OCP – Preparation

Inhoud

[Chapter 1 (Welcome to Java) 5](#_Toc116886215)

[Major Components 5](#_Toc116886216)

[Identifying Benefits of Java 5](#_Toc116886217)

[Understanding the Java Class Structure 6](#_Toc116886218)

[Understanding package declarations and imports 7](#_Toc116886219)

[Ordering Elements in a Class 9](#_Toc116886220)

[Code Formatting on the exam 9](#_Toc116886221)

[Exam Essentials 11](#_Toc116886222)

[Chapter 2 (Java Building Blocks) 12](#_Toc116886223)

[Creating Objects 12](#_Toc116886224)

[Understanding Data Types 12](#_Toc116886225)

[Declaring Variables 14](#_Toc116886226)

[Managing variable scope 15](#_Toc116886227)

[Destroying Objects 15](#_Toc116886228)

[Exam Essentials 16](#_Toc116886229)

[Chapter 3 (Operators) 17](#_Toc116886230)

[Understanding Java Operators 17](#_Toc116886231)

[Applying Unary Operators 17](#_Toc116886232)

[Working with Binary Arithmetic Operators 18](#_Toc116886233)

[Assignment Values 18](#_Toc116886234)

[Comparing Values 19](#_Toc116886235)

[Making Decisions with the Ternary Operator 21](#_Toc116886236)

[Exam Essentials 21](#_Toc116886237)

[Chapter 4 (Making Decisions) 22](#_Toc116886238)

[Creating Decision-Making Statements 22](#_Toc116886239)

[Writing while Loops 23](#_Toc116886240)

[Constructing for loops 24](#_Toc116886241)

[Controlling flow with branching 26](#_Toc116886242)

[Exam Essentials 27](#_Toc116886243)

[Chapter 5 (Core Java APIs) 28](#_Toc116886244)

[Creating and manipulating Strings 28](#_Toc116886245)

[Using the StringBuilder Class 30](#_Toc116886246)

[Understanding Equality 31](#_Toc116886247)

[Understanding Java Arrays 31](#_Toc116886248)

[Understanding an ArrayList 35](#_Toc116886249)

[Creating Sets and Maps 39](#_Toc116886250)

[Calculating with Math APIs 41](#_Toc116886251)

[Exam Essentials 42](#_Toc116886252)

[Chapter 6 (Lambdas and Functional Interfaces) 43](#_Toc116886253)

[Writing Simple Lambdas 43](#_Toc116886254)

[Introducing Functional Interfaces 44](#_Toc116886255)

[Working with variables in Lambdas 44](#_Toc116886256)

[Calling APIs with Lambdas 45](#_Toc116886257)

[Summary 46](#_Toc116886258)

[Exam Essentials 46](#_Toc116886259)

[Chapter 7 (Methods and Encapsulation) 47](#_Toc116886260)

[Designing methods 47](#_Toc116886261)

[Working with Varargs 48](#_Toc116886262)

[Applying Access Modifiers 49](#_Toc116886263)

[Applying the *static* Keyword 49](#_Toc116886264)

[Passing Data among Methods 51](#_Toc116886265)

[Overloading Methods 52](#_Toc116886266)

[Overloading Complex scenarios 52](#_Toc116886267)

[Encapsulating Data 53](#_Toc116886268)

[Exam Essentials 54](#_Toc116886269)

[Chapter 8 (Class design) 54](#_Toc116886270)

[Understanding Inheritance 54](#_Toc116886271)

[Creating Classes 55](#_Toc116886272)

[Declaring Constructors 56](#_Toc116886273)

[Inheriting Members 60](#_Toc116886274)

[Understanding Polymorphism 64](#_Toc116886275)

[Exam Essentials 68](#_Toc116886276)

[Chapter 9 (Advanced Class Design) 69](#_Toc116886277)

[Creating Abstract Classes 69](#_Toc116886278)

[Implementing Interfaces 71](#_Toc116886279)

[Polymorphism and Interfaces 73](#_Toc116886280)

[Reviewing Interface Rules 74](#_Toc116886281)

[Introducing Inner Classes 75](#_Toc116886282)

[Exam Essentials 76](#_Toc116886283)

[Chapter 10 (Exceptions) 77](#_Toc116886284)

[Understanding Exceptions 77](#_Toc116886285)

[Recognizing Exception Classes 78](#_Toc116886286)

[Handling Exceptions 79](#_Toc116886287)

[Chapter 11 (Modules) 84](#_Toc116886288)

[Introducing Modules 84](#_Toc116886289)

[Creating and Running a Modular Program 84](#_Toc116886290)

[Updating Our Example for Multiple Modules 86](#_Toc116886291)

[Diving into the module-info File 88](#_Toc116886292)

[Discovering Modules 89](#_Toc116886293)

[Reviewing Command-Line Options 91](#_Toc116886294)

[Options you need to know for the exam 92](#_Toc116886295)

[Exam Essentials 93](#_Toc116886296)

[Chapter 12 (Java Fundamentals) 93](#_Toc116886297)

[Applying the final Modifier 93](#_Toc116886298)

[Working with Enums 95](#_Toc116886299)

[Creating Nested Classes 97](#_Toc116886300)

[Declaring an inner Class 98](#_Toc116886301)

[Creating a static Nested Class 99](#_Toc116886302)

[Writing a Local Class 100](#_Toc116886303)

[Defining an Anonymous Class 100](#_Toc116886304)

[Understanding Interface Members 102](#_Toc116886305)

[Introducing Functional Programming 105](#_Toc116886306)

[Exam Essentials 108](#_Toc116886307)

[Chapter 13 (Annotations) 109](#_Toc116886308)

[Introducing Annotations 109](#_Toc116886309)

[Creating Custom Annotations 109](#_Toc116886310)

[Applying Annotations 109](#_Toc116886311)

[Declaring Annotation-Specific Annotations 110](#_Toc116886312)

[Using Common Annotations 111](#_Toc116886313)

[Chapter 14 (Generics and Collections) 112](#_Toc116886314)

[Chapter 15 (Functional Programming) 112](#_Toc116886315)

[Chapter 16 (Exceptions, Assertions and Localizations) 112](#_Toc116886316)

[Chapter 17 (Modular Applications) 112](#_Toc116886317)

[Chapter 18 (Concurrency) 112](#_Toc116886318)

[Chapter 19 (I/O) 112](#_Toc116886319)

[Chapter 20 (NIO.2) 112](#_Toc116886320)

[Chapter 21 (JDBC) 112](#_Toc116886321)

[Chapter 22 (Security) 112](#_Toc116886322)

# Chapter 1 (Welcome to Java)

## Major Components

* JDK – (Java Development Kit) Contains the minimum software you need to do Java Development.
* Compiler – (javac) Converts .java files to .class files, and the launcher java, which creates the virtual machine and executes the program.
* JDK also contains other tools including the archiver (jar) command, which can package files together, and the API documentation (Javadoc) command for generating documentation.
* Javac program generates instructions in a special format that the java command can run called bytecode, then java launches the Java Virtual Machine (JVM) before running the code.
* The JVM knows how to run your .class files and knows how to run bytecode.
* Java comes with a large suite of **application programming interfaces (APIs)** that you can use from the start, such as a StringBuilder class to create large String and a method in Collections

## Identifying Benefits of Java

**Object Oriented –** Java is an object-oriented language. Means all code is defined in classes, and most of those classes can be instantiated into objects.

**Encapsulation –** Java supports access modifiers to protect data from unintended access and modification.

**Platform Independent –** Java is an interpreted language that gets compiled to bytecode. A key benefit is that Java code gets compiled once rather than needing to be recompiled for different operating systems. This is known as “write once, run everywhere.”

**Robust –** Java prevents memory leaks. Java manages memory on its own and does garbage collection automatically.

**Simple –** Java was intended to be simpler to understand than C++ removal of pointers and operator overloading.

**Secure –** Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on.

**Multithreaded –** Java is designed to allow multiple pieces of code to run at the same time. There are also many APIs to facilitate this task.

**Backward Compatibility –** The Java Language architects pay careful attention to making sure old programs will work with later versions of Java. By using the Deprecation technique, they accomplish this where code is flagged to indicate it shouldn’t be used.

## Understanding the Java Class Structure

**Object** – is a runtime instance of a class in memory, also referred to as an instance since it represents a single representation of the class.

**State of the program** – all the various objects of all the different classes.

**Reference** – is a variable that points to an object.

**Fields and Methods**

Java classes have two primary elements: **methods** – often called functions or procedures in other languages, and **fields**, more generally known as variables. Together they are called the members of the class.

* **Variables hold** the state of the program,
* and **methods operate** on that state.

If a change is important to remember, a variable store that change.

**Classes vs. Files**

Most of the time Java classes are defined in their own .java file. Generally, they are public meaning any other class can call the class. Interestingly, Java does not require that the class be public. You can put two or more classes in a single .java file BUT only one class may be public, and it must have the same name as the file name else it will not compile!

**Writing a main() method**

* each file can contain only one public class but can contain multiple classes.
* The filename must match the class name, including case, and have a .java extension.
* The main() method signature is:
  + public static void main(String[] args){}
* java file is compiled with fist with command javac filename.java
* secondly its then run with the command java filename (omitting the .class JDK knows what file to use)
* if you want to pass parameters as arguments then you can add it after the filename example:
  + java filename 1 2 3 4
    - (parameters 1 2 3 4 is 4 different STRING INPUT PARAMETERS
    - These parameters are always of type String []

**Running a program in one line**

* Adding the .java to the filename using the java command gives us this function since Java 11 ONLY for single .java file programs. When you have more than one .java file you still need to use the javac filename.java -> java filename process to launch your program.
* This helps when creating small one file applications ideal for testing small single file programs.

## Understanding package declarations and imports

Java comes with thousands of built-in classes, and there are countless more from developers.

* Java puts classes in packages
* import java.util.Random; //example import tells us where to find Random class

**Wildcards (\*)**

import java.util.\* // this imports all classes included in the package java.util.\*

* doesn’t import child packages, fields, or methods; it imports only classes.
* There is a special type of import called the static import that imports other types

**Redundant imports**

* Java.lang package is automatically imported
* import java.lang.System
* import java.lang.\*

**naming conflicts**

* main reason to use packages is so that class names don’t need to be unique across all of Java.
* An example is the Date class. Java provides the java.util.Data and java.sql.Data
* Error : reference to Data is ambiguous
* If you import both imports with same name then the class named import takes precedence over wildcards. Yet still you can’t have two same name default classes imported, thus one default and one explicit reference.

**Creating a new package**

* The directory structure on your computer is related to the package name.
  + package packagea;
    - public class ClassA
  + package packageb;
    - import packagea.ClassA
* when running a java program, java knows where to look for those package names. In this case running from C:\temp works because both packagea and packageb are underneath it.
  + Running it from C:\temp\packageb/ClassB.java – this doesn’t work because of the java command to run a file directly only when that program is contained within a single file. Here ClassB.java relies on ClassA

**Compiling and running code with packages**

1. First compile all the .java files with command javac
   1. Example
      1. javac packagea/ClassA.java packageb/ClassB.java
      2. if you successful then two .class files created and can be run by java packageb.ClassB
2. Compiling with Wildcards
   1. You can use an asterisk to specify that you’d like to include all java files in a directory.
      1. example – javac packagea/\*.java packageb/\*.java

**Using an alternate Directory**

* by default javac command places the compiled classes in the same directory as the source code.
* An alternative option
  + javac packagea/ClassA.java packageb/ClassB.java
    - packagea
      * ClassA.java
      * ClassA.class
    - packageb
      * ClassB.java
      * ClassB.class
  + javac -d classes packagea/ClassA.java packageb/ClassB.java
    - packagea
      * ClassA.java
    - packageb
      * ClassB.java
    - classes
      * packagea
        + ClassA.class
      * Packageb
        + ClassB.class
  + java -cp classes packageb.ClassB
  + java -classpath classes packageb.ClassB
  + java --class-path classes packageb.ClassB
  + options for javac (location of classes needed to compile the program)
    - -cp <classpath>
    - -classpath <classpath>
    - --class-path <classpath>
    - -d <dir> (// directory to place generate class files)
  + Options for java (location of classes needed to run program)
    - -cp <classpath>
    - -classpath <classpath>
    - --class-path <classpath>

**Compiling with JAR Files**

Java archive (JAR) is like a zip file of mainly Java class files

java -cp “.;C:\temp\someOtherLocation;c:\temp\myJar.jar” myPackage.MyClass

java -cp “.:/tmp/someOtherLocation:/tmp/myJar.jar” myPackage.MyClass

java -cp “C:\temp\directoryWithJars\\*” myPackage.MyClass

this command will add all the JARs to the classpath that are in directoryWithJars.

**Creating a JAR File**

jar -cvf myNewFile.jar

jar –create –verbose –file myNewFile.jar

(alternatively specify the dir)

jar -cvf myNewFile.jar -C dir

|  |  |
| --- | --- |
| Option | Description |
| -c  --create | Creates a new JAR file |
| -v  --verbose | Prints details when working with JAR files |
| -f <filename>  --file <filename> | JAR Filename |
| -c <directory> | Directory containing files to be used to create the JAR |

**Running a program in one line with Packages**

You can use single-file source-code programs from within a package as long as they rely only on classes supplied by the JDK.

You can run it by:

* java Learning.java // from within the singleFile directory
* java singleFile/Learning.java //from the directory above singleFile

## Ordering Elements in a Class

**Order for declaring a class**

1. Package declaration (First line of the file)
2. Import statements
3. Class declaration (Required)
4. Field declarations
5. Method declarations

## Code Formatting on the exam

Common cases where you don’t need to check the imports:

* Code that begins with a class name
* Code that begins with a method declaration
* Code that begins with a code snippet that would normally be inside a class or method
* Code that has line numbers that don’t begin with 1

## Exam Essentials

**Identify benefits of Java**

* Object-oriented design
* Encapsulation
* Platform independence
* Robustness
* Simplicity
* Security
* Multithreading
* And backward compatibility

**Define common acronyms**

* JDK -> Java Development Kit and contains the compiler and JVM launcher
* JVM stands for Java Virtual Machine, and it runs bytecode.
* API is an application programming interface, which is code that you can call

**Be able to write code using a main() method**

public static void main(String[] args){

}

Arguments are referenced starting with args[0]. Accessing an argument that wasn’t passed in will cause the code to throw and exception.

**Understand the effect of using packages and imports**

* Packages contain Java classes
* Classes can be imported by class name or wildcard.
* Wildcards do not look at subdirectories
* In event of a conflict, class name imports take precedence

**Be able to recognize misplaced statements in a class**

* Package and import statements are optional
* If present both go before the class declaration in that order
* Fields and methods are also optional and are allowed in any order within the class declaration

# Chapter 2 (Java Building Blocks)

## Creating Objects

**Calling constructors**

Create an instance of a class

* Write “new” before the class name and add parentheses after it
* Example –> Park = new Park();

Key points:

* The name of the constructor matches the name of the class
* And there’s no return type

**Reading and Writing member fields**

* Create an instance of the class and use the reference of the instance to get the field/method (members) of the class.

**Executing instance initializer blocks**

{} – code block

{} outside a method in the class -> instance initializers

Balanced parentheses problem -> you can’t use the enclose brace “}” if there’s no corresponding open brace “{“that it matches written earlier in the code

**Following Order of Initialization**

1. Fields and instance initializer blocks are run in the order in which they appear in the file
2. The constructor runs after all fields and instance initializer blocks have run

## Understanding Data Types

**Using Primitive types**

Java has 8 built =-in data types, referred to as the Java primitive types.

The 8 building blocks of Java objects, because all java objects are just a complex collection of these primitive data types.

Primitive types are JUST a value and not a type of object.

|  |  |  |
| --- | --- | --- |
| boolean | True or false | True |
| byte | 8-bit integral value (-128->127) | 123 |
| short | 16-bit integral value | 123 |
| int | 32-bit integral value | 123 |
| long | 64-bit integral value | 123L |
| float | 32-bit floating value | 123.45f |
| double | 64-bit floating value | 123.46 |
| char | 16-bit Unicode value | ‘a’ |

**Writing Literals**

Literals are actual values that is assigned to a variable/ field for example:

longValue = 32432L; (L -> specify that it’s a long type)

VALID digits in several other formats:

- OCTAL (digits 0-7), which uses the number 0 as a prefix -- for example 017

- hexadecimal (digits 0-9 and letters A-F)

- binary (digits 0/1)

**Literals and the underscore character**

int million1 = 1000000; same as:

int million2 = 1\_000\_000;

invalid double values with INVALID USE OF \_

\_1000.00

3200\_.00

3300.00\_

These does NOT COMPILE

**Using Reference Types**

- a reference can be assigned to another object of the same type  
- a reference can be assigned to a new object using the new keyword  
  
A Reference type refers to an object (an instance of a class).  
A reference "points" to an object by storing the memory address where the object is located, a consept refeerred to as a pointer.

**Key Differences**

1. Reference types can be assigned NULL, (currently not referencing an object in memory)  
2. Primitive types will give you compiler error if you attempt to assign them null.  
3. Reference types can be used to call methods when they do not point to null.  
4. Primitives do not have methods declared on them they are just a value.  
5. Primitives types have lowercase type names. All classes that come with Java begin with uppercase.

## Declaring Variables

**Identifying identifiers**

* Identifiers must begin with a letter, a $ symbol, or a \_ symbol
* Identifiers can include numbers but not start with them
* Since Java 9, a single underscore \_is not allowed as an identifier
* You cannot use the same name as a java reserved word.

Style: camelCase

* First letter of each word is capitalized

Style: snake\_case

* Each word is separated by an \_

**Multiple variables in one line**  
// creates 4 string variables referencing NULL object  
public String s1**,**s2**,**s3**,**s4**;**// declared multiple variables with different string values all in one line, generally not good practice.  
public String s5 = "test1"**,**s6 = "test2"**,**s7 = "test3"**,**s8 = "test4"**;**// Catch question, only the i3 is given value 3;  
public int i1**,**i2**,**i3 = **3;**// int i2, String s2; does not compile for multiple line variables the same type needs to be shared.

**local variables :**

* Is a variable defined within a constructor, method or initializer block
* must be initialized before use
* do not have a default value and contain garbage data until initialized.

**Instance Variables** are variables that’s not local and are defined within the class like these below  
 DEFAULT initialization for instance and class variables only

instance variables can be initialized in an instance blocks

Boolean false  
byte,short,int,long 0  
float,double 0.0  
char '\u0000' (NUL)  
all object references (everything else) null

**Introducing var**

Starting in Java 10, you have the option of using the keyword var instead of the type for local variables under certain conditions.

* Example: var name = “hello”;
* Formal name of this feature is local variable type inference

**Type Inference of var**

**Type inference** -> when you type var, you are instructing the compiler to determine the type for you.

var number = 7;

number = 4;

number = “five”; // this will not compile. Initial var type inference is set to type of int thus a string is not of type int;

Type inference of the keyword var variable can’t be changes at runtime thus type is set at compile time.

**Var and null**

While a var cannot be initialized with a null value without a type, it can be assigned a null value after it is declared, provided that the underlying data type of the var is an object.

**Review of var Rules**

1. A var is used as a local variable in a constructor, method or initializer block
2. A var cannot be used in constructor parameters, method parameters, instance variables or class variables
3. A var is always initialized on the same line (or statement0 where it is declared
4. The value of a var can change but the type cannot
5. A var cannot be initialized with a null value without a type
6. A var is not permitted in a multiple-variable declaration
7. A var is a reserved type name but not a reserved word, meaning it can be used as an identifier except as a class, interface or enum name.

## Managing variable scope

**RULES OF SCOPE**  
1. local variables - in scope from declarations to end of block  
2. instance variables - in scope from declaration until object garbage collected  
3. class variables - in scope from declaration until program ends

## Destroying Objects

**GARBAGE Collection**  
- the process of automatically freeing memory on the heap by deleting objects that are  
 no longer reachable in your program  
- System.gc() is not guaranteed to run  
 + this method java provides merely suggests that now might be a good time for java to kick of GC.  
 java can ignore this suggestion  
- A object is ready for GC when its is no longer reachable by the program in two situations  
 + the object no longer has any references pointing to it  
 + all reference to the object have gone out of scope.  
- it is the object that gets GC not the reference

**finalize()**java allows objects to implement a method called finalize() that might get called.  
- gets called if the Garbage collector tries to collect the object  
- if garbage collector doesn't run this method doesn't run  
- will never be called twice!!

## Exam Essentials

**Be able to recognize a constructor**

* A constructor has the same name as the class, looks like a method without a return type

**Be able to identify legal and illegal declarations and initialization**

* Multiple variables can be declared and initialized in the same statement when they share a type
* Local variables require an explicit initialization, others use the default value for that type
* Identifiers may contain letters, numbers $, \_ although they may not begin with numbers
* Cannot define an identifier that is just a single underscore character \_
* Numeric literals may contain underscores between two digits, such as 1\_000, but not other places, such as \_100\_.0\_
* Numeric literals can begin with 1-9, 0,0x,0x,0b, and 0B, with the latter four indicating a change of numeric base.

**Be able to use var correctly**

* A var is used for a local variable inside a constructor, a method, or an initializer block
* It cannot be used for constructor parameters, method parameters, instance variables, or class variables.
* A var is initialized on the same line where it is declared, and while it can change value, it cannot change type.
* A var cannot be initialized with a null value without a type, nor can it be used in multiple variable declarations
* Var is not a reserved word in Java and can be used as a variable name

**Be able to determine where variables go into and out of scope**

* All variables go into scope when they are declared
* Local variables go out of scope when the block they are declared in ends
* Instance variables go out of scope when the object is eligible for garbage collection.
* Class variables remain in scope as long as the program is running

**Know how to identify when an object is eligible for garbage collections**

* Draw a diagram to keep track of references and objects as you trace the code.
* When no arrows point to a box (object), it is eligible for garbage collection

# Chapter 3 (Operators)

## Understanding Java Operators

**Types of Operators**

1. Unary
2. Binary
3. Ternary

These three operators can be applied to one, two or three operands respectively.

**Operator Precedence**

**TABLE 2.1 Order of operator precedence  
Operator Symbols and examples**

Post-unary operators expression++, expression--  
Pre-unary operators ++expression, --expression  
Other unary operators +, -, !, ~, (type)  
Multiplication/Division/Modulus \*, /, %  
Addition/Subtraction +, -  
Shift operators <<, >>, >>>  
Relational operators <, >, <=, >=, instanceof  
Equal to/not equal to ==, !=  
Logical operators &, ^, |  
Short-circuit logical operators &&, ||  
Ternary operators boolean expression ? expression1 : expression2  
Assignment operators =, +=, -=, \*=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

## Applying Unary Operators

**Unary operators**

* ! -> Inverts a boolean’s logical value
* + -> Indicates a number is positive, although numbers are assumed to be positive in java unless accompanied by a negative unary operator
* - -> Indicates a literal number is negative or negates an expression
* ++ -> Increments a value by 1
* -- -> Decrements a value by 1
* (type) -> Casts a value to a specific type

**Logical Complement (!) and negation operators (-)**

*Boolean isAnimalAsleep = false;*

*System.out.println(isAnimalAsleep); // false*

*isAnimalAsleep = !isAnimalAsleep;*

*System.out.println(isAnimalAsleep); // true*

*double dValue = 2.2;*

*System.out.println(dValue); // 2.2*

*System.out.println(-dValue); // -2.2*

**Increment and Decrement Operators**

*int parkAttendance = 0;*

*System.out.println(parkAttendance); // 0*

*System.out.println(++parkAttendance); // 1*

*System.out.println(parkAttendance); // 1*

*System.out.println(parkAttendance--); // 1*

*System.out.println(parkAttendance); // 0*

CATCH on the exam:

Multiple on the same line on same variable!

Example:

*int lion = 3;*

*int tiger = ++lion \* 5 / lion--; // ++lion(4) \* 5 / lion--(value = 4 but lion after value is 3)*

*System.out.println(“lion is “ + lion); // lion = 3*

*System.out.println(“tiger is “ + tiger); // tiger = 5*

## Working with Binary Arithmetic Operators

**Binary arithmetic operators**

* + -> Adds two numeric values
* - -> Subtracts two numeric values
* \* -> Multiplies two numeric values
* / -> Divides one numeric value by another
* % -> Modulus operator returns the remainder after division of one numeric value by another

**Numeric promotion**

* if two values have different data types one is promoted to the larger of the two data types:
* if an int and a float is added they will be promoted to float and result will be float.
* the result is the same data type as the promoted type.
* byte short and char are first promoted to int even if no int is in the equation

## Assignment Values

**Assignment Operator**

‘=’ Assigns the value on the right to the variable on the left

**Casting Values**

Casting is performed by placing the data type, enclosed in parentheses, to the left of the value you want to cast.

*int fur = (int)5;*

*int hair = (short)2;*

*short tail = (short)(4 + 10);*

* during the exam, remember to keep track of parentheses and return types any time casting is involved

**Compound Assignment Operators**

Besides the = simple assignment operator java supports numerous compound assignment operators:

* += -> Adds the value on the right to the variable on the left and assigns the sum to the variable
* -= -> Subtracts the value on the right from the variable on the left and assigns the difference to the variable
* \*= -> Multiplies the value on the right with the variable on the left and assigns the product to the variable
* /= -> Divides the variable on the left by the right and assigns the quotient to the variable

**Assignment Operator Return Value**

One final thing to know about assignment operators is that the result of an assignment is an expression in and of itself, equal to the value of the assignment. For example

*long wolf = 5;*

*long coyote = (wolf=3);*

*System.out.println(wolf); // 3*

*System.out.println(coyote); //3*

## Comparing Values

**Equality Operators**

* ==
  + Returns true if the two values represent the same value
  + Returns true if the two values reference the same object
* !=
  + Returns true if the two values represent different values
  + Returns true if the two values do not reference the same object

The equality operators are used in one of three scenarios:

1. Comparing two numeric or character primitive types. If the numeric values are of different types, the values are automatically promoted. For example 5 == 5.0 // true the int 5 gets promoted to 5.0
2. Comparing two boolean values
3. Comparing two objects, including null and String values

**Relational Operators**

* < -> Returns true if the value on the left is strictly less than the value on the right
* <= -> Returns true if the value on the left is less than or equal to the value on the right
* > -> Returns true if the value on the left is strictly greater than the value on the right
* >= -> Returns true if the value on the left is greater than or equal to the value on the right
* a instanceof b -> Returns true if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b

**Numeric Comparison Operators**

Only applies to numeric values

int gibbonNumFeet = 2, wolfNumFeet = 4, ostrichNumFeet = 2;

< -> gibbonNumFeet < wolfNumFeet // true

<= -> gibbonNumFeet <= wolfNumFeet // true

> -> gibbonNumFeet >= ostrichNumFeet // true

>= -> gibbonNumFeet > ostrichNumFeet // false

**instanceof Operator**

useful for determining whether an arbitrary object is a member of a particular class or interface at runtime.

* Any instance can be assigned to an Object reference
* It is common to use casting and instanceof together when working with objects that can be various different types, since it can give you access to fields available only in the more specific classes.

**Invalid instanceof**

* One area the exam might try to trip you up on is using instanceof with incompatible types.
* Example
  + Number cannot possibly hold a String value, so the following would cause a compilation error:
    - Public static void openZoo(Number time) {
      * If(time instanceof String) // DOES NOT COMPILE
      * …
  + It gets even more complicated as the previous rule applies to classes, but not interfaces.

**null and the instanceof operator**

calling instanceof on a null ALMOST always returns false except when calling instanceof on null instanceof null then it does not compile.

**Logical Operators**

* & -> Logical AND is true only if both values are true.
* | -> Inclusive OR is true if at least one of the values is true
* ^ -> Exclusive XOR is true only if one value is true and the other is false

Tips:

1. AND is only true if both operands are true
2. Inclusive OR is only false if both operands are false.
3. Exclusive OR is only true if the operands are different

**Short-Circuit Operators**

* && -> Short-circuit AND is true only if both values are true. If the left side is fals, thenthe right side will not be evaluated
* || -> Short-circuit OR is true if at least one of the values is true. If the left side is true, then the right side will not be evaluated

**Avoiding a NullPointerException**

if(duck!=null & duck.getAge() <5){} // could throw a nullpointerexception

if(duck!=null && duck.getAge() <5){} // short-circuit prevents a Nullpointerexception

**Checking for unperformed Side Effects**

* ***Unperformed side effect*** -> Be wary of short-circuit behaviour on the exam, as questions are known to alter a variable on the right side of the expression that may never be reached.

*int rabbit = 6;*

*Boolean bunny = (rabbit >= 6) || (++rabit <= 7);*

*System.out.println(rabbit);*

Because rabbit >= 6 is true, the increment operator on the right side of the expression is never evaluated, so the output is 6.

## Making Decisions with the Ternary Operator

Conditional operator ( : ? )

booleanExpression ? expression1 : expression2;

Ternary Expression also can have unperformed side effects

## Exam Essentials

* Be able to write code that uses Java operators
* Be able to recognize which operators are associated with which data types
* Understand when casting is required, or numeric promotion occurs
* Understand Java operator precedence
* Be able to write code that uses parentheses to override operator precedence

# Chapter 4 (Making Decisions)

## Creating Decision-Making Statements

**Statements and blocks**

Control flow statements break up the flow of execution by using decision-making, looping, and branching, allowing the application to selectively execute particular segments of code

*// single statement*

*patrons++;*

*//statement inside a block*

*{*

*patrons++;*

*}*

*//single statement*

*if(ticketsTaken >)*

*patrons++;*

*// statement inside a block*

*if(ticketsTaken > 1){*

*}*

**The *if* Statement**

*// basic check*

*if (x == 29){*

*/\**

*code to fire if expression x=29 is true*

*\*/*

*}*

*//if else statement*

*if (true){*

*/\**

*CODE TO FIRE WHEN EXPRESSION IS TRUE*

*\*/*

*} else {*

*/\**

*CODE TO FIRE WHEN EXPRESSION IS FALSE*

*\*/*

*}*

*// nested if else statement*

*if (x == 29){*

*/\**

*CODE TO FIRE WHEN EXPRESSION x == 29*

*\*/*

*} else if (x > 29){*

*/\**

*CODE TO FIRE WHEN EXPRESSION x > 29*

*\*/*

*} else {*

*/\**

*CODE TO FIRE WHEN EXPRESSION x is not 29 or > 29*

*\*/*

*}*

**The Switch Statement**

*Switch statement can be one of the following :*

*byte / Byte*

*char / Charachter*

*short / Short*

*int / Integer*

*enum*

*string*

*var (if the type resolves to one of the preceding types)*

*the value in the switch case statements has to be compile time constants*

*- only literals, enums constants or final constant variables of the same data types*

*NOT SUPPORTED FOR THE SWITCH STATEMENT IS THE double, float and long data types and their wrapper classes*

*switch (x){*

*case 1: {*

*break;*

*}*

*case 2: break;*

*case 3:*

*System.out.println("Hello int = 3");break;*

*case 4:*

*System.out.println("this is going to print 4 and 5 because no break or exists for int = 4");*

*case 5:*

*System.out.println("5");*

*default:*

*System.out.println("default int = " + x);break;*

*}*

**Numeric promotion and casting**

Numeric promotion is permitted that does not require an explicit cast.

*short size = 4;*

*final int small = 15;*

*final int big = 1\_000\_000;*

*switch(size) {*

*case small:*

*case 1+2:*

*case big: // does not compile*

*}*

## Writing while Loops

**While Statement**

public void whileLoop(){  
 boolean whileBool = true**;** int x = **0;** // while block runs only if conditional expression is true  
 while(whileBool){  
 x++**;** if(x == **5**)  
 whileBool = false**;** }  
}

**do/while statement**

public void doWhileLoop() {  
 boolean whileBool = true**;** int x = **0;** // do block runs always at least once and then checks while condition  
 do {  
 x++**;** if(x == **5**)  
 whileBool = false**;** }while(whileBool)**;**}

**Infinite loop**

* A loop that is true and continues infinitely
* int pen = 2;
* int pigs = 5;
* while(pen < 10)
  + pigs++;

## Constructing for loops

**The for loop**

public void forLoop(){  
 //for (count variable; condition; incremental/ decremental sequence)  
 for(int i = **0;** i <= **10;** i++){  
 System.*out*.println(i)**;** if (i == **5**)  
 break**;** }  
  
 //multiple variables are valid  
 for(int i = **0,** j = **10;** i < j**;** i++**,** j--){  
 System.*out*.println(i + " " + j)**;** if (i == j)  
 break**;** }  
  
// THIS IS A VALID LOOP BUT ITS INFINITE LOOP!!!  
// for(; ;){  
//  
// }  
 int counter = **0;** infinite: for(**; ;**){  
 ++counter**;** if (counter > **5**){  
 break infinite**;** // break statement jumps out of the current loop or to the pointed label  
 }else {  
 continue**;** // continue just skips the rest of the code in the loop and continue with the next sequence in the loop  
 }  
 }  
 }

**Printing elements in reverse**

for (var counter = 4; counter >= 0; counter--) {

System.out.print(counter + “ “ );

}

**Working with for loops**

1. Creating an infinite loop

for(**; ;**)  
 System.*out*.println("helloworld")**;**

1. Adding multiple terms to the for statement

int x = **0;**for(long y = **0,**z = **4;** x < **5** && y < **10;** x++**,** y++){  
 System.*out*.println(y + " ")**;**}  
System.*out*.println(x + " ")**;**

1. Redeclaring a variable in the initialization block

int x = **0;**for(int x = **4;** x < **5;** x++){ // does not compile  
 System.*out*.println(x + " ")**;**}

int x = **0;**for(x = **4;** x < **5;** x++){   
 System.*out*.println(x + " ")**;**}

1. Using incompatible types in the initialization block

int x = **0;**for(long y = **0,** int z = **4;** x < **5;** x++){ //does not compile  
 System.*out*.println(y + " ")**;**}

1. Using loop variables outside the loop

for(long y = **0,** x = **4;** x < **5** && y < **10;** x++**,** y++){  
 System.*out*.println(y + " ")**;**}  
System.*out*.println(x)**;** // Does not compile

**Modifying Loop Variable**

*for(int i=0; i<10; i++) // infinite loop*

*i=0;*

*for(int j=1; j<10; j++) // infinite loop*

*j--;*

*for(int k=0;k<10;)*

*k++;*

**the enhanced for loop**

/\*  
 for (datatype instance : collection){  
 //code  
 }  
 \*/  
String[] names = new String[]{"martin"**,** "michelle"**,** "logan"}**;**for (String name : names){  
 System.*out*.println("name = " + name)**;**}

* The right side needs to be an array or Iterable object and
* the left side has a matching type

**Switching Between for and for-each loops**

for (String name : names){  
 System.*out*.println("name = " + name)**;**}  
for(int i=**0;**i<names.length**;** i++){  
 String name = names[i]**;** System.*out*.println(name + " ")**;**}

## Controlling flow with branching

**Nested Loops**

* is a loop that contains another loop including while, do/while, for, and for-each loops.

int[][] twoD = new int[**3**][**2**]**;**for (int i = **0;** i < twoD.length**;** i++){  
 for(int j = **0;** j < twoD[i].length**;** j++){  
 System.*out*.print(twoD[i][j] + " ")**;**// printing element  
 }  
 System.*out*.println()**;** // time for new line between each array  
}

**Adding optional Labels**

int[][] twoD = new int[**3**][**2**]**;**OUTER\_LOOP: for (int i = **0;** i < twoD.length**;** i++){  
 INNER\_LOOP: for(int j = **0;** j < twoD[i].length**;** j++){  
 System.*out*.print(twoD[i][j] + " ")**;**// printing element  
 }  
 System.*out*.println()**;** // time for new line between each array  
}

**the break Statement**

a break statement transfers the flow of control out to the enclosing statement.

*optionalLabel: while(booleanExpression) {*

*//body*

*// somewhere in loop*

*break optionalLabel;*

**the continue statement**

*optionalLabel: while(booleanExpression) {*

*//body*

*// somewhere in loop*

*continue optionalLabel;*

**the return statement**

* creating methods and using return statements can be used as an alternative to using labels and break statements.

**Unreachable code**

* one facet of break, continue and return that you should be aware of is that any code placed immediately after them in the same block is considered unreachable and will not compile.

int checkDate = **0;**while(checkDate<**10**){  
 checkDate++**;** if(checkDate>**100**){  
 break**;** checkDate++**;** // does not compile (UNREACHABLE CODE)  
 }  
}

**Reviewing branching**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Allows optional labels | Allows break statement | Allows continue statement |
| while | Yes | Yes | Yes |
| do-while | Yes | Yes | Yes |
| for | Yes | Yes | Yes |
| switch | Yes | Yes | No |

## Exam Essentials

* Understand if and else decision control statements
* Understand switch statements and their proper usage
* Understand while loops
* Be able to use for loops
* Understand how break, continue, and return can change flow control.

# Chapter 5 (Core Java APIs)

## Creating and manipulating Strings

**The String Pool**

The String pool , also known as the intern pool, is a location in the java virtual machine (JVM) that collects all these strings.

Strings not in the String pool are garbage collected just like any other object that doesn't have a reference.

**Concatenation**

1. If both operands are numeric, + means numeric addition
2. If either operand is a String, + means concatenation
3. The expression is evaluated left to right

System.*out*.println(**1** + **2**)**;**// 3  
System.*out*.println("a" + "b")**;**// ab  
System.*out*.println("a" + "b" + **3**)**;**// ab3  
System.*out*.println(**1** + **2** + "c")**;**// 3c  
System.*out*.println("c" + **1** + **2**)**;**// c12

**Immutability**

String is immutable - not allowed to change!  
immutable object -> an object that can't be change once it's created

example:  
class Mutable {  
 private String s;  
 public void setS(String newS){ s = newS };//setters make class mutable  
 public String getS() {return s;}  
}  
  
class Immutable {  
 private String s;  
 public String getS() {return s;}  
}

**Important String Methods**

public static void ImportantStringMethods(){  
 String s1 = "12345"**;** String s2 = "martin"**;** String s3 = "Hello World"**;** // string methods:  
 System.*out*.println(s1.length())**;** // 5  
 System.*out*.println(s2.charAt(**3**))**;**// t  
 System.*out*.println(s3.indexOf('l'))**;**// 2  
 System.*out*.println(s1.substring(**1,4**))**;**//234  
 System.*out*.println(s3.toLowerCase())**;** // hello world  
 System.*out*.println(s3.toUpperCase())**;** // HELLO WORLD  
 System.*out*.println(s1.equals("12345"))**;** // true  
 System.*out*.println(s3.equalsIgnoreCase("hello WORLD"))**;** // true  
 System.*out*.println(s2.startsWith("mar"))**;** // true  
 System.*out*.println(s2.endsWith("tin"))**;** // true  
 System.*out*.println(s3.contains(" "))**;** // true  
 System.*out*.println(s2.replace("martin"**,**"michelle"))**;** // michelle  
 System.*out*.println(" Hello World ".trim().equals(s3))**;**// method chaining and wil return true.  
 System.*out*.println(" Hello World ".trim())**;** // Hello World  
 System.*out*.println(" Hello World ".strip())**;** // Hello World  
 System.*out*.println(" Hello World ".stripLeading())**;** // "Hello World "  
 System.*out*.println(" Hello World ".stripTrailing())**;** // " Hello World"  
 System.*out*.println(s2.intern())**;**}

**length()** -> returns the number of characters in the String, *int length()*

**charAt()** -> lets you query the string to find out what character is at a specific index, char charAt(int index)

**indexOf()** -> looks at the characters in the string and finds the first index that matches the desired value

* + - int indexOf(int ch)
    - int indexOf(int ch, int fromIndex)
    - int indexOf(String str)
    - int indexOf(String str, int fromIndex)

**substring()** -> looks for characters in a string. Returns parts of the string. First parameter is the index to start with for the returned String. As usual, this is zero-based index. There is an optional second parameter, which is the end index you want to stop at.

**toLowerCase() and toUpperCase()** -> String toLowerCase() / String toUpperCase()

**equals() and equalsIgnoreCase()** -> checks whether two String objects contain exactly the same characters in the same order. The equalsIgnoreCase() method checks whether two String objects contain the same characters with the exception that it will convert the characters case if needed.

-> boolean equals(Object obj)

-> boolean equalsIgnoreCase(String str)

**startsWith() and endsWith()** -> look at whether the provided value matches part of the String

-> boolean startsWith(String prefix)

-> boolean endsWith(String suffix)

**replace()** -> a simple search and replace on the string. There’s a version that takes char parameters as well as a version that takes CharSequence parameters

-> String replace(char oldChar, char newChar)

-> String replace(CharSequence target, CharSequence replacement)

**contains()** -> looks for matches in the String. It isn’t as partical as startsWith() and endsWith() – the match can be anywhere in the String

**trim(), strip(), stripLeading(), and stripTrailling()**

-> strip() and trim() removes whitespace from the beginning and end of a String.

-> whitespace for exam is spaces along with the \t(tab) and \n (newline)

-> strip() is new in SE11 it’s the same as trim() but supports Unicode

-> stripLeading() method removes whitespace from the beginning of the String and leaves it at the end.

-> stripTrailing() method does the opposite

**intern()** -> returns the value from the string pool if it is there. Otherwise, it adds the value to the string pool.

-> we will explain about the string pool and give examples for intern() later in the chapter

**Method Chaining**

A series of methods on a variable output in sequence.

String result = “AniMal “.trim().toLowerCase().replace(‘a’, ‘A’);

System.out.println(result);

## Using the StringBuilder Class

StringBuilder class creates a String without storing all those interim String Values. Unlike the String class, StringBuilder is not immutable

**Mutability and Chaining**

Exam tests mutability between String and StringBuilder.

Chaining makes this even more interesting.

StringBuilder sb = new StringBuilder(“start”);

Sb.append(“+middle”);

StringBuilder same = sb.append(“+end”);

**Creating a StringBuilder**

public void creatingStringBuilder(){  
 // StringBuilder = mutable / changeable  
 StringBuilder a = new StringBuilder("abc")**;** StringBuilder b = a.append("de")  
 b.append("5") = b.append("f").append("g")**;** System.*out*.println("a="+ a)**;** //abcdefg  
 System.*out*.println("b="+ b)**;** //abcdefg  
 //both a and b points to object new StringBuilder("abc");  
 // thus any changes on either a or b returns the same value since its the same object  
}

**Important StringBuilder Methods**

public void ImportantStringBuilderMethods(){  
 StringBuilder sb = new StringBuilder("1234")**;** sb.charAt(**2**)**;** sb.indexOf("3")**;** sb.length()**;** sb.substring(**0,3**)**;** sb.append("abc")**;** sb.append("3232"**,0,4**)**;** sb.insert(**0,**"cde")**;** sb.delete(**3,5**)**;** sb.deleteCharAt(**4**)**;** sb.reverse()**;** sb.replace(**0,3,**"321")**;**

sb.toString()**;** System.*out*.println(sb.toString())**;**}

## Understanding Equality

**Comparing equals() and ==**

StringBuilder one = new StringBuilder()**;**StringBuilder two = new StringBuilder()**;**StringBuilder three = one.append("a")**;**System.*out*.println(one == two)**;** // false  
System.*out*.println(one == three)**;** // true

Authors of StringBuilder did not implement equals() thus .equals() on stringbuilder is the same as ==

## Understanding Java Arrays

**Creating an Array of Primitives**

int[] numbers1 = new int[3];

(type of array + array symbol) (var name) = new int[(size)];

When you use this form to instantiate an array, all elements are set to the default value for that type.

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 2 |

Element :

Index:

int[] numbers2 == new int[] {42, 55, 99};

|  |  |  |
| --- | --- | --- |
| 42 | 55 | 99 |
| 0 | 1 | 2 |

Element :

Index:

**Shortcut:**

int[] numbers2 = {42,55, 99};

int[] numAnimals;

int [] numAnimals2;

int []numAnimals3;

int numAnimals4[];

int numAnimals5 [];

int[] ids, types; // two arrays

int ids[], types; // one array and one int var

**Creating an Array with Reference Variables**

You can choose any Java type to be the type of the array. This includes classes you create yourself.

String [] bugs = { “cricket”, “beetle”, “ladybug” };

|  |  |  |  |
| --- | --- | --- | --- |
| Elemant | Cricket | Beetle | Ladybag |
| Index | 0 | 1 | 2 |

**Using an Array**

String[] mammals = {"monkey"**,** "chimp"**,** "donkey"}**;**System.*out*.println(mammals.length)**;** // 3  
System.*out*.println(mammals[**0**])**;** // monkey  
System.*out*.println(mammals[**1**])**;** // chimp  
System.*out*.println(mammals[**2**])**;** // donkey

Reading items of an array or writing values of an array:

int[] numbers = new int[**10**]**;**for (int i = **0;** i < numbers.length**;** i++){  
 numbers[i] = i + **5;**}

**Sorting**

import java.util.\*; // import whole package including Arrays

import java.util.Arrays; // import just Arrays

int[] numbers = { **6, 9, 1**}**;**Arrays.*sort*(numbers)**;**for (int i = **0;** i < numbers.length**;** i++){  
 System.*out*.println(numbers[i] + "")**;** // 1,6,9  
}

**Searching**

Java also provides a convenient way to search – but only if the array is already sorted.

|  |  |
| --- | --- |
| **Scenario** | **Result** |
| Target element found in sorted array | Index of match |
| Target element not found in sorted array | Negative value showing one smaller than the negative of the index, where a match needs to be inserted to preserve sorted order |
| Unsorted array | A surprise – this result isn’t predictable |

int[] numbers1 = {**2,4,6,8**}**;**System.*out*.println(Arrays.*binarySearch*(numbers1**,2**))**;** // 0  
System.*out*.println(Arrays.*binarySearch*(numbers1**,4**))**;** // 1  
System.*out*.println(Arrays.*binarySearch*(numbers1**,1**))**;** // -1  
System.*out*.println(Arrays.*binarySearch*(numbers1**,3**))**;** // -2  
System.*out*.println(Arrays.*binarySearch*(numbers1**,9**))**;** // -5

// ONLY WHEN THE ARRAY IS ALREADY SORTED!!  
//BINARY SEARCH  
int[] i1 = {**8,6,2,4**}**;**Arrays.*sort*(i1)**;** // numbers sort fine.  
System.*out*.println("Binary search returns the index where a match has been found")**;**System.*out*.println("Binary search returns a negative index where a match needs to be inserted to preseve sorted order")**;**/\*  
 \*/

We negate and subtract 1 for consistency, getting -1-1, also known as -2. Finally searching for 9 in the i1 array it tells us that 9 should be inserted in index 4 and we again negate and subtract 1 getting -4 -1 = -5  
**Comparing**

compare()

* A negative number means the first array is smaller than the second
* A zero means the arrays are equal
* A positive number means the first array is larger than the second
  + Example
    - System.out.println(Arrays.compare(new int[] {1}, new int[] {2}));
    - This code prints a negative number. It should be pretty intuitive that 1 is smaller than 2, making the first array smaller.
* Comparing arrays of different lengths
  + If both arrays are the same length and have the same values in each spot in the same order, return zero
  + If all the elements are the same but the second array has extra elements at the end, return a negative number
  + If all the elements are the same but the second array has extra elements at the end, return a negative number
  + If all the elements are the same but the first array has extra elements at the end, return a positive number
  + If the first element that differs is smaller in the first array, return a negative number
  + If the first element that differs is larger in the first array, return a positive number
* What does smaller mean?
  + Null is smaller than any other value
  + For numbers, normal numeric order applies.
  + For strings, one is smaller if it is a prefix of another
  + For strings/ characters, numbers are smaller than letters
  + For strings/ characters, uppercase is smaller than lowercase

|  |  |  |  |
| --- | --- | --- | --- |
| **First Array** | **Second array** | **Result** | **Reason** |
| new int[] {1,2} | new int[] {1} | Positive number | The first element is the same, but the first array is longer |
| new int[] {1,2} | new int[] {1,2} | Zero | Exact match |
| new String[] {“a”} | new String[] {“aa”} | Negative number | The first element is a substring of the second |
| new String[] {“a”} | new String[] {“A”} | Positive number | Uppercase is smaller than lowercase |
| new String[] {“a”} | new String[] {null} | Positive number | Null is smaller than a letter |

**mismatch()**

if the arrays are equal mismatch() returns -1. Otherwise, it returns the first index where they differ.

|  |  |  |
| --- | --- | --- |
| Method | When arrays are the same | When arrays are different |
| equals() | true | false |
| compare() | 0 | Positive or negative number |
| mismatch() | -1 | Zero or positive index |

**Varargs**

When you’re creating an array yourself, it looks like what we’ve seen thus far. When one is passed to your method, there is another way it can look.

public static void main(String[] args)

public static void main(String args[])

public static void main(String… args) // varargs …

varargs is used as an array

**Multidimensional Arrays**

An array with array elements.

**Creating a multidimensional Array**

int[][] vars1; // 2D array

int vars2 [][]; // 2D array

int[] vars3[]; // 2D array

int[] vars4[], space[][]; //a 2D and a 3D array

**Using a Multidimensional Array**

int[][] twoD = new int[**3**][**2**]**;**for (int i = **0;** i < twoD.length**;** i++){  
 for(int j = **0;** j< twoD[i].length**;**j++){  
 System.*out*.println(twoD[i][j] + " ")**;**// print element  
 }  
 System.*out*.println()**;**//new row  
}

for (int[] inner: twoD){  
 for(int num: inner){  
 System.*out*.println(num + " ")**;** }  
}

## Understanding an ArrayList

**Creating an ArrayList**

Old method of ArrayList constructors

ArrayList list1 = new ArrayList()**;**ArrayList list2 = new ArrayList(**10**)**;**ArrayList list3 = new ArrayList(list2)**;**

New method of ArrayList constructors

ArrayList<String> list4 = new ArrayList<String>()**;**ArrayList<String> list5 = new ArrayList<>()**;**

**Using an ArrayList**

1. import java.util.\*; // OR  
 import java.util.ArrayList;   
2. ArrayLists can change size at runtime as needed  
3. ArrayList is an ordered sequence that allows duplicates

**add()** -> insert a new value in the ArrayList.

* boolean add(E element)
* void add(int index, E element)

**remove()** -> remove the first matching value in the ArrayLIst or remove the element at a specified index.

* boolean remove(Object object)
* E remove(int index)

**set()** -> changes one of the elements of the elements of the ArrayList without changing the size.

* E set(int index, E newElement)

**isEmpty() and size()** -> methods look at how many of the slots are in use

* boolean isEmpty()
* int size()

**clear()** -> method provides an easy way to discard all elements of the ArrayList

* void clear()

**contains()** -> method checks whether a certain value is in the ArrayList

* boolean contains(Object object)

**equals()** -> ArrayList has a custom implementation of equals(), so you can compare two lists to see whether they contain the same elements in the same order.

* boolean equals(Object object)

**Wrapper Classes**

Up to now, we’ve only put String objects in the ArrayList. What happens if we want to put primitives in? each primitive type has a wrapper class, which is an object type that corresponds to the primitive.

|  |  |  |  |
| --- | --- | --- | --- |
| **Primitive type** | **Wrapper class** | **Example of creating** | **Example of constructing** |
| boolean | Boolean | Boolean.valueOf(true) | new Boolean(true) |
| byte | Byte | Byte.valueOf((byte)1) | new Byte((byte) 1) |
| short | Short | Short.valueOf((short)1) | new Short((short) 1) |
| int | Int | Integer.valueOf(1) | new Integer(1) |
| long | Long | Long.valueOf(1) | new Long(1) |
| float | Float | Float.valueOf((float)1.0) | new Float(1.0) |
| double | Double | Double.valueOf(1.0) | new Double(1.0) |
| char | Char | Character.valueOf(‘c’) | new Character('c') |

***Converting from String***

|  |  |  |
| --- | --- | --- |
| **Wrapper class** | **Converting String to a primitive** | **Converting String to a wrapper class** |
| Boolean | Boolean.parseBoolean(“true”) | Boolean.valueOf(“TRUE”) |
| Byte | Byte.parseByte(“1”) | Byte.valueOf(“2”) |
| Short | Short.parseShort(“1”) | Short.valueOf(“2”) |
| Int | Integer.parseInt(“1”) | Integer.valueOf(“2”) |
| Long | Long.parseLong(“1”) | Long.valueOf(“2”) |
| Float | Float.parseFloat(“1”) | Float.valueOf(“2.2”) |
| Double | Double.parseDouble(“1”) | Double.valueOf(“2.2”) |
| Char | None | None |

**Autoboxing and Unboxing**

**autoboxing** -> Java will convert it to the relevant wrapper class for you. Primitive <-> Wrapper class

List<Double> weights = new ArrayList<>()**;**weights.add(**50.5**)**;** //[50.5]  
weights.add(new Double(**60**))**;** //[50.5, 60]  
weights.remove(**50.5**)**;** //[60]  
double first = weights.get(**0**)**;** //60.0  
//unboxing a null gives nullpointer exception

**Converting Between array and List**

ArrayList -> Array

List<String> list = new ArrayList()**;**list.add("hawk")**;**list.add("robin")**;**Object[] objectArray = list.toArray()**;**String[] stringArray = list.toArray(new String[**0**])**;**list.clear()**;**System.*out*.println(objectArray)**;**//2  
System.*out*.println(stringArray.length)**;**//2

Array -> ArrayList

String[] array = {"hawk"**,** "robin"}**;** // [hawk, robin]  
List<String> list = Arrays.*asList*(array)**;** // returns fixed size list  
System.*out*.println(list.size())**;** // 2  
list.set(**1,**"test")**;** // [hawk, test]  
array[**0**] = "new"**;** // [new, test]  
System.*out*.println(Arrays.*toString*(array))**;** // [new, test]  
list.remove(**1**)**;** // throws UnsupportedOperstionException  
  
String[] array1 = {"hawk"**,** "robin"}**;** // [hawk, robin]  
List<String> list2 = List.*of*(array1)**;** // returns immutable list  
System.*out*.println(list2.size())**;** // 2  
list2.set(**1,**"test")**;** // [hawk, test]  
array1[**0**] = "new"**;** // [new, test]  
System.*out*.println(Arrays.*toString*(array1))**;** // [new, test]  
list2.remove(**1**)**;** // throws UnsupportedOperstionException

**Using Varargs to Create a List**

Varargs allows you to create a List in a cool way:

* List<String> list1 = Arrays.asList(“one”, “two”);
* List<String> list2 = List.of(“one”, “two”);

|  |  |  |  |
| --- | --- | --- | --- |
|  | toArray() | Arrays.asList() | List.of() |
| Type converting from | List | Array (or varags) | Array (or varargs) |
| Type created | Array | List | List |
| Allowed to remove values from created object | No | No | No |
| Allowed to change values in the created object | Yes | Yes | No |
| Changing values in the created object affects the original or vice versa | No | Yes | N/A |

**Sorting**

Sorting an ArrayList is similar to sorting an array you just use a different helper class:

List<Integer> numbers = new ArrayList<>()**;**numbers.add(**99**)**;**numbers.add(**5**)**;**numbers.add(**81**)**;**Collections.*sort*(numbers)**;**System.*out*.println(numbers)**;** //[5,81,99]

## Creating Sets and Maps

**Introducing Sets**

A set is a collection of objects that cannot contain duplicates. If you try to add a duplicate to a set, the API will not fulfil the request.

* A set is not an ordered
* When trying to add a duplicate value, the method returns false and does not add the value.
* Two common classes that implement Set
  + HashSet
  + TreeSet

Set<Integer> set = new HashSet<>()**;**System.*out*.println(set.add(**66**))**;** // true  
System.*out*.println(set.add(**66**))**;** // false  
System.*out*.println(set.size())**;** // 1  
set.remove(**66**)**;**System.*out*.println(set.isEmpty())**;** // true

**Introducing Maps**

A map uses a key to identify values.

* Most common use of a map is a HashMap. Some of the methods are the same as those in ArrayList like clear(), isEmpty(), and size().

**Common Map Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| V get(Object key) | Returns the value mapped by key or null if none is mapped |
| V getOrDefault(Object key, V other) | Returns the value mapped by key or other if none is mapped |
| V put(K key, V value) | Adds or replaces key/value pair. Returns previous value or null |
| V remove(Object key) | Removes and returns value mapped to key. Returns null if none |
| boolean containsKey(Object key) | Returns whether key is in map |
| boolean containsValue(Object value) | Returns whether value is in map |
| Set<K> keyset() | Returns set of all keys |
| Collection<V> values() | Returns Collection of all values |

Map<String**,** String> map = new HashMap<>()**;**map.put("koala"**,** "bamboo")**;**String food = map.get("koala")**;** // bamboo  
String other = map.getOrDefault("ant"**,** "leaf")**;** // leaf  
for (String key: map.keySet()){  
 System.*out*.println(key + " " + map.get(key))**;**}

## Calculating with Math APIs

**min() and max()**

compares two values and return one of them

* double min(double a, double b)
* float min(float a, float b)
* int min(int a, int b)
* long min(long a, long b)
* double max(double a, double b)
* float max(float a, float b)
* int max(int a, int b)
* long max(long a, long b)

int first = Math.max(3,7); // 7

int second = Math.min(7,-9); // -9

**round()**

* method gets rid of the decimal portion of the value, choosing the next higher number if appropriate.
* If the fractional part is .5 or higher, we round up.

long round(double num)

int round(float num)

long low = Math.round(123.45); // 123

long high = Math.round(123.5); // 124

int fromFloat = Math.round(123.45f) //123

**pow()**

* method handles exponents, as you may recall from your elementary school math class, 3 to power 2 means three squared. This is 3\*3 or 9
* fractional exponents are allowed as well.

double pow(double number, double exponent)

double squared = Math.pow(5,2); // 25.0

**random()**

method returns a value greater than or equal to 0 and less than 1

double random()

double num = Math.random();

## Exam Essentials

**Be able to determine the output of code using String** ->

* know the rules for concatenating Strings and how to use common String methods.
* Know that Strings are immutable
* Pay special attention to the fact that indexes are zero-based and that substring() gets the string up until right before the index of the second parameter

**Be able to determine the output of code using StringBuilder** ->

* Know that StringBuilder is mutable and how to use common StringBuilder methods.
* Know that substring() does not change the value of a StringBuilder, whereas append(), delete() and insert() do change it.
* Also note that most StringBuilder methods return a reference to the current instance of StringBuilder

**Understand the difference between == and equals()** ->

* == checks object equality
* equals() depends on the implementation of the object it is being called on
* for Strings, equals() checks the characters inside of it

**Be able to determine the output of code using arrays** ->

* know how to declare and instantiate one-dimensional and multidimensional arrays
* be able to access each element and know when an index is out of bounds
* recognize correct and incorrect output when searching and sorting

**Be able to determine the output of code using ArrayList** ->

* know that ArrayList can increase in size
* be able to identify the different ways of declaring and instantiating an ArrayList
* identify correct output from ArrayList methods, including the impact of autoboxing

# Chapter 6 (Lambdas and Functional Interfaces)

## Writing Simple Lambdas

**Lambda Example**

Deferred execution-> that code is specified now but will run later.

**Lambda Syntax**

One of the simplest lambda expression you can write is the one you just saw

a -> a.canHop();

* lambdas work with interfaces that have only one abstract method
* the lambda indicates that Java should call a method with an Animal parameter that returns a Boolean value that’s the result of a.canHop().below is the same:
  + a -> a.canHop()
    - A single parameter specified with the name a
    - The arrow operator to separate the parameter and body
    - A body that calls a single method and returns the result of that method
  + (Animal a) -> { return a.canHop(); }
    - A single parameter specified with the name a and stating the type is Animal
    - The arrow operator to separate the parameter and body
    - A body that as one or more lines of code, including a semicolon and a return statement.

|  |  |
| --- | --- |
| **Lambda** | **# parameters** |
| () -> true | 0 |
| a -> a.startsWith(“test”) | 1 |
| (String a) -> a.startsWith(“test”) | 1 |
| (a, b) -> a.startsWith(“test”) | 2 |
| (String a, String b) -> a.startsWith(“test”) | 2 |

NOTES:

* All of these examples have parentheses around the parameter list except the one that takes only one parameter and doesn’t specify the type.
* The first row takes zero parameters and always returns eh boolean value true.
* The second row takes one parameter and calls a method on it, returning the result.
* The third row does the same except that it explicitly defines the type of the variable.
* The final two rows take to parameters and ignore one of them – there isn’t a rule that says you must use all defined parameters.

**INVALID Syntax to identify invalid**

|  |  |
| --- | --- |
| Invalid lambda | Reason |
| a, b -> a.startsWith(“test”) | Missing parentheses |
| a -> { a.startsWith(“test”); } | Missing return |
| a -> { return a.startsWith(“test”) } | Missing semicolon |

*Remember that the parentheses are optional only when there is one parameter and it doesn’’t have a type declared.*

## Introducing Functional Interfaces

* Lambdas work with interfaces that have only one abstract method (functional interfaces)
* SAM -> Single Abstract Method rule

**Predicate**

public interface Predicate<T> {

boolean test(T t);

}

**Consumer**

void accept(T t)

**Supplier**

T get()

**Comparator**

Comparator<Integer> ints = (i1, i2) -> i1 – i2;

s1.compareTo(s2); // ascending

s2.compareTo(s1); // decending

**Basic Functional interfaces**

|  |  |  |
| --- | --- | --- |
| **Functional interface** | **# parameters** | **Return Type** |
| Comparator | Two | int |
| Consumer | One | void |
| Predicate | One | boolean |
| Supplier | none | One (type varies) |

## Working with variables in Lambdas

Variables can appear in three places with respect to lambda: the parameter list, local variables declared inside the lambda body, and variables referenced from the lambda body.

All three of these are opportunities for the exam to trick you

|  |  |
| --- | --- |
| Variable type | Rule |
| Instance variable | Allowed |
| Static variable | Allowed |
| Local variable | Allowed if effectively final |
| Method parameter | Allowed if effectively final |
| Lambda parameter | Allowed |

**Parameter List**

Specifying the type of parameters is optional, the following are all valid.

* Predicate<String> p = x -> true;
* Predicate<String> p = (var x) -> true;
* Predicate<String> p = (String x) -> true;

Exam can ask you to identify the type of the lambda parameter.

* A lambda infers the types from the surrounding context.
* Lambda is being assigned to a predicate that takes a String
* A place to look for the type is in a method signature
* Sorting a list, we can use the type of the list to determine the type of the lambda parameter

**Local Variables inside the Lambda Body**

While it is most common for a lambda body to be a single expression, it is legal to define a block. That block can have anything that is valid in a normal Java block, including local variable declarations.

* When writing your own code, a lambda block with a local variable is a good hint that you should extract that code into a method

**Variables Referenced from the Lambda Body**

Lambda bodies are allowed to reference some variables from the surrounding code.

* Method parameters and local variables are allowed to be referenced if they are effectively final
* Means that the value of a variable doesn’t change after it is set, regardless of whether it is explicitly marked as final.
* If you aren’t sure whether a variable is effectively final

## Calling APIs with Lambdas

Most common methods that use them on the exam.

1. removeIf()
2. sort()
3. forEach()

List<String> bunnies = new ArrayList<>()**;**bunnies.add("long ear")**;**bunnies.add("fluffy")**;**bunnies.add("hoppy")**;**System.*out*.println(bunnies)**;** // ["long ear", "fluffy", "hoppy"]  
bunnies.sort((b1**,** b2) -> b1.compareTo(b2))**;**System.*out*.println(bunnies)**;** // ["fluffy", "hoppy", "long ear"]  
bunnies.removeIf(s -> s.charAt(**0**) != 'h')**;**System.*out*.println(bunnies)**;** // ["long ear", "fluffy"]  
bunnies.forEach(b -> System.*out*.println(b))**;**Map<String**,** Integer> bunniesMap = new HashMap<>()**;**bunniesMap.put("long ear"**,1**)**;**bunniesMap.put("fluffy"**,2**)**;**bunniesMap.put("hoppy"**,3**)**;**bunniesMap.forEach((k**,**v) -> {  
 System.*out*.println(k + " " + v)**;**})**;**

## Summary

Lambda expressions, or lambdas, allow passing around blocks of code

(String a, String b) -> { return a.equals(b); }

* the parameter types can be omitted
* when only one parameter is specified without a type the parentheses can also be omitted
* the braces and return statement can be omitted for a single statement, making the short form as follows:
  + a -> a.equals(b)
* Lambdas are passed to a method expecting an instance of a functional interface.
* A functional interface is one with a single abstract method
* Predicate is a common interface that returns a Boolean and that takes any type.
* Consumer takes any type and doesn’t return a value.
* Supplier returns a value and does not take any parameters.
* Comparator takes two parameters and returns an int.
* A lambda can define parameters or variables in the body as long as their names are different from existing local variables
* The body of a lambda is allowed to use any instance or class variables
* Additionally, it can use any local variables or method parameters that are effectively final
* Three common APIs that use lambdas
  + removeIf() -> method on a List and a Set takes a Predicate
  + sort() -> method on a List interface takes a Comparator
  + forEach() -> methods on a List and a Set interface takes a Consumer

## Exam Essentials

**Write simple lambda expressions**

* look for the presence or absence of optional elements in lambda code.
* Parameter types are optional
* Braces and the return keyword are optional when the body is a single statement
* Parentheses are optional when only one parameter is specified and the type is implicit

**Identify common functional interfaces**

* From a code snippet, identify whether the lambda is a Comparator, Consumer, Predicate, or Supplier.
* You can use the number of parameters and return type to tell them apart.

**Determine whether a variable can be used in a lambda body**

* Local variables and method parameters must be effectively final to be referenced
* This means the code must compile if you were to add the final keyword to theses variables
* Instance and class variables are always allowed

**Use common APIs with lambdas**

* Be able to read and write code using forEach(), removeIf(), and sort().

# Chapter 7 (Methods and Encapsulation)

## Designing methods

**Method declaration example**

public final void nap(int minutes) throws InterruptedException {

//take a nap

}

|  |  |  |
| --- | --- | --- |
| **Element** | **Value in nap() example** | **Required?** |
| Access modifiers | public | No |
| Optional specifier | final | No |
| Return type | void | Yes |
| Method name | nap | Yes |
| Parameter list | (int minutes) | Yes, but can be empty parentheses |
| Optional exception list | throws InterruptedException | No |
| Method body\* | {  // take a nap  } | Yes, but can be empty braces |

\*Body omitted for abstract methods.

**Access Modifiers**

* **private** -> Only accessible within the same class
* **Default** (Package-Private) Access -> private plus other classes in the same package, no keyword you simply omit the access modifier.
* **protected** -> can be called only from classes in the same package or subclasses
* **public** -> can be called from any class

**Optional Specifiers**

* static -> used for class methods
* abstract -> used when a method body is not provided
* final -> used when a method is not allowed to be overridden by a subclass
* synchronized -> used with multithreaded code
* native -> the native modifier is used when interacting with code written in another language such as c++
* strictfp -> used for making floating-point calculations portable

**Return Type**

* value type that is returned
* if no value is returned then the return type is void

**Method Name**

* an identifier may only contain letters, numbers, $ or \_
* first character is not allowed to be a number
* reserved words are not allowed
* a single \_ character is not allowed
* by convention a method begins with a lowercase letter but are not required to

**Parameter List**

* parameter list is required but it doesn’t have to contain any parameters
* multiple parameters separated with a comma

**Optional Exception List**

* you can list as many types of exceptions as you want in this clause separated by commas

**Method Body**

* method body is simply a code block
* contains zero or more Java statements

## Working with Varargs

* a varargs parameter must be the last element in a method’s parameter list
* only allowed to have one varargs parameter per method
* when calling a method with a varargs parameter, you have a choice :
  + you can pass in an array or you can list the elements of the array and let Java create it for you
  + you can even omit the varargs values in the method call and Java will create an arary of length zero for you
* accessing an varargs variable is like accessing an array with an index

private void varargs() {  
 walk(**1**)**;** // 0  
 walk(**1,2**)**;** // 1   
 walk(**1,2,3**)**;** // 2  
 walk(**1,**new int[] {**4,5**})**;** // 2

walk(**1,**null)**;** // throws a nullPointerException in walk()

}  
  
private void walk(int i**,** int... nums) {  
 System.*out*.println(nums.length)**; //** length of null = nullPointexception}

## Applying Access Modifiers

* **private** -> Only accessible within the same class
* **Default (Package-Private) Access** -> private plus other classes in the same package
* **protected** -> Default access plus child classes
* **public** -> protected plus classes in the other packages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A method in \_\_\_\_\_ can access a \_\_\_\_\_ member** | **private** | **Default(package-private)** | **protected** | **public** |
| the same class | Yes | Yes | Yes | Yes |
| another class in the same package | No | Yes | Yes | Yes |
| in a subclass in a different package | No | No | Yes | Yes |
| an unrelated class in a different package | No | No | No | Yes |

## Applying the *static* Keyword

* static keyword is applied to a variable, method, or class, it applies to the class rather than a specific instance of the class
* you will see that the static keyword can also be applied to import statements

**Designing static Methods and Fields**

* static methods don’t require an instance of the class
* they are shared among all users of the class
* static variable is a member of the single class object that exists independently of any instances of that class

a class main method can also be called from another class since it also is just a static method.

* for utility or helper methods that don’t require any object state, since there is no need to access instance variables, having static methods eliminates the need for the caller to instantiate an object just to call the method.
* for state that is shared by all instances of a class, like a counter. All instances must share the same state. Methods that merely use that state should be static as well.

**Accessing a static Variable or Method**

* you just put the class name before the method or variable and you are done
  + System.out.println(Koala.count);
  + Koala.main(new String[0]);
* you can use an instance of the object to call a static method
  + the compiler checks for the type of the reference anduses that instead of the object

Valid:

Koala k = new Koala()**;**System.*out*.println(k.*count*)**;**k = null**;**System.*out*.println(k.*count*)**;**System.*out*.println(Koala.*count* == k.*count*)**;// true**

Koala.*count* = **4;**Koala koala1 = new Koala()**;**Koala koala2 = new Koala()**;**koala1.*count* = **6;**koala2.*count* = **5;**System.*out*.println(Koala.*count*)**;** // 5

**Static vs. Instance**

**static**

* cannot call an instance member without referencing an instance of the class
* the compiler will give you an error about making a static reference to a nonstatic method
* a static method or instance method can call a static method because static methods don’t require an object to use
* only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object

a common use for static variables is counting the number of instances:

private static int *count***;**public Counter() {*count*++}  
public static void main(String[] args) {  
 Counter c1 = new Counter()**;** Counter c2 = new Counter()**;** Counter c3 = new Counter()**;** System.*out*.println(*count*)**;**}

**static Variables**

* static variables are meant to change as the program runs.
* Counters are a common example of this. We want the count to increase over time
* just as with instance variables, you can initialize a static variable on the line it is declared:

public class Initializers {

private static int counter = 0; // initiailization

}

Other static variables are meant to never change during the program. this type of variables is known as a ***constant*** it uses the final modifier to ensure the variable never changes.

**Constants** use the modifier **static final** and a different naming convention than other variables.

example:

public class Initializers {  
 private static final int *NUM\_BUCKETS* = **45;** public static void main(String[] args) {  
 *NUM\_BUCKETS* = **5;** // compile error since constants can't be changed  
 }  
}

private static final ArrayList<String> *values* = new ArrayList<>()**;**public static void main(String[] args){  
 *values*.add("changed")**;**}

* it actually compiles since values is a reference variable. we are allowed to call methods on reference variables
* all the compiler can do is check that we don’t try to reassign the final values to point to a different object.

**Static Initialization**

* code blocks with the static keyword in front of it to specify they should be run when the class is first loaded.
* all static initializers run when the class is first used in the order they are defined
* the statements in them run and assign any static variables as needed.
* all static variables needs to be initialized will give compile error if not

private static final int *NUM\_SECONDS\_PER\_MINUTE***;**private static final int *NUM\_MINUTES\_PER\_HOUR***;**private static final int *NUM\_SECONDS\_PER\_HOUR***;**static{  
 *NUM\_SECONDS\_PER\_MINUTE* = **60;** *NUM\_MINUTES\_PER\_HOUR* = **60;**}  
  
static{  
 *NUM\_SECONDS\_PER\_HOUR* = *NUM\_SECONDS\_PER\_MINUTE* \* *NUM\_MINUTES\_PER\_HOUR***;**}

**Static Imports**

* importing a specific class or all the classes in a package

imports are convenient because you don’t need to specify where each class comes from each time you use it. regular imports are for importing classes.

* static imports are for importing static members of classes

import java.util.List**;**import static java.util.Arrays.*asList***;**public class ChapterSevenMain {  
 public static void main(String[] args){  
 List<String> list = *asList*("one"**,**"two")**;** }  
}

## Passing Data among Methods

Java is a “pass-by-value” language this means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller.

public static void main(String[] args){  
 int num = **4;** *newNumber*(num)**;** System.*out*.println(num)**;** // 4  
   
 String name = "Webby"**;** *speak*(name)**;** System.*out*.println(name)**;** // Webby  
   
 StringBuilder nameSB = new StringBuilder()**;** *speakSB*(nameSB)**;** System.*out*.println(nameSB)**;** // Sparky  
}  
  
private static void speakSB(StringBuilder s) {  
 s.append("Sparky")**;**}  
  
private static void speak(String name) {  
 name = "sparky"**;**}  
  
private static void newNumber(int num) {  
 num = **8;**}

public static void main(String[] args){  
 int num = **1;** // num = 1  
 String letters = "abc"**;** // letters = abc  
 *number*(num)**;** // num = 1  
 letters = *letters*(letters)**;** // letters=abcd  
 System.*out*.println(num + " " + letters)**;** // 1abcd  
}  
  
private static String letters(String letters) {  
 letters += "d"**;** return letters**;**}  
  
private static int number(int num) {  
 num++**;** return num**;**}

## Overloading Methods

Method overloading occurs when

* methods have the same name but different method signatures, which means they differ by method parameters,
* different number of method parameters
* different type, more types, or the same types in different order.
* return type access modifier and exception list are irrelevant to overloading

### Overloading Complex scenarios

**Varargs**

parameter list is the same for an varargs variable and an array thus:

public void fly(int[] lengths){}  
public void fly(int... lengths){} // Does not compile

**Autoboxing**

public void fly(int numMiles){}  
public void fly(Integer numMiles){}

* Java will match the int numMiles version.
* Java tries to use the most specific parameter list it can find
* When the primitive int version isn’t present, it will autobox.

**Reference Types**

Java picking the most specific version of a method that it can.

**Primitives**

* Java picking the most specific version of a method that it can.
* Java has no problem calling a larger primitive, if no specific type is available
* Java can only accept wider types.
* Java will not automatically convert to a narrower type -> for this you need explicit casting

**Generics**

**type erasure** -> where generics are used only at compile time

public void walk(List<String> strings) {}  
public void walk(List<Integer> integers) {} // DOES NOT COMPILE

public void walk(List strings) {}  
public void walk(List integers) {} // DOES NOT COMPILE

**Arrays**

public void walk(int[] ints) {}  
public void walk(Integer[] integers) {}

Arrays have been around since the beginning of Java. they specify their actual types and don’t participate in type erasure.

**Putting it all Together**

* java calls the most specific method it can
* when some of the types interact, the Java rules focus on backward compatibility.
* autoboxing and varargs come last when Java looks at overloaded methods
  + As accommodating as Java is with trying to find a match, it will do only one conversion when it comes to autoboxing

Official order that java uses to choose the right overloaded method

|  |  |
| --- | --- |
| **Rule** | **Example of what will be chosen for glide(1,2)** |
| Exact match by type | String glide(int i, int j) |
| Larger primitive type | String glide(long i, long j) |
| Autoboxed type | String glide(Integer i, Integer j) |
| Varargs | String glide(int… nums) |

## Encapsulating Data

**Encapsulation** -> means only methods in the class with the variables can refer the instance variables.

-> Callers are required to use these methods

-> getter(accessor) and setter(mutator) methods

-> the data (an instance variable) is private and getters/setters are public

**Naming conventions for getters and setters**

|  |  |
| --- | --- |
| **Rule** | **Example** |
| Getter methods most frequently begin with ***is*** if the property is a boolean. | public boolean isHappy(){  return happy;  } |
| Getter methods most frequently begin with ***get*** if the property is not a boolean. | public int getNumber(){  return number;  } |
| Setter methods begin with set | public void setHappy(boolean \_happy){  happy = \_happy;  } |

## Exam Essentials

* Be able to identify correct and incorrect method declarations
* Identify when a method or field is accessible
* Recognize valid and invalid uses of static imports
* State the output of code involving methods
* Recognize the correct overloaded method
* Identify properly encapsulated classes

# Chapter 8 (Class design)

## Understanding Inheritance

***Inheritance*** is the process by which a subclass automatically includes any public or protected members of the class, including primitives, objects, or methods, defined in the parent class.

***subclass / child class*** -> any class that inherits from another class.

***superclass / parent class*** -> the class that the child inherits from.

Example:

class Z { }  
  
class Y extends Z { }  
  
class X extends Y { }

**Inheritance** is transitive

if child class X inherits from parent class Y, which in turn inherits from a parent class Z, then class X would be considered a subclass, or descendant, of class Z. By Comparison, X is a direct descendant only of class Y, and Y is a direct descendant only of class Z.

* When one class inherits from a parent class, all public and protected members are automatically available as part of the child class.
* Package-private members are available if the child class is in the same package as the parent class.
* private members are restricted to the class they are defined in and are never available via inheritance. This doesn’t mean the parent class doesn’t have private members that can hold data or modify an object; it just means the child class has on direct reference to them.

class BigCat {  
 public double size = **0;**}  
  
class Jaquar extends BigCat {   
 public Jaquar(){  
 size = **10.2;** }  
 public void print(){  
 System.*out*.println(size)**;** }  
}

**Single vs. Multiple Inheritance**

* Java supports ***single inheritance***, by which a class may inherit from only on direct parent class.
* Java supports ***multiple levels of inheritance***, by which one class may extend another class, which in turn extends another class.
  + any number of levels are allowed
* Java DOES NOT support multiple inheritance
* Java does allow ***multiple interfaces***
* Parent class can have multiple children
* it is possible in Java to prevent a class from being extended by marking the class with the final modifier.
* if you try to define a class that inherits from a final class, then the class will fail to compile.

**Inheriting Object**

in Java all classes inherit from java.lang.Object, or Object for short. Furthermore, Object is the only class that doesn’t have a parent class.

## Creating Classes

Creating Class relationships with inheritance

**Extending a Class**

Defining a extending a class

public abstract class ElephantSeal extends Seal {  
 // Methods and Variables defined here  
}

**Applying Class Access Modifiers**

**top-level class** -> is a class that is not defined inside another class

**inner class** -> is a class defined inside of another class and is the opposite of a top-level class. In addition to public and package-private access, inner classes can also have protected and private access.

a java file can have many top-level classes but at most one public top-level class. it may also have no public class at all.

**Accessing the “*this”* Reference**

* the “this” reference refers to the current instance of the class and can be used to access any member of the class, including inherited members.
* it can be used in any instance method, constructor, and instance initializer block
* it cannot be used when there is no implicit instance of the class, such as in a static method or static initializer block.
* class Duck {  
   private String color**;** private int height**;** private int length**;** public void setData(int lenght**,** int theHeight){  
   length = this.length**;** // backwards - no good  
   height = theHeight**;** this.color = "white"**;** }  
    
   public static void main(String[] args) {  
   Duck b = new Duck()**;** b.setData(**1,2**)**;** System.*out*.println(b.length + " " + b.height + " " + b.color)**;** // 0 2 white  
   }  
  }

**Calling the “super” Reference**

* in Java a variable or method can be defined in both a parent class and a child class.
* **super (keyword)** -> the super reference is similar to the “this reference, except that it excludes any members found in the current class.
* in other words, the member must be accessible via inheritance

## Declaring Constructors

**Creating a Constructor**

* The name of the constructor matches the name of the class
* no return type
* Java is case sensitive
* Constructors are used when creating a new object. This process is called instantiation because it creates a new instance of the class.
* a constructor is called when we write new followed by the name of the class we want to instantiate. Example
  + Turtle newTurtle = new Turtle();
* declaring multiple constructors with different signatures is referred to as constructor overloading. Example:

public class Turtle {  
 private String name**;** public Turtle(){}  
 public Turtle(String name) {  
 this.name = name**;** }   
 public Turtle(int age){}  
 public Turtle(long age){}  
 public Turtle(String newName**,** String... favoriteFoods){}  
}

**Default Constructor**

* Every class in Java has a constructor whether you code one or not.
* if you don’t include any constructors in the class, Java will create one for you without any parameters
* this Java-created constructor is called the ***default constructor*** and is added anytime a class is declared without any constructors.
* also referred to as the default no-argument constructor for clarity

**Calling Overloaded Constructor with *this()***

* a single class can have multiple constructors, this is referred to as constructor overloading, because all constructors have the same inherent name but a different signature. (See Turtle class example above)
* to avoid duplication you can have one constructor call another constructor in the same class with this();
* When this() is used with parentheses, Java calls another constructor on the same instance of the class.
* the this() call must be the first statement in the constructor else it will not compile and it’s the only call a constructor can make to another constructor.

**this vs. this()**

same keyword but very different

this -> refers to an instance of the class

this() -> refers to a constructor call within the same class

**Calling Parent Constructors with *super()***

In Java, the first statement of every constructor is either a call to another constructor within the class, using ***this()***, or a call to a constructor in the direct parent class, using ***super()***

* one or the other can be used in the first line within the constructor but not both this gives compiler error
* the super() can be used to any valid constructor signature in the direct parent class

public class Animal {  
 private int age**;** public Animal(int age){  
 super()**;** // Refers to constructor in java.lang.Object  
 this.age = age**;**

}  
}

public class Zebra extends Animal {  
 public Zebra(int age) {  
 super(age)**;**// Refers to constructor in Animal  
 }  
 public Zebra() {  
 this(**4**)**;**// Refers to constructor in Zebra with int argument  
 }  
}

**super vs. super()**

super -> is used to reference members of the parent class

super() -> is used to call a parent constructor

**Understanding Compiler Enhancements**

Java compiler automatically inserts a call to the no-argument constructor super() if you do not explicitly call this() or super() as the first line of a constructor.

These three Donkey classes are the same java automatically compiles them into the last one with a no-args constructor referencing a super();

public class Donkey {}  
  
public class Donkey {  
 public Donkey (){}  
}  
  
public class Donkey {  
 public Donkey (){  
 super()**;** }  
}

**Missing a Default No-Argument Constructor**

* Recall that the default no-argument constructor is not required and is inserted by the compiler ONLY IF there is no constructor defined in the class.

public class Mammal {  
 public Mammal (int age){}  
}  
  
public class Elephant extends Mammal {  
 public Elephant (){  
 super()**;** // DOES NOT COMPILE  
 }  
}

* Since the mammal class has at least one constructor declared, the compiler does not insert a default no-argument constructor. therefore, the super() call in the Elephant class declaration does not compile

public class Elephant extends Mammal {  
 public Elephant (){  
 super(**10**)**;**   
 }  
}

* this fixes the compile error, we have added a constructor with an explicit call to a parent constructor.
  + notice that the class Elephant now has a no-argument constructor even if their parent classes do not. provided the constructor of the child maps to a parent constructor via an explicit call of the super() command.
* that subclasses of the Elephant can rely on compiler enhancements.
* be wary of any exam question in which a class defines a constructor that takes arguments and doesn’t define a no-argument constructor.

**Constructors and *final* Fields**

* constructor is part of the initialization process, so it is allowed to assign final instance variables in it.
* RULE-> by the time the constructor completes, all final instance variables must be assigned a value.
* final fields should be assigned a value only once, and failure to assign a value is considered a compiler error in the constructor.

On the exam, be wary of any instance variables marked final. Make sure they are assigned a value in:

* the line where they are declared, in an instance initializer, or in a constructor

**Order of Initialization**

**Class Initialization**

* a class must be initialized before it is referenced or used
* the class containing the program entry point, aka the main() method, is loaded before the main() method is executed

*Initialize Class X*

1. if there is a superclass Y of X, then initialize class Y first.
2. Process all static variable declarations in the order they appear in the class
3. Process all static initializers in the order they appear in the class.

example:

public class Animal {  
 static{System.*out*.println("A")**;**}  
}  
  
public class Hippo extends Animal {  
 static {  
 System.*out*.println("B")**;** }  
  
 public static void main(String[] args) {  
 System.*out*.println("C")**;** new Hippo()**;** new Hippo()**;** new Hippo()**;** }  
}

this example prints “ABC” exactly once.

**Instance Initialization**

An instance is initialized anytime the new keyword is used.

*Initialize Instance of X*

1. If there is a superclass Y of X, then initialize the instance of Y first
2. Process all instance variables declarations in the order they appear in the class
3. Process all instance initializers in the order they appear in the class
4. Initialize the constructor including any overloaded constructors referenced with this()

**Reviewing Constructor Rules**

1. The first statement of every constructor is a call to an overloaded constructor via this(), or a direct parent constructor via super().
2. if the first statement of a constructor is not a call to this() or super(), then the compiler will insert a no-argument super() as the first statement of the constructor
3. Calling this() and super() after the first statement of a constructor results in a compiler error.
4. if the parent class doesn’t have a no-argument constructor, then every constructor in the child class must start with an explicit this() or super() constructor call
5. if the parent class doesn’t have a no-argument constructor and the child doesn’t define any constructors, then the child class will not compile.
6. if a class only defines private constructors, then it cannot be extended by a top-level class
7. all final instance variables must be assigned a value exactly once by the end of the constructor. any final instance variables not assigned a value will be reported as a compiler error on the line the constructor is declared.

## Inheriting Members

Java allows subclasses to replace, or override, the parent method implementation at runtime

**Calling Inherited Members**

* Java classes may use any public or protected member of the parent class, including methods, primitives, or object references.
* if the parent class and child class are part of the same package, then the child class may use any package-private members defined in the parent class.
* a child class may never access a private member of the parent class, at least not through any direct reference.
* To reference a member in a parent class, you can just call it directly, as in the following.

**Inheriting Methods**

Inheriting a class not only grants access to inherited methods in the parent class but also sets the stage for collisions between methods defined in both the parent class and the subclass

**Overriding a Method**

Scenario: If there is a method in a child class that needs to behave differently, than the method of the parent class. Solution is overriding.

Java, overriding a method occurs when a subclass declares a new implementation for an inherited method with the same signature and compatible return type. Remember that a method signature includes the name of the method and method parameter.

* when you override a method, you may reference the parent version of the method using the super keyword.

public class Canine {  
 public double getAverageWeight(){  
 return **50;** }  
}

public class Wolf extends Canine{  
 public double getAverageWeight() {  
 return super.getAverageWeight() + **20;** }  
  
 public static void main(String[] args) {  
 System.*out*.println(new Canine().getAverageWeight())**;** // 50.0  
 System.*out*.println(new Wolf().getAverageWeight())**;** // 70.0  
 }  
}

**Method Overriding and Recursive Calls**

public double getAverageWeight() {  
 return getAverageWeight() + **20;// STACKOVERFLOW ERROR**}

the super keyword is used to reference the parent class in order to ensure a termination condition and prevent a recursive infinite loop call to happen and produce a stackoverflow error at runtime.

Compiler performs the following checks when you override a method:

1. the method in the child class must have the same signature as the method in the parent class
2. the method in the child class must be at least as accessible as the method in the parent class
3. the method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. if the method returns a value, it must be the same or a subtype of the method in the parent class, known as covariant return types.

***Defining Subtype and Supertype***

A subtype is the relationship between two types where one type inherits the other. if we define X to be a subtype of Y, then one of the following is true:

* X and Y are classes, and X is a subclass of Y
* X and Y are interfaces, and X is a sub interfaces of Y
* X is a class and Y is an interface, and X implements Y (either directly or through an inherited class)

A supertype is the reciprocal relationship between two types where one type is the ancestor of the other.

**Overriding a Generic Method**

**type erasure** -> where generics are used only at compile time

public void walk(List<String> strings) {}  
public void walk(List<Integer> integers) {} // DOES NOT COMPILE

For the same reason, you also can’t overload a generic method in a parent class.

public class LongTailAnimal {  
 protected void chew(List<Object> input) {}  
}  
  
public class Anteater extends LongTailAnimal {  
 protected void chew(List<Double> input) {} // does not compile  
}

Both of these examples fail to compile because of type erasure. in the compiled form,

**Generic Method Parameters**

On the other hand, you can override a method with genetic parameters, but you must match the signature including the generic type EXACTLY. Example:

public class LongTailAnimal {  
 protected void chew(List<String> input) {}  
}  
  
public class Anteater extends LongTailAnimal {  
 protected void chew(List<String> input) {} // does not compile  
}

**Generic Return Types**

The generic parameter type must match its parent’s type exactly

Once you’ve determined which methods are overridden and which are being overloaded, work backward, making sure the generic types match for overridden methods.

* generic methods cannot be overloaded by changing the generic parameters type only.

**Redeclaring *private* Methods**

In Java, you can’t override private methods since they are not inherited.

* a child class can still redeclare its own method of its own.
* Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class.
* this method in the child class is a separate and independent method, unrelated to the parent version’s method.

**Hiding Static Methods**

**hidden method** -> occurs when a child class defines a static method with the same name and signature as an inherited static method defined inn a parent class.

Method hiding is similar but not exactly the same as method overriding. The previous four rules for overriding a method must be followed when a method is hidden.

In addition, a new rule is added to the rules of overriding for hiding a method:

1. the method defined in the child class must be marked as static if it is marked as static in a parent class.

put simply. it is method hiding if the two methods are marked static, and method overriding if they are not marked static. if one is marked static and the other is not, the class will not compile.

public class Bear {  
 public static void eat(){  
 System.*out*.println("Bear is eating")**;** }  
}  
  
public class Panda extends Bear {  
 public static void eat(){  
 System.*out*.println("Panda is chewing")**;** }  
  
 public static void main(String[] args) {  
 *eat*()**;** }  
}

NOTE: The four rules for overriding method must be followed when hiding static methods.

**Creating *final* Methods**

* marking a method final, forbids a child class from replacing this method. This rule is in place both when you override a method and when you hide a method.
* You cannot hide a static method in a child class if it is marked final in the parent class.

**Hiding Variables**

Java doesn’t allow for variables to be overridden, BUT can be hidden

a *hidden variable* occurs when a child class defines a variable with the same name as an inherited variable defined in the parent class.

* this creates two distinct copies of the variable within an instance of the child class: one instance defined in the parent class and one defined in the child class.

public class Carnivore {  
 protected boolean hasFur = false**;**}  
  
public class Meerkat extends Carnivore {  
 protected boolean hasFur = true**;** public static void main(String[] args) {  
 Meerkat m = new Meerkat()**;** Carnivore c = m**;** System.*out*.println(m.hasFur)**;** // true  
 System.*out*.println(c.hasFur)**;** // false  
 }  
}

* even though there is only one object created by the main() method, both variables exist independently of each other.
* the output changes depending on the reference variable used.

## Understanding Polymorphism

Java supports ***polymorphism***, the property of an object to take on many different forms.

* a java object may be accessed using
  + a reference with the same type as the object,
  + a reference that is a superclass of the object, or
  + a reference of that defines an interface the object implements, either directly or through a superclass.
* a cast is not required if the object is being reassigned to a super type or interface of the object.

**Interface Primer**

* interface can define abstract methods
* a class can implement any number of interfaces
* a class implements an interface by overriding the inherited abstract methods
* an object that implements an interface can be assigned to a reference for that interface.

public class Primate {  
 public boolean hasHair(){  
 return true**;** }  
}  
  
public interface HasTail{  
 public abstract boolean isTailStriped()**;**}  
  
public class Lemur extends Primate implements HasTail{  
 public boolean isTailStriped(){  
 return false**;** }  
 public int age = **10;** public static void main(String[] args){  
 Lemur lemur = new Lemur()**;** System.*out*.println(lemur.age)**;** HasTail hasTail = lemur**;** System.*out*.println(hasTail.isTailStriped())**;** Primate primate = lemur**;** System.*out*.println(primate.hasHair())**;** }  
}

This code compiles and prints the following output:

10

false

true

* Polymorphism enables an instance of Lemur to be reassigned or passed to a method using one of its supertypes, such as Primate or HasTail
* Once the object has been assigned to a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast.

**Object vs. Reference**

objects are accessed by reference, so as a developer you never have direct access to the object itself.

* The object as the entity that exists in memory, allocated by the Java runtime environment.
* regardless of the type of the reference you have for the object in memory, the object itself doesn’t change.
  + Lemur lemur = new Lemur();
  + Object lemurAsObject = lemur;
* even though the Lemur object has been assigned to a reference with a different type, the object itself has not changed and still exists as a Lemur object in memory.
* what has changed, then, is our ability to access methods within the lemur class with the lemurAsObject reference. Without an explicit cast back to Lemur, as you’ll see in the next section, we no longer have access to the Lemur properties of the object.
* it is possible to reclaim access to the variable age by explicitly casting the hasTail reference to a reference of type Lemur.

Rules

1. the type of the object determines which properties exist within the object in memory
2. the type of the reference to the object determines which methods and variables are accessible to the Java program

**Casting Objects**

We created a single instance of a Lemur object and accessed it via superclass and interfaces references. once we changed the reference type, though, we lost access to more specific members defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:

Primate primate = lemur**;**Lemur lemur2 = primate**;** // DOES NOT COMPILE  
System.*out*.println(lemur2.age)**;**Lemur lemur3 = (Lemur)primate**;** // Explicit Cast  
System.*out*.println(lemur3.age)**;**

Casting Rules summarized:

1. Casting a reference from a **subtype to a supertype doesn’t require an explicit cast**
2. Casting a reference from a **supertype to a subtype requires an explicit cast**
3. The compiler disallows casts to an unrelated class
4. At runtime, an invalid cast of a reference to an unrelated type results in a ClassCastException being thrown

public class Rodent{}  
public class Capybara extends Rodent {  
 public static void main(String[] args) {  
 Rodent rodent = new Rodent()**;** Capybara capybara = (Capybara) rodent**;** // ClassCastException  
 }  
}

**The *instanceof* Operator**

if(rodent instanceof Capybara){  
 Capybara capybara = (Capybara) rodent**;** // ClassCastException   
}

just as the compiler does not allow casting an object to unrelated types, it also does not allow instanceof to be used with unrelated types. (// DOES NOT COMPILE)

**Polymorphism and Method Overriding**

Polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class.

public class Penguin{  
 public int getHeight(){  
 return **3;** }  
 public void printInfo(){  
 System.*out*.println(this.getHeight())**;** }  
}  
public class EmperorPenguin extends Penguin {  
 public int getHeight(){  
 return **8;** }  
 public static void main(String[] fish){  
 new EmperorPenguin().printInfo()**; // prints 8;** }  
}

* key points
  + the Reference type determines the value and which methods are used
  + even the this. call can be used to be tricked in the exam.
* the facet of polymorphism that replaces methods via overriding is one of the most important properties in all of Java.
* it allows you to create complex inheritance models, with subclasses that have their own custom implementation of overridden methods.
* it also means the parent class does not need to be updated to use the custom or overridden method.
* if the method is properly overridden, then the overridden version will be used in all the places that it is called.

**Overriding vs. Hiding members**

* Overriding replaces the method everywhere it is called, static method and variable hiding does not.
* hiding members is not a form of polymorphism since the methods and variables maintain their individual properties.
* unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.
* CrestedPenguin example below using Hidden method with the static keyword and changes the output from the previous example which is similar.

public class Penguin{  
 public static int getHeight(){  
 return **3;** }  
 public void printInfo(){  
 System.*out*.println(this.*getHeight*())**;** }  
}  
public class CrestedPenguin extends Penguin {  
 public static int getHeight(){  
 return **8;** }  
 public static void main(String[] fish){  
 new CrestedPenguin().printInfo()**; // prints 3** }  
}

more complex example:

class Marsupial{  
 protected int age = **2;** public boolean isBiped()(){  
 return false**;** }  
}  
public class Kangaroo extends Marsupial {  
 protected int age = **6;** public boolean isBiped(){  
 return true**;** }  
  
 public static void main(String[] args) {  
 Kangaroo joey = new Kangaroo()**;** Marsupial moey = joey**;** System.*out*.println(joey.isBiped())**;** // true  
 System.*out*.println(moey.isBiped())**;** // false  
 System.*out*.println(joey.age)**;** // 6  
 System.*out*.println(moey.age)**;** // 2  
 }  
}

## Exam Essentials

* Be able to write code that extends other classes
* be able to distinguish and make use of this, this(), super, and super()
* Evaluate code involving constructors
* Understand the rules for method overriding
* Understand the rules for hiding methods and variables
* Recognize the difference between method overriding and method overloading
* Understand polymorphism
* Recognize valid reference casting

# Chapter 9 (Advanced Class Design)

## Creating Abstract Classes

**Introducing Abstract Classes**

abstract class -> is a class that cannot be instantiated and may contain abstract methods.

abstract method -> is a method that does not define an implementation when it is declared

both abstract classes and methods are denoted with the abstract modifier.

abstract class Bird{  
 public abstract String getName()**;** public void printName(){  
 System.*out*.println(getName())**;** }  
}  
  
public class Stork extends Bird {  
 public String getName(){ return "Stork!"**;**}  
 public static void main(String[] args){  
 new Stork().printName()**;** }   
}

**Defining Abstract Methods**

an abstract class can include:

1. nonabstract methods
2. all of the same members as a nonabstract class, including variables, static and instance methods, and inner classes.
3. constructors

* One of the most important features of an abstract class is that it is **not actually required to include any abstract methods.**
* although an abstract class doesn’t have to declare any abstract methods, **an abstract method can only be defined in an abstract class**
* the abstract modifier cannot be placed after the class keyword in a class declaration, nor after the return type in a method declaration.

**Constructors in Abstract Classes**

abstract classes are initialized with constructors in the same way as nonabstract classes. for example if an abstract class does not proved a constructor, the compiler will automatically insert a default no-argument constructor.

primary difference:

* a constructor in abstract class can be called only when it is being initialized by a nonabstract subclass, this makes sense as abstract classes cannot be instantiated.

**Invalid Abstract Method Declarations**

public abstract class Turtle(){  
 public abstract long eat() // DNC - missing ;  
 public abstract void swim() {}**;** // DNC - marked abstract but has implementation  
 public abstract int getAge() {  
 return **10;** } // DNC - marked abstract but has implementation  
 public void sleep**;** // DNC - missing parentheses   
 public void goInShell()**;** // DNC - not marked as abstract and has no implementation   
}

**Invalid Modifiers**

* if you mark something as final you are preventing anyone from extending or implementing it
* thus, abstract and final are in direct conflict with each other
* Java does not permit a class or method to be marked both abstract and final.
* a method cannot be marked as both abstract and private
* a method cannot be marked as both abstract and static

**Creating a Concrete Class**

an abstract class becomes usable when it Is extended by a concrete subclass.

**concrete class** -> nonabstract class.

* the first concrete subclass that extends an abstract class is required to implement all inherited abstract methods.
* this includes implementing any inherited abstract methods from inherited interfaces
* in exam make sure a concrete class implements ALL methods from the abstract class

**Reviewing Abstract Class Rules**

1. Abstract classes cannot be instantiated
2. all top-level types, including abstract classes, cannot be marked protected or private
3. Abstract classes cannot be marked final
4. Abstract classes may include zero or more abstract and nonabstract methods
5. an abstract class that extends another abstract class inherits all its abstract methods
6. The first concrete class that extends an abstract class must provide an implementation for all the inherited abstract methods
7. abstract class constructors follow the same rules for initialization as regular constructors, except they can be called only as part of the initialization of a subclass.

**Abstract Method Definition Rules**

1. Abstract methods can be defined only in abstract classes or interfaces
2. Abstract methods cannot be declared private or final
3. abstract methods must not provide a method body/implementation in the abstract class in which they are declared
4. implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations, etc.

## Implementing Interfaces

**Defining an interface**

* interface is an abstract data type are that declares a list of abstract methods that any class implementing the interface must provide.
* an interface can also include constant variables
* both abstract methods and constant variables included with an interface are implicitly assumed to be public.

**Defining an Interface**

in Java, an interface is defined with the interface keyword, analogous to the class keyword used when defining a class.

public interface CanBurrow {  
 public abstract Float getSpeed(int age)**;** public static final int *MINIMUM\_DEPTH* = **2;**}

==

public interface CanBurrow {  
 Float getSpeed(int age)**;** int *MINIMUM\_DEPTH* = **2;**}

* implicit modifier is a modifier that the compiler automatically adds to a class interface, method or variable declaration.
* interfaces are not required to define any methods
* interfaces can’t be marked as final

public class FieldMouse implements Climb**,** CanBurrow {  
 public Float getSpeed(int age){  
 return **11f;** }  
}

* A java file may have at most one public top-level class or interface, and it must match the name of the file
* a top-level class or interface can only be declared with public or package-private access

**Inserting Implicit Modifiers**

implicit modifiers for interfaces that you need to know for the exam

* interfaces are assumed to be abstract
* interface variables are assumed to be public, static, and final
* interface methods without a body are assumed to be abstract and public

both these are equivalent:

public interface Soar {  
 int *MAX\_HEIGHT* = **10;** final static boolean *UNDERATER* = true**;** void fly(int speed)**;** abstract void takeoff()**;** public abstract double dive()**;**}  
  
public abstract interface Soar {  
 public static int *MAX\_HEIGHT* = **10;** public final static boolean *UNDERATER* = true**;** public abstract void fly(int speed)**;** public abstract void takeoff()**;** public abstract double dive()**;**}

**Conflicting Modifiers**

when working with class members, omitting the access modifier indicates default (package-private) access. When working with interface members, though, the lack of access modifier always indicates public access.

when a member access is less than the class then the line will not compile.

Also on the exam you are tested on a interface that cant be final because it has a implicit abstract keyword.

**Differences between Interfaces and Abstract Classes**

abstract class Husky{  
 abstract void play()**;**}  
  
interface Poodle{  
 void play()**;**}

* both considered abstract types
  + Husky wont compile if play() is not marked abstract
  + the method in the Poodle interface will compile with or without the abstract modifier.
* method access modifier on the methods inside
  + the play() method in Husky class is considered default (package-private),
  + whereas the method in the Poodle interface is assumed to be public
* class Webby extends Husky{  
   void play() {  
     
   }  
  }  
    
  class Georgette implements Poodle{  
   public void play() { // have to specify public modifier  
     
   }  
  }

**Inheriting an Interface**

An interface can be inherited in one of three ways:

1. an interface can extend another interface
2. a class can implement an interface
3. a class can extend another class whose ancestor implements an interface

* when an interface is inherited, all the abstract methods are inherited
* If the type inheriting the interface is also abstract, such as an interface or abstract class, it is not required to implement the interface methods.
* the first concrete subclass that inherits the interface must implement all the inherited abstract methods

**Mixing Class and Interface Keywords**

* a class can implement an interface and extend another class
* an interface can extend another interface, an interface cannot implement another interface

**Duplicate Interface Method Declarations**

Java allows for multiple inheritance via interfaces, what will happen if you define a class that inherits from two interfaces that contain the same abstract method.

* in a scenario where both signatures for the two interface methods are duplicates and have identical method declarations and thus are compatible
  + means that the compiler can resolve the differences between the two declarations without finding any conflicts
  + in this case you just need to be able to create a single method that overrides both inherited abstract methods at the same time
* different signatures:
  + method name is the same, but the input parameters are different, there is no conflict because this considered a method overload
* duplicate methods have the same signature but different return types:
  + need to review the rules for overriding methods
    - if return types are covariant, they can still compile (String <-> CharSequence)
    - if return types are not covariant (eg. int / bool) then does not compile
* the compiler will also throw an exception if you define an abstract class or interface that inherits from two conflicting abstract types.

### Polymorphism and Interfaces

**Abstract Reference Types**

….List<String> animals….

at no point is this class interested in what the actual underlying object for animals is. it might be an ArrayList, which you have seen before, but it may also be a LinkedList or a Vector.

**Casting Interfaces**

interface Canine{}  
  
class Dog implements Canine{}  
class Wolf implements Canine{}  
  
public class BadCasts{  
 public static void main(String[] args){  
 Canine canine = new Wolf()**;** Canine badDog = (Dog)canine**; // throws ClassCastException** }  
}

* java can’t be sure which specific class type the canine instance, therefore, it allows the invalid cast to the Dog reference type, even though Dog and Wolf are not related. The code compiles but throws a ClassCastException at runtime.
* the compiler does not allow a cast from an interface reference to an object reference if the object type does not implement the interface.

### Reviewing Interface Rules

**Interface Definition Rules**

1. Interfaces cannot be instantiated
2. All top-level types, including interfaces, cannot be marked protected or private
3. Interfaces are assumed to be abstract and cannot be marked final
4. Interfaces may include zero or more abstract methods
5. An interface can extend any number of interfaces
6. an interface reference may be cast to any reference that inherits the interface, although this may produce an exception at runtime if the classes aren’t related.
7. The Compiler will only report an unrelated type error for an instanceof operation with an interface on the right side if the reference on the left side is a final class that does not inherit the interface
8. An interface method with a body must be marked default, private, static, or private static

**Abstract Interface Method Rules**

1. Abstract methods can be defined only in Abstract classes or interfaces
2. abstract methods cannot be declared private or final
3. abstract methods must not provide a method body/implementation in the abstract class in which is it declared
4. implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declaration, etc.
5. Interface methods without a body are assumed to be abstract and public

**Interface variables Rules**

1. Interface variables are assumed to be public, static, and final
2. Because interface variables are marked final, they must be initialized with a value when they are declared.

## Introducing Inner Classes

**Defining a Member Inner Class**

a member inner class is a class defined at the member level of a class (the same level as the methods, instance variables, and constructors). It is the opposite of a top-level class, in that it cannot be declared unless it is inside another class.

* often define a member inner class inside another class if the relationship between the two classes is very close.
* while top-level classes and interfaces can only be set with public or package-private access, member inner classes do not have the same restriction. A member inner class can be declared with all the same access modifiers as a class member, such as public protected, default (package-private), or private.
* a member inner class can contain many of the same methos and variables as a top-level class
  + some members are disallowed in member inner classes, such as static members

public class Zoo {  
 private interface Paper{  
 public String getId()**;** }  
 public class Ticket implements Paper {  
 private String serialNumber**;** public String getId(){  
 return serialNumber**;** }  
 }  
}

**Using a Member Inner Class**

one of the ways a member inner class can be used is by calling it in the outer class.

* the advantage of using a member inner class in this example is that the zoo class completely manages the lifecycle of the Ticket class.

public class Zoo {  
 public static void main(String[] args) {  
 var z = new Zoo()**;** var t = z.sellicket("12345")**;** System.*out*.println(t.getId() + " Ticket sold!")**;** }  
 private interface Paper{  
 public String getId()**;** }  
 public class Ticket implements Paper {  
 private String serialNumber**;** public String getId(){  
 return serialNumber**;** }  
 }  
 public Ticket sellicket(String serialNumber){  
 var t = new Ticket()**;** t.serialNumber = serialNumber**;** return t**;** }  
}

## Exam Essentials

* Be able to write code that creates and extends abstract classes
* be able to write code that creates, extends, and implements interfaces
* know the implicit modifiers that the compiler will automatically apply to an interface
* distinguish between top level and inner classes/interfaces and know which access modifiers are allowed

# Chapter 10 (Exceptions)

## Understanding Exceptions

**The Role of Exceptions**

exception -> is Java’s way of saying “I give up. I don’t know what to do right now. you deal with it”

two approaches java uses to handle exceptions

1. a method can handle the exception case itself
2. or make it the caller’s responsibility

**Understanding Exception Types**

Java has a Throwable superclass for all objects that represent these events.

Diagram

Description automatically generated

**Checked Exceptions**

checked exception -> is an exception that must be declared or handled by the application code where it is thrown.

* checked exceptions all inherit exception but not RuntimeException.
* checked exceptions tend to be more anticipated- for example, trying to read a file that doesn’t exist.
* checked exceptions also include any class that inherits throwable, but not Error or RuntimeException. for example, a class that directly extends Throwable would be a checked exception.
* java has a rule called the handle or declare rule, means that all checked exceptions that could be thrown within a method are either wrapped in compatible try and catch blocks or declared in the method signature. Checked exceptions MUST handle or declare exception

void fall(int distance) throws IOException{  
 if(distance > **10**){  
 throw new IOException()**;** }  
}

void fall(int distance){  
 try{  
 if(distance > **10**){  
 throw new IOException()**;** }  
 } catch (Exception e){  
 e.printStackTrace()**;** }  
}

Notice that the catch statement uses Exception, not IOException, not IOException. Since IOException is a subclass of Exception, the catch block is allowed to catch it.

**Unchecked Exceptions**

unchecked exception -> is any exception that does not need to be declared or handled by the application code where it is thrown.

unchecked exceptions are often referred to as runtime exceptions, although in Java, unchecked exceptions include any class that inherits runtime exceptions.

runtime exception -> is defined as the RuntimeException class and its subclasses.

* Runtime exception tend to be unexpected but not necessarily fatal. ex. accessing an invalid array index is unexpected, even though they do inherit the exception class, they are not checked exception.

**Throwing an Exception**

two ways Java can throw an exception

1. ArrayIndexOutOfBoundsException
2. explicitly request Java to throw one
   1. throw new Exception();
   2. throw new Exception(“Ow! I fell”);
   3. throw new RuntimeException();
   4. throw new RuntimeException(“Ow! I fell”);

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **How to recognize** | **Okay for program to catch** | **Is program required to handle or declare?** |
| Runtime exception | Subclass of RuntimeException | Yes | No |
| Checked exception | Subclass of Exception but not subclass of RuntimeException | Yes | Yes |
| Error | Subclass of Error | No | No |

## Recognizing Exception Classes

**RuntimeException Classes**

|  |  |
| --- | --- |
| ArithmeticException | Thrown when code attempts to divide by zero. |
| ArrayIndexOutOfBoundsException | thrown when code uses an illegal index to access an array |
| ClassCastException | thrown when an attempt is made to cast an object to a class of which it is not an instance |
| NullPointerException | thrown when there is a null reference where an object is required |
| IllegalArgumentException | thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument |
| NumberFormatException | Subclass of IllegalArgumentException thrown when an attempt is made to convert a string to a numeric type but the string doesn’t have an appropriate format. |

**Checked Exception Classes**

|  |  |
| --- | --- |
| IOException | Thrown programmatically when there’s a problem reading or writing a file |
| FileNotFoundException | Subclass of IOException thrown programmatically when code tries to reference a file that does not exist. |

**Error Classes**

|  |  |
| --- | --- |
| ExceptionInInitializerError | thrown when a static initializer throws an exception and doesn’t handle it |
| StackOverflowError | thrown when a method calls itself too many times (this is called infinite recursion because the method typically calls itself without end.) |
| NoClassDefFoundError | Thrown when a class that the code uses is available at compile time but not runtime |

## Handling Exceptions

**Using try and catch Statements**

**Syntax:**

try{

// protected code

} catch (exception type identifier) {

//exception handler

}

void fall() throws RuntimeException{  
 throw new RuntimeException()**;**}  
  
void explore() {  
 try{  
 fall()**;** System.*out*.println("Never get here")**;** } catch (RuntimeException ex){  
 getUp()**;** }  
 seeAnimals()**;**}

**Chaining catch Blocks**

void vistPorcupine() {  
 try{  
 seeAnimals()**;** } catch(AnimalsOutForAWalk e) { // first catch block  
 System.*out*.println("try back later")**;** } catch(ExhibitClosed e){ // second catch block  
 System.*out*.println("not today")**;** }  
}

* chaining catch blocks needs to be carefully ordered so that all code is reachable

**Applying a Multi-catch Block**

public static void main(String[] args) {  
 try{  
 System.*out*.println(Integer.*parseInt*(args[**0**]))**;** } catch (ArrayIndexOutOfBoundsException | NumberFormatException e){  
 System.*out*.println("missing or invalid input")**;** }  
}

* the exceptions grouped in one catch should not be subclasses of each other.

**Adding a finally Block**

try{

// protected code

} catch (exception type identifier) {

//exception handler

} finally { // block always executes whether r not an exception occurs

// finally block

}

**Finally Closing Resources**

* often used to with read/write of resources, finally block used to close all resources.

void readFile(String file) {  
 FileInputStream is = null**;** try{  
 is = new FileInputStream("myfile.txt")**;** // Read file data  
 } catch (IOException e) {  
 e.printStackTrace()**;** } finally {  
 if(is != null){  
 try {  
 is.close()**;** } catch ( IOException e2){  
 e2.printStackTrace()**;** }  
 }  
 }  
}

**Basics of Try-with-Resources**

* one or more resources can be opened in the try clause
* when there are multiple resources opened, they are closed in the reverse order from which they were created.
* parentheses are used to list those resources and semicolons are used to separate the declarations
* catch block is optional with a try-with-resources statement.

try (FileInputStream in = new FileInputStream(“data.txt”); FileOutputStream out = new FileOutputStream(“output.txt”);){

// Protected code

}

void readFile(String file) {  
 try (FileInputStream is = new FileInputStream("myfile.txt")){  
 // read file data  
 }   
}

|  |  |  |  |
| --- | --- | --- | --- |
| **try-statement** | **0 finally blocks** | **1 finally block** | **2 or more finally blocks** |
| 0 catch blocks | Not Legal | legal | not legal |
| 1 or more catch blocks | legal | legal | not legal |
| **try-with-resources** |  |  |  |
| 0 catch blocks | legal | legal | not legal |
| 1 or more catch blocks | legal | legal | not legal |

**Declaring Resources**

* a try-with -resources statement does not support multiple variable declarations

**Scope of Try-with-Resources**

* the resources created in the try clause in the try clause are in scope only within the try block
* this is another way to remember that the implicit finally runs before any catch / finally blocks that you code yourself

**Following Order of Operation**

Two rules for the order in which code runs in a try with resources statement

* resources are closed after the try clause ends and before any catch /finally clauses
* resources are closed in the reverse order from which they were created.

public static void main(String... xyz) {  
 try (MyFileClass a1 = new MyFileClass(**1**)**;** MyFileClass a2 = new MyFileClass(**2**)){  
 throw new RuntimeException()**;** } catch (Exception e){  
 System.*out*.println("ex")**;** } finally {  
 System.*out*.println("finally")**;** }   
}

Closing:2

Closing:1

ex

finally

**Throwing Additional Exceptions**

* a catch or finally block can have any valid java code in it, including another try statement
* the last exception thrown is the one that is important.

**Calling Methods That Throw Exceptions**

* Checked exceptions needs to be handled or declared
* the compiler is still on the lookout for unreachable code
* when calling a method that throws an exception you need to handle or declare that exception

**Declaring and Overriding Methods with Exceptions**

* when overriding a method from a super class or implements a method from an interface, it’s not allowed to add new checked exceptions to the method signature
  + no subclass implementation of a method can has additional exceptions than the super class/ interface
* an overridden method in a subclass is allowed to declare fewer exceptions than the superclass or interface
* a class is allowed to declare a subclass of an exception type, the superclass or interface has already taken care of a broader type.
* this applies only to checked exceptions that needs to be handled or declared.

**Printing an Exception**

three ways to print an exception:

1. let java print it out,
2. print just the message
3. print where the stack trace comes from

Example:

public static void main(String... xyz) {  
 try {  
 hop()**;** } catch (Exception e){  
 System.*out*.println(e)**; // print it out** System.*out*.println(e.getMessage())**; // print the message** e.printStackTrace()**; // print the stack trace** }  
}

**Exam Essentials**

* Understand the various types of exceptions
* Differentiate between checked and unchecked exceptions
* Understand the flow of a try statement
* Be able to follow the order of a try-with-resources statement
* Identify whether an exception is thrown by the programmer or the JVM
* Write methods that declare exceptions
* Recognize when to use throw versus throws

# Chapter 11 (Modules)

## Introducing Modules

JAR -> a JAR is a zip file with some extra information, and the extension is .jar

Open Source -> is software with the code supplied and is often free to use

JPMS -> Java Platform Module System group code a higher level and tries to solve the problems that java has.

JPMS includes:

* A format for module JAR files
* Partitioning of the JDK into modules
* Additional command-line options for Java tools

**Benefits of Modules**

* Better Access Control
  + Modules solve the problem of code only accessible to specific packages in a project by acting as a fifth level of access control
* Clearer Dependency Management
* Custom Java Builds
* Improved Performance
* Unique Package Enforcement

## Creating and Running a Modular Program

**Creating the Files**

Directory: (Modules Project Example)

* care
  + zoo.animal.care
    - details
      * HippoBirthday.java
    - medical
      * Diet.java
  + module-info.java
* feeding
  + zoo.animal.feeding
    - Task.java
  + module-info.java
* modules
  + zoo.animal.care.jar
  + zoo.animal.feeding.jar
  + zoo.animal.staff.jar
  + zoo.animal.talks.jar
* staff
  + zoo.animal.staff
    - Jobs.java
  + module-info.java
* talks
  + zoo.animal.talks
    - content
      * ElephantScript.java
      * SeaLionScript.java
    - media
      * Announcement.java
      * Signage.java
    - schedule
      * Weekend.java
      * Weekday.java
  + module-info.java

**key differences between a module-info file and a regular java class:**

* the module-info file must be in the root directory of your module.
  + regular java classes should be in packages
* the module-info file must use the keyword module instead of class, interface or enum
* the module name follows the naming rules for package names. it often includes periods. in its name.
  + regular class and package names are not allowed to have dashes. module names follow the same rule

**Compiling Our First Module**

|  |  |  |
| --- | --- | --- |
| **Use for** | **Abbreviation** | **Long form** |
| Directory for class files | -d <dir> | n/a |
| Module path | -p <path> | --module-path <path> |

example:

**classes:**

package zoo.animal.feeding;

public class Task{

public static void main(String[] args) {

System.out.println("All fed!");

}

}

module zoo.animal.feeding {}

javac --module-path mods -d feeding .\feeding\zoo\animal\feeding\\*.java .\feeding\module-info.java

**Running our First Module**

java --module-path feeding --module zoo.animal.feeding/zoo.animal.feeding.Task

**Packaging Our First Module**

jar -cvf .\mods\zoo.animal.feeding.jar -C .\feeding\ . (run with no module-info.class file)

java -p mods -m zoo.animal.feeding/zoo.animal.feeding.Task ( running jar module )

## Updating Our Example for Multiple Modules

**Updating the Feeding Module**

exports keyword is used to indicate that a module intends for those packages to be used by Java code outside the module.

------------

module zoo.animal.feeding {

exports zoo.animal.feeding;

}

------------

package zoo.animal.feeding;

public class Task{

public static void main(String[] args) {

System.out.println("All fed!");

}

}

Commands:

* javac -p modules -d feeding .\feeding\zoo\animal\feeding\\*.java .\feeding\module-info.java
* java -p modules -m zoo.animal.feeding/zoo.animal.feeding.Task
* jar -cvf .\modules\zoo.animal.feeding.jar -C .\feeding\ .

**Creating a Care Module**

module zoo.animal.care {

exports zoo.animal.care.medical;

requires zoo.animal.feeding;

}

--------------

package zoo.animal.care.medical;

public class Diet { }

-------------

package zoo.animal.care.details;

import zoo.animal.feeding.\*;

public class HippoBirthday {

private Task task;

}

--------------

Commands:

* javac -p modules -d care .\care\zoo\animal\care\details\\*.java .\care\zoo\animal\care\medical\\*.java .\care\module-info.java
* jar -cvf modules/zoo.animal.care.jar -C .\care\ .

**Creating the Talks Module**

module zoo.animal.talks {

exports zoo.animal.talks.content to zoo.staff;

exports zoo.animal.talks.media;

exports zoo.animal.talks.schedule;

requires zoo.animal.feeding;

requires zoo.animal.care;

}

---------------

package zoo.animal.talks.schedule;

public class Weekend {}

--------------

package zoo.animal.talks.schedule;

public class Weekday {}

---------------

package zoo.animal.talks.media;

public class Signage {}

---------------

package zoo.animal.talks.media;

public class Announcement {

public static void main(String[] args){

System.out.println("We weill be having talks");

}

}

-------------

package zoo.animal.talks.content;

public class SeaLionScript{}

------------

package zoo.animal.talks.content;

public class ElephantScript {}

Commands:

* javac -p modules -d .\talks\ .\talks\zoo\animal\talks\content\\*.java .\talks\zoo\animal\talks\media\\*.java .\talks\zoo\animal\talks\schedule\\*.java .\talks\module-info.java
* jar -cvf .\modules\zoo.animal.talks.jar -C .\talks\ .

**Creating the Staff Module**

module zoo.staff {

requires zoo.animal.feeding;

requires zoo.animal.care;

requires zoo.animal.talks;

}

-----------

package zoo.staff;

public class Jobs {}

Commands:

* javac -p modules -d staff .\staff\zoo\staff\Jobs.java .\staff\module-info.java
* jar -cvf .\modules\zoo.animal.staff.jar -C .\staff\ .

## Diving into the module-info File

**Exports**

“ exports zoo.animal.talks.content **to zoo.staff**;”

Exporting a module member to only specific packages specifying in the exports keyword within the module-info file.

**Access Control with Modules**

|  |  |  |
| --- | --- | --- |
| **level** | **Within module code** | **Outside module** |
| private | Available only within class | No access |
| default (package-private) | Available only within package | No access |
| protected | Available only within package or to subclasses | Accessible to subclasses only if package is exported |
| public | Available to all classes | Accessible only f package is exported |

**Requires transitive**

requires transitive zoo.animal.feeding;

transitive -> that any module that requires this module will also depend on moduleName

if moduleX depends on moduleY and moduleZ on moduleX, then moduleZ has a transitive relationship to moduleY.

module zoo.animal.feeding {

exports zoo.animal.feeding;

}

module zoo.animal.care {

exports zoo.animal.care.medical;

requires transitive zoo.animal.feeding;

}

module zoo.animal.talks {

exports zoo.animal.talks.content to zoo.staff;

exports zoo.animal.talks.media;

exports zoo.animal.talks.schedule;

requires transitive zoo.animal.care;

}

module zoo.staff {

requires zoo.animal.talks;

}

**Effects of requires transitive**

* module zoo.animal.talks can optionally declare it requires the zoo.animal.feeding module, but it is not required
* module zoo.animal.care cannot be compiled or executed without access to the zoo.animal.feeding module.
* Module zoo.animal.talks cannot be compiled or executed without access to the zoo.animal.feeding module.

**Duplicate requires statements**

* The exam tries to trick you by mixing requires and requires transitive together but having them together doesn’t compiled
* Java doesn’t allow you to repeat the same module in a requires clause.
* It is redundant and most like an error in coding.
* Requires transitive is like requires plus some extra behaviour.

**Provides, uses, and opens**

provides -> specifies that a class provides an implementation of a service

-> to use it you supply the API and class name that implements the API:

-> provides zoo.staff.ZooApi with zoo.staff.ZooImpl

uses -> specifies that a module is relying on a service

-> to code it you supply the API you want to call

-> uses zoo.staff.ZooApi

opens -> Java allows callers to inspect and call code at runtime with a technique called reflection.

-> It can be used to subvert access control!

-> the module system requires developers to explicitly allow reflection in the module-info if . they want calling modules to be allowed to use it.

-> opens zoo.animal.talks.schedule;

opens zoo.animal.talks.media.to zoo.staff;

## Discovering Modules

**the Java Command**

the java command has three module-related options

1. Describes a module
2. Lists the available modules
3. Shows the module resolution logic

**Describing a module**

Suppose you are given the zoo.animal.feeding module JAR file and want to know about its module structure. You could unjar it and open the module-info file and see what is exports.

However the java command has an option to describe a module in two different but equivalent ways

1. java -p mods -d zoo.animal.feeding
2. java -p mods --describe-module zoo.animal.feeding

This prints:

zoo.animal.feeding file: ///absolutePath/mods/zoo.animal.feeding.jar

exports zoo.animal.feeding

requires java.base mandated (// this is automatically added)

**Listing Available Modules**

List the modules that are available, the simplest form lists the modules that are part of the JDK:

* java --list-modules
* java -p mods –list-modules
  + gives the lists of all jar files in folder mods

**Showing Module Resolution**

It spits out a lot of output when the program starts up then it runs the program

java --show-module-resolution -p feeding -m zoo.animal.feeding/zoo.animal.feeding.Task

it starts out by listing the root module then following packages included. Then lists modules that have dependencies. Finally outputs the result of the program All fed!...

**The jar Command**

The jar command can describe a module. Bot of theses commands are equivalent:

* jar -f mods/zoo.animal.feeding.jar -d
* jar --file mods/zoo.animal.feeding.jar --describe-module

the jar command includes the module-info.class as part of the description

**The jdeps Command**

* The jdeps command gives you information about dependencies within a module
* Unlike describing a module, it looks at the code in addition to the module-info file
* This tells you what dependencies are actually used rather than simply declared
  + jdeps -s mods/zoo.animal.feeding.jar
  + jdeps -summary mods/zoo.animal.feeding.jar
    - -s / -summary gives a summary of the dependencies where if omitted then it gives output in long format
  + Output has 3 sections
    - Filename and required dependencies
    - Summary showing the two module dependencies with an arrow
    - Package-level dependencies

**The jmod Command**

* JMOD files are recommended only when you have native libraries or something that can’t go inside a JAR file.
* Jmod is only for working with the JMOD files
* Below is a table of the most common used modes that you need to be aware of for the exam

**Modes using jmod**

|  |  |
| --- | --- |
| operation | Description |
| create | Creates a JMOD file |
| extract | Extracts all files from the JMOD. Works like unzipping |
| describe | Prints the module details such as requires |
| list | Lists all files in the JMOD file |
| hash | Shows a long string that goes with the file |

## Reviewing Command-Line Options

|  |  |
| --- | --- |
| **Description** | **Syntax** |
| Compile nonmodular code | **javac** **-cp** classpath -d directory classesToCompile  **javac** **--class- classpath** -d directory classesToCompile  **javac** **-classpath** classpath -d directory classesToCompile |
| Run nonmodular code | **java** **-cp** classpath package.className  **java** **-classpath** classpath package.className  **java** **--class-path** classpath package.className |
| Compile a module | **javac** **-p** moduleFolderName -d directory classesToCompileIncludingModuleInfo  **javac** **--module-path** moduleFolderName -d directory classesToCompileIncludingModuleInfo |
| Run a module | **java** **-p** moduleFolderName **-m** moduleName/package.className  **java** **--module-path** moduelFolderName **--module** moduleName/package.className |
| Describe a module | **java** **-p** moduleFolderName **-d** moduleName  **java** **--module-path** moduleFolderName **--describe-module** moduleName  **jar** --file jarName **--describe-module**  **jar** -f jarName **-d** |
| List available modules | **java** **--module-path** moduleFolderName **--list-modules**  **java** **-p** moduleFolderName **--list-modules**  **java** **--list-modules** |
| View dependencies | **jdeps** **-summary --module-path** moduleFolderName jarName  **jdeps** **-s --module-path** moduleFolderName jarName |
| Show module resolution | **java** **--show-module-resolution -p** moduleFolderName **-m** moduleName  **java** **--show-module-resolution --module-path** moduleFolderName **--module** moduleName |

### Options you need to know for the exam

**javac**

|  |  |
| --- | --- |
| **Options** | **Description** |
| -cp <classpath>  -classpath <classpath> --class-path <classpath> | Location of JARs in a nonmodular program |
| -d <dir> | Directory to place generated class files |
| -p <path> --module-path <path> | Location of JARs in a modular program |

**java**

|  |  |
| --- | --- |
| **Options** | **Description** |
| -p <path> --module-path <path> | Location of JARs in a modular program |
| -m <name> --module <name> | Module name to run |
| -d --describe-module | Describes the details of a module |
| --list-modules | Lists observable modules without running a program |
| --show-module-resolution | Shows modules when running program |

**jar**

|  |  |
| --- | --- |
| **Options** | **Description** |
| -c --create | Create a new JAR file |
| -v --verbose | Prints details when working with JAR files |
| -f --file | JAR filename |
| -C | Directory containing files to be used to create the JAR |
| -d --describe-module | Describes the details of a module |

**jdeps**

|  |  |
| --- | --- |
| **Options** | **Description** |
| --module-path <path> | Location of JARs in a modular program |
| -s -summary | Summarizes output |

## Exam Essentials

* Identify benefits of the Java Platform Module Systems
* Use command-line syntax with modules
* create basic module-info files
* Identify advanced module-info keywords
* display information about modules

# Chapter 12 (Java Fundamentals)

## Applying the final Modifier

**Declaring final Local Variable**

* we don’t need to assign a value when a final variable is declared.
* The rule is only that it must be assigned a value before it can be used
* once assigned a value it can not be assigned a new one
* to remember a final stringbuilder type can still be manipulated since its mutable where a string or a primitive type can be final and not be changed.

**Adding final to Instance and static Variables**

* Instance and static class variables can also be marked final
  + When instance variable is marked as final then it needs to be assigned a value when declared or when the object is initiated in initialize blocks or in the constructor
  + The rules for static final variables are similar to instance final variables, except they do NOT use static constructors (there is no such thing!). and use static initializers instead of instance initializers.

**Writing final methods**

* Methods marked final cannot be overridden by a subclass.
* This essentially prevents any polymorphic behaviour on the method call and ensures that a specific version of the method is always called.
* Remember that methods can be assigned an abstract or final modifier,
  + Abstract method is one that does not define a method body and can appear only inside and abstract class or interface.
  + A final method is on that cannot be overridden by a subclass
  + A method CAN NOT be final and abstract this gives compile error.

**Marking classes final**

* The final modifier can be applied to class declarations as well.
* A final class is one that cannot be extended
  + If tried it gives compile error
* Classes can’t be marked abstract and final either
* As well as it is not possible to write an interface as final this does not compile because an interface needs to be overridden and hence final won’t be possible

## Working with Enums

**Creating Simple Enums**

Us the enum keyword instead of the class or interface keyword and list all of the valid types for that enum example:

Public enum Season {

WINTER, SPRING, SUMMER, FALL

}

* Behind the scenes an enum is a type of class that mainly contains static members
* It also includes some helper methods like name().

Season s = Season.SUMMER;

System.out.println(Season.SUMMER); // SUMMER

System.out.println(s == Season.SUMMER); // true

* Enums print the name of the enum when toString() is called
* They can be compared using == because they are like static final constants, thus you can use both == and .equals() to compare enums
* An enum provides a values() method to get an array of all the values.
* You can use this like any normal array, including in an enhanced for loop often called for-each loop.

for(Season season: Season.values()) {

System.out.println(season.name() + “ “ + season.ordinal());

}

Prints:

WINTER 0

SPRING 1

SUMMER 2

FALL 3

* Another useful feature is retrieving an enum value from a String using the valueOf() method
  + This is helpful when working with older code,
  + The string passed in must match the enum value exactly though
    - Season s = Season.valueOf(“SUMMER”); // SUMMER
    - Season t = Season.valueOf(“summer”); // throws an exception at runtime
      * Throws IllegalArgumentException
  + You can’t extend an enum will give an compile error.

**Using Enums in Switch Statements**

Enums can be used in switch statements example:

Season summer = Season.SUMMER;

switch( summer){

case WINTER:

System.out.println();

break;

case SUMMER:

System.out.println();

break;

default:

System.out.println();

}

**Adding Constructors, Fields, and Methods**

Enums can have morein them than just a list of values.

Example:

public enum Season {

WINTER(“Low”), SPRING(“Medium”), SUMMER(“High”), FALL(“Medium”);

// list of enums end with a ;

private final String expectedVisitors;

private Season(String expectedVIsitors) {

this.expectedVisitors = expectedVisitors;

}

Public void printExpectedVisitors() {

System.out.println(expectedVisitors);

}

}

* All enum constructors are implicitly private, with the modifier being optional.
* an enum constructor will not compile if it contains a public or protected modifier.
  + The private constructor will be called as follow:
    - Season.SUMMER.printExpectedVisitors();
* First time that we ask for any of the enum values, java constrracts all of the enum values, after that java just returns the already constructed enum values.
* Thus

public enum OnlyOne {

ONCE(true);

Private OnlyOne(booleanb) {

System.out.println(“constructing,”);

}

}

public class PrintTheOne {

public static void main(String[] args) {

System.out.print(“begin”);

OnlyOne firstCall == OnlyOne.ONCE; // prints constructing,

OnlyOne secondCall = OnlyOne.Once; // doesn’t print anything

System.out.print(“end”);

}

}

This class prints the following:

begin, constructing, end

Practical example:

public enum Season {

WINTER {

Public String getHours() { return “10am-3pm”;}

},

SUMMER {

Public String getHours() { return “9am-7pm”;}

},

SPRING, FALL;

public String getHours() { return “9am-5pm”;}

}

## Creating Nested Classes

A nested class is a class that is defined within another class. A nested class can come in one of four flavours.

1. Inner class: a non-static type defined at the member level of a class
2. Static class: a static type defined at the member level of a class
3. Local class: a class defined within a method body
4. Anonymous class: a special case of a local class that does not have a name

Benefits of using nested classes

* They can encapsulate helper classes by restricting them to the containing class.
* They can make it easy to create a class that will be used in only one place.
* They can make the code cleaner and easier to read.

### Declaring an inner Class

Inner class, also called a member inner class, is a non-static type defined at the member level of a class (the same level as the methods, instance variables, and constructors). Inner classes have the following properties:

* Can be declared public, protected, package-private (default), or private
* Can extend any class and implement interfaces
* Can be marked abstract or final
* Cannot declare static fields or methods, except for static final fields
* Can access members of the outer class including private members
  + This means that the inner class can access variables in the outer class without doing anything special

Example:

public class Outer {

private String greeting = “Hi”;

protected class Inner {

public int repeat = 3;

public void go() {

for (int I = 0; I < repeat; i++)

System.out.println(greeting);

}

}

public void callInner() {

Inner inner = new Inner();

inner.go();

}

public static void main(String[] args) {

Outer outer = new Outer();

outer.callInner();

}

}

**POSSIBLE FOR THE EXAM NOT FOR PRACTICAL USE**

* Multiple nested classes and access a variable with the same name in each:

public class A {

private int x = 10;

class B {

private int x = 20;

class C {

private int x = 30;

public void allTheX() {

System.out.println(x); //30

System.out.println(this.x); //30

System.out.println(B.this.x); //20

System.out.println(A.this.x); //10

}

}

}

public static void main(String[] args) {

A a = new A();

A.B b = a.new B();

A.B.C c = b.new C();

c.allTheX();

}

}

### Creating a static Nested Class

* A static nested class is a static type defined at the member level.
* Unlike an inner class, a static nested class can be instantiated without an instance of the enclosing class.
* Trade-off is that it can’t access instance variables or methods in the outer class directly. It can be done but requires an explicit reference to an outer class variable.
* Its like the top-level class except for the following:
  + The nesting creates a namespace because the enclosing class name must be used to refer to it.
  + It can be made private or use one of the other access modifiers to encapsulate it
  + The enclosing class can refer to the fields and methods of the static nested class.
* Example

public class Enclosing {

static class Nested {

private int price = 6;

}

public static void main(String[] args) {

Nested nested = new Nested();

System.out.println(nested.price);

}

}

### Writing a Local Class

A local class is a nested class defined within a method. Like local variables, a local class declaration does not exist until the method is invoked, and it goes out of scope when the method returns.

Local classes have the following properties:

* They do not have an access modifier
* They cannot be declared static and cannot declare static fields or methods, except for static final fields
* They have access to all fields and methods of the enclosing class ( when defined in an instance method)
* They can access local variables if the variables are final or effectively final.

### Defining an Anonymous Class

An anonymous class is a specialized form of a local class of a local class that does not have a name. It is declared an instantiated all in one statement using the new keyword, a type name with parentheses, and a set of braces {}.

Anonymous classes are required to extend an existing class or implement an existing interface.

They are useful when you have a short implementation that will not be used anywhere else.

Example:

public class ZooGiftShop{

abstract class SaleTodayOnly {

abstract int dollarsOff();

}

Public int admission(int basePrice) {

SaleTodayOnly sale = **new SaleTodyOnly() {**

**int dollarsOff() { return 3; }**

**};**// don’t forget the semicolon!

return basePrice – sale.dollarsOff();

}

}

Same example just using an interface

Example:

public class ZooGiftShop{

interface SaleTodayOnly {

abstract int dollarsOff();

}

Public int admission(int basePrice) {

SaleTodayOnly sale = **new SaleTodyOnly() {**

**int dollarsOff() { return 3; }**

**};**// don’t forget the semicolon!

return basePrice – sale.dollarsOff();

}

}

* Something to note is that an anonymous class can also be used even as an argument to another method
* Can be used very similar as an lambda expression \*

**Reviewing Nested Classes**

Modifiers in nested classes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Permitted Modifiers** | **Inner class** | **static nested class** | **Local class** | **Anonymous class** |
| Access modifiers | All | All | None | None |
| abstract | Yes | Yes | Yes | No |
| Final | Yes | Yes | Yes | No |

Members in nested classes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Permitted Members** | **Inner class** | **static nested class** | **Local class** | **Anonymous class** |
| Instance methods | yes | yes | yes | yes |
| Instance variables | yes | yes | yes | yes |
| static methods | no | yes | no | no |
| static variables | yes(if final) | yes | yes (if final) | yes (if final) |

Nested class access rules

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Inner class | static nested class | Local class | Anonymous class |
| Can extend any class or implement any number of interfaces | Yes | Yes | Yes | No- must have exactly one superclass or one interface |
| Can access instance members of enclosing class without a reference | Yes | No | Yes (if declared in an instance method) | Yes (if declared in an instance method) |
| Can access local variables of enclosing method | N/A | N/A | Yes (if final or effectively final) | Yes (if final or effectively final) |

## Understanding Interface Members

Interface member types

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Since Java version | Membership type | Required modifiers | Implicit modifiers | Has value or body? |
| Constant variable | 1.0 | Class | - | public static final | Yes |
| Abstract method | 1.0 | Instance | - | public abstract | No |
| Default method | 8 | Instance | default | public | Yes |
| Static method | 8 | Class | static | public | Yes |
| Private method | 9 | Instance | private | - | Yes |
| private static method | 9 | Class | private static | - | Yes |

**Relying on a default interface method**

default method -> is a method defined in an interface with the default keyword and includes a method body.

contrast default methods with abstract methods in an interface which do not define a method body.

* a default method may be overridden by a class implementing the interface.
* the name default comes from the concept that it is viewed as an abstract interface method with a default implementation.

**Default Interface Method definition rules:**

1. a default method may be declared only within an interface
2. a default method must be marked with the default keyword and include a method body
3. a default method is assumed to be public
4. a default method cannot be marked abstract, final or static
5. a default method may be overridden by a class that implements the interface
6. if a class inherits two or more default methods with the same method signature, then the class must override the method.

**Inheriting duplicate default methods**

* Java will have a compile error when a class implements 2 or more interfaces that has the same default method signatures.
* by overriding the conflicting method, the ambiguity about which version of the method to call has been removed thus
  + having a class implements a default method that exists in more than one implemented interface then you need to override that method in order to resolve the ambiguity of the method signature to resolve the compile error

**Calling a Hidden default Method**

* when an concrete class has an interface with an default method this can be called via an <InterfaceName>.super.<methodname>();

**Using static Interface Methods**

**Static Interface method Definition Rules:**

1. a static method must be marked with the static keyword and include a method body
2. a static method without an access modifier is assumed to be public
3. a static method cannot be marked abstract or final
4. a static method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.

**Introducing private Interface Methods**

rules:

1. a private interface method must be marked with the private modifier and include a method body
2. a private interface method may be called only by default and private (non-static) methods within the interface definition.

**Introducing private static Interface methods**

rules:

1. a private static method must be marked with the private and static modifiers and include a method body
2. a private static interface method may be called only by other methods within the interface definition

* both private and private static methods can be called from default and private methods.
* this is equivalent to how an instance method is able to call both static and instance methods.
* on the other hand a private method cannot be called from a private static method.

**Reviewing Interface Members**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accessible from** default **and** private **methods within the interface definition?** | **Accessible from** static **methods within the interface definition?** | **Accessible from instance methods implementing or extending the interface?** | **Accessible outside the interface without an instance of interface?** |
| Constant variable | Yes | Yes | Yes | Yes |
| abstract method | Yes | No | Yes | No |
| default method | Yes | No | Yes | No |
| private method | Yes | No | No | No |
| static method | Yes | Yes | Yes | Yes |
| private static method | Yes | Yes | No | No |

## Introducing Functional Programming

* a functional interface is an interface that contains a single abstract method
  + Single Abstract Method (SAM)

**Declaring a functional interface with object methods**

* String toString()
* boolean equals(Object)
* int hashCode()

if a functional interface includes an abstract method with the same signature as a public method found in Object, then those methods do not count towards the single abstract method test.

public interface Soar {

abstract String toString();

}

//// THE ABOVE IS NOT AN Functional interface because the toString() is an Object class method

public interface Dive {

String toString();

public boolean equals(Object o);

public abstract int hashCode();

public void dive();

}

/// the above is indeed a functional interface since the tostring, equals and hascode is public methods in Object class, and dive() is the single abstract method.

the exam will try and catch you by playing with these three public Object methods implementation by example using an public boolean equals(Hibernate o) where the equals parameter is not of type Object thus making it not the same as the public Object method implementation.

**Implementing Functional Interfaces with Lambdas**

public interface Predicate<T> {

boolean test(T t);

}

* <T> allows the interface to take an object of a specified type.
* the relationship between functional interfaces and lambda expressions is as follow
  + any functional interface can be implemented as a lambda expression
* Predicate<T> example

**// object class animal**

public class **Animal** {

private String species;

private boolean canHop;

private boolean canSwim;

public Animal(String speciesName, boolean hopper, boolean swimmer) {

species = speciesName;

canHop = hopper;

canSwim = swimmer;

}

public boolean canHop() {return canHop;}

public boolean canSwim() {return canSwim;}

public String toString() {return species;}

}

// traditional search class

import java.util.\*;

import java.util.function.Predicate;

public class **TraditionalSearch** {

public static void main(String[] args) {

//list of animals

var animals = new ArrayList<Animal>();

animals.add(new Animal(“fish”, false, true));

animals.add(new Animal(“kangaroo”, true, true));

animals.add(new Animal(“rabbit”, true, false));

animals.add(new Animal(“turtle”, false, true));

//pass lambda that does check;

**print(animals, a ->a.canHop());**

}

private static void print(List<Animal> animals, **Predicate**<**Animal**> **checker**){

for (Animal animal : animals)

if (**checker**.**test**(**animal**))

System.out.print(animal + “ “);

}

}

**Writing Lambda Expressions**

* syntax of lambda expressions is tricky because many parts are optional. Structure remains the same
* Lambda syntax omitting optional parts
  + **a -> a.canHop()**
* Lambda syntax including optional parts
  + (Animal **a** ) **->** { **return a.canHop()** ; }
* Structure:
  + parameter name (Animal a) / a
  + arrow ->
  + body a.canHop() / { return a.canHop() ; }
* parentheses can be omitted only if there is a single parameter and its type is not explicitly stated.
* as with if-statements we can omit {} when we have only a single statement.
  + when no brackets are used you don’t need to add return keyword or the ‘;’
  + multiple lines /statements then {} and return and ‘;’ needs to be used
  + if return type is void then this is a valid lambda expression
    - s -> {} // in this example the return and ‘;’ is not required since nothing is returned.

**Be able to validate lambda syntax**

**Valid:**

() -> new Duck()

d -> { return d.quack(); }

(Duck d) -> d.quack()

(Animal a, Duck d) -> d.quack()

**invalid / compile error: // DOES NOT COMPILE EXAMPLES**

a, b -> a.startsWith(“test”) //Require parenthesis

Duck d -> d.canQuack(); //Require parenthesis

a -> { a.startsWith(“test”); } //Missing the return keyword

a -> { return a.startsWith(“test”) } //missing the semicolon

(Swan s , t ) -> s.compareTo(t) != 0 //missing the parameter type for t

NOTE: if one parameter has a type all parameters should have a type.

**Working with Lambda Variables Review Chapter 6**

**Restrictions on Using var in the parameter list**

* while you can use var inside a lambda parameter list, there is a rule you need to be aware of. if var is used for one of the types in the parameter list then it must be used for all parameters in the list.
* valid compiles var lambda usage
  + (var num) -> 1
  + (var a, var b) -> “hello”
  + (var b, var k, var m) -> 3.13145
* INVALID COMPILE Errors
  + var w -> 99
  + (var a, Integer b) -> true
  + (String x, var y, Integer z) -> true
  + (var x, y) -> “goodbey”

**Local variables inside the lambda body ( Review in Chapter 6 )**

Code creates a local variable named c that is scoped to the lambda block

(a,b) -> { int c = 5; return c; }

(a,b) -> { int a = 0; return 5; } // DOES NOT COMPILE variable a already exists in scope

**Variables Referenced from the Lambda Body**

Lambda bodies are allowed to use static variables, instance variables, and local variables if they are final or effectively final. (Same rules for access as local and anonymous classes.)

Example

public class Crow {

private String color;

public void caw(String name) {

String volume = “ loudly”

Predicate<String> p = s -> (name+volume+color).length()==10;

}

}

## Exam Essentials

* be able to correctly apply the final modifier
* be able to create and use enum types
* identify and distinguish between types of nested classes
* be able to declare and use nested classes
* be able to create default, static, private and private static interface methods
* Determine whether an interface is a functional interface
* Write simple lambda expressions
* determine whether a variable can be used in a lambda body

# Chapter 13 (Annotations)

## Introducing Annotations

* Annotations are all about metadata
* they allow you to add a lot of value to your code

**Understanding Metadata**

Metadata is data that provides information about other data.

**example:**

imagine our zoo is having a sale on tickets.

The **attribute data** includes the price, the expiration data, and the number of tickets purchased.in other words, the attribute data is the transactional information that makes up the ticket sale and its contents.

the **metadata** includes the rules, properties, or relationships surrounding the ticket sales. Patrons must buy at least one ticket, as a sale of zero or negative tickets are silly. These metadata rules describe information about the tickets, but are not part of the ticket sale.

**Purpose of Annotations**

the purpose of an annotation is to assign metadata attributes to classes, methods, variables, and other Java types.

by using annotations we leave the class structure intact.

Rules:

1. annotations function a lot like interfaces
   1. annotations can be applied to any declaration including classes, methods, expressions, and even other annotations
   2. annotations allow us to pass a set of values where they are applied
2. annotations establish relationships that make it easier to manage data about our application
3. annotations are optional metadata and by themselves do not do anything
   1. this means you can take a project filled with thousands of annotations and remove all of them, and it will still compile and run, albeit with potentially different behaviour at runtime
   2. the opposite is not true, if you add annotations can trigger compiler error. the compiler validates that annotations are properly used and include all required fields.
4. Annotations aren’t utilized where they are defined. It’s up to the rest f the application, or more likely the underlying framework, to enforce or use annotations to accomplish tasks.

## Creating Custom Annotations

* give it a name,
* define a list of optional and required elements,
* specify its usage.

**Creating an Annotation**

* our zoo wants to specify the exercise metadata for various zoo inhabitants using annotations
* we use the @interface (all lowercase) annotation to declare an annotation.
* like classes and interfaces, they are commonly defined in their own file as a top-level type, although they can be defined inside a class declaration like an inner class.

public @interface Exercise{} // annotation created

* using the newly created @Exercise annotation is easy simply use the at (@) symbol, followed by the type name. And then apply it to other java code.

applying to some classes example:

@Exercise() public class Cheetah {}

@Exercise public class Sloth {}

@Exercise

public class ZooEmployee {}

* parenthesis is optional in @Exercise() /@Exercise usage, when no elements are used
* if an annotation is declared on a line by itself, then it applies to the next nonannotation type found on the proceeding lines. this applies when multiple annotation exist

**Specifying a Required Element**

an annotation element is an attribute that stores values about the particular usage of an annotation.

changing previous @Exercise marker to an annotation that includes an element

public @interface Exercise {

int hoursPerDay();

}

* element hoursPerDay() looks like an abstract method, although we’re calling it an element (or attribute).
* usage changes now to:

@Exercise(hoursPerDay=3) public class Cheetah {}

@Exercise hoursPerDay=0 public class Sloth {} //DOES NOT COMPILE

@Exercise public class ZooEmployee {} //DOES NOT COMPILE

* when declaring an annotation, any element without a default value is considered required

**Providing an Optional Element**

public @interface Exercise {

int hoursPerDay();

int startHour() default 6;

}

next is applying the annotation to our classes

@Exercise(startHour=5, hoursPerDay=3) public class Cheetah {}

@Exercise (hoursPerDay=0) public class Sloth {}

@Exercise (hoursPerDay=2, startHour=”8”) //DOES NOT COMPILE incompatible type

public class ZooEmployee {}

**Selecting an Element Type**

similar to a default element value, an annotation element cannot be declared with just any type. it must be primitive type, a String , a Class, an enum, another annotation, or an array of any of these types.

public class Bear {}

public enum Size {SMALL, MEDIUM, LARGE}

public @interface Panda {

Integer height(); // does not compile Wrapper classes not supported

String[][] generalInfo(); // does not compile array is supported but not [][]

Size size() default SIze.SMALL; // COMPILES

Bear friendlyBear(); // does not compile Bear is not a supported type

Exercise exercise() default @Exercise(hoursPerDay=2); // COMPILES

}

**Applying Element modifiers**

* annotation elements are implicitly abstract and public whether you declare them that way or not.
* examples

public @interface Material {}

public @interface Fluffy {

int cuteness();

public abstract int softness() default 11;

protected Material material(); // DOES NOT Compile its implicitly public

private String friendly(); // DOES NOT Compile its implicitly public

final boolean isBunny(); // DOES NOT Compile can’t be final

}

**Adding a Constant variable**

* annotations can include constant variables that can be accessed by other classes without actually creating the annotation.
* just like interface variables, annotation variables are implicitly public, static, final
* these constant variables are not considered elements, though example marker annotations can contain constants

public @interface ElectricititySource {

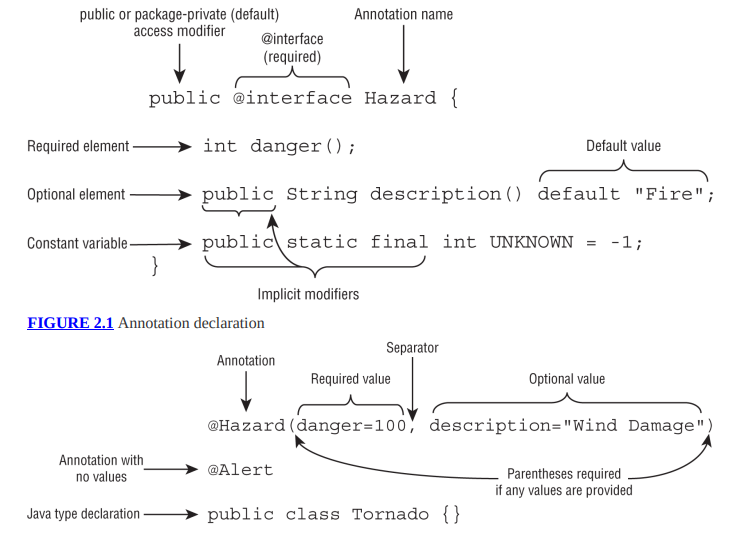
public int voltage();

int MIN\_VOLTAGE = 2;

public static final int MAX\_VOLTAGE = 18;

}

**Reviewing Annotation Rules**



## Applying Annotations

**Using Annotations in Declarations**

**Mixing Required and optional elements**

**Creating a value() Element**

**Passing an Array of Values**

## Declaring Annotation-Specific Annotations

**Limiting Usage with @ Target**

**Learning the ElementType Values**

**Understanding the TYPE\_USE Value**

**Storing Annotations with @Retention**

**Generating Javadoc with @Documented**

**Inheriting Annotations with @Inherited**

**Supporting Duplicates with @Repeatable**

**Reviewing Annotation-Specific Annotations**

## Using Common Annotations

**Marking Methods with @Override**

**Declaring Interfaces with @FunctionalInterface**

**Retiring Code with @Deprecated**

**Ignoring Warning with @SuppressWarnings**

**Protecting Arguments with @SafeVarargs**

**Reviewing Common Annotations**

## Exam Essentials

# Chapter 14 (Generics and Collections)

# Chapter 15 (Functional Programming)

# Chapter 16 (Exceptions, Assertions and Localizations)

# Chapter 17 (Modular Applications)

# Chapter 18 (Concurrency)

# Chapter 19 (I/O)

# Chapter 20 (NIO.2)

# Chapter 21 (JDBC)

# Chapter 22 (Security)