

As promised, here are the slides for the Java 1Z0-815 exam course.

The course only went live today (December 20th, 2019), and I am quickly finishing off the code snippets and quizzes and this document.

As you can see, this is a big document, over 500 pages! That's a lot of slides! If you haven't already, check out my other Java courses.

[Java Masterclass](#)

[Data Structures and Algorithms](#)

[Java Design Patterns](#)

[Java Spring Tutorial](#)

[Learn Java Programming Crash Course](#)

[Java Enterprise Edition 8](#)

[Advanced Algorithms in Java](#)

[Search Algorithms in Java](#)

## **Current version**

The current version of this slides document is v1.006 created on 20th December 2019

## **Programming Tips and Career advice**

I have a Youtube channel that I post regular videos to. In the past three months there have been 100 programming tips and career advice related videos released. Having been a programmer for 35 years I know a thing or two about programming and the industry so check out my playlist of [Programming Tips and Career Advice Videos.](#)

## Summary

Well that's it for me. If you want to drop me a line you can contact me at my blog. Visit [My Personal Blog here.](#)

I hope you enjoy the course and these slides.

Regards

Tim Buchalka



## OCP Java SE 11 Developer Certification Part 1 - 1Z0-815 Slides

# Main Course Slides.

---

OCP JAVA SE 11 DEVELOPER CERTIFICATION PART 1 - 1Z0-815

OCP Java SE 11 Developer Certification Part 1 - 1Z0-815 Slides



# Course Overview

---

- Course Purpose: To Pass Oracles Java 11 Programmer exam - 1Z0-815.
- Java 8 had Oracle Certified Associate (OCA) and Oracle Certified Professional (OCP) Certification - each certification required a separate exam to be passed.
- To become an Java 8 Oracle Certified Professional needed you to pass both exams.
- With Java 11, Oracle have abolished the Oracle Certified Associate certification and have only the one certification - Oracle Certified Professional: Java SE 11 Developer - to obtain this you need to pass two exams (same as Java 8 Oracle Certified Professional needed both exams passed).

# Course Overview

---

- This course focuses on the first exam, the 1Z0-815.
- In 2020 I'll be releasing a course covering the second exam, the 1Z0-816.
- This course covers 100% of those topics!
- Passes on the exact skills you need to pass the exam.

# Course Overview

---

- Course combines videos, slides and quizzes.
- Source code and slides are downloadable.
- No fluff- videos show code pasted in and explained rather than wasting your time watching me type it!
- All code snippets and completed source code available for you to download so you can follow along.

# Course Pre-requisites

---

- Been through part or all of my Java Masterclass (or another Java course).

OR

- You have some professional programming experience in Java or another language.
- No need to be a Java expert to go through this course!

# Java Tools

---

- I'd suggest you use IntelliJ Community edition (free) and Amazon Corretto JDK 11 (also free) - video in the extras section give my reasons.
- Links to various installation videos on Youtube if you need them.
- Extra's section towards the end of the course has full details.
- Portions of the course use the command line - to simulate real exam conditions.

# The Java Basics

---

- The basics of creating an executable Java program.
- Talk about the main class.
- Compiling and running from the command line.
- Using Jar to create executable jar files and so on.

So I want to get that all in one hit in this long video, but a lot of the future videos you'll find will be fairly short and compact.

# Java Development Environment

---

I'm assuming you do have some development experience in Java - either by going through a Java course like my Java Masterclass or actually professional programming experience.

There's an extra section, right towards the end of the course, and there's videos there on:

- Setting up IntelliJ.
- Setting up JDK, version 11 which were using in this course.
- And also importantly setting up so that Java and the various build tools, Java Compiler, Jdeps and a few other things, that they all actually work from the command line.

# Using both an IDE and the Command Line

---

You may encounter some exam questions about compiling and running a Java class from the command line, which we'll discuss using a terminal session from inside IntelliJ.

# Java Programs and Utilities

---

Before we proceed, let's talk about some Java tools which are integrated into the Integrated Development Environment, but which we'll also be using a few of manually.

The exam may not test you exclusively on these, but Oracle expects you to know what is available and some of the more common options.

# Main Java Tools

---

The following table shows the Main Tools to Create and Build Applications: You'll note that JShell is not part of this list, it is not considered a 'main tool'.

We won't be talking about JShell which was introduced in JDK 9 for this course, but you'll note that the JShell Console is available to you in IntelliJ under the Tools menu.

Tool	Purpose	Some common options
java	Launch a Java application.	<code>--class-path</code> , <code>-classpath</code> , <code>-cp</code> (parameter is a semicolon ( ; ) separated list of directories, JAR archives, and ZIP archives to search for class files.).  <code>-verbose:class</code> (Displays information about each loaded class)  <code>-verbose:gc</code> (Displays information about each garbage collection (GC) event).

# Main Java Tools

---

Tool	Purpose	Some common options
javac	Read Java class and interface definitions and compile them into bytecode and class files.	<ul style="list-style-type: none"><li>-d (Sets the destination directory for class files).</li><li>-g (Generates all debugging information).</li><li>-g:[lines, vars, source] (Generates only the kinds of debugging information specified by the comma-separated list of keywords).</li><li>-g:none (Doesn't generate debugging information).</li><li>-verbose (Outputs messages about what the compiler is doing).</li></ul>

# Main Java Tools

---

Tool	Purpose	Some common options
jar	Create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.	<code>-c, --create</code> (Creates the archive). <code>-C</code> (Changes the specified directory and includes the <code>files</code> specified at the end of the command line). <code>-f, --file</code> (Specifies the archive file name). <code>-m, --manifest</code> (Includes the manifest information from the given manifest file).
javap	Disassemble one or more class files.	<code>-c</code> (Prints disassembled code, for example, the instructions that comprise the Java bytecodes, for each of the methods in the class.).

# Main Java Tools

---

Tool	Purpose	Some common options
javadoc	Generate HTML pages of API documentation from Java source files.	Usage not part of first exam.
jlink	Assemble and optimize a set of modules and their dependencies into a custom runtime image.	Usage not part of first exam.
jmod	Create JMOD files and list the content of existing JMOD files.	Usage not part of first exam.
jdeps	Launch the Java class dependency analyzer.	Discussed in Modules videos.
jdeprscan	Static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.	Discussed in Modules videos.

# The Main Method Signature

---

I've included a slide with some of the many surprising variations on this required method a bit later on, but first we'll show you some of the more common declarations.

We'll be discussing the modifiers in a later video, but it is important to note that if you want the class to be executable, the Java Virtual Machine will look for a main method that is both public and static.

Removing either of these modifiers means you will no longer have an executable class.

Also, the method signature, the name ('main') and its parameters (String[]) must match.

# Main Methods Declarations

---

Here is a slide showing the valid main method declarations in Java - some are a little tricky, and it's possible, you might see an obscure declaration in the exam.

For all declarations:

- public and static modifiers are both required.
- return type is always void.

# Main Methods Declarations

Point to remember	Code Samples
The keywords public and static can be in any order but both are mandatory.	<pre>public static void main(String[] args) {}</pre> <pre>static public void main(String[] args) {}</pre>
Other modifiers are allowed, these include final, strictfp, synchronized in any order, in any combination.	<pre>public final static void main(String[] args) {}</pre> <pre>public final strictfp static void main(String[] args) {}</pre> <pre>public final strictfp synchronized static void main(String[] args) {}</pre> <pre>public final static synchronized strictfp void main (String[] args) {}</pre>

# Main Methods Declarations

Point to remember	Code Samples
The parameter of the main method can take the var-args syntax as well as all valid array syntax.	<pre>public static void main(String[] args) {}  public static void main(String args[]) {}  public static void main(String []args) {}  public static void main(String... args) {}  public static void main(String...args) {} //no space after ellipsis</pre>
The parameter of the main method can be declared final.	<pre>public static void main(final String[] args) {}</pre>

# Main Methods Declarations

Point to remember	Code Samples
The parameter of the main method can be named anything.	<pre>public static void main(String... arguments) {}  public static void main(String[] stuff) {}  public static void main(String[] vars) {}  public static void main(String ARGs[]) {}</pre>
The method can declare a throws clause for unhandled exceptions of any kind.	<pre>public static void main(String[] args) throws Exception {}  public static void main(String[] args) throws RuntimeException {}  public static void main(String[] args) throws Throwable {}  public static void main(String[] args) throws Error {}</pre>

# Summary

---

In this video we have reviewed the requirements for executing a simple Java program as follows:

- The structure of the source code must meet these minimum requirements:
  - The name of the class which has the main method doesn't matter.
  - The access modifier of the class does not matter - in the case of a class you can define it as 'public' or use the default modifier.
  - If you do declare the class public, you must name the file the same as the class name with a .java extension.

# Summary

---

- The number of classes in the file does not matter - you can define as many classes as you want in a single file, but only one can be public, and the filename must be the same as the public class.
- When you compile the file, the compiler will create a separate .class file for each defined class.
- You can define a main method in any or all of the classes.

# Summary

---

- The main method of the class where your application starts must have the signature: `public static void main(String[] args)`
- From Java's point of view these signatures are equivalent for the JVM to execute the main method:
  - `public static void main(String... args);`
  - `public static void main(String args[]) throws Exception;`

# Summary

---

- You can execute code in the following ways:
  - launch a single source-file program
    - `java [options] source-file [args...]`
  - To launch a class file either in a directory or a jar file
    - `java [options] mainclass [args...]`
  - launch the main class in a JAR file
    - `java [options] -jar jarfile [args...]`

# Summary

---

- Executing a class in a module (which we'll discuss in a later video)
  - `java [options] --module module[/mainclass] [args...]`
  - `java [options] -m module[/mainclass] [args...]`
- You can pass data on the command line to your application as a space delimited set of strings. These are the `String[] args` represented by the parameter of `main`.

# Packages

---

As per the Oracle Java Documentation: "A package is a namespace that organizes a set of related classes and interfaces.

Conceptually you can think of packages as being similar to different folders on your computer."

# Create Packages Using Package Statement in your Class

Now that we've created a class in a package, we next need to learn how to use it.

We've seen that to execute it on the command line, we provided a FQCN (Fully Qualified Class Name).

This is one option we can use to reference another class in another package, or we can use an import statement in our code.

# Reference Methods

---

Referencing a type from another package.

Reference Method	Example Code	When to use
<b>A Fully Qualified Class Name FQCN</b>	<code>java.util.ArrayList = new java.util.ArrayList();</code>	If you are only referencing a class once in source, or you are referencing two classes with the same name but from different packages, you will still need to use the Fully Qualified Class Name. It can be very verbose.
<b>A single-type-import declaration</b>	<code>import java.util.ArrayList; ... ArrayList aList= new ArrayList();</code>	This is the most common way for both readability of code and information for future developers.

# Reference Methods

---

Reference Method	Example Code	When to use
<b>A type-import-on-demand declaration</b>	import java.util.*; ... ArrayList aList= new ArrayList();	Discouraged but still needed to be recognized as valid for the exam. If you use this for all your import statements, anyone reading the code after you may not know where the class referenced actually resides. Of course, IDEs will fix this pretty quick. Older code, written before IDEs, are more likely to use this type of import statement;

# Reference Methods

---

Reference Method	Example Code	When to use
<b>A single-static-import declaration</b>	<code>import static java.lang.Math.PI;</code>	This is a way to pull in a single static member of a class in and reference the member's name without including a reference to the declared type.
<b>A static-import-on-demand declaration</b>	<code>import static java.lang.Math.*</code>	This is a way to pull in ALL static members (variables and methods) of a class and reference them without referencing the declared type.

# Using Import Statements in your Class

To use a class in your code which resides in another package, even a parent package, you must use one of the mechanisms I just described, a Fully Qualified Class Name (which can get tedious) or one flavor of an import statement.

For the following exercises, I'm going to make IntelliJ less intelligent.

IntelliJ and most IDEs will handle most of the work of maintaining, the correct import statements required by your code, and remove unused or redundant statements.

# Using Import Statements in your Class

When would you use one or the other.

For the most part, you will always use a single-type-import declaration and your IDE, would be set to automatically include the import statement.

However, it is important to know that the single-type-import declaration:

- Takes precedence of the import-on-demand declaration.
- Will prevent you from creating a class of the same name, in your source file.

# Static Import Statements

---

The next two import statements (A single-static-import declaration, A static-import-on-demand declaration) were introduced in Java 7, and are specific only to static methods and static class variables.

I will discuss static methods and static class variables in future videos but for our purposes here, we'll use the static class variables defined on the `java.lang.Math` class.

# Package and Import Summary

---

So to summarize the last 3 videos:

- The package statement is used to group classes into a logical set of classes, that have some commonality.
  - Only one package can be defined in a .java source file.
  - The package statement must be the first line of code (excluding comments and empty lines).
  - No package statement makes the class, or type you define, belong to the unnamed package.

# Package and Import Summary

---

- To have the compiler create class files in exploded directories, that represent your package names: use **javac -d {directory} foo.java**, where {directory} represents the directory you want the package directories to be created ({} means current working directory).
- A class file does NOT have to reside in a directory structure, that represents its package structure.
- To reference a class in a package, you must:
  - Use a Fully Qualified Class Name (FQCN).
  - Or include an import statement for the package class name.

# Package and Import Summary

---

- Import statements are used to inform the compiler, where the classes outside of the current package, are defined while allowing easier readability.
- You could write all code without a single import statement, using just fully qualified class names, but it is harder to read.
  - You can have 0 to many import statements.
  - Import statement(s) must follow any package statement, and precede any other code, excluding comments and empty lines.

# Package and Import Summary

---

- Required if not using a Fully Qualified Class Name (FQCN), such as "java.util.ArrayList".
- You can not just use the simplified "ArrayList" as a type if you do not have an import statement.
- `java.lang.{classname}` is implicit whenever you use any `java.lang` class.
- The other two import statements, prefixed with 'import static', actually don't import classes at all, but static class attributes or methods.

# Static Imports and Packages: Out of the Ordinary Concepts

---

Items to look for on the exam which should be low hanging fruit if you do come across them:

- Multiple package statements.
- A package statement that is not the first line of code.
- Import statement not in the correct order, must be after a package statement if one exist and before any other code.
- Import static statement used in place of an import statement.

# Static Imports and Packages: Out of the Ordinary Concepts

---

- Import statement using a higher level package with a wild card - **\***, a wild card does not include classes in other packages. Example. `import a.*` does not also mean `import a.b.*` for example.
- A misplaced wildcard, for example the **\*** at the end of a class name or a partial class name.

# Understanding Java Technology and Environment

---

Let's start the course with the question 'Why Java?', because maybe someone has asked you this question in your past and you didn't have the best response.

The responses to 'Why Java?' are the key features which you may be tested on. These are:

- Java is derived from C, C++ but simplified.
- Java is Architecture neutral.
- Java is Object Oriented.
- Java is Statically typed.
- Java is a dynamic programming language.

# Understanding Java Technology and Environment

---

- Java supports Multi-threaded processing.
- Distributed Computing is supported by Java.
- Memory management is handled by the Java environment, and not the developer.

# Java is Derived from C, C++ but Simplified

Why Java? We might ask the original developers.

Java was initially started to enhance the C++ language, which is the object oriented extension of the C language.

C and C++ languages are compiled into machine specific assembly language, but Java was designed as a write once run anywhere language, since it's initial goal was to be deployed on multiple non-pc devices.

Java code is compiled into byte code, which is interpreted by the Java virtual machine, and then translated, to execute on the machine it is being hosted on.

# Java is Derived from C, C++ but Simplified

The architects of Java eliminated more complicated features such as pointers, operator overloading, multiple class inheritance, and destructors.

They added garbage collection for automated memory management, and the ability to write a multi-threaded application, as well as library support for network programming.

# Java is Architecture Neutral

---

Architecture neutral is another way of saying platform independent.

An application developer can write Java code without knowing or caring, what hardware platform(s) the application will be deployed to.

Java compiles source code into Java byte code, which is interpreted by the Java Virtual Machine, using just-in-time compilation to optimize and run, the application on the target machine.

# Java is Object Oriented

Java is a class based Object Oriented Language.

From an Object Oriented theorist's point of view, Java is not a pure Object Oriented language, since it supports some primitive data types for optimization of performance.

Java allows precise access control of class members, supporting encapsulation of data.

You can hide both data and implementation details from other classes, which use or extend the class you create.

Java supports dynamic binding, which delays the determination of the code, to be executed for a specific method until run-time, which supports polymorphism.

# Java is Object Oriented

Java supports both overloaded and overridden methods.

When you override a method, you are using polymorphism, declaring that the class has its own custom behavior for a particular method or additional behavior to its extended class, and it is this behavior that will execute at runtime, when the object's run time type is fully known.

Dynamic binding is also known as late binding or runtime binding.

Static binding is binding that can be known at compile time, and is also known as early binding.

# Java is Object Oriented

---

Java supports both the Inheritance feature of an Object Oriented language, by allowing classes to extend or be subtypes of other classes - this is the 'IS A' feature of Object Orientation.

Java also supports the composition feature of an Object Oriented language, the 'HAS A' feature, by allowing class data to be of any other defined class type.

It is important to note that Java supports a Single Inheritance Class Structure, a class can only extend one class, or only have a single parent class in its hierarchy.

# Types of Inheritance

---

Type of Inheritance	Java Supports Multiple Inheritance
Multiple Inheritance of State	NO - State is inherited from a supertype and classes can only have one supertype.
Multiple Inheritance of Implementation	NO - Here implementation means inheriting method definitions from a supertype and classes in Java can only have one supertype.
Multiple Inheritance of Type	YES - Java allows classes to extend its type as well as implement other types.

# Java is Statically Typed

---

Java is a statically typed language which requires you to declare the data types of your variables before you use them, while dynamically-typed languages do not.

Statically typed languages perform type checking at compile time.

Dynamically typed languages are checked at run time.

Every variable in Java must be declared with a type.

Java is considered more strongly typed than C and C++, because it eliminated pointer arithmetic.

But it is considered not as strongly typed as some other languages, because it will perform some implicit type conversions, in order to cut down programmer effort.

# Java is a Dynamic Programming Language

A dynamic programming language is an extensible language which proves the ability to:

- Add new code.
- Extend objects and definitions.
- Modify the type system.

# Java is a Dynamic Programming Language

We already know that Java is a statically typed language, but it is still considered a dynamic programming language because of the following features:

- The environment is dynamically extensible, whereby classes are loaded on the fly as required.
- The language supports reflection, which allows objects to be queried at runtime, and manipulated.
- Java supports class extension and inheritance, allowing behavior to be altered at runtime .
- Low level operations are dynamically executed by the Java Virtual Machine.

# Java Supports Multi-threaded Processing

---

Java supports multi threaded execution, however it is NOT automatic.

From the application programmer's point of view, you start with just one thread, called the main thread.

This thread has the ability to create additional threads, through programmatic constructs.

# Distributed Computing is Supported by Java

---

In addition to the Java Virtual Machine being portable, to many hardware platforms across a distributed environment, Java has many features which support communication across distributed systems.

Java contains extensive TCP/IP networking facilities.

Library routines support protocols, such as HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP).

## Memory Management is Handled by the Java Environment and not the Developer

Java was written to take the work of memory management out of the hands of the developer, who no longer has to write destructors to destroy and deallocate objects.

Instead, Java provides a garbage collection mechanism.

The Oracle documentation describes automatic garbage collection, as the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects (the underlying memory is reclaimed and reused for future object allocation).

## Memory Management is Handled by the Java Environment and not the Developer

An in use object, or a referenced object, means that some part of your program still maintains a pointer or reference to that object.

You will need to know when an object is considered no longer referenced for the exam.

In the Java platform, there are currently four supported garbage collection alternatives, and all but one of them, the serial collector, execute the work in parallel to improve performance.

We'll discuss garbage collection in a later video.

## Memory Management is Handled by the Java Environment and not the Developer

For 24-7 applications, or applications which manipulate large data sets, Java's garbage collection could have a detrimental effect on an application's performance, since garbage collection could tie up much needed resources while it is executing.

In these environments, you may need to get more familiar with the different types of garbage collectors available, in the Java Virtual Machine, to tune the right collector for your type of application environment.

# Java Environment

---

Finally, we'll discuss some brief points about the Java Environment:

- The Java Development Kit (JDK) includes the development tools, the compiler, the debugger, class inspector and documentation generator, as well as a Java Runtime Environment to run applications.
- The Java Runtime Environment (JRE) includes only the class libraries and executables required, to run a Java program.
- A Java application does not need the Java Development Kit to run.

# Java Environment

---

- Java is Write once Run Anywhere (WORA) which means it is Platform Independent, and Portable Java source code is compiled into Java bytecode class files, which are then interpreted by the Java Virtual Machine, the JVM, which is available for diverse platforms.
- The Java Virtual Machine interacts with the Operating System, interpreting the class files, and performing Just in Time Compilation to specific machine code, optimizing the code for the environment it is executing on.
- A Java program never really executes natively on the host machine.

# Working With Java Primitive Data Types and String APIs

- Declare and initialize variables (including casting and promoting primitive data types)
- Identify the scope of variable
- Use local variable type inference
- Create and manipulate Strings
- Manipulate data using the StringBuilder class and its methods

WORKING WITH JAVA PRIMITIVE DATA TYPES AND STRING API'S  
JSE11DCP1-1-1

# Primitive Data Type Recap

---

Java supports two types of data, reference data types and primitive data types.

The data value of a primitive data type is stored in memory and is not a location reference to the data.

This makes accessing primitive data types faster and leads to more efficient performance when it really matters.

Unlike C#, these are not objects.

Because Java allows primitive data types which do not inherit from 'Object' like every other instance, Java is not considered to be a 'pure' object oriented language.

# Primitive Data Type Recap

---

There are 8 primitive data types as shown on the slide. The first five are represented in two's complement format, which is important, especially for the 4 signed primitives (only the char is unsigned), where the left most bit represents the sign (negative, positive).

Primitive Data Type	Storage in bits (width)	Range of Values	Wrapper
<b>byte</b>	<b>8</b>	<b>-128:127</b>	Byte
<b>char</b>	<b>16</b>	<b>\u0000:\uFFFF</b>	Character
<b>short</b>	<b>16</b>	<b>-32768:32767 (-2^15) - (2^15)-1)</b>	Short
<b>int*</b>	<b>32</b>	<b>(-2^31) - (2^31)-1)</b>	Int
<b>long</b>	<b>64</b>	<b>(-2^63) - (2^63)-1)</b>	Long
<b>float</b>	<b>32</b>	approx 3.4028235E38	Float
<b>double*</b>	<b>64</b>	approx 1.7976931348623157E308	Double
<b>boolean</b>	<b>1</b>	<b>true,false</b>	Boolean

# Primitive Data Type Recap

---

Float and Double are floating point representations and approximations, and will not overflow (more on this later) - these are used when precision after the decimal point is critical.

Primitive Data Type	Storage in bits (width)	Range of Values	Wrapper
<b>byte</b>	<b>8</b>	<b>-128:127</b>	Byte
<b>char</b>	<b>16</b>	<b>\u0000:\uFFFF</b>	Character
<b>short</b>	<b>16</b>	<b>-32768:32767 (-2^15) - (2^15)-1)</b>	Short
<b>int*</b>	<b>32</b>	<b>(-2^31) - (2^31)-1)</b>	Int
<b>long</b>	<b>64</b>	<b>(-2^63) - (2^63)-1)</b>	Long
<b>float</b>	<b>32</b>	approx 3.4028235E38	Float
<b>double*</b>	<b>64</b>	approx 1.7976931348623157E308	Double
<b>boolean</b>	<b>1</b>	<b>true,false</b>	Boolean

# Primitive Data Type Recap

---

Note that Java supports a wrapper object ( a true reference data type) for each primitive data type.

Primitive Data Type	Storage in bits (width)	Range of Values	Wrapper
<b>byte</b>	<b>8</b>	<b>-128:127</b>	Byte
<b>char</b>	<b>16</b>	<b>\u0000:\uFFFF</b>	Character
<b>short</b>	<b>16</b>	<b>-32768:32767 (-2^15) - (2^15)-1)</b>	Short
<b>int*</b>	<b>32</b>	<b>(-2^31) - (2^31)-1)</b>	Int
<b>long</b>	<b>64</b>	<b>(-2^63) - (2^63)-1)</b>	Long
<b>float</b>	<b>32</b>	approx 3.4028235E38	Float
<b>double*</b>	<b>64</b>	approx 1.7976931348623157E308	Double
<b>boolean</b>	<b>1</b>	<b>true,false</b>	Boolean

# Declaring Primitive Types

---

A declaration consists of the data type and variable name(s).

A variable name must be valid identifier.

You can identify multiple variables of the same type on a single line, but not multiple variables of different types.

# Declaring Primitive Types

---

So now we have data variables with no data.

Data in the form of literals can be assigned to these variables.

The exam explores some of the implications of setting literal data to typed variables where there is a type mismatch.

Valid literals are numeric values in base ten, hexadecimal or octal forms.

# Declaring Primitive Types

---

An unmodified integer value is assumed to be 32 bit int primitive.

A value containing a decimal point is assumed to be 64 bit double.

In Java, a single character value in single quotes is considered a valid char literal, but any characters surrounded by double quotes is considered to be a string literal.

'A' is not equal to "A" when it comes to literals in Java.

# Declaring Primitive Types

Literal Types	Valid Literal Examples
numeric	400 (32 bit signed int primitive) -6000 (32 bit signed int primitive) 5280L or 5280l (declared long) 0xABCD (hexadecimal primitive) 010 (octal primitive) 5.077 (assumed double) 5.077F or 5.077f (declared float) 3_000_000 (as of Java 7, underscores can be used to help readability of a number). 0b00100001 (binary literal introduced in java7)
char	'\u0041' (unicode for letter 'A') 65 (decimal value for letter 'A') 'A' '\n' (escape characters with '\` - represents line feed)
boolean	true false

# Declaring Primitive Types

---

In some cases numeric literals will have a suffix.

Valid suffixes exist for types long, float and double and case of the suffix does not matter, which force the literal value being assigned to the variable to be the type specified by the suffix.

# Local Variable Initialization

---

To use a primitive data type variable in code, you must initialize it.

This is not true for class static variables and instance variables which will have default values assigned to them.

This initialization or lack thereof is a common theme in the exam questions.

# Local Variable Initialization

---

So we've learned that it's ok to have an uninitialized local variable as long as you never use it. A good IDE will prompt you to get rid of this variable.

You will get a compile error if the local variable is not initialized and then referenced as we've just seen.

Now we need to talk about partial initialization of variables.

# Local Variable Initialization

---

Partial initialization of a local variable is a compile error.

Some examples of partial initialization are:

- Initialization in an if statement block without a corresponding else block.
- Initialization in a switch statement but not in a default statement.
- Initialization in a while loop, note that do/while always executed once, so ok.

# Local Variable Initialization

---

A primitive data type can initialized and/or assigned a literal value, which we discussed somewhat in the previous video, or it can be assigned to other variables as described in this chart.

# Local Variable Initialization

Type of initialization or assignment	Example	
To a Literal as described in previous section	<pre>int i1 = 10; // decimal integer int i2 = 012; // octal int i3 = 0xA; // Hexadecimal int i4 = 0b00001010; //binary</pre>	All of these set the decimal representation of 10 to an int primitive data type
To a variable of same data type	<pre>int i1=10; int i2=i1; // Assigned to another int variable.</pre>	
To a variable of smaller data type.	<pre>short s1=10; int i2=s1; // Assigned to a smaller primitive data variable, in this case short.</pre>	
To a wrapper class which will automatically do casting (auto un-boxing) as long as wrapper's data type size is less than or same as your variable's declared data type size.	<pre>Short s1Wrapper=10; Integer i1Wrapper = 10;  int i1= i1Wrapper; //Assigned to a wrapper class of the corresponding primitive data type int i2 = s1Wrapper; //Assigned to a wrapper class for a smaller primitive data type</pre>	
To a casted variable or literal	<pre>int i1 = (int) 100L; // 100L is a literal for a long, you can cast this and assign it to an int as shown.</pre>	
To an instance variable of an object or class as long as the above rules apply and access modifier allows it. More on this later.	<pre>Foo foo = new Foo(); // Assume Foo class exists and has a myInteger instance variable attribute  int i1 = foo.myInteger;</pre>	
To a Return value of a method in scope, as long as the above rules apply. More on this later.	<pre>Foo foo = new Foo(); // Assume Foo class exists and has a method getMethod() which returns an int or wrapper Int or any type of data as described in this table.  int i1= foo.getMethod();</pre>	

# Local Variable Initialization

---

A local variable is not initialized with a default value.

You must initialize FULLY (not in a conditional statement / switch statement/ while loop if all conditions are not met) if you are using the variable in code.

If you never use the variable, no compiler error occurs.

# Narrowing and Widening

---

Narrowing is when you assign a larger primitive data literal or variable to a smaller one.

Widening is when you are putting something small in a larger variable.

As you can imagine, the compiler is more forgiving with widening attempts, and has more constraints on narrowing.

So why not just use doubles and longs everywhere? Well you could, but you'd be eating up as much as twice the memory unnecessarily.

# Narrowing and Widening

---

We've mentioned this before but this is a good place to repeat it:

- Every numeric value literal that does not contain a decimal point is a 32 bit int.
- Every numeric literal with a decimal is a double.

# Casting

---

There are times when you want to force the compiler to overlook its narrowing and widening checks, because you have knowledge of the actual values occurring in the program during execution.

You can do this by using casting.

You cast by referencing the type you want to be widened, or narrowed to in parenthesis preceding the variable or value that is to be converted.

# Casting

---

You can cast to a larger sized variable (widening) - This is the more common case and poses less risk since there is no risk of loss of value.

You can cast to a smaller sized variable (narrowing) - There may be a valid case for this, however be aware that unexpected results may occur.

Perhaps you want to use a method that requires a smaller variable type and you know your value will not exceed the smaller variable range at the time of execution.

# Casting

---

The problem with casting is that if your value does not fall into the valid value range, your data may underflow or overflow.

- Underflow is defining or casting a value less than the minimum value for the datatype.
- Overflow is defining or casting a value greater than the maximum value for the datatype.

# Casting

---

In the example we just demonstrated, we've created both of these conditions, so that the s2 values in order were :

32767 [ This is what happens when you underflow a short by 1, the sign bit gets toggled and now value is Short.MAX\_VALUE, probably not what you want]

-32768

.... The 65534 values between -32768 and 32767

32767

-32768 [ This is what happens when you overflow a short by 1, the sign bit gets toggled and now value is Short.MIN\_VALUE, probably not what you want]

# Casting

---

In some cases you might want to cast to a less precise data type, from a float or double to an int or long, for example. This will truncate the number to a whole number.

# Casting

---

We've talked about the following:

- Narrowing and widening for literals, as well as for variables.
- Shown when the Java compiler allows it or does not.

You can always force the compiler's hand and use a cast, but you should always ask yourself why you need a cast, and evaluate the risks associated with casting and try to code for the risks.

# Declare and Initialize Variables: Out of the Ordinary Concepts

---

In this code sample, we'll explore a Java 8 feature, that has some support for unsigned ints (and unsigned longs).

# Declare and Initialize Variables: Things to Remember

---

Here are some things to remember when reviewing code samples on the exam, that find themselves into many questions:

- Literals with decimal default to double, not float.
- Doubles and floats do not overflow, since they are approximated.
- Local variable primitives are not initialized.
- Class static and instance members are initialized.
- null is not a valid value for a primitive data type.

# Declare and Initialize Variables: Things to Remember

---

Finally, you will see a lot of constructors and methods in the next sections on String and StringBuilder, that have parameters or methods that refer to code points.

The Java specification includes the following description of the char data type, and it's limitations to represent all the possible characters available, and Java's implementation to address the limitation.

You do not need to memorize this description for the exam, but it is enlightening.

# Unicode Character Representations

---

The char data type (and therefore the value that a Character object encapsulates) are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities.

The Unicode Standard has since been changed, to allow for characters whose representation requires more than 16 bits.

The range of legal code points is now **U+0000** to **U+10FFFF**, known as Unicode scalar value.

# Unicode Character Representations

---

The set of characters from **U+0000** to **U+FFFF**, is sometimes referred to as the Basic Multilingual Plane (BMP).

Characters whose code points are greater than **U+FFFF**, are called supplementary characters.

The Java platform uses the UTF-16 representation, in char arrays, and in the String and StringBuffer classes.

In this representation, supplementary characters are represented as a pair of char values, the first from the high-surrogates range, (\uD800-\uDBFF), the second from the low-surrogates range (\uDC00-\uDFFF).

# Scope

---

Oracle's Java Specification states "The scope of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name, provided it is visible."

A local variable, formal parameter, exception parameter, and local class can only be referred to using a simple name, not a qualified name.

Because of this, some declarations are not permitted within the scope of these elements.

This can be re-stated - If you cannot qualify a variable name in its current scope, you cannot create another local variable name in a more limited scope.

# Scope

---

There are several scopes you'll need to understand and will be tested on when creating local variables.

Scope	Qualifier
Class	{DefinedClassType}
Instance	this
Method	none
Loop	none
Loop Block	none
Block Including Exception Block	none

# Scope

---

	<b>Age Range</b>
The Greatest Generation	> 95
The Silent Generation	70 - 95
Baby Boomer	55 - 70
Gen X	40 - 55
Millenials	20 - 40
Gen Z	0 - 20

# More on Scope

---

Again, we've introduced a compile error.

Just because you are in your own block, the if block, you still have access to variables outside that scope, and the compiler would be unable to distinguish between the i in this if block and the i in the surrounding block.

# More on Scope

---

So in short, the best way to quickly see if your local variable definition is going to raise a compile error is to ask yourself if you can qualify it by a class type, or the words this, or super, etc.

If you cannot, and it has already been defined in an outer scope, then it will cause an error.

This is especially hard to spot in for loop initialization statements.

Shadowing is when a local variable is legally declared, but another declaration that can be qualified is not used or accessed because the local variable takes precedence.

# Local Variable Scope: Out of the Ordinary Concepts

You can define a nested or inner class within a method which seems to break the local variable rule we learned.

The following code compiles and executes without problems. This is because an inner class has an IMPLIED qualifier reference to the outer class object and its attributes and methods.

# Local Variable Scope: Out of the Ordinary Concepts

If you can remember that a class defined locally has an implied qualifier applied to any attributes and methods in the outer class, you can keep track of your variables.

An inner class has an IMPLIED qualifier reference to the outer class object and its attributes and methods.

# Local Variable Type Inference

---

Local Variable Type Inference (LVTI) is also known as var declaration.

This is a Java 10 feature introduced to reduce the verbosity of code.

This feature is only available for local variables inside a method body.

# What Local Variable Type Inference is and is NOT

What it is:

- A shortcut to reduce verbosity of code.

What it is not:

- A data type , the data type must be able to be inferred by compiler when using it.
- A keyword, you can actually use the text 'var' as an identifier.

**IMPORTANT:** var is not a type, it's short hand.

It can only be used inside methods for local variables, and the type of variable on right side of equation must be inferable by compiler, to avoid compilation errors.

# Local Variable Type Inference: Out of the Ordinary Concepts

var is NOT to be confused with variable type.

Just because you define a local variable with the var designation, it does not mean it is a variable type.

The type is inferred by its initialization, and cannot be changed.

All the narrowing, widening rules for the inferred type apply.

# Local Variable Type Inference: Out of the Ordinary Concepts

You cannot use Local Variable Type Inference (var) for:

- Class or Instance variables.
- Method return types.
- Method parameters.
- Constructor parameters.

You can use Local Variable Type Inference (var) for:

- Local variables in methods & code blocks.
- Loop variables.

# Local Variable Type Inference: Out of the Ordinary Concepts

VALID uses of Local Variable Type Inference are for local variables only.

<b>IMPORTANT: These are all assumed to be local variables</b>	<b>Explanation</b>
<code>var i = 1;</code>	Since the literal 1 is an int, var i is inferred to be an int
<code>var j = 1.0f/2;</code>	j is inferred to be a float
<code>short s = 1;</code> <code>var k = +s;</code>	k is inferred to be an int, not a short, since the operator + (which doesn't change the value of 1) forces s to promoted to an int.
<code>var list = new ArrayList&lt;&gt;();</code>	This is ok, because ArrayList does not need to be typed to be initialized.
<code>var myArray = new String[5];</code>	This is ok because type is inferred to be an array of String.

# Local Variable Type Inference: Out of the Ordinary Concepts

INVALID declaration of Local Variable Type Inference (for local variables only).

<b>IMPORTANT: These are all assumed to be local variables</b>	<b>Explanation</b>
<code>var i = 1, i = 2;</code>	var cannot be used in compound statement
<code>var i,j = 1;</code>	var cannot be used in compound statement
<code>var integer;</code>	integer here is variable name and therefore type cannot be inferred without an initializer
<code>var myObject = null;</code>	var variable cannot be initialized to null, type cannot be inferred
<code>var myArray={"A","B"};</code>	This looks legal, it's an array of String, right? but var variable cannot be initialized with an array initializer

# String Recap

---

A String in java is an object of class `java.lang.String` and represents an array or series of characters, but is NOT an array of the primitive data type `char`.

When a string is created without a constructor, aka not using `new`, the string is stored in a special area of the heap called the string pool, whose purpose is to maintain a set of unique strings - this is called interning.

You can manually intern, using the `intern` method on a `String` object.

When you assign two variables to the same string literal, these strings are considered equal and the comparator `==` evaluates to true.

# String Recap

---

We'll remind you that Strings are immutable, meaning they cannot be changed.

Once you create a String, with a string literal, for example "Hello", that string remains "Hello" on the string pool with a single reference to it.

If you do a string concatenation, you are not changing the current String, but creating a new String object with the concatenated Strings.

# String Recap

---

Summary:

- Literal Strings have their own special area of the heap memory called the String Pool.
- Any assignment of a literal will refer to the same String in this pool.
- You can force a string to be added to the pool by calling the intern() method on the String object.
- Strings are immutable, any methods that seem to operate on a String are actually creating new Strings, and these new Strings need to be assigned to the same variable (self assignment) or assigned to a new variable which represents a reference to the new String.

# Creating Strings

---

You can create strings using the methods shown on the slide:

String creation method	Examples
Assign string literal to a String variable. A string literal is contained in double quotes	<code>String s = "hello";</code>
String constructors, of which there are several. It is important to note that string constructors, for the byte, char and int arrays, take a second parameter which represents the starting index and the third parameter is a count of elements which will be copied from the array, not an ending index.	<code>String s1 = new String("hello");</code> <code>byte[] bytes = {'a','b','c'}; String s2 = new String( bytes);</code> <code>char[] chars= {'a','b','c'}; String s3 = new String( chars);</code> <code>StringBuilder sb = new StringBuilder("hello"); String s4 = new String(sb);</code>
String concatenation	<code>String s5 = "hello" + " world " + getMethodString();</code>

# Creating Strings

---

In the last video, we covered creating String Objects using string literals and the String constructor.

In this video, we'll review some of the constructor methods which you may not have used as often as the single parameter constructors.

These constructors have an offset parameter and a count parameter, and set the new String object to a partial region of the character sequence being passed to the constructor.

In other words, you are setting the value of the new string to some substring of the parameter string passed.

The beginning offset is an INCLUSIVE offset, meaning the character at that offset will be including in the substring.

# Creating Strings

---

What is interesting to note and may cause headaches on the exam, is that for constructors, the parameter after the starting index parameter, is not an ending index parameter, but a count of characters to be included after the beginning offset.

If this count + the offset exceeds the number of characters in the sequence being passed, you will get a runtime error.

# String Concatenation

---

The most common method of manipulating strings in Java is the Concatenation operator, the Concatenation operator for a string is '+'.

# Reference Variables

---

When you concatenate a string to a reference variable, if the variable is a reference type, the `toString()` method on the object is called, if the variable is a primitive data type, the variable is boxed to a wrapper and its `toString()` method is called.

# Manipulating Strings

---

The String Object Methods are too numerous to review individually and you are probably very familiar with many of them.

The thing to remember about String object methods is their starting index is always zero, and the ending index is the string length minus 1 (`string.length()-1`).

Many string methods use a starting and ending index to copy, replace, or search through the String.

The beginning index is INCLUSIVE, which means the method will use the character at that beginning index;

The ending index is EXCLUSIVE, which means the character at the ending index will not be included.

# Manipulating Strings

---

We will talk about the following categories of methods and show a few examples by category.

# Comparison

---

- **equals**
- **equalsIgnoreCase**
- **contentEquals**
- **compareTo**
- **isEmpty**
- **isBlank**

# Text Searches

---

- **contains**
- **equalsIgnoreCase**
- **endsWith**
- **indexOf**
- **lastIndexOf**
- **matches**
- **startsWith**

# Text Manipulation

- **concat**
- **join (introduced in Java 8)**
- **replace**
- **replaceAll**
- **replaceFirst**
- **split**
- **substring**
- **subSequence**

# Text Transformation

---

- **chars (introduced in Java 9)**
  - **codePoints (introduced in Java 9)**
  - **format**
  - **lines (introduced in Java 11)**
  - **repeat (introduced in Java 11)**
  - **strip**
  - **stripLeading**
  - **stripTrailing**
  - **toCharArray**
-

# Manipulating Strings

---

Let's talk about String Comparison Methods a little bit using this chart.

	Description	Origination	Parameter Type	Evaluation
<code>==</code>	Compares two objects			returns true when reference is the same.
<code>equals</code>	Object .equals is same as <code>==</code> , but String.equals tests for String and tests value	Overrides Object.equals	Can be any object, but only a String might evaluate to true	returns true if references are the same, or if parameter Object is a String AND the value is the same
<code>equalsIgnoreCase</code>	Tests string value ignoring case		Must be String	returns true if 2 Strings have same value ignoring case
<code>contentEquals</code>	Compares the value of String to the value in the parameter		CharSequence	returns true if values in String and the parameter are the same.
<code>compareTo</code>	Compares the value of String to the value in the parameter	Interface Comparable	String	returns 0 if values in Strings are the same returns negative or positive number if Strings are not equal
<code>compareToIgnoreCase</code>	Compares the value of String to the value in the parameter		String	returns 0 if values in Strings are the same ignoring case returns negative or positive number if Strings are not equal

# Manipulating Strings

---

Before we move on to methods for searching text, there are 2 other methods which are comparative in nature, these are isEmpty and isBlank which return booleans.

The following slide demonstrates the differences between these methods:

# Manipulating Strings

---

The difference between isEmpty() and isBlank().

	isEmpty()	isBlank()
New in Java 11	No	Yes
string has length of 0	evaluates to true	evaluates to true
string has only whitespace characters as defined by Character.isWhitespace	evaluates to false	evaluates to true

# Text Search In String

---

The few ones we'll cover in this lecture:

- lastIndexOf matches.
- regionMatches.

Just remember that for the other methods, the start offset parameter is **inclusive**, and the end offset is **exclusive**.

# Text Search In String

---

regionMatches does not evaluate whether a substring is in the full string exactly, but evaluates whether the region specified by the index offsets and length match a region in the source string, with same length.

# Text Search In String

---

Matches returns true if the WHOLE string you are passing as a parameter matches the String.

You have to use a regular expression pattern generally to use this method as a way to test for a substring.

We are not going to go into regular expressions here, just know that  
mississippi.matches("miss") is not the same as mississippi.indexOf("miss")>-1)

# Replacement Methods and Text Transformation

---

This is very straightforward code, but points out something very important about the replace and substring methods.

The methods do not change the current String value, which is always immutable for a String.

Rather, the method returns a new string with either the substring or the replaced value.

If you see examples like this on the certification exam, just remember that the String value will not change, unless you set the result to a self reference as shown in next code segment.

# Replacement Methods and Text Transformation

---

Java 11 introduced the methods `strip()`, `stripLeading()` and `stripTrailing()`.

	<code>trim()</code>	<code>strip()</code>
New in Java 11	No	Yes
Preferred Method	No	Yes
Trims both leading and trailing white space	Yes	Yes
White Space definition	<code>&lt;= 'U+0020'</code>	<code>Character.isWhitespace</code>

The thing to note about this slide is that `strip()` strips more space characters than `trim()` and offers `stripLeading()` and `stripTrailing()` methods as well

# Replacement Methods and Text Transformation

---

Remember these things:

- Strings are immutable, any method you use that returns a value needs to be assigned to a different String reference or back to the current one. We'll be reviewing this again when we talk about StringBuilder which ARE mutable.
- String constructors and valueOf methods use a start offset (inclusive) and a count of characters. All other methods, where applicable, use a start offset (inclusive) and an end offset (exclusive).

# Replacement Methods and Text Transformation

---

- Methods that use regular expressions are
  - matches() (must match entire String)
  - split()
  - replaceAll and replaceFirst.

# Replacement Methods and Text Transformation

---

As of Java 9, 3 streaming methods were added to String: chars, codepoints and lines.

Streams are not part of the first certification exam objective, but are part of the second so we won't review these here.

This concludes our review of the String methods.

There was a lot to cover and we went through it pretty quickly, assuming this section was more than familiar to you.

# Creating and Manipulating Strings: Out of the Ordinary Concepts

Note that the `compareTo` method and `compareToIgnoreCase` do not return just 0, 1, -1 values but returns 0 if same string, and the result is calculated as follows:

s1	s2	method call	Result	Calculation	When is calculation used
"abc"	"abcdef"	s1.compareTo(s2)	-3	s1.length() - s2.length()	string s1 is substring of s2 starting at index 0
		s2.compareTo(s1)	3	s2.length()-s1.length()	string s2 contains s1 starting at index 0
"abc"	"ABC"	s1.compareTo(s2)	-32	s1.charAt(indexWhereStringsDiffer)-s2.charAt(indexWhereStringsDiffer)	indexWhereStringsDiffer is equal to 0 in this instance
		s2.compareTo(s1)	32	s2.charAt(indexWhereStringsDiffer)-s1.charAt(indexWhereStringsDiffer)	indexWhereStringsDiffer is equal to 0 in this instance
"abc"	"abc"	s1.compareTo(s2)	0		

# Creating and Manipulating Strings: Out of the Ordinary Concepts

The String constructor and the valueOf method can be used to create new strings which are full or partial strings to the current string, similar to the substring method with one very important difference - the constructor and valueOf method's parameters have same types but the last parameter has different meaning.

# Creating and Manipulating Strings: Out of the Ordinary Concepts

The String is the powerhouse of many applications and there was a lot of information to cover.

There will be many questions on the exam testing your knowledge of the String and its methods.

# Manipulate Data Using the StringBuilder

---

We've already discussed that Strings are immutable.

StringBuilder objects are not.

Strings should always be used unless StringBuilder objects offer an advantage in terms of simpler code or better performance.

For example, if you need to concatenate a large number of strings, such as a temporary error message, or dynamic XML or HTML, then appending to a StringBuilder object is more efficient.

Every StringBuilder has a capacity which is the number of character spaces allotted to it.

Capacity is automatically extended as additions are made to the StringBuilder object.

# Manipulate Data Using the StringBuilder

---

To create a `StringBuilder` Object, you use one of the following constructors:

Constructor	Description	Capacity
<code>StringBuilder()</code>	Empty <code>StringBuilder</code> Object with capacity of 16 (16 empty elements)	16
<code>StringBuilder(CharSequence cs)</code>	<code>StringBuilder</code> Object with same characters as the specified <code>CharSequence</code> , plus an extra 16 empty trailing elements.	<code>cs.length() + 16</code>
<code>StringBuilder(int initCapacity)</code>	Empty <code>StringBuilder</code> Object with specified intital capacity	<code>initCapacity</code>
<code>StringBuilder(String s)</code>	<code>StringBuilder</code> Object with same characters as the specified <code>String</code> , plus an extra 16 empty trailing elements	<code>s.length() + 16</code>

# Manipulate Data Using the StringBuilder

---

The StringBuilder Class has a series of append methods and insert methods.

The append methods simply add data to the end of the current value of the StringBuilder, expanding capacity if needed.

The insert methods insert the specified data into the StringBuilder value at the specified starting index.

# Manipulate Data Using the StringBuilder

---

We'll now just touch on a few of the `StringBuilder` methods that are not methods similar to `String` methods.

It is important to note, however, that the append/insert methods in `StringBuilder` are changing the value of the `StringBuilder` object, and not returning a new value from the method calls as do the `String`'s concat methods.

# Manipulate Data Using the StringBuilder

---

Finally, just a note on String, StringBuffer and StringBuilder.

StringBuffer will not be on the exam, but it's interesting to note its out there, if you need a thread safe but mutable object with a string value.

# Manipulate Data Using the StringBuilder

---

Object	Features	Notes
String	immutable	Want to use String vs. StringBuilder when your strings are short and most likely to be re-used throughout application
StringBuffer	Mutable, only option before JDK 5 Threadsafe	All public methods are synchronized which have a negative impact on performance. You want to use StringBuffer if the attribute is being accessed by multiple concurrent threads
StringBuilder	Mutable, introduced in JDK 5 Garbage Collected Capacity can be adjusted	Want to use StringBuilder vs String if you are dynamically building large string data sets, like log statements or generated HTML/XML.

# StringBuilder: Out of the Ordinary Concepts

The javadoc for trimToSize() describes the method's purpose as "Attempts to reduce storage used for the character sequence".

How would you trim or strip leading/trailing white space from a StringBuilder?

There aren't any actual methods to do this on StringBuilder.

Generally, any cleaning or transformation of Strings is best done as you are appending data to your StringBuilder.

# StringBuilder: Out of the Ordinary Concepts

Let's look at the `getChars` method for `String` or `StringBuilder` in more detail.

There may be some coverage of this on the exam, and I am not sure this is a method that is used a lot.

It's structured a bit differently, because it doesn't return anything from the method itself, but alters data in one of the parameters passed to it.

# StringBuilder: Out of the Ordinary Concepts

---

You can see how easily, if the `getChars` method is one you do not use a lot, that a question on the certification exam could confuse you:

- String and StringBuilder both implement this method, but not through the CharSequence interface, so if you see an exam question applying this method to a CharSequence, you can flag it as a compile error.
- The destination parameter for the characters (the third parameter) must be an initialized char array. If you see an exam question with any other type or the array is not initialized, you can flag it as a compile error.

# StringBuilder: Out of the Ordinary Concepts

---

- The destination character array size must be greater or equal to the length of the selected source + the defined destination index.
- You receive a runtime error if you exceed the array size.
- If you do not select any characters from the source (by specifying sourceEndIndex = sourceStartIndex+1), then your char array does not change at all.

# Section Intro

---

A program is made up of expressions and statements.

Expressions are formed with operators and variables to manipulate data.

Statements manipulate the flow of the program.

Operators are the glue that hold expressions together.

An expression can have one, two or many operands joined by an operator to generate some change to the data.

In this section we are going to review java's operators as well as some of java's most fundamental control statements (if/else and switch statements) and loops.

# Using Operators and Decision Constructs

- Use Java operators including the use of parenthesis to override operator precedence
- Use Java control statements including if, else, and switch
- Create and use do/while, while, for and for each loops, including nested loops, use break and continue statements

USING OPERATORS AND DECISION CONSTRUCTS  
JSE11DCP1-1-2

# Java Operator Overview

---

Java operators are symbols that are used to perform mathematical or logical manipulations and have two types of classifications, the first classification is based on the number of operands the operator has (unary, binary, ternary), and the second classification is the type of operation it performs.

# Java Operator Overview

---

When talking about operators, it is important to know:

- Precedence of an operator (unary operators are evaluated before binary operators).
- Among the unary operators, the postfix increment and decrement operators have the highest precedence.
- Unary operators with the exception of the prefix/postfix operators, promote the variable to an int if it is smaller than an int.

# Java Operator Overview

---

When talking about operators, it is important to know:

- If all operators have the same precedence, the expression will be evaluated from left to right, with the exception of the simple and compound assignment operators which are evaluated right to left.
- Parentheses always take precedence.

# Java Operator Overview

---

Unary Operators: Expressions with unary operators group right-to-left, so that  $\sim x$  means the same as  $\sim(\sim x)$ .

Symbol	Name	Examples	Description	Notes
--	Prefix Decrement	--a	decrements value of a before expression is evaluated	It is possible that a variable will go out of scope before the post decrement can occur.
	Postfix Decrement	a--	decrements value of a after expression is evaluated	
++	Prefix Increment	++a	increments value of a before expression is evaluated	It is possible that a variable will go out of scope before the post increment can occur.
	Postfix Increment	a++	increments value of a after expression is evaluated	
-	Unary Minus	-a	returns negated value of a without changing the value of a	If a is a data type with size less than an int, a will get promoted to an int
+	Unary Plus	+a	allowable, but does nothing to the value of a	If a is a data type with size less than an int, a will get promoted to an int
!	Logical Complement Operator	!a	returns complement of a boolean value	
~	Bitwise Complement Operator	~a	bitwise complement, turns a 1 to 0 or a 0 to a 1, In all cases, $\sim x$ equals $(\sim x) - 1$	
	Cast Operator	(int) a		

# Pre and Postfix Increment and Decrement Operators

---

First, we'll explore the prefix decrement, prefix increment and postfix decrement, postfix increment operators.

It is very important to remember that the postfix decrement and postfix increment operators, do not change the value of its unary operand, until the expression it is operating in is considered completed OR (and this OR is kind of important) the operand is used again in the same statement.

This will be tested on the Certification Exam in very complex ways.

So remember that the unary variable being operated on does get operated on (the value at the end is either incremented or decremented) but any assignments, or expressions this statement is part of, will be using the unchanged value until the expression statement is evaluated successfully.

# Pre and Postfix Increment and Decrement Operators

If the statement is interrupted somehow, then the postfix increment or decrement may not actually be applied.

It needs to be noted that the increment and decrement operators are changing the value contained in the reference itself.

If you make the statement `a++`, you are incrementing the value in the variable `a`. These operands can be stand alone statements.

This is different, for example, from the unary minus `-a`, which has no impact on the value in the operand itself, and cannot be a statement on its own.

Its output must be assigned or used in an expression.

# Pre and Postfix Increment and Decrement Operators

The unary minus returns a negative value if the value in the operand is positive, and a positive value if the value in the operand is negative.

The unary plus returns a value with the same sign as the operand, and has no effect on the value.

Both operators, will however, promote the return values to int.

# Pre and Postfix Increment and Decrement Operators

The bitwise complement operator flips the bit for the entire value of the variable.

The binary literal value of the integer 0, gets every bit flipped to 1, making it's integer value -1.

The logical complement operator only operates on a boolean, and changes false to not false (true).

# Binary Operators Code Part 2

---

I am sure you know that:

- An OR bit is 1 if one or the other operand bit is 1.
- An AND bit is 1 if both operand bits are 1.
- An XOR bit is 0 if both operand bits are the same value, otherwise it is 1.

# Binary Operators Code Part 2

---

Finally, we want to look at the ternary operator, the only operator that has three operands.

It's format is:

```
operand1 ? operand2 : operand3
```

The ternary operator is often compared to an if statement, but it is an operator and not a statement, and therefore is either used in an expression, or returns a value to a variable.

It cannot be a standalone statement.

# Binary Operators Code Part 2

---

The first operand must be a boolean value or be an expression whose result is a boolean value.

The other operands' data types can be anything but both operands must share a common type.

If operand1 evaluates to true, then the resulting value will be operand2.

If operand1 is false, then the resulting value will be operand 3.

If operand2 and operand3 are both expressions, only one of the expressions is ever evaluated, based on the value of operand1.

It will never be the case that both expressions are evaluated.

# Binary Operators Code Part 2

---

The literal value integer can be autoboxed to an Integer, a child of Object, but an Integer operand and a String operand cannot return a result to a String, since Integer is not a child of String.

# Binary Operators Overview

---

With the exception of the conditional operator, the rest are binary, and we'll group them by the type of operation they perform.

These operators are listed by their precedence level in the table.

In other words, Multiplicative Operators have a higher precedence than the Additive operators and so on.

Note that all the unary operators have a higher precedence than the binary.

# Binary Operators Overview

---

Category	Symbol	Simple Description	Notes
Multiplicative Operators	*	Multiplication	Precedence equal among them, group left-to-right
	/	Division	
	%	Modulus	
Additive Operators	+ -	Addition Subtraction	Precedence equal among them, group left-to-right
Shift Operators	<< >> >>>	left shift signed right shift unsigned right shift	The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance. Precedence equal among them, group left-to-right

# Binary Operators Overview

---

Category	Symbol	Simple Description	Notes
Relational Operators	< <= > >= <code>instanceOf</code>	less than less than or equal greater than greater than or equal	Precedence equal among them, group left-to-right
Equality Operators	<code>==</code> <code>!=</code>	Equals Not Equals	
Bitwise and Logical Operators	<code>&amp;</code> <code>^</code> <code> </code>	AND XOR (Exclusive OR) OR	These operators have different precedence, with <code>&amp;</code> having the highest precedence and <code> </code> the lowest precedence.  Bitwise operators are not on the exam, but logical operators are

# Binary Operators Overview

---

Category	Symbol	Simple Description	Notes
Conditional AND Operator	&&		The conditional-and operator <b>&amp;&amp;</b> is like <b>&amp;</b> but evaluates its right-hand operand only if the value of its left-hand operand is true.
Conditional OR Operator			The conditional-or operator <b>  </b> operator is like <b> </b> but evaluates its right-hand operand only if the value of its left-hand operand is false.
Conditional Operator	? :		This is a ternary operator, has 3 operands***
Assignment Operators	= *= /= %= += -= <<=>>= >>>= &= ^=  =	Simple Assignment  Compound Assignment	*** they group right-to-left***. Thus, $a=b=c$ means $a=(b=c)$ , which assigns the value of $c$ to $b$ and then assigns the value of $b$ to $a$ .
Lambda Operator	->		

# Binary Operators Overview

---

At this point we need a quick word about numeric promotion.

- For a unary operator, that is not the pre/post decrement/increment operator, if the type of the operand is smaller than an int, the operand will automatically be promoted to an int.
- For a binary operator, both operands are promoted to int, if they are smaller than an int, but if any of the operands is larger than an int, than it is promoted to the larger type. Note that this is NOT true for the compound assignment operators.
- What this means is that any operations on numeric values, will never result in a value that is smaller than an int.

# Java Operators: Out of the Ordinary Concepts

---

If the type of operand is smaller than an int, it is promoted to an int before applying the operator (with the exception of the increment/decrement operators).

# Java Operators: Out of the Ordinary Concepts

---

It can be very confusing when your expression is using the same variable multiple times, and the postfix/prefix increment/decrement operators are being used on them.

Look at this interesting chart. Do any of the results surprise you? Don't try to memorize it.

We are going to walk through sample code that tests the veracity of each statement.

Expression	Result
<code>--number - number--;</code>	result is always zero
<code>number-- - --number;</code>	result is always number -2
<code>(number--) ==(number+=1);</code>	This evaluates to true
<code>int number= 10; number = number--;</code>	Final value in number is 10

# Java Operators: Out of the Ordinary Concepts

---

So, exploring the first iteration:

- number = i; i=10, number = 10;
- result = --number - number--
- We evaluate values left to right...

```
for (int i = 10; i <= 50; i += 10) {  
    number = i;  
    result = --number - number--;  
  
    System.out.println("i = " + i + ", number = " + number +  
        ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

- `result = 9 - number--` //number gets assigned the value of 9 from the prefix decrement operator
  - `result = 9 - (9)` // number gets assigned the value of 8 from the postfix decrement operator, but after this math is done...
    - `result = 0, number = 8`

```
for (int i = 10; i <= 50; i += 10) {  
    number = i;  
    result = --number - number--;  
  
    System.out.println("i = " + i + ", number = " + number +  
                      ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

Let's do the same thing with the next expression in our table:

```
int result = number-- - --number;
```

Expression	Result
--number - number--;	result is always zero
number-- - --number;	result is always number -2
(number--) ==(number+=1);	This evaluates to true
int number= 10; number = number--;	Final value in number is 10

# Java Operators: Out of the Ordinary Concepts

---

We'll go through same process as before, but this one, I think is a little harder to grasp.

We have to remember that subsequent references to the same variable, in a single expression, will result in the post decrement/increment occurring on those variables, before they are further evaluated in expressions.

# Java Operators: Out of the Ordinary Concepts

---

- number = i; i=10, number = 10;
- result = number-- - --number
- We evaluate left to right...
  - result = (10) (but number is now 9) - --number

```
for (int i = 10; i <= 50; i += 10) {  
    number = i;  
    result = number-- - --number;  
    System.out.println("i = " + i + ", number = " + number +  
        ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

- result = 10 - --(9)
- result = 10 - 8
- result = 2

```
for (int i = 10; i <= 50; i += 10) {  
    number = i;  
    result = number-- - --number;  
    System.out.println("i = " + i + ", number = " + number +  
        ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

Let's step through the expression left to right for the first iteration, evaluating left to right as before, and applying multiplicative precedence after the values are known.

- result = number-- - number++ \* --number;
- result = 10 (number is now 9 after postfix decrement) - number++ \* --number;
- result = 10 - 9 (number is now 10 after postfix increment) \* --number

```
for (int i = 10; i <= 20; i += 10) {  
    number = i;  
    result = number-- - number++ * --number;  
    System.out.println("i = " + i + ", number = " +  
        number + ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

- result = 10 - 9 \* -(10)
- result = 10 - 9\*9
- result = 10 - 81
- result = -71

```
for (int i = 10; i <= 20; i += 10) {  
    number = i;  
    result = number-- - number++ * --number;  
    System.out.println("i = " + i + ", number = " +  
        number + ", result = " + result);  
}
```

# Java Operators: Out of the Ordinary Concepts

---

And finally let's explore the last two rows from our slide.

Expression	Result
<code>--number - number--;</code>	result is always zero
<code>number-- - --number;</code>	result is always number -2
<code>(number--) ==(number+=1);</code>	This evaluates to true
<code>int number= 10; number = number--;</code>	Final value in number is 10

# Java Operators: Out of the Ordinary Concepts

---

We evaluate left to right:

- `isEqual = (number--) == (number += 1);`
- Evaluating left to right...
  - `isEqual = (10) (but number is 9 now) == (number+=1)`

```
number = 10;  
boolean isEqual = (number--) == (number += 1);  
System.out.println("isEqual = " + isEqual +  
    ", and number = " + number);
```

# Java Operators: Out of the Ordinary Concepts

---

- `isEqual = 10 == (9)+=1`
- `isEqual = 10 == 10;`

```
number = 10;  
boolean isEqual = (number--) == (number += 1);  
System.out.println("isEqual = " + isEqual +  
    ", and number = " + number);
```

# Java Operators: Out of the Ordinary Concepts

---

Expression	Result
<code>--number - number--;</code>	result is always zero
<code>number-- - --number;</code>	result is always number -2
<code>(number--) ==(number+=1);</code>	This evaluates to true
<code>int number= 10; number = number--;</code>	Final value in number is 10

# Java Operators: Out of the Ordinary Concepts

You may find some of these confusing chains of increments and decrements in expressions on the exam or in loops.

Slow down and write it down is the best advice I can give you.

# if else Decision Construct

---

We will be reviewing Java's decision constructs. These are the if/else statements as well as the switch statement.

The if statement allows conditional execution of a statement, or block of statements, or a choice of two statements or statement blocks. You know it well.

# if else Decision Construct

---

Some things to remember:

- The words 'then', 'elseif' are not valid, these are from other languages.
- If you do not have a bracket after the if or else, then only one line of code is contained within the conditional block.
- You can have an empty statement block after the if or else as long as semi-colon follows.
- Dangling elses are assumed to go with the inner most if statement.

We will be walking through some common coding mistakes, and items that might find their way into an exam question.

# switch Decision Construct

---

The other decision construct in Java is the switch statement.

The switch statement transfers control to one of several statements, depending on the value of an expression.

# switch Decision Construct

---

Some things to remember about a switch statement:

- A switch statement can only work with byte, short, char, and int primitive data types.
- It also works with enumerated types, the String class, and the classes that wrap certain primitive types: Character, Byte, Short, and Integer.
- The default case label only matches an argument if all the other labels do not, regardless of its position.
- The default label does not have to be the last label in the case label list.
- You can only have one default case label and block.
- You do not need any case labels in the switch block.

# switch Decision Construct

---

Some things to remember about a switch statement:

- The break statement is optional, however, if it is omitted, you are creating a fall-through situation.
- The default case label and block are optional.
- Case labels must be same data type as the result of the switch expression.

# switch Decision Construct

---

We will run through several examples of switch statements which are valid, but perhaps not common.

The first example will demonstrate, is something called fall through, and show what happens, when your default label is not the last case label in the statement.

# switch Decision Construct

---

The switch variable cannot be a long, double, float or boolean, or any object, other than the specific wrappers, enum, or a String that we mentioned earlier.

If you see a switch statement on the exam using one of these types without a specific cast, consider it a gift.

Pick compiler error and move on.

# Java Control Statements: Out of the Ordinary Concepts

How would you break out of the loop from a switch statement?

You can't do it with a simple break, can you?

Because a break in a switch statement is just a break in the switch label.

To break out of a loop from a switch conditional block, you'd need to create a label for your for loop, which we'll cover in much more detail in the video on loops, but you are probably already familiar with labels.

We'll add a label here and we'll answer the question we asked, by changing our code for case labels 104 and 107.

# Java Control Statements: Out of the Ordinary Concepts

When you see a switch statement on an exam:

- Examine its location - is it participating in a loop?
- Don't let the breaks and continue statements mislead you.
- Examine the switch expression, can you determine the type?
- Does the type match the case labels?
- Is the type valid?
- doubles, floats, longs, booleans are not valid.
- Objects that are not Strings or wrappers are also not valid.
- If the switch expression is a local variable, has it been initialized?

# Java Control Statements: Out of the Ordinary Concepts

When you see a switch statement on an exam:

- If it hasn't, does the switch statement have a default block?
- If not, then the answer is compiler error.
- Examine the case labels, are they all compiler constants?
- Variables are not permitted.
- If case blocks do not have breaks, the code falls through, so multiple blocks could be executed.
- Finally, remember that a default label does not have to be last or be included at all.

# Loop Structures

---

Most of our examples included loop processing, but here we'll talk formally about loops, which cause a set of instructions to execute repeatedly, until a certain condition is met, or the element list being looped through is exhausted.

Loops are fundamental to structured programming.

Java supports 3 types of loops, the while loop, the do while loop, and the for loop.

# Loop Structures

---

The while and do/while loops are similar by nature, the do while loop always executes it's statement block once before checking the expression.

While Loop - while loop executes until the defined condition is met. Any iteration counters are done in the code block.

```
while (expression) {  
    statement(s)  
}
```

# Loop Structures

---

Do While Loop always executes one time before the condition is checked, and subsequently works similarly to the while loop.

```
do {  
    statement(s)  
} while (expression);
```

# Loop Structures

---

The continue statement is similar to the break statement, such that the code below the statement will not execute, but it does not break out of the loop, instead it executes the next iteration of the loop, first testing the condition of the loop.

# Loop Structures

---

Next, we'll look at the for loop, which is used either when your iteration count is known, or you have an Iterable collection.

There are two flavors of the for loop, the traditional for loop, and the enhanced for loop.

# Loop Structures

---

The traditional for loop, has built-in mechanisms to control the initialization, and updating of the loop variable, as well as the comparison condition (termination section).

The general form for the for loop is:

```
for (initialization; termination; increment) {  
    statement(s)  
}
```

# Loop Structures

---

When using this version of the `for` statement, keep in mind:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins. It's *initialization* expression is optional.
- When the *termination* expression evaluates to `false`, the loop terminates. The *termination* expression is optional. If default value is `true`;
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to *increment* or *decrement* a value. The *increment* expression is optional.

# Loop Structures

---

The enhanced for loop is used to loop through an Iterable set of data and do something:

```
for ( FormalParameter : Expression ) Statement;
```

- The type of the Expression must be Iterable, or an array type.

# Loop Structures

---

Some Additional Notes on Loops:

- It is interesting to note that loop is not a reserved word.
- Any loop can have a break in it, usually a conditional break.
- Any loop can have a continue statement in it, also usually conditional.
- Statements after the continue statement are skipped until the next iteration.

# Loop Structures

---

Some Additional Notes on Loops:

- This is a perfectly legal for loop, an infinite loop.

```
for ( ; ; ) ;
```

# for Loop: Out of the Ordinary Concepts

---

- You cannot reference a label with the continue, break statements that is not in the loop scope, a compiler error occurs.
- An outer loop cannot reference the inner loop's labels, but an inner loop can reference the outer loop's labels.
- You can break out of a nested loop from a nested loop.
- You can also completely break out of the parent loop, from the nested loop.

# for Loop: Out of the Ordinary Concepts

In the next example, we are going to test postfix and prefix increment unary operators in our for loops, a common theme in exam questions.

We are going to call a method in all three sections of the for loop declaration, and explore what happens if something goes wrong in each of these declaration blocks: initialization, termination, increment.

# for Loop: Out of the Ordinary Concepts

This example demonstrated in what instance and order, each of the for loop declaration blocks was executed:

- The initialization block is executed only once, prior to the loop termination condition being evaluated for the first time.
- The loop termination condition is evaluated prior to each loop iteration.
- The increment block is executed after the iteration, prior to the next loop termination condition evaluation.

# Introduction to Working with Java Arrays

---

We have been working all along with Java arrays, since most of our examples have had a main method with a `String[] args` parameter.

The `args` parameter is a single dimensional array made up of `java.lang.String` references.

- An array is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created.
- An array can reference duplicate values.

# Introduction to Working with Java Arrays

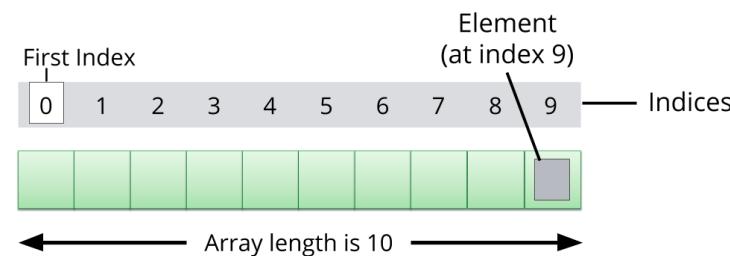
---

- An array can be one dimensional, two dimensional or multi-dimensional representing a matrix of data.
- Note that 'true support' of multi-dimensional arrays is not supported, but Java offers an array of arrays to support a multi-dimensional structure.
- You can store a subclass object in an array, declared to be the type of the superclass, or interface, supporting polymorphism.
- For example, you can create an array of CharSequence, that can store references to Strings, or StringBuilder or both.

# Introduction to Working with Java Arrays

---

Each array is indexed, starting with the zero index, such that the last index is equal to (length of array - 1).



# Array Declaration and Initialization Overview

---

When you declare an array:

- You define the type of the elements in the array.
- The brackets [] are the array identifier.
  - One set of brackets [] on the declaration side is required, to identify a one-dimensional array.
  - The brackets can be to the right of the declared type.
  - Or the brackets can be to the right of the declared variable name.

# Array Declaration and Initialization Overview

---

- Adding bracket sets to both the right of the type and the variable name is actually a legal declaration, but it declares a two-dimensional array, not a one-dimensional array.
- The type of an array can be any type, a primitive, class, or an interface.
- You never include the size of the array, on the declaration side (left side), of a declaration/initialization.

# Valid Array Declarations

---

Declaration	Notes
<code>int[] intArray;</code>	Brackets can be associated to the type, indicating that variable is an array.
<code>short shortArray[];</code>	Brackets can be associated to the variable name, indicating that the variable is an array.
<code>int a, b[], c;</code>	You can define multiple variables of a type on one line, including arrays of that type. Here we have primitive data type variables, a and c, and an array of int stored in b.
<code>int[] d, e, f[];</code>	You can define multiple array variables in a single declaration variable. Note that variable f in this statement is a declaration of a two dimensional array.
<code>String[] stringArray[];</code>	This is valid, but might be unintended, since it declares a two dimensional array.

# Invalid Array Declarations

---

Declaration	Notes
<code>int[2] intArray;</code>	Size is not part of the declaration; Compile Error
<code>int intArray[2];</code>	Size is not part of the declaration; Compile Error
<code>int a, float b[];</code>	You can NOT define multiple variables of different types in the same line, separated by commas.

# Creating an Array

---

To create an array:

- You can use the new operator:
  - Examples:

- `int[] intArray = new int[10];` // creates an array of int with 10 elements initialized to 0.
- `String[] stringArray = new String[10];` // creates an array of String with  
// 10 elements, all elements initialized to null.

# Creating an Array

---

- Notes
  - Even when creating an array of objects, you are not calling the Object constructor, so you are not using parentheses after the new statement.
  - You must define the size of the array, in brackets.
  - Once a size is defined, you cannot change the size of an array.

# Creating an Array

---

- When you create an array this way, the Java Virtual Machine automatically assigns defaults:
  - Numeric primitives are set to 0.
  - Boolean primitives are set to false.
  - References, including primitive data type wrappers, are set to null.

# Creating an Array

---

- You can use an array initializer. An array initializer is a shortcut, that allows you to create and initialize each element in an array, in one statement.

- Examples:

- ```
String[] stringArray = {"one", "two", null, "three"}; // Creates an array of String with 4 elements,  
// all initialized with specified values
```

```
int[] intArray = {10, 9, 8, 7, 6, 5, 4 ,3, 2, 1}; //Creates an array of int with 10 elements,  
// all initialized with specified values.
```

- Notes

- You can only use the array initializer, in the same statement as the declaration.

# Valid Array Creation Statements

---

| Statement                                                                | Notes                                                                                                                |
|--------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>int[] myIntArray = new int[50];</code>                             | Declaration with brackets associated to type using the new operator.                                                 |
| <code>short myShortArray[] = new short[20];</code>                       | Declaration with brackets associated to variable name using the new operator.                                        |
| <code>int[] mySecondIntArray = null;</code>                              | You can set an array to null.                                                                                        |
| <code>String[] myStringArray;<br/>myStringArray = new String[12];</code> | You can declare an array without initialization, and then initialize using the new operator in a separate statement. |
| <code>int[] myIntArray, mySecondIntArray = new int[50];</code>           | This is valid statement, but only the second array is initialized.                                                   |

# Valid Array Creation Statements

---

| Statement                                                                         | Notes                                                                                 |
|-----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <code>int[] mySecondIntArray, myIntArray = mySecondIntArray = new int[50];</code> | This statement initializes both arrays.                                               |
| <code>String[] myArray = {"one", "two", null, "three"};</code>                    | This is array of string with four elements created with array initializer.            |
| <code>String[] myArray = {};</code>                                               | This is setting array to a 0 length array.                                            |
| <code>String[] stringArray[] = {{"one", "two"}, {"three", "four"}};</code>        | This creates a two dimensional array. We will discuss later, Just know this is valid. |

# Valid Array Creation Statements

| Statement                                        | Notes                                                                                                                                                                                                                                     |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Object o = new int[5];</code>              | Because arrays are ultimately objects, this is valid initialization of an array and assignment to an object.                                                                                                                              |
| <code>int a[] = new int[]{1, 2, 3, 4, 5};</code> | You can combine the array initializer with the new operator, as long as you do not state the size of the array in brackets.                                                                                                               |
| <code>int[][] matrix = new int[2][];</code>      | This is a valid 2 dimensional array declaration and initialization. Only the first dimension is required to have a size, because you can have an array of arrays and the arrays referenced by the element indices can be different sizes. |

# Invalid Array Creation Statements

| Statement                                                                   | Notes                                                                                                 |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>String[] stringArray = new String()[3];</code>                        | The parentheses in this statement generate a compile error.                                           |
| <code>String[3] stringArray = new String[];</code>                          | Size needs to be on the right side of the equation.                                                   |
| <code>int a[] = int[]{ 1, 2, 3, 4, 5} ;</code>                              | An array initializer does not require a restatement of the array.                                     |
| <code>int b[] = new int[5] {1, 2, 3, 4, 5} ;</code>                         | This is invalid because you are stating the size, but the array initializer already defines the size. |
| <code>String[] myArray;<br/>myArray = {"one", "two", null, "three"};</code> | You cannot use the array initializer on a separate statement line from the declaration of the array.  |

# Invalid Array Creation Statements

| Statement                                                                                     | Notes                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>short[] mySecondShortArray, myShortArray = mySecondShortArray = {1, 2, 3, 4, 5};</code> | This is not a valid statement, unlike using the new operator on the right hand side of this equation. You cannot use array initializer in a compound statement. |
| <code>int[][] matrix = new int[][][2];</code>                                                 | Only the first dimension in a 2-d array is required to have a size, and this initialization does not give it a size, so it is incorrect.                        |
| <code>int c[] = new int[5][];</code>                                                          | You cannot initialize a two-dimensional array and assign it to a one-dimensional array.                                                                         |

# Array Declaration and Initialization

---

The Java array is a useful data structure, but it comes with limited methods for manipulating the data.

Hence, manipulating arrays usually includes utilizing the `java.util.Arrays` utility class.

For our first example we will demonstrate using the array length attribute, accessing elements, and changing elements in a single dimensional array.

This example also introduces `java.util.Arrays` - we will be using the `Arrays` utility class's static method `toString`, to print the array in a comma delimited String to the output.

# Manipulating Arrays

---

In this video we are going to look at the methods on the `java.util.Arrays` class more closely, because it would be a rare event if you created an array of data and didn't wish to use some of the methods supplied to you by this utility class.

We will also briefly talk about `ArrayList` and `List`, which are alternative methods for manipulating collections of data and which may find their way into an exam question.

# Manipulating Arrays

---

Note that Lists, and ArrayLists are subjects under the Section Programming Abstractly Through Interfaces, but it seems appropriate to discuss some of their methods here in conjunction with arrays.

The javadoc definition of the java.util.Arrays class is:

The class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

# Manipulating Arrays

---

This slide lists some of the methods available to you to manipulate data in an array, or through the List interface.

| Type of Functionality | Arrays Class methods                                                                                                     | List Interface methods                                                                                |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| Comparison            | <code>compare (Java 9)</code><br><code>compareUnsigned (Java 9)</code><br><code>deepEquals</code><br><code>equals</code> | <code>equals</code><br><code>isEmpty</code>                                                           |
| Searches              | <code>binarySearch</code><br><code>mismatch (Java 9)</code>                                                              | <code>contains</code><br><code>containsAll</code><br><code>indexOf</code><br><code>lastIndexOf</code> |

# Manipulating Arrays

| Type of Functionality | Arrays Class methods                                                                                                                                                                                                                                                                                     | List Interface methods                                                                                                                                                                                                                                                         |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data manipulation     | <code>deepHashCode</code><br><code>deepToString</code><br><code>fill</code><br><code>hashCode</code><br><code>parallelPrefix (Java 8)</code><br><code>parallelSort (Java 8)</code><br><code>parallelSetAll (Java 8)</code><br><code>setAll (Java 8)</code><br><code>sort</code><br><code>toString</code> | <code>add</code><br><code>addAll</code><br><code>clear</code><br><code>get</code><br><code>hashCode</code><br><code>remove</code><br><code>removeAll</code><br><code>replaceAll</code><br><code>retainAll</code><br><code>set</code><br><code>size</code><br><code>sort</code> |

# Manipulating Arrays

| Type of Functionality | Arrays Class methods                                                                                                                        | List Interface methods                                                                                                                                                                               |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data transformation   | <code>asList</code><br><code>copyOf</code><br><code>copyOfRange</code><br><code>spliterator (Java 8)</code><br><code>stream (Java 8)</code> | <code>copyOf (Java 10)</code><br><code>iterator</code><br><code>listIterator</code><br><code>of (Java 9)</code><br><code>spliterator (Java 8)</code><br><code>subList</code><br><code>toArray</code> |

# Manipulating Arrays

---

First, to understand the compare method, we need to understand what an array prefix is.

- A prefix is the set of elements in common, starting at element index 0 (or alternatively an index we define - we'll look at this a bit later).
- If we compare two String arrays that have the exact same elements, the prefix is the entire set of elements.

# Manipulating Arrays

---

First, to understand the compare method, we need to understand what an array prefix is.

- If we compare the firstString with the partialString array in our example, the prefix is the entire partialString since all elements in partialString exist in the firstString array, starting at offset index 0, and in the same order.

```
15 String[] firstString = {"abc", "def", "ghi", "jkl", "mno",
16   "pqr", "stu", "vwx", "yz"};
17 String[] copyOfFirstString = {"abc", "def", "ghi", "jkl", "mno",
18   "pqr", "stu", "vwx", "yz"};
19 String[] firstStringUnsorted = {"jkl", "mno", "pqr", "stu", "vwx",
20   "yz", "abc", "def", "ghi"};
21 String[] partialFirstString = {"abc", "def", "ghi", "jkl", "mno"};
```

# Manipulating Arrays

---

First, to understand the compare method, we need to understand what an array prefix is.

- If we compare the firstString array with firstStringUnsorted array in our example, there is no prefix - there is no common set of elements starting at offset index 0.

```
15     String[] firstString = {"abc", "def", "ghi", "jkl", "mno",
16         "pqr", "stu", "vwx", "yz"};
17     String[] copyOfFirstString = {"abc", "def", "ghi", "jkl", "mno",
18         "pqr", "stu", "vwx", "yz"};
19     String[] firstStringUnsorted = {"jkl", "mno", "pqr", "stu", "vwx",
20         "yz", "abc", "def", "ghi"};
```

# Manipulating Arrays

---

The Arrays.compare method then follows these rules:

- If Arrays.equals is true, return 0.
- If first array passed to parameter is null, return -1, else if second array is null, return 1.
- If length of first array is 0, return (0 - length of second array).
- If length of second array is 0, return (length of first array - 0).

# Manipulating Arrays

---

- If one array represents the entire prefix of another, the difference in lengths of the arrays is returned.
- This number will be negative if you are comparing the smaller array to a larger array.
- If no prefix is identified, then the first element of each array is compared, lexicographically.
- This number will be negative, if the first element of the first array, is less than the first element of the second array.
- If a prefix is identified, but neither arrays represents a full subset of the other, the index where the prefix stops is used to then compare the elements at that index.

# Manipulating Arrays

---

- The following table restates these bullet points:

| array a               | array b               | Compare Result<br><code>Arrays.compare(a, b)</code> | Notes                                                                                              |
|-----------------------|-----------------------|-----------------------------------------------------|----------------------------------------------------------------------------------------------------|
| a = { el1, el2 }      | b = { el1, el2 }      | 0                                                   | arrays equal using <code>Arrays.equals(a,b)</code>                                                 |
| a = null;             | b = { el1, el2 }      | -1                                                  | Note that this is a different result than empty array                                              |
| a = { el1, el2 }      | b = null              | 1                                                   | Note that this is a different result than empty array                                              |
| a = { }               | b = { el1, el2 }      | a.length - b.length                                 | a.length == 0                                                                                      |
| a = { el1, el2 }      | b = { }               | a.length - b.length                                 | b.length == 0                                                                                      |
| a = { el1, el2 }      | b = { el1, el2, el3 } | a.length - b.length                                 | prefix is {el1, el2} which is complete set of one of the arrays.                                   |
| a = { el1, el2 }      | b = { el3, el4 }      | a[0].compareTo(b[0])                                | no prefix identified                                                                               |
| a = { el1, el2, el3 } | b = { el1, el2, el4 } | a[2].compareTo(b[2])                                | prefix is {el1, el2} but this not a complete set of either array.<br>evaluates elements at index 2 |

# Arrays.binarySearch Method

---

The binarySearch method searches for a matching element and returns an integer indicating the array index where there is a match.

If no match is found, the method returns a -1.

When using a binarySearch, your array needs to be sorted - you can perform a binarySearch on a non-sorted array but the results cannot be relied on.

If your array contains duplicate values there is no guarantee which index of the duplicate elements will be returned.

# Arrays.binarySearch Method

---

Remember to sort your array before using this method.

If you want the first match in an array which might have duplicates, use List.indexOf method instead.

# Two Dimensional Arrays

---

In this lecture:

- We've shown you how to create a two-dimensional array in Java, and how to navigate through the dimensions using for loops.
- We've also demonstrated how to clone a two-dimensional array and pointed out that clones and copies are shallow.

# Manipulating Data in Arrays

---

This code goes through most of the `Arrays` methods that alter data in an array (`fill`, `setAll`, `sort`, and `parallelPrefix`).

Some of these methods make use of lambda expressions, which we'll discuss in a later section, but show these methods now, so that you can see there is a great deal of functionality supported now with these methods.

# Manipulating Data in Arrays

---

- I'm not going to demonstrate parallelSetAll or parallelSort - these methods are operationally the same as setAll and sort, but the parallel versions can be used for large sets of data, allowing multiple threads to do the work.
- Since there is no corresponding method to parallelPrefix, I will show that here.
- This method executes a binary operation, in the example shortly, adding the elements together, in parallel, storing the collection in memory until the results are all computed, then it replaces the array with the new values.

# Manipulating Data in Arrays

---

You can see that you can do practically anything you might want to do with an array, using the methods on the `Arrays` Class, with the exception of changing the size of the array itself.

Let's talk about manipulating the array using a fixed-size list, backed by the specified array, by calling `Arrays.asList` method.

This technique allows you to use some methods of the `List` interface, which do not change the size of the array.

# Manipulating Data in Arrays

---

These are List.of and List.copyOf.

The thing to note about both of these methods is that they return an immutable list, meaning you cannot change the list.

# Array Data Transformation Methods

---

We are not going to cover spliterator or stream methods at this time.

These will not be tested on the 1Z0-815 exam.

We already talked about Arrays.asList in our previous code sample, so I'm not going to duplicate that discussion here.

Let's review some of the List data transformation methods.

# Array Data Transformation Methods

---

We have seen that the `java.lang.Arrays` utility class, provides many of the methods you would need to manipulate data in an array.

And additionally, you can use some of the `List` methods, to directly access your array, if you use the `Arrays.asList` transformation method.

The alternative to an array is an `ArrayList`, which is a collection of data that is resizable, and implements the interfaces `List` and `Collection` to name a couple - we'll be discussing `ArrayLists` in a later section.

# Arrays: Out of the Ordinary Concepts - Unboxing

---

First, arrays are fussy about their datatypes.

Although we can rely on Java to autobox to and unbox from wrappers for us in many instances, this is not true for arrays.

# Arrays: Out of the Ordinary Concepts - Unboxing

---

When comparing different typed arrays, they need to implement the Comparable interface, and be cast to a Comparable array, and the compareTo method must compare a super class shared by both.

java.lang.Number does not implement comparable.

# Arrays: Out of the Ordinary Concepts - Lists

---

19  
20

```
List firstList = List.copyOf(Arrays.asList(firstString));  
List secondList = List.of(firstList);
```

The method `List.of` can take an array of elements and create a list out of it, or it can take a variable argument list of objects (var args for short).

Because we did not pass an array to it in the code above, it assumed it was a var args parameter and created an array of 1 element - the first (and only) element being the list we passed as the argument.

This could be very hard to spot on an exam question, whose subject appears to be something else.

# Arrays: Out of the Ordinary Concepts - Summary of Copying

Creating copies/clones of arrays. Here are 5 ways to create a copy of an array...

|                      |                                    | Description                                                                              | Example(s) ; Given:<br><code>String[] stringArray = {"abc", "def", "ghi"};</code>                                                                                                                                                       | Notes                                                                                                                                                                     |
|----------------------|------------------------------------|------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| clone()              | Inherited from Object.             | Creates a new array reference but elements are not cloned (shallow).                     | <code>String[] clonedArray = stringArray.clone();</code>                                                                                                                                                                                | Mutable.<br>Shallow copy.                                                                                                                                                 |
| Arrays.copyOf()      | Static method on java.lang.Arrays. | Creates a new array reference but elements are not cloned (shallow).                     | <code>String[] copiedArray = Arrays.copyOf(stringArray, stringArray.length);</code><br><br><code>String[] copiedArray = Arrays.copyOf(stringArray, 2);</code><br><br><code>String[] copiedArray = Arrays.copyOf(stringArray, 7);</code> | Mutable.<br>Shallow copy.<br><br>Can create array which is truncated from original.<br><br>Can create array larger than original, additional elements get default values. |
| Arrays.copyOfRange() | Static method on java.lang.Arrays. | Creates a new array reference from selected range but elements are not cloned (shallow). | <code>String[] copiedArrayRange = Arrays.copyOfRange(stringArray, 0, 1);</code>                                                                                                                                                         | Mutable.<br>Shallow copy.                                                                                                                                                 |

# Arrays: Out of the Ordinary Concepts - Summary of Copying

Creating copies/clones of arrays. Here are 5 ways to create a copy of an array...

|                  |                          | Description                                                  | Example(s) ; Given:<br><code>String[] stringArray = {"abc", "def", "ghi"};</code>                                                                                     | Notes                                                                                                               |
|------------------|--------------------------|--------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| List.copyOf      | Static method on List.   | Creates a new list.                                          | <code>List&lt;String&gt; listCopy = List.copyOf(Arrays.asList(stringArray));</code><br><code>String[] newArray = listCopy.toArray(new String[0]);</code>              | ** Makes Immutable copy of the List.<br><br>But resulting array from toArray is not immutable.<br><br>Shallow copy. |
| System.arraycopy | Static method on System. | Populates an array reference with a copy of array passed in. | <code>String[] systemClonedArray = new String[stringArray.length];</code><br><code>System.arraycopy(stringArray, 0, systemClonedArray, 0, stringArray.length);</code> | Mutable.<br><br>Shallow copy.                                                                                       |

# Section Introduction

---

I am assuming that if you are preparing for this Java certification exam, you are well versed in the principles of object oriented programming.

We are not going to be discussing these principles abstractly, but discussing them in the context of how the Java language implements them.

For example, I expect that you know that an object is an instantiation of a class, and a class is a custom-defined data type, that contains attributes and methods available to its instances, using the dot operator to access some of these members from the instances.

Here are the items, which the certification exam covers, and which we'll discuss in this section.

# Describing and Using Objects and Classes

- Declare and instantiate Java objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
- Define the structure of a Java class
- Read or write to object fields

DESCRIBING AND USING OBJECTS AND CLASSES  
JSE11DCP1-1-2

# Declare and Instantiate

---

A Java reference variable, is used to address and manipulate an object, and declaring such a variable is basically the same as declaring primitive data type variables, String objects and arrays, without the special constructs those variables have.

Some of the same rules apply.

An object is either known by the compiler by the current context, or needs an import statement, or a Fully Qualified Class Name, to satisfy the compiler.

Multiple variables can be declared in a single statement, as long as they are of the same type.

# Declare and Instantiate

---

| Valid Declaration              | Notes                                                                                                                                                                                      |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Object o;                      | Object does not need the Fully Qualified Class Name, package java.lang is automatically in context for any java class and does not need an import statement or Fully Qualified Class Name. |
| java.util.ArrayList arrayList; | This declaration uses a Fully Qualified Class Name for a variable of type ArrayList.                                                                                                       |
| Object x, y, z;                | Multiple variables can be declared on a single line for a single type.                                                                                                                     |
| Invalid Declaration            | Notes                                                                                                                                                                                      |
| String x, y, z, Object o;      | Trying to declare multiple variables with different types in one statement is not allowed.                                                                                                 |

# Declare and Instantiate

---

Instantiating an object requires, in most cases, the use of the **new** keyword, which we've seen already with primitive data wrappers & String objects.

The **new** keyword for an object, will call a special initialization method called a constructor.

You can define your own constructor(s), or Java may generate a default one for you.

Alternative ways of instantiating a new object are the `clone()` method, which we reviewed in the section on arrays and won't go over here; or deserializing an object, which will not be part of the exam.

# Declare and Instantiate

---

```
Object o = new Object();
```

Is an example of using new to create a new object of type (class) java.lang.Object.

The referenced address to this new object in memory is passed to the variable o.

You can declare an object and instantiate it in the same statement, as we've just demonstrated.

You can declare and instantiate multiple objects in one statement, as long as they are of the same type.

You can chain declarations and instantiations.

# Declare and Instantiate

---

Here are examples of declarations and instantiations that are valid and invalid:

| Valid Statement                                                  | Notes                                                                                                                        |
|------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>Object a = new Object();</code>                            | Straightforward declaration and initialization of single object.                                                             |
| <code>Object a = new Object(), b = new Object();</code>          | Straightforward declaration and initialization of multiple objects of same type.                                             |
| <code>Object b, a = b = new Object();</code>                     | Object b is declared, and a is declared and assigned to b which is assigned a new Object reference.                          |
| <code>Object a = new Object(), s = new String("testing");</code> | This works because s (a String) is declared as an Object, String is a subtype of Object so this compound statement is valid. |

# Declare and Instantiate

---

Here are examples of declarations and instantiations that are valid and invalid:

| Invalid Statement                                                       | Notes                                                                                                                                                                          |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Object a = b, b = new Object();</code>                            | Object b is not actually declared in this statement                                                                                                                            |
| <code>Object a = b = new Object();</code>                               | Object b is not actually declared in this statement                                                                                                                            |
| <code>String s = new String("testing"), a = new Object();</code>        | This does not work because an Object is not a subtype of String                                                                                                                |
| <code>Object a = new Object(), String s = new String("testing");</code> | You cannot define and instantiate classes of different types in same statement, declaring different types for each, even if the different type is a subtype of the first type. |

# Coding Classes and Garbage Collection

---

When you instantiate an object in Java, the Java Virtual Machine allocates memory from the heap, large enough to store the data required for the object.

The address of this memory location is assigned to the declared variable.

This memory stays allocated as long as a variable maintains a reference to it.

You can't actually deallocate memory manually in Java, but you can dereference an object, by setting it to null or assigning a different reference to the variable, allowing the object which lost its reference to be available for garbage collection.

# Coding Classes and Garbage Collection

---

Objects declared locally and instantiated in method blocks, will automatically be dereferenced, when the method returns execution back to the calling thread, unless the object is itself, the returned value of the method.

Objects never assigned to a variable, are immediately available for garbage collection.

# Coding Classes and Garbage Collection

---

One of the benefits of the Java Runtime Environment, is that it frees developers from the complexity of memory management, by providing a garbage collector.

The garbage collector scans for deallocated references, and reclaims that memory.

Garbage collection is a process that could have a big impact on your own application's performance, especially if your application processes large amounts of data.

At some point you may need to examine, and tune garbage collection options.

The exam will not expect you to be an expert on the subject, but may quiz you on the basics.

# Coding Classes and Garbage Collection

---

You can run java.exe with one of the following options, to log some of the garbage collection information:

- -Xlog:gc (-Xlog:gc\* is more verbose and will print any message, that has the gc tag and any other tag).
- -verbose:gc command is an alias for the above -Xlog:gc.

Try running the References class above, with the VM option -Xlog:gc\* set in your Run/Debug configuration setting in IntelliJ.

# Coding Classes and Garbage Collection

---

We'll repeat the garbage collection options in this slide.

| Name                                | VM option to use:     | Description                                                                                                                                                                                                  | Notes                                                                                                                           |
|-------------------------------------|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <b>Serial Collector</b>             | -XX:+UseSerialGC      | The serial collector uses a single thread to perform all garbage collection work.                                                                                                                            | Best-suited to single processor machines.                                                                                       |
| <b>Parallel Collector</b>           | -XX:+UseParallelOldGC | The parallel collector has multiple threads that are used to speed up garbage collection.                                                                                                                    | Intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multi-threaded hardware. |
| <b>Garbage-First (G1) Collector</b> | -XX:+UseG1GC          | The G1 collector is a server-style collector for multiprocessor machines with a large amount of memory. It meets garbage collection pause-time goals with high probability, while achieving high throughput. | <b>Default as of JDK 11.</b>                                                                                                    |

# Coding Classes and Garbage Collection

---

| Name                                         | VM option to use:       | Description                                                                                                                                                                                | Notes                                                                                                                                                          |
|----------------------------------------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Z Garbage Collector</b>                   | -XX:+UseZGC             | The Z Garbage Collector (ZGC) is a scalable low latency garbage collector, performing all expensive work concurrently, without stopping the execution of application threads.              | <b>Introduced with JDK 11.</b><br>Intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes). |
| <b>Concurrent Mark Sweep Collector (CMS)</b> | -XX:+UseConcMarkSweepGC | The Concurrent Mark Sweep Collector (CMS) was for applications that preferred shorter garbage collection pauses and could afford to share processor resources with the garbage collection. | <b>Deprecated as of JDK 9.</b>                                                                                                                                 |

# Defining the Structure of a Class

---

Since an object is an instantiation of a class, it seems appropriate to discuss Java's class structure here.

We've discussed some of this previously in the video on packages and import statements.

# Defining the Structure of a Class

---

So you know that a source file consists of the following sections:

- One or zero package statements. This must be the first line of source code, excluding comments.
- Zero to many import statements. These statements follow the package statement if one exists, and precede any other source code.
- You already know that 'import java.lang.\*;' is implied, if you are using any classes in the java.lang package.

# Defining the Structure of a Class

---

So you know that a source file consists of the following sections:

- If you are creating a class with same name as any class in the java.lang package, you will need to either add a single import statement, where the name conflicts or use Fully Qualified Class Name on the java.lang class.
- We showed an example of this previously.
- One or many type references, where type is class, interface or enum.
- Only one type reference can be public in a single source file, with the exception of nested reference types which we'll discuss later.
- The source file name must match the outermost public type's reference name.

# Defining the Structure of a Class

---

We will talk about interfaces and enum structures later.

For this section, we'll look at the class type reference itself.

# Defining the Structure of a Class

---

A Java class type consists of two sections, the declaration and the class body block.

Blocks are contained in bracket sets {}.

The class declaration requires only the reserved word 'class', and the TypelIdentifier (name of class).

{ClassModifier} class TypelIdentifier [TypeParameters] [Superclass] [Superinterfaces]

The absolute minimal class declaration is shown below:

```
class Test {}
```

# Defining the Structure of a Class

---

Sections of the Declaration are defined below. We will be discussing most of the implementations in future videos.

| Section         | Description                                                                                                                                                                    |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ClassModifier   | Modifiers declare the accessibility of the class as well as additional information about a class ( <code>abstract</code> , <code>static</code> , <code>final</code> ).         |
| TypeParameters  | A class is <i>generic</i> if it declares one or more type variables. This will not be on the first part of the exam.                                                           |
| Superclass      | This section is optional, the <code>extends</code> clause in a normal class declaration specifies the <i>direct superclass</i> of the current class.                           |
| Superinterfaces | This section is optional, the <code>implements</code> clause in a class declaration lists the names of interfaces that are direct superinterfaces of the class being declared. |

{ClassModifier} class TypIdentifier [TypeParameters] [Superclass] [Superinterfaces]

# Defining the Structure of a Class

---

In this slide we talk about the class modifier section's structure and options.

It is a compile-time error if the same keyword appears more than once as a modifier for a class declaration, or if a class declaration has more than one of the access modifiers public, protected, and private.

The class modifier section can declare zero or one access modifier.

Note if no modifier is specified, it is by default package access (also known as package-private).

# Defining the Structure of a Class

---

We'll discuss each of these and their declaration's implications in a later section:

- public.
- protected (only a nested class can be protected).
- private (only a nested class can be protected).

# Defining the Structure of a Class

---

In addition, the class modifier section can declare zero or several of the following modifiers:

- abstract - declaring a class abstract, means it has one or more methods, that are declared abstract. It expects a subclass to implement fully the abstract method(s).
- static - the static modifier pertains only to member classes, and NOT to top level, or local or anonymous classes.

# Defining the Structure of a Class

---

In addition, the class modifier section can declare zero or several of the following modifiers:

- **final** - declares that a class's definition is complete, and no subclasses are desired or required. A final class cannot have its name, in an extends declaration of another class.
- **strictfp** - The effect of the strictfp modifier, is to restrict floating-point calculations, to ensure portability across platforms.

# Defining the Structure of a Class

---

| Valid declarations         | Notes                                                                          |
|----------------------------|--------------------------------------------------------------------------------|
| public class Test          | Java file must be Test.java if this is your declaration.                       |
| public abstract class Test | Declaring a class abstract means you cannot create an object directly from it. |
| static class Test          | You can only declare a class static if it is a member of another class.        |
| public final class Test    | A final class cannot be in an extends clause - it cannot have a subclass.      |

# Defining the Structure of a Class

---

| Invalid declarations         | Notes                                                                                                                                                  |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| public public class Test     | Cannot duplicate modifiers.                                                                                                                            |
| private protected class Test | Cannot have more than one access modifier declared.                                                                                                    |
| final abstract class Test    | final and abstract are mutually exclusive declarations. Abstract requires a class to extend and complete it, Final states that no class can extend it. |

# Valid Class Declarations

---

Finally, we'll review some declaration examples which have modifiers and/or extends/implements declarations, that are valid and invalid:

| Valid declarations                                                    | Notes                                                           |
|-----------------------------------------------------------------------|-----------------------------------------------------------------|
| public class Test extends SuperTest                                   | The Test class is a subclass of the SuperTest class             |
| public abstract class Test extends SuperTest                          | An abstract class can be a subtype of a class, abstract or not. |
| public class Test implements AInterface                               | The Test class implements an interface called Ainterface        |
| public class Test extends SuperTest implements AInterface             | A class can both extend a class and implement an interface.     |
| public class Test extends SuperTest implements AInterface, BInterface | A class can extend a class and implement multiple interfaces.   |

# Invalid Class Declarations

---

| Invalid declarations                                                                                                                                                     | Notes                                                                                                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| public class Test extends SuperTest,<br>SuperExam                                                                                                                        | A class can only extend one class, it can only be a subtype of one supertype.                                     |
| Given<br><pre data-bbox="570 668 862 700">public class SuperTest {}</pre><br>Then<br><pre data-bbox="570 779 1051 811">public class Test implements SuperTest</pre>      | Invalid declaration. You implement an interface, not a class.                                                     |
| Given<br><pre data-bbox="570 917 923 949">public interface AInterface {}</pre><br>Then<br><pre data-bbox="570 1029 1006 1060">public class Test extends AInterface</pre> | Invalid declaration. You implement an interface. You cannot extend an interface unless your type is an interface. |

# Class Body Structure

---

Next, we'll discuss the class body and its structure.

The following image was taken from Oracle's Java Specification.

Page Link:

<https://docs.oracle.com/javase/specs/jls/se11/html/jls-8.html#jls-ClassBody>

*ClassBody:*  
{ {ClassBodyDeclaration} }

*ClassBodyDeclaration:*  
ClassMemberDeclaration  
InstanceInitializer  
StaticInitializer  
ConstructorDeclaration

*ClassMemberDeclaration:*  
FieldDeclaration  
MethodDeclaration  
ClassDeclaration  
InterfaceDeclaration  
;

# Class Body Structure

---

Class Body block rules are as follows:

- The class body may contain declarations of members of the class, that is, fields, methods, classes, and interfaces.
- The class body may also contain instance initializers, static initializers, and declarations of constructors for the class.
- Constructors, static initializers, and instance initializers are NOT members, and therefore are not inherited.

*ClassBody:*

{ [ClassBodyDeclaration](#) }

*ClassBodyDeclaration:*

[ClassMemberDeclaration](#)  
[InstanceInitializer](#)  
[StaticInitializer](#)  
[ConstructorDeclaration](#)

*ClassMemberDeclaration:*

[FieldDeclaration](#)  
[MethodDeclaration](#)  
[ClassDeclaration](#)  
[InterfaceDeclaration](#)

;

# Class Body Structure

Class Body block rules are as follows:

- Declarations can be in any order but with some caveats:
  - The order of declarations is important in the execution of initializers, which are executed in the order they are defined.
  - An initializer block cannot use an unqualified variable declared and initialized after its declaration in a statement, other than making an assignment to the variable.

*ClassBody:*

{ [ClassBodyDeclaration](#) }

*ClassBodyDeclaration:*

[ClassMemberDeclaration](#)  
[InstanceInitializer](#)  
[StaticInitializer](#)  
[ConstructorDeclaration](#)

*ClassMemberDeclaration:*

[FieldDeclaration](#)  
[MethodDeclaration](#)  
[ClassDeclaration](#)  
[InterfaceDeclaration](#)

;

# Class Body Structure

---

Class Body block rules are as follows:

- A static initializer block cannot use an unqualified static variable declared and initialized after its declaration in a statement, other than making an assignment to the variable.
- An instance variable cannot use an unqualified reference to another instance variable, declared after its declaration, in its assignment statement.

*ClassBody:*

{ [ClassBodyDeclaration](#) } }

*ClassBodyDeclaration:*

[ClassMemberDeclaration](#)  
[InstanceInitializer](#)  
[StaticInitializer](#)  
[ConstructorDeclaration](#)

*ClassMemberDeclaration:*

[FieldDeclaration](#)  
[MethodDeclaration](#)  
[ClassDeclaration](#)  
[InterfaceDeclaration](#)  
;

# Class Body Structure

---

Class Body block rules are as follows:

- Fields, methods, and member types of a class type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures.
- However, this is discouraged as a matter of style.

We'll talk about class members and constructors in the next couple of videos, but let's review initializers on our next slide.

```
ClassBody:  
{ {ClassBodyDeclaration} }  
  
ClassBodyDeclaration:  
ClassMemberDeclaration  
InstanceInitializer  
StaticInitializer  
ConstructorDeclaration  
  
ClassMemberDeclaration:  
FieldDeclaration  
MethodDeclaration  
ClassDeclaration  
InterfaceDeclaration  
;
```

# Instance Initializer

---

Instance Initializer - An instance initializer is a block of code declared in a class, whose statements are executed when an instance of the class is created.

- A return statement cannot appear anywhere within an instance initializer.
- You can have multiple initializer blocks, they will be executed in the order they are declared.
- Initializer blocks are executed prior to any constructor block of code being executed, but after any call to a parent type's constructor.
- These blocks are not inherited by subclasses.

# Static Initializer

---

Static Initializer - A static initializer is a block of code declared in a class prefaced by the keyword 'static', whose statements are executed when the class is initialized.

- A return statement cannot appear anywhere within a static initializer.
- The keywords this and super are not permitted in this code section.
- Remember that 'this' and 'super' refer back to instances, and class initializers do not refer to a single instance.
- Any type variable declared outside the static initializer, cannot appear anywhere within a static initializer.

# Static Initializer

---

- You can have multiple static initializer blocks, they will be executed in the order they are declared.
- These blocks are not inherited by subclasses.

# Initializers and Static Initializers in Code

---

We learn from this that we can use instance variables in:

- Statements in the initializer blocks.
- The assignment declarations of instance variables themselves.
- The constructor.

# Initializers and Static Initializers in Code

---

However, this statement is not always true for all instance variable references in all initializer blocks.

For now, it is important that you know the order that the statements are executed.

The exam may quiz you about this order.

Remember, variable and initializer statements are always executed in the same order they are declared, but before any statements in a constructor (with the exception of the super() statement , implied or explicit).

# Initializers and Static Initializers in Code

---

Output is similar to initializer block examples, such that they are executed in the order they are declared.

These statements only get executed when the class itself is initialized.

We will talk a lot more about the static elements of a class in a future video.

# Initializers and Static Initializers in Code

---

In Summary,

- We've discussed the structure of declaring a class.
- Deferring topics on fields and constructors to future videos.
- And I've shown you examples of both initializer blocks and static initializer blocks.

# Out of the Ordinary Class Structures

We showed two examples of code with initializer blocks in the last video.

The first example demonstrated order of statements, and the second demonstrated the use of local variables in the blocks and variable initialization statements.

In that example, we were well behaved, and never tried to use a variable before we had declared it and always referenced it with the 'this' keyword modifier.

Let's see what happens if we relax that discipline. The exam may present sample code that expects you to be knowledgeable of something called forward variable declarations.

# Out of the Ordinary Class Structures

When you execute initializer blocks and constructor methods, you are in a state of not quite fully initialized. Initialization occurs in these steps:

1. Java Virtual Machine checks whether class has been initialized. If not, Java Virtual Machine loads and initializes class.
2. Java Virtual Machine allocates memory to house data for the new instance.
3. Java Virtual Machine initializes instance variables to their default values.
4. Java Virtual Machine executes custom initialization code found in assignment declarations of instance variables, initializer blocks and constructor(s).

# Out of the Ordinary Class Structures

The compiler will not qualify variables with 'this' implicitly, until after it has evaluated all initialization steps.

Generally, it is considered an error to try to access an instance variable you haven't yet declared in initialization code and the error is a useful red flag.

If for some reason, you want to force the issue, and really do mean to just use the default value assigned in step 3, you can do this with the 'this' qualifier.

# Out of the Ordinary Class Structures

In the previous video, we stated initializer blocks are executed prior to the constructor block of code being executed.

This is actually a statement that needs to be clarified a little bit.

If your constructor calls the super constructor (manually or implied), the initializer code is executed after this statement and prior to any other statements in your constructor.

# Read or Write to Object Fields Code Examples

---

Let's walk through some examples in code.

First, we'll look at static variables, or class fields, compared to instance variables.

# Read or Write to Object Fields Code Examples

---

In summary:

- Access fields (static, instance, final or not) both within a class that defines the fields optionally using the this qualifier, and outside using the reference name with the dot operator.
- The fields are accessible outside the class, because of their access modifier which will review in a video coming up.
- We have not discussed transient or volatile fields, which are outside the scope of this exam, except to know their meaning and that they are valid modifiers.

# Section Introduction

---

Methods contain zero to many statements, which define some behavior, which you name (method name) and which can be reused.

A method can accept zero to many parameters (including a variable number of parameters) and can optionally return an element back.

If you've worked with other programming languages, you'll probably want to know if the parameters are passed by value or passed by reference.

Java always passes parameters by value NOT by reference, but it can be very confusing because the value can be a reference.

We'll review this in the next video as well as the items listed below.

# Creating and Using Methods

- Create methods and constructors with arguments and return values
- Create and invoke overloaded methods
- Apply the static keyword to methods and fields

CREATING AND USING METHODS  
JSE11DCP1-1-1

# Structure of a Method Declaration

---

The following is from Oracle's Java 11 specification describing the structure of a method declaration:

{MethodModifier} MethodHeader MethodBody

Method Modifiers are similar to Field Modifiers.

# Method Modifiers

---

## {MethodModifier} MethodHeader MethodBody

- There are the access modifiers (the default modifier is package - no modifier):
  - public
  - protected
  - private

# Method Modifiers

---

## {MethodModifier} MethodHeader MethodBody

- And the non-access modifiers:
  - abstract - declaring a method abstract means it is not implemented, it has no body. A non-abstract method is called a concrete method.
  - static - a static method is called a class method, it can be called using the class name.
  - final - declares that a method's definition is complete and no subclasses can override it or hide it.
  - strictfp - The effect of the strictfp modifier is to restrict floating-point calculations to ensure portability across platforms.

# Method Modifiers

---

**{MethodModifier}** MethodHeader MethodBody

- And the non-access modifiers:
  - synchronized - creates a threadsafe method.
  - native - A method that is native is implemented in platform-dependent code, typically written in another programming language such as C.

# Method Modifier Rules

---

## {MethodModifier} MethodHeader MethodBody

Modifier Rules are as follows:

- A keyword can only appear once as a modifier for a method declaration.
- Only one of the access modifiers public, protected, and private is allowed for a single method.
- If you use the abstract modifier your method can only use public or protected modifier (or no access modifier) and must be defined in an abstract class.
- native and strictfp are not modifiers that can be used in the same declaration.

# Method Header

---

{MethodModifier} **MethodHeader** MethodBody

The Method Header consists of:

- The *result* of a method declaration either declares the type of value that the method returns (the *return type*), or uses the keyword void to indicate that the method does not return a value.
- The name of the method.
- The *formal parameters* of a method or constructor, if any, are specified by a list of comma-separated parameter specifiers.
  - Each parameter specifier consists of a type (optionally preceded by the final modifier) and an identifier that specifies the name of the parameter.

# Method Header

---

{MethodModifier} **MethodHeader** MethodBody

The Method Header consists of:

- If a method or constructor has no formal parameters, and no receiver parameter, then an empty pair of parentheses appears in the declaration of the method or constructor.
- A formal parameter of a method or constructor may be a variable arity (varargs) parameter, indicated by an ellipsis following the type. At most, one varargs parameter is permitted and it has to be in the last position.
- A method can optionally declare a throws clause, to denote any checked exception classes - we'll discuss exceptions in the exceptions section further on in the course.

# Method Declaration Example

---

At a minimum, a method declaration looks as follows:

```
void method()
```

# Valid Method Declarations

---

The following slide shows valid method declarations:

| Valid Declarations                       | Notes                                                                                                                                          |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| void method()                            | This method has no parameters and returns no result.                                                                                           |
| void method(String a)                    | This method has one parameter of type String and returns no result back.                                                                       |
| void method(Object... o)                 | This method has a varargs parameter, meaning you can pass 0 to many Objects in a comma delimited list.                                         |
| Object method()                          | This method has no parameters and returns an object back.                                                                                      |
| String method (final int a, int b)       | This method passes two int parameters, but the final key word prevents the method from changing the value of the parameter in the method code. |
| String method (final int a, final int b) | You can declare multiple parameters final.                                                                                                     |

# Invalid Method Declarations

---

This slide shows examples of invalid method declarations.

| Invalid Declarations                     | Notes                                                                           |
|------------------------------------------|---------------------------------------------------------------------------------|
| method()                                 | Method requires a return type or the keyword void indicating no value returned. |
| void method(String a, b)                 | Each parameter must be defined with its own type.                               |
| void method(Object... o, String a)       | varargs parameter must be the last parameter in a method's parameter list.      |
| void method(Object... o, Object... a)    | You can only have one varargs parameter.                                        |
| String method (final final int a, int b) | duplicate final keyword cannot be used for one parameter.                       |
| String method(int a, Object a)           | You cannot give two parameters the same name.                                   |

# Method Signature

---

Before we talk about the structure of the method's body, it is important to explain a method signature.

A method signature uniquely defines a method.

Two methods with the same signature cannot be defined in one class.

What the signature consists of:

- Method name.
- Ordered list of parameter types.

# Method Signature

---

What the signature does NOT consist of:

- Modifiers
- Return Type
- Parameter Names
- Throws Clause

That sounds very simple and usually it is, but when we talk about overloaded methods, overridden methods, class inheritance, and implementing interfaces, you will see that the definition of the same type can get more complicated.

We'll save these examples for later.

# Method Body

---

{MethodModifier} MethodHeader **MethodBody**

A *method body* is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation.

A concrete method requires brackets.

# Methods - Pass by Value

---

Before we go any further we have to address the statement I made in the introduction of methods, that Java always passes parameters by value NOT by reference.

Pass by value (or call by value) means anything passed into a function or method call is unchanged, in the caller's scope when the method returns.

Call by reference (also referred to as pass by reference) - a method receives an implicit reference to the variable used as a parameter and not a copy of its value.

# Methods - Pass by Value

---

Any modifications to the parameter would be seen by the calling code.

Call by reference can be imitated (purposely or not) in languages that use call by value by making use of references.

This is similar to call by sharing (passing an object, which can then be mutated).

Java is said to have call by sharing. This means that String objects (which are immutable) and primitive data types will not have their values changed (a copy is made and passed to methods) when the method returns to the calling scope.

However, if you are passing an object reference of any kind, a change to the underlying object which the reference refers to will be reflected when the method returns.

# Methods - Pass by Value

---

This is still considered pass by value, because the reference that the method receives in the form of a parameter, is a copy of the reference in the calling scope.

In other words, the reference will not change when returning from the method, however the data on the referenced object may.

# Valid Return Examples

---

The following slide is a review of some of the rules for the return type of methods:

| Valid return examples                                                            | Notes                                                                                   |
|----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <pre>Number getNumber() {<br/>    return Integer.valueOf(1);<br/>}</pre>         | You can return a subtype of declared return type. Here, Integer is a subtype of Number. |
| <pre>Number getNumber() {<br/>    return null;<br/>}</pre>                       | You can return a null for a reference type return type.                                 |
| <pre>long getNumber() {<br/>    int myInt = 2;<br/>    return myInt;<br/>}</pre> | You can return a narrower primitive data type if the return type is a primitive.        |

# Valid Return Examples

---

| Valid return examples                                                   | Notes                                                                                           |
|-------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <pre>long getNumber() {<br/>    return Long.valueOf(10L);<br/>}</pre>   | You can return a primitive data type wrapper if the return type is a primitive.                 |
| <pre>long getNumber() {<br/>    return Integer.valueOf(10);<br/>}</pre> | You can return a wrapper of a primitive to a primitive that is the same or wider.               |
| <pre>public void returnNothing() {<br/>    return;<br/>}</pre>          | You can just have a return statement, returning no reference or primitive, when method is void. |

# Invalid Return Examples

---

| Invalid return examples                                                                          | Notes                                                          |
|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| <pre>Integer getNumber() {<br/>    Number n = Integer.valueOf(10);<br/>    return n;<br/>}</pre> | You cannot return a supertype of the defined return type.      |
| <pre>public void returnNothing() {<br/>    return null;<br/>}</pre>                              | You cannot return null if the return type is void.             |
| <pre>long getNumber() {<br/>    return null;<br/>}</pre>                                         | You cannot return null if return type is primitive.            |
| <pre>int getNumber() {<br/>    return 10L;<br/>}</pre>                                           | You cannot return a primitive that is larger than return type. |

# Constructors

---

Finally, let's talk a little bit about constructors.

A constructor is a special method used in the creation of an object that is an instance of a class.

- The constructor method name must be the same as the class.
- The constructor method does not have any return type, including void.
- The return statement cannot be used in a constructor.

# Constructors

---

- The constructor method is required, but you don't have to declare one.
  - If you declare 0 constructors, Java will create an implicit noargs constructor with the same access modifier as the class and a call to super(); which executes the parent's constructor.
  - If you declare 1 or more constructors, Java will NOT create an implicit noargs constructor for you, even if you do not yourself declare a noargs constructor.

# Method Pass by Value and Constructors

---

It is important to remember two things about constructors:

- If you don't define one, you get one anyway, with no arguments and an implicit call to super().
- If you do declare one, and your class is a child of another class, the implicit call to super() is still there even if you do not explicitly call super() yourself.

# Methods: Out of the Ordinary Concepts

---

Our first example is going to demonstrate why you might want to create a class with no public or package constructors, but only a private constructor.

We'll be talking a lot about access modifiers in an upcoming video, but the private constructor is an interesting test case.

# Overloaded Methods

---

An overloaded method is a method with the same name as another method, but with a different signature, in other words a different parameter set.

There is no limit to the number of overloaded methods you can create.

# 3 Phases of the Method Signature

---

The information on this slide is an extremely simplified synopsis from the Java Specification page link below

<https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.12.2>

There is a lot of information on the specification.

We are going to focus on Section 15.12.2. Compile-Time Step 2: Determine Method Signature, which describes three phases.

# 3 Phases of the Method Signature

---

**Phase 1:** Performs Overload Resolution without permitting boxing or unboxing conversion, or the use of variable arity (var args) method invocation.

Within Phase 1, if multiple methods meet this criteria, the best method is selected by:

- Choose method with Exact Match(es).
- Choose method with More Specific Match(es).

# 3 Phases of the Method Signature

---

- Note that for primitives, subtypes are considered to climb up either of these branches
  - ie char is a subtype of int. byte is not a subtype of char, int can be considered a subtype of double (for method selection)
  - byte, short, int
  - char, int, long, float, double
- If a primitive is passed and a method exists with a parameter that uses widening, this will be selected. A narrowing conversion won't be selected.

# 3 Phases of the Method Signature

---

**Phase 2:** Allows boxing and unboxing, but ignores variable arity (var args) method invocation.

**Phase 3:** Allows overloading to be combined with variable arity (var args) methods, boxing, and unboxing.

Put more simply, var args methods will match dead last, if better options exist.

You may get one question on the exam regarding this selection.

Pay careful attention to the exam question.

It is pretty easy to overlook the ellipsis for the var args parameter, as you are reading 'int...' vs 'int' for example.

# Determining which Overloaded Methods get called

---

Why? Again, the rules described in our slide indicate that parameters won't be evaluated for wrappers with boxing, unboxing until the second phase, IF the first phase completes with no matches.

So even though we think Character is a better match, the Java Virtual Machine won't use it.

The reason for this has to do with backwards compatibility of older code.

You can certainly force the issue, by making the call, an exact match and passing the char in a wrapper yourself.

# Determining which Overloaded Methods get called

---

Finally, since a constructor is in actuality just a method with a few special restrictions, constructors can be overloaded too with the same set of rules applied.

There are only two ways to execute constructors - using new in the process of instantiating a new object and passing the appropriate parameters, or by doing something called constructor chaining, which is calling a different overloaded constructor from another constructor.

Since you can only use this(...) as the first statement in a constructor, you can only create a single chain.

# Applying the Static Keyword

---

You can apply the keyword static to the following elements of a class: field, initializer, method and a nested type (class, interface, enum).

By declaring something static, you are associating the element with the Class and not an instance of the Class.

The elements exist and are available to be used even if you never create a single instance of the Class.

You can not use the modifier static on an outer class itself, or a local variable.

The modifier only has meaning when it describes the relationship to a class and its elements.

# Applying the Static Keyword

Link For Oracle documentation on Static keyword below and in resources section of video.

<https://docs.oracle.com/javase/specs/jls/se11/html/jls-12.html#jls-12.5>

We derive the following scenarios of when a class itself gets initialized:

A class or interface type T will be initialized immediately before the first occurrence of any one of the following:

- T is a class and an instance of T is created.
- A static method declared by T is invoked.

# Applying the Static Keyword

---

- A static field declared by T is assigned.
- A static field declared by T is used and the field is not a constant variable.

These rules are nuanced and we'll show examples of each in the next two videos.

You access a static element (with the exception of an initializer) by calling it with the dot operator on the class name itself.

A constructor cannot be static because a constructor is used in creating an instance, and I've said anything that is modified with the static key word will not be associated with the instance, although it is available for use by the instance.

# Applying the Static Keyword

I've used a static method in almost every example of code I have demonstrated so far (the main method) to execute a class.

The main method is static and callable without executing an instance of the class itself.

I have also reviewed examples of static fields and static initializers briefly.

# Section Introduction

---

In object oriented design, we avoid tight-coupling of classes, which means we try to make them as independent of each other as possible.

When we bundle data and behavior into a single class, we avoid coupling.

Another way to avoid coupling is to prevent classes from directly accessing another class's instance variables.

A well-encapsulated object is considered one that does not expose its fields publicly to other classes, but allows access via methods with appropriate access modifiers.

# Section Introduction

---

This allows the encapsulated object to change all the aspects of a particular field, as well as to protect the data in it.

In this video, we'll cover the following exam topics.

# Applying Encapsulation

- Apply access modifiers
- Apply encapsulation principles to a class

APPLYING ENCAPSULATION  
JSE11DCP1-1-5

# Access Modifiers

---

At the very simplest level, you would identify some class members (fields and internal methods) as private and some as public (methods that a consuming class would need).

The public members would be the 'public face', or the 'contract' shown to the consumers, which need to access an instance of the object for some information, or some function.

The private members would be those fields, or methods, that are needed only for the class to perform it's work, and used only by the class itself.

It gets a bit more complicated than that because we might want a subclass to have access to some internal, private members of it's parent class, especially if we want to create methods that override the parent class.

The protected access modifier was created for this type of access.

# Access Modifiers

---

In addition, if we have a group of classes in a package, it might be assumed that these classes have relationships that aren't strictly parent-child relationships, and that these relationships might require access to some of the internals of a class.

The default modifier, which has a description (package or package-private), but does not have a keyword allows this type of access.

It should also be noted that the protected access modifier, in addition to allowing a subclass access to its members, also allows classes in the same package access to members with the protected modifier.

# Access Modifiers

---

Let's look at a table that shows the modifiers in order of least restrictive to most restrictive:

| Access Modifier | Description                                                                                                                                                                                                        | Available to classes in any package | Available to classes in same package | Available to subclasses |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|--------------------------------------|-------------------------|
| public          | Using the public modifier allows access to a member, regardless of package or hierarchy. Any class in any package can access this type of member.                                                                  | Y                                   | Y                                    | Y                       |
| protected       | Using the protected modifier allows access to any class in the same package. In addition any subclass of the class, regardless of the package it is in, will have access to these members in a restricted fashion. | N                                   | Y                                    | Y                       |
| {no modifier}   | This is the default. When you specify no modifier, the access is package (package-private) or default access.<br><br>This access allows the member to be accessed by any other class in the same package.          | N                                   | Y                                    | N                       |
| private         | Using the private modifier prevents any class, other than the one the member is declared in, from accessing the member. Note that nested classes can access the outer class's private members.                     | N                                   | N                                    | N                       |

# Access Modifiers

---

In this video we will talk about the constructs, the rules and implications of each modifier with the type of member it is applied to.

Unlike methods, fields in a subclass hide those of the same name in the superclass.

# Access Modifiers in Code

---

You can see that three of the modifiers (**public**, **private** and **{no modifier}**) are very straightforward.

The protected access modifier is ripe for tricky exam questions.

We are going to be talking more about the protected modifier when we delve into inheritance further.

# Access Modifiers in Code

---

This section wouldn't be complete if we did not talk about overriding a little bit here, though overriding is part of the inheritance section.

There are different rules to which access modifiers are allowable based on the inherited elements.

You can override a parent's attribute, creating an attribute on the child class, with the same name and type as the parent.

This is called *attribute hiding*, because you are hiding the parent's attribute with the child's.

A child's attribute can be either more restrictive or less restrictive than the parent's.

There are no invalid combinations.

# Access Modifiers in Code

---

You cannot make methods more restrictive on the child or subclass.

Extending a class implies you will be adding functionality to it.

By trying to restrict access on methods, you are attempting the opposite, limiting the functionality of a parent class, which is not permissible.

The parent class created a contract at a public and/or package or protected level, and any class extending it cannot disregard or override the contract.

# When Access Modifiers is not Allowed

---

The example we just talked about demonstrated two examples, where you were restricted from using one of the four access modifiers.

This chart tries to show all the cases where a particular access modifier is not allowed.

| Element                                       | public | protected   | {no modifier} | private     | What no modifier means |
|-----------------------------------------------|--------|-------------|---------------|-------------|------------------------|
| Top Level Types (Class, Interface, Enum)      |        | NOT ALLOWED |               | NOT ALLOWED | package                |
| overriding public method of a parent class    |        | NOT ALLOWED | NOT ALLOWED   | NOT ALLOWED | package                |
| overriding protected method of a parent class |        |             | NOT ALLOWED   | NOT ALLOWED | package                |
| overriding package method of a parent class   |        |             |               | NOT ALLOWED | package                |

# When Access Modifiers is not Allowed

---

| Element                                                                  | public      | protected   | {no modifier} | private                                      | What no modifier means |
|--------------------------------------------------------------------------|-------------|-------------|---------------|----------------------------------------------|------------------------|
| Abstract methods                                                         |             |             |               | NOT ALLOWED                                  | package                |
| Interface methods                                                        |             | NOT ALLOWED |               | (method must be implemented if private used) | public (as of Java 8)  |
| Interface attributes                                                     |             | NOT ALLOWED |               | NOT ALLOWED                                  | public (as of Java 8)  |
| Enum constants (no modifier can be specified, always defaults to public) | NOT ALLOWED | NOT ALLOWED |               | NOT ALLOWED                                  | public                 |
| Enum constructors                                                        | NOT ALLOWED | NOT ALLOWED |               | (redundant if used)                          | private                |

# Apply Access Modifiers: Out of the Ordinary Concepts

---

We explored the concept of a private constructor in a previous video (Section 7 - overloading methods), showing how you can use it to prevent any instances of a particular class from being created.

Let's explore access modifiers and constructors a bit more.

# Apply Access Modifiers: Out of the Ordinary Concepts

---

Removing the protected modifier on LevelTwoClass makes it a package-private constructor.

This constructor is not available to LevelThreeClass even if it extends LevelTwoClass, because LevelThreeClass is not defined in the same package as LevelTwoClass.

# Apply Encapsulation Principles to a Class

---

A bean is a Java class which follows certain programming conventions that follow the JavaBeans guidelines, allowing applications and tools to figure out the bean's properties, methods, and events.

For the certification exam, the convention you need to know is the one to define a property in a bean class, supplying public getter and setter methods.

- A getter method (also known as an accessor method) starts with the prefix 'get' followed by a property name with camel case and returns an element of the property's type - this retrieves the property's data from the bean.
- `getFirstName` for example will get the value of `firstName` from the object.

# Apply Encapsulation Principles to a Class

---

- A setter method starts with the prefix 'set' followed by a property name with camel case and takes as its argument, a value that the property can be set to. The method generally does not return a value.
- A special case for boolean properties allows the accessor method to be defined using prefix 'is' instead of 'get'. For example, the accessor for a boolean property first could be `isFirst();`
- A property is usually an attribute on the class with the same name in the setter/getter methods but this is not a requirement. A property could be any attribute or even a code block of some sort.

# More About Encapsulation

---

The java bean is one way to implement encapsulation in Java, and a standard, but it is not the only way. Look for questions on the exam that expect you to know that encapsulation:

1. Combines data and behavior into a single class
2. Hides and protect attributes by making them private
3. Provides a mechanism that sets attribute(s).
  - a. This mechanism could be a public constructor, a public setter method or some other mechanism.
  - b. If the attribute is a variable reference, you should remember that the underlying object passed to the reference, setting it, could change it. Making a copy of the object referenced is a way to ensure the data does not change

# More About Encapsulation

---

4. Provides a mechanism for getting data from the attribute
  - a. This mechanism could be a standard getter or a public method with any kind of name that returns the data.
  - b. Watch out for methods that return variable references directly, remembering that the underlying object could be altered by the calling code.

# More About Encapsulation

---

- b. If the attribute is a variable reference, you should remember that the underlying object passed to the reference, setting it, could change it. Making a copy of the object referenced is a way to ensure the data does not change
4. Provides a mechanism for getting data from the attribute
- a. This mechanism could be a standard getter or a public method with any kind of name that returns the data.
  - b. Watch out for methods that return variable references directly, remembering that the underlying object could be altered by the calling code.

# Encapsulation Principles

---

I've stated in the introduction to this section that encapsulation means information hiding. And we've learned that Java provides different ways to hide class members.

We can hide a member from every consumer (private), from consumers in other packages (no modifier {package}).

Or we can hide members from other packages but still provide access to classes in the inheritance tree (protected).

# Encapsulation Principles

---

I've stated that there are two objectives when hiding information:

- The first objective is to protect the data of a class from unintended or unwelcome changes.
- The second objective is to protect upstream consumers of the class from unintended consequences, causing a minimum of disruption should the class need to change.

# Encapsulation Principles

---

With this in mind, all class and instance variables should be declared private with access to the data strictly controlled through methods which we can examine and verify the requests being made. This is the convention used by a special type of class called a bean.

# Apply Encapsulation Principles: Out of the Ordinary Concepts

I've mentioned that encapsulation is information hiding and enforcing data integrity around an object.

But implementing encapsulation may not always be perfectly straightforward.

# Section Introduction

---

Inheritance in an object-oriented language allows entities to be modeled and built, from the most general to the most specific type in a hierarchical fashion.

Allowing each class derived from another, to 'inherit' properties and behavior from the parent class, and extend both properties and behavior to be more specific, generally.

# Section Introduction

---

We talked in one of the first videos about inheritance, and described three types of inheritance:

- Inheritance of state - state is defined by the class's static and instance fields.
- Inheritance of implementation - implementation is defined by the behavior of the class, its methods.
- Inheritance of type - an inherited type in Java can be a class or an interface.

# Section Introduction

---

In this section we are going to focus on the first two types of inheritance, which are both inherited from another class.

Java supports only single inheritance of state and implementation, meaning that classes can only extend one class, or be a subclass of one class directly.

The hierarchical tree can certainly be deeper than one level, but a child only has one direct parent or superclass.

# Section Introduction

---

A class that extends another class is a subclass.

A class that can be extended by another class is a superclass.

The ultimate superclass in Java is `java.lang.Object`, from which all classes are implicitly extended.

# Reusing Implementations Through Inheritance

- Create and use subclasses and superclasses
- Create and extend abstract classes
- Enable polymorphism by overriding methods
- Utilize polymorphism to cast and call methods, differentiating object type versus reference type
- Distinguish overloading, overriding, and hiding

REUSING IMPLEMENTATIONS THROUGH INHERITANCE  
JSE11DCP1-1-5

# Subclasses and Superclasses

---

Every class you create is a subclass of another, even if you do not declare it.

Every class in java is a subclass of `java.lang.Object` implicitly, and many of the examples we've used in our videos, have used subclasses and superclasses, but we'll review it formally and briefly here.

# Class Structure

---

To explicitly declare class inheritance, you use the `extends` keyword in the class declaration, an example is shown below.

You already know a class can only extend one class.

```
class SubClass extends SuperClass
```

SubClass in this example is said to be a subclass of the class SuperClass, and can also be said to be a child class or a derived class.

SuperClass is said to be a superclass, or may also be called a base class or a parent class.

# Subclasses and Superclasses

---

```
class SubClass extends SuperClass
```

A subclass is said to be of type SuperClass, and an instance of the subclass can be used anywhere an instance of the superclass can be used.

In other words, the subclass passes the 'IS A' test of object oriented programming.

A subclass IS A SuperClass. For example, if your superclass is Dog, and your subclass is Pug, then it can be said a Pug 'Is A' Dog.

Any application where a Dog is expected, a Pug should work as well.

However, the opposite is not necessarily true. A Dog may not always be a Pug.

# Class Elements

---

A class consists of two types of elements. These are:

- Members: Members include any fields, any methods, and any nested types (classes, enums, interfaces). Members of a class can be inherited by subclasses.
- Elements that are not members: These are constructors, initializer blocks and static initializer blocks. These are not inherited by subclasses.
  - Although constructors are not inherited, there is an implied call to the superclass's constructor if an explicit call is not made. We've discussed this in many samples of code in previous videos.
  - There is no way to call initializer blocks or static initializer blocks on the superclass, explicitly from the subclass.

# Rules for Inheriting Class Members

---

The rules for inheriting class members are as follows:

- A subclass does NOT inherit the private members of its parent.
- A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in.
- If the subclass is in the same package as its parent, it also inherits the package-private members of the parent.

# Inherited Fields Applications

---

You can use the inherited members as is, or replace them, hide them, or supplement them as follows:

- Inherited Fields Can Be:
  - Used directly as is, like any other fields.
  - Hidden by the subclass - the subclass declares a field with the same name as the one in the superclass (hiding is not recommended).
  - Supplemented - the subclass can declare new fields that are not in the superclass.

# Inherited Methods Applications

---

- Inherited Methods Can Be:
  - Used directly as is, like any other methods.
  - Overridden - the subclass can declare a new instance method, that has the same signature and return type as the one in the superclass.
  - Hidden by the subclass - the subclass can declare a new static method, that has the same signature as the one in the superclass.
  - Supplemented - the subclass can declare new methods in the subclass, that are not in the superclass.

# Inheriting Class Members

---

It is important to note that, although we say that a subclass does not inherit private fields of its superclass, the object which is instantiated from the subclass type, is instantiated with the private field variables, of the superclass.

# Creating and Using Subclasses and Superclasses

There was no apparent reason to hide ChipDate on the parent, in fact doing so made executing the method getChipDate() confusing, since it could not return the child's chipDate attribute.

If we had created getChipDate() on Dog, we would have removed our ability to access the parent class's chipDate attribute altogether.

It is generally a good idea to avoid hiding class attributes (including private ones).

# Hiding

---

Let's pause here a moment to talk about the concept of hiding.

- You hide an instance variable, by creating a variable of same name and type on the subclass.
- You hide a static variable, by creating a static variable of same name and type on the subclass.
- You hide a static method, by creating a static method with same name and parameter types on the subclass.
- You do not hide an instance method from a subclass, you override it. We will be talking at length about overriding methods in a future video.

# Hiding

---

Hiding does not necessarily mean that you cannot access the parent's declared members, it just means that if you do not use a qualifier or casting, the JVM will 'see' only the subclass's members, and not the parent's.

In the case of instance variables, your object will maintain placeholders for both the parent's variable and the child's.

# Abstract Class

---

An abstract class is a class that is declared with the `abstract` modifier, and it may or may not include abstract methods.

An abstract class defers some or all of its implementation to its subclasses, which is why the abstract classes themselves cannot be instantiated.

An abstract class is used to define common attributes and behavior, for a set of classes that will extend it.

It is usually modeled after an abstract concept, a type you would be unlikely to create an instance of, but which could be used to describe common features of a set of objects.

# Abstract Class

---

I have showed several examples of an Animal class which was extended by Dog for example.

The Animal class is a perfect candidate for an abstract class, because creating an individual object of Animal is unlikely to happen.

And yet you can identify many common attributes across several types of animals into the Animal Class, as well as common behavior that any type of animal might have (walk, bite, sit, play for example).

A class that is not abstract is called a concrete class.

# Abstract Class

---

The difference between using an abstract class and a concrete class, is the abstract class actually requires a subclass to implement its methods, thereby forcing commonality of interface.

It is similar to an interface in this way, but it also permits providing functionality that subclasses might have in common, and that you would just implement once on the abstract class.

# Abstract Class

---

The absolute minimal declaration of an abstract class is shown below.

```
abstract class AbstractClass {}
```

Any method that is not implemented (i.e. has no method body block) must be declared abstract.

Any class that has even one abstract method must be declared abstract. The reason is probably obvious but I'll state it anyway.

You would not want to create an object that does not have its behavior implemented yet.

There is no such thing as an abstract field.

# Create and Extend Abstract Class

---

In this video, we reviewed abstract classes, which are classes that will never get implemented, but which can be used to force implementation, of a standard set of behavior by any subclass that extends it.

It allows calling classes to be knowledgeable of only the abstract (most general) entity, and use operations on it, without knowledge of the actual behavior that will actually occur at runtime.

We have seen that if a derived class does not implement ALL of the parent's abstract methods, it also must be declared abstract, whether it contains an additional abstract method or not.

# Create and Extend Abstract Class

---

Let's review some rules:

- Any class that has an abstract method must be declared abstract.
- Any subclass must implement ALL of the abstract class's abstract methods, or be declared abstract itself if it does not.
- An abstract class does not actually have to have any abstract methods.
- An abstract class can have implemented methods of any type.
- An abstract class can have attributes, and other inner types, such as class, enum, and interface.

# Create and Extend Abstract Class

---

Let's review some rules:

- An abstract class cannot also be declared as final. The two modifiers are mutually exclusive.
- An abstract class cannot be declared as private, because no other class would be able to extend it.
- An abstract method cannot be declared as private, because no other class would be able to extend it.
- An abstract method cannot be static. A static method cannot be overridden so a static abstract method makes no sense.

# Create and Extend Abstract Class

---

Let's review some rules:

- You cannot create an abstract constructor.



# Abstract Classes: Out of the Ordinary Concepts

---

Your options here are to pass it one or the other of the String parameters, but you cannot get out of calling the Abstract method's one parameter constructor.

This behavior is not limited to an abstract class, but it's interesting here, because the abstract class can force the call of a non-implemented method, in the instantiation of an object, that is forced to implement the method.

# Detailed Static Keyword Example

---

These examples have demonstrated when a class is initialized and how to access static members of a class using the class name qualifier, or the instance reference variable, even when the variable itself is null.

I've stated that you cannot define a static class as a stand-alone class, but it has to be a nested class.

# Section Introduction

---

In this section, we will talk about how Java provides mechanisms to support Encapsulation, a core concept of an object oriented programming language.

Encapsulation has two aspects:

- Information hiding.
- Bundling data and behavior together into a single unit.

# Section Introduction

---

It means providing the outside world that will interact with the object, only the information the outside world needs to know, and not the details of the internal mechanisms.

Proper encapsulation protects the integrity of data in the object, preventing abuses and alterations from outside the object.

It also allows implementation details to change, without effecting the consumers upstream of the object.

As long as the 'contract' that was published does not change, an application should not have to change, even if the object's implementation does.

# Section Introduction

---

In object oriented design, we avoid tight-coupling of classes, which means we try to make them as independent of each other as possible.

When we bundle data and behavior into a single class, we avoid coupling.

Another way to avoid coupling is to prevent classes from directly accessing another class's instance variables.

A well-encapsulated object is considered one that does not expose its fields publicly to other classes, but allows access via methods with appropriate access modifiers.

# Section Introduction

---

This allows the encapsulated object to change all the aspects of a particular field, as well as to protect the data in it.

In this video, we'll cover the following exam topics.

# Applying Encapsulation

- Apply access modifiers
- Apply encapsulation principles to a class

APPLYING ENCAPSULATION  
JSE11DCP1-1-5

# Access Modifiers

---

At the very simplest level, you would identify some class members (fields and internal methods) as private and some as public (methods that a consuming class would need).

The public members would be the 'public face', or the 'contract' shown to the consumers, which need to access an instance of the object for some information, or some function.

The private members would be those fields, or methods, that are needed only for the class to perform it's work, and used only by the class itself.

It gets a bit more complicated than that because we might want a subclass to have access to some internal, private members of it's parent class, especially if we want to create methods that override the parent class.

The protected access modifier was created for this type of access.

# Access Modifiers

---

In addition, if we have a group of classes in a package, it might be assumed that these classes have relationships that aren't strictly parent-child relationships, and that these relationships might require access to some of the internals of a class.

The default modifier, which has a description (package or package-private), but does not have a keyword allows this type of access.

It should also be noted that the protected access modifier, in addition to allowing a subclass access to its members, also allows classes in the same package access to members with the protected modifier.

# Access Modifiers

---

Let's look at a table that shows the modifiers in order of least restrictive to most restrictive:

| Access Modifier | Description                                                                                                                                                                                                        | Available to classes in any package | Available to classes in same package | Available to subclasses |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|--------------------------------------|-------------------------|
| public          | Using the public modifier allows access to a member, regardless of package or hierarchy. Any class in any package can access this type of member.                                                                  | Y                                   | Y                                    | Y                       |
| protected       | Using the protected modifier allows access to any class in the same package. In addition any subclass of the class, regardless of the package it is in, will have access to these members in a restricted fashion. | N                                   | Y                                    | Y                       |
| {no modifier}   | This is the default. When you specify no modifier, the access is package (package-private) or default access.<br><br>This access allows the member to be accessed by any other class in the same package.          | N                                   | Y                                    | N                       |
| private         | Using the private modifier prevents any class, other than the one the member is declared in, from accessing the member. Note that nested classes can access the outer class's private members.                     | N                                   | N                                    | N                       |

# Access Modifiers

---

In this video we will talk about the constructs, the rules and implications of each modifier with the type of member it is applied to.

Unlike methods, fields in a subclass hide those of the same name in the superclass.

# Access Modifiers in Code

---

You can see that three of the modifiers (**public, private and {no modifier}**) are very straightforward.

The protected access modifier is ripe for tricky exam questions.

We are going to be talking more about the protected modifier when we delve into inheritance further.

# Access Modifiers in Code

---

This section wouldn't be complete if we did not talk about overriding a little bit here, though overriding is part of the inheritance section.

There are different rules to which access modifiers are allowable based on the inherited elements.

You can override a parent's attribute, creating an attribute on the child class, with the same name and type as the parent.

This is called *attribute hiding*, because you are hiding the parent's attribute with the child's.

A child's attribute can be either more restrictive or less restrictive than the parent's.

There are no invalid combinations.

# Access Modifiers in Code

---

You cannot make methods more restrictive on the child or subclass.

Extending a class implies you will be adding functionality to it.

By trying to restrict access on methods, you are attempting the opposite, limiting the functionality of a parent class, which is not permissible.

The parent class created a contract at a public and/or package or protected level, and any class extending it cannot disregard or override the contract.

# When Access Modifiers is not Allowed

---

The example we just talked about demonstrated two examples, where you were restricted from using one of the four access modifiers.

This chart tries to show all the cases where a particular access modifier is not allowed.

| Element                                       | public | protected   | {no modifier} | private     | What no modifier means |
|-----------------------------------------------|--------|-------------|---------------|-------------|------------------------|
| Top Level Types (Class, Interface, Enum)      |        | NOT ALLOWED |               | NOT ALLOWED | package                |
| overriding public method of a parent class    |        | NOT ALLOWED | NOT ALLOWED   | NOT ALLOWED | package                |
| overriding protected method of a parent class |        |             | NOT ALLOWED   | NOT ALLOWED | package                |
| overriding package method of a parent class   |        |             |               | NOT ALLOWED | package                |

# When Access Modifiers is not Allowed

---

| Element                                                                  | public      | protected   | {no modifier} | private                                      | What no modifier means |
|--------------------------------------------------------------------------|-------------|-------------|---------------|----------------------------------------------|------------------------|
| Abstract methods                                                         |             |             |               | NOT ALLOWED                                  | package                |
| Interface methods                                                        |             | NOT ALLOWED |               | (method must be implemented if private used) | public (as of Java 8)  |
| Interface attributes                                                     |             | NOT ALLOWED |               | NOT ALLOWED                                  | public (as of Java 8)  |
| Enum constants (no modifier can be specified, always defaults to public) | NOT ALLOWED | NOT ALLOWED |               | NOT ALLOWED                                  | public                 |
| Enum constructors                                                        | NOT ALLOWED | NOT ALLOWED |               | (redundant if used)                          | private                |

# Apply Access Modifiers: Out of the Ordinary Concepts

---

We explored the concept of a private constructor in a previous video ( Section 7 - overloading methods), showing how you can use it to prevent any instances of a particular class from being created.

Let's explore access modifiers and constructors a bit more.

# Apply Access Modifiers: Out of the Ordinary Concepts

---

Removing the protected modifier on LevelTwoClass makes it a package-private constructor.

This constructor is not available to LevelThreeClass even if it extends LevelTwoClass, because LevelThreeClass is not defined in the same package as LevelTwoClass.

# Apply Encapsulation Principles to a Class

---

A bean is a Java class which follows certain programming conventions that follow the JavaBeans guidelines, allowing applications and tools to figure out the bean's properties, methods, and events.

For the certification exam, the convention you need to know is the one to define a property in a bean class, supplying public getter and setter methods.

- A getter method (also known as an accessor method) starts with the prefix 'get' followed by a property name with camel case and returns an element of the property's type - this retrieves the property's data from the bean.
- `getFirstName` for example will get the value of `firstName` from the object.

# Apply Encapsulation Principles to a Class

---

- A setter method starts with the prefix 'set' followed by a property name with camel case and takes as its argument, a value that the property can be set to. The method generally does not return a value.
- A special case for boolean properties allows the accessor method to be defined using prefix 'is' instead of 'get'. For example, the accessor for a boolean property first could be `isFirst();`
- A property is usually an attribute on the class with the same name in the setter/getter methods but this is not a requirement. A property could be any attribute or even a code block of some sort.

# More About Encapsulation

---

The java bean is one way to implement encapsulation in Java, and a standard, but it is not the only way. Look for questions on the exam that expect you to know that encapsulation:

1. Combines data and behavior into a single class
2. Hides and protect attributes by making them private
3. Provides a mechanism that sets attribute(s).
  - a. This mechanism could be a public constructor, a public setter method or some other mechanism.
  - b. If the attribute is a variable reference, you should remember that the underlying object passed to the reference, setting it, could change it. Making a copy of the object referenced is a way to ensure the data does not change

# More About Encapsulation

---

4. Provides a mechanism for getting data from the attribute
  - a. This mechanism could be a standard getter or a public method with any kind of name that returns the data.
  - b. Watch out for methods that return variable references directly, remembering that the underlying object could be altered by the calling code.

# More About Encapsulation

---

- b. If the attribute is a variable reference, you should remember that the underlying object passed to the reference, setting it, could change it. Making a copy of the object referenced is a way to ensure the data does not change
4. Provides a mechanism for getting data from the attribute
- a. This mechanism could be a standard getter or a public method with any kind of name that returns the data.
  - b. Watch out for methods that return variable references directly, remembering that the underlying object could be altered by the calling code.

# Encapsulation Principles

---

I've stated in the introduction to this section that encapsulation means information hiding. And we've learned that Java provides different ways to hide class members.

We can hide a member from every consumer (private), from consumers in other packages (no modifier {package}).

Or we can hide members from other packages but still provide access to classes in the inheritance tree (protected).

# Encapsulation Principles

---

I've stated that there are two objectives when hiding information:

- The first objective is to protect the data of a class from unintended or unwelcome changes.
- The second objective is to protect upstream consumers of the class from unintended consequences, causing a minimum of disruption should the class need to change.

# Encapsulation Principles

---

With this in mind, all class and instance variables should be declared private with access to the data strictly controlled through methods which we can examine and verify the requests being made. This is the convention used by a special type of class called a bean.

# Apply Encapsulation Principles: Out of the Ordinary Concepts

I've mentioned that encapsulation is information hiding and enforcing data integrity around an object.

But implementing encapsulation may not always be perfectly straightforward.

# Section Introduction

---

Inheritance in an object-oriented language allows entities to be modeled and built, from the most general to the most specific type in a hierarchical fashion.

Allowing each class derived from another, to 'inherit' properties and behavior from the parent class, and extend both properties and behavior to be more specific, generally.

# Section Introduction

---

We talked in one of the first videos about inheritance, and described three types of inheritance:

- Inheritance of state - state is defined by the class's static and instance fields.
- Inheritance of implementation - implementation is defined by the behavior of the class, its methods.
- Inheritance of type - an inherited type in Java can be a class or an interface.

# Section Introduction

---

In this section we are going to focus on the first two types of inheritance, which are both inherited from another class.

Java supports only single inheritance of state and implementation, meaning that classes can only extend one class, or be a subclass of one class directly.

The hierarchical tree can certainly be deeper than one level, but a child only has one direct parent or superclass.

# Section Introduction

---

A class that extends another class is a subclass.

A class that can be extended by another class is a superclass.

The ultimate superclass in Java is `java.lang.Object`, from which all classes are implicitly extended.

# Reusing Implementations Through Inheritance

- Create and use subclasses and superclasses
- Create and extend abstract classes
- Enable polymorphism by overriding methods
- Utilize polymorphism to cast and call methods, differentiating object type versus reference type
- Distinguish overloading, overriding, and hiding

REUSING IMPLEMENTATIONS THROUGH INHERITANCE  
JSE11DCP1-1-5

# Subclasses and Superclasses

---

Every class you create is a subclass of another, even if you do not declare it.

Every class in java is a subclass of `java.lang.Object` implicitly, and many of the examples we've used in our videos, have used subclasses and superclasses, but we'll review it formally and briefly here.

# Class Structure

---

To explicitly declare class inheritance, you use the `extends` keyword in the class declaration, an example is shown below.

You already know a class can only extend one class.

```
class SubClass extends SuperClass
```

SubClass in this example is said to be a subclass of the class SuperClass, and can also be said to be a child class or a derived class.

SuperClass is said to be a superclass, or may also be called a base class or a parent class.

# Subclasses and Superclasses

---

```
class SubClass extends SuperClass
```

A subclass is said to be of type SuperClass, and an instance of the subclass can be used anywhere an instance of the superclass can be used.

In other words, the subclass passes the 'IS A' test of object oriented programming.

A subclass IS A SuperClass. For example, if your superclass is Dog, and your subclass is Pug, then it can be said a Pug 'Is A' Dog.

Any application where a Dog is expected, a Pug should work as well.

However, the opposite is not necessarily true. A Dog may not always be a Pug.

# Class Elements

---

A class consists of two types of elements. These are:

- Members: Members include any fields, any methods, and any nested types (classes, enums, interfaces). Members of a class can be inherited by subclasses.
- Elements that are not members: These are constructors, initializer blocks and static initializer blocks. These are not inherited by subclasses.
  - Although constructors are not inherited, there is an implied call to the superclass's constructor if an explicit call is not made. We've discussed this in many samples of code in previous videos.
  - There is no way to call initializer blocks or static initializer blocks on the superclass, explicitly from the subclass.

# Rules for Inheriting Class Members

---

The rules for inheriting class members are as follows:

- A subclass does NOT inherit the private members of its parent.
- A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in.
- If the subclass is in the same package as its parent, it also inherits the package-private members of the parent.

# Inherited Fields Applications

---

You can use the inherited members as is, or replace them, hide them, or supplement them as follows:

- Inherited Fields Can Be:
  - Used directly as is, like any other fields.
  - Hidden by the subclass - the subclass declares a field with the same name as the one in the superclass (hiding is not recommended).
  - Supplemented - the subclass can declare new fields that are not in the superclass.

# Inherited Methods Applications

---

- Inherited Methods Can Be:
  - Used directly as is, like any other methods.
  - Overridden - the subclass can declare a new instance method, that has the same signature and return type as the one in the superclass.
  - Hidden by the subclass - the subclass can declare a new static method, that has the same signature as the one in the superclass.
  - Supplemented - the subclass can declare new methods in the subclass, that are not in the superclass.

# Inheriting Class Members

---

It is important to note that, although we say that a subclass does not inherit private fields of its superclass, the object which is instantiated from the subclass type, is instantiated with the private field variables, of the superclass.

# Creating and Using Subclasses and Superclasses

There was no apparent reason to hide ChipDate on the parent, in fact doing so made executing the method getChipDate() confusing, since it could not return the child's chipDate attribute.

If we had created getChipDate() on Dog, we would have removed our ability to access the parent class's chipDate attribute altogether.

It is generally a good idea to avoid hiding class attributes (including private ones).

# Hiding

---

Let's pause here a moment to talk about the concept of hiding.

- You hide an instance variable, by creating a variable of same name and type on the subclass.
- You hide a static variable, by creating a static variable of same name and type on the subclass.
- You hide a static method, by creating a static method with same name and parameter types on the subclass.
- You do not hide an instance method from a subclass, you override it. We will be talking at length about overriding methods in a future video.

# Hiding

---

Hiding does not necessarily mean that you cannot access the parent's declared members, it just means that if you do not use a qualifier or casting, the JVM will 'see' only the subclass's members, and not the parent's.

In the case of instance variables, your object will maintain placeholders for both the parent's variable and the child's.

# Abstract Class

---

An abstract class is a class that is declared with the `abstract` modifier, and it may or may not include abstract methods.

An abstract class defers some or all of its implementation to its subclasses, which is why the abstract classes themselves cannot be instantiated.

An abstract class is used to define common attributes and behavior, for a set of classes that will extend it.

It is usually modeled after an abstract concept, a type you would be unlikely to create an instance of, but which could be used to describe common features of a set of objects.

# Abstract Class

---

I have showed several examples of an Animal class which was extended by Dog for example.

The Animal class is a perfect candidate for an abstract class, because creating an individual object of Animal is unlikely to happen.

And yet you can identify many common attributes across several types of animals into the Animal Class, as well as common behavior that any type of animal might have (walk, bite, sit, play for example).

A class that is not abstract is called a concrete class.

# Abstract Class

---

The difference between using an abstract class and a concrete class, is the abstract class actually requires a subclass to implement its methods, thereby forcing commonality of interface.

It is similar to an interface in this way, but it also permits providing functionality that subclasses might have in common, and that you would just implement once on the abstract class.

# Abstract Class

---

The absolute minimal declaration of an abstract class is shown below.

```
abstract class AbstractClass {}
```

Any method that is not implemented (i.e. has no method body block) must be declared abstract.

Any class that has even one abstract method must be declared abstract. The reason is probably obvious but I'll state it anyway.

You would not want to create an object that does not have its behavior implemented yet.

There is no such thing as an abstract field.

# Create and Extend Abstract Class

---

In this video, we reviewed abstract classes, which are classes that will never get implemented, but which can be used to force implementation, of a standard set of behavior by any subclass that extends it.

It allows calling classes to be knowledgeable of only the abstract (most general) entity, and use operations on it, without knowledge of the actual behavior that will actually occur at runtime.

We have seen that if a derived class does not implement ALL of the parent's abstract methods, it also must be declared abstract, whether it contains an additional abstract method or not.

# Create and Extend Abstract Class

---

Let's review some rules:

- Any class that has an abstract method must be declared abstract.
- Any subclass must implement ALL of the abstract class's abstract methods, or be declared abstract itself if it does not.
- An abstract class does not actually have to have any abstract methods.
- An abstract class can have implemented methods of any type.
- An abstract class can have attributes, and other inner types, such as class, enum, and interface.

# Create and Extend Abstract Class

---

Let's review some rules:

- An abstract class cannot also be declared as final. The two modifiers are mutually exclusive.
- An abstract class cannot be declared as private, because no other class would be able to extend it.
- An abstract method cannot be declared as private, because no other class would be able to extend it.
- An abstract method cannot be static. A static method cannot be overridden so a static abstract method makes no sense.

# Create and Extend Abstract Class

---

Let's review some rules:

- You cannot create an abstract constructor.

# Abstract Classes: Out of the Ordinary Concepts

---

In the last video we talked about some of the rules regarding abstract classes.

I said that an abstract class cannot be declared final, but I did **NOT** say you cannot have final methods on your abstract class.

# Abstract Classes: Out of the Ordinary Concepts

---

Your options here are to pass it one or the other of the String parameters, but you cannot get out of calling the Abstract method's one parameter constructor.

This behavior is not limited to an abstract class, but it's interesting here, because the abstract class can force the call of a non-implemented method, in the instantiation of an object, that is forced to implement the method.

# Polymorphism

---

Polymorphism simply means many forms. In the Object Oriented world, an object can be many things (types).

It can be treated as it's declared type, it's inherited type, it's parent's inherited type, a `java.lang.Object`, or any one of the interfaces it implements.

I touched on polymorphism in a previous video, when we implemented the drive and park methods, on different types of Vehicles.

I'll demonstrate this principle in another example, using the Vehicle classes we created previously.

# Polymorphism

---

What exactly did we just demonstrate?

- That Vehicle published a public interface that told all consumers, that any Vehicle (or subclass of Vehicle) would have the following behavior: drive, park and makeNoise.
- That each unique subclass of Vehicle, could customize its implementation of these methods (override the methods) and do its own specific thing.
- That the print method in TestVehicles, can now execute the print method, on any type of Vehicle passed into the code.
- There is no need for the calling code to know anything more about the specifics of the object passed to it.

# Polymorphism

---

We also demonstrated....

- That although we pass a Vehicle to the methods on TestVehicles, the objects passed were different and concrete incarnations of Vehicle.
- Executing the contractual methods drive, park, and makeNoise, on a parameter object of type Vehicle, actually executed the overridden method of the subclass.

# Difference Between Overriding and Overloading Methods

---

The key to implementing this rather elegant way of writing code, is the combination of inheritance and overriding methods, that are defined on parent classes or interfaces.

Let's look at the differences between overriding methods, and overloading methods.

First, overloading methods is done within a single class, generally, to support similar behavior, with different sets of parameters or inputs.

Overriding methods is declaring a method in the current class, that is the same as a method in a parent class (either abstract or concrete), and creating its own custom behavior of the named method.

# Difference Between Overriding and Overloading Methods

This table will help you remember the differences between overloaded and overridden methods:

| Rule                                                                                  | Overloaded | Overridden |
|---------------------------------------------------------------------------------------|------------|------------|
| Method must have the same name.                                                       | TRUE       | TRUE       |
| Method must have the same number of parameters.                                       | FALSE      | TRUE       |
| Method must have the same types of parameters in the same order. Names do not matter. | FALSE      | TRUE       |

# Difference Between Overriding and Overloading Methods

| Rule                                                                                                         | Overloaded                  | Overridden                                                                                             |
|--------------------------------------------------------------------------------------------------------------|-----------------------------|--------------------------------------------------------------------------------------------------------|
| Method must have the same return type or a covariant of the type. Primitive return types must exactly match. | Irrelevant in determination | TRUE                                                                                                   |
| Method must have same access or less restrictive.                                                            | Irrelevant in determination | TRUE                                                                                                   |
| Throws clause.                                                                                               | Irrelevant in determination | Not required to have a Throws Clause, but if included, the exception cannot be wider or less specific. |

# Difference Between Overriding and Overloading Methods

---

The similarities between an overridden method and an overloaded method, are actually not many.

The methods have the same name as another method, but that is basically where the similarities end.

An overridden method will have the same name of a method, defined on a parent class or an interface.

An overloaded method can have the same name as a method in its own class, or can overload a method on a parent class or interface.

# Difference Between Overriding and Overloading Methods

---

Overloaded methods have much fewer restrictions, because they are basically new methods in the class they are declared in, that happen to have the same method name as another, but a different parameter list.

When you override a method, you are intentionally (one would hope) replacing or extending another method that exists in another class, either because you want to, or because you are forced to.

And it is the overridden method's particular signature, that allows a calling class to be agnostic about the class that is passed to it.

# Polymorphism Code

---

Maybe the `goodMethod` is good, and the derived class is happy with it, or maybe the derived class has a better way to do it, or they want to add additional stuff (extend the code in the method).

The way for `ExtendedClass` to do this is by overriding.

An overridden method in `ExtendedClass` must have the same signature, the same return type, and its access modifier must not be more restrictive.

# Polymorphism Code

---

This video has reviewed the subject of polymorphism and how inheritance and overridden methods support polymorphism in Java.

I discussed the differences between overloaded and overridden methods, and we went through the rules for proper overriding.

If you do not override correctly, you could get a compiler error or no error, because your method is a valid overloaded method instead.

# Polymorphism Casting Object vs Reference

I mentioned in one of the earliest videos, that Java is said to be a strongly typed programming language.

You declare variables as a particular type, and you cannot change the declared type, though it is possible to assign an object to your declared variable that is a different type.

# Polymorphism Casting Object vs Reference

We've already seen examples like the following that are perfectly valid:

```
// String Literal assigned to an Object reference  
Object o1 = "Hello";  
  
// StringBuilder object assigned to an Object reference  
Object o2 = new StringBuilder("hello");  
  
// int 10 will get boxed to Integer wrapper and assigned  
// to Object Reference  
Object o3 = 10;
```

# Polymorphism Casting Object vs Reference

You cannot change the actual type of the reference variable.

The following produces an error. o3 cannot have its type changed.

```
// int 10 will get boxed to Integer wrapper and assigned  
// to Object Reference  
Object o3 = 10;  
  
Integer o3 = 10;
```

# Polymorphism Casting Object vs Reference

You also know that the compiler does some type checking at compile time.

You cannot assign an object that is not a derivative of the declared type to the variable.

For example:

```
// You cannot assign an object to a variable declared of a type  
// where there is nothing in common between the types at all  
Thread t = "hello";  
  
// Both String and StringBuilder are subclasses of CharSequence  
// but StringBuilder is not derived from String so this is not valid  
String s = new StringBuilder("hello");
```

# Polymorphism Casting Object vs Reference

We've seen instances where we think an assignment should work but it doesn't.

The compiler won't let us do the following:

```
short s = 10;
```

```
// s+10 should be a valid short value since 20 fits  
// in a short's range.  
short s2 = s + 10;
```

We know it will work, but the compiler isn't going to execute code to figure it out for us.

# Polymorphism Casting

Polymorphism occurs at runtime, not at compile time, and sometimes we need the ability to force an object into a different form of itself, to produce results we want.

We need this code to compile, so we need a way to inform the compiler to ignore some of its type-checking rules. We do this by using casting.

You can use casting:

- In assignments.
- In expressions.
- In passing objects to method calls.

# Polymorphism Casting

Casting can go in either direction.

- Downcasting is casting to a more specific type from the defined type. The downcast must meet the 'IsA' criteria.
- Upcasting is casting to a more generic type.
- We will talk about casting and interfaces in a future video on interfaces.

# Polymorphism Casting

The methods execute using the reference types we assigned the objects to.

When we assigned a Dog object to a Dog reference, the call to the testAnimal method used the overloaded method with a Dog parameter.

When we assigned a Dog object to an Animal reference, the call to the testAnimal method used the overloaded method with an Animal parameter.

# Polymorphism: Out of the Ordinary Concepts Casting

In this video, we want to explore casting specifically for arrays and generics.

We haven't discussed generics yet - they are not officially part of the first certification exam, but you may bump into them in some of the example code on the exam, so we'll review them ever so briefly in the context of casting.

# Polymorphism: Out of the Ordinary Concepts Generics

We haven't discussed generics and they are officially part of the the second exam i.e. 1Z0-816, and not the exam we are preparing for in this course, the 1Z0-815.

But you may encounter some sample code in the first exam that uses them.

The term generics is used because it's purpose was to implement more generic programming, extending Java's type system to allow "a type or method to operate on objects of various types while providing compile-time type safety".

From the example code we just walked through, the objective would be to prevent adding a StringBuilder object in a collection, we want to contain only NextClass objects, by having the compiler recognize the problem and flag it as an error.

# Polymorphism: Out of the Ordinary Concepts Generics

The Java collections framework supports generics to specify the type of objects stored in a collection instance.

A generic type in Java is one that is parameterized with (in the case of Collections) the type of the objects, which will be in the collection.

You can parameterize any class or interface as well as methods.

We are not going to discuss how to create a generically typed type, or a generically typed method.

For this exercise all you need to know is the syntax of using one.

# Syntax for Declaring and Instantiating a Generic Variable

The syntax for declaring a variable for a generic type and instantiating it is shown below:

```
{GenericType} <TYPE> varname = new {GenericType}<TYPE>();
```

```
ArrayList<String> list = new ArrayList<String>();
```

# Syntax for Declaring and Instantiating a Generic Variable

---

```
{GenericTypeName}<TYPE> varname = new {GenericTypeName}<TYPE>();
```

Alternatively you can omit the specific passed type on the right of the assignment, and just use the 'diamond operator'.

This is short hand and can be used because the type is known from the declaration.

```
ArrayList<String> list = new ArrayList<>();
```

# Syntax for Declaring and Instantiating a Generic Variable

---

Both of these statements declare a variable of ArrayList, which will house **ONLY** String objects, and assign a new instance of an ArrayList (typed to String) to the variable.

```
ArrayList<String> list = new ArrayList<String>();
```

```
ArrayList<String> list = new ArrayList<>();
```

We'll be talking a lot about the ArrayList but for now, realize it's an object that implements the List interface, and maintains an array.

# Syntax for Declaring and Instantiating a Generic Variable

---

The only other thing you need to know about generics at this level, is that they do not support primitive data types.

The code below is invalid.

```
ArrayList<int> list = new ArrayList<int>();
```

# Syntax for Declaring and Instantiating a Generic Variable

---

There are actually two types of ArrayList, the raw (pre-generics) ArrayList and the typed ArrayList.

This is true of many of the Collections classes and interfaces in java.

Let's compare them side by side.

# Polymorphism: Out of the Ordinary Concepts Generics

What did we learn?

- You can assign a typed generic object to a raw variable type.
- But you cannot use the 'Is A' test for casting, calling methods, or assignments of a parameterized type.
- An exact match to the parameterized type is required.

# Polymorphism: Out of the Ordinary Concepts Generics

In this video, we've reviewed casting examples around arrays and generics, and learned that:

- A subclass IsA superclass, but an array of subclass (subclass[]), cannot be said to be a array of superclass for casting decisions.
- A generic typed to a subclass also cannot be used, in place of a generic type to its superclass.

# Interfaces

---

An interface is a way to describe a behavior, or group of behaviors that is shared among disparate types of objects.

You are probably familiar with several of Java's supplied interfaces, such as `java.lang.Comparable`, `java.io.Serializable`, and `java.util.List`.

Any object can implement these interfaces, and supply code for the required behavior (methods) to complete the implementation.

# Interfaces

---

Once a class implements an interface, it can be treated as an object, with the type of that interface in any method calls.

In other words, once we implement Comparable in any of our own classes, we can pass instances of our Comparable classes to any methods, that specify a Comparable parameter type.

A class can implement many interfaces.

# Types of Inheritance

---

We will repeat the three types of inheritance once more here.

- Inheritance of state - state is defined by the class's static and instance fields.
- Inheritance of implementation - implementation is defined by the behavior of the class, its methods.
- Inheritance of type - an inherited type in Java can be a class or an interface.

Java is said to support multiple inheritance of type, because it allows a class to both extend another class and implement multiple interfaces.

In this section we'll be covering the following topics.

# Programming Abstractly Through Interfaces

- Create and implement interfaces
- Distinguish class inheritance from interface inheritance including abstract classes
- Declare and use List and ArrayList instances
- Understanding lambda Expressions

PROGRAMMING ABSTRACTLY THROUGH INTERFACES  
JSE11DCP1-1-4

# Creating and Implementing Interfaces

In the Java programming language, an interface is one of the three reference types supported, along with class and enum.

An interface can contain the following elements: All members are implicitly public and cannot be declared with any other access modifier, unless specified below:

- fields - all variables are public static final implicitly.
- method signatures
- default methods (permitted as of Java 8) - methods declared with 'default' modifier.
- static methods (permitted as of Java 8)

# Creating and Implementing Interfaces

---

- private methods (permitted as of Java 9), both static and non-static concrete methods can be private.
- Nested types
- Method bodies exist only for default, private and static methods.

An interface that declares one and only one abstract method, is called a functional interface.

# Differences Between Classes and Interfaces

---

The following table will hopefully help you quickly see the differences between classes and interfaces.

| Feature                                                      | Class                                       | Interface                                                                                                                                                                                        |
|--------------------------------------------------------------|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Inheritance                                                  | A class can extend only one class           | A class can implement multiple interfaces.<br>An interface can extend multiple interfaces.                                                                                                       |
| Default access modifier (no modifier) for members, elements. | package-private implicitly (cannot declare) | public implicitly (redundant to declare)                                                                                                                                                         |
| Support for other modifiers                                  | public, protected, private                  | Concrete methods must be declared with one set of the following modifiers: <ul style="list-style-type: none"><li>• static</li><li>• private static</li><li>• private</li><li>• default</li></ul> |

# Differences Between Classes and Interfaces

---

| Feature                             | Class                                                                                               | Interface                                                                                               |
|-------------------------------------|-----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Abstract type support               | Declare class with abstract modifier. Explicitly Abstract.                                          | Always abstract. Implicitly abstract.                                                                   |
| Support for state                   | Instance and Class fields are supported and objects can be instantiated which have their own state. | No support for state except global constants (public static final), an interface is never instantiated. |
| Support for the 'default' modifier. | None                                                                                                | Permitted for public concrete methods only.                                                             |

# Creating and Implementing Interfaces

---

At a minimum an interface is declared as follows.

```
interface AnInterface {}
```

An interface can only be public or package-private.

# Creating and Implementing Interfaces

An interface can extend one or more other interfaces.

It does not implement interfaces or extend classes:

```
public interface TestInterface extends Comparable, AnInterface {}
```

# Creating and Implementing Interfaces

This video demonstrated that interfaces give you all the benefits of polymorphism for objects that implement them, but keep you free of having to add behavior to your business entities, that maybe does not quite fit your model, or are there to facilitate behavior that is not specific to your entity's type.

You can declare methods using interface types, you can cast to interface types, and declare variables with an interface type, including arrays.

# Extending Interfaces: Out of the Ordinary Concepts

---

In this video, we've covered:

- Compiler differences when casting to an interface vs casting to a super class.
- Inheriting duplicate methods from multiple interfaces.
- Extending interfaces.
- Using an abstract class implementing an interface.
- Ambiguous references and method clashes, which may occur because of 'multiple inheritance of type' which Java supports.

# Distinguish Class and Interface Inheritance

---

Because Java only supports a class inheriting from one parent class directly, the inheritance tree looking from the most specific class to the least specific is a straight line up the tree, a direct path to the root.

A base class can have many branches, but a branch has only one trunk from which it is derived.

Java enforces rules through each derivative, so there is no conflict or ambiguity with inherited methods and fields.

We've already seen that implementing interfaces can cause ambiguity and confusion if you are inheriting a method from your parent class that has the same signature but different return type from the interface you might want to implement, or if you are implementing multiple interfaces with clashing methods.

# Distinguish Class and Interface Inheritance

---

Both types of inheritance support polymorphism, allowing you to write code for the most generic case possible, but accepting any object typed to the class or interface as long as the agreed-on behavior exists, which it must.

You can cast to a class type or an interface.

We've seen that the compiler is not as stringent with its type checking when casting to interface types, because unlike that direct look up from the derived class to its outer most parent, inheriting from interfaces may create an overlapping web.

The compiler can make no assumptions about which classes might or might not be implementing a particular interface or be inheriting the interface behavior from a parent.

# Distinguish Class and Interface Inheritance

---

Interface inheritance alone will not get you very far.

Class inheritance can be used in an application without ever implementing interface inheritance, but your business entities may get cluttered with behavior that is not really specific to their identity;

Your application code may require more type checking and casting to force certain functions to occur without interfaces; or worse yet, your entity model may need to extend from base classes that mean nothing as an identifier but are there to facilitate a behavior system wide.

# Distinguish Class and Interface Inheritance

---

The following table describes some features and demonstrates where these two constructs overlap.

| Features/ Limitations                                                          | Abstract Class | Interface                                                                                                                                                  |
|--------------------------------------------------------------------------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| You cannot instantiate an instance of this type.                               | TRUE.          | TRUE.                                                                                                                                                      |
| They may contain a mix of methods declared with or without an implementation.  | TRUE.          | TRUE, but any concrete method must be private, default or static.                                                                                          |
| You can declare fields that are not public, static and final.                  | TRUE.          | FALSE ( all fields are automatically public, static, and final whether you declare them that way or not - any other declaration will be a compiler error). |
| You can define public, protected, package-private or private concrete methods. | TRUE.          | FALSE (only private concrete methods allowed). All other methods are public implicitly.                                                                    |

# Distinguish Class and Interface Inheritance

---

|                                                                        |       |                                                                                                                                                                |
|------------------------------------------------------------------------|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| You can define public, protected, or package-private abstract methods. | TRUE. | FALSE (all abstract methods are public implicitly).                                                                                                            |
| An abstract method must be declared abstract.                          | TRUE. | FALSE (it is implied if method has no body).                                                                                                                   |
| You must declare the type abstract if it contains an abstract method.  | TRUE. | FALSE (an interface is not declared abstract, though you can do it - it is just redundant since all methods are implicitly abstract unless they are concrete). |

# Distinguish Class and Interface Inheritance

---

When would you use one vs another?

Generally speaking you create an interface for a very limited set of system wide behavior without respect to any particular entity, so that regardless of what the entity is, it can execute behavior, either reusable or customizable, in a common way among all types.

You create an abstract class to fit your entity model for a specific application, abstracting a set of common behavior and attributes into some extensible reusable unit.

The abstract class makes sense logically as a way to describe your entity's identity ( to say your entity IS A (abstract something)), and it lends itself to reuse of both attributes and behavior.

# Distinguish Class and Interface Inheritance

---

Each derivative class is then responsible only for the additional functionality and features that are particular to its own specific type.

The derivative classes can focus on what makes them different if you've abstracted the common features and behavior into an inheritable unit.

# Distinguish Class and Interface Inheritance

---

|                                                                                                                                   | <b>Use Abstract Class</b> | <b>Use Interface</b> |
|-----------------------------------------------------------------------------------------------------------------------------------|---------------------------|----------------------|
| You want to abstract and conceptualize a business entity                                                                          | YES                       |                      |
| You want to define attributes and behavior at an abstract level for a set of classes that will inherit the behavior and the state | YES                       |                      |
| Your class requires access modifiers other than public (such as protected and private) for its members.                           | YES                       |                      |
| Your class requires non-static or non-final fields, to access and modify state                                                    | YES                       |                      |
| You are defining a behavior that disparate classes might share.                                                                   |                           | YES                  |
| You want to take advantage of multiple inheritance of type                                                                        |                           | YES                  |

# Code Examples Distinguish Class and Interface Inheritance

So what's the big deal? It's basically the exact same amount of code and functions the same, right? The difference is two-fold:

- First, we don't have some abstract class that has no relationship to our model as a base class. We are able to maintain the integrity of our business model.
- Second, if we need behavior that only the Dog and Tree require, we can create another interface with just that behavior described, and have only those classes implement that method, having zero impact on other areas of our application.
- Implementing an interface here instead of perhaps just delegating the counting method to another class, also allows you to describe your objects as Countable, and use any functionality available to Countable types.

# Declare and Use List and ArrayList

---

In Section 5, we talked about arrays and using some of the methods on List to manipulate arrays.

In this video, we are going to revisit the List interface, as well as talk about the `java.util.ArrayList`.

Now that we are freshly armed with the knowledge of the differences between interfaces and classes, let's look at the List and ArrayList types together.

# java.util.List

---

Declaration of `java.util.List`, an interface that extends the `Collection` interface.

```
public interface List<E> extends Collection<E> {
```

`java.util.List` is an ordered collection. The user of this interface has precise control over where in the list, each element is inserted.

The user can access elements by their integer index, and search for elements in the list.

Unlike sets, lists typically allow duplicate elements, and they typically allow multiple null elements, if they allow null elements at all.

[---

OCP JAVA SE 11 DEVELOPER CERTIFICATION PART 1 - 1Z0-815  
Declare and Use List and ArrayList Instances](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util>List.html</a></p></div><div data-bbox=)

# java.util.ArrayList

---

Declaration of a java.util.ArrayList, a class that extends AbstractList, and implements List among other interfaces.

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html>

java.util.ArrayList is a resizable-array implementation of the List Interface. Implements all optional list operations, and permits all elements, including null.

# java.util.ArrayList

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

In addition to implementing the List interface, this class provides methods to manipulate the size of the array, that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

It is important to note that ArrayLists do not support primitive data types.

You will have to use primitive data wrappers.

# Generics

---

Why use generics?

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Similar to parameters used in method declarations, these type parameters provide a way for you to re-use, the same code with different types of inputs.

Before Java 5, an ArrayList was a collection of any object you wanted to put in there.

If you were always putting a String element in the ArrayList, you would have no way of enforcing this rule, and you always had to cast to a String, when you retrieved an element from the ArrayList.

# Generics

---

Java 5 introduced a generic ArrayList which allows the person using an ArrayList in their code, to provide a type for the ArrayList.

You can create your own types that are parameterized: the ArrayList class was changed in Java 5 to be such a type, allowing the consumer of the type to decide what elements are permissible, when adding elements to the collection, prompting the compiler to do type-checking.

# Benefits of Declaring your Type

---

Generics and parameterized types and methods aren't part of the 1Z0-815 exam, they are part of the second one, 1Z0-816 which I'll be doing in a second course, but you should recognize the syntax and impact, of defining your type in a variable declaration.

Some of the benefits gained by declaring your type:

- Stronger type checks at compile time.
- Elimination of casts.
- Enabling programmers to implement generic algorithms.

# Declare and Use List and ArrayList Instances

---

We know that we cannot instantiate an interface, so we cannot create a List instance.

We can create an instance of a class that implements the List interface and assign it to the List type, which we've actually done in multiple examples in Section 5 videos.

# ArrayList Methods and Data Manipulation

---

This slide should look a little familiar. We saw a similar one in Section 5.

In this slide we've removed the array column and just show the ArrayList methods, most of which are on the List interface.

| Type of Functionality | ArrayList methods                                                                                     |
|-----------------------|-------------------------------------------------------------------------------------------------------|
| Comparison            | <code>equals</code><br><code>isEmpty</code>                                                           |
| Searches              | <code>contains</code><br><code>containsAll</code><br><code>indexOf</code><br><code>lastIndexOf</code> |

# ArrayList Methods and Data Manipulation

| Type of Functionality | ArrayList methods                                                                                                                                                                                                                                                                                                               |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data manipulation     | <code>add</code><br><code>addAll</code><br><code>clear</code><br><code>forEach</code><br><code>get</code><br><code>hashCode</code><br><code>remove</code><br><code>removeAll</code><br><code>removeIf</code><br><code>replaceAll</code><br><code>retainAll</code><br><code>set</code><br><code>size</code><br><code>sort</code> |

# ArrayList Methods and Data Manipulation

| Type of Functionality | ArrayList methods                                                                                                                                                                                    |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data transformation   | <code>copyOf (Java 10)</code><br><code>iterator</code><br><code>listIterator</code><br><code>of (Java 9)</code><br><code>spliterator (Java 8)</code><br><code>subList</code><br><code>toArray</code> |

# ArrayList Methods and Data Manipulation

---

In Section 5, we showed almost all of the methods of the List interface, but let's review the Data Manipulation methods here, because most of our previous review referenced methods we could use, with a List backed by an array, and removing and adding data was not supported, but this time we'll use an ArrayList instance.

# The Two New List Interface Methods

---

I want to review again two methods that were added to the List interface since Java 9:

- List.of (introduced in Java 9)
- List.copyOf (introduced in Java 10)

Both of these are static factory methods creating unmodifiable lists.

The object returned is an instantiation, of a member type defined in the  
ImmutableCollections class.

```
static final class ListN<E> extends AbstractImmutableList<E> implements Serializable {
```

# The Two New List Interface Methods

---

Some features of these List objects are:

- They are NOT modifiable. Elements cannot be added, removed, or replaced. If the elements themselves are mutable, this may cause the List's contents to appear to change.
- They disallow null elements. Attempts to create them with null elements result in NullPointerException.
- The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.

We reviewed code in Section 5 but let's just review one more time.

# ArrayList toArray Method

---

It is highly likely you will see this method in an exam question.

We looked at it previously in the arrays section, but let's put it through its paces here.

# 3 Scenarios when Calling ArrayList.toArray() Method

This table shows the 3 scenarios that occur when calling ArrayList.toArray() method:

- Calling the method with no parameter.
- Calling the method with a parameter that is an array larger than or equal to the list size.
- Calling the method with a parameter that is an array smaller than the list size.

| Parameter Input                                               | Result                                                                                                                                                                                                                                                                          | Notes                                                                                                                |
|---------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| no parameter                                                  | Array of Object returned with elements.                                                                                                                                                                                                                                         | Result is the same for a raw ArrayList and generic ArrayList: Object[] returned whose size is list.size().           |
| Array Length >= element size, passed array elements not null. | <p>Result = Parameter Array,</p> <p>Elements at index &lt; list.size() get populated with the list's elements.</p> <p>The element at index == list.size() is set to null.</p> <p>Elements at index &gt; list.size() remain the same as they were when passed to the method.</p> | If ArrayList is raw, you must cast to a typed array if assigning output to a variable.                               |
| Array Length < element size                                   | Array returned is the set of values in the list, and the type of the parameter passed.                                                                                                                                                                                          | ArrayStoreException occurs if the type of the array of the parameter isn't compatible with element type in the list. |

# ArrayList toArray Method

---

What did we learn?

- Both lists returned the array that was passed, but changed its elements (copying elements from the list to the array).
- Using the raw ArrayList and assigning the result of the method to var required us to cast it, that's when retrieving a single element from the returned array.

# ArrayList toArray Method

---

What did we learn?

- It doesn't matter what type of ArrayList you use when executing toArray with no parameter, the result will always be an Object[] array, and casting will be required when you access its individual elements.
- The elements are the elements copied from the list.

# ArrayList toArray and Other Methods

---

Putting it all together:

- The type of array returned is determined by the parameter passed, not by the type associated to the ArrayList itself.
- Typing the ArrayList does allow you to eliminate casting in most cases (exception is when you have no parameter, or your parameter array is a different type).

# List and ArrayList Exam Gotchas

---

This is a subtle difference which you might easily overlook on an exam question, as you get weary.

If you remember that **add**, is really adding more elements (either appending to the list or inserting additional elements at a specified index) and **set** is changing the object being referenced at a particular index, you'll know that setting an object at an index past the list's size will cause an error.

# List and ArrayList Exam Gotchas

---

This video reviewed things that were not out of the ordinary but things that are ordinary and easy to overlook:

- Be careful if you see methods that are similarly named to those on List, like setAll (not a method on list but a method on Arrays class).
- Or methods that have the wrong number of parameters such as indexOf, lastIndexOf methods with additional parameters (String usage not List usage).
- Be careful to remember that using set(), with an index out of the range, will cause a runtime exception.

# Lambda Expressions

---

To understand lambda expressions, you must first understand their genesis.

Before lambda expressions, anonymous inner classes were the feature that enabled you to make your code more concise.

They did and still do enable you to declare and instantiate a class at the same time, and use methods on that class, all within the same segment of code.

Anonymous classes are like local classes, except that they do not have a name.

You use them if you need to use a local class only once.

# Lambda Expressions

---

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of using an anonymous class may seem unwieldy and unclear.

Lambda expressions replace the bulkiness of anonymous inner classes.

# Functional Interfaces

---

From the oracle java specification, functional interfaces are described below:

Functional interfaces provide target types for lambda expressions and method references.

Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted.

Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context.

In other words, any where you can use an interface type in code, you can replace the instantiated object, which might have been used with a lambda expression.

# Lambda Expression Syntax

---

This information was on :

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax>

I like this image, because it very quickly tells you what you need to know about writing a lambda expression.

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

# Parameters (Argument List)

---

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

A lambda expression is composed of three parts.

- Parameters (Argument List):
  - A comma-separated list of formal parameters enclosed in parentheses.
    - You can omit the data type of the parameters in a lambda expression.
    - If you include data types, you must declare a data type for each parameter.

# Parameters (Argument List)

---

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

- In addition, you can omit the parentheses if there is only one parameter.
- If you have no parameters, you represent this with an empty parentheses set ().

Example:

p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25

# Arrow Token

---

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

- The arrow token ->

# Body

---

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

- Body:

- Can consist of a single expression.

If you specify a single expression, then the Java runtime evaluates the expression and then returns its value.

Example:

```
p -> p.getName() == "RALPH"
```

# Body

---

| Argument List  | Arrow Token | Body  |
|----------------|-------------|-------|
| (int x, int y) | ->          | x + y |

You do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

```
s -> System.out.println(s)
```

- Can consist of a statement block.

A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}).

Example

```
p -> {  
    return Calculate.add(1,2);  
}
```

# Lambda Expressions

---

\*\* Note that lambda expressions resemble method declarations.

In fact, you can consider lambda expressions to be anonymous methods, methods without a name, and no longer requiring an anonymous class to wrap them.

Sometimes a lambda expression does nothing but call an existing method.

In these cases, you can use a method reference - they are compact, easy-to-read lambda expressions for methods that already have a name.

# Lambda Expressions

---

Syntax for a Method reference:

```
System.out::println
```

A method reference can completely replace a lambda expression with an arrow token, if the method referred to meets the requirements, for the functional interface method.

# Correct Parameter Declarations in Lambda Expressions

Examples of syntactically correct parameter declarations in lambda expressions:

| Valid Lambda Expressions                          | Notes                                                       | Parentheses in the Parameter List Declaration |
|---------------------------------------------------|-------------------------------------------------------------|-----------------------------------------------|
| <code>() -&gt; System.out.println("Hello")</code> | No parameter syntax: () required.                           | Required                                      |
| <code>a -&gt; a + 1</code>                        | Single parameter syntax, no parentheses required.           | Optional                                      |
| <code>(a) -&gt; a + 1</code>                      | Single parameter syntax, parenthesis valid but optional.    | Optional                                      |
| <code>(int a) -&gt; a + 1</code>                  | Single parameter syntax, specifying data type.              | Required                                      |
| <code>(var a) -&gt; a + 1</code>                  | Single parameter syntax with Local Variable Type Inference. | Required                                      |

# Correct Parameter Declarations in Lambda Expressions

| Valid Lambda Expressions | Notes                                                                                                                    | Parentheses in the Parameter List Declaration |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------|
| (int a, int b) -> a + b  | Multiple parameter syntax with declared types. All parameters must specify a type.                                       | Required                                      |
| (a, b) -> a + b          | Multiple parameter syntax with no types. If you are not specifying data types for one, you must omit for all.            | Required                                      |
| (var a, var b) -> a + b  | Multiple parameter syntax with Local Variable Type Inference. You must use var for all parameters if you use it for one. | Required                                      |

# Incorrect Parameter Declarations in Lambda Expressions

Examples of syntactically incorrect parameter declarations in lambda expressions:

| Invalid Lambda Expressions     | Notes                                                                              | Parentheses |
|--------------------------------|------------------------------------------------------------------------------------|-------------|
| -> System.out.println("Hello") | No parameter syntax: () required.                                                  | Required    |
| int a -> a + 1                 | Single parameter syntax, specifying data type, parentheses required.               | Required    |
| var a -> a + 1                 | Single parameter syntax with Local Variable Type Inference, parentheses required.  | Required    |
| (int a, b) -> a + b            | Multiple parameter syntax with declared types. All parameters must specify a type. | Required    |

# Incorrect Parameter Declarations in Lambda Expressions

| Invalid Lambda Expressions | Notes                                                                                                                    | Parentheses |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------|-------------|
| a, b -> a + b              | Multiple parameter syntax with no types. You must use parentheses.                                                       | Required    |
| (a, var b) -> a + b        | Multiple parameter syntax with Local Variable Type Inference. You must use var for all parameters if you use it for one. | Required    |
| (int a, var b) -> a + b    | You cannot mix var with actual data types.                                                                               | Required    |

# Correct Body Declarations in Lambda Expressions

Examples of syntactically correct parameter declarations in lambda expressions:

| Valid Lambda Expressions                              | Notes                                                                                                                                                                                                       |
|-------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>()-&gt; System.out.println("Hello")</code>      | Single line with a statement (not an expression) is valid for methods which have void return type.                                                                                                          |
| <code>()-&gt; isTrue &amp;&amp; isPlausible</code>    | Single line lambda expression must be an expression which evaluates to the correct return type or covariant, if method has a return type.                                                                   |
| <code>(a) -&gt; {<br/>    return a + 2;<br/>};</code> | Using return requires the use of curly braces for the expression body.<br>Using curly braces requires a return statement if method has a return type.<br>All statements must be terminated with semi-colon. |

# Correct Body Declarations in Lambda Expressions

| Valid Lambda Expressions                                                                                              | Notes                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| (a) <pre>-&gt; {<br/>    if (a % 2 == 0) a = 100;<br/>    return a + 2;<br/>}</pre>                                   | Multiple statements permitted in curly braces. Every statement must be terminated with semi-colon. |
| (a) <pre>-&gt; {<br/>    int modValue = 2;<br/>    if (a % modValue == 0) a = 100;<br/>    return a + 2;<br/>};</pre> | You can declare local variables in the expressions.                                                |

# Incorrect Body Declarations in Lambda Expressions

Examples of syntactically incorrect body declarations in lambda expressions:

| Invalid Lambda Expressions                                                            | Notes                                                                |
|---------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <code>() -&gt; return 1</code>                                                        | Using return requires the use of curly braces.                       |
| <code>(a) -&gt; {<br/>    if (a % 2 == 0) a = 100,<br/>    return a + 2;<br/>}</code> | Trying to use a comma to separate statements. Must use a semi-colon. |

# Multiple Parameters for Lambda Expressions

---

I stated that lambda expressions require functional interfaces, to provide a targeted method and type.

Java provides several interfaces for you that satisfy common patterns, and several of these are in the `java.util.function` package.

# Generic and Primitive Interfaces

---

The first column represents the generic interface, the second column are the primitive typed interfaces. These interfaces cover a wide swath of functionality.

| Functional Interface<br>Generic | Functional Interface<br>Typed                                   | Description                                                                                                      |
|---------------------------------|-----------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| UnaryOperator<T>                | DoubleUnaryOperator<br>IntUnaryOperator<br>LongUnaryOperator    | Represents an operation on a single operand that produces a result of the same type as its operand.              |
| BinaryOperator<T>               | DoubleBinaryOperator<br>IntBinaryOperator<br>LongBinaryOperator | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| Consumer<T>                     | DoubleConsumer<br>IntConsumer<br>LongConsumer                   | Represents an operation that accepts a single input argument and returns no result.                              |

# Generic and Primitive Interfaces

---

| Functional Interface<br>Generic | Functional Interface<br>Typed                          | Description                                                            |
|---------------------------------|--------------------------------------------------------|------------------------------------------------------------------------|
| Predicate<T>                    | DoublePredicate<br>IntPredicate<br>LongPredicate       | Represents a predicate (boolean-valued function) of one argument.      |
| Supplier<T>                     | DoubleSupplier<br>IntSupplier<br>LongSupplier          | Represents a supplier of results.                                      |
| Function<T,R>                   | DoubleFunction<R><br>IntFunction<R><br>LongFunction<R> | Represents a function that accepts one argument and produces a result. |

These interfaces cover a wide swath of functionality.

# java.util.function Interfaces and Lambda Expressions

In this video, we want to talk about a few more of them, because once you understand them, your life will get easier, and you may see some of them in example code on the exam.

# Section Introduction

---

Bad stuff happens. You can plan for the bad stuff and be proactive about what to do when the bad stuff happens, and/or you can be reactive to the unexpected and unplanned.

In programming, as in life, usually you try to do both.

# Section Introduction

---

Java provides a framework for dealing with errors in your application, giving you as much control and customization as you desire.

You can use out-of-the-box exception classes as defined by `java.lang` package and other sdk libraries, and/or you can extend and customize one of Java's exception classes, and the information attached to it.

You can prevent an exception from interrupting the flow of your application, by handling it immediately, or just ignoring it, or you can interrupt the flow of the current method, it's calling method and so on, up the stack trace, depending on your own identification of the severity of the problem.

# Section Introduction

---

Java actually supports several varieties of exceptions, the Error - considered the most severe and usually a system error from which there is no recovery, and the Exception which is less severe.

In addition, Java categorizes errors as either a Checked exception or not.

Checked exceptions must be dealt with, using something called the Catch or Specify Requirement, or you will get a compiler error.

Unchecked exceptions (which can be either Error or Exception parent class) are not checked by the compiler for the same requirements.

# The Advantages of Exceptions

---

- Error-Handling Code can be separated from "Regular" Code.
- Errors can be propagated up the call stack.
- Error Types can be grouped and differentiated, like any other objects.

In the next set of videos, we'll be covering the following topics.

# Handling Exception

- Describe the advantages of Exception handling and differentiate among checked, unchecked exceptions, and Errors
- Create try-catch blocks and determine how exceptions alter program flow
- Create and invoke a method that throws an exception

HANDLING EXCEPTION  
JSE11DCP1-1-5

# Exception Handling Overview

---

An exception is an event that occurs during the execution of a program, that disrupts the normal flow of instructions.

It is usually caused by an abnormal condition or an unexpected use of a resource.

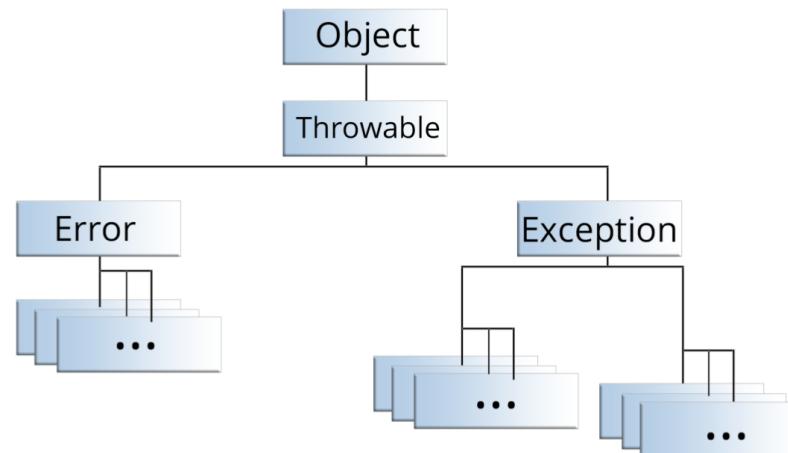
The `java.lang.Throwable` class and all its subclasses are, collectively, the exception classes.

# Exception Handling Overview

---

This diagram is from :

<https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html>



Throwable has two subclasses, Exception and Error.

# Error Class

---

Error Class - Java's naming standard is to use the 'Error' suffix for classes that extend Error.

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error.

The IOException is probably the best known Error in this hierarchy, but some others are:

- AnnotationFormatError
- AssertionError
- CoderMalfunctionError
- FactoryConfigurationError

# Error Class

---

- LinkageError which includes: BootstrapMethodError, ClassCircularityError, ClassFormatError, ExceptionInInitializerError, IncompatibleClassChangeError, NoClassDefFoundError, UnsatisfiedLinkError, and VerifyError
- ServiceConfigurationError
- ThreadDeath
- TransformerFactoryConfigurationError
- VirtualMachineError which includes: InternalError, OutOfMemoryError, StackOverflowError, and UnknownError

# Exception Class

---

Exception Class - Java's naming standard is to use the 'Exception' suffix for classes that extend Exception.

Objects that derive from the Exception class are generally the 'predictable' or known errors, they indicate that a problem occurred, but it is not a serious system problem, and recovery options may exist.

We'll discuss the Exception class at length.

# What Happens when an Exception Occurs

Now that we know what an exception is, we need to review what happens when one occurs.

When an error happens, the method, from which the error originated, creates a special object (an exception object) and hands it off to the runtime system.

The exception object contains information about the error, and the state of the program when the error occurred.

Creating an exception object and handing it to the runtime system, is called throwing an exception.

You can manually throw an exception by identifying a scenario you consider an error, but in many cases it is the Java Virtual Machine throwing exceptions automatically.

# What Happens when an Exception Occurs

Throwing an exception is only half the story. The runtime system then attempts to find something to 'handle' the exception.

Handling the exception means there is code that identifies itself as responsible, for responding to the error condition that occurred. And like life, there aren't always volunteers or assignees, however.

The runtime system trying to find a handler for an exception object could be likened to a frustrated sales customer, working his or her way up through management, as they query whether this particular manager can handle their complaint.

The manager will refer the customer to the next manager, above themselves or, hopefully, at some point, handle the complaint themselves, and let the customer progress to the next step of a sale or refund.

# What Happens when an Exception Occurs

A method can identify itself as a responsible party (able to handle the exception) by declaring a block of code called an exception handler.

The runtime system searches the call stack, for a method that contains such a block of code, starting with the current method, and searching the call stack in reverse order, until it finds a method with a handler.

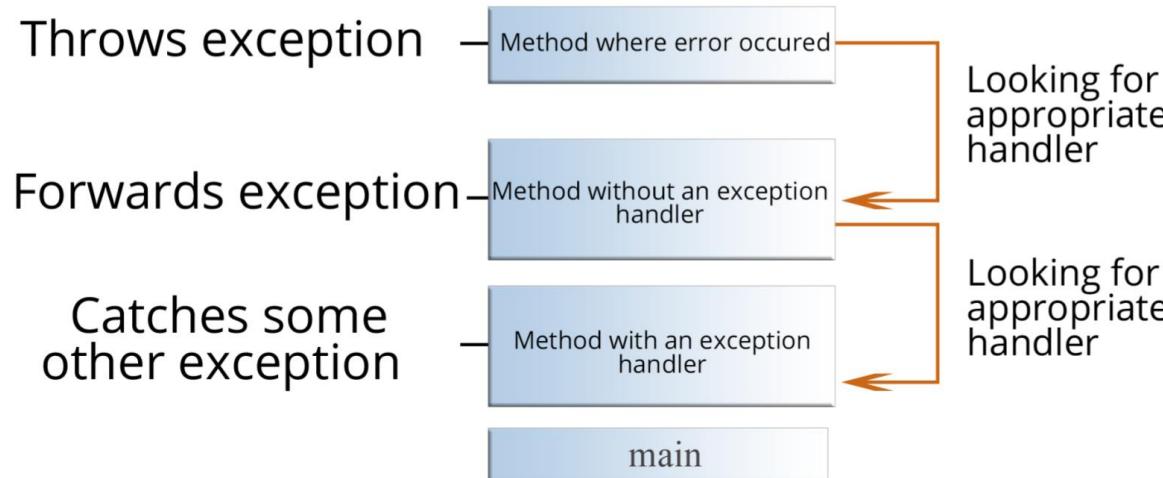
Remember that a call stack is a listing of program counter addresses (PCs) representing instructions (method calls) from within the program which identifies the path, the application executed to get to the current instruction or statement.

# What Happens when an Exception Occurs

The runtime system looks for a handler, that must 'catch' the right type of exception that was thrown, or the runtime system will continue to look for a matching responsible party.

If no handler is ever found, the thread that created the error or exception dies.

# What Happens when an Exception Occurs



The following diagram is from: <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

# Checked and Unchecked Exceptions

---

This was a generic example that did not discriminate between Error or Exception, or between the two significant categories of Exception, Checked and Not Checked.

The following table explains what types of errors are Checked and Unchecked Exceptions:

| Class Type                                                                                                               | Checked or Unchecked | Must satisfy<br>The Catch or Specify Requirement |
|--------------------------------------------------------------------------------------------------------------------------|----------------------|--------------------------------------------------|
| Throwable (or any custom exception which extends Throwable).                                                             | Checked Exception    | YES                                              |
| Error (or any custom exception which extends Error).                                                                     | Unchecked Exception  | NO                                               |
| Exception - Any Exception subclassed directly or indirectly from RuntimeException - these are called runtime exceptions. | Unchecked Exception  | NO                                               |
| Exception - Any Exception not subclassed from RuntimeException or its subclasses.                                        | Checked Exception    | YES                                              |

# Checked and Unchecked Exceptions

---

A checked exception is a special designation, for a group of exceptions that the compiler forces compliance of 'The Catch or Specify' Requirement.

An unchecked exception will not be checked by the compiler.

# Runtime Exceptions

---

Before I explain 'The Catch or Specify' requirement, let's look at some runtime exceptions (there are many more and reviewing them prior to the exam might be helpful), but in this table I list some of the most common ones, you are probably already familiar with:

| Exceptions                     | Details                                                                    |
|--------------------------------|----------------------------------------------------------------------------|
| ArithmaticException            | For example, an integer "divide by zero" throws an instance of this class. |
| ArrayIndexOutOfBoundsException | Thrown to indicate that an array has been accessed with an illegal index.  |
| ArrayStoreException            | Attempt made to store the wrong type of object into an array of objects.   |
| ClassCastException             | Attempted to cast an object to a subclass of which it is not an instance.  |

# Runtime Exceptions

---

| Exceptions                 | Details                                                                                                           |
|----------------------------|-------------------------------------------------------------------------------------------------------------------|
| IndexOutOfBoundsException  | Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. |
| NegativeArraySizeException | Thrown if an application tries to create an array with negative size.                                             |
| NullPointerException       | Thrown when an application attempts to use null in a case where an object is required.                            |
| SecurityException          | Thrown by the security manager to indicate a security violation.                                                  |

Because these are runtime exceptions, the compiler does not force you to write code to check for the exception (try/catch block) or specify the exception, in the throws section of the wrapping method declaration.

# Exception Handling in Code

---

Only a checked exception requires compliance, to 'The Catch or Specify' Requirement.

I repeat that a checked exception is neither an Error, nor a runtime exception.

Stated another way. Errors and runtime exceptions are unchecked exceptions, everything else is a checked exception.

# Exception Handling in Code

---

The Catch or Specify Requirement states that code that might throw certain exceptions, must be enclosed by either of the following:

- A try statement that catches the exception. The try must provide an appropriate handler for the exception.
- A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception or an appropriate exception type.

Why does Java have these two different types of exceptions, and if you are creating your own exceptions, which should you use?

# Exception Handling in Code

---

Java designers argue that an exception thrown by a method, is part of the method's public programming interface, as much as its parameters and return values.

If a client can be expected to recover from the exception, you inform the client by making it a checked exception, and identifying the exception in the throws clause of the method.

If the client cannot be expected to recover from the exception, you do not need to make it a checked exception.

# Try-catch Blocks and Program Flow

---

At a minimum, the try catch block looks like the following, which catches nothing and does nothing.

```
try {  
}  
} catch (Throwable t) {  
}  
}
```

Unlike other statements, such as if and else, the enclosing braces {} sets are required.

# Try-catch Blocks and Program Flow

---

Alternately you can replace the catch clause, with a finally clause.

Just declaring a try clause without either a catch, or finally clause, results in a compiler error.

```
try {  
}  
} finally {  
}  
}
```

# Try-catch Blocks and Program Flow

---

Any exception that occurs in a statement wrapped by the try block, will immediately stop the instruction set, at the statement that triggered the error, and fall through to the catch statement.

In other words, all statements after the offending statement, will be skipped.

If an error occurs in a loop or nested loop, and the try block is outside of the loop, the code breaks out of any participating loops.

# Try-catch Blocks and Program Flow

---

You could wrap every statement in a try/catch if you never wanted to miss a statement, but in general, an error is a significant event, and you may not want to proceed with succeeding code.

You can design placement of your try/catch blocks strategically, to allow flow in some instances, and break flow in others.

# Create and Invoke Methods that Throw Exceptions

---

In this video, we discussed the four types of errors that can be thrown (Throwable, Error, Exception (not RuntimeException) and RuntimeException).

I demonstrated

- Customizing each of the types.
- Throwing each of the types from methods.
- Identifying which of the four are considered checked, and the additional requirements to satisfy the compiler when they are checked.

# Create and Invoke Methods that Throw Exceptions

---

- Adding one or multiple exceptions in the throw clause, of a method declaration.
- Testing multiple clauses associated to a single try clause, noting that order matters, and you should declare exceptions in the catch clauses, in order of most specific to the least specific.

# Exceptions Finally Clause and Summary

---

I think we've exhausted varieties with the catch clause. Let's review some of the catch clause constraints:

- A catch clause is not required for unchecked exceptions.
- A catch clause is required for checked exceptions, which are thrown from the code in the try block.
- Adding a catch clause for a checked exception, that is never thrown from the code in the try block, is a compiler error.
- You can have multiple catch clauses, each catch clause must specify a unique exception.

# Exceptions Finally Clause and Summary

---

- Multiple catch clauses should be declared from most specific to least specific. It is a compiler error to define a more specific exception, after a lesser specific exception in the same inheritance tree.
- A clause that contains multiple exceptions in one clause, cannot declare two that are in the same inheritance tree, in the same declaration.
- An exception parameter is not final in a single exception clause.
- An exception parameter IS implicitly final, if it is in a multiple exception clause.

# Exceptions in Static and Instance Initializers

---

A finally clause is always executed, but if a new exception is not created in the finally clause, the Java Virtual Machine still looks for the calling method, to handle the exception, which does not happen here.

This is true of any finally clause, not just one in a static initializer method.

The Java Virtual Machine won't initialize the class, with the exception thrown, even though we have a finally clause.

Can we catch and fix the exception?

# Exceptions in Static and Instance Initializers

---

What have we learned about exceptions in initializer statements?

- An unhandled exception in a static initializer is fatal to the application, checked or not checked, if the class is ever initialized.
- A checked exception in a static initializer, is a compiler error that can only be caught and handled, by the static initializer code itself.
- An unhandled exception in an instance initializer, is not fatal to the application.

# Exceptions in Static and Instance Initializers

---

- A checked exception in an instance initializer, that is not caught and handled by the instance initializer code itself is allowable, if and only if all constructors declare the exception in their throws clauses.
- A no args constructor with a throws declaration, that includes a checked exception, will cause trouble for any class that extends it, which does not also declare a no args constructor, that either throws the same exception or a broader one.

# Section Introduction

---

A module is a named set of packages, resources, and native libraries.

Java 9 introduced the concept of a module, a new java entity that is similar to an executable jar in its production state, but that includes a name and a descriptor class that provides benefits that did not exist prior to Java 9.

The benefits of a module are:

- Stronger encapsulation at a higher level which improves security.

Prior to Java 9 any class in any package with a public or protected access modifier that was included in a jar on the classpath was by its nature publicly accessible or extensible by all of the application code. There was no way to hide this information.

# Section Introduction

---

In JDK 9, you have three ways to expose your public types - you can make them:

1. Public to everyone (unqualified exposure).
2. Public to just specific modules deemed 'friends' (qualified exposure).
3. Not public to any other module, but only to the packages included in the module itself.

A module descriptor will specifically declare which packages are visible to other modules executing in the same application space.

# Section Introduction

---

- Reliable dependencies which improves stability

A module descriptor requires a module to specifically declare which modules it is dependent upon, publishing its dependencies to consumers.

# Understanding Modules

- Describe the Modular JDK
- Declare modules and enable access between modules
- Describe how a modular project is compiled and run

UNDERSTANDING MODULES  
JSE11DCP1-1-4

# Modular JDK

---

Before Java 9, the Java Development Kit was a monolithic set of packages, packaged in one jar file, the rt.jar, which any java application would need to deploy along with its own code.

The Java platform had grown over time to include about 25 different frameworks.

Even if your application only ever used one class in a package, you would need the rt.jar, which included all of these frameworks.

Project Jigsaw was the name of the project created to modularize the JDK.

# Goals of Modularizing the Java Development Kit

---

The goals of modularizing the Java Development Kit were summarized by the Project Jigsaw Team, and restated here:

- More scalability, by allowing the Java Development Kit to more easily scale down to smaller platforms.
- Improved security by protecting the internals of the Java Development Kit.
- Improved maintainability by publishing dependencies and reducing the amount of interdependencies.
- Improved application performance by allowing configuration flexibility.
- Easier construction of and maintenance of libraries and large applications.

# Command Line Tool Options to Examine Modules

The Java designers spent a lot of time deconstructing the monolith, decoupling the frameworks, and reconstructing the code into smaller sets of code (a few dozen modules) that met the objectives above.

The smaller sets of code, nicely packaged into modular jar files, comprise the Modular JDK.

You can review the Modular Java Development Kit Specification at

<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

# Modular Java Development Kit – Java Standard Edition

The Modular Java Development Kit is split into two sections:

- The Java SE - The Java Platform, Standard Edition (Java SE) APIs.
  - These APIs define the core Java platform for general-purpose computing.
  - These APIs are in modules whose names start with a 'java.' prefix.

# Modular Java Development Kit – Java Standard Edition

- All of the modules in the Java SE are standard modules as defined by the Java Community Process (JCP), however not all standard modules defined by the Java Community Process are in the Java SE, just the foundational standard modules.
- A standard module, may contain both standard and non-standard API packages.
- A standard package is prefixed with 'java.' or 'javax.', such as java.lang and javax.net for example.

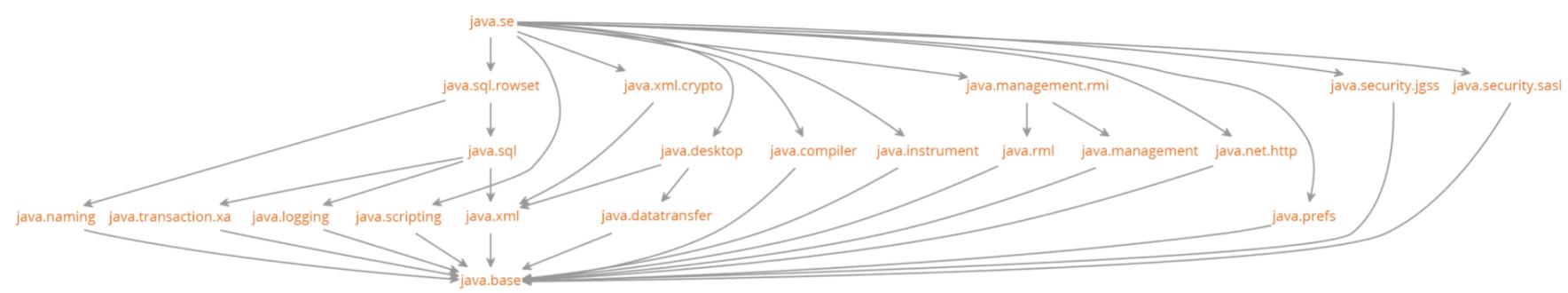
# Modular Java Development Kit – Java Development Kit

- The JDK - The Java Development Kit (JDK) APIs are specific to the JDK.
  - These APIs may not necessarily be available, in all implementations of the Java SE Platform.
  - These APIs are in modules whose names start with a 'jdk.' prefix.
  - These are non-standard modules.
  - A non-standard module must not export any standard API packages.
  - A non-standard package is generally, prefixed with 'jdk.' such as `jdk.internal.logger` for example.

# Modular SE Modules Graph

The following diagram is the module graph, of the Modular SE modules (which is part of the Modular Java Development Kit).

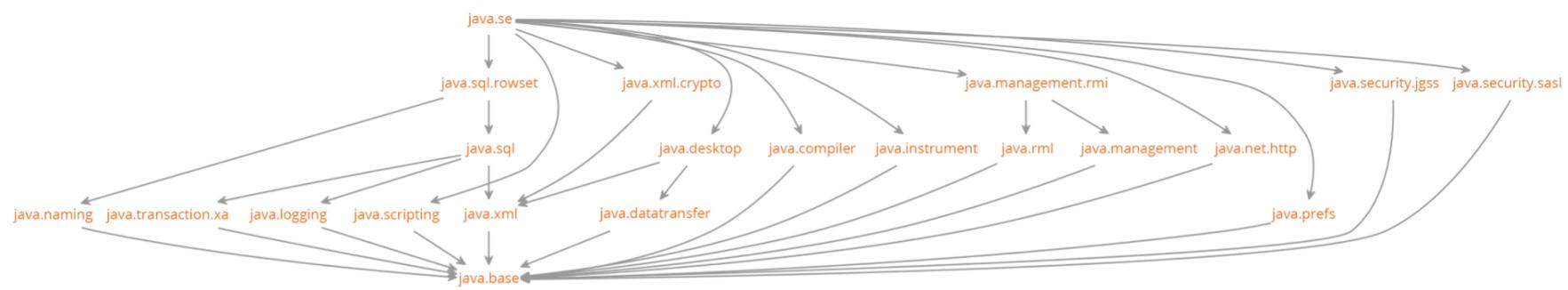
A module graph is a diagram, which attempts to visualize the module dependencies of a particular module.



# Modular SE Modules Graph

In the diagram below, the `java.se` module, has dependencies on all of the modules shown, whereas `java.base` has no dependencies, but is a required dependency for all the modules shown.

The arrows are called read edges - read because a module is said to be able to 'read' another, if properly configured as dependent on it.

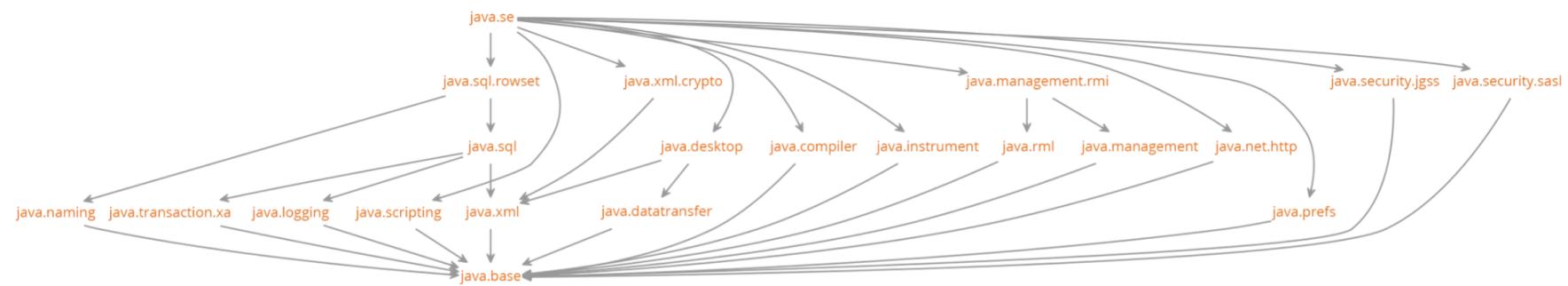


# Modular SE Modules Graph

For example, the java.xml module is said to read the java.base module.

Note that a module graph only displays module dependencies, and will not include packages.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.se/module-summary.html>



# Modular SE Modules Graph

---

Each module shown is part of the Java API specification.

For example, the link:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html>

Will give you all the information about the java.base module and the packages and resources it contains.

The java.base module is described as the foundational APIs of the Java SE Platform (which include the core packages such as java.lang, java.util, java.io, etc).

# Modules Summary

---

In this video we've :

- Discussed the goals of the Modular JDK (as stated by Project Jigsaw).
- Introduced you to standard and non-standard modules.
- Introduced you to a module graph and showed you the JAVA SE module graph.
- Introduced you to some of the options that Java and jdeps tools support for examining modules.

# Command Line Tool Options to Examine Modules

Getting familiar with these will hopefully cover any exam question you see on tool support for modules.

| Executable | Java options        | Example(s)                                                          | Explanation                                                                                                                                                                                                                                                                |
|------------|---------------------|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| java       | --list-modules      | java --list-modules<br>java -p MyFirstModule.jar --list-modules     | List visible modules. If you do not specify a module path, you'll get a listing of the modular Java Development Kit.<br><br>If you specify directories in the module path, you'll include visible modules in those directories (either exploded directories or jar files). |
| java       | -p<br>--module-path | java --module-path . -m MyFirstModule<br>java -p . -m MyFirstModule | You need to specify the module path when executing a module jar.                                                                                                                                                                                                           |
| java       | -m<br>--module      | java -p . -m MyFirstModule<br>java -p . --module MyFirstModule      | You need to specify the module name when executing a module jar.                                                                                                                                                                                                           |

# Command Line Tool Options to Examine Modules

| Executable | Java options            | Example(s)                                                                         | Explanation                                                                                                                                          |
|------------|-------------------------|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| java       | --dry-run               | java --dry-run -p . --module MyFirstModule                                         | The --dry-run option loads the main class but does not run it, used for testing module dependencies are satisfied. A successful result is no output. |
| java       | -d<br>--describe-module | java --module-path . --describe-module MyFirstModule<br>java -p . -d MyFirstModule | The --describe-module option gives you summary information about a module including its path, its dependencies, and its packages and resources.      |

# Command Line Tool Options to Examine Modules

| Executable | Java options            | Example(s)                                                                                                                                   | Explanation                                                                                                       |
|------------|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| javac      | --module<br>-m          | javac -d out\production --module-source-path . --module MyFirstModule<br><br>javac -d out\production --module-source-path . -m MyFirstModule | When you compile using --module, you must also specify --module-source-path.                                      |
| javac      | -p<br>--module-path     | javac -d out\production -p . --module-source-path . -m MySecondModule                                                                        | You use --module-path with javac when your code has a dependency on existing modules.                             |
| jar        | -d<br>--describe-module | jar --file MyFirstModule.jar --describe-module<br><br>jar -f MyFirstModule.jar -d                                                            | The --describe-module option for the jar command is similar to the java one, but includes the main-class as well. |

# Command Line Tool Options to Examine Modules

| Executable | Java options        | Example(s)                                                                                                                                                                     | Explanation                                                                                                                                                                                                                         |
|------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| jdeps      |                     | jdeps MyFirstModule.jar                                                                                                                                                        | jdeps with no options will give you summary information about the jar file, including module dependencies if this is a modular jar.                                                                                                 |
| jdeps      | --list-deps         | jdeps --module-path ;out --list-deps MyFirstModule.jar<br>jdeps --module-path ;out --list-deps --module MyFirstModule<br>jdeps --module-path ;out --list-deps -m MyFirstModule | --list-deps option will list module dependencies. You can use either a specified jar or a module with the --module (or -m) option.<br><br>It is important to note that for jdeps tool -p is NOT interchangeable with --module-path. |
| jdeps      | --list-reduced-deps | jdeps --module-path ;out --list-reduced-deps -m MyFirstModule<br>jdeps --list-reduced-deps MyFirstModule.jar<br>jdeps --list-reduced-deps --module java.sql                    | Same as --list-deps but does not include implied reads.                                                                                                                                                                             |

# Command Line Tool Options to Examine Modules

| Executable | Java options        | Example(s)                                                                                                 | Explanation                                                                                                                                                               |
|------------|---------------------|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| jdeps      | --print-module-deps | jdeps --print-module-deps --module java.sql<br>jdeps --module-path . --print-module-deps MyFirstModule.jar | Same as --list-reduced-deps but prints dependencies in a comma delimited list.                                                                                            |
| jdeps      | --check             | jdeps --module-path . --check MyFirstModule<br>jdeps --check java.sql                                      | Prints the module descriptor, the resulting module dependencies after analysis and the graph after transition reduction. It also identifies any unused qualified exports. |

# Command Line Tool Options to Examine Modules

In this video:

- I've shown you how to build and execute a module in IntelliJ.
- How to jar the module.
- Executing both jarred module and the exploded directory from the command line.
- I've shown you multiple examples of using the tools to get information about modules.
- I've given you a brief sample of how to create modules that work together.

# Enabling Access Between Modules

---

At a minimum a module directive file (module-info.java) for a module named 'NamedModule' has the form:

```
module NamedModule { }
```

The official specification for the module declaration is shown below:

ModuleDeclaration:

```
{Annotation} [open] module Identifier { . Identifier} { ModuleDirective} }
```

# Module Directives

---

From the java specification, the following directives should populate {ModuleDirective} are as follows.

Note: There are three others (opens, uses and provides) which are not part of the first exam objective, and we won't be discussing here.

| Module Directive | Type | Directive arguments           | Examples                                                                                                                | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------|------|-------------------------------|-------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| requires         |      | {RequiresModifier} ModuleName | requires org.module.a;<br>requires java.logging;<br>requires transitive<br>org.module.d<br>requires static org.module.e | A module specified in the requires directive is one which exports package(s) that the current module may or does have a dependency on. The current module is said to 'read' the module specified in the required directive.<br><br>A requires directive with the transitive modifier allows any module which requires the current module to have an implicit 'requires directive' on the specified module.<br><br>A static directive requires the specified module at compile time but it's optional at run time. |

# Module Directives

---

| Module Directive | Type        | Directive arguments                        | Examples                                           | Description                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------|-------------|--------------------------------------------|----------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| exports          | unqualified | PackageName                                | exports org.pkg.base;                              | A package specified in the exports directive will expose its public and protected types (and the public and protected members of those types) to modules that 'read' the current module (in other words - specifies a requires directive to the current module).                                                                                                                                                       |
| exports          | qualified   | PackageName [to ModuleName {, ModuleName}] | exports org.pkg.util to org.module.a, org.module.b | The 'to' key word makes an exports directive qualified and will be followed by a comma delimited list of modules that are termed 'friends' of the current module. Friends of a module have access to public and protected types (and the public and protected members of those types) of the exported package. No other modules will have access. You limit the exposure of the exported package types to its friends. |

# Modules Summary

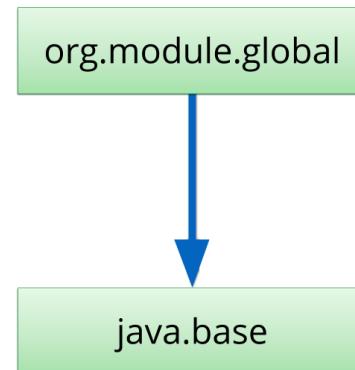
---

In the last 3 videos, I've demonstrated how to create and declare modules using IntelliJ's features, as well as editing the module-info.java files manually, and we've used a combination of some of the directives available.

# Module Graphs

---

You can interpret a module graph by the way the arrow (called the read edge) is directing you, in this case, the graph says pictorially "org.module.global reads java.base"



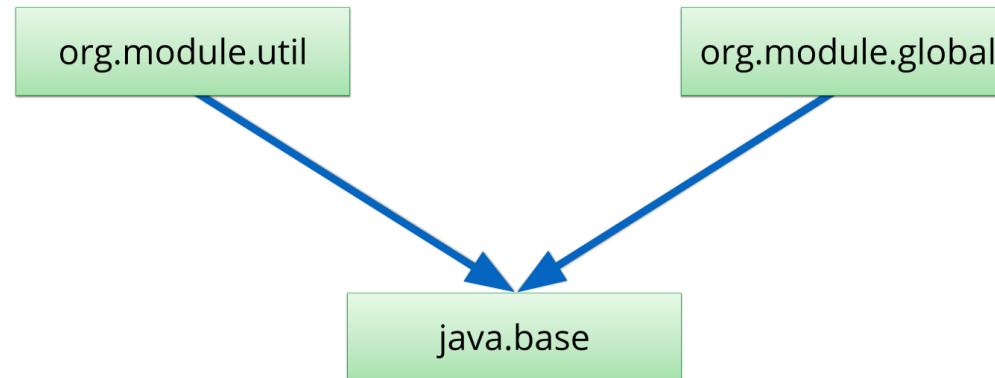
# Module Graphs

---

org.module.util reads java.base, and org.module.global also reads java.base.

You'll note that org.module.util does not read org.module.global.

The read edges correspond to the number of elements listed in the jdeps -s listing.

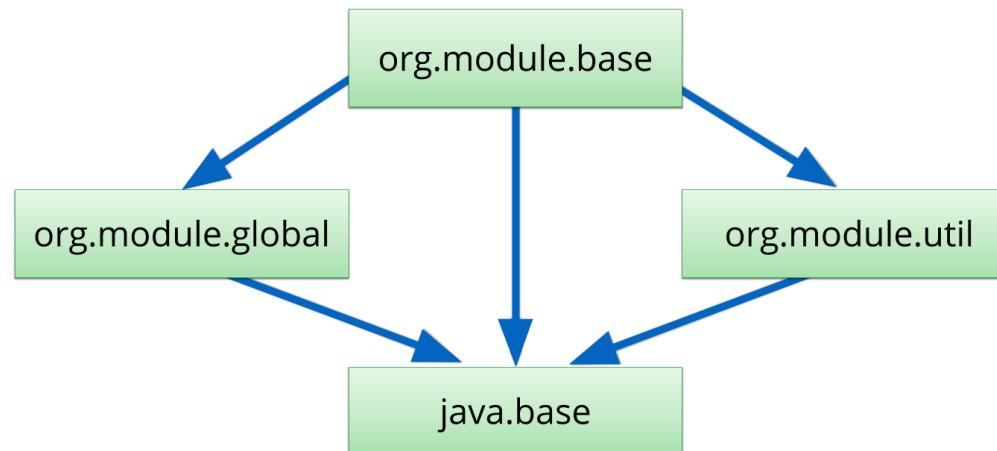


# Module Graphs

---

Drawing it looks like the following. You'll note that we have 5 arrows which represent our read edges.

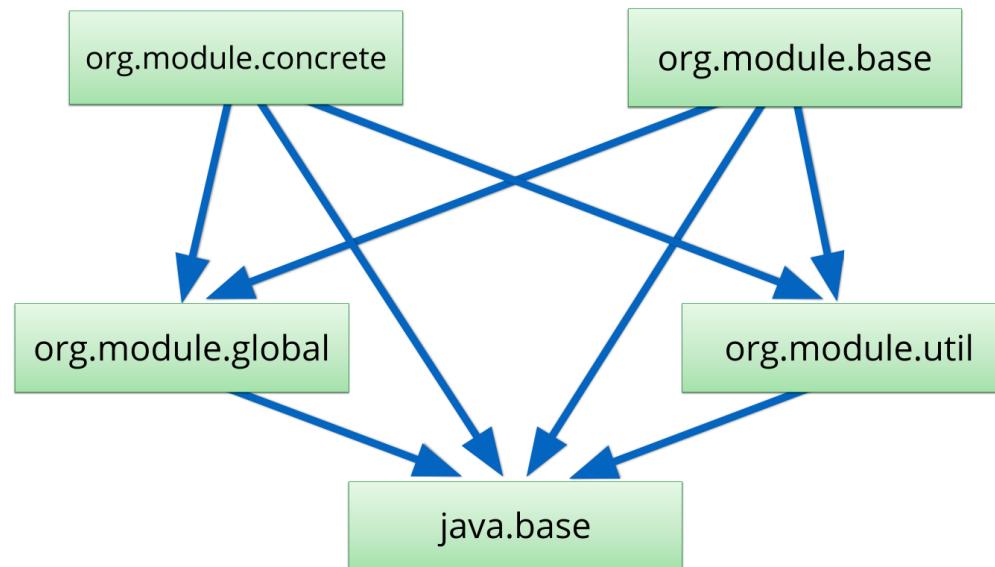
org.module.base has a dependency on org.module.global, although it is not specified in the directive file - this was created by the requires transient directive in org.module.util.



# Module Graphs

---

So our full module graph is shown below:



# Module Graphs

---

The exam may have a question about the module graph, but will probably be restricted to two or three modules.

- Read the descriptors carefully, looking for qualified exports and transitive dependencies - these actually reduce the dependencies.
- The `java.base` module is an implicit dependency for every module - if you see a module graph with any module that does not read `java.base`, it is not an accurate graph.
- If you see a module graph with a node that is a package, it is not a valid module graph.