Java Basic:

- the class must be the name of the java source and Java byte code(___.java, and ___.class)
- a class can define multiple components.
    - package statement
    - import statement
    - comments
    - class declarations and definitions
    - variables
    - methods
    - constructors
- All Java classes are part of a package
    - A Java class can be explicitly defined in a named package
        - otherwise becomes part of a default package (doesn't have a name)
    - explicitly defines package a class in
    - a package statement cannot appear within a class declaration or after the class declaration
        - MUST be before the class declaration
- classes and interfaces in the same package can use each other without prefixing their names with the package
    - to use a class or interface from another package, must use its fully qualified name or use the import statement
    - import statement follows package statement, but is before the class declaration
- you can add comments anywhere in the code
    - multiple line comments start with /* and end with */
    - usually, uses an asterisk(*) to start the comment in the next line, but isn't required
    - End-of-line comments start with //
        - can write comments just on the end of that line
    - comments can proceed packages
- class declaration marks the start of a class
    - class declaration composed of:
        - access modifiers
        - non-access modifiers
        - Class name
        - Name of the superclass (with extend statement)
        - all implemented interfaces (with implement statement)
        - class body, included within a pair of curly braces ({})
- a class is a design used to specify the properties and behavior of an object
    - the properties of an object are implemented using variables
    - the behavior is implemented using methods
- a class constructor is used to create and initialize the objects of a class
    - a class can define multiple constructors that accept different sets of method parameters

Java Source code file

- Java source code is used to define classes and interfaces
- an interface is a grouping of related methods and constants, but the methods are abstract and CANNOT define any implementation

    - interface starts with the keyword interface
- you can define either a single class or interface, or multiple classes or interfaces in one source code
- there is no required order for the multiple classes or interfaces defined in a single Java source code
- a source code can only have one public class or interface
    - and the name of the source code should be the name of the file
- **GUPTA EXAM TIP:** classes and interfaces defined in the same Java source code CAN'T be defined in separate packages. Classes and interfaces imported using the import statement are available to all the classes and interfaces defined in the same Java source code file

Executable Java applications

- a class which starts its execution at a particular point in the class with the main method defined in the class
    - starts when handed over to the JVM
- CANNOT hand over a non-executable Java class to the JVM
- typically an application consists of a number of classes and interfaces that are defined in multiple Java source code files
- the first requirement in creating an executable Java application is to create a class with a method whose signature matches the main method
    - the main method must be
        - marked public, static
        - named main
        - return type void
        - method parameters must have a String array or a variable argument of type String
            - ellipsis(...)
                - has to go after the data type not the variable name
            - the variable name can be anything, but is usually args
        - doesn't have to public static void main(String args[])
            - can be static public void main(String args[])
- JVM calls main NOT the compiler
- **GUPTA EXAM TIP:** the method parameters that are passed on the main method are also called command-line values. As the name implies, these values are passed on to a method from the command line
- a final non-access modifier is an acceptable standard modifier
- making a static method final prevent s a subclass from implementing the same static method
- main can be anywhere as long as the public class matches the file name

Java Packages

- use packages to group related set of classes and interfaces
- also provide access protection and namespace management
- packaged classes are part of a named package
    - defined by a package statement in a class
- All classes and interfaces are packaged
    - if don't include a package, it's part of a default package
- common for organizations use subpackages to define all their classes

- a fully qualified name for a class or interface is formed by prefixing its package name with its name (separated by a period)
- the hierarchy of the packaged classes should match the hierarchy of the directories in which these classes and interfaces are defined
- import statement lets you use the simple names of classes instead of the fully qualified names
    - for example: instead of java.util.Scanner, you can put an import java.util.* or java.util.Scanner and just say Scanner in your code
    - the import statement doesn't embed the contents of the imported class in your class
    - it just tells the JVM where to find it
    - You CAN'T use the import statement to access multiple classes or interfaces with the same name from different packages
        - have to use the fully qualified names
    - you can use the asterisk to import all of the public members, classes, and interfaces of a package
    - can't import classes from a subpackage by using an asterisk in the import statement
- default package automatically imported in the Java classes and interfaces defined within the same directory of system
    - can't be used in any named packaged class
- can import static members of a class by using import static statement

Java access modifiers

- public(least restrictive)
    - can be accessible across all packages, from derived to unrelated classes
- protected
    - can be accessible to classes and interfaces defined in the same package
    - all subclasses, even if they're defined in separate packages
    - **GUPTA EXAM TIP:** a subclass in a separate package can't access protected members of its superclass using reference variables
- default
    - members of a class defined without using any explicit access modifier
    - only accessible to classes and interfaces defined in the same package
    - GUPTA EXAM TIP: default access can be compared to package-private, and protected access can be compared to package-private + kids(kids refer to subclasses).
- private(most restrictive)
    - only accessible to the class
    - not accessible outside the class they are defined
    - cannot put a method argument as private
- the top-level class can be defined only using the public or default access modifiers
- top-level class is a class that isn't defined within any other class
    - a class that is defined within another class is called an inner class
- can be applied to classes, interfaces, instance and class variables, and methods
- if a class, interface, method, or variable isn't explicitly defined using an explicit access modifier, it is said to be defined using the default access.
- control accessibility of your class and members outside the class and package

Java non-access modifiers

- change the default properties of a class and its members

- static
- final
- abstract
- synchronized
- native
- strictfp
- transient
- volatile
- the test only covers abstract, final, and static

Abstract modifier

- can change a class, so that it can't be instantiated, even if it doesn't have any abstract methods
    - but a concrete class cannot have any abstract methods
- an interface is abstract by default
    - the compiler automatically puts the abstract keyword to the definition of an interface
- abstract methods don't have a body
    - usually implemented by a subclass
- no variable can be abstract
- **GUPTA EXAM TIP:** Don't be tricked by code that tries to apply the abstract to a variable. The code won't compile

Final modifier

- can change the behavior of a class, variable, or method
- a final class CANNOT be extended by another class
- an interface CANNOT be marked final.
    - it is abstract by default, and your code won't compile
    - but its variables have to be marked as final static
- a final variable CANNOT be reassigned to another value
    - it can be assigned ONLY ONCE
    - if a reference variable is defined as a final variable, you CANNOT reassign another object to it, but you CAN call methods.
    - must initialize final variables at the object's time of creation
- a final method in a superclass CANNOT be overridden by a subclass

Static modifier

- changes default behavior to variables, methods, and interfaces
- static variables belong to a class
    - NOT an object
    - exist independently
    - may be accessed when no instances of the class have been created
    - shared by ALL of the objects of a class
    - **GUPTA EXAM TIP:** a static variable can be accessed using the name of the object reference variable or the name of a class
- static and final non-access modifiers can be used to define constants(variables whose value can't be changed).
- static methods aren't associated with objects
    - can't use any instance variables of a class
    - can define static methods to access or manipulate static variables

    - can use to manipulate method parameters to compute and return an appropriate value
- non-private static variables and methods are inherited by subclasses
- CANNOT override static methods
    - not involved with polymorphism
    - but can redefine the method
- static methods or variables CANNOT access non-static methods or variables in the class
    - but non-static variables or methods can access static variables or methods
- static classes and interfaces are types of inner classes
    - not covered on the exam
- You can't prefix the static definition to a top-level class or an interface
    - not defined within another class or interface

Working with data types

- Java is a strongly typed language.
- MUST declare a variable and define its type BEFORE you can assign a value to it
- a literal is a fixed value that doesn't need further calculations in order for it to be assigned to any variable
- primitive variables are variables defined with one of the primitive data types
- primitive data types are the simplest data types in a programming language
- **GUPTA EXAM TIP:** Watch out for questions that use incorrect names for primitive data types. Remember only int and char are shortened, the rest of the primitive data types ARE NOT!
- Eight primitive data types in Java language:

bits	Integer	Decimal	Character	Boolean
8 bits	byte			boolean (either true or false no bit)
16 bits	short		char	
32 bits	int	float		
64 bits	long	double		

boolean values

- can only be true or false
- **GUPTA EXAM TIP:** the questions tests your ability to select suitable data type for a condition that can only have two states: yes/no or true/false
- true/false are the only two boolean literals

Numeric values

- defines two subcategories: integers and floating point(decimals)
  Integers:
    - hold WHOLE numbers
    - includes negative and positive

Data Type	Size(bitwise)	Range of values
byte	8 bits (one byte)	-128 to 127, inclusive
short	16 bits (2 bytes)	-32,768 to 32,767, inclusive
int	32 bits(4 bytes)	-2,147,483,648 to 2,147,483,647, inclusive
long	64 bits(8 bytes)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive

- Integer literals values come in four flavors:
    - binary
        - a base-2 system, which uses a 0 and 1
        - use prefix 0B or 0b
    - octal
        - a base-8 system, use 0-7.
        - The decimal number 8 is represented as octal 10, decimal 9 as 11, and so on
        - use prefix 0
    - decimal
    - the base-10 number system that is used everyday.
    - it's based on 10 digits, 0-9
    - hexadecimal
        - a base-16 system, uses digits 0-9 and letters A through F (a total of 16 digits and letters)
        - the number 10 is represented as A, 11 as B, 12 as C, 13 as D, 14 as E, 15 as F
            - can use lowercase a,b,c,d,e,f
        - 0x or 0X
- You can assign integer literals in base decimal, binary, octal, and hexadecimal
- As of Java 7, you can also use underscores as part of literal values, which makes literal values more readable
    - can't start or end a literal value with an underscore
    - can't place an underscore right after the binary or hexadecimal prefixes
        - but can place an underscore right after the octal prefix
    - cannot place an underscore prior to a long suffix
    - cannot use an underscore in positions where a string of digits is expected
        - when parsing an integer it will compile because a String will accept underscores, but will fail at runtime and throw an exception
    - can put as many underscores as you want as long as you follow the rules

Floating-Point numbers

- decimal numbers
- default type is a double
    - can add a d or D, but is redundant
- have to suffix an f or F for floats

- as of Java 7, can use underscores with floating point literals
    - cannot place an underscore prior to a suffix
    - cannot place an underscore next to a decimal point

Data Type	Size	suffix
float	32 bits(4 bytes)	F or f
double	64 bits (8 bytes)	D or d or nothing at all

Character values

- only char data type
- can store 16 bit unicode-character
- can store virtually all the existing scripts and languages
- char use single quote (‘’)
    - common mistake using double quotes to assign a value to a char
    - **GUPTA EXAM TIP:** Never use double quotes to assign a letter to a char variable. Double quotes are used to assign a value to a String variable
- can assign a positive integer literal value
    - but integer value is NOT equal to unicode value
    - unicode is a number in base 16
- \u is used to mark unicode values
- can cast a negative integer value to a char to get the char value
    - can cast a char into an int and vice versa.
    - can put a char into an int, but CANNOT put an int into a char WITHOUT casting
- **GUPTA EXAM TIP:** the exam will test your understanding of the possible values that can be assigned to a variable of type char, including whether an assignment will result in a compilation error

Identifiers

- names of packages, classes, interfaces, methods, and variables
- valid identifiers follow
    - unlimited length
    - starts with a letter, a currency sign, or an underscore
    - can use a digit(but not at the starting point)
    - can use underscore
    - can use currency signs
- there are Java keywords you CANNOT USE:

abstract	default	goto	package	this
assert	do	if	private	throws
boolean	double	implements	protected	transient
break	else	import	public	true
byte	enum	instanceof	return	try

case	extends	int	short	void
catch	false	interface	static	volatile
class	final	long	strictfp	while
const	finally	native	super	
continue	float	new	switch	
char	for	null	synchronized	

- variables can be categorized into two types
  - primitive variables
  - reference variables

Reference variables
- objects are instances of classes
- object reference is a memory address that points to a memory area where an object's data is located
- object reference variable is like a handle to an object that allows access to the object's attributes
- if a reference variable is equal to null, then the object is eligible for garbage collection
- if the reference variable points to another object, AND no other reference variable points to the object, it is eligible for garbage collection
- the literal value for ALL reference variables is null

Primitive variables
- store values

Using operators and decision constructs:

- four different types of operators:
  - assignment
  - arithmetic
  - relational
  - logical

Assignment operators
- simplest assignment operator is =
- +=, -=, *=, and /= are shortened forms of addition, subtraction, multiplication and division with assignment
  - += is like a = a+b
  - -= is like a = a-b
  - *= is like a = a*b
  - /= is like a = a/b
- **GUPTA EXAM TIP:** You can't use the assignment operators to assign a boolean value to variables of type char, byte, int, short, long, float, or double, or vise versa

Arithmetic operators

operator	purpose

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Unary increment operator, or increments value by 1
--	Unary decrement operator, or decrements value by 1

- unary operators can be put before or after the notation

Relational Operators

- used to check one condition
- can use to determine whether a primitive value is equal to another value or whether it is less than or greater than the other value
- can be divided into two categories:
  - comparing greater values (>. >=) and lesser values(<, <=)
  - comparing values for equality(==) and non-equality(!=)
- greater and lesser operators can work with all types of numbers(including char)
- **GUPTA EXAM TIP:** cannot compare incomparable values, meaning you can't compare a boolean with an int, a char, or a floating-point number. Code WILL NOT compile
- the equality operators (==) and (!=) can be used to compare all types of primitives
  - == returns true if they are equal
  - != returns true if they are not equal
- **GUPTA EXAM TIP:** the result of a rational operation is ALWAYS a boolean value
- common mistake to use the assignment operator in place of the equality operator
  - BE CAREFUL

Logical operators

- used to evaluate one or more expressions
- returns a boolean value
- You can use the logical operators AND, OR, and NOT to check multiple conditions and proceed accordingly
- when both are true, use && operator
- when either are true, use || operator
- when you want the reverse outcome, use ! operator

Operators && (AND)	Operator || (OR)	Operator !(NOT)
true && true → true	true || true → true	!true → false
true && false → false	true || false → true	!false → true
false && true → false	false || true → true	
false && false → false	false || false → false	

- && and || are short-circuit operators
  - because of the way they evaluate their operands to determine the result

- | and & can also be used to manipulate individual bits of a number value

Operator precedence

1. postfix(a++, a--)
2. unary(++a,--a, +a, -a)
3. multiplication, division, modulus
4. addition, subtraction
5. relational
6. equality
7. logical &&
8. logical ||
9. assignment

if-else

- if construct enables to execute a set of statements in your code based on the result of a condition
- must alway evaluate to a boolean value
- **GUPTA EXAM TIP:** In Java, then ISN'T a keyword, so it shouldn't be used with the if statement
- else statements ALWAYS follow an if statement
    - CANNOT put code in between
    - can have an if statement without an else statement
- don't need brackets if only one line of code
- can put an if statement after an else
    - becomes if-else-if
- **GUPTA EXAM TIP:** the if-else-if-else is an if-else construct in which the else part defines another if construct
- the boolean expression used as a condition for the if construct can also include assignment operations
- **GUPTA EXAM TIP:** Watch out for code like misleading identification in if constructs. In the absence of a defined code block (marked with a pair of {}), only the statement following the if construct will be considered to be part of it
    - multiple lines of an if statement MUST be within curly braces
    - it is similar with else statements
- **GUPTA EXAM TIP:** Watch out for code in the exam that uses the assignment operator to compare a boolean value in the if condition. It won't compare the boolean value; it'll assign a value to it.
- a nested if construct is an if construct within another if construct.
    - there is no limit on the levels of nested if and if-else construct
- when nested if-else statement, matching each else with its nearest unmatched if statement

Switch statement

- can use a switch statement to compare the value of a variable with multiple values
- only a single default label.
    - executes when no matching value is found in the case labels
    - a default label can be anywhere.
    - will print until it sees a break in the statement
- a break statement is used to exit a switch, after the code completes
    - can also be put in an if statement to break a loop

- can improve readability of code by replacing a set of related if-else-if-else with a switch statement
- **GUPTA EXAM TIP:** the if-else-if-else construct evaluates all of the conditions until it finds a match. A switch construct compares the argument passed to it with its labels
- can't compare all types of values with a switch

Accepts	Doesn't accept
char	long
byte	float
short	double
int	any other object
enum	
Integer	
Character	
Byte	
Short	
String	

- **GUPTA EXAM TIP:** Watch out for questions in the exam that try to pass a primitive decimal type such as float or double to a switch statement. It won't compile
- the value of a case value MUST BE a compile-time constant value; that is, the value should be known at the time of code compilation
    - you can use variables in an expression if they're marked final because the value of final variables can't change once they're initialized
- Code that tries to compare the variable passed to the switch statement with null won't compile

Creating and using arrays and ArrayList:

Arrays:

- an object that stores a collection of values
- an array is an object itself
- can store two types of data:
    - a collection of primitive data type
    - a collection of objects
- an array of primitives stores a collection of values that constitute the primitive values themselves
    - there are no objects to reference
- an array of objects stores a collection of values
    - are in-fact heap-memory addresses or pointers
- You can define a one-dimensional or multidimensional arrays
    - one-dimensional array is an object that refers to a collection of scalar values

    - a multidimensional array refers to a collection of objects in which each of the objects is a one dimensional array
- the brackets could be on behind the type of the array or the variable name
    - example:
        - int[] anArray
        - int anArray[]
- An array can be
    - primitive data type
    - interface
    - abstract class
    - concrete class
- array allocation will allocate memory for the elements of an array
- it's allocated using the new keyword, followed by the type of value that it stores, and then its size
- **GUPTA EXAM TIP:** once allocated the array elements store their default values. Elements in an array that store objects default null. Elements of an array that store primitive data types store 0 for integer types(byte, short, int, long), 0.0 for decimal types (float, double), false for boolean, or u/000 for char
- you can initialize the array with a loop or you can manually initialize the array
    - you can also initialize an array with curly braces ({})
- multidimensional arrays would use a nested loop
- arrays start at sub 0
    - you can't go past the length - 1
        - otherwise it will throw an ArrayIndexOutOfBoundsException at runtime
- Code to access an array element will fail to compile if you don't pass it a char, byte short or int data type(wrapper classes are not on this exam)
- **GUPTA EXAM TIP:** Code to access an array index will throw a runtime exception if you don't pass it an invalid array index value
- **GUPTA EXAM TIP:** When you combine an array declaration, allocation, and initialization in a single step, you can't specify the size of the array. The size of the array is calculated by the number of values that are assigned to the array
- If the type array is an interface, its elements are either null or objects that implement the relevant interface type
- If the type of an array is an abstract class, its elements are either null or objects of concrete classes that extend the relevant abstract class
- Because all classes extends to the object class, you can put anything in an object array
- array length is an array field
    - versus String length is a method

ArrayList

- one of the most widely used classes from Collections framework.
- It offers the best combination of features offered by an array and the List data structure.
- the most commonly used operations with a list are:
    - add();
    - modify
    - delete
    - iterate

- a resizable array
- when creating an ArrayList, you have to tell the compiler what you are putting into the list
  - you use the<Object> to tell Java
  - until Java 7 you would have to do the same thing on both sides
  - now you can just add the <> after new arraylist
- Because you can use an ArrayList to store any type of Object, ArrayList defines an instance variable elementData of type Object
- when you add an element to the end of list, the ArrayList first checks whether its instance variable elementData has an empty slot at the end.
  - if there is an empty slot at its end, it stores the element at the first available empty slot.
  - if no slots are available, the ensureCapacity creates another array with a higher capacity and copies the existing values to this newly created array.
    - it then copies the newly added value at the first available empty slot in the array
- you can add an element at a particular position, an ArrayList creates a new array and inserts all its positions other than the position you specified
- if there are any subsequent elements to the right of the position that you specified, it shifts them to the right
- to access an element from an ArrayList, you can either use a for each loop, Iterator, or ListIterator
- **GUPTA EXAM TIP:** An ArrayList preserves the order of insertion of its elements. Iterator, ListIterator, and the for each loop will return the elements in the order in which they were added to the ArrayList. An iterator lets you remove elements as you iterate an ArrayList. It's not possible to remove elements from an ArrayList while iterating it using a for loop
- You can modify an ArrayList by either replacing an existing element in ArrayList or modifying all of its existing values
- the set() method is like replace() method
- ArrayList defines two methods to remove its elements
  - remove(int index) this method removes the element at the specified position in this list
  - remove(Object o) this method removes the first occurrence of the specified element from this list
- get(int index)-returns the element at the specified position in this list
- size()-returns the number of elements in the list
- contains(Object o)- returns true if this list contains the specified element, otherwise returns false
- indexOf(Object o)- returns the index of the first occurrence of the specified element in this list, or returns -1 if the list doesn't contain the element
- lastIndexOf(Object o)- returns the index of the last occurrence of the specified element in this list, or -1 if this list doesn't contain the element
- GUPTA EXAM TIP: An ArrayList can accept duplicate object values
- clone() method- returns a shallow copy of this ArrayList instance
  - shallow copy- creates a new ArrayList that points to the original elements

Object Equality

- **GUPTA EXAM TIP**: watch out for questions about the correct implementation of the equals method to compare two objects versus questions about the equals methods that simply compile correctly. If you'd been asked whether equals() in the previous example code would compile correctly, the correct answer is yes

- the equals method defines a method parameter of type Object, and its return type is boolean. Don't change the name of the method, its return type, or the type of method parameter when you define (override) this method in your class to compare two objects
- You may get to answer explicit questions on the contract of the equals method. An equals method that returns true for a null object passed to it will violate the contract. Also equals method modifies the value of any of the instance variables of the method parameter passed to it, or of the object on which it is called, it will violate the equals contract

hashcode() method

- not called by the equals method to determine the equality of two objects
- not on the exam

arraycopy

- arraycopy method copies an array from the specified source array, beginning at the specified position of the destination array. The last parameter is the number of elements you want to copy
    - parameters are Object source, int source position, Object dest, int destination position, and int length

String and StringBuilder:

Strings:

- an array of chars
- the first letter is stored at position 0
- the most used class in the Java API
- can create objects of the class String by using the new operator, by using the assignment operator (=), or by closing a value within the double quotes (“”)
- Strings are objects.
    - when you use the new operator, it points to two different objects
    - but when you just use the double quotes, it points to one object
- the literal value for String is null
- **GUPTA EXAM TIP:** If a String object is created using the keyword new, it always results in the creation using the assignment operator or double quotes only if a matching String object with the same value isn't found in the String constant pool
- Strings are immutable
    - meaning they can't be changed
    - stores its values in a private variable of the type char array.
    - The value is marked final
    - none of the methods defined in the String class manipulate the individual element
- methods of Strings don't change the char array
    - they modify the contents of the char array, and return a new string object
- Because String objects are immutable, their values won't change if you execute on them
- You can reassign them, though
- Concatenation operators (+ and +=) have a special meaning for Strings
    - Java has other functions for these operators

charAt method:

- you can use the method charAt(int index) to retrieve a character at a specified index of a String

indexOf() method:

- You can search a String for the occurrence of a char or a String.

- If it is not found in the target, it will return a -1
- by default, the indexOf () method starts its search from the char of the target String

substring() method:

- has two different overload methods
- one returns a substring of a String from the position you specify to the end of String
- the other can tell where you end the substring
  - the beginning index is INCLUSIVE!!
  - the end index is EXCLUSIVE!!
- **GUPTA EXAM TIP:** the substring method doesn't include the character at the end position in its return value

trim() method:

- returns a new String by removing all the leading and trailing white space (new lines, spaces, or tabs)

replace() method:

- returns a new String by replacing ALL the occurrences of a char with another char

length() method:

- retrieves the length of a String
- the length of a String is one number greater than the position that stores its last character.
- String length is a method while array length is a field

startsWith() and endWith() method:

- determines whether a String starts with or ends with a specified prefix, specified as a String.
- you can also have startsWith() from a particular position

Method chaining

- it's a common practice to use multiple String methods in a single line of code
- the methods are evaluated from left to right.

StringBuilder

- **GUPTA EXAM TIP:** You can expect questions on the need for the StringBuilder class and its comparison with the String class
- StringBuilders are mutable
  - meaning they use a non final char array to store its value
- You can use multiple overloaded constructors with different parameters
  - a stringbuilder
  - a number of characters
  - a String
  - nothing in it
- indexOf() method is different than String indexOf() method.
  - StringBuilder you can only pass in Strings

append() method:

- adds the specified value at the end of the existing value of a StringBuilder object.
- it is INCLUSIVE
- this method has been overloaded to return any type
- **GUPTA EXAM TIP:** For classes that haven't overridden the toString method, the append method appends the output from the default implementation of method toString defined in class Object

insert() method:

- as powerful as the append method
- the main difference between the append and insert methods is that the insert method enables you to insert the requested data at a particular position
    - append only allows you to add the requested data at the end of the StringBuilder object
- **GUPTA EXAM TIP:** Take note of the start and end positions when inserting in a StringBuilder. Multiple flavors of the insert method defined in StringBuilder may confuse you because they can be used to insert either single or multiple characters

delete() and deleteCharAt() method:

- delete method removes the characters in a substring of the specified StringBuilder
    - the first parameter is inclusive
    - the second parameter is exclusive
- deleteCharAt removes the char at the specified position
- **GUPTA EXAM TIP:** Combinations of the deleteCharAt and insert methods can be quite confusing

trim() method:

- unlike the trim method in the String class, the StringBuilder doesn't define the trim method

reverse() method:

- reverses the sequence of characters of a StringBuilder
- **GUPTA EXAM TIP:** You can't use the reverse method to reverse a substring of StringBuilder

replace() method:

- unlike the replace method defined in the String class, the replace method in the StringBuilder replaces a sequence of characters, identified by their positions

subSequence() method:

- apart from using the substring method, you can also use the method subSequence to retrieve a subsequence of a StringBuilder object
- the subsequence doesn't modify the existing value of a StringBuilder object

Using loop constructs:

for loop

- usually used to execute a set of statements a fixed number of times
- defines three types of statements separated with semicolons:
    - Initialization statements
    - Termination condition
    - Update clause (executable statement)
- statements defined within the loop body execute until the termination condition is false
- the update statement executes after all the statements defined within the for loop body
- initialization block executes only once
    - can declare multiple variables, but have to be variables of the same type
- termination condition evaluates each iteration before executing the statements defined within the body of the loop.
    - terminates when false
- usually use the update block to update the variable that you specify the termination condition
- a nested loop is a loop enclosed by another loop
    - outer loop encloses another loop

    - inner loop enclosed loop
    - often use to initialize or iterate multi-dimensional arrays
    - can label loops
- enhanced loops are also called for-each loops
    - offers some advantages over the regular for loop
    - use a colon to represent “in”
    - can also have nested for-each loops
    - there are some limitations for-each loops
        - can define a counter outside of the for-each loop, and add and modify the array elements, but it defeats the purpose of for-each
    - because the for-each loop hides the iterator used to iterate through the elements of a collection, you can’t use it to remove or delete collection values
- **GUPTA EXAM TIP:** use the for-each loop to iterate arrays and collections. Don’t use it to initialize, modify, or filter them
- should try using a for loop when you know the number of iterations

while and do-while

- both of these loops execute a set of statements as long as their condition evaluates to true
- the while loop checks the condition before it starts execution
- each time the condition used in the while loop to check whether or not to execute the loop
    - will terminate when false
    - otherwise it will execute indefinitely
- while loops accept boolean values
- do-while loop is used to repeatedly execute a set of statements until the condition that uses evaluates to false.
    - checks after it completes the execution of all the statements in its loop body
    - accepts arguments of type boolean
- can use curly braces with do-while and while loops to define multiple lines
    - otherwise only the first statement is executed
- should try to use a do-while or while when don’t know the number of iterations beforehand

break and continue statements:

- can use to exit a loop completely or to skip remaining statements in a loop iteration
- break is used to exit a loop as well as switch constructs
- continue skips the remaining code in the loop and goes directly to the start of the next loop iteration
- can also break out of the inner loop in a nested loop
- continue exits the current iteration of the loop
- continue doesn’t exit the loop, it restarts with the next loop iteration

labeled statements

- can add labels to certain types of statements:
    - code block defined using { }
    - all loop statements
    - conditional constructs(if and switch statements)
    - expressions
    - assignments
    - return statements

    - try blocks
    - throws statements
- can't add labels to declarations
- can use break statement to exit a loop that is outside the loop that you are in
- can use a continue statement to skip the iteration outside the loop that you are in

Working with methods and encapsulation:

Local variables:

- defined within a method
- the life span of a local variable is determined by its scope
- **GUPTA EXAM TIP:** Local variable topic is a favorite of OCA Java SE 7 Programmer I exam authors. Some questions may trick you to think it is talking about a complex subject, but it is really testing your knowledge on the scope of a local variable.

Method parameters:

- variables that accept values in a method.
- a special kind of local variable
- only available to the method

Instance variables:

- declared within a class, and outside a method.
- instance variables or instance attributes store the state of an object
- in an object
- **GUPTA EXAM TIP:** the scope of an instance variable is bigger than that of a local variable or a method parameter

  Class variable:
  - defined by using the keyword *static*
  - belongs to a class **NOT** an object
  - You can access a static variable through the object as well as a calling the class
    - for example:
      - Phone.softkeyboard and Phone p1.softkeyboard are both acceptable

Class variable:

- also a static variable by all the objects of a class

Scope:

- **GUPTA EXAM TIP:** Different local variables can have different scopes. The scope of local variables may be shorter than or as long as the scope of method parameters. The scope of method parameters. The scope of local variables is less than the duration of a method if they're declared in a sub-block (within braces {}) in a method. This sub-block can be an if statement, a switch construct, a loop, or a try-catch block.
- think from the inside to outside

Object Life Span

- An object starts when the keyword operator `new` is used. You can initialize a reference variable with this object.
    - Strings are a special class.
      - They can be initialized with a new operator
      - They can also be initialized with just " ".

- **GUPTA EXAM TIP:** Watch out for a count of the total objects created in any given code-- the ones that can be accessed using a variable and the ones that can't be accessed using any variable. The exam may question you on the count of objects created
- Once an object is created, it can be accessed using its reference variable until it goes out of scope or becomes `null`
- **GUPTA EXAM TIP:** An object is marked as eligible to be garbage collected when it can no longer be accessed, which happens when the object goes out of scope. It can also happen when an object's reference variable is assigned an explicit `null` value or is reinitialized
- The garbage collector is a low priority thread that marks the objects eligible for garbage collection in the JVM and clears memory of these objects.

Methods

- a group of statements identified with a name
- used to define the behavior of an object
- the return type of a method states the type of value that a method will return
- the only method that doesn't have to `return` anything is void.
    - void can just be `return;`
- it can return a primitive value or an object of any class.

Method parameters

- the variables that appear in the definition of a method
- The ellipsis(...) that follows the data type indicates that the method parameters may pass an array or multiple comma-separated values
    - You can only define one variable argument in a parameter list
- **GUPTA EXAM TIP:** You may be questioned on the valid return types for a method that doesn't accept any method parameters. Note that there are no valid or invalid combinations of a number and type of method parameters that can be passed to a method and the value that it can return.

Return statement

- used to exit from a method, with or without a value.
- For methods with a return type, the return statement must be immediately followed by a returned value
- The return statement MUST be the last statement to execute in a method

Overloaded method

- methods with the same name but different method signature
    - different return type or parameter
- may or may not define a different return type
- may or may not define different access modifiers
- can't be defined by only changing their return type or access modifiers

Argument list

- Overloaded methods accept different lists of arguments.
- can differ in terms of any of the following:
    - change in the number of parameters that are accepted
    - change in the types of parameters that are accepted
    - change in the positions of the parameters that are accepted (based on parameter type, not variable names)

Constructors

- special methods that create and return an object of the class in which they're defined

- have the same name as the name of the class in which they're defined
- don't specify a return type--not even void
- a constructor CAN have the same type parameter
- two types of constructors:
    - user-defined constructors
    - default constructors

User-Defined constructors

- the author defines a constructor in a class
- Because a constructor is called as soon as an object is created, you can use it to assign default values to instance variables
- does have an implicit return type
    - the class in in which it's defined
- **GUPTA EXAM TIP:** You can define a constructor using all four access modifiers: `public`, `protected`, default, and `private`.
- The method (with the return type `void`) is reduced to the state of another method in the class.
    - The logic applies to all of the other data types
    - it no longer is a constructor
- **GUPTA EXAM TIP:** A constructor MUST NOT define any return type. Instead, it creates and returns an object of the class in which it's defined. If you define a return type for a constructor, it's no longer treated as a constructor.

Default Constructor

- Java inserts in the absence of a user defined constructor
- doesn't accept any method arguments
- calls the constructor of the super class and assigns default values to ALL instance variables
- Once you create a constructor, the default constructor goes away

Overloaded Constructors

- must be defined using different argument lists
- can't be defined by just a change in the access modifiers
- are invoked by using the keyword `this`
- can't be defined by just a change in the access modifier
- may be defined using different access modifiers

Initializer Blocks

- defined within a class, not as a part of a method
- if you define an initializer and a constructor for a class, both of these will execute.
- used to initialize the variables of anonymous classes

Object Field

- another name for an instance variable defined in a class
- A setter method is used to set the value of a variable
- A getter method is used to retrieve the value of a variable
- the name of an object field IS NOT determined by the name of its getter or setter methods.
- Java uses the dot notation (.) to execute a method on a reference variable

Encapsulation

- a well-encapsulated object doesn't expose its internal parts to the outside world

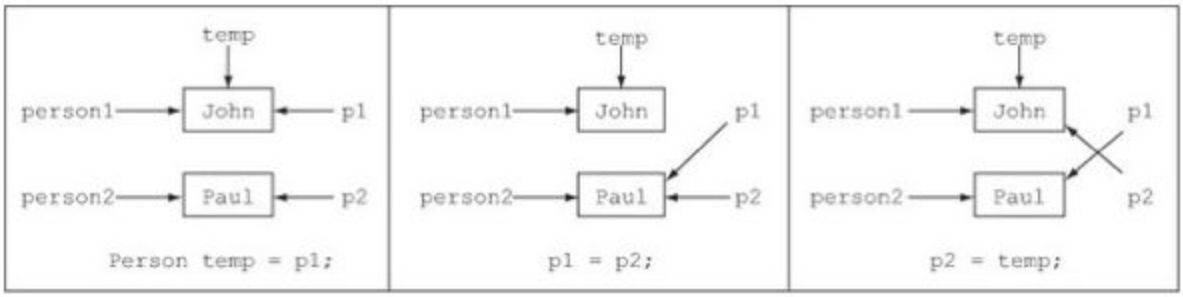
- a class may need a number of variables and methods to store an object's state and define its behavior
- usually variables are encapsulated as private
- **GUPTA EXAM TIP:** The terms encapsulation and information hiding are used interchangeably. By exposing object functionality only through methods, you can prevent your private variables from being assigned any values that don't fit your requirements.

Passing Objects and primitives to methods

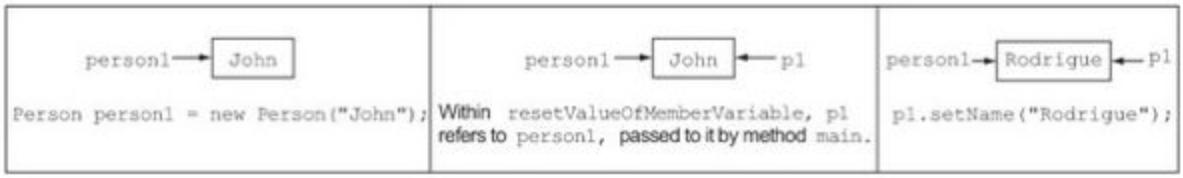
- Object references and primitives behave in a different manner when they're passed to a method because of the differences in how these two data types are internally store by Java

  Passing primitives to methods
  - The value of a primitive data type is copied and passed on to a method
  - It's okay to define a method parameter with the same name as an instance variable(scope)
  - Within a method a method parameter takes precedence over an object field
    - to call the object field use the keyword `this`
  - **GUPTA EXAM TIP:** When you pass a primitive variable to a method, its value remains the same after the execution of the method. The value doesn't change, regardless of whether the method reassigns the primitive to another variable or modifies it.

  Passing Object references to methods
  - Two main cases:
    - when a method reassigns the object reference passed to it to another variable
      - when you pass an object reference to a method, the method can assign it to another variable(scope)



    - when a method modifies the state of the object reference passed to it



Working with Inheritance:

- A class uses the keyword extends to inherit a class
- also called subclassing

- the top class called a superclass, base class, or parent class
- the class that inherits from the superclass is called a subclass, derived class, extended class, or child class
- code that works with a superclass works with all subclasses that inherit the class
    - but subclasses can override the superclass' method
- **GUPTA EXAM TIP:** Inheritance enables you to reuse code that has already been defined by a class.
- a subclass inherits ALL non-private members(Default, protected, public) of its base class, but there are specific rules for inheritance
- a subclass cannot inherit:
    - private members of the superclass
    - superclass members with default access, if the superclass is in a separate package than the subclass
    - constructors of the superclass. subclasses have to call the superclass' constructors, but they can't override it
- subclasses can define additional properties and behaviors
- subclasses can also have their own constructors
    - which hides or overrides its superclass' constructor
- inheritance works with concrete or abstract classes as a superclass

Abstract Class

- groups common properties and behaviors, but prevents itself from being instantiated
- forces subclasses to define their own implementations for a behavior by defining the method as abstract
- an abstract class doesn't have to have an abstract method
- if there is one or more abstract methods in a class, the class HAS to be defined as abstract
- abstract methods must be implemented in the subclass
- if the subclass doesn't implement all abstract methods, then the method has to be put as abstract in the class

Interfaces

- abstract classes to the extreme
- only abstract methods and constants
- All members of an interface are implicitly public
- if the signature of a method is changed in an interface, all classes that implement the interface will fail to compile
- **GUPTA EXAM TIP:** the method signatures of a method defined in an interface and the classes that implement the interface must match, or the classes won't compile
- can only define constants.
    - once assigned, you can't change the value of a constant.
    - the variables of an interface are implicitly public, final, and static
- **GUPTA EXAM TIP:** Because a derived class may inherit different implementations for the same method signature from multiple super classes, multiple inheritance is not allowed in Java
- an interface can extend multiple interfaces
- the type of the object reference variable and the type of the object being referred to may be different

Abstract Classes	Interfaces
• cannot be instantiated • can have concrete methods • can have abstract methods ○ subclasses must implement abstract methods • can have fields • MUST be extended	• any fields are implicitly static, final • all members are implicitly public • all methods are implicitly abstract • cannot instantiate an interface • (an abstract class with all abstract methods) • can extend an interface • MUST be implemented

Reference variable vs. object type

- object reference variable and the type of the object may be different.
- rules for different object reference variable and the type of object:
  - its own type
  - its base class
  - implemented interfaces
- when using different reference variables, it can only access what is the reference type
- compiler concerns itself of reference type
- when using reference variables, you may want to use the superclass or interface when you might not be interested in all the members of a derived class
  - it can also be used to create an array or List of different subclasses from the superclass
- the reverse WILL NOT compile
  - you can't refer to an object of a base class by using a reference variable of its derived class
  - all members of a subclass can't be accessed using an object of the superclass or interface
- **GUPTA EXAM TIP:** you may see multiple questions on the exam that try to assign an object of a superclass to a reference variable of a sub class

Casting

- the process of forcefully making a variable behave as a variable of another type
- if a class shares an IS-A relationship with another class or interface, their variables can be cast to each other's type
  - compiler will let you get past with reference variable for the hierarchy
  - but at runtime, it will give you a CastClassException
  - compiler will prevent you from casting to a datatype that is not in your inheritance hierarchy
    - compiler will allow you to cast any interface type, but will get a ClassCastException if the object does not implement the interface you're casting
  - can always store a subclass into a variable of type superclass
    - opposite not true
    - see reference variable vs. object type
  - siblings can't see each other, only see parent class
  - casts right to left
  - you can always substitute an implementing class for its interface
  - you MUST cast if you want to store an interface into a class that implements it

    - once you cast an object on the hierarchy, you must cast it back down before you store it to a variable whose type is lower in the hierarchy
- use casting to get past the Java compiler and access the members of the object
- tells the compiler “I know that the actual object being referred to the subclass, even though I’m using a superclass reference variable
- when having a different reference variable, you can’t access all the members of an object if you access it using a reference variable of any of its implemented interfaces or of a base class

this and super

- this and super are implicit object references.
- these variables are defined and initialized by the JVM for every object in its memory
- the this reference always points to an object’s own instance
- any object can use the this reference to refer to its own instance
- the this reference may be used only when code executing within a method block needs to differentiate between an instance variable and its local variable or method parameters
- when there is a clash in names, the local variable will take precedence within the scope
- also can differentiate one constructor from another by using the keyword this() or super()
    - has to be the first line if use it
- super refers to the superclass’ instance
- the variable reference super can be used to access a variable or method from the superclass if there is a clash between names
    - shadowed scenario
- you can use super to access a method defined with the same name in the superclass
- super and this cannot access static methods
    - static methods belong to a class
    - this and super belong to an object

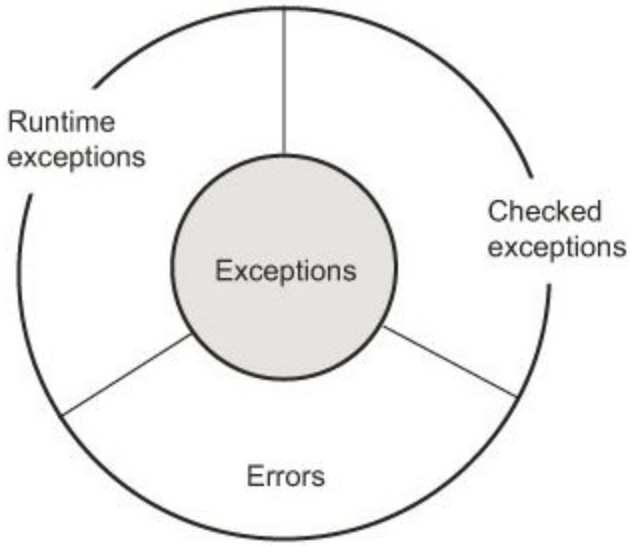
Polymorphism

- literal meaning is many forms
- comes into play when both the super and subclass define methods with the same method signature
    - same method name and method parameters
- gives the object the ability to behave in its own specific manner when the same action is passed to it
- overridden methods have the same name, same argument list, or same return type
    - otherwise it won’t be an overridden method, it would be an overloaded method
- you have to override abstract methods
    - polymorphism doesn’t always have to be abstract methods
    - you can also override concrete methods
- you can use reference variables of a superclass to refer to an object of a subclass
- when compiled, it accesses the super method, but during runtime it accesses the subclass method
- **GUPTA EXAM TIP:** Watch out for code in the exam that uses variables of the superclass to refer to objects of the subclass and then accesses variables and methods of the referenced object. Remember variables bind at compile time, methods bind at runtime
- can also be implemented using interfaces

- polymorphism with interfaces always involves abstract methods from the implemented interface because interfaces can define only abstract methods
- shadowed variable, reference type wins
- the subclass method cannot be less accessible than the superclass method
    - can be more accessible
- the parameter list must be the same with the method being overridden
- the return type on an overridden method can be a subclass
    - typically the same though
- can't override:
    - private methods
    - final methods
    - static methods

Handling Exceptions

- separate concerns between defining the regular program logic and the exception handling code
- help pinpoint the offending code(code that throws exception), together with the method in which it is defined, by providing a stack trace of the exception or error
- the stack trace gives you a trace of the methods that were called when the JVM encountered an unhandled exception.
    - Stack traces are read for the bottom
    - can range from a few lines to a hundred lines of code
    - works with handled and unhandled exceptions
- an exception is an object.
    - all exceptions subclass `java.lang.Throwable`
- an operating system keeps track of the code that it needs to execute using a stack
    - a stack is a type of list in which items that are added last to it are the first ones to be taken off it -- Last In First Out
    - this stack uses a stack pointer to point to the instructions that the OS should execute
- you can use try-catch-finally blocks to define code that doesn't halt execution when it encounters an exceptional condition
- exceptions can be divided by three main categories
    - checked exceptions
    - Runtime Exceptions(unchecked exceptions
    - errors



- a subclass method can throw the same exception, a subclass of the exception, or no exception at all
- a subclass constructor must throw all CHECKED exceptions defined in its superclass constructor, or a superclass of the exceptions thrown
    - it can also throw additional exceptions
- think throwing exceptions like throwing a ball
    - something has to catch a check exception
        - like playing catch with a person
    - the JVM catches an unchecked and checked exceptions
        - like playing catch with a dog

Try-Catch-Finally

- first you try to execute your code.
- if it doesn't execute as planned, you handle the exceptional conditions using a catch block
    - you catch the exceptional event arising from the code enclosed within the try block and handle the event by defining appropriate exception handlers
- finally, you execute a set of code, in all conditions, regardless of whether the code in the try block throws exceptions
- you can create an exception of your own -- a custom exception -- by extending the class Exception (or any of its subclasses).
- Although the creation of custom classes is not on this exam, you may see questions in the exam that create and use custom exceptions
- checked exceptions MUST throw the exception
    - can use a try-catch-finally block
- **GUPTA EXAM TIP:** the finally block executes regardless of whether the try block throws an exception
- For a try block, you can define multiple catch blocks, but you can only have a single finally block.
- multiple catch blocks handle different types of exceptions
- a finally block is used to cleanup code
    - code that closes and releases resources, such as file handlers and database or network connections

- there are a few scenarios in Java in which a finally block does not execute
    - the try or catch block execute System.exit, which terminates the application
    - fatal errors- a crash of the JVM or the OS
- if a catch and finally blocks define return statements, the finally block ALWAYS wins
- if a catch block returns a block returns a primitive data type, the finally block can't modify the value being returned by it
    - control in the catch block makes a copy of the catch return and modifies the copy
    - a finally block can modify an object
    - Remember primitives are passed by value, objects are passed by reference
- **GUPTA EXAM TIP:** Watch out for code that returns a value from the catch block and modifies it in the finally block
- Order doesn't matter for unrelated classes in the catch block.
- it does mated for related classes charing an IS-A relationship
- if you try to catch an exception of the superclass before an exception of the subclass
    - will fail to compile
- Java doesn't compile if it contains unreachable statements
- You can do whatever you want with an exception, including rethrow it
- when you rethrow a checked exception, it's treated like a regular thrown checked exception
    - all rules of handling checked exceptions apply
- you can rethrow a runtime exception, but you're not required to catch it, nor must you modify your method signature to include the throw clause
- if a method doesn't wish to handle the check exceptions thrown by a method it calls, it can throw these exceptions using the throws clause in its own method signature
- You can define a try-catch-finally block within another try-catch-finally block
    - no limits of level of nesting
- a try MUST HAVE a finally or catch block

Checked Exceptions

- checked exceptions take most of our attention
- a checked exception is an unacceptable condition foreseen by the author of a method but outside the immediate control of the code
- a subclass of the java.lang.Exception, BUT NOT a subclass of java.lang.RuntimeException
- **GUPTA EXAM TIP:** you may have to select which type of reference variable to use to store the object of the thrown checked exception in a handler.
- if a method uses another method that may throw a checked exception, one of the two following should be true:
    - method should be enclosed within a try-catch block
    - method should specify this exception to be thrown in its method signature
- unacceptable conditions that a programmer foresees at the time of writing a method
- if a method uses multiple catch blocks, make sure that derived exceptions are before parent Exception or else it will not compile (for example, you need to catch ArrayIndexOutOfBoundException before IndexOutOfBoundException).
- Checked exceptions must be part of the method signatures

Runtime Exceptions

- a representation of a programming error
- a subclass of java.lang.RuntimeException

- may or may not be part of the method signature, even if a method may throw it
- you can catch RuntimeExceptions the same way you can catch a checked exception
- ArrayIndexOutOfBoundsException is a runtime exception
    - it is thrown when a piece of code tries to access an array out of its bounds(either an array is accessed at a position less than 0 or at a position greater or equal to its length).
    - Is a derived class of IndexOutOfBoundsException
- IndexOutOfBoundsException is a runtime exception
    - thrown when a piece of code tries to access a list using an illegal index
- ClassCastException is a runtime exception
    - thrown when an object fails an IS-A test with the class type to which it's being cast
    - you can use the instanceof operator to verify whether an object can be cast to another class before casting it
- IllegalArgumentException is a runtime exception
    - thrown to specify a method has passed illegal or inappropriate arguments
    - programmers usually use this exception to validate the arguments that are passed to a method. The exception constructor is passed a descriptive message, specifying the exception details.
    - Each object of the IllegalArgumentException is passed a different String message that briefly describes it
- IllegalStateException is a runtime exception
    - signals a method has been invoked at an illegal or inappropriate time
    - the Java environment or Java application is not in an appropriate state for the requested operation
    - can throw to signal to the calling method that the method being requested for execution can't be called for the current state of an object
- NullPointerException is a runtime exception
    - thrown by the JVM if you try to access a method or a variable with a null value.
    - by default static and instance variables of a class are assigned a null value
    - local variables aren't assigned to a value, not event null
    - if you attempt to use an uninitialized variable, your code will fail to compile.
- NumberFormatException is a runtime exception
    - may throw from own method to indicate that there's an issue with the conversion of a String value to a specified numeric format(decimal, octal, hexadecimal, binary), and you can add a customized exception message.
    - One of the most common candidates for this exception is methods that are used to convert a command-line argument (accepted as a String value) to a numeric value.
    - Please note that all command-line arguments are accepted in a String array as String values
- SecurityException
    - thrown by the security manager to indicate security violations

Errors

- a serious exception thrown by the JVM as a result of an error in the environment state that processes your code
- a subclass of java.lang.Error
- doesn't need to be part of a method signature

- can be caught by an exception handler, but it shouldn't be
- ExceptionInInitializerError
  - typically thrown by the JVM when a static initializer in your code throws any type of RuntimeExceptions
  - initialization of a static variable
  - execution of a static method(called from either of the previous two items)
  - **GUPTA EXAM TIP:** The class DemoExceptionInInitializerError seems deceptively simple, but it's a good candidate for an exam question.
    - this class throws the error ExceptionInInitializerError when the JVM tries to load
- StackOverflowError
  - extends Virtual-Machine Error
  - it should be managed by the JVM
  - thrown by JVM when a Java program calls itself so many times that the memory stack allocated to execute the Java program "overflows"
- NoClassDefFoundError
  - thrown if the JVM or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found
  - class goes away, get this error
  - should not be handled by the code and should be left to be handled exclusively by the JVM
- OutOfMemoryError
  - an error that the JVM runs when it may run out of memory on the heap, and the garbage collector may not be able to free more memory for the JVM.
  - The JVM is unable to create any more objects on the heap
  - You will always work with a finite heap size, no matter what platform you work on, so you can't create and use an unlimited number of objects in your application