OCA Notes

Chapter 1: Java Building Blocks.

Object: a runtime instance of a class in memory. All the various objects of all the different classes represent the state of your program.

Elements of a java class: methods and fields. They are the members of a class. Variables hold the state of the program and methods operate on that state.

Keyword: reserved words of the Java language.

Method signature: the full declaration of a method. E.g.: *public int numberVisitors (int month)*

There are two types of comments:

```
// Single-line comment                  /* And
/*                                        * // they can be
Multiple-line comment                     * combined
*/                                        */
```

Classes and files.

Classes are .java extension. Public classes are not required. You can put multiple classes in one file, at most one of those classes is allowed to be public. The name of the file needs to match the public class.

The main() method is the entry point of the Java program. This is managed by the JVM. The JVM calls the OS to allocate memory and CPU time, access files and so on.

```
public class Zoo {
      public static void main(String[] args) { }
}
```

Public is the access modifier (it can be protected or private)
Static binds the method to its class.
Void is the return type (can be any other type, primitive or user defined). It's a good practice to use the a void method to change an object's state.
Main is the name of the method.
Inside the parenthesis is the argument. **Args** is the name. **String** is an array of java.lang.String type. These are acceptable as well for main: String args[] or String… args.

To compile a java file: *javac Zoo.java.* The result will be a file of bytecode by the same name, but with a .class extension.
To run the file: *java Zoo*
To run passing parameters: java Zoo Bronx "San Diego" Zoo

Package declarations and imports.

OCA Notes

Java come with thousands of built-in classes which are organized in packages. In order to use them, you would need to import them otherwise an error will be thrown.

There are a few things to keep in mind. Wildcards access all classes inside of a package. Java only looks at classnames in a package, it will not read / import classes contained in other sub-packages.
Static imports can import other types.
A classname has priority over wildcard when importing multiple packages and classes.
Java.lang is automatically imported.
Java.nio.file for files and paths.

When the need to import different classes with the same naming convention, it's a good approach to import one of them and use the other's fully qualified name, or the fully qualified name for both.

```
import java.util.Date;
public class Conflicts {
      Date date;
      java.sql.Date sqlDate;
}
```

```
public class Conflicts {
      java.util.Date date;
      java.sql.Date sqlDate;
}
```

Importing both classes using the classnames or the wildcard, would be recognized by java as code error and will not compile.

Creating Objects.
To create an instance of a class, the keyword 'new' is used. E.g.: Random r = new Random();
Random() is a constructor. The purpose of the constructor is to initialize fields. It must match the name of the class and there isn't any return type in the method signature.

Order of initialization: fields and instance initializers blocks are run in the order in which they appear in the file. However, the constructor runs after all fields and instance initializers blocks have run.
Instance Initializer example: `{ System.out.println("setting constructor"); }`

Data types.
Java contains two types of data: primitives and reference types. There are 8 primitives types built in the Java language:

boolean	true or false		true
byte	8-bit integral	2^8 = 2*2=4*28*2= 16*2=32*2=64*2= 128*5=256 This means the range for byte is -128 to 127	123
short	16-bit integral	2^{16}	123
int	32-bit integral	2^{32}	123

long	64-bit integral	2^{64}	3123456789L
float	32-bit floating-point	2^{32}	123.45f
double	64-bit floating-point	2^{64}	123.456
char	16-bit Unicode		'a'

You can now the max value of a primitive type by calling the MAX_VALUE field. E.g.: `System.out.println(Integer.MAX_VALUE);` and it will print ***2,147,438,647***

A number defined in a variable is called a *literal*. E.g.: ***`long max = 3123456789L;`***
In literals, you can use underscore for big numbers. This is NOT accepted by Java → ***`double a = _1000_._00_;`*** They can go anywhere between the numbers in order to make easier the value to read. E.g.: ***`double a = 1_000_000.0_0;`***

Base 10 numbers – decimal number system (0-9).
Base 8 numbers / octal (0-7), uses 0 as the prefix. E.g.: 017
Base 16 numbers / hexadecimal (0-9, A-F), uses l followed by x or X as a prefix. E.g.: 0xFF
Base 2 numbers / binary (0-1), uses 0 followed by b or B as a prefix. E.g.: 0b10

Reference Types.
These types refer to an object (instance of a class). They hold the address where the object is located in memory – pointer. A value can be assigned in one of two ways: a reference can be assigned to another object of the same type or a new object using the new keyword.

Reference types can be assigned null value, while primitives cannot. Reference types can be used to call methods when they do not point to null. Primitives do not have methods.

Identifiers. There are three rules for identifiers' names:

- The name must begin with a letter or the symbol $ or _.
- Subsequent characters may also be numbers.
- You cannot use a java reserved word for this.

Understanding default initialization of variables.
Local variables are defined within a method. They do not have a default value.
Instance (also called fields) and class variables are not local. Class variables have the *static* keyword. These are not required to initialize them, they get assigned a default value in that case.

boolean → false
byte, short, int, long → 0
float, double → 0.0
char → '\u0000' (NUL)
all objects reference type → null

OCA Notes

<u>Variable scope.</u>
Always verify the scope of local, class and instance variables.

- Local variables – in scope from the declaration to the end of the block.
- Instance variables – in scope from the declaration until the object is garbage collected.
- Class variables – in scope from the declaration until the program ends.

<u>Ordering elements in a class.</u>
This the following order for the elements in a class:

Package // may not be required
Import // may not be required – goes immediately after package
Class // required – goes immediately after import
{ fields and methods } // may not be required – anywhere inside the class
// comments – anywhere in the file

Garbage collection.
All java objects are stored in the program memory's heap. The heap (free store) is a large pool of unused memory allocated to your java app. The Garbage Collector deletes the objects from memory that are no longer reachable by the app.

System.gc(); → a request for Garbage Collector to run, but this request can be ignored by Java.
An object is not reachable when:

1) No reference points to it.
2) The reference to the object goes out of scope.

Finalize() can be implemented but is only run when the object is eligible for the garbage collector. The method can run zero or one time. If garbage collector fails to collect the object and runs a second time, finilaize() won't be called again.

Benefits of java.

- Object oriented. Organized in classes.
- Encapsulation. Supports access modifiers to protect data from unintended access and modifications.
- Platform independent. Java code compiles to bytecode. Can be read by the JVM on any OS.
- Robust. Prevents memory leaks.
- Simple. No pointers, no operator overloading.
- Secure. Java code runs in the JVM, which creates a sandbox to execute the code.

Chapter 2: Operators and Statements.

<u>Operator</u>: a special symbol that can be applied to a set of variables, values or literals – referred as operands – and that returns a result. Operators can be: unary, binary and ternary. Unless overwritten by parenthesis, Java follows order of operation, then Java guarantees left-to-right evaluation. Order of operator precedence:

Post-unary operators	*expression++, expression--*
Pre-unary operators	*++expression, --expression*
Other unary operators	*~, +, -, !*
Multiplication / Division / Modulus	**, /, %*
Addition / Substraction	*+, -*
Shift operators	*<<, >>, >>>*
Relational operators	*==, !=*
Logical operators	*&, ^, |*
Short-circuit logical operators	*&&, ||*
Ternary operators	*boolean expression ? expression1 : expression2*
Assignment operators	*=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=*

<u>Arithmetic Operators.</u>
Addition (+), subtraction (-), multiplication (*), division (/) and modulus (%). They include ++ and -– unary operators. *, /, % have a higher precedence than + and -. They can be applied to all primitives type except for boolean and String. When +, += is applied to String, it's concatenation.

Numeric Promotion.
Rules:

- If two values have different data types, Java will automatically promote one of the values to the larger of the two.
  *int x = 1; long y = 33; x * y data type will be long.*
- If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
  double x = 39.21; float y = 2.1f; x + y data type will be double.
- Smaller data types (byte, short, char) are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operand is int.
  short x = 10; short y =3; x / y data type will be int.
- After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.
  short x = 14; float y = 13; double z = 30;
  *x * y / z data type result will be double. X is first promoted to int and then to float to be multiplied with y. The result will be promoted to double to be divided by z.*

OCA Notes

Unary Operators.
Only requires one operand / variable to function.

+	*Positive number*
-	*Negative number*
++	*Increments value by 1*
- -	*Decrements value by 1*
!	*Inverts a Boolean's logic value*

Logical Complement: !, flips the logical value of a Boolean.
Cannot be applied to a numeric variables.
Negation Operator: -, reverses the sign in a numeric expression.
Increment and Decrement Operators: ++, - -, to be applied to numeric operands and have higher precedence or order, as compared to binary operators. These can be:

- Pre-decrement: --variable
- Pre-increment: ++variable
- Post-decrement: variable- -
- Post-increment: variable++

In the scenario in which variable is 5, all the operations above produce a different output. Pre operators will increment or decrement the value first, then return the result. Post operators will return the result first, then increment or decrement the value.

Binary Operators.

- Assignment operators. Modifies the value of the variable with the value on the right-hand side.
  E.g.: *int x = 1;*
  Java will automatically promote from smaller to larger data type. Error will be thrown if it's attempted to assign larger to smaller data type.
  E.g.:
  int x = 1.0;
  short y = 1921222;
  int z = 9f;
  long t = 192301398193810323;

  Casting primate values. Examples above can be fixed by casting the large data types to smaller ones.
  E.g.:
  int x = (int)1.0;
  short y = (short)1921222; // value overflows to 20,678
  int z = (int)9f;
  long t = 192301398193810323L;

  Example of underflow: System.out.println(2147483647+1); // prints -2147483648

  Another casting example to override JVM behavior:

short x = 10; short y = 3;
*short z = (short)(x * y);*

- Compound Assignment Operators. Includes +=, -=, /=, *=.
  E.g.:
  int x = 2, z = 3;
  *x = x * z; // simple assignment.*
  *x *= z; // Compound assignment.*

  Compound operators can apply a cast of larger values to small automatically.
  E.g.:
  long x = 10; int y = 5;
  *y *= x;*

  Another valid and not common compound assignment:
  long x = 5;
  long y = (x=3);
  System.out.println(x);
  System.out.println(y);
  Both output 3.

- Rational Operators. Compares two expressions and returns a Boolean value. This applies only to numeric primitive data types. As other operators, the smaller operand will be promoted the larger data type.
  E.g:
  int x = 10, y = 20, z = 10;
  System.out.println(x < y); // true
  System.out.println(x <= y); // true
  System.out.println(x >= z); // true
  System.out.println(x > z); // false

- Logical Operators. These &, | and ^ are applied to numeric and Boolean primitive data types. When applied to Boolean is referred as logical operators. When applied to numeric values is referred as bitwise operators, because they perform bitwise comparisons of the bits that compose the number.

x & y (AND)

	y = true	y = false
x = true	True	False
x = false	False	False

x | y (Inclusive OR)

	y = true	y = false
x = true	True	True
x = false	True	False

X ^ y (Exclusive OR)

	y = true	y = false
x = true	False	True
x = false	True	False

Short-circuit operators (&&) and (||) are very similar to the previous ones, expect that the right hand of the expression may never be evaluated.
E.g.: *boolean x = true || (y < 4);*

Other examples:
```
If(x != null %% x.getValue() < 5) {
        // Do something
}
```

- Equality Operators.
  Equals (==) and not Equals (!=) operators are used in the following scenarios:
  1. Comparing two numeric primitives types. If data types are different, the smaller will be promoted to the larger type.
  2. Comparing two boolean values.
  3. Comparing two objects, including null and String values.

Java Statements.

A statement in Java is a complete unit of execution, terminated with a semicolon (;). Control flow statements break up the flow of executing by using decision making, looping and branching, allowing the application to selectively execute specific segments of code.

The if-then Statement.
We only want to execute a block of code under certain circumstances. This statement allows the code to execute if the boolean expression evaluates to true at runtime.

```
if ( booleanExpression ) {
   // Branch if true
}
```

The if-then-else Statement.
We want to execute different blocks of code when our condition may have different outputs.

```
if ( booleanExpression ) {
   // Branch if true
} else {
   // Branch if false
}
```

Ternary Operator. It's the only operator in java that takes 3 operands and is of the form:
`booleanExpression ? expression1 : expression2`
The first one must be a boolean expression, and the second and third can be any expression that returns a value.

The Switch Statement.
It's a complex decision-making structure in which a single value is evaluated, and flow is redirected to the first matching branch, known as a ***case*** statement. If no ***case*** is found that matches the value, an optional ***default*** statement will be called. If no ***default*** option is available, the whole block will be skipped.

Supported data types: byte and Byte (wrapper class), short and Short (w. class), char and Character (w. class), int and Integer (w. class), String and enum values. Does not support boolean and long.

```
switch(variableToTest) {
      case constantExpression1:
            // Branch for case1;
            break;
      case constantExpression2:
            // Branch for case2;
            break;
      default:
            // branch for default;
      }
```

Compile-time Constant values.
The values in each case statement must be compile-time contact values of the same data type as the switch value. This means only literals, enum constants or final constant variables of the same data type can be used.

The While Statement.
A repetition control structure / loop. It's the simplest loop in Java.
```
while(booleanExpression) { // Body }
```
The body will execute as long as the boolean expression is true.

The do-while Statement.
Do-while loop is guaranteed that the body will execute at least once, since the boolean expression is evaluated at the end of the block:
```
do { // Body } while(booleanExpression);
```

The Basic for Statement.
This contains an initialization block, a boolean expression block as the previous loops, and an update statement block.
```
for(initialization; booleanExpression; updateStatement) { // Body }
```

Variables declared in the initialization block has a scope limited to the loop.
The boolean expression is evaluated in ever iteration before the loop executes.

5 variations of this for statement to familiarize with:

1. Creating an infinite loop.
   ```
   for ( ; ; ) { System.out.println("Hello World");
   ```
2. Adding multiple terms to the for statement.
   ```
   int x = 0;
   for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " "); }
   ```
3. Redeclaring a variable in the initialization block.
   ```
   int x = 0;
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // Does not compile
   System.out.print(x + " "); }
   ```
4. Using incompatible data types in the initialization block.
   ```
   for (long y = 0, int x = 4; x < 5 && y < 10; x++, y++) { // Does not
   compile
   System.out.print(x + " "); }
   ```
5. Using loop variables outside the loop.
   ```
   for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " "); }
   System.out.print(x); // Does not compile
   ```

<u>The For-each Statement / Enhanced loop.</u>
It was specifically design for iterating over arrays and Collections objects. It's composed of an initialization section and an object to be iterated over.

```
for(datatype instance : collection) { // Body }
```

The right-hand side of the loop must be a built-in java array or an object whose class implements java.lang.Iterable, which includes most of the Java Collections framework. The left-hand side must include a declaration of an instance of a variable, whose type matched the type of a member of the array or collection in the right-hand side. For each iteration, the instance will have a different value from the collection in the right-hand side.

<u>Nested Loops.</u>
Loops can be nested, one inside of the other. And optional labels can be applied in order to make the code more understandable.

```
OUTER_LOOP: for(datatype instance : collection) {
INNER_LOOP:       for(initialization; booleanExpression; updateStatement) {
      // Body
}}

while(booleanExpression) {
      do { // Body } while(booleanExpression);
// Body }
```

<u>The Break Statement.</u>
It transfers the flow of control out to the enclosing statement. It applies to all kinds of loop seen previously.

```
optionalLabel : while (booleanExpression) {
```

```
      // Body
      break optionalLabel;
}
```

Without the label, break will terminate the nearest inner loop. The optional label allows us to break out of a higher-level outer loop.

The Continue Statement.

It causes the flow to finish the execution of the current loop.

```
optionalLabel : while (booleanExpression) {
      // Body
      continue optionalLabel;
}
```

The continue statement transfers the control to the boolean expression that determines if the loop should continue. It only ends the current iteration of the loop. As the break statement, it is applied to the nearest inner loop, unless optional label is used.

	Allows optional label	Allows unlabeled break	Allows continue statement
if	Yes	No	No
while	Yes	Yes	Yes
do while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No

Chapter 3: Core Java APIs.

API stands for Application Programming Interface. In java, an interface is something special. In the context of an API, it can be a group of class of interface definitions that gives you access to a service or functionality.

Creating and Manipulating Strings.
String class is fundamental in Java. It doesn't need a new keyword statement to create an instance:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Both creates a reference type variable that points to Fluffy.

Concatenation: placing one String before the other and combining them. + operand is used for this purpose. If both operands are numeric, it's an addition. If at least one operand is a String, it's concatenation. The expression is always evaluated from left to right.

Immutability: once a String is created is not allowed to change. It can't be made larger or smaller, and you cannot change one of the characters inside it.

The String pool: also known as the intern pool, is a location in the JVM that collects all strings, for reusing them and taking less memory while the program is executing. For the first instance of "Fluffy" above, Java will add it to the pool. For the second one with the new keyword, we are requesting Java to not use the pool and create an object, though is less efficient. This is not a good practice, but it's allowed.

Important String Methods.
Remember that String is a sequence of characters and Java counts from 0 when indexed. Here are most commonly used methods:

- length()
  Returns the number of characters in the String.
  ```
  variable.length()
  ```

- charAt()
  Lets you query the string to find out what character is at a specific index. Throws an exception if match not found.
  ```
  variable.charAt(2)
  ```

- indexOf()
  Search characters in the string and finds the first index that matches the desired value. It can work with individual characters or a whole String as input. It doesn't throw an exception if it cannot find a match.
  ```
  int indexOf(int ch)
  int indexOf(int ch, int fromIndex)
  int indexOf(String str)
  ```

int indexOf(String str, int fromIndex)

- substring()
  Also looks for characters in a string and returns parts of it. Frist parameter is the index t start with for the returned string. The second parameter is optional (end of string). It throws an exception if indexes are backwards or greater than the length of the string.
  String substring(int beginIndex)
  String substring(int beginIndex, int endIndex)

- toLowerCase() and toUpperCase()
  Converts your data to lower and upper case.
  string.toLowerCase();
  string.toUpperCase();

- equals() and equalsIgnoreCase()
  Checks whether two String objects contain exactly the same characters in the same order. The second one does the same with the exception that it will convert the characters' case if needed.
  boolean equals(Object obj)
  boolean equalsIgnoreCase(String str)

- startsWith() and endsWith()
  They look at whether the provided value matches part of the String.
  boolean startsWith(String prefix)
  boolean endsWith(String suffix)

- contains()
  Also looks for matches in the String. The searched value can be anywhere within the String.
  boolean contains(String str)

- replace()
  It does a simple search and replaces the value in the String.
  boolean replace(char oldChar, char newChar)
  boolean replace(CharSequence oldChar, CharSequence newChar)

- trim()
  It removes whitespaces from the begging and end of the String but leaves the ones in the middle of it. Whitespaces consists of spaces along with the \t (tab) and \n (newline) characters; also including \r (carriage return).
  public String trim()

Method chaining.
Method chaining refers to calling methods / functionality chained one to another, which helps to save on lines of code and to avoid instantiating new variables. This is an example:

```
String result = "AniMaL     ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

Using the StringBuilder Class.

A small program can create a lot of String objects very quickly. Due to immutability, new objects will be created when trying to "modify" a string. With StringBuilder, the same space is reused when appending characters in a string. StringBuilder is not immutable, and when applying method chaining, the object changes its own state and returns a reference to itself.

```
StringBuilder a = new StringBuilder("abc");
StringBuilder b = a.append("de");
B = b.append("f").append("g");
System.out.println("a=" + a);
System.out.println("b=" + b);
// Both print "abcdefg" as they reference to the same object.
```

Creating a StringBuilder: We can make an instance in three ways using the new keyword, in which we may pass nothing, a string or a size. In the last scenario, we are suggesting Java what the size of the string will be, but the size can be increased if needed.

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

Important StringBuilder Methods.

- charAt(), indexOf(), length() and substring()
  These work the same as in the String class.

- append()
  It adds the parameter to the StringBuilder and returns a reference to the current StringBuilder.
  ```
  StringBuilder append(String str)
  ```

- insert()
  Same as append, adds a parameter and returns a reference to the current StringBuilder, with the exception that we need to specify the offset / index where to insert it.
  ```
  StringBuilder insert(int offset, String str)
  ```

- delete() and deleteCharAt()
  Delete() removes characters from the sequencer and returns the current StringBuilder. DeleteCharAt() is convenient when you want to delete only one character in the sequence.
  ```
  StringBuilder delete(int start, int end)
  StringBuilder deleteCharAt(int index)
  ```

- reverse()

Reverses the characters in the sequence and returns a reference to the current StringBuilder.

```
StringBuilder reverse()
```

- toString()
  Converts a StringBuilder to String. `String s = sb.toString();`

StringBuilder vs. StringBuffer.
StringBuilder was introduced in Java 5. Before that, StringBuffer was used, which does the same thing but it's slower as it is thread safe.

Understanding Equality.
In Chapter two we saw equality for primitive types. Now, we see for non-primitives and to point out to never use == in real life scenarios, but rather for the certification exam.

```
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
```

Both above are checking on reference equality. Next String examples check for reference equality as well. Remember how JVM reuses String literals by adding a single one to the pool / memory.

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);         // true
String z = " Hello World".trim();
System.out.println(x == z);         // false
```

Above, z variable is computed at runtime. Since x and z are not the same at compile time, a new String object is created. Below, we check for logical object equality. StringBuilder does not implements equals(), therefore if this method is called, it will check reference equality only.

```
System.out.println(x.equals(z));         // true
```

Understanding Java Arrays.
An array is an area of memory on the heap with space for a designated number of elements. A String is implemented as an array with some methods that you might want to use when dealing with characters. A StringBuilder is implemented as an array where the array object is replaced with a new bigger one when it runs out of space to store all the characters. A big difference is that an array can be of any other Java type.

```
char[] letters;
```

The variable letters is a reference, char is a primitive but it’s the data type that goes into the array itself. It’s an ordered list and can contain duplicates.

Creating an Array of primitives.

```
int[] numbers1 = new int[3];
```

Where int is the type of array, the brackets is required right after the type and the brackets at the end of the right-hand side indicates the size. All elements are set to the default value of the type. In this case, 0.

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

In this case, we specify the initial values of the elements instead of using the default. The right-hand size is redundant for which Java lets you write the following:

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

These declarations are the same:

```
int[] variable; int [] variable; int variable[]; int variable [];
```

Multiple declarations in a single line:

```
int[] ids, types; int ids[], types[]; int ids[], types;
```

Creating an Array of reference types.

```
String [] bugs = {"cricket", "bettle", "ladybug");
String [] alias = bugs;
System.out.println(bugs.equals(alias));   // true
System.out.println(bugs.toString());      // [Ljava.lang.String;@160bc7c0
```

In the sample above, we use String (non-primitive type) to build the object. Equals method checks for reference equality, it doesn’t look for the values contained in the object. This array doesn’t allocate space for the String objects. The array allocates space for a reference (pointer) to where the String objects are really stored. If not specified, the default values is null.

Using an Array.

Sorting. Java makes it easy by providing a sort method that you can pass almost anything to it: Arrays.sort(). Package java.util.Arrays or java.util.* is needed for this operation.

```
Arrays.sort(arrayVariable);
```

Searching. Java provides binary search but the array needs to be sorted. If not, the result is not predictable. When value is found, the index of the match is returned, otherwise a negative value.

```
Arrays.binarySearch(arrayVariable, searchedValue);
```

Multidimensional Arrays. It’s basically putting an array inside of an array. These are the ways that can be declared:

```
int[][] vars1;
int vars2 [][];
int[] vars3[];
int[] vars4 [], space [][]; // a 2D and a 3D array
```

Size can be specified in the declaration. The following is a symmetric array:
```
String[][] rectangle = new String[3][2];
```
This is an asymmetric array:
```
int[][] differentSize = {{1,4}, {3}, {9,8,7}};
```
Another way to declare an asymmetric array:
```
int[][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

In order to iterate through a dimensional array, nested loops are used. These can handle both symmetric and asymmetric arrays. These are enhanced loops but regular ones are used as well:
```
for(int[] inner : twoD) {
     for(int num : inner)
          System.out.print(num + " ");
     System.out.println();
}
```

Understanding an ArrayList.
It has one shortcoming: you need to know how many elements will be in the array when you create it and then you are stuck with that choice. Like a StringBuilder, the size can change at runtime as needed. Like an Array, it is an ordered sequence that allows duplicates. It needs one of the following imports: java.util.* or java.util.ArrayList.

How to instance it:
```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```
Using generics, introduced in Java 5:
```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```
Since ArrayList implements the interface List, the following can be done:
```
List<String> list6= new ArrayList<>();
ArrayList<String> list7 = new List<>();        // Does not compile
```

Important ArrayList Methods.
A new class or element is used in the method signatures: E.
E is used by convention in generics to mean any class that this array can hold. E means anything you put inside < >, and when nothing is specified, E means Object. ArrayLists implement toString().

- add()
  Insert a new value in the ArrayList. Method signature:
  ```
  boolean add(E element)      // Always returns true
  void add(int index, E element)
  ```

- remove()

Removed the first matching value in the ArrayList or remove the element at the specified index. Method signature:

```
boolean remove(Object object) // Returns true or false
E remove(int index)          // Returns element removed
```

- set()
  Changes one of the element of the ArrayList without changing its size. Method signature:
  ```
  E set(int index, E newElement)
  ```

- isEmpty() and size()
  Look at how many slots are in use. Methods signatures:
  ```
  boolean isEmpty()
  int size()
  ```

- clear()
  It discards all elements of the ArrayList at once. Method signature:
  ```
  void clear()
  ```

- contains()
  Checks whether a certain value is in the ArrayList. Method signature:
  ```
  boolean contains(Object object)
  ```

- equals()
  This is a custom implementation of equals) and it's able to compare two lists to see if they contain the same elements in the same order. Method signature:
  ```
  boolean equals(Object object)
  ```

Wrapper Classes.

What happens if we want to put a primitive type in an ArrayList? We can put a wrapper class, which is an object type that corresponds to each primitive type.

Primitive	**Wrapper Class**	
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

Converting from a String:

Wrapper Class	**Converting String to Prim.**	**Converting String to Wrapper**
Boolean	Boolean.parseBoolean("true");	Boolean.valueOf("TRUE");
Byte	Byte.parseByte("1");	Byte.valueOf("2");

Short	Short.parseShort(“1”);	Short.valueOf(“2”);
Integer	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”);
Character	None	None

Autoboxing. When converting from String to primitive or wrapper class, we don’t need to worry which one is returned. Since Java 5, you can type any primitive value and Java will convert it to the relevant wrapper class for you. **Examples in WrapperClasses.java file**.

Converting between array and List.
From List to Array:
```
List<String> list = new ArrayList<>();
Object[] objectArray = list.toArray();
String[] stringArray = list.toArray(new String[0]);
```

The last one is the right way to convert from ArrayList to Array. By specifying the size as 0, Java will create a new array with the proper size for the return value.
From Array to List:
```
String[] array = {“hawk”, “robin”};
List<String> list = Arrays.asList(array);
```
Now both array and list variables reference to the same data, and the data objects can be modified through any of the two references. Deleting and adding not possible since the object is a fixed size, even when we convert array into list.

Sorting.
In order to sort, you need to use a different helper class:
```
Collections.sort(numbers2);
```

Working with Dates and Times.
In order to work with Dates and Times, the following import is necessary: java.time.*
Three ways to work with dates in java and the OCA exam: **LocalDate** (contains just a date – no time, no time zone), **LocalTime** (contains just time – no date, no time zone) and **LocalDateTime** (contains both date and time – no time zone). Oracle recommends to avoid time zones unless they are really needed. For this, **ZonedDateTime** handles it.

```
LocalDate.now();
LocalTime.now();
LocalDateTime.now();
```

Method signatures for Date:
```
public static localDate of(int year, int month, int dayOfMonth)
public static localDate of(int year, Month month, int dayOfMonth)
```

Month is an enum (enmus are java classes). For months, Java counts starting from 1, making Months an exception from everything else, which Java starts counting from 0.

Method signatures for Time:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)
```

Method signatures for DateTime:

```
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute, int second)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour,
int minute)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour,
int minute, int second)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour,
int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime time)
```

The date and time classes have private constructors to for you to use the static method. Therefore, the new keyword / constructor instance can't be applied:

```
LocalDate d = new LocalDate();     // Does not compile
```

Dates and Times are immutable, just like String was. This means we need to remember to assign the results to a reference variable, so they are not lost.

Working with Periods.

We use a Java class called Periods to work with date times that change in our code, so the code can be reusable by just adjusting the period. There are 5 ways to create a Period class:

```
Period annually = Period.ofYears(1);
Period quarterly = Period.ofMonths(3);
Period everyThreeWeeks = Period.ofWeeks(3);
Period everyOtherDay = Period.ofDays(2);
Period everyYearAndWeek = Period.of(1, 0, 7);
```

Method chaining for Periods is not allowed, since only the last method in the chain will be applied to the period. E.g.: `Period.ofYears(1).ofWeeks(1);`

There's another class called *Duration*, which is applied to hours, minutes, seconds and nanos, but it's out of the scope for the OCA.

Formatting Dates and Times.

We can specify the format we want to handle the data of the objects. ISO is the standard for dates and would like this:
dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
Where dateTime is an object and DateTimeFormatter is the implementation of the class. Same can be applied to either Date or Time.

Also, a format can be defined in a object before applying to the Date object as follow:
DateTimeFormatter shortDateTime =
DateTimeFormatter.___________(FormatStyle.SHORT);
Where blank can be filled with: **ofLocalizedDate** [for Date and DateTime], **ofLocalizedDateTime** [for DateTime only] or **ofLocalizedTime** [for Time and DateTime].

There are two predefined formats: SHORT and MEDIUM.
A SHORT DateTime format looks like this: *01/20/20 11:12 AM*
A MEDIUM DateTime format looks like this: *Jan 20, 2020 11:12:34 AM*

You can also create your own format by using the method ofPattern(String str). E.g.:
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
The more Ms, the more verbose Java will display the month.

You can also use the parse() method to pass a String to be converted to Date / Time / DateTime. E.g.:
LocalDate date = LocalDate.parse("01 02 2015", f);
Where f is a DateTimeFormatter object.
LocalTime time = LocalTime.parse("11:22");

Chapter 4: Methods and Encapsulation.

Designing Methods.

Method declaration refers to all the parts of method, the information needed to call it.

```
public final void nap(int minutes) throws InterruptedException ( // take nap }
```

Where "public" is the access modifier, "final" is the optional specifier, "void" is the return type, "nap" is the method name, "minutes" is a parameter and "InterruptedException" is an exception declaration / handling.

Access Modifiers.

Java offers 4 choices:

- *Public*. The method can be called from any class.
- *Private*. The method can be called only from within the same class.
- *Protected*. Method can be called only from any class in the same package or subclass.
- *Default*. Method can be called only from the classes in the same package. There's no keyword for this one, you just don't put any access modifier.

Optional Specifiers.

Unlike access modifiers, you can have multiple optional specifiers in the same method declaration (not all combinations are legal). You can have them in any order, in case of using more than one. Here's the list:

- *Static*. Covered in Chapter 5. Used for class method.
- *Abstract*. Covered in Chapter 5. Used when not providing method body.
- *Final*. Covered in Chapter 5. Used when a method is not allowed to be overridden by a subclass.
- *Synchronized*. Covered in OCP scope.
- *Native*. Covered in OCP scope.
- *Sctrictfp*. Covered in OCP scope.

Return Type.

The return type might be an actual Java type such as String or int. It can be a custom java object created by the programmer. If there is no return type, the *void* keywork is used. The methods that possess a return type, must have a return statement inside of the body. Only void can omit this return statement.

```
public String walk() { return “”; }
```

Method Name.

It follows the same rules previously stated in Chapter 1: an identifier may only contains letters, numbers, $ or _. First characters cannot be a number and reserved words are not allowed. By conventions, methods begin with a lowercase letter.

Parameter List.

Although the parameter list is required, it doesn't have to contain any parameters. If you do have multiple, they are separated by comma. Some rules will be shared shortly when talking about varagrs.

Optional Exception List.
Code can indicate something went wrong by throwing an exception (will be covered in Chapter 6). It is optional to include as many exceptions needed in the method signature, if needed and it goes right after the parameter list, followed by the keyword "throws".

Method body.
It's simply a block of executable code, enclosed by two curly braces (there are exceptions with abstract classes and interfaces, covered in Chapter 5).

<u>Working with Varargs.</u>
A method can use a vararg parameter as if it is an array. It's different than an array. It needs to be strictly the last element in the parameter list, for which it can be used only once.
When calling a method with vargars, you can pass an array, or a list of elements of the array and let Java crate it for you. Also, you can omit it by passing zero parameters.

<u>Applying Access Modifiers.</u>
Private Access.
Only code in the same class can call private method or access private fields. See FatherDuck.java and BadDuckling.java classes in package pond.duck for examples.

Default (Package Private) Access.
Allows classes in the same package to access the members of the class. No access modifier / keyword is used. See MotherDuck.java and GoodDuckling.java classes in package pond.duck for examples.

Protected Access.
Allows everything that default access allows and more. Adds the ability to access members of a parent class (this means by a subclass, using the 'extends' keyword) and by other classes as long as they are under the same package, by creating an instance ('new' keyword). See all classes under "pond" folder as examples.

However, there are some other scenarios, using an instance ('new' keyword): in class under a different package, where inheritance is present ('extends'), using an instance that refers to itself it's valid, but using an instance that refers to the parent class with the protected members, the code won't compile. See pond.swan.Swan.java example. To paraphrase, these are the two scenarios for protected access:

1) A member is used without using a variable. This is done through inheritance, extending the parent class with protected members.

2) A member is used through a variable. This is done by creating an instance, with the 'new' keyword. Only and exclusively works when under the same package as the class with protected members. Works in the subclass when referring to itself.

Public Access.
Anyone can access the members from anywhere.

<u>Designing Static Methods and Fields.</u>
Static methods don't require an instance of the class. You can thing of it as a member of the single class object that exists independently of any instances of that class.

Note: instance and static methods has only one copy of the code. Parameters and local variables (as the reference variable) gets space in the stack. Only data gets its "own copy". No need to duplicate copies of the code itself.

Calling a static variable or method.
You just put the classname before the method or variable and you are done. Example:
```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Instance variables can be used to access static methods and variables. Be advise that it doesn't matter how many reference variables point to the same object, a static method or variable is not duplicated, only one exists in memory. E.g.:
```
Koala.count = 4;
Koala k1 = new Koala();
Koala k2= new Koala();
k1 = null;
System.out.println(k1); // 4
k2 = 6;
k1 = 5;
System.out.println(Koala.count); // 5
```

Static vs. Instance.
There are static and instance members of a class (methods and fields); for which it's not allowed to try to call an instance member without creating an instance first, from a static method. However, only an instance method can call another instance member on the same class without using a reference variable, because instance methods do require an object. See Static and Gorilla examples.

Type	Calling	Legal?	How?
Static Method	Another static method or variable.	Yes	Using classname.
Static Method	An instance method or variable.	No	Not without instantiating the object.

Instance Method	A static method or variable.	Yes	Using the classname or a reference variable.
Instance Method	Another instance method or variable.	Yes	Using a reference variable.

Static Variables.
Some static variables are meant to change as the program runs. Counters are common examples of this. A static variable can be initialized in the same line it's declared: ***private static int counter = 0;***

Other static variables are meant to never change, to which we refer as a constant using the final keyword. A different naming convention is used: all uppercase letters. E.g: ***private static final int NUM_BUCKETS = 45;***

Static Initialization.
Similar to the instance initializers from Chapter 1, for static initializers, we add the static word before the curly braces. The statement can hold and assign as many variables as needed.

```
static { // body }
```

Static Imports.
Static imposts are for importing static members of classes, as regular imports are used for the class. You can use wildcard or import a specific member. The idea is that you shouldn't have to specify where each static method or variable comes from each time you use it. See StaticInJava.java for examples.

By importing the following member: ***import static java.util.Arrays.asList;***
Java will allow to replace this: ***List<String> list = Arrays.asList("one", "two");***
With this instead: ***List<String> list = asList("one", "two");***

<u>Passing Data Among Methods.</u>
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.

<u>Overloading Methods.</u>
It occurs when there are different method signatures with the same name but different types of parameters. Also allows different numbers of parameters. Everything other than the method name can vary for overloading methods (access modifiers, specifiers, return types and exception lists).

Overloading and Varargs.
Java treats varargs as if they were an array, but we need to keep in mind that they don't compile exactly the same. For which, the following case is not valid for overloading:

```
public void fly2(int[] lengths) {}
public void fly2(int... lengths) {}       // Does not compile
```

Then, in order to call a method, these are the ways we can pass parameters:

```
fly2(new int[] {1, 2, 3});          // Valid for array and varargs
fly2(1, 2, 3);                      // Valid only for varargs
```

Autoboxing.

```
Public void fly(Integer numMiles) {}
```

When parameters are wrapper classes, Java will do the autoboxing to transform the value to the primitive data type. However, java can recognize that if there's a method overloading in which there is a primitive type available (for this example, int), then it won't do extra work and use the method that receives the primitive type.

When it comes to Reference and Primitives types overloaded methods, java will always try to find the most specific version that it can, when you call the method and pass the data, by evaluating the data type of the parameter passed.

Creating Constructors.

It's a special method that matches the name of the class and has no return type. Constructors are used when creating a new object. This process is called instantiation because it creates a new instance of the class. The constructor gets called when we write 'new' keyword followed by the name of the class. See Bunny.java for examples.

Whenever you don't create a constructor in a class, java provides it automatically. This is called the *default constructor*. Java will provide it, empty, when the code gets compiled. Java won't create it if a constructor is found, no matter if it's empty or if it has parameters, or the access modifier. This code snippet calls a default constructor:

```
public class Rabbit {
     public static void main(String[] args) {
          Rabbit rabbit = new Rabbit();
} }
```

Interesting note: if you code a constructor to be private, the class cannot be instantiated. It's a good practice if all methods within that class are static or the class wants to control all calls to create new instances of itself.

Overloading Constructors.

You can have multiple constructors in the same class as long as they have different method signatures. See Hamster.java class for examples and see BetterHamster.java for examples on how to avoid code duplication.

The *this()* keyword when used in a constructor, it needs to be the first uncommented line of code that is called, otherwise it won't compile.

Constructor chaining: it's when constructors call each other until getting to the constructor that does all the work. For this sample, see Mouse.java.

Final fields. We can also initialize final variables in the constructor, as long as they haven't been initialized anywhere else. See MouseHouse.java for examples.

Order of Initialization.

1. If there's a superclass, initialize it first (this rule will be covered in Chapter 5).
2. Static variable declarations and static initializers in the order the appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Rule 1 and 2 apply if the class is referred without a 'new' call / instance. The four rules apply when a class instance is created. For examples see: InitializationOrderSimple.java, CallInitializationOrderSimple.java, InitializationOrder.java and YetMoreInitializationOrder.java classes.

Encapsulating Data.

Encapsulation means we set up the class so only methods in the class with the variables can refer t the instance variables. Callers are required to use these methods.

For each private member field, we set up an accessor method or getter and a mutator method or setter. Only these members have access to private fields. The setters and getters are set to public access modifier.

JavaBeans rules for naming conventions.

Rule	Example
Properties are private.	`private int numEggs;`
Getter methods begin with *is* or *get* if the property is a boolean.	`public boolean isHappy() { return happy; }`
Getter methods begin with *get* if the property is not a boolean.	`public int getNumEggs() { return numEggs; }`
Setter methods begin with *set*.	`public void setHappy(boolean happy) { this.happy = happy; }`
The method name must have a prefix of set/get/is, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) { numEggs = num; }`

Creating Immutable Classes.
By making a class immutable, it cannot be changed at all. This makes a class easier to maintain and helps with performance by limiting the number of copies made in memory. One step to make the class immutable is to omit the setters and allow to pass initial values through a constructor. See ImmutableSwan.java class as example. Also see NotImmutable.java class for examples of how a class can still be mutable by the fact is handling an datatype that allows changes (StringBuilder).

Writing Lambdas.
In Java 8, the language added the ability to write code using another style: functional programming. You specify what you want to do rather than dealing with the state of the object. You focus more in expressions than loops. It uses lambda expressions to write code. A lambda expression is a block of code that gets passed around.

It has parameters and a body like a full-fledge method do, but it doesn't have a name. For some traditional code and basic lambda samples to see the difference, see Animal classes in Lambda folder of this repo. You will see this code below in TraditionalSearch.java:

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

Java relies on context to figure our what lambda expressions mean. In the TraditionalSearch sample, we are passing a lambda to the "*print*" method. The lambda being passed is: *a -> a.canHop();*

The *print* method expect a CheckTrait as second parameter, since it's a lambda, Java tries to map our lambda to that interface: *boolean test(Animal a)*
Since the interface's method takes an Animal, that mean the lambda parameter has to be an Animal. And since that interface returns boolean, we know the lambda returns a boolean.

Now let's talk about syntax.

```
a -> a.canHop()
```

Where it has 3 parts. The first specify a single parameter with the name **a**. The second is an operator to separate the parameter and the body. The last one it's the body, calls a single method and returns the result (boolean).

```
(Animal a) -> { return a.canHop(); }
```

This one does the same the previous one but it's more verbose. The first part specifies a single parameter with he name **a** and states the type is **Animal**. The second is the operator that separates the parameter and the body. The last one is the body, that includes curly braces, return type and finish the statement with a semicolon.

Parenthesis is needed in the parameter, when it's more than one or the datatype is stated. Curly braces in the body is used for more than one statement, or we want to use the **return** keyword; also keep in mind the semicolons are needed to end each statement.

Predicates.
Lambdas work with interfaces that have only one method, like CheckTrait.java. These are called functional interfaces. But as the code expand, we would need more of these interfaces for each reference datatype we handle in our code. Therefore, Java has solved that problem by providing a package java.util.funtion that allow us using Predicate / Generics:

```
public interface Predicate<T> { boolean test(T t); }
```

Where T is the datatype (and it can be ANY datatype) that is being passed to the interface. Java 8 integrated the Predicate interface into some existing classes. ArrayList declares a removeIf() method that takes a Predicate. Please see PredicateSearch.java for examples.

Chapter 5: Class Design.

<u>Introducing Class Inheritance.</u>

When creating a new class in java, you can define the class to inherit from an existing class. Inheritance is the process by which the new child subclass automatically includes any public or protected primitives, objects, or methods defined in the parent class.

Java supports single inheritance, by which a class may inherit from only one direct parent class. A parent class may have multiple children in this case. Java also support multiple levels of single inheritance.

Java does not support multiple inheritance, in which a child class inherits from multiple parents, as it can lead to complex code and difficult to maintain. Java does allow one exception to the single inheritance rule: classes may implement multiple interfaces. Marking a class as final won't make it possible to apply inheritance; the compiler will throw an error.

In order to apply it, we use the 'extends' keyword.

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

Where *public* is the access modifier, *abstract* or *final* keyword is optional, *class* keyword required, *ElephantSeal* is the class name, *extends* followed by the parent class name to be inherited.

Applying class access modifiers.

As we know we can apply access modifiers (public, private, protected, default) to both classes and methods. For the top class that is in the file, only public and default package-level modifiers can be applied to it. The protected and private modifiers can be applied to inner classes.

If a class is default / no access modifier, only other classes in the same package can see it and extend it.

Creating Java objects.

In Java, all classes inherit a single class: java.lang.Object. Object is the only class without a parent class. But we don't see this in our code, and that's because at compile time java inserts code. If in your class, you don't extend any other class, java will automatically extend Object:

```
public class Zoo extends java.lang.Object { }
```

Only when the child class is extending a parent, the compile won't add this code, but it will to the parent if it doesn't extend from another parent. Java.lang.Object will always be on top of the tree.

Defining constructors.

We have learned that java creates a default no-argument constructor if no constructor is coded in a class, and that we can refer to inner constructors using the word this().

When inheritance is applied, we can use the super() keyword to refer to the constructor/s in the parent class. As this) keyword, super() needs to be the first line, otherwise it won't compile. For more examples see Animal.java and Zebra.java classes.

Understanding compiler enhancements.
Java automatically creates a no-argument constructor for all classes. If a child inherits a class, with no constructor definition, the java compile will automatically call the parent class' constructor with the super() keyword. The following Java classes definitions are the same, as Java will convert them all to the last one:

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

However, in the case in which the parent class does have a constructor defined, the child class won't compile if no constructor is defined in the child per se, as Java will insert a no-argument constructor during compile time, as well as adding the super() keyword:

```
public class Mammal { public Mammal(int age) {} }
public class Elephant extends Mammal { } // Does not compile.
```

The solution to this problem is to explicitly create a constructor in the child, and call super, passing an int argument.

Reviewing Constructor rules.

1. Frist statement is a call to another constructor within the class [this()] or a constructor in the direct parent class [super()].
2. The super() call may not be used after the first statement of the constructor.
3. If not super() call is declared, Java will insert a no-argument super(() as the first statement of the constructor.
4. If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

When calling a constructor, the parent one will always execute first.

Calling inherited class members.
If the parent class and child class are part of the same package, the child class may also use any default members defined in the parent class. A child class may never access a private member of the parent, at least not through direct reference. For examples, see Fish.java and Shark.java classes.

Super() vs. super
In the same way this() and this are unrelated to Java, super() and super are as well. Super() will explicitly call the parent constructor and may only be used in the first line of a constructor in the child class.
The second super is a keyword used to reference a member defined in a parent class and may be used throughout the child class:

```
public Rabbit(int age) {
	super();
	super.setAge(10);
}
```

<u>Inheriting Methods.</u>
When inheriting classes, there might be conflicts between the methods defined in both parent and child classes. We'll review how Java handles these scenarios.

Overriding a method.
If you may want to define a new version of an existing method of the parent class, in the child, that's what we call overriding a method. We override a method by defining a new one with the same signature and return type as the method in the parent class. See Canine.java and Wolf.java classes for examples.

The compiler will perform the following checks when you override a nonprivate method:

1. The method in the child class must have the same signature (including parameter list) as the method in the parent class. Note: if not, it would be overloading.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or boarder than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.

Redeclaring private methods.
If a member of the parent class is private, it's not accessible to the child. However, Java allows you to declare or redeclare a new version of the same method in the child class, with its unique implementation. This is not overriding in this case and therefore none of the rules are applied. Java allows to do this with the same or modified signature as the method in the parent class. They would end up totally independent, unrelated from each other.

Hiding static methods.
A hidden method occurs when a child class defines a static method with the same name and signature as a static method defined in the parent class. Method hiding is similar but slightly different than overriding. The same rules apply but one rule is added:

1. The method in the child class must have the same signature (including parameter list) as the method in the parent class. Note: if not, it would be overloading.
2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
3. The method in the child class may not throw a checked exception that is new or boarder than the class of any exception thrown in the parent class method.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.
5. The method defined in the child class must be marked as *static* if its marked as *static* in the parent class. If it's not marked *static* in the parent class, it cannot be *static* in the child.

Real life scenario: it is a good practice to avoid hidden methods, as it entitles complications in the code, also makes it hard to read.

Overriding vs. Hiding Methods.
The code behaves differently at runtime.

a) The child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method. In this manner, the parent method is never used unless an explicit call to the parent method is references, using the syntax *super.method().*
b) The parent version of a hidden method is always executed if the call to the method is defined in the parent class.

For this, see examples under Overriding vs. Hiding folder.

Creating final methods.
The rule is very simple: final methods cannot be overridden or hidden. By doing this, we are ensuring the method / behavior in the parent class is always present / executed.

<u>Inheriting Variables.</u>
Java doesn't allow variables in the parent class to be overridden, but instead, hidden.

Hiding variables.
To hide a variable, you only need to define it with the same name in the child as in the parent class. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference. The rule for this is similar as hiding a method. If you are referencing the variable within the parent class, the variable in parent is used, same as if you reference the variable in the child, the one in the child is used. You can also reference the variable in the parent class by using the super keyword. See Rodent.java and Mouse.java classes for examples.

Real life scenario: hiding variables is considered poor coding practice.

Creating Abstract Classes.

An abstract class is marked with the abstract keyword and cannot be instantiated. An abstract method is marked with the same keyword defined inside of an abstract class, for which no implementation is provided. It is extended by the child classes, and another difference from interfaces is that methods can be defined with code implementation in them.

Defining an abstract class.

Some rules to keep in mind. An abstract class is not required to have abstract methods, it may have nonabstract members. However, only an abstract member can be declared in a class like this. When a method is abstract, no code implementation can be included, otherwise it won't compile, only the method signature is allowed.

No abstract class or member can be marked as final, including nonabstract method with code implementation that we want the child classes to extend and override. Finally, a method may not be marked as both abstract and private.

Creating a concrete class.

An abstract class becomes useful when it's extended by a concrete class. A concrete class is the first nonabstract subclass that extends an abstract class and is required to implement all inherited abstract methods. When you see this, make sure it is implemented all abstract method, otherwise it won't compile.

Extending an abstract class.

An abstract class can extend another abstract class, without the need to implement the method of the parent. However, the fist concrete class that does extend the abstract class, does need to implement all abstract methods. See Animal.java, BigCat.java and Lion.java classes for examples.

There is an exception rule: a concrete class is not required to provide an implementation of an abstract method if an intermediate abstract class provides it. See BigCatV2.java and LionV2.java classes for examples.

Implementing Interfaces.

Java doesn't allow multiple inheritance but does allow a class to implement any number of interfaces. An interface is an abstract data type that fines a list of abstract public methods that any class implementing the interface must provide. Can also include constant variables and default methods, and does not provide any code implementation, ever. We use the *interface* keyword in the class signature.

Defining an interface:

```
public abstract interface CanBurrow {
     public static final int MAXIMUM_DEPTH = 2;
     public abstract int getMaximumDepth();
}
```

Where the abstract keyword in both class definition and method signature, as well as the static keyword in that constant above, are assumed. This means that whether or not you provide it, the compiler will insert it for you.

Implementing the interface:

```
public class FieldMouse implementd CanBurrow {
     public int getMaximumDepth() {
          return 10;
     } }
```

Think of an interface as a kind if an abstract class, since they share many rules:

1. Interfaces cannot be instantiated directly.
2. Is not required to have any methods.
3. May not be marked as final.
4. All top-level interfaces are assumed to have public or default access. They are assumed to be abstract whether you use the keyword or not. Making a method private, protected or final will trigger a compiler error.
5. All nondefault methods inside are assumed to be abstract and public in their definition. Making it private, protected or final will trigger a compiler error.

Inheriting an interface.

There are two inheritance rules to keep in mind:

1. An interface that *extends* another interface, as well as an abstract class that *implements* an interface, inherits all the abstract methods as its own abstract methods.
2. The first concrete class that *implements* an interface, or *extends* an abstract class that implements an interface, must provide an implementation for all inherited abstract methods.

When an abstract class implements an interface, it's not required to provide an implementation, but a concrete class is required to do so.

Abstract methods and multiple inheritance.

If a concrete class implements multiple interfaces, with they have method with the same name, as long as it has the same signature, only one will be implemented. If the method signatures are the same but different return type, it won't compile. If they are different return type, and have different parameters, which now will be consider different method signatures, then both will be implemented. Same as if an interface is extending other conflictive interfaces or an abstract method is implementing them, compile error will be thrown.
See Bears, Herbivores, Omnivore classes for more examples.

Interface variables.

Rules:

- Interface variables are assumed to be public, static and file. Making a variable private or protected will throw a compiler error.
- Its value must be defined when it's declared, since it's final.

Default interface methods.
It's a method defined inside the interface, with the *default* keyword, which body implementation is provided. A class that extends an interface with default methods, have the option to override them, but not required to do so. If the class doesn't override the method, default implementation will be used. In this manner, the method definition is concrete, not abstract.

The purpose of these is backward combability. In the case scenario in which a interface is extended by many classes, in case a code change needs to take place, a default method won't break the other classes implementation. Otherwise, all classes that extends the interface would need to make code changes.

```
public interface IsWarmBlooded {
	boolean hasScales();
	public default double getTemperature() { return 10.0; }
}
```

The default keyword is different from the access modifiers shown in Chapter 4. That's why getTemerature() is public. This default method only refers to its special use inside of an interface.
Rules:

- A default method may only be declared within an interface and not within a class or abstract class.
- A default method must be marked with the *default* keyword and a body must be provided.
- It's not assumed to be static, final or abstract, as it may be used or overridden by a class that implements the interface.
- It's assumed to be public. It won't compiled if marked as private or protected.

Default methods and multiple inheritance.
If a class implements two interfaces that have the default methods with the same name and signature, the compiler will throw an error. There is one exception to this rule: of the subclass overrides the duplicate default methods, the code will compile without issue – the ambiguity about which version of the method to call has been removed.

Static interface methods.
Rules:

- A static method is assumed to be public. If marked private or protected it won't compile.
- To reference the static method, a reference to the name of the interface must be used. This happens because static methods are not inherited when subclass implements the interface.

<u>Understanding Polymorphism.</u>
It's the property of an object to take 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.

- A reference that defines an interface the object implements, either directly or through a superclass.

A cast s not required if the object is being reassigned to a super type or interface of the object.

Please see *Lemur.java* for examples. In this example, only *Lemur* object is instantiated and then assigned to new objects declared for *HasTail* interface and *Primate* parent class, which is the nature of polymorphism. The new objects only have access to the members defined within them; therefore they cannot access the members unique to *Lemur* class.

Object vs. Reference.
All objects extend from the java.lang.Object class, for which the following example is possible:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

When polymorphism is applied, the object in the Heap memory remains the same, only the reference we are using to access the object is changed, along the ability to what members we can access. Rules:

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

Casting objects.
However, if we want to access all members of an object when a reference doesn't allow us to, can solve that problem by doing an explicit cast:

```
Primate primate = lemur;
Lemur lemur2 = (Lemur)primate;
```

Basic rules to keep in mind for casting objects:

1. Casting an object from a subclass to a superclass doesn't require an explicit cast.
2. Casting an object from a superclass to a subclass requires an explicit cast.
3. The compiler will not allow casts to unrelated types.
4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

Virtual methods.
The most important feature of polymorphism is to support virtual methods. It's a method in which the specific implementation is not determined until runtime. In fact, all non-final, non-static, non-private Java methods are considered virtual, since any of them can be overridden at runtime.

In simpler language, a class can extends another and override one of its methods, but Java may be unaware of that during compiling time. At run time, while making an instance, the implementation is determined when you call the overridden version of the method. See Bird.java and Peacock.java classes under Polymorphism folder for examples.

Polymorphic parameters.
It’s the ability to pass instances of a subclass or interface to a method, as a parameter. For example, you can define a method that takes an instance of an interface as a parameter. In this manner, any class that implements the interface can be passed to the method. Since you’re casting from a subtype to a supertype, an explicit cast is not required. See Reptile.java, Alligator.java, Crocodile.java and ZooWorker.java as examples.

Chapter 6: Exceptions.

Understanding Exceptions.

The role of exceptions. When you write a method, you can either deal with the exception or mate the calling code's problem. The role is to deal largely with the mistakes in the program. The key point to remember is that exceptions alter the program's flow.

Exception Types.

Java has a Throwable superclass for all objects that represent these events. Not all of them have the word *exception* in their classname, which can be confusing. Hierarchy is the following:

java.lang.Object → java.lang.Throwable →
java.lang.Error
java.lang.Exception → java.lang.RuntimeException

Error (subclass of Throwable) means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive "disappeared".

Runtime exception (subclass of Exception) include RuntimeException class and its subclasses, and tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. They are also known as *unchecked exceptions*. Note: compile errors are not runtime / unchecked exceptions.

A *checked exception* includes Exception class and all subclasses that do not extend RuntimeException. They tend to be more anticipated. For example, trying to read a file that doesn't exist. Why is it called checked? Java has the rule called the handle or declare rule, in which you need to handle or declare the checked exceptions in the method signature:

```
void fall() throws Exception { throw new Exception(); }
```

Keyword throws is used above, with the classname of the Exception / or any subclass that extend Exception. This method above might or might not throw an exception.

Throwing an exception.

There are two way of throwing an exception. As an example, something that's wrong in the code will throw an exception, as follow:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

This will throw an ArrayIndexOutOfBoundsException.

The second one, is to explicitly tell java to throw an exception, as follow:

```
throw new Exception();              throw new RuntimeException();
throw new Exception("Ow! I fell."); throw new RuntimeException("Ow! I fell.");
```

Using a Try Statement.

Java uses a try statement to separate the logic that might throw an exception, from the logic to handle it.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
}
```

If an exception is thrown in the code inside of the try statement, the code stops in that line, the catch clauses attempt to catch it. If no exception is thrown, the catch clause won't run. The *exception_type* is the type you are trying to catch, and the identifier refers to the caught exception object. Curly braces are required, even if there is only one statement in the block.

Adding a finally block: Java lets you add a *finally* clause that will execute whether or not an exception is thrown. Note: catch is not required if finally is present.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
} finally {
        // finally block
}
```

Catching various types of exceptions.

Custom exceptions are out of scope in OCA, but necessary to recognize them. Examples:

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

All of them are unchecked, as they directly or indirectly extend from RuntimeException. See more examples of how to catch multiple exceptions on CatchingMultipleExceptions.java.

Throwing a second exception.

Inside a catch or finally clause, any valid java code is acceptable, even another try and catch block. This is how you can throw a second exception.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
} finally {
        try { // try block or protected code
        } catch (exception_type identifier) {
              // exception handler
        }
}
```

See SecondExceptionSample1.java and SecondExceptionSample2.java for examples.

Recognize Common Exception Types.

Runtime Exceptions.

They extend from *RuntimeException*. They don't have to be handled or declared and can be thrown by programmer or the JVM.

- ***ArithmeticException***. Thrown by the JVM when code attempts to divide by zero.
- ***ArrayIndexOutOfBoundsException***. Thrown by the JVM when code uses an illegal index to access an array.
- ***ClassCastException***. Thrown by the JVM when an attempt is made to cast an object to a subclass of which it is not an instance. E.g.
```
String type = moose;
Object obj = type;
Integer number = (Integer) obj; // Throws ClassCastException
```
- ***IllegalArgumentException***. Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument. E.g.
```
public void setNumberEggs(int numberEggs) {
      if (numberEggs < 0)
         throw new IllegalArgumentException("# eggs must not be negative");
      this.numberEggs = numberEggs;
}
```
- ***NullPointerException***. Thrown by the JVM when there is a null reference where an object is required.
- ***NumberFormatException***. Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format. It's a subclass of *IllegalArgumentException*.

Checked Exceptions.

They extend from *Exception*. They must be handled or declared and can be thrown by the programmer or the JVM.

- ***FileNotFoundException***. Thrown programmatically when code tried to reference a file that does not exist.
- ***IOException***. Thrown programmatically when there's a problem reading or writing a file.

Errors.

They extend the Error class. Thrown only by the JVM and should not be handled or declared. They are rare.

- ***ExceptionInInitializerError***. Thrown by the JVM when a static initializer throws an exception and doesn't handle it. Java fails to load the whole class.
```
static {
int[] countsOfMoose = new int[3];
int num = countsOfMoose[-1]; } // Throws java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
```
- ***StackOverflowError***. Thrown by JVM when a method calls itself too many times (infinite recursion). The Stacks gets full / overflows.
- ***NoClassDefFoundError***. Thrown by JVM when a class that the code uses is available at compile time but not runtime.

Calling Methods That Throw Exceptions.

When calling a method that may throw a checked exception, the exception must be handled / declared in the method where the call is made. See Bunny.java and Bunny2.java classes for examples.

Subclasses.

- When a class overrides a method from a superclass or implements a method form an interface, it's not allowed to add new checked exceptions to the method signature. Example:

```
class CanNotHopException extends Exception { }
class Hopper {
     public void hop() { } }
class Bunny extends Hopper {
     public void hop() throws CanNotHopException { } // Does not compile
}
```

- A subclass is allowed, thought, to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them. Example:

```
class Hopper {
     public void hop() throws CanNotHopException { }  }
class Bunny extends Hopper {
     public void hop() { }
}
```

- A class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of the broader type. Example:

```
class Hopper {
     public void hop() throws Exception { }  }
class Bunny extends Hopper {
     public void hop() throws CanNotHopException { }  }
```

All these rules apply to checked exceptions only. In the first example, declaring a runtime exception in the Bunny class would be perfectly legal.

Printing an Exception.

Three ways: you let Java print it, print just the message or print where the stack trace comes from. The first one prints the default: exception type and message. Second option prints message. The last one prints the stack trace, showing where the exception occurred in each method that it passed through. See PrintingAnException.java for examples.