```
/********   ********/
System.out.println("Hello World");
System.out.println(6+6);

/* declare, initialize */
// 8 primitive types - int, short, long, byte, float, double, char, boolean
// int ranges from -2B to 2B
int maxInt = 2147483647;
Integer integerNumber = null;
// Almost always use double but not float
float floatNumber = 7.7f;
// Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, Double.NaN
double div = 5/2.0;    // forward slash; ans = 2.5
// Casting a variable
int divInt = (int) div    // answer is 2

if (Double.isNaN(Double.valueOf("NaN"))) {
    System.out.println("NaN");
}

// Conversion between int and Integer is automatic (boxing/unboxing)
// Caution: == doesn't work with wrappers
// Caution: wrappers can be null; int can never be null but Integer can
Integer x = 10;
Integer y = 10;
if (x.intValue() == y.intValue()) {
    System.out.println("values are the same");
}
if (x.equals(y)) {
    System.out.println("using Objects.equals() to check values");
}

/* Using lower camel case while creating variable names */
String myString = "double quotation mark"    // char[]
char mychar = 'a';
/*
    .length()
    .toUpperCase()
*/

// String is immutable; can reassign string variable
// myString.equals();    // Never use == (== checks whether they are identical
objects (same memory location))
// myString.equals(); can only be used when myString isn't null; solution is to
import java.util.Objects; and use Objects.equals(myString, "pattern");
// "string".equalsIgnoreCase();
// Empty string and null are difference
// myString != null && myString.isEmpty()
/*
    .trim()
    .toLowerCase()
    .indexOf
    .lastIndexOf()
    .substring()    // Caution: some unicode characters requires more than one char
value
*/
// Concatenation (+) strings: if one operand isn't a string, it's turned into a
string
int age = 13;
String rating = "PG" + age;
// Easy way to convert everything to String is to concatenate with an empty string
""

boolean myboolean = true;    // false

/*
boolean expressions

relational operators:
!=    // not equal

boolean operators:
&&, ||, !

bitwise operators:
&, |, ^, ~, >>, >>>, <<

conditional operator:
x < y ? x : y    // if x < y, then value is x, otherwise value is y

order of operation: parentheses, Not, AND, OR
*/

// {variable scope} : used inside {} only
enum newType { SMALL, LARGE };    // Can hold null value
// Enumeration class: all enumeration classes are subclasses of Enum and inherit
methods: toString and ordinal
/* useful static methods
    Enum.valueOf(Size.class, "SMALL") yields Size.SMALL
    Size.values() yields all values in an array of type Size[]
*/
// enum can use == for comparison
public enum Size{
    SMALL("S"), MEDIUM("M");

    private String abbreviation;

    private Size(String abbreviation) {
        this.abbreviation = abbreviation;
    }
    public String getAbbreviation() {
        return abbreviation;
    }
}

int[] newArray;    // array variable declaration
int[] newArray = new int[100];    // creation of array
int[] newArray = { 2, 3, 4, 5 };
newArray.length    // Check length of an array, no parentheses

int[] newArray2 = newArray;    // Copy array
newArray2[1] = 100;    // newArray[1] now is also 100
// Array variable contains reference to the array where it stores in memory. While
copying, only copy reference
// Use Arrays.copyOf to make a true copy
// Use Arrays.sort(myArray) to sort an array
// Multi-dimensional arrays; [row][column]
int[][] squareArray = {
    {1,2,3},
    {4,5,6},
    {7,8,9}
};


/* ##################

if (myCondition) {
    myAction;
} else if ( ) {
    myAction;
} else {
    myAction;
}

################## */

/* ##################

switch (myChoice) {
    case myCase:
        myAction;
        break;
    case myCase1:
    case myCase2:
    case myCase3:
        myAction;
        break;
    default : myAction;
        break;
```

```
}

################## */

/* ##################

while (myCondition) {
    myAction;
    myCondition += 1;
}

// do/while statement    // Condition is at the end
################## */

/* ##################

int breakPoint = 5;
ArrayList myArrayList = new ArrayList();

for (int i = 0; i < 25; i++) {
    myArrayList.add(i);
}

for (int i = 0; i < 20; i++) {
    if (i > breakPoint) {
        System.out.println("break");
        break;
    }
    System.out.println("break for loop " + i);
}

// for each loop
// myArray is collection which will be iterated
for (int element : myArray) {
    System.out.println(element);
}

for (Object obj : myArrayList) {
    System.out.println("iterate (another way to loop) " + obj);
}

################## */

/* ##################

public void myFunction(String myParameter, int myParameter2) {
    // access modifier: public, private
    // return type examples: void, int, long, double, String, char, boolean
}
```

```
// function call
myFunction("try", 3);

public int myFuncCount(String myParameter, int myParameter2) {
    return countLike;
}

// function definition
// Class is a group of variables, functions, etc..
// method is the same thing as function; object.method(parameters);

// method takes a variable number of parameters
public class PrintStream{
    public PrintStream printf(String fmt, Object... args) {
        System.out.println("use ... for various parameters");
    }
}

################## */

/* ##################

try {

} catch (Exception e) {

}

################## */

// break & continue
// continue will skip the rest of loop body and continue the next record

/* cell index of array starts from zero. myarray[0] */
// define an integer array
int [] arrayNum = {12,1,-100,0,12};
String [] arrayStr = {"CNN","CBS"};

// 2-D array
int [][] myGrades;    //[row][column]


/* install JDK; compile & run java code on Windows commend (edit environment
variable) */
>> javac myCode.java
>> java myCode
```

```
/********   ********/
/**
 * lowerCarmelCase: object, field, method, package
 * UpperCarmelCase: Class, Constructor, Interfaces
 * UPPER_CASE: constants
 */

/* object, field, method */
// Object holds data which describes itself
// Object has fields and can perform actions using methods
// Fields store the object's data while methods perform actions to use or modify
those data
// Object combines variables together to make your code meaningful, easier to
understand, and simplier to maintain

/* field, attribute, member variable */
// Fields of an object are all the data variables that make up that object. They are
also sometimes referred to as attributes or member variables
// Accessing a field in an object is done using dot modifier
/*
A book object contains:
String title;
String author;
int numOfPages;

book.title;    // access title field
*/

/* method */
// Methods in Java are functions that belong to a particular object
// Calling a function is done using dot modifier
/*
A book object has a method
void setBookmark(int pageNum);

book.setBookmark(12);    // call a method
*/

/* class */
// Class is the blueprint that defines what an object should look like
// Object is actual entities that is created from a class
// Object is instance of class
// Each class should be created in its own file (.java)

/* Main.java */
public class Main {
    /* the main method */
    public static void main(String [] args){
        System.out.println("My Java Hello World");
    }
}


/* constructors */
// Constructors are special types of methods that are responsible for creating and
initializing an object of that class - Like method, except that 1. no any return
types & 2. have the same name as the class itself
// Constructor is for initializing instance fields which in general are private
// A class can have more than one constructor (multiple constructors are being
called).  In this situation, the constructor name is overloaded.  While overloading,
the compiler picks the appropriate version from the argument types.  return type
isn't used in overloading resolution because it isn't part of the method signature
(name + parameter type).
// User can also overload method
// A field that isn't explicitly set in a constructor is 0, false, or null.  This is
different from local variables.  It's an error if local variables are not
initialized.
// Accidentally uninitialized variables can lead to null pointer errors.

/* creating a class */
public class Game{
    private int mScore;

    // Default constructor
    public Game(){
        mScore = 0;
    }
}

/* creating a class */
public class Game{
    private int mScore;

    // Default constructor allows empty class declariation
    public Game(){}

    // Constructor by starting score value
    public Game(int startingScore){
        mScore = startingScore;
    }
}

/* access constructors */
// Unlike normal methods, constructors cannot be called using the dot . modifier
// Instead, every time you creat an object variable of a class type, the appropriate
constructor is called
// To create an object of a certain class, you will need to use the new keyword
followed by the constructor you want to use
```

```
Game tetris = new Game();    // create an object called tetris using the default
constructor (initial mScore = 0)
Game darts = new Game(501);    // initialized with a different starting score
Game darts = null;    // null keyword; uninitialized object; null object has no
fields & methods

/* self reference */
// Sometimes you'll need to refer to an object within one of its methods or
constructors, to do so you can use the keyword this
// this is a reference to the current object - the object whose method or
constructor is being called
// You can refer to any field of the current object from within a method or
constructor by using this
public class Position {
    private int row = 0;
    private int column = 0;

    // constructor
    public Position(int row, int column) {
        this.row = row;    // refer to the field named row rather than the input
parameter
        this.column = column;
    }
}

/* creating a class */
public class Contact{
    public String name;    // better use private
    private String email;
    public String phoneNumber;    // better use private
}

/* creating a class */
public class ContactsManager {
    // fields
    private Contact [] myFriends;    // object variable
    private int friendsCount;    // primitive variable

    // Constructor
    public ContactsManager(){
        this.myFriends = new Contact[500];
        this.friendsCount = 0;
    }

    // methods
    public void addContact(Contact myContact){
        myFriends[friendsCount] = myContact;    // ???when to use this. & when not
to???
        friendsCount++;
    }
    public Contact searchContact(String searchName){
        for(int i = 0; i < friendsCount; i++){
            if(myFriends[i].name.equals(searchName)){
                return myFriends[i];
            }
        }
        return null;  // return Contact object if a match found, null if not match
    }
}

/* Main.java */
public class Main{
    // the main function
    public static void main(String [] args){
        // create the ContactsManager object
        ContactsManager myContactsManager = new ContactsManager();

        // create a new contact object for James
        Contact friendJames = new Contact();
        // set the fields
        friendJames.name = "James";    // better use setter
        friendJames.phoneNumber = "111-111-1111";
        // add James to ContactsManager
        myContactsManager.addContact(friendJames);

        // create a new contact object for Chris
        Contact friendChris = new Contact();
        // set the fields
        friendChris.name = "Chris";
        friendChris.phoneNumber = "222-222-2222";
        // add Chris to ContactsManager
        myContactsManager.addContact(friendChris);

        // search a contact and print phone number
        Contact result = myContactsManager.searchContact("Chris");
        System.out.println(result.phoneNumber);
    }
}


/* Using correct access modifier */
// private/public for field/method/class
// field
// private can only be used inside the class that created it.  It's strongly
recommended in java to label all fields as private
// A better design would be to declare a field that can be modified by other classes
as private and then create public methods that return the value of such hidden field
(known as getters) as well as public methods that provide a way to set of change its
value (known as setters)
```

```
// method
// private methods are usually known as helper methods (for organizing code and
keeping it simple and readable)
// public methods are the actual actions that the class can perform and are pretty
much what the rest of the program can see and call
// set methods to private that are considered helper methods
// set methods to public that are considered actions
// set all classes to public


public class Book{
    // always try to declare all fields as private
    private String title;
    private String author;
    private boolean isBorrowed;

    // create a constructor that accepts those private fields as inputs
    public Book(String title, String auther){
        this.title = title;
        this.author = author;
    }

    // create a public method that returns each private field; getter method
    public boolean isBookBorrowed(){
        return isBorrowed;
    }

    // create public a method to set each private field; setter method
    public void borrowBook(){
        isBorrowed = True;
    }

    public void returnBook(){
        isBorrowed = False;
    }
}


/* encapsulation */
// Each class is just like a capsule that contains everything it needs and nothing
more
// Encapsulation: only methods can access object data

/* inheritance */
// Passing down traits or characteristics from a parent to their child
// Classes can not only use other classes but can also inherit from them and extend
their capibilities

// parent class
class BankAccount {
    String acctNumber;
    double balance;
}
// child class
class Checking extends BankAccount {
    double limits;
}

/* polymorphism */
// multiple shapes & forms; polymorphism defines how Java objects can have multiple
identities
// A variable can refer to multiple types
// A class could be treated as if it is its parent as well as its own type
// BankAccout myBankAccount = new Checking()  // using parent class and child
constructor

/* override */
// When a class extends another class, all public methods declared in that parent
class are automatically included in the child class without you doing anything
// However, you are allowed to override any of those methods.  Overriding basically
mean re-declaring them in the child class and then re-defining what they should do
// Argument types of overriding method must match exactly; use @Override annotation
to make the compiler check
// Return type can be covariant as long as its a sub type
// If the method is private, static, or final then the compiler knows exactly which
method to call (static binding).  Otherwise, the exact method is found at runtime
(dynamic binding - the appropriate method is selected)

/* super */
// Using the keyword super means that we want to run the actual method in the super
(or parent) class from inside the implementation in "this" class
// Subclass constructor can invoke superclass constructor; use super() to call the
parent's constructor; this is usually done when implementing the child's
constructor; super(filed1, list of constructor parameters); call using super must be
the first statement
// Subclass methods cannot access private superclass fields; super.method()

// A class can only extend one single class; a class can only has one parent

/* interfaces */
// Interfaces define what a class should do but not how to do it
// An interface's solo purpose is to be implemented by one or more classes
// You can not create an instance (object) from an interface

// Movable.java
public interface Movable{
    void move(int distance);
    boolean canMove();
}
// Habitable.java
```

```
public interface Habitable{
    boolean canFit(int inhabitants);
}
// Caravan.java
public class Caravan implements Habitable, Movable{
    void move(int distance){
        location = location + distance;
    }
    boolean canMove(){

    }
    boolean canFit(int inhabitants){
        return max <= inhabitants;
    }
}

/* comparable interface */
// This interface includes a single method definition called compareTo
public class Book implements Comparable<Book>{

}

/* final fields/methods  */
// field
// A final field has nothing to do with inheritance
// A final field is simply a constant variable and not allowed to be changed
public class MathLib{
    public final double PI = 3.14;
}
// method
// If you want to protect your method from being overriden in a child class you can
prefix it with the keyword final

// final method & class: only use it for design not necessary for performance
// A final class cannot be extended (String.class)
// The instanceof operator compares an object to a specified type. You can use it to
test if an object is an instance of a class, an instance of a subclass, or an
instance of a class that implements a particular interface

/* static fields/methods */
// field
// Objects that are created from a class don't last forever
// Declaring a field as static means that these values are no longer stored within
the object itself but within the class instead
// If that value changes, it will update it in every single object of that class
again
// Java allows you to access a static field directly from the class instead of
having to create an object of that
// static means share
// method
// Just like static fields, static methods also belong to the class rather than on
object
// It's ideally used to create a method that doesn't need to access any fields in
the object, in other words, a method that is a standalone function
// A static method takes input arguments and returns a result based only on those
input values and nothing else
// A static method doesn't use "this" and doesn't operate on objects

public class Calculator {
    public static int add(int a, int b) {
        return a + b;
    }
    public static int subtract(int a, int b) {
        return a - b;
    }
}
// You can call them directly using the class name Calculator without the need to
create an object variable at all
// Calculator.add(10,5);


/* array, collections */
/* array */
// store multiple items of the same type
// limitations: 1. have to know exact number of items will use while initializing an
array 2. not allow to add or remove cells after being created
String [] names;

/* collections */
// Collections are a bunch of different classes and interfaces Java offers you to
use to simplify dealing with multiple items of the same type
// List, Stack, Map, Queue
// ArrayLists is the most common type of lists

/* List */
// A List in Java is an interface that behaves very similar to an array
// - It's an ordered collection (also known as a sequence)
// - The user of this interface has precise control over where each item is inserted
in the list
// - The user can access items by their integer index (position in the list)
// - The user can search for items in the list by looping over the items in it

/* ArrayList */
// An ArrayList is a class that implements the interface List.  It's simply a
wrapper around an array
// An item in an ArrayList is known as an element
// ArrayList can only hold objects not primitive types such as int values
// An object of the Integer wrapper class wraps an int value
/*
    Some methods of ArrayList
    - add(E element)
    - add(int index, E element)
    - get(int index)
    - set(int index, E element)
    - contains(Object o)
    - remove(int index)
    - size()
    - indexOf(Object o)    // return -1 if this list doesn't contain the element
*/
ArrayList myGrades = new ArrayList();
myGrades.add(100);
myGrades.get(0);    // access the first item in the List
System.out.println(myGrades.size().toString());    // ??? IntelliJ throws error ???
myGrades.remove(0);
myGrades.clear()

/* Stacks */
// The Stack collection represents a last-in-first-out (LIFO) stack of objects
/*
    Stack class has 5 methods
    - push(E item);  // add element to top of this stack
    - pop();    // return type is Object and might need to cast it to String, int,
etc.. yourself
    - peak();
    - empty();
    - search (Oject o);
*/

/* Queue */
// Queue is another common collection type used in Java and it's a
First-In-First-Out (FIFO) data type
// Queue is an interface and defines two methods
/*
    - add(E element)
    - poll()
*/
// A special type of Queue is known as Deque which is a double-ended queue.  Meaning
that you can add/remove elements from either end of a Deque
/*
    - addFirst(E element)
    - addLast(E element)
    - pollFirst()
    - pollLast()
*/
// Java also provides a few classes that implement the Queue interface, perhaps the
most popular of all is the LinkedinList
Queue orders = new LinkedList();
orders.add("order1");

/* Generics */
```

```
// In a nutshell, Generics enable classes to accept parameters when defining them,
much like the more familiar parameters used in method declarations
// For example: ArrayLists use Generics to allow you to specify the data type of the
elements you are intending to add into that ArrayList
ArrayList<String> listOfStrings = new ArrayList();
// Can omit type parameter in the constructor (diamond syntax)
ArrayList<Integer> arrayList = new ArrayList<>();
// Compiler will show an error, if you try to insert wrong type
// Generics eliminate the need for casting; for example, you can specify data type
of stack or queue or any collection when declaring it, thus eliminating the need to
case any return types
List list = new ArrayList();
list.add("hello");
String string = (String) list.get(0);

List<String> list = new ArrayList<String>();
list.add("hello");
String string = list.get(0);    // no cast
// user can also define your own Generic types

/* Collection & Polymorphism */
// Both video and audio classes are subclasses of media class
ArrayList<Media> playList = newArrayList();
Video someVideo = new Video();
Audio someAudio = new Audio();
playList.add(someVideo);
playList.add(someAudio);
// retrieve item from playList
Media media = playList.get(0);    // now have flexible to retrieve both video &
audio types
media.play();    // common method

Video movie = (Video) playList.get(0);
movie.playFullScreen();    # allow you to access more specific methods only in child
class

/* HashMaps */
import java.util.HashMap;
// The reason it is important to use the correct data structure for a variable or a
collection is performance
// A HashMap is another type of collection that was introduced in Java to speed up
the search process in an ArrayList
// HashMaps allow you to store a key for every item you want to add.  This key has
to be unique for the entire list and can be any Object of any Type
// The point is to be able to find an item in this collection instantly without
having to loop through all the items inside and hence save that precious run-time

// book class
public class Book{
    String ISBN;
    String title;
}
public class Library{
    HashMap<String, Book> allBooks;
}
// initialize hash map
allBooks = new HashMap<String, Book>();
// add a book
Book taleOfTwoCities = new Book();
allBooks.put("uniqueISBN", taleOfTwoCities);
// search a book
Book findBookByISBN(String isbn) {
    Book book = allBook.get(isbn);
    return book;
}
```

```
/********   ********/

JVM executes Java programs
distributed (code can be moved to different platforms)

$ javac -version
$ echo $PATH

/* parameter passing in Java */
// Call by value: the method gets copies of argument values
// A method cannot change the contents of variables passed to it; method cannot
directly modify its variables
// In Java, everything is passed by value; Object references are passed by value

// modules are collections of packages

// class design hints
// Always keep data private & initialize data
// Don't use too many basic types in a class (group them to another element)
// Not all fields need field accessors and mutators (getters/setters)
// Break up classes that have too many responsiblilties
// Make names of classes and methods reflect their responsibilities
// Prefer immutable classes if possible (example: calendar dates)

// subclass (child) & superclass (parent)

/* inheritance */
/* Abstract Methods and Classes */
```

```
// Abstract Methods
// When factoring out common classes, it can become difficult to implement methods
in the most general classes.
// Declare method as abstract and don't provide implementation: public abstract
String getDescription();

// Abstract Classes
// Class with abstract methods must be declared abstract
// Ok for abstract classes to have fields, constructors, and concrete methods
// Abstract classes cannot be instantiated
// A class can be declared abstract even if it has no abstract methods
public abstract class Person {
    private String name;
    public String getName() {
        return name;
    }
}


// public, private, package visible, protected
// protected fields: available for all sub-classes

// Object is superclass of all Java classes; only primitive types are not objects
// Object class has useful methods - equals, hashCode, toString
// Object.equals tests whether the object reference are identical
// Not allow to invoke any method including equals on a null reference
// Static Objects.equals method is null safe
// hash code is an integer derived from an object
// hash code should be scrambled and it must be consistent - if x and y are equal,
then their hash codes must be equal; hash collision is rare
// Override hashCode whenever you override equals!    // Objects.hash();
// toString yields a string representation of an object
// Object.toString yields class name and hash code

/* use reflection to work with arbitrary objects */
// Reflection: analyze and manipulate arbitrary Java objects at runtime
```

```
/*
* Core Java SE 9 for the Impatient, Second Edition
* by Cay S. Horstmann
*/
// 4.5 Reflection
// Reflection allows a program to inspect the contents of objects at runtime and to
invoke arbitrary methods on them

// 4.5.2

// 5
throw new RuntimeException(e);    // throw statement is used to “throw” an object of
a class
/*
 * RuntimeException
 * IllegalArgumentException
 * NumberFormatException
 */
```

```
/********   ********/
/* Set */

/* Serialization and deserialization in Java */
Serialization is a mechanism of converting the state of an object into a byte
stream.  Deserialization is the reverse process where the byte stream is used to
recreate the actual Java object in memory.  This mechanism is used to persist the
object.

/* <T> <E>*/
ArrayList<T>    (template)
ArrayList<Employee> arrayList = new ArrayList<>();
```

```
/********   ********/
/* ?????????? */
???What is comparable interface? <> related questions?  What is <>?
???public class Book implements Comparable<Book>{
???}
???is true equal to one?
???array[-1] will work or not?
???2-D array.length? row or column?
??? string comparison has to use a function .equals since they are not the same
String object (exercise in chapter 4)
```