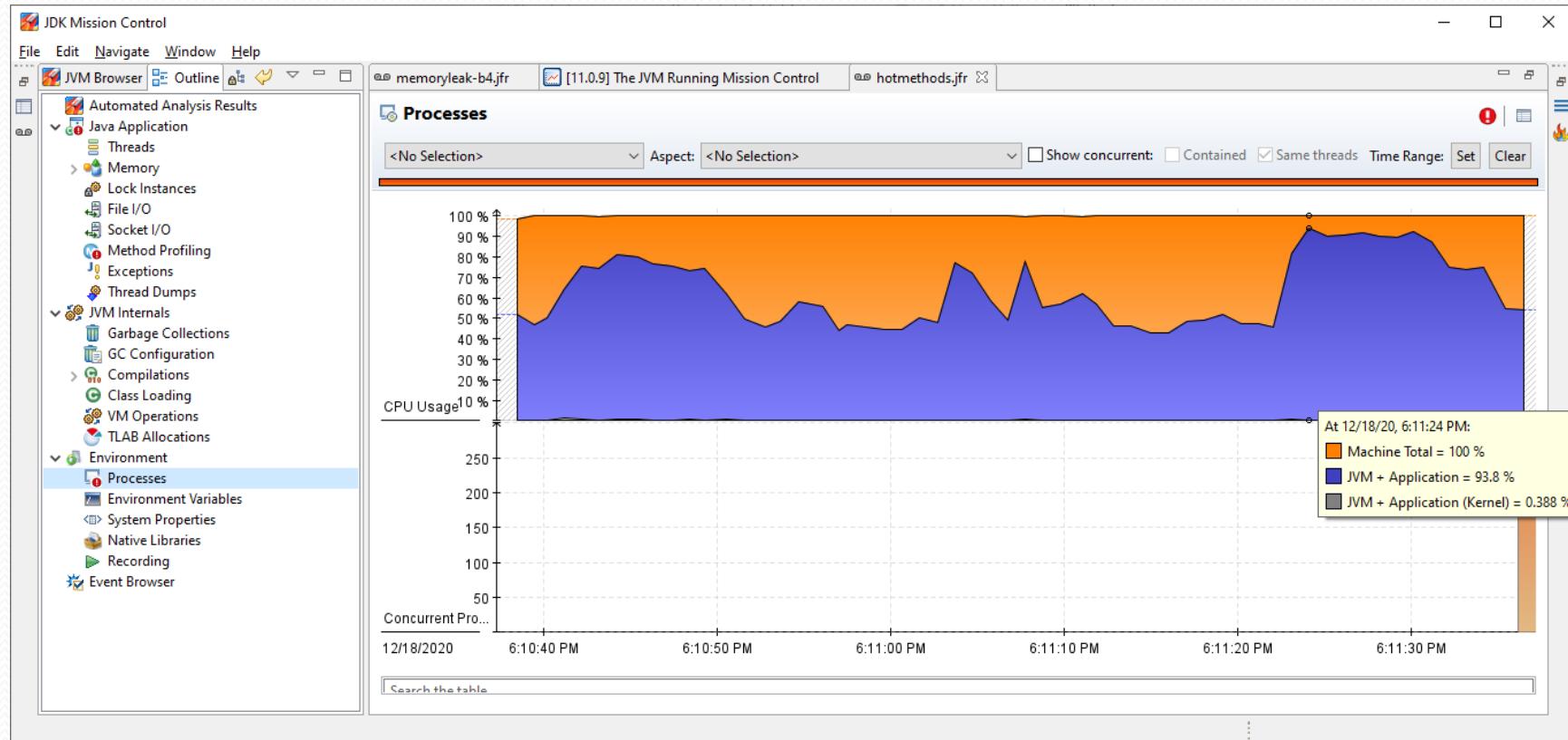


Threading

- Define code in terms of **Runnable** objects
- What are the smallest units of work that can be spread across threads?
- Application servers
 - Thread per HTTP request
 - Thread per message in queue
 - Tune the thread pool size so clients aren't waiting
 - Static content may be best served by a non-Java solution
- Threads tend to collect at resource points
 - Database calls are often a choke point



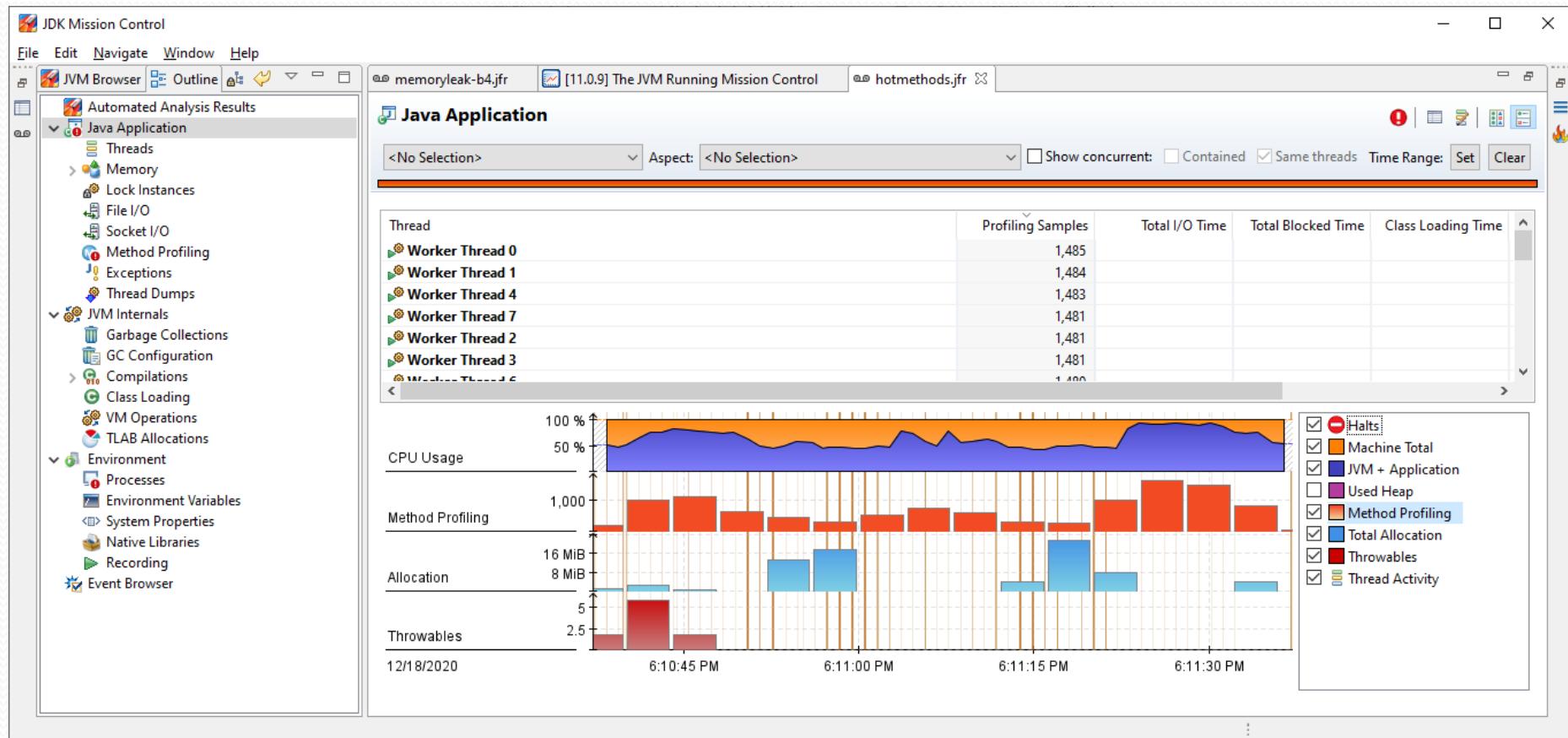
Is there a CPU issue?



CPU Performance

- Describes performance around maximizing the CPU utilization
- Makes sense if the application is CPU bound
- Multi-Core architectures can take advantage of threads
- It is often said that ‘the processor lies’
 - Threading, memory and hardware issues can all manifest themselves as CPU performance problems.

Object Allocations?



Method Profiling

JDK Mission Control

File Edit Navigate Window Help

JVM Browser Outline Method Profiling

Automated Analysis Results

Java Application

- Threads
- Memory
- Lock Instances
- File I/O
- Socket I/O

Method Profiling

- Exceptions
- Thread Dumps

JVM Internals

- Garbage Collections
- GC Configuration
- Compilations
- Class Loading
- VM Operations
- TLAB Allocations

Environment

- Processes
- Environment Variables
- System Properties
- Native Libraries
- Recording

Event Browser

[11.0.9] The JVM Running Mission Control [memoryleak-b4.jfr] [hotmethods.jfr]

Method Profiling

<No Selection> Aspect: <No Selection>

Show concurrent: Contained Same threads

Top Package

Package	Count
java.lang	9,088
java.util	2,598
com.mcnz.jmc	167
jdk.jfr.internal	4

Top Class

Class	Count
java.lang.Integer	9,088
java.util.LinkedList	2,360
java.util.LinkedList\$ListItr	238
com.mcnz.jmc.HotJavaMethodRunner	167
jdk.jfr.internal.RequestEngine	2
jdk.jfr.internal.Logger	1
jdk.jfr.internal.PlatformRecorder	1

Profile of a Memory Leak

JDK Mission Control

File Edit Navigate Window Help

memoryleak-b4.jfr

Garbage Collections

<No Selection> Aspect: <No Selection>

Show concurrent: Contained Same threads Time Range: Set Clear

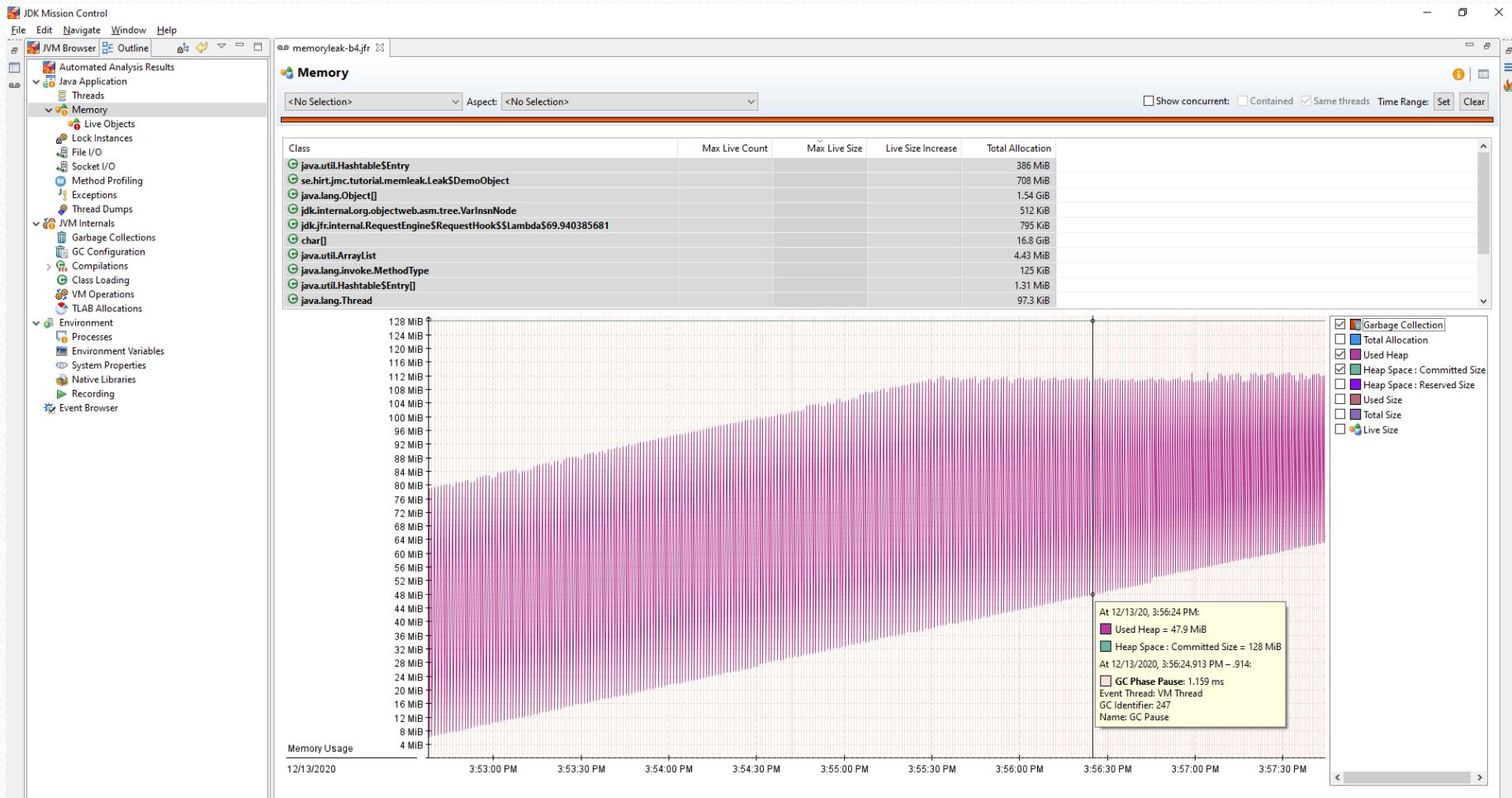
GC ID	Cause	Collector Name	Longest Pause	Sum
271	G1 Evacuation Pause	G1New	1.892 ms	
319	G1 Evacuation Pause	G1Old	1.691 ms	
272	G1 Evacuation Pause	G1New	1.619 ms	
66	G1 Evacuation Pause	G1New	1.572 ms	
131	G1 Evacuation Pause	G1New	1.545 ms	
134	G1 Evacuation Pause	G1New	1.457 ms	
31	G1 Evacuation Pause	G1New	1.441 ms	
132	G1 Evacuation Pause	G1New	1.427 ms	
145	G1 Evacuation Pause	G1New	1.418 ms	
136	G1 Evacuation Pause	G1New	1.416 ms	
137	G1 Evacuation Pause	G1New	1.399 ms	
46	G1 Evacuation Pause	G1New	1.399 ms	

Event Type	Name	Duration	Start Time
GC Phase Pause Level 1	Reconsider SoftReferen...	622 ns	12/13/20, 3:5
GC Phase Pause Level 1	Notify Soft/WeakRefer...	7.525 µs	12/13/20, 3:5
GC Phase Pause Level 1	Notify and keep alive fi...	53 ns	12/13/20, 3:5
GC Phase Pause Level 1	Notify PhantomReferen...	2.334 µs	12/13/20, 3:5
GC Phase Pause Level 1	Reconsider SoftReferen...	562 ns	12/13/20, 3:5
GC Phase Pause Level 1	Notify Soft/WeakRefer...	16.126 µs	12/13/20, 3:5
GC Phase Pause Level 1	Notify and keep alive fi...	57 ns	12/13/20, 3:5
GC Phase Pause Level 1	Notify PhantomReferen...	1.973 µs	12/13/20, 3:5
GC Phase Pause Level 1	Reconsider SoftReferen...	528 ns	12/13/20, 3:5
GC Phase Pause Level 1	Notify Soft/WeakRefer...	6.875 µs	12/13/20, 3:5
GC Phase Pause Level 1	Notify and keep alive fi...	55 ns	12/13/20, 3:5

Used Heap
Heap Space : Committed Size
Heap Space : Reserved Size
Used Heap Post GC
Metaspace : Used
Metaspace : Committed
Metaspace : Reserved
Longest Pause
Sum of Pauses
Pause Phases
Thread Activity

The visualization shows memory usage over time from 12/13/2020 at 3:53:00 PM to 3:57:00 PM. The Y-axis represents memory size in MiB (64 MiB, 32 MiB, 8 MiB, 4 MiB) and time in minutes (25 ms). The X-axis shows the timeline. The 'Used Heap' series (purple) shows high volatility with frequent spikes, starting around 64 MiB and ending near 68 MiB. The 'Used Heap Post GC' series (pink) shows a steady linear increase from approximately 20 MiB to 35 MiB. The 'Longest Pause' series (orange) and 'Sum of Pauses' series (light orange) show periodic spikes, indicating regular GC events.

A Closer Look at the Heap



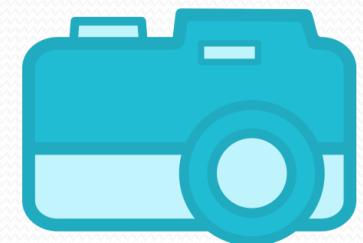
Mission Control GC Configuration

The screenshot shows the JDK Mission Control application window. The left sidebar navigation tree is expanded, showing categories like Java Application, JVM Internals, and Environment. Under Java Application, 'GC Configuration' is selected. The main panel displays the 'GC Configuration' screen with three tabs: 'GC Configuration', 'Heap Configuration', and 'Young Generation Configuration'. The 'GC Configuration' tab contains settings for the Young Garbage Collector (G1New), Old Garbage Collector (G1Old), Concurrent GC Threads (2), Parallel GC Threads (8), Concurrent Explicit GC (false), Disabled Explicit GC (false), Uses Dynamic GC Threads (true), and GC Time Ratio (12). The 'Heap Configuration' tab includes settings for Initial Heap Size (128 MiB), Minimum Heap Size (128 MiB), Maximum Heap Size (128 MiB), If Compressed Oops Are Used (true), CompressedOops Mode (32-bit), Heap Address Size (32), and Object Alignment (8 B). The 'Young Generation Configuration' tab lists settings for Minimum Young Generation Size (1.3 MiB), Maximum Young Generation Size (1.3 MiB), New Ratio (2), Initial Tenuring Threshold (7), Maximum Tenuring Threshold (15), TLABs Used (true), Minimum TLAB Size (2 KiB), and TLAB Refill Waste Limit (64 B).

GC Configuration		Heap Configuration		Young Generation Configuration	
Young Garbage Collector	G1New	Initial Heap Size	128 MiB	Minimum Young Generation Size	1.3 MiB
Old Garbage Collector	G1Old	Minimum Heap Size	128 MiB	Maximum Young Generation Size	1.3 MiB
Concurrent GC Threads	2	Maximum Heap Size	128 MiB	New Ratio	2
Parallel GC Threads	8	If CompressedOops Are Used	true	Initial Tenuring Threshold	7
Concurrent Explicit GC	false	CompressedOops Mode	32-bit	Maximum Tenuring Threshold	15
Disabled Explicit GC	false	Heap Address Size	32	TLABs Used	true
Uses Dynamic GC Threads	true	Object Alignment	8 B	Minimum TLAB Size	2 KiB
GC Time Ratio	12			TLAB Refill Waste Limit	64 B

Other Profiling Tools

- **JVisualVM** - comes with JDK
 - Connect to local or remote JVMs
 - Shows Heap, Memory Pools, CPU utilization
 - Can open snapshots, core dumps and heapdumps
 - Can connect to JMX agents in Application Servers
 - Run once - capture snapshot
 - Run again - compare against snapshot
- **Java HPROF** - comes with JDK
 - Agent that inserts profiling context in classes on load
 - Uses class loader hooks to register event listeners
 - Java development team uses it for internal development
- **YourKit** - popular commercial solution
 - Integrates with IDEs very well
- **JProfiler** - less popular commercial solution
 - Integrates well with IDEs



Optimization Advice

- There's always a bottleneck
- Don't blame the app when it's the stack
- The CPU lies
- Validate assumptions
- Performance test throughout the lifecycle
- Don't micro-tune
- Don't blame the stack when it's the app
- The rules change
- Pay attention to the tails
- The game changes, so adjust accordingly

You're Not On Your Own

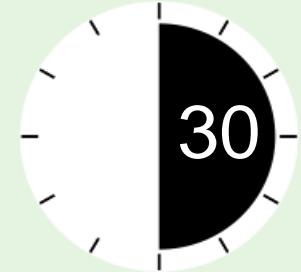


Summary

- Java Performance is affected by many factors
- Tuning an application is different for every application
- Understand the application profile
- Using good design performance makes tuning easier
- Exception handling can affect performance
- Performance means different things based on context
- "Premature optimization is the death of a project" ~ Dr. Knuth
- When is it good enough?
- **Continue to Lab Introduction on next slide**

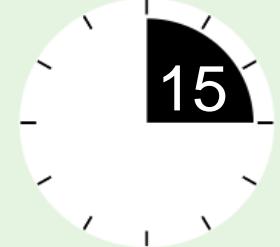


Lab 7 Introduction



- **Performance**
- Example program reads in an image and blurs it in four different degrees based on a **width** parameter
- Use two options for diagnosing performance
 - **HPROF** text file for post-execution statistics
 - **JVisualVM** to monitor real time performance
 - Discover how coding changes have a significant performance impact
- After the lab, we will have a lab review
 - If virtual, green check when you are done

Lab 7 Review - Discussion



- **Performance**
- Use two options for diagnosing performance
 - HPROF text file for post-execution statistics
 - JVisualVM to monitor real time performance
 - How coding changes impact performance?
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

Writing Effective Java

Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- **Writing Effective Java**
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Explore some good coding practices

Examine some common poor performing techniques

Examine tradeoffs between some common code strategies

Discuss application of some common OO techniques to produce better code

Creating and Destroying Objects

- Avoid duplicate instances

```
String s = new String("some string"); // now 2
```

- Avoid creating constants in a loop or instance method

- Create them once and reuse

```
static START_OF_FISCAL_YEAR;  
static {  
    START_OF_FISCAL_YEAR = somedate; //derive date  
}
```

- Remove references to obsolete objects from collections

- Example - stack or queue implemented in array with an offset being the location on the stack
 - As elements are inserted or removed, only the index is changed
 - Leaves the object references still in the array

```
private Object[] elements = new Object[1];  
...  
elements[size] = anObject;  
int size = 0;
```

- Instead

```
elements[size] = null; // remove reference to anObject  
elements = null; // remove reference to the allocated array
```

Finalizers

- Avoid
 - Unpredictable - no guarantee they will be executed
 - Dangerous - may happen at the wrong time
 - Unnecessary
- Run when the GC decides to reclaim an object's space
- No guarantee of the order finalizers are executed
- Exceptions are ignored
- GC thread usually low priority so may never run
- Only use to clean up as a safety net
 - Example: clean up resource if **close()** not called

Static Factory Methods

- Instead of providing public constructors with different parameter lists, provide **Factory** methods
- Clearer names
- Possible to return interfaces not class types
- Common factory method names
 - **getInstance()** used by Calendar and DateFormat
 - **valueOf()** used with Number types
- If immutable, then can return only one static instance
 - Can become a **Singleton**
- CONS:
 - Not really obvious how to use static factory methods
 - Can't subclass instances - may be a pro or a con

Common Methods (1 of 2)

- Overriding: equals, hashCode, toString, clone, and finalize
- **equals** - two objects are equivalent but not the same instance
 - Symmetrical `a.equals(b) == b.equals(a)`
 - Reflexivity `a.equals(a)`
 - Transitive `a.equals(b) and b.equals(c) and c.equals(a)`
 - **Super** works if all variables are primitives
 - Check for null
 - Check for instance type
 - Check every primitive variable using `==`
 - Check every Object variable using `equals(this.variable)`

Common Methods (2 of 2)

- Overriding: `equals`, `hashCode`, `toString`, `clone`, and `finalize`
- **hashCode** - always override with `equals` and include the same variables used in `equals` test
- **toString**
 - Should produce a String representation of the interesting variables in the class
 - Document the format
- **clone** - override rarely - represents a deep copy

Classes and Interfaces

- Minimize access to data
- Prefer interface to class when returning values from methods
- Favor immutability
- Favor composition over inheritance
 - Breaks encapsulation
 - Functionality of subclass becomes transparent when not desired
- Favor interfaces over abstract classes
 - Multiple inheritance issues are avoided
- Judiciously use nested classes
 - Consider if really appropriate

Immutability

- Class data cannot be changed - ever!
- Classes
 - String
 - BigInteger and BigDecimal
- Characteristics
 - No **mutator methods** (setters)
 - All fields final
 - All fields private
 - All methods final
- Usage
 - If using client-provided objects, make copies

Composition vs. Inheritance

- Composition - **contains a type**
 - Class has a field with the type
- Inheritance - **is a type**
 - Class extends a type
- Composition needs to provide methods to allow access to composed type functionality
- Inheritance exposes all accessible methods and fields
- Composition often referred to as **Wrapper** class
- Inheritance can introduce unexpected behavior in superclass

Threads

- Anticipate thread safety issues
- Use classes that are known to be thread-safe
- Synchronization must be used to protect atomic data
- When in a synchronized block, never transfer control to unsynchronized methods that can be overridden
- Always invoke wait in a loop and handle the exception and perform a readiness check
- Document the thread safety of your implementations
- It will be assumed that it is not thread-safe

Synchronization

- Always synchronize access to the data that needs to be thread safe
- Try to reduce the atomic requirements of the class to the minimum
- Once a lock is acquired, keep it only as long as needed to complete the work
- Synchronized blocks can be used instead of synchronized methods to reduce scope of locking

Serialization

- Making a class serializable is as simple as **implements Serializable**
- Serializable classes should declare a **serialVersionUID** to indicate the implementation of a class has not changed between VMs
- Generate new **serialVersionUID**
 - Change inheritance
 - Add or remove fields
 - Change field type or transience
- Cost associated with use
 - Maintenance - each change of class has to determine serialization; behavior might have changed
 - Affects inheritance
 - Inner classes can be a problem
- Should implement a custom **writeObject** and **readObject**

read/writeObject() Considerations

- Implementing **readObject** and **writeObject** may be necessary when implementing Serializable
- Export the state of the object in the most neutral form possible
- Consider each field that is non-transient carefully
- Can the information be derived from something already available in the VM?
- Consider how fields will serialize across VMs
 - Will fields be supportable in the same type?
 - JVM 6 vs. JVM 8 types are not the same
- Consider containment hierarchies and the object map

Summary

- Effective Java is relative - no such thing as perfect
- Every choice a developer makes has a cost
- Think before coding
- Good coding practices will come naturally over time
- Not everyone will agree with every choice made
- Static code analyzers will catch some of these
- During code review, ask Why...



Data Structures

Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- **Data Structures**
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

Objectives

Explore the use of generic types

Discuss the various collection implementations

Discuss how to use collections effectively

Learn about the Collection utility class

Learn about the collection enhancements for JDK 8

Generics Overview

- Generics produce type safe code
 - Bugs are detectable at compile time instead of runtime
 - Prevents unexpected **ClassCastException** at runtime
- Enables classes, interfaces and methods to be parameterized by types
- Allows the compiler to cast return values for parameterized methods
- Java SE 5.0 updated all collection classes to be generic classes
- Java SE 7.0 improved generic support
 - Diamond Operator - Compiler infers the types

```
List<Integer> l = new ArrayList<>();
```

Declaring a Generic Type

- Define a class or interface as usual, except:
 - Following the type name, enclose the type parameter(s) in angle brackets (<>)
 - The type variable can be used within instance variables or methods
 - By convention, type parameter names are single, uppercase letters
 - Common type parameter names:
 - E - Element
 - K - Key
 - N - Number
 - T - Type
 - V - Value
- ```
public class MyGenericType<T> {
 private T t;
 // ...
}
```

# Invoking a Generic Type

- An invocation of a generic type is also known as a **parameterized type**
  - A generic class can have one or more type parameters
  - Type parameter can be any type or a wildcard type `<?>`
  - A type parameter can't be any of the primitive data types
- To instantiate a parameterized type, use the following format:  
`Type<TypeParameter> o = new Class<TypeParameter>();`

```
List<Integer> l = new ArrayList<>();
```

```
Map<String, String> m = new HashMap<>();
```

# Simple Generic Class

```
package simple.examples;

public class MyGenericType<T> {
 private T t;
 public void add(T t) { this.t = t; }
 public T get() { return t; }

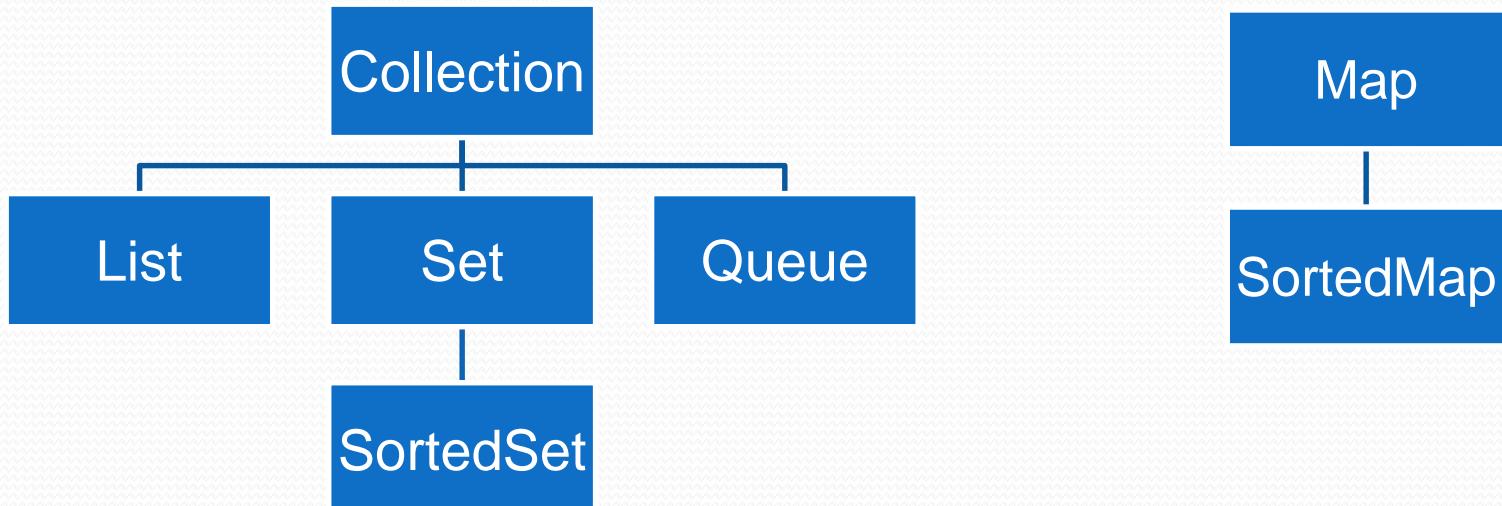
 public static void main(String[] args) {
 MyGenericType<String> gen = new MyGenericType<String>();
 gen.add("Only Strings");
 String s = gen.get();
 gen.add(8); //Compile error;
 MyGenericType<Integer> gen2 = new MyGenericType<Integer>();
 gen2.add(8);
 gen2.add("Only Strings"); //Compile error;
 }
}
```

# Collection Framework

- The Java Collection Framework consists of the following:
  - **Interfaces** - abstract data types representing collections
  - **Implementations** - concrete implementations of the collection interfaces
  - **Algorithms** - methods that provide collection-related operations such as searching and sorting
- The Java Collection Framework offers many benefits
  - Reduces development effort
  - Improves performance
  - Enables interoperability between APIs
  - Promotes reuse

# Collection Interfaces

- **Collection** - group of objects that may contain duplicates
- **Set** - group of objects that MUST NOT contain duplicates
- **List** - an ordered group of objects that may contain duplicates
- **Map** - group of key-value pairs
- **SortedSet** - a Set sorted in ascending order
- **SortedMap** - a Map sorted in ascending key order
- **Queue** — holds a group of elements prior to processing



# Interface Methods (1 of 2)

| METHOD                                         | FUNCTION                                                                            |
|------------------------------------------------|-------------------------------------------------------------------------------------|
| <b>add(E e)</b>                                | Inserts the specified element.                                                      |
| <b>addAll(Collection&lt;? extends E&gt; c)</b> | Adds all the elements in the given collection.                                      |
| <b>clear()</b>                                 | Removes all of the contained elements.                                              |
| <b>remove(Object o)</b>                        | Removes a single instance of the given element.                                     |
| <b>removeAll(Collection&lt;?&gt; c)</b>        | Removes all the contained elements that are also contained in the given collection. |
| <b>retainAll(Collection&lt;?&gt; c)</b>        | Retains only the contained elements that are also found in the given collection.    |

# Interface Methods (2 of 2)

| METHOD                                    | FUNCTION                                                                                        |
|-------------------------------------------|-------------------------------------------------------------------------------------------------|
| <b>contains(Object o)</b>                 | Determines if this collection contains the specified element.                                   |
| <b>containsAll(Collection&lt;?&gt; c)</b> | Determines if this collection contains all of the elements in the given collection.             |
| <b>isEmpty()</b>                          | Determines if this collection contains zero elements.                                           |
| <b>iterator()</b>                         | Returns an iterator over the contained elements.                                                |
| <b>size()</b>                             | Returns the number of contained elements.                                                       |
| <b>toArray()</b>                          | Returns an array containing all of the elements in this collection.                             |
| <b>toArray(T[ ] a)</b>                    | Returns an array of the specified array type containing all of the elements in this collection. |

# Iterator

- The **Iterator** interface is provided to traverse the contents of a Collection
- This interface defines the following methods:
  - **hasNext()** - Determines if the iteration has more elements
  - **next()** - Returns the next element
  - **remove()** - Removes the last element returned by the iterator from the underlying collection

```
Collection<?> coll = getAll();
Iterator<?> it = coll.iterator();
while (it.hasNext()) {
 Object o = it.next();
}
```

# Enhanced For Loop

- Used to *forward* iterate through a collection
- You do not have access to the loop counter or the Iterator's **remove()** method

```
Collection<String> coll = getAll();
Iterator<String> it = coll.iterator();

for (String s : it) {
 log(s);
}
```

# Implementations (1 of 2)

| Interface                               | Implementation                                 | Details                                                                                                                                   |
|-----------------------------------------|------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Set</b>                              | <b>HashSet</b>                                 | Stores its elements in a hash table. Best-performing <b>Set</b> implementation.                                                           |
| <b>Set</b>                              | <b>LinkedHashSet</b>                           | Stores its elements in a hash table ordered by insertion-sequence.                                                                        |
| <b>SortedSet</b><br><b>NavigableSet</b> | <b>TreeSet</b>                                 | Sorts its elements by value. Slowest-performing.                                                                                          |
| <b>Set, EnumSet</b>                     | <b>RegularEnumSet</b> ,<br><b>JumboEnumSet</b> | A specialized Set implementation for use with Enum types. RegularEnumSet supports up to 64 values.                                        |
|                                         | <b>BitSet</b>                                  | This class implements a vector of bits that grows as needed. Is not a Set but has Set like behavior.                                      |
| <b>List</b>                             | <b>ArrayList</b>                               | Resizable-array implementation permits all elements, including null.                                                                      |
| <b>List</b>                             | <b>LinkedList</b>                              | Permits all elements, including null. Provides additional methods to get, remove and add an element at the beginning and end of the list. |

# Implementations (2 of 2)

| Interface                               | Implementation                           | Details                                                                                        |
|-----------------------------------------|------------------------------------------|------------------------------------------------------------------------------------------------|
| <b>Queue</b>                            | <b>ArrayQueue</b>                        | A linear collection that supports element insertion and removal.                               |
| <b>Deque</b>                            | <b>ArrayDeque</b> ,<br><b>LinkedList</b> | A linear collection that supports element insertion and removal at both ends.                  |
| <b>SortedMap</b><br><b>NavigableMap</b> | <b>TreeMap</b>                           | Red-Black tree based implementation maintains mappings in ascending key order.                 |
| <b>Map</b>                              | <b>HashMap</b>                           | Hash table-based implementation permits null values and the null key. Mappings are not sorted. |
| <b>Map</b>                              | <b>LinkedHashMap</b>                     | Extends HashMap and provides predictable iteration order.                                      |

# Synchronized Collections

- Synchronized Wrappers
  - The Collections class provides static factory methods that return a *synchronized* Collection of each interface
  - ALL access to the underlying collection must be performed through the returned collection
- Un-modifiable Wrappers
  - The Collections class provides static factory methods that return an *un-modifiable* Collection of each interface
  - ALL access to the underlying collection must be performed through the returned collection

```
import java.util.concurrent.Collections;
List<String> syncList = Collections.synchronizedList(list);
List<Integer> unmodList = Collections.unmodifiableList(list);
```

# Collections Utilities

- **copy()** - Copies all the elements from one list to another
- **binarySearch()** - Searches the given list for the given element using the binary search algorithm
- **fill()** - Replaces all of the elements of the given list with the given element
- **indexOfSubList()** - Returns the index of the first occurrence of the given target list within the given source list
- **lastIndexOfSubList()** - Returns the index of the last occurrence of the given target list within the given source list
- **min()** - Returns the minimum element of the given collection
- **max()** - Returns the maximum element of the given collection

# More Collection Utilities

- **replaceAll()** - Replaces all occurrences of the specified value in the given collection with another value
- **reverse()** - Reverses the order of the list's elements
- **rotate()** - Rotates the list's elements by a specified distance
- **shuffle()** - Randomly arranges the list's elements
- **sort()** - Sorts the given list according to its elements' natural ordering or a comparator (if specified)
- **swap()** - Swaps the specified elements in the given list

# Comparable

- Create a Comparable type by implementing **java.lang.Comparable**
  - Implement its single method: **int compareTo(T o)**
  - Return a **negative** integer if this object is less than the given object
  - Return a **zero** if the objects are equal
  - Return a **positive** integer if this object is greater than the specified object
- Sort collections by providing a **Comparator** implementation
  - Implement the compare method: **int compare(T o1, T o2)**
  - Return a **negative** integer if first argument is less than the second
  - Return **zero** if the arguments are equal
  - Return a **positive** integer if the first argument is greater than the second
- If two objects cannot be compared, throw a **ClassCastException**

# Comparator Example

```
Collections.sort(workers, new Comparator<WorkerVO>() {
 public int compare(WorkerVO w1, WorkerVO w2) {
 int comp = w1.getLastName().compareToIgnoreCase(
 w2.getLastName());
 if (comp == 0) {
 comp = w1.getFirstName().compareToIgnoreCase(
 w2.getFirstName());
 }
 return comp;
 }
});
```

# Java 8 Collection Improvements

- **Stream** - a sequence of elements
  - Different from a collection
  - Not a data structure but a pass thru to a pipeline
- **Pipeline** - sequence of aggregate operations
  - Source - collection, array, IO Channel
  - Intermediate operators - Zero or more streams or filters
  - Terminal operator - forEach
- **Filter** - return streams upon which a filter has been applied

# Aggregation vs. Iteration

- Developers iterate over elements using Iterator interfaces
- Aggregate operations handle iteration internally
  - Developers say what collection to use
  - Developers say what operations to perform
  - JDK figures out how
  - Operations process elements from streams
  - Support behavior as a parameter

# Aggregator Example

```
double average = customers ← Source Data (a collection)
```

```
.stream() ← Convert to stream
```

```
.filter(p -> p.getGender() == Customer.Sex.MALE)
```

```
.mapToInt(Customer::getAge)
```

```
.average()
```

```
.getAsDouble();
```

Terminal  
Operation

Intermediate  
Operations

# String Usage Tradeoffs

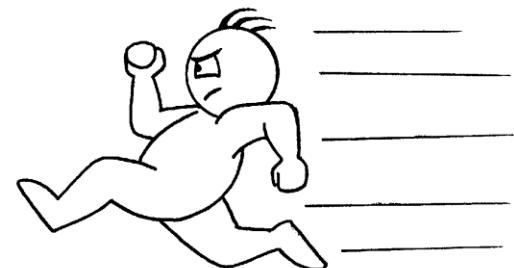
- Everyone uses **StringBuffer** or **StringBuilder** to build Strings
- If we are building tons of Strings, that might make sense
- Trades performance for cluttered code
- **String.intern** can save memory but developers have to code it
  - Constants and literals are automatically interned
  - Use when there is a significant amount of dynamic String duplication

# JDK 8 String dedup

- Alternative to `String.intern()`
- Available with the G1 garbage collector
- `XX:+UseStringDeduplication`
- During GC looks for candidate String instances
- Dedups identical `char[]` by pointing both String instances to the same `char[]`
- Batch processors not good candidates
- Web servers likely can take advantage

# Arrays

- Array is the only JVM built-in collection
- Use when you can know the bounds
- It is *very fast* for access by index
- `java.util.Arrays` has many helper methods
  - Copying, filling, equality checking, hashCode, sort
- Collections can be built from them
- Collections can be loaded into them



# Algorithm Analysis

- Know the data that the application will be using
- What is the most common operation?
  - Insertion
  - Searching
  - Iterating
    - Does order matter?
- HashMap performance consideration
  - Pre-allocate the size
- HashMap collisions get progressively more expensive

# Summary

- The Java Collection Framework provides many implementations but it is important to know the data to pick the right one
- Generics provide a very useful way to provide data agnostic algorithms
- Performance is very important so anticipate performance problems and design accordingly
- Security is a never ending problem as hackers will always be looking for weaknesses
- **Continue to Lab Introduction on next slide**



# Lab 8 Introduction



- **Collections**
- Use two collections to find which best performs to load data from a CSV file into a collection and manipulate it. The time to perform the operations will be logged.
  - **ArrayList** and **ConcurrentSkipListMap**
- Provided application has infrastructure and you will code all the behavior for creating the collections, and inserting and removing elements
- After the lab, we will have a lab review
  - For virtual, green check when you are done

# Lab 8 Review - Discussion



- **Collections**
- Used **ArrayList** and **ConcurrentSkipListMap**
  - Load data from a CSV file into a collection and manipulate it
- What did you discover about the performance of each collection?
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?



This slide has been intentionally left blank.

# Internationalization

# Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- **Internationalization**
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

# Objectives

Understand the concept of Internationalization (I18N)

Discuss the importance of I18N

Discuss some enhancements in Java for I18N with Date and Time

Learn how to output and format based on language and region

# Fundamentals of Localization

- Internationalization (**I18N**)
- Most applications are being used by a broader audience
- Region and language specific presentation is now expected
  - US - **\$1,001,234.99 Jan 3, 1999**
  - Europe - **1.001.234,99 3 Jan, 1999**
- Developers usually develop in one language
  - English - **The time is:**
  - Spanish - **El hora es:**
- Some customization within same language, different country
  - US English **color** vs. UK English **colour**
- Developers don't want to have **tons/tonnes** of language customizations to write
- Imagine coding all sorts of if-then-elses or switches!



# Localizing Strings and Numbers

- Use **ResourceBundles** for Strings

```
String prompt = "Enter color";
```

- Becomes

```
Bundle = ResourceBundle.loadResource("Message");
String prompt = bundle.getString(COLOR_KEY);
```

- Reads in a **Message.properties** or a localized version

**Message\_en.properties** (ENGLISH)

**Message\_es.properties** (SPANISH)

**Message\_de\_DE.properties** (GERMAN, GERMANY)

- Numbers

```
NumberFormat.getCurrencyInstance(currentLocale).format(12345678.99)
```

# Localizing Dates

```
DateTimeFormatter dateformatter =
 DateTimeFormatter.ISO_LOCAL_DATE.ofLocalizedDate(
 TextStyle.MEDIUM).withLocale(currentLocale);
```

```
DateTimeFormatter timeformatter =
 DateTimeFormatter.ISO_LOCAL_TIME.ofLocalizedTime(
 TextStyle.MEDIUM).withLocale(currentLocale);
```

```
LocalDate.now().format(dateformatter) + " locale "
+dateformatter.getLocale()
```

```
LocalTime.now().format(timeformatter) + " locale "
+timeformatter.getLocale()
```

# Summary

- Internationalization is increasingly more important
- Think about it early in the lifecycle of the project
- Design libraries with it in mind
- Make it part of testing
- **Continue to Lab Introduction on next slide**



# Lab 9 Introduction



- **Internationalization**
- Easy lab ☺
- Support internationalization of applications
- Use **ResourceBundles** to manage the international content of strings
- Use **Locales** to provide content for numbers, dates and times that are internationalized
- After the lab, we will have a lab review
  - Green check when you are done

# Lab 9 Review - Discussion



- **Internationalization**
- **ResourceBundles** to manage the international content of strings
- **Locales** to provide content for numbers, dates and times that are internationalized
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

# Java Security

# Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- **Java Security**
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

# Objectives

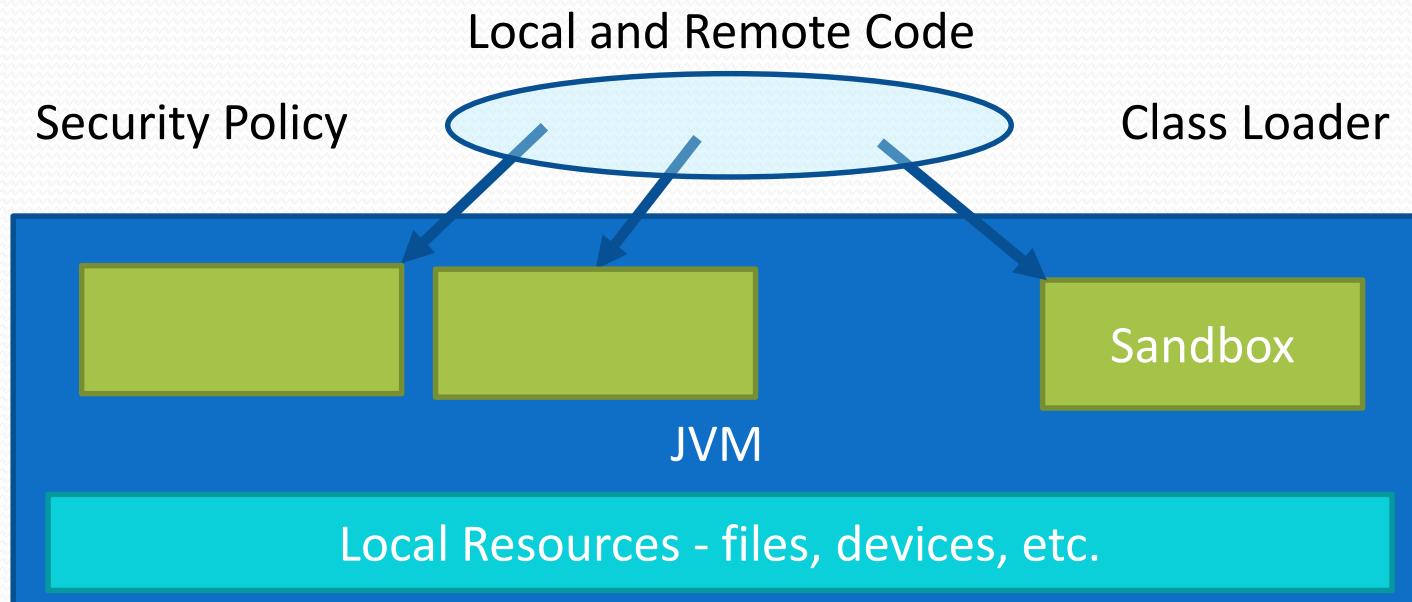
Understand some of the Java features to secure the JVM

Understand the difference between declarative and programmatic security

Discuss security in the context of JEE containers

# Java 2 Security Architecture

- Sandbox Model
  - Local code is considered secure
  - Remote code is considered insecure
  - Code can be signed, marked trusted and execute as local



# JVM Security Features

- Security Policy
- Class Loader Isolation
- Permissions
  - FilePermission
  - SocketPermission
  - BasicPermission
  - RuntimePermission
- Roles
- SecurityManager
- AccessController
- ProtectionDomain

# SecurityManager

- Methods to check access
  - `checkRead("path/file")`
  - `checkWrite("path/file")`
  - `checkPermission(perm)`
  - `checkAccess("host", port)`
  - `checkExec("CMD")`
  - `checkConnect("host", port)`
  - `checkPackageAccess("package.name")`
  - `checkPropertyAccess("aProperty")`
- Exceptions are thrown when permission not allowed

# Byte Code Verifier

- Where do the .class files come from?
  - Are they trustworthy?
- Java class loaders don't trust the network
  - No forged pointers
  - Class checked for known parameter types
  - No illegal class accesses
  - No operand stack overflows or underflows
- If class is known, operationally safe code can be executed without JVM checks later

# Class Loaders

- Primordial class loader
  - Native implementation to bootstrap the JVM
  - Not manifest to the Java context
  - System specific implementation
  - Classes loaded by this loader have a null class loader
- Responsible for finding classes at runtime
- Searches current class loader and parent class loaders for class already loaded
- If a delegation parent class loader exists, delegate
- If not loaded by delegate, tries to load somehow
- Classes loaded by one class loader are only visible to that class loader and child class loaders

# Class Loader Security Issues

- Distributed network platforms need to serialize data between machines
- Class references can be passed between machines
- Malicious user could package a class or groups of classes in the stream
- Apache commons-collection used this technique and was found to have a security hole
- Java is not alone in these types of vulnerabilities
  - Any language which supports dynamic classes has potential exposure

# Trusted Code

- Code loaded from the local filesystem in the JEE trusted locations
- Code signed with a key marked as an alias that exists in the keystore and has a valid certificate in the trusted certificate chain

# JDK Tools

- **Keytool**
  - Create and manage keys in a keystore
  - List keys
  - Mark key as trusted
- **Jarsigner**
  - Sign JARs so they can be associated with a key
- **Policytool**
  - Maintain the policy file
  - Add policy entries

# Summary

- Java was designed from the ground up around internet support
- Security is a major concern with connection to the internet
- Java has many mechanisms to protect against attempts to penetrate the execution environment
- Some security features in the JEE environment will be vendor specific



# Cryptography and JCA

# Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- **Cryptography and JCA**
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL

# Objectives

Understand Public Key Cryptography

Learn how Java implements PKI

Learn how the various PKI pieces fit together

Understand how to implement security in Java

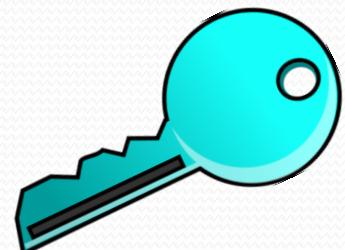
# Internet Insecurity

- Internet is full of untrustworthy sorts that want to steal from you
- Hackers want your bank accounts, business and personal, your identity
- Companies want your company secrets
- Governments (foreign) want your company secrets
- You want to do business on the internet
- Any device connected a network can be compromised
- Intranets are not as safe as they used to be



# Key Generation, Exchange, Encryption

- Keys are the key
- Trust is established using trusted keys
- Your organization gives another organization a key and gets a trusted key in return
- Where does the key come from?
- How do you exchange it?
- How do you keep a secret key *secret* if you have to give it to someone to trust them?
- What about data in transit?
- Led to **Public Key Infrastructure (PKI)**



# PKI

- **Certificate Authority (CA)** - issues Certificates
  - Digitally sign and distribute public key
- **Registration Authority (RA)** - validates registration
- **Digital Certificate** - map public key to entity
- **Private Key** - used to sign
  - Stored in a vault - under lock and key
- **Public Key** - used to verify signer
- Alternate to CA - **Web of Trust**
  - Hand the person the certificate on digital media

# Java Cryptography Architecture

- JCA - How Java handles cryptography
- JCE - While separate, can be thought of as part of JCA
- JCA is like the *API* and JCE is like a vendor-specific *implementation*
- Both are now part of JDK since Java 1.5
- Classes like **SunJCE** act as a provider for JCE
- Java supports PKI certificate exchanges

# JCA Components

- **Provider** - provides the security implementation classes as a package
- **Signature** - signs data
- **MessageDigest** - calculates message digests
- **Cipher** - initialized with keys - encrypt and decrypt
  - Ciphers need algorithms to perform the encrypt/decrypt
  - Providers register cipher and algorithms with the JVM
- **Keys** - stored in keystores
  - Private
  - Public
- **Keystore** - database of keys
- **Certificate** - used to exchange public keys

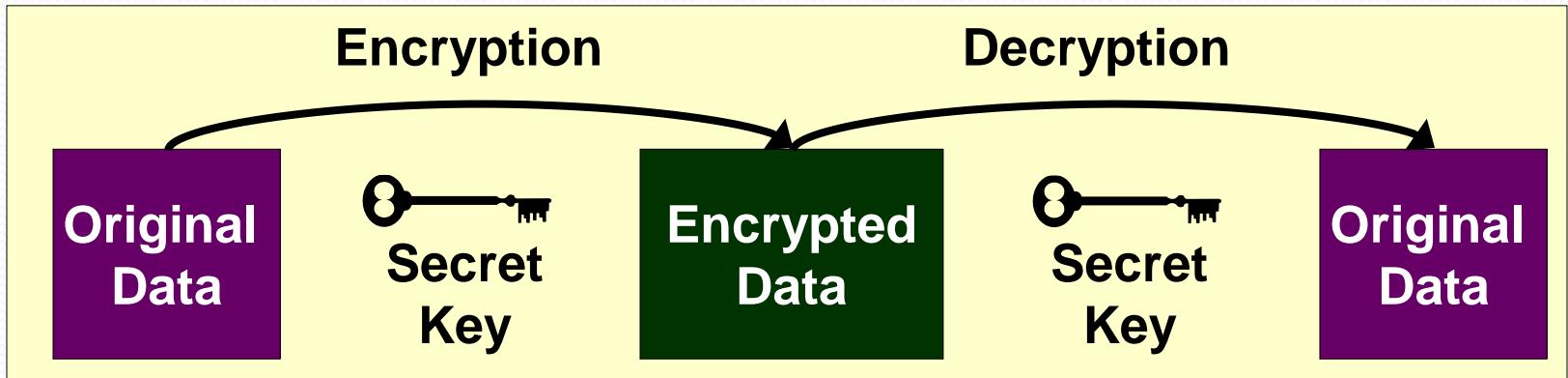
# PKI Elements

- Assure message content is secure
  - Authentication - *I sent it*
  - Confidentiality - *Nobody else can see what is in it*
  - Integrity - *Nobody altered it in transit*
  - Non-repudiation - *You got it; you can't tell me you didn't*
- Public - part of the certificate you give to everybody
- Private - part of the certificate you keep to yourself

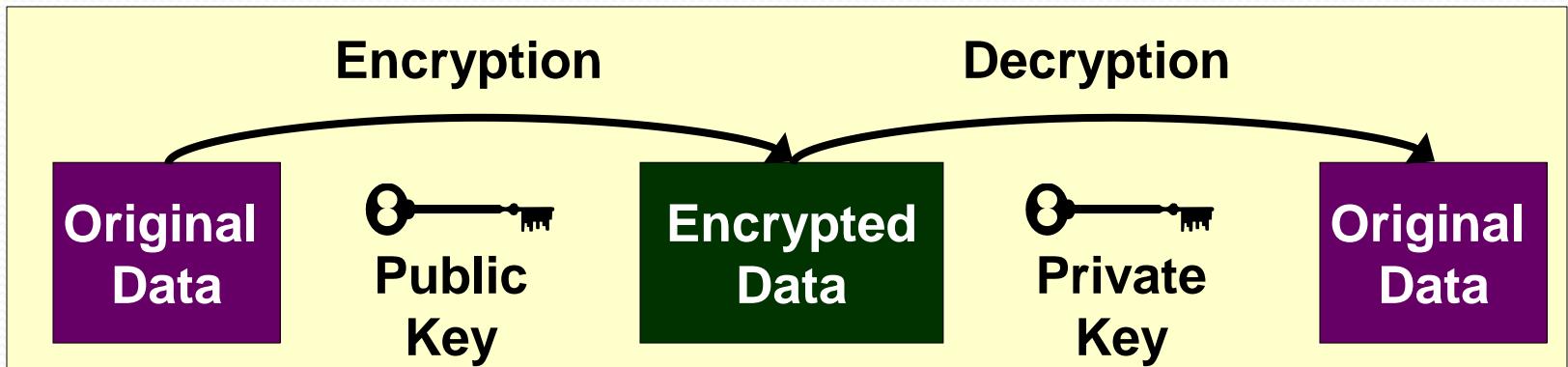
# Keys

- **Secret-key cryptography** - a single key is used for encryption and decryption
  - When a session is created, one party creates a session key and sends it to the other party
  - The same key is used by both parties to encrypt and decrypt
  - When the session is ended, both parties delete the key
  - Highly efficient
- **Public-key cryptography** - a pair of keys are used for encryption and decryption
  - The decryption key is private and the encryption key is public
  - Messages encrypted using the public key can be decrypted only by the owner of the private key
  - Slower than secret key

# Secret-Key and Public-Key Cryptography



Secret Key Cryptography



Public Key Cryptography

# Digital Certificates



- **Digital certificate** - electronic document used for identification
- **Certificate authorities (CAs)** - agencies that verify identities and issue certificates
- Public-key cryptography uses certificates
  - A certificate associates a public key with the entity identified by the certificate
  - Only the public key published in the certificate is compatible with the private key possessed by the owner of the certificate
- A certificate also contains the entity's name, an expiration date, a serial number and the issuing CA's name and digital signature

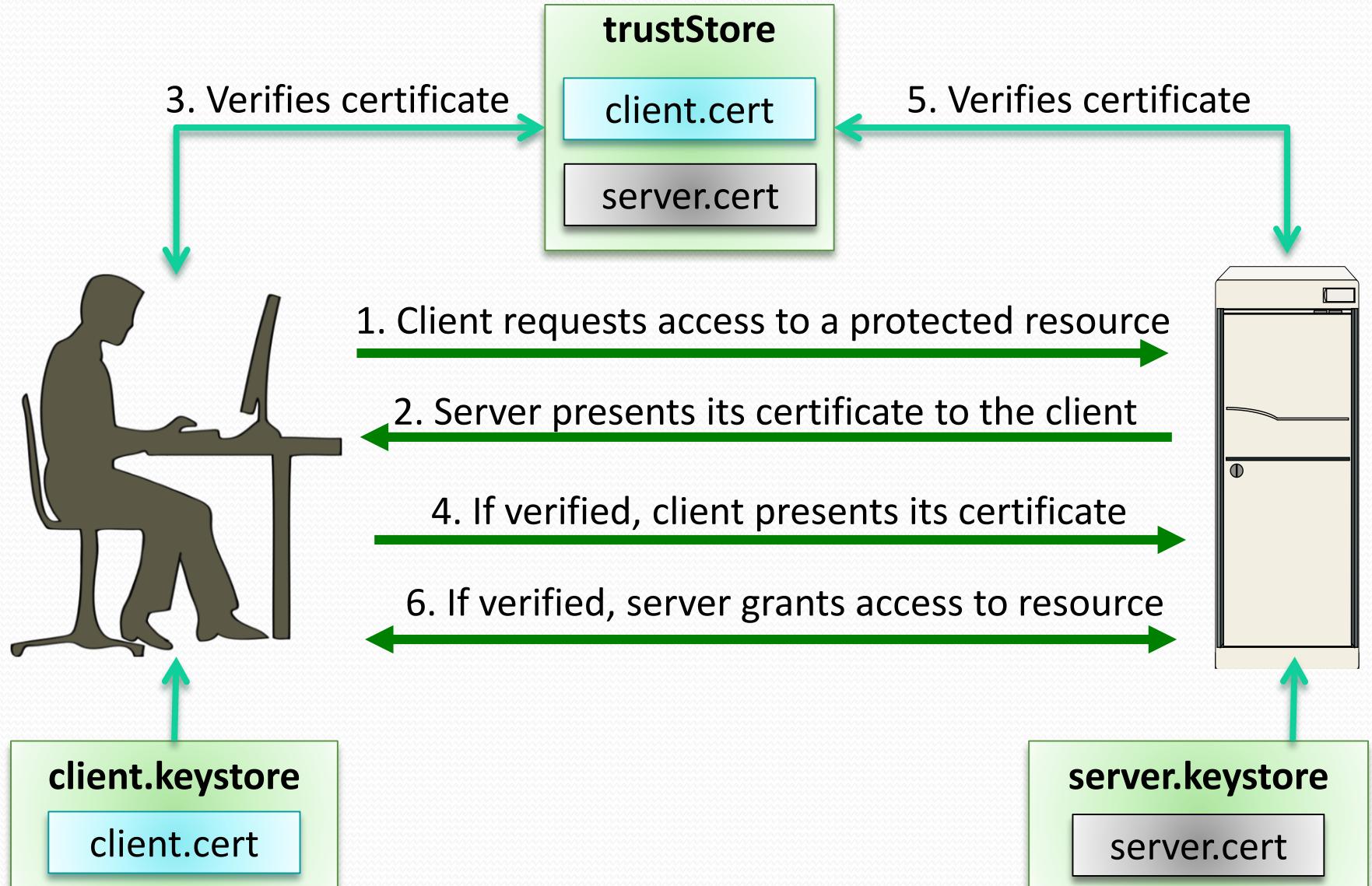
# Self-Signing

- Digital Certificate is signed by the same entity whose identity it certifies
- Not verified by an independent CA
- Pros
  - It's free!
  - Can be easily customized (key size, data, etc.)
- Risks
  - Browser security warning (pop-up dialogs)
  - Cannot be revoked
    - Attackers could gain access to a certificate, and use compromised certificate for spoofing

# Server vs. Mutual Authentication

- Server Authentication
  - Only the server authenticates
  - Proves the identity of the server to the client
    - Server provides Digital Certificate to the client
    - Reduces vulnerability to **MITM** attacks
- Mutual Authentication
  - Server and client authenticate each other
  - Client is subjected to the same strength of authentication from the server (Client must send its Digital Certificate)
    - Provides additional security
    - Private key never leaves the remote party
    - Helps to prevent vulnerability from password theft
      - No password is transmitted

# Mutual Authentication



# Summary

- Java Cryptography API assists developers in providing secure contexts for the application
- Java Cryptography Extension provides additional options for security
- Keys and digests provide a mechanism for ensuring the source of a message is authentic
- No technique is guaranteed at this point in time



# Declarative and Programmatic Security

# Agenda

- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- **Declarative and Programmatic Security**
- Defining Security
- Java Authentication
- SSL

# Objectives

Understand security in JEE

Understand authentication and authorization techniques

Learn how to authenticate a client and server

Learn about authorization techniques

Learn how to protect against some common hacks

# Container Based Security

- Security model is a role-based declarative model
- Resources are protected by roles
- Roles are assigned to users
- Deployment descriptors define security expectations
- Annotations can define security data
- Deployment descriptors can reference annotated contexts

# Security: Declarative/Programmatic

- Declarative
  - Defined in deployment descriptors
  - Web module
  - EJB modules
  - Web Services
- Programmatic
  - Security attributes in code
  - Annotations
  - Direct API invocations

# Application Isolation in JEE

- Cross application access is forbidden
- Resources owned by separate class loaders
- EJBs or Web Services can expose resources

# Security Roles and Role Mapping

- **Role** - abstract name for a permission to access a resource
- **Realm** - a database of users and groups
  - Associated with a specific application or group of applications
- **User** - an identity defined in the Application Server
- **Group** - a set of authenticated users classified by common traits
- **Principal** - an entity that can be authenticated by a realm
- **Role Mapping** - association of a user or group to a role

# Enabling Security

- Administration screen for the application server
- Usually a wizard to guide through enabling all aspects at one time
  - Some settings are dependent on other settings
  - Wizards help to make sure all the relationships are enforced
- Most Application Servers store configurations in XML
  - XML files can be directly edited
  - Properties must be coordinated without much assistance

# Defining Security Roles Permissions

- Specify an authentication method
- Create security roles
- Protect Web resources by defining security constraints
  - Define Web resource collections to identify the resources that require security
  - Specify roles that are authorized to access the protected resources
  - Define network security requirements
- Specify Security (*Run As*) Identities if required

# Custom User Registries

- Some way to identify users and the roles
- JBOSS uses **.properties** files
- WebSphere has a local database
- Simple to create the registry
- Find using JNDI
- Responds to requests for authentication of a user and password
- Returns a Principal with Role mappings

# LDAP User Registry

- Use JNDI to connect to an LDAP registry
- Need to specify the domain and context

```
env.put(Context.SECURITY_PRINCIPAL, "cn=S. User,
ou=NewHires, o=THECOMPANY");
```
- Provide the security credentials to lookup the entities
- Almost all application servers support an LDAP registry
- Microsoft AD supports LDAP as a protocol

# Security Configurations

```
<security-role>
 <role-name>administrator</role-name>
</security-role>
<security-constraint>
 <web-resource-collection>
 <web-resource-name>AdminCollection</web-resource-name>
 <url-pattern>/admin/*</url-pattern>
 <http-method>GET</http-method>
 <http-method>POST</http-method>
 </web-resource-collection>
 <auth-constraint>
 <role-name>administrator</role-name>
 </auth-constraint>
 <user-data-constraint>
 <transport-guarantee>CONFIDENTIAL</transport-guarantee>
 </user-data-constraint>
</security-constraint>
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

# Authentication Mechanisms

- Basic authentication
  - User attempts to access site
  - Site prompts the browser to authenticate
  - Browser presents user with login popup
  - Authentication credentials are automatically sent
- Form based login
  - User attempts to access site
  - Redirected to a page with fields to enter credentials
- Certificate based authentication
  - User attempts to access site
  - Server queries browser for certificate
  - Mutual certificate exists between the two machines - authenticate

# Basic and Form Authentication

- Negotiated at the application layer
- May be transmitted over HTTP or HTTPS
- Easy to fake credentials
  - Only need userid and password
- Often credentials kept in insecure database

# Authentication vs. Authorization

- Authentication - Forces the client to prove its identity
  - HTTP basic authentication
  - SSL mutual authentication
  - Form based authentication
- Authorization - Determines the resources the client is allowed to access, i.e. Role
  - All users may be able to browse a store catalog
  - Only authenticated users may be allowed to make online orders
  - Only certain authenticated users may be able to edit corporate account information

# Lazy Authentication

- There is a cost associated with authentication
- Lazy authentication means performing authentication only when required
- Users are not required to authenticate until they request access to a protected resource
  - Example: Customers can browse an online store but are prompted to authenticate when they go to check out

# Single Sign-on Properties

- Each server involved may or may not use the same user registry and authentication mechanism
- Users are authenticated once
- User credentials are associated with a session
- The container uses the credentials to establish a security context, which is used to determine authorization
- Authorized users can access resources on different servers within the same domain without authenticating themselves again

# LTPA - IBM

- Authentication performed using one of the following:
  - User ID and password
  - LTPA certificate (token)
- The LDAP registry is used to verify the user information

# Secure Socket Layer in a Nutshell

- Problem: Anyone can spy on internet traffic as it goes by
- Solution: Scramble it so nobody can spy on it
- Conversation startup
  - Client sends key
  - Server gets key
  - Server sends key
  - Client gets key
  - Both sides agree that all is good
- Conversation then can continue
  - Client digitally encrypts the content to send
  - Server digitally decrypts the content before processing
  - Server digitally encrypts the response content
  - Client digitally decrypts the response content



# SSL Handshake

## SSL/TLS Key Exchange

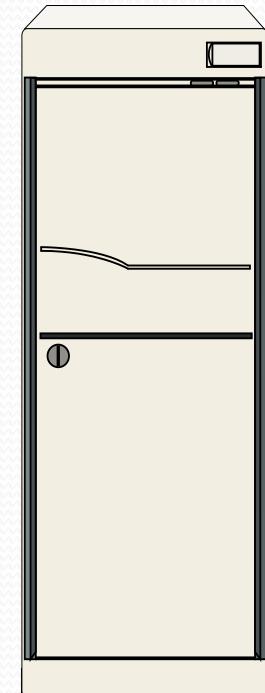


1. Client sends its SSL version number, cipher settings, etc.  

2. Server sends its SSL version number, cipher settings, digital certificate, etc.  

3. Client uses certificate to authenticate server.  

4. Session keys are generated to encrypt data.



# GSS

- **Generic Security Services**
- Providers of GSS are registered
- Provide GSS APIs to isolate the code from the security infrastructure
- Deals only with authentication, not authorization
- Can discover new security mechanisms using SPNEGO
- Services run over Kerberos

# SASL Protocols

- **Simple Authentication Security Layer (SASL)**
- Made available to Java using **libsasl**
- **libsasl** is a native implementation that sits between Java and the SASL providers
- Used by IMAP, LDAP to exchange authentication information between client and server
- Similar to GSS and JSSE but more lightweight
- Use SASL when the protocols are defined to use it

# Java Secure Socket Extension

- 100% Pure Java
- Support for SSL 2.0, SSL 3.0 and TLS 1.0 and later
- Provides SSL sockets
- Provides several cipher suites and negotiation for SSL handshaking
- Oracle implementation is trivial to use for HTTPS
- Supports most common digests and ciphers
- Supports X509 certificates and tools to manage keys
- **SSL/TLS Server Name Indication (SNI) Extension**

# Establishing SSL on App Servers

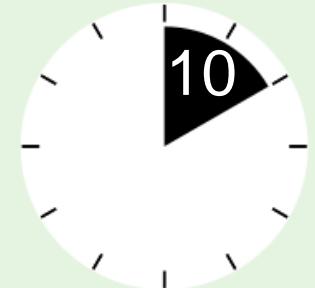
- Use an administration screen to turn on SSL support
- Defining the location of keystores
- Add keys with an alias to the keystore
- Select option to enable trust
- Usually can import a Certificate Chain

# Summary

- Java has many ways to provide to secure client/server conversations
  - GSS
  - JSSE
  - SASL
- Each has advantages
- Choice is often made by the server side when implementing a client
- **Continue to Lab Introduction on next slide**



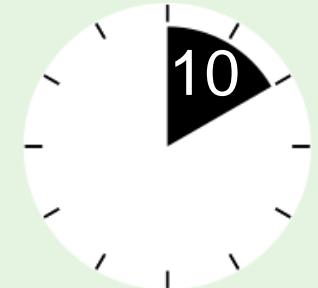
# Lab 10 Introduction



- **Secure a Web Application - Part II**
- Enable SSL
  - **keystore.jks**
  - **server.xml** for SSL settings
- Force the servlets to be only accessible using SSL
  - **@HTTPConstraint** set to **TransportGuarantee.CONFIDENTIAL**
- After the lab, we will have a lab review

# Lab 10 Review - Discussion

- **Secure a Web Application - Part II**
- Enable SSL
  - **keystore.jks**
  - **server.xml** for SSL settings
- Redirect client to https
  - **@HTTPConstraint** set to **TransportGuarantee .CONFIDENTIAL**
- Code review (provided solution and participant solutions)
- What did you learn? What are your take-aways?
- Issues, Questions, Comments?

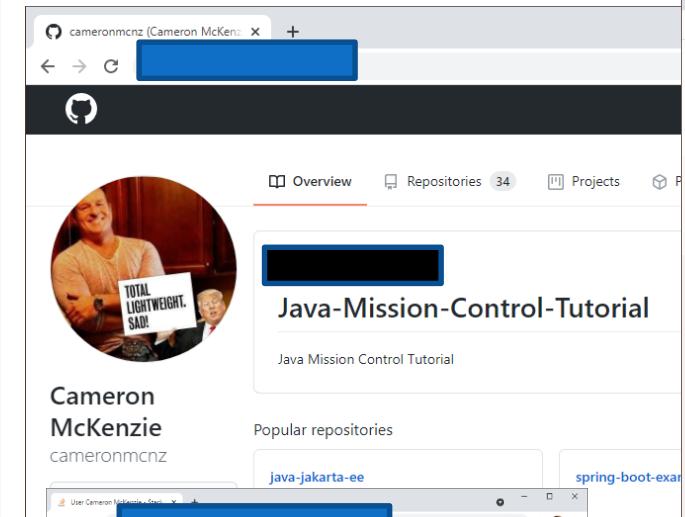


Welcome to the class. I'm Cameron McKenzie



# About Me

I'm Cameron McKenzie



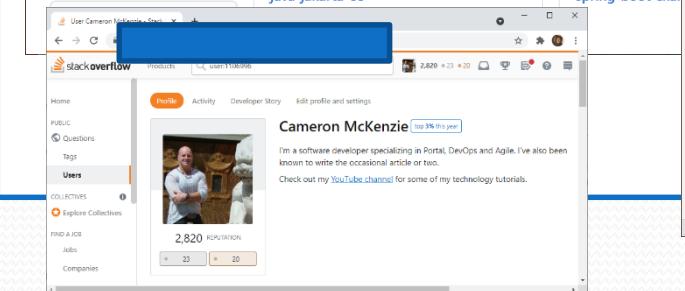
cameronmcnz (Cameron McKenzie) Overview Repositories 34 Projects

 Cameron McKenzie TOTAL LIGHTWEIGHT. SAD!

**Java-Mission-Control-Tutorial**  
Java Mission Control Tutorial

Popular repositories

- java-jakarta-ee
- spring-boot-exam



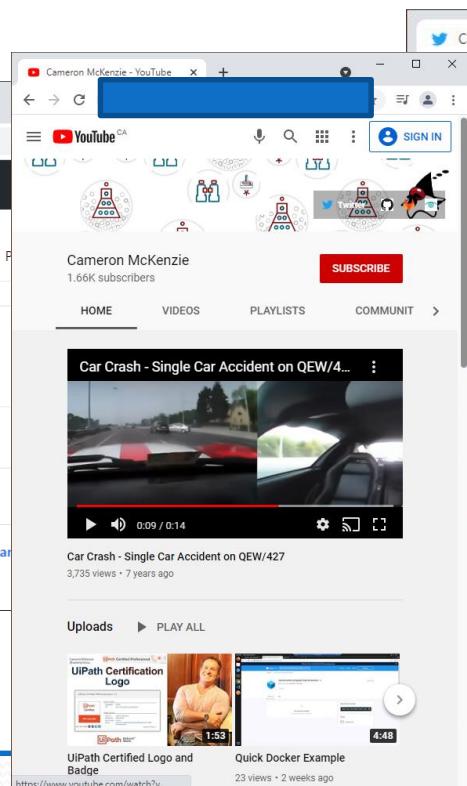
User Cameron McKenzie - Home Questions Tags Users EXPLORE COLLECTIVES FIND A JOB Jobs COMPANIES

Activity Developer Story Edit profile and settings

**Cameron McKenzie** top 3% this year

I'm a software developer specializing in Portal, DevOps and Agile. I've also been known to write the occasional article or two. Check out my [YouTube channel](#) for some of my technology tutorials.

2,820 REPUTATION 23 20



YouTube CA Cameron McKenzie - YouTube 1.66K subscribers SUBSCRIBE

HOME VIDEOS PLAYLISTS COMMUNITY

**Car Crash - Single Car Accident on QEW/4...** 0:09 / 0:14

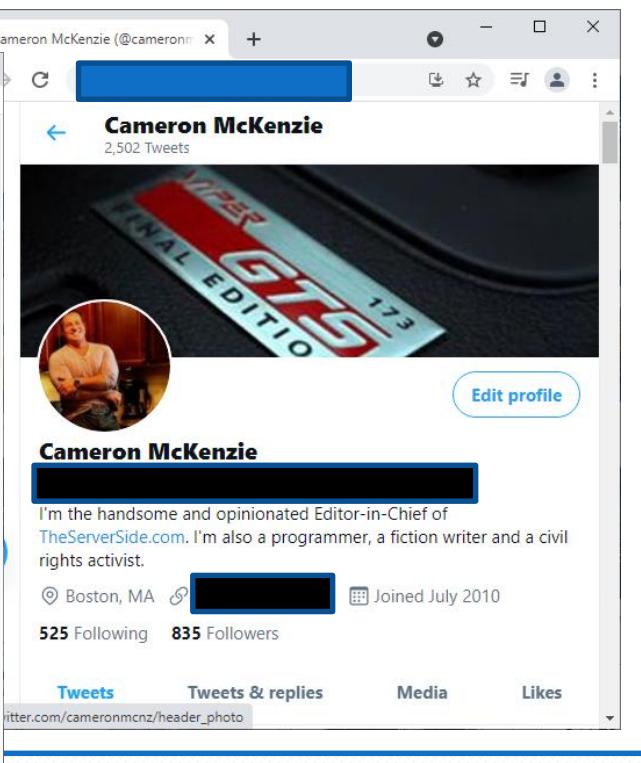
Car Crash - Single Car Accident on QEW/427 3,735 views • 7 years ago

Uploads PLAY ALL

**UiPath Certification Logo** 1:53

UiPath Certified Logo and Badge https://www.youtube.com/watch?v=...

**Quick Docker Example** 4:48



Cameron McKenzie (@cameronmcnz) + 2,502 Tweets

 Cameron McKenzie FINAL EDITION 173

**Cameron McKenzie** Edit profile

I'm the handsome and opinionated Editor-in-Chief of [TheServerSide.com](#). I'm also a programmer, a fiction writer and a civil rights activist.

Boston, MA Joined July 2010

525 Following 835 Followers

Tweets Tweets & replies Media Likes

# Agenda

- Course Introduction
- Java Virtual Machine
- Packaging Applications
- Best Practices for Exception Handling
- Threading and Concurrency
- Networking
- Advanced JDBC
- Performance
- Writing Effective Java
- Data Structures
- Internationalization
- Java Security
- Cryptography and JCA
- Declarative and Programmatic Security
- Defining Security
- Java Authentication
- SSL
- Course Wrap-up

# Questions?

- Do you have any questions?
- Are there any topics you'd like me to review?
- Would you like more depth on any topic?
- Is there a related topic you'd like to know about?



# Resources

Excellent Web sites to visit for the latest Java technology information:

- <http://www.oracle.com/us/technologies/java/overview/index.html>
  - Java home page on Oracle
- <http://eclipse.org/documentation/>
  - Eclipse documentation
- <http://www.eclipse.org/downloads/>
  - Eclipse downloads
- <http://www.ibm.com/developerworks/rational/products/rad/>
  - Rational Application Developer Works
- <http://www.redbooks.ibm.com/>
  - IBM Redbooks and other publications

# References

The Java information and references contained within this course are based upon the Java APIs from Oracle Corporation.

<http://www.oracle.com>

Information about the Eclipse IDE is based, in part, on the documentation from The Eclipse Foundation.

<http://www.eclipse.org/>

# Course Evaluations

- If you haven't already completed the course evaluation, please give us feedback regarding your impressions of this workshop:
  - Instructor
  - Materials
  - Equipment and facilities
  - or Virtual Experience
- We appreciate your feedback. Thank you!





This slide has been intentionally left blank.