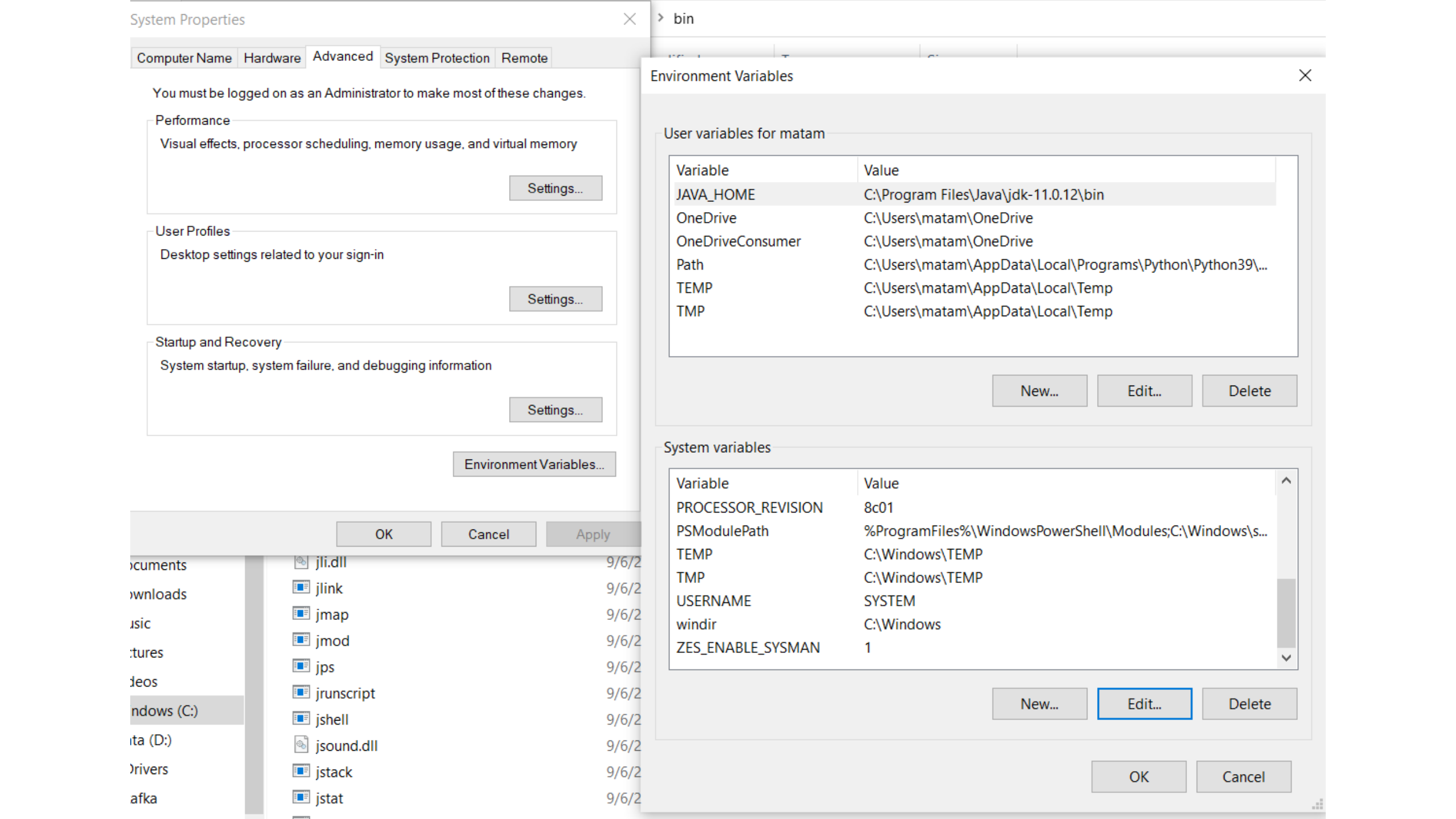
JAVA

Core Java with Kiran

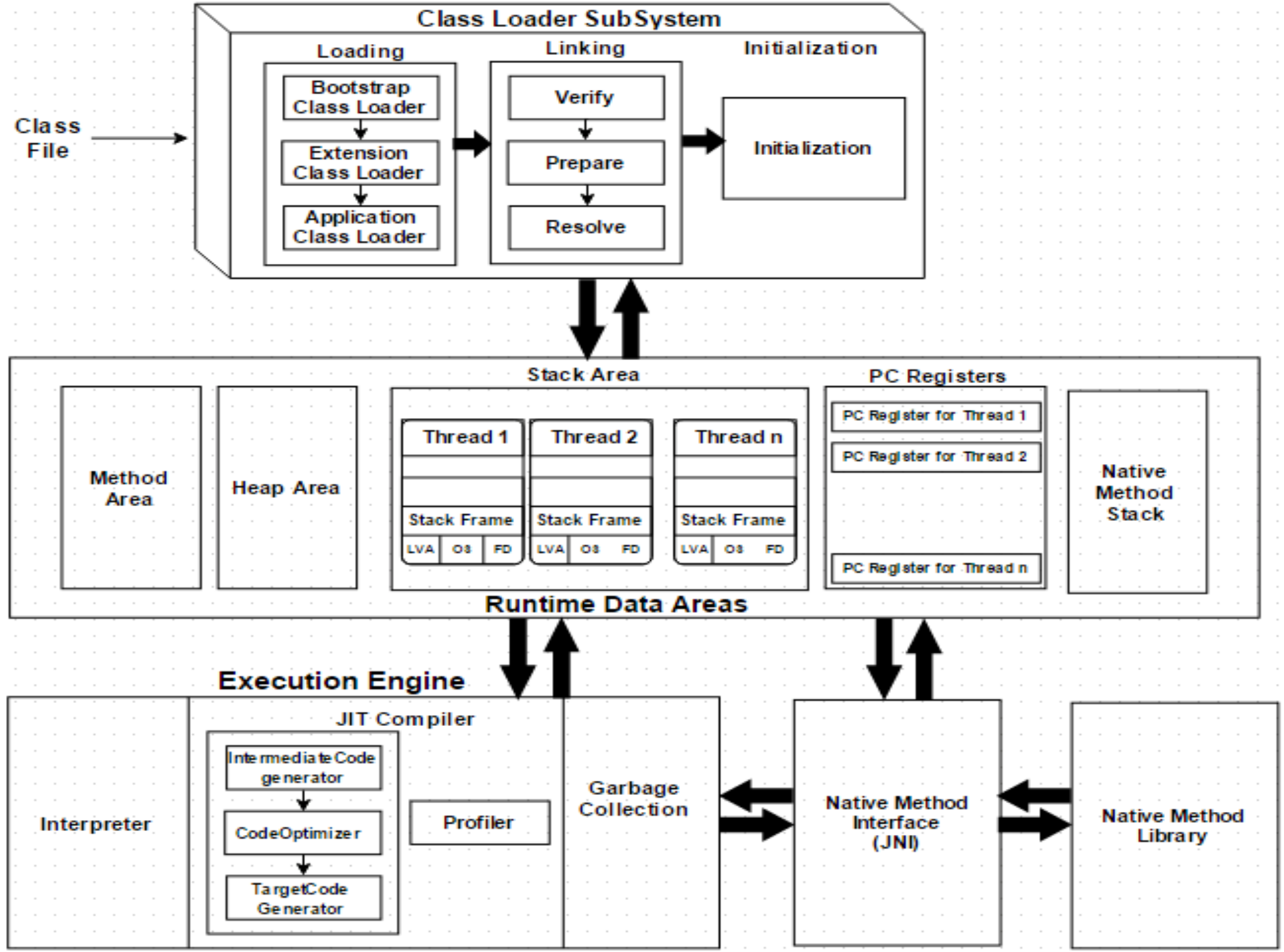
How to set classpath?

```
C:\Users\matam>path
PATH=C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\ProgramData\Anaconda3;C:\ProgramData\Anaconda3\Library\mingw-w64\bin;C:\ProgramData\Anaconda3\L
ibrary\usr\bin;C:\ProgramData\Anaconda3\Library\bin;C:\ProgramData\Anaconda3\Scripts;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSS
H\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:
\Windows\System32\OpenSSH\;C:\Program Files\nodejs\;C:\Program Files\Docker\Docker\resources\bin;C:\ProgramData\DockerDesktop\version-bin;C:\Users\matam\AppData\Local\Programs\Python\Python39\Scripts\;C:\Users\m
atam\AppData\Local\Programs\Python\Python39\;C:\Users\matam\AppData\Local\Microsoft\WindowsApps;C:\Users\matam\AppData\Roaming\npm;C:\Users\matam\AppData\Local\GitHubDesktop\bin

C:\Users\matam>set path=C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\ProgramData\Anaconda3;C:\ProgramData\Anaconda3\Library\mingw-w64\bin;C:\Prog
ramData\Anaconda3\Library\usr\bin;C:\ProgramData\Anaconda3\Library\bin;C:\ProgramData\Anaconda3\Scripts;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Wind
ows\System32\OpenSSH\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Windows
PowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files\nodejs\;C:\Program Files\Docker\Docker\resources\bin;C:\ProgramData\DockerDesktop\version-bin;C:\Users\matam\AppData\Local\Programs\Python\Python39\
Scripts\;C:\Users\matam\AppData\Local\Programs\Python\Python39\;C:\Users\matam\AppData\Local\Microsoft\WindowsApps;C:\Users\matam\AppData\Roaming\npm;C:\Users\matam\AppData\Local\GitHubDesktop\bin;C:\Program Fil
es\Java\jdk-11.0.12\bin;

C:\Users\matam>path
PATH=C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\ProgramData\Anaconda3;C:\ProgramData\Anaconda3\Library\mingw-w64\bin;C:\ProgramData\Anaconda3\L
ibrary\usr\bin;C:\ProgramData\Anaconda3\Library\bin;C:\ProgramData\Anaconda3\Scripts;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSS
H\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:
\Windows\System32\OpenSSH\;C:\Program Files\nodejs\;C:\Program Files\Docker\Docker\resources\bin;C:\ProgramData\DockerDesktop\version-bin;C:\Users\matam\AppData\Local\Programs\Python\Python39\Scripts\;C:\Users\m
atam\AppData\Local\Programs\Python\Python39\;C:\Users\matam\AppData\Local\Microsoft\WindowsApps;C:\Users\matam\AppData\Roaming\npm;C:\Users\matam\AppData\Local\GitHubDesktop\bin;C:\Program Files\Java\jdk-11.0.12
\bin;

C:\Users\matam>java -version
java version "11.0.12" 2021-07-20 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.12+8-LTS-237)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.12+8-LTS-237, mixed mode)
```


System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your sign-in
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables...
OK
Cancel
Apply
bin
Environment Variables
User variables for matam
Variable
Value
JAVA_HOME
C:\Program Files\Java\jdk-11.0.12\bin
OneDrive
C:\Users\matam\OneDrive
OneDriveConsumer
C:\Users\matam\OneDrive
Path
C:\Users\matam\AppData\Local\Programs\Python\Python39\...
TEMP
C:\Users\matam\AppData\Local\Temp
TMP
C:\Users\matam\AppData\Local\Temp
New...
Edit...
Delete
System variables
Variable
Value
PROCESSOR_REVISION
8c01
PSModulePath
%ProgramFiles%\WindowsPowerShell\Modules;C:\Windows\s...
TEMP
C:\Windows\TEMP
TMP
C:\Windows\TEMP
USERNAME
SYSTEM
windir
C:\Windows
ZES_ENABLE_SYSMAN
1
New...
Edit...
Delete
OK
Cancel
jli.dll
jlink
jmap
jmod
jps
jrunscript
jshell
jsound.dll
jstack
jstat
9/6/2

JVM Architecture

JVM , JRE , JDK


Class Loader SubSystem
Loading
Linking
Initialization
Class File
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Verify
Prepare
Resolve
Initialization
Stack Area
PC Registers
Method Area
Heap Area
Thread 1
Thread 2
Thread n
Stack Frame
LVA
OS
FD
PC Register for Thread 1
PC Register for Thread 2
PC Register for Thread n
Native Method Stack
Runtime Data Areas
Execution Engine
JIT Compiler
Intermediate Code generator
CodeOptimizer
Target Code Generator
Profiler
Interpreter
Garbage Collection
Native Method Interface (JNI)
Native Method Library

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader.

There are three built-in classloaders in Java.

1.Bootstrap ClassLoader: This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.

2.Extension ClassLoader: This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.

3.System/Application ClassLoader: This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

The PICTURE

What is the JVM?	• **Class Loader Subsystem + Runtime Data Area + Execution Engine**
What is JRE ?	• **JVM + Libraries**
What is JDK ?	• **JRE + Development Tools**
What is JIT compiler?	• **which compiles the entire Java bytecode once and changes it to native code.**

- JVM is an essential part of JDK and JRE because it is contained or inherent in both. No matter java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java code line by line thus it's too known as an interpreter.

- Therefore, you don't require to install JVM on an individual basis into your computer because it is inbuilt into your JDK or JRE installation package.

- JVM comes in two completely different flavors — client and server.

- The JVM is termed virtual because it provides a machine interface that doesn't depend upon the underlying OS and machine hardware design.

- This independence from hardware and therefore the operating system is a foundation of the write-once-run-anywhere worth of Java programs.

A **Java virtual machine (JVM)** is a virtual machine that can execute Java ByteCode. It is the code execution component of the Java software platform.

The **Java Development Kit (JDK)** is an Oracle Corporation product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java Software Development Kit (SDK).

Java Runtime Environment, is also referred to as the Java Runtime, Runtime Environment
OpenJDK (Open Java Development Kit) is a free and open source implementation of the Java programming language.

It is the result of an effort Sun Microsystems began in 2006. The implementation is licensed under the GNU General Public License (GPL) with a linking exception.

ClassLoader: ClassLoader is a subsystem used to load class files. ClassLoader first loads the Java code whenever we run it.

Class Method Area: In the memory, there is an area where the class data is stored during the code's execution. Class method area holds the information of static variables, static methods, static blocks, and instance methods.

Heap: The heap area is a part of the JVM memory and is created when the JVM starts up. Its size cannot be static because it increase or decrease during the application runs.

Stack: It is also referred to as thread stack. It is created for a single execution thread. The thread uses this area to store the elements like the partial result, local variable, data used for calling method and returns etc.

Native Stack: It contains the information of all the native methods used in our application.

Background Information

- The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

- JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Objectives

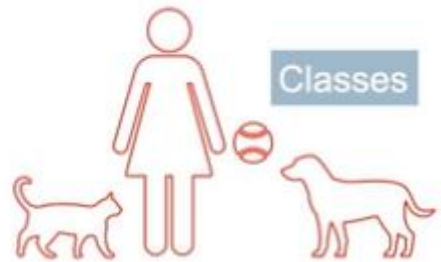
1. Describe the purpose of a variable in the Java language
2. List and describe four data types
3. Declare and initialize `String` variables
4. Concatenate `String` variables with the '+' operator
5. Make variable assignments
6. Declare and initialize `int` and `double` variables

Classes

Class and Object are two key object-oriented concepts.

Java code is structured with classes.

- Class represents a type of thing or a concept, such as Dog, Cat, Ball, Person.
- Each class defines what kind of information (attributes) it can store:
  - A Dog could have a name, color, size.
  - A Ball would have type, material, and so on.
- Each class defines what kind of behaviors (operations) containing program logic (algorithms) it is capable of:
  - A Dog could bark and fetch a Ball.
  - A Cat could meow but is not likely to play fetch.



✣ Objects are covered next

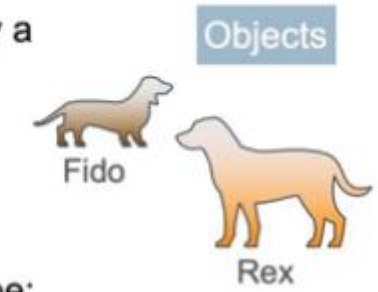
```
class Person {
  void play() {
    Dog dog = new Dog();
    dog.name = "Rex";
    Ball ball = new Ball();
    dog.fetch(ball);
  }
}
```

```
class Dog {
  String name;
  fetch(Ball ball) {
    ball.find();
    ball.chew();
  }
}
```

Objects

An Object is a specific instance (example of) a Class.

- Each object would be capable of having specific values for each attribute defined by a class that represents its type. For example:
  - A dog could be called Fido, and be brown and small.
  - Another could be called Rex, and be orange and big
- To operate on an object, you can reference it using a variable of a relevant type.
- Each object would be capable of behaviors defined by a class that represents its type:
  - At run time, objects invoke operations upon each other to execute program logic.




```
class Person {
  void play() {
    Dog dog = new Dog();
    dog.name = "Rex";
    Ball ball = new Ball();
    dog.fetch(ball);
  }
}
```

```
class Dog {
  String name;
  fetch(Ball ball) {
    ball.find();
    ball.chew();
  }
}
```

Inheritance

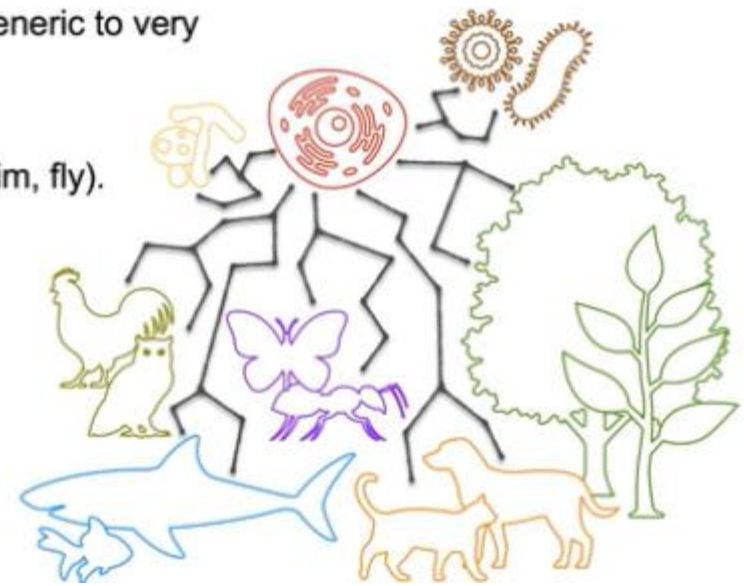
You can reuse (inherit) attributes and behaviors across class hierarchy.

- Classes can form hierarchical relationships.
- Superclass represents a more generic, parent type (living organism).
- Superclasses define common attributes and behaviors (eat, propagate).
- A subclass represents a more specific, child type (animal, plant, and so on).
- There could be any number of levels in the hierarchy, from very generic to very specific child types (dog, cat, and so on).
- Subclasses inherit all attributes and behaviors from their parents.
- Subclasses can define more specific attributes and behaviors (swim, fly).

```
class Animal extends LivingOrganism {
  // generic attributes and behaviours
}
```

inherited

```
class Dog extends Animal {
  // specific attributes and behaviours
}
```



Java APIs

Java Development Kit (JDK) provides hundreds of classes for various programming purposes:

- To represent basic data types, for example, `String`, `LocalDateTime`, `BigDecimal`, and so on
- To manipulate collections, for example, `Enumeration`, `ArrayList`, `HashMap`, and so on
- To handle generic behaviors and perform system actions, for example, `System`, `Object`, `Class`, and so on
- To perform input/output (I/O) operations, for example, `FileInputStream`, `FileOutputStream`, and so on
- Many other API classes are used to access databases, manage concurrency, enable network communications, execute scripts, manage transactions, security, logging, build graphical user interfaces, and so on

- Application Programming Interface (API) is a term that describes a collection of classes that are designed to serve a common purpose.
- All Java APIs are thoroughly documented for each version of the language. For example, Java 11 documentation can be found at: https://docs.oracle.com/en/java/javase/11/docs/api/

Java Keywords, Reserved Words, and a Special Identifier

Keywords available since 1.0
Keywords no longer in use
Keywords added in 1.2
Keywords added in 1.4
Keywords added in 5.0
Keywords added in 9.0

Reserved words for literals values

Special identifier added in 10

if	implements	boolean	assert
else	extends	try	enum
continue	interface	catch	module
break	class	finally	requires
for	static	throw	transitive
do	final	throws	exports to
while	return	new	uses
switch	transient	this	provides
case	void	super	with
default	byte	instanceof	opens to
private	short	native	
protected	int	synchronized	true
public	long	volatile	false
import	char	goto	null
package	float	const	
abstract	double	strictfp	var

Notes

- Keywords and Literals cannot be used as identifiers (names of classes, variables, methods, and so on).
- Actual use and meaning of these keywords and literals are covered later in this course.

Java Naming Conventions

- Java is case-sensitive; `Dog` is not the same as `dog`.
- Package name is a reverse of your company domain name, plus the naming system adopted within your company.
- Class name should be a noun, in mixed case with the first letter of each word capitalized.
- Variable name should be in mixed case starting with a lowercase letter; further words start with capital letters.
- Names should not start with numeric characters (0-9), underscore _ or dollar $ symbols.
- Constant name is typically written in uppercase with underscore symbols between words.
- Method name should be a verb, in mixed case starting with a lowercase letter; further words start with capital letters.

```
package:   com.oracle.demos.animals
class:     ShepherdDog
variable:  shepherdDog
constant:  MIN_SIZE
method:    giveMePaw
```

```
package:   animals
class:     Shepherd Dog
variable:  _price
constant:  minSize
method:    xyz
```

- Note: The use of the _ symbol as a first or only character in a variable name produces a compiler warning in Java 8 and an error in Java 9 onward.

Java Basic Syntax Rules

- All Java statements must be terminated with the " ; " symbol.
- Code blocks must be enclosed with " { " and " } " symbols.
- Indentations and spaces help readability, but are syntactically irrelevant.

```
package com.oracle.demos.animals;
class Dog {
  void fetch() {
    while (ball == null) {
       keepLooking();
    }
  }
  void makeNoise() {
    if (ball != null) {
      dropBall();
    } else {
      bark();
    }
  }
}
```

- Note: Example shows some constructs such as `if`/`else` and `while` that are covered later in the course.

Define Java Class

- Class name is typically represented by one or more nouns, for example, `Dog`, `SabreToothedCat`, `Person`.
- Class must be saved into a file with the same name as the class and extension `.java`.
- Classes are grouped into packages.
- Packages are represented as folders where class files are saved.
- Package name is typically a reverse of your company domain name, plus a naming system adopted within your company. Example: `com.oracle.demos`, `org.acme.something`
- Package and class name must form a unique combination.

```
/somepath/com/oracle/demos/animals/Dog.java
```

```
package <package name>;
class <ClassName> {

}
```

```
package com.oracle.demos.animals;
class Dog {
   // the rest of this class code
}
```

- Note: If package definition is missing, class would belong to a "default" package and would not be placed into any package folder. However, this is not a recommended practice.

Access Classes Across Packages

To access a class in another package:

- Prefix the class name with the package name
- Use the import statement to import specific classes or the entire package content
  - The import of all classes from the `java.lang.*` package is implicitly assumed.

The example shows three alternative ways of referencing the class `Dog` in the package `animals` from the class `Owner` in the package `people`:

```
package animals;
public class Dog {
}
```

```
package people;
public class Owner {
   animals.Dog myDog;
}
```

```
package people;
import animals.Dog;
public class Owner {
   Dog myDog;
}
```

```
package people;
import animals.*;
public class Owner {
   Dog myDog;
}
```

Notes

- Imports are not present in a compiled code. An `import` statement has no effect at runtime efficiency of the class. It is a simple convenience to avoid prefixing class name with package name throughout your source code.
- Access modifiers (such as `public`) are explained in the following slide.

Use Access Modifiers

Access modifiers describe the visibility of classes, variables, and methods.

- `public` - Visible to any other class
- `protected` - Visible to classes that are in the same package or to subclasses
- `<default>` - Visible only to classes in the same package
- `private` - Visible only within the same class

```
package <package name>;
import <package name>.<class name>;
import <package name>.*;
<access modifier> class <ClassName> {
   <access modifier> <variable definition>
   <access modifier> <method definition>
}
```

```
package b;
import a.*;
public class Y extends X {
   public void doThings() {
      X x = new X();
      x.y1;
      x.y2;
      x.y3;
      x.y4;
   }
}
```

```
package a;
public class X {
   public Y y1;
   protected Y y2;
   Y y3;
   private Y y4;
}
```

Notes

- `<default>` means that no access modifier is explicitly set.
- Subclass-Superclass relationship (use of the `extends` keyword) is covered later in the course.
- Any nonprivate parts of your class should be kept as stable as possible, because changes of such code may adversely affect any number of other classes that may be using your code.

Create Main Application Class

The `main` method is the entry point into your application.

- It is the starting point of program execution.
- The method name must be called `main`.
- It must be `public`. You intend to invoke this method from outside of this class.
- It must be `static`. Such methods can be invoked without creating an instance of this class.
- It must be `void`. It does not return a value.
- It must accept array of String objects as the only parameter.
  (The name of this parameter "`args`" is irrelevant.)

```
package demos;
public class Whatever {
  public static void main(String[] args) {
    // program execution starts here
  }
}
```

✣ Note: Use of `static` and `void` keywords and handling of arrays are covered later in the course.

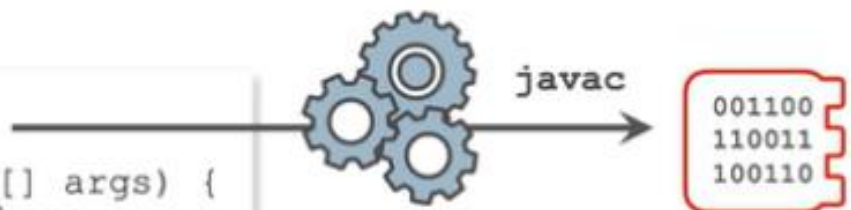
Compile Java Program

Compile classes with the **`javac`** Java compiler.

- The `-classpath` or `-cp` parameter points to locations of other classes that may be required to compile your code.
- The `-d` parameter points to a path to store compilation result.
  (The compiler creates package subfolders with compiled class files in this path.)
- Provide path to source code.

```
javac -cp /project/classes -d /project/classes /project/sources/demos/Whatever.java
```

```
package demos;
public class Whatever {
  public static void main(String[] args) {
    // program execution starts here
  }
}
```




```
/project/classes/demos/Whatever.class
```

Execute Java Program

Execute program using **`java`** executable Java Virtual Machine (JVM).

- Specify `-classpath` or `-cp` to point to folders where your classes are located.
- Specify fully qualified class name. Use package prefix; do not use the `.class` extension.
- Provide a space separated list of parameters after the class name.

Access command-line parameters:

- Use array object to access parameters.
- Array index starts at 0 (first parameter).

`java`

```
package demos;
public class Whatever {
  public static void main(String[] args) {
    String param1 = args[1];
    System.out.println("Hello "+param1);
  }
}
```

```
java  -cp /project/classes demos.Whatever Jo John "A Name" Jane
Hello John
```

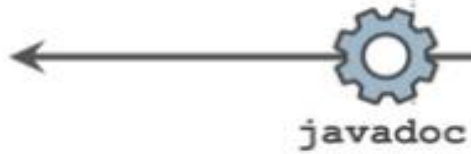
- Since Java 11, it is also possible to run single-file source code as if it is a compiled class.
  JVM will interpret your code, but no compiled class file would be created:

```
java /project/sources/demos/Whatever.java
```

Comments and Documentation

- Code Comments can be placed anywhere in your source code.
- Documentation Comments:
  - May contain HTML markups
  - May contain descriptive tags prefixed with @ sing
  - Are used by the `javadoc` tool to generate documentation

```
// single-line comment

/*
   multi-line comment
*/

/**
* The {@code Whatever} class
* represents an example of
* documentation comment
* @version 1.0
* @author oracle
*/
```



Package demos

Class Whatever

java.lang.Object
    demos.Whatever

public class **Whatever**
extends Object

The Whatever class represents an example of documentation comment

Version:

1.0

Author:

oracle

```
javadoc -d <documentation path>
        -sourcepath <source code path>
        -subpackages <name of the root package>
```

- Note: All APIs in the Java development kit are documented using the javadoc utility.

Source code is saved in ____ files, and code is compiled into ____ files.

- .java, .exe
- .java, .bytes
- .text, .java
- .java, .class

Which is an invalid variable name?

- _SystemValue
- MyNewValue
- 4ScoreAnd8Years
- $_exceptionValue
- object

Which statements are true about Java? (Choose two)

- Java is a specialized programming language for use in browsers.
- Java is an object-oriented programming language.
- Java source code must be compiled into classes for each platform.
- Java source code is plain text.

Objectives

After completing this lesson, you should be able to:

- Describe primitive types
- Describe operators
- Explain primitives type casting
- Use Math class
- Implement flow control with `if/else` and `switch` statements
- Describe JShell

Java Primitives

Java language provides eight primitive types to represent simple numeric, character, and boolean values.

Whole numbers			
byte	**short**	**int**	**long**
8 bits	16 bits	32 bits	64 bits
-128	-32,768	-2,147,483,648	-9,223,372,036,854,780,000
127	32,767	2,147,483,647	9,223,372,036,854,780,000
default value `0` or `0L`			
Binary `0b1001` Octal `072` Decimal `1234` Hex `0x4F` Upper or lowercase `L` at the end indicates `long` value			

Floating point numbers	
float	**double**
32 bits	64 bits
1.4E-45	4.9E-324
3.4028235E+38	1.7976931348623157E+308
default value `0.0F` or `0.0`	
Normal `123.4` or exponential notations `1.234E2`; Upper or lowercase `F` at the end indicates `float` value	

Boolean
boolean
default value `false`
`true` or `false`

Character (represents single character value)
char
16 bits
0
65,535
default value `'\u0000'`
Character `'A'` ASCII code `'\101'` Unicode `'\u0041'` Escape Sequences: tab `'\t'` backspace `'\b'` new line `'\n'` carriage return `'\r'` form feed `'\f'` single quote `\'` double quote `\"` backslash `\\`

Declare and Initialize Primitive Variables

Primitive declaration and initialization rules:

- Variable declaration and initialization syntax:
  `<type> <variable name> = <value>;`
- A variable can be declared with no immediate initialization, so long as it is initialized before use.
- Numeric values can be expressed as binary, octal, decimal, and hex.
- Float and double values can be expressed in normal or exponential notations.
- Multiple variables of the same type can be declared and initialized simultaneously.
- Assignment of one variable to another creates a copy of a value.
- Smaller types are automatically promoted to bigger types.
- Character values must be enclosed in single quotation marks.

```
int a = 0b101010; // binary
short b = 052;    // octal
byte c = 42;      // decimal
long d = 0x2A;    // hex
float e = 1.99E2F;
double f = 1.99;
long g = 5, h = c;
float i = g;
char j = 'A';
char k = '\u0041', l = '\101';
int s;
s = 77;
```

Restrictions on Primitive Declarations and Initializations

Primitive declaration and initialization restrictions:

- Variables must be initialized before use.
- A bigger type value cannot be assigned to a smaller type variable.
- Character values must not be enclosed in double quotation marks.
- A character value cannot contain more than one character.
- Boolean values can be expressed only as `true` or `false`.

```
byte a;
byte b = a;
byte c = 128;
int d = 42L;
float e = 1.2;
char f = "a";
char g = 'AB';
boolean h = "true";
boolean i = 'false';
boolean j = 0;
boolean k = False;
```

✣ Note: Each incorrect example given here would cause Java code not to compile.

What are primitive data types ?

What are wrapper classes?

What is the diff b/w primitive and wrapper classes?

Java Operators

List of Java operators in the order of precedence:

Operators	Precedence
postfix increment and decrement	`++ --`
prefix increment and decrement, and unary	`++ -- + - ~ !`
multiplicative	`* / %`
additive	`+ -`
bit shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Assignment and Arithmetic Operators

Assignments and arithmetics

```
= + - * / %
```

```
int a = 1;   // assignment (a is 1)
int b = a+4; // addition (b is 5)
int c = b-2; // subtraction (c is 3)
int d = c*b; // multiplication (d is 15)
int e = d/c; // division (e is 5)
int f = d%6; // modulus (f is 3)
```

Compound assignments are combinations of an operation and an assignment that is acting on the same variable.

```
+= -= *= /= %=
```

```
int a = 1, b = 3;
a += b; // equivalent of a=a+b (a is 4)
a -= 2; // equivalent of a=a-2 (a is 2)
a *= b; // equivalent of a=a*b (a is 6)
a /= 2; // equivalent of a=a/2 (a is 3)
a %= a; // equivalent of a=a%a (a is 0)
```

Operator evaluation order can be changed using round brackets.

```
( )
```

```
int a = 2, b = 3;
int c = b-a*b;   // (c is -3)
int d = (b-a)*b; // (c is 3)
```

Increment and decrement operators has prefix and postfix positions:

`y=++x` x is incremented first and then the result is assigned to y.

`y=--x` x is decremented first and then the result is assigned to y.

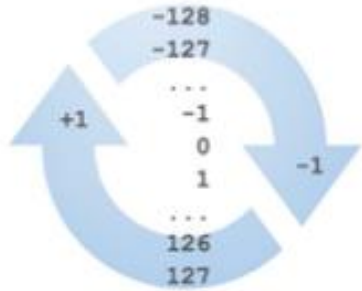
`y=x++` y is assigned the value of x first and then x is incremented.

`y=x--` y is assigned the value of x first and then x is decremented.

```
++ --
```

```
int a = 1, b = 0;
a++;     // increment (a is 2)
++a;     // increment (a is 3)
a--;     // decrement (a is 2)
--a;     // decrement (a is 1)
b = a++; // increment postfix (b is 1, a is 2)
b = ++a; // increment prefix  (b is 3, a is 3)
b = a--; // increment postfix (b is 3, a is 2)
b = --a; // increment prefix  (b is 1, a is 1)
```

Arithmetic Operations and Type Casting



Rules of Java arithmetic operations and type casting:

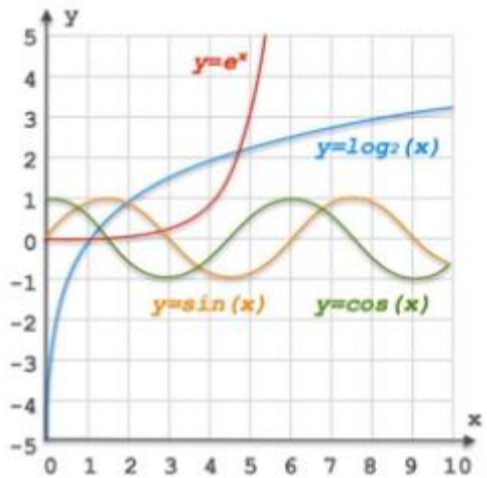
- Smaller types are automatically casted (promoted) to bigger types.
  `byte->short->char->int->long->float->double`
- A bigger type value cannot be assigned to a smaller type variable without explicit type casting.
- Type can be explicitly casted using the following syntax: `(<new type>)<variable or expression>`
- When casting a bigger value to a smaller type, beware of a possible overflow.
- Resulting type of arithmetic operations on types smaller than `int` is an `int`; otherwise, the result is of a type of a largest participant.

```
   byte a = 127, b = 5;
⊗  byte c = a+b;            // compilation fails
             int
✓  int d = a + b;           // d is 132
△  byte e = (byte)(a+b);    // e is -124  (type overflow, because 127 is the max byte value)
△  int f = a/b;             // f is 25    (a/b is 25 because it is an int)
△  float g = a/b;           // g is 25.0F (result of the a/b can be implicitly or
△  float h = (float)(a/b);  // h is 25.0F  explicitly casted to float, but a/b is still 25)
✓  float i = (float)a/b;    // i is 25.4F (when either a or b
✓  float j = a/(float)b;    // j is 25.4F  is float the a/b becomes float)
△  b = (byte)(b+1);         // explicit casting is required, because b+1 is an int
✓  b++;                     // no casting is required for ++ and -- operators
   char x = 'x';
✓  char y = ++x;            // arithmetic operations work with character codes
```

More Mathematical Operations

Class `java.lang.Math` provides various mathematical operations:

- Exponential such as e^x
- Logarithmic such as $\log_2(x)$
- Trigonometric such as `sin(x)` `cos(x)`
- and many more...



Examples of Math functions:

```
double a = 1.99, b = 2.99, c = 0;
c = Math.cos(a);     // cosine
c = Math.acos(a);    // arc cosine
c = Math.sin(a);     // sine
c = Math.asin(a);    // arc sine
c = Math.tan(a);     // tangent
c = Math.atan(a);    // arc tangent
c = Math.exp(a);     // e^a
c = Math.max(a,b);   // greater of two values
c = Math.min(a,b);   // smaller of two values
c = Math.pow(a,b);   // a^b
c = Math.sqrt(a);    // square root
c = Math.random();   // random number 0.0>=c<1.0
```

Numeric rounding example :

```
int a = 11, b = 3;
long c = Math.round(a/b);                          // c is 3
double d = Math.round(a/b);                        // d is 3.0
double e = Math.round((double)a/b*100)/100.0;      // e is 3.67
```

Binary Number Representation

All Java numeric primitives are signed *(that is, could represent positive and negative values).*

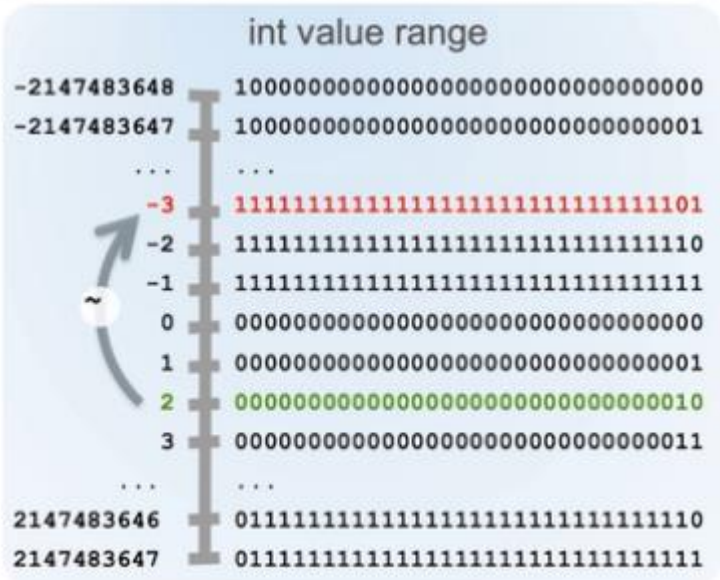
- Java uses a two's complement implementation of a signed magnitude representation of an integer.
- For example, a byte zero value is represented as the `00000000` sequence of bits.
- Changing a sign (negative or positive) is done by inverting all the bits and then adding one to the result.
- For example, a byte value of one is represented as `00000001` and minus one is `11111111`.

Bitwise Complement operator inverts all bits of a number:

The result of the bitwise compliment operator **~a** would be its "mirrored" binary value: `-(a+1)`.

```
int a = 2;  //
int b = ~a; // b is -3
```

✣ The next slide shows the rest of the bitwise operators.



Bitwise Operators

Compare corresponding bits of two operands with bitwise operators:

- ***Bitwise AND*** `&` when both bits are `1`, the result is `1`, or either of the bits is not `1`, the result is `0`.
- ***Bitwise OR*** `|` when either of the bits is `1`, the result is `1`; otherwise, the result is `0`.
- ***Bitwise Exclusive OR*** `^` when corresponding bits are different, the result is `1`; otherwise, the result is `0`.

```
byte a = 5;              // 00000101
byte b = 3;              // 00000011
byte c = (byte)(a & b); // 00000001 (c is 1)
byte d = (byte)(a | b); // 00000111 (d is 7)
byte e = (byte)(a ^ b); // 00000110 (e is 6)
```

Shift bits to the left or right with bitwise operators:

- ***Signed Left Shift*** `<<` shifts each bit to the left by specified number of positions, fills low-order positions with `0` bit values
- ***Signed Right Shift*** `>>` shifts each bit to the right by specified number of positions
- ***Unsigned Right Shift*** `>>>` same as above, but fills high-order positions with `0` bit values

```
int a =  5;       // 00000000000000000000000000000101
int b = -5;       // 11111111111111111111111111111011
int c =  a << 2;  // 00000000000000000000000000010100 (c is 20)
int d =  b << 2;  // 11111111111111111111111111101100 (d is -20)
int e =  a >> 2;  // 00000000000000000000000000000001 (e is 1)
int f =  b >> 2;  // 11111111111111111111111111111110 (f is -2)
int g =  a >>> 2; // 00000000000000000000000000000001 (e is 1)
int h =  b >>> 2; // 00111111111111111111111111111110 (h is 1073741822)
```

Short-Circuit Evaluation

Short-circuit evaluation enables you to not evaluate the right-hand side of the AND and OR expressions, when the overall result can be predicted from the left-hand side value.

`&&` `||` *(short-circuit evaluation)*

`&` `|` `^` *(full evaluation)*

```
 true && evaluated
false && not evaluated
false &  evaluated
false || evaluated
 true || not evaluated
 true |  evaluated
 true ^  evaluated
false ^  evaluated
```

```
int a = 3, b = 2;
boolean c = false;
c = (a > b && ++b == 3); // c is true,  b is 3
c = (a > b && ++b == 3); // c is false, b is 3
c = (a > b || ++b == 3); // c is false, b is 4
c = (a < b || ++b == 3); // c is true,  b is 4
c = (a < b |  ++b == 3); // c is true,  b is 5
```

✣ Note: It is not advisable to mix boolean logic and actions in the same expression.

Flow Control Using `if/else` Construct

Conditional execution of the algorithm using the `if/else` construct:

- The `if` code block is executed when the boolean expression yields true; otherwise `else` block is executed.
- The `else` clause is optional.
- There is no `else/if` operator in Java, but you can embed an `if/else` inside another `if/else` construct.

```
if (<boolean expression>) {
  /* logic to be executed when
     if expression yields true */
} else {
  /* logic to be executed when
     if expression yields false */
}
```

- Note: Compilation fails because a block containing more than one statement must be enclosed with `{}`.

```
int a = 2, b = 3;
if (a > b)
  a--;
  b++;
else
  a++;
```

```
int a = 2, b = 3;
if (a > b) {      // is false
  a--;            // not executed
} else {          // algorithm enters else block
  if (a < b) {    // is true
    a++;          // a is 3
  } else {        // this else block is not executed
    b++;          // not executed
  }
}
```

- Note: It is optional to put curly brackets `{}` around `if` or `else` blocks of code, when they contain only a single statement. This code fragment is identical to the example above, but `{}` omissions could make it harder to read.

```
int a = 2, b = 3;
if (a > b)
  a--;
else if (a < b)
  a++;
else
  b++;
```

- Note: Carriage returns and indentations in Java improve readability, but are irrelevant from the compiler perspective.

Ternary Operator

The ternary operator is used to perform conditional assignment.

- You can use the ternary operator `?:` instead of writing an `if/else` construct, if you only need to assign a value based on a condition.
- When the boolean expression yields true, value after the `?` is assigned. When the boolean expression yields false, value after the `:` is assigned.

```
<variable> = (<boolean expression>) ? <value one> : <value two>;
```

```
int a = 2, b = 3;
int c = (a >= b) ? a : b; // c is 3
```

These constructs produce identical results.

```
int a = 2, b = 3;
int c = 0;
if (a >= b) {
  c = a;
} else {
  c = b;
}
```

- Note: The ternary operator should be used to simplify conditional assignment logic. Do not use it instead of `if/else` statements to perform other actions, as it can make your code less readable.

```
int a = 2, b = 3;
int c = (a >= b) ? a : (--b == a) ? a : b; // c is 2
```

Flow Control Using `switch` Construct

Control program flow using the `switch` construct.

- Switch expression must be of one of the following types:
  `byte, short, int, char, String, enum`
- Case labels must match the expression type.
- Execution flow proceeds to the case in which the label matches the expression value.
- Execution flow continues until it reaches the end of switch or encounters an optional break statement.
- If the switch expression did not match any of the cases, then the default case is executed. It does not have to be the last case in the sequence and it is optional.

```
switch (<expression>) {
  case <label>:
    <case logic>
  case <label>:
    <case logic>
    break;
  default:
    <case logic>
}
```

Example cases:

- Special, increase price by 1
- New, increase price by 2
- Discounted, decrease price by 4
- Expired, set price to 0
- Any other, set price to 3

Execution path within the switch when `status` is `'N'` (New)

```
char status = 'N';
double price = 10;
switch (status) {
  case 'S':
    price += 1; // not executed
  case 'N':
    price += 2; // price is 12
  case 'D':
    price -= 4; // price is 8
    break;
  case 'E':
    price = 0;  // not executed
    break;
  default:
    price = 3;  // not executed
}
//the rest of the program logic
```

✣ Note: Strings and enums are covered later in the course.

JShell

- JShell is an interactive Read-Evaluate-Print Loop (REPL) command-line tool.
- Its purpose is to help to learn Java programming language and prototype Java code.
- It evaluates declarations, statements, and expressions as they are entered.
- It shows the results immediately.

JShell use example:

```
$ jshell
jshell> int x = 1
x ==> 1
jshell> int y = 1
y ==> 1
jshell> x+y
$3 ==> 2
jshell>/exit
$
```

JShell command reference:

```
/list      list the source you have typed
/edit      edit a source entry
/drop      delete a source entry
/save      save snippet source to a file
/open      open a file as source input
/vars      list the declared variables and their values
/methods   list the declared methods and their signatures
/types     list the type declarations
/imports   list the imported items
/exit      exit the jshell tool
/env       view or change the classpath or modules context
/reset     reset the jshell tool
/reload    reset and replay relevant history
/history   history of what you have typed
/help      or /? get information about using the jshell tool
/set       set configuration information
/!         rerun last snippet
/<id>      rerun snippets by ID or ID range
/-<n>      rerun n-th previous snippet
```

Why would you use the ternary operator ?: instead of writing an if/else construct?

- If you only need to assign a value based on a condition
- When the boolean expression is not to be evaluated
- None of the options listed
- If you run out of room
- To change the value returned

Which is not a Java primitive type?

- boolean
- float
- short
- String
- long

What is the correct evaluation of this expression:
8*8/2+2-3*2?

- 28
- 10
- 30
- 26

Which is the correct way to declare and initialize a variable?

- `int age;`
- `char midInit = "D";`
- `age = 42;`
- `boolean x = 4>5;`

Examine the following code:

```
int x = 1, y = 1, z = 0;
if (x == y | x < ++y) {
        z = x+y;
}
else{
        z = 1;
}
System.out.println(z);
```

What would be printed?

- 1
- 2
- 3

Text, Date, Time, and Numeric Objects

Objectives

After completing this lesson, you should be able to:

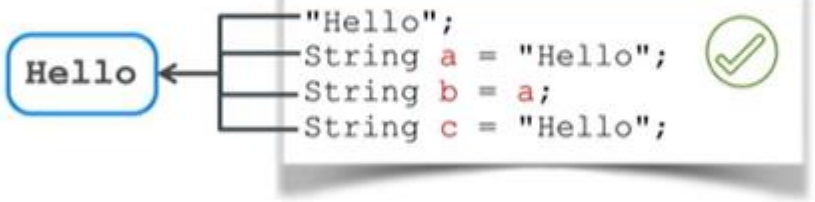
- Manipulate text values using String and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers using BigDecimal class
- Handle date and time values
- Describe Localization and Formatting classes

String Initialization

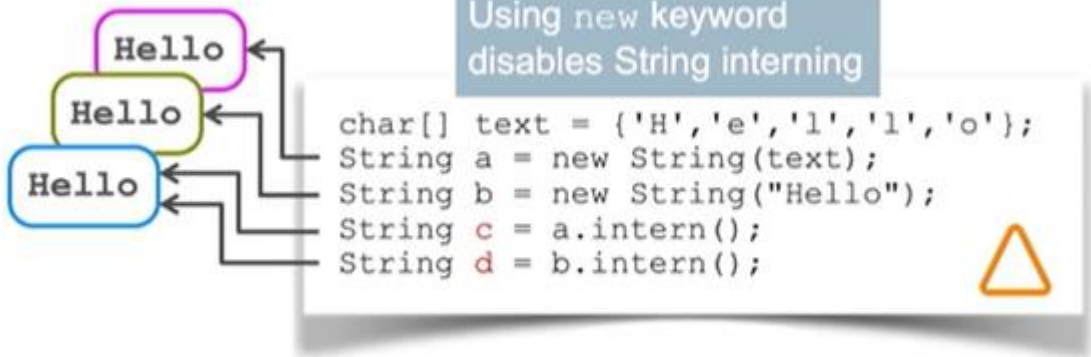
The `java.lang.String` class represents a sequence of characters.

- String is a class (not a primitive). Its instances represent sequences of characters.
- Just like any other Java object, the String object can be instantiated by using the `new` keyword.
- However, String is the only Java object that allows simplified instantiation as a text value enclosed with double quotes: `"some text"` and that is a recommended approach.
- JVM can optimize memory allocated to store String objects by maintaining a single copy of each String literal in the String Pool memory area, regardless of how many variables reference this copy. (This process is called interning.)
- The `intern()` method returns a reference to an interned (single) copy of a String literal.

A simple `"<value>"` initialization of String object is recommended



Using `new` keyword disables String interning



- Reminder: A primitive `char` represents a single character. Its values are enclosed in single quotes: `'a'`.
- Note: Example uses a char array `char[];` arrays are covered later in the course.

String Operations

String objects are immutable.

- Once a string object is initialized, it cannot be changed.
- String operations such as `trim()`, `contact()`, `toLowerCase()`, `toUpperCase()`, and so on would always return a new String, but would not modify the original String.
- It is possible to reassign the string reference to point to another string object.
- For convenience reasons, String allows the use of the + operator instead of the `concat()` method. However, remember that + is also an arithmetic operator.


```
String s = "";
s=1+1+"u";    // s is 2u
s="u"+1+1;    // s is u11
s="u"+(1+1); // s is u2
```

String Indexing

String contains a sequence character indexed by integer.

- String index starts from `0`.
- When getting a `substring` of a string, the begin index is inclusive of the result, but end index is not.
- If a substring is not found, the `indexOf` method returns `-1`.
- Both `indexOf` and `lastIndexOf` operations are overloaded (have more than one version) and accept a char or a String parameter and may also accept a search starting from the index position.
- An attempt to access text beyond the last valid index position (length-1) will produce an exception.


```
String a = "HelloWorld";
String b = a.substring(0,5); // b is "Hello"
int c = a.indexOf('o');      // c is 4
int d = a.indexOf('o',5);    // d is 6
int e = a.lastIndexOf('l');  // e is 8
int f = a.indexOf('a');      // f is -1
char g = a.charAt(0);        // g is H
int h = a.length();          // h is 10
char i = a.charAt(10);
```

throw StringIndexOutOfBoundsException

✣ Note: Exception handling is covered later in the course.

StringBuilder: Introduction

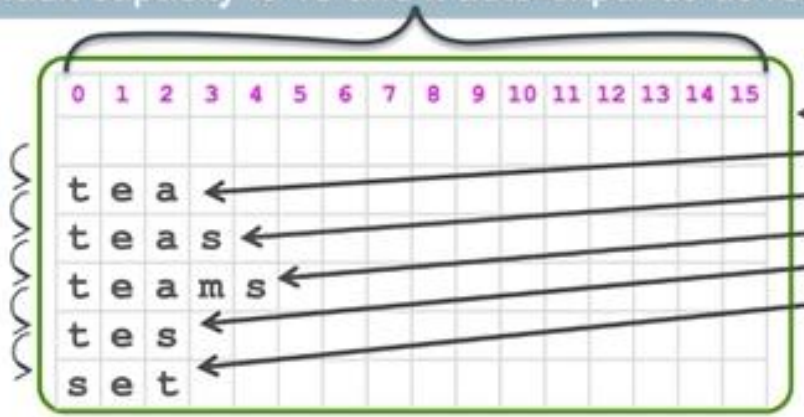
Another way of handling text in Java is with the `java.lang.StringBuilder` class.

- StringBuilder objects are mutable, allowing modifications of the character sequences they store.
- Handling text modifications with StringBuilder reduces the number of string objects you need to create.
- Some methods such as `substring`, `indexOf`, `charAt` are identical to that of a String class.
- Extra methods are available: `append`, `insert`, `delete`, `reverse` accepting String or char parameters.
- Sequence of characters must be continuous. It may contain spaces, but you may not leave gaps of no characters at all.
- Like other classes, StringBuilder objects are instantiated using the `new` keyword.
- You may instantiate a StringBuilder with predefined content or capacity.

```
new StringBuilder();
new StringBuilder("text");
new StringBuilder(100);
```

Default capacity is 16 and it auto-expands as required.

Content changes within the StringBuilder object.




```
StringBuilder a = new StringBuilder();
a.append("tea");
a.append('s');
a.insert(3,'m');
a.delete(2,4);
a.reverse();
int length = a.length();      // 3
int capacity = a.capacity(); // 16
a.insert(4,'s');
throw StringIndexOutOfBoundsException
```

Wrapper Classes for Primitives

Wrapper classes apply object-oriented capabilities to primitives.

- A wrapper class is capable of holding a primitive value provided for every Java primitive.
- Construct wrapper object out of primitive or string using the `valueOf()` methods.
- Extract primitive values out of the wrapper using the `xxxValue()` methods.
- Instead of formal conversion of wrapper to primitive and back, you can use direct assignment known as auto-boxing and auto-unboxing.
- Create wrapper or primitive out of the string using the `parseXXX()` methods.
- You may convert a primitive to a string using the `String.valueOf()` method.
- Wrapper classes provide constants, such as min and max values for every type.

```
int a = 42;
Integer b = Integer.valueOf(a);
int c = b.intValue();
b = a;
c = b;
String d = "12.25";
Float e = Float.valueOf(d);
float f = d.parseFloat(d);
String b = String.valueOf(a);
Short.MIN_VALUE;
Short.MAX_VALUE;
```

Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Notes

- Advanced text formatting and parsing is covered later.
- Avoid too many auto-boxing and auto-unboxing operations for performance reasons.

What primitives don't have? Why would you want to use wrapper classes? Primitives don't have operations. You can apply operators to them, but they don't have operations. That don't have methods.
They're just simple values. That's it. They don't have behaviors beyond what the value they have.
Wrapper classes do. So they give you actually some operations.

For example, you may take the value of primitive or text, and using the valueOf method produce an actual instance of wrapper, or the other way around maybe. Maybe if you have a wrapper object, then you can extract a primitive out of it. That would be some value method, like int value, double value, well, depending on the type that you want to get.

You don't actually have to do the formal conversion between the wrapper and a primitive. You can just do the direct assignment instead that's called auto-box and **auto-unboxing.**

So for example, a more formal approach is to say, I've got an int a is 42, and I, using the valueOf method, convert it to an integer. And if I want to convert it back, like that integer b I want to convert to primitive c, then I just call the method int value.

Representing Numbers Using `BigDecimal` Class

The `java.math.BigDecimal` class is useful in handling decimal numbers that require exact precision.

- All primitive wrappers and `BigDecimal` are immutable and signed
  (cannot be changed and may represent positive or negative numbers).
- However, unlike other numeric wrapper classes, `BigDecimal` has arbitrary precision
  (for example, `Double` has a limited precision as a binary 64 bit number).
- It is designed to work specifically with decimal numbers and provide convenient `scale` and `round` operations.
- It provides arithmetic operations as methods such as `add`, `subtract`, `divide`, `multiply`, `remainder`.
- It is typically used to represent decimal number that require exact precision, such as fiscal values.

```
BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                  // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

Method Chaining

When an operation returns an object you may invoke next operation upon this object immediately

- It is possible to chain method invocations technique with any operation that returns an object.
- Examples:
  - Arithmetic operations of `BigDecimal` return `BigDecimal` objects.
  - Text manipulating operations of `String` return `String` objects.

```
String s1 = "Hello";
Strung s2 = s1.concat("World").substring(3,6); // s2 is "loW"

BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                  // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

- Note: Method chaining is a common coding technique used when a method returns an object, followed by an invocation of another method upon that object and so on...
- Without method chaining, code appears to be cluttered with unnecessary intermediate variables:

```
BigDecimal taxedPrice = price.add(tax);
price = taxedPrice.setScale(2,RoundingMode.HALF_UP);
```

Now, there was one thing that you probably already noticed happening in this chapter, and that was something we did with strings, that was something we did with BigDecimals,
and you see it all over the place in Java. This is called method chaining.

Now, if you invoke an operation upon an object, so this is the string, and you call the method concat.
What does the method concat return? It returns another string.
You concatenate one string to another. You produce another string. So the return type of concat is text string.

Can you call concat upon the string? Yes, sure. So you can call concat again. Actually, you may call any operation upon t he results. So if I invoke a method, invoke an operation, and that operation results in another object, I can just keep cal ling next method and next method and the next method, so long as the method I invoke actually results in another obj ect. And this is a good approach. This is a good way of doing things.
So I have this string **"hello,"** and there's several things I want to do then. I want to concat it with world, I want to take a substrate. And I don't need to create intermediate variables to store these intermediate strings. I can just call dot and call next month, dot next method until I get a result that I desire.

With BigDecimal, exact same story. So I got a couple of BigDecimal variables. I guess I can call operation multiplying, ta ke price, multiply the tax rate, and that will be my new BigDecimal tax. Sure, I can do that.
But actually, I don't have to do that. I could do this multiplication and say, for example, subsequent addition all within t he same line of code. I don't need to create an intermediate variables. I could just say, price, multiple tax rate, wrap th at up, this piece of code. Instead of this variable text, just put it inside a method add. And then say price.add that, and then call the .setScale.

Local Date and Time

Local date and time API is used to handle date and time values.

- Classes `LocalDate`, `LocalTime`, and `LocalDateTime` from `java.time` package
- Date and time objects can be created using methods:
  `now()` to get current date and time, or
  `LocalDateTime.of(year, month, day, hours, minutes, seconds, nanoseconds)`
  `LocalTime.of(hours, minutes, seconds, nanoseconds)`
  `LocalDate.of(year, month, day)` for specific date and time or
  combining other date and time objects using `atTime()` and `of(localDate, localTime)` or
  extracting date and time portions from LocalDateTime using `toLocalDate()` and `toLocalTime()`

```
LocalDate today = LocalDate.now();
LocalTime thisTime = LocalTime.now();
LocalDateTime currentDateTime = LocalDateTime.now();
LocalDate someDay = LocalDate.of(2019,Month.APRIL,1);
LocalTime someTime = LocalTime.of(10,6);
LocalDateTime otherDateTime = LocalDateTime.of(2019,Month.MARCH,31,23,59);
LocalDateTime someDateTime = someDay.atTime(someTime);
LocalDate whatWasTheDate = someDateTime.toLocalDate();
```

✣ Note: Date and Time API (`java.time` package) was introduced in Java SE 8.
Earlier Java versions used `java.util.Date` class to represent date and time values.

More Local Date and Time Operations

Characteristics of local date and time operations

- Local Date and Time objects are immutable.
- All date and time manipulation methods will produce new date and time objects.
- New date and time objects can be produced out of existing objects using `plusXXX()` or `minusXXX()` or `withXXX()` methods.
- It is possible to chain method invocations together, because all date and time manipulation operations return date and time objects.
- Individual portions of date and time objects can be retrieved with `getXXX()` methods.
- Operations `isBefore()` and `isAfter()` check if a date or time is before or after another.

```
LocalDateTime current = LocalDateTime.now();
LocalDateTime different =  current.withMinute(14).withDayOfMonth(3).plusHours(12);
int year = current.getYear();
boolean before = current.isBefore(different);
```

- Reminder: Method chaining helps to avoid cluttering code with unnecessary intermediate variables.

Instants, Durations, and Periods

In addition to dates and times, the API can also represent periods and durations of time.

- The `java.time.Duration` class can represent an amount of time in nanoseconds.
- The `java.time.Period` class can represent an amount of time in units such as years or days.
- The `java.time.Instant` class can represent an instantaneous point on the time-line (time-stamp).
- Just like the rest of the Local Date and Time API, Duration, Period, and Instant objects are immutable.
- Provide methods such as `now()`, `ofXXX()`, `plusXXX()`, `minusXXX()`, `withXXX()`, and `getXXX()`.
- Provide methods such as `between()`, `isNegative()` to handle distances between points in time.
- Use identical coding techniques such as method chaining.

```
LocalDate today = LocalDate.now();
LocalDate foolsDay = LocalDate.of(2019, Month, APRIL, 1);
Instant timeStamp = Instant.now();
int nanoSecondsFromLastSecond = timeStamp.getNano();
Period howLong = Period.between(foolsDay, today);
Duration twoHours = Duration.ofHours(2);
long seconds = twoHours.minusMinutes(15).getSeconds();
int days = howLong.getDays();
```

- Note: Instant and Duration are more suitable for implementing system tasks, such as using a timestamp when writing logs. Period is more suitable for implementing business logic.

Zoned Date and Time

Time zones can be applied to local date and time values.

The `java.time.ZonedDateTime` class:

- Represents date and time values according to time zone rules
- Has the same time management capabilities as LocalDateTime
- Provides time zone specific operations such as `withZoneSameInstant`
- Accounts for daylight saving time and time zone differences

```
ZoneId london = ZoneId.of("Europe/London");
ZoneId la   = ZoneId.of("America/Los_Angeles");
LocalDateTime someTime = LocalDateTime.of(2019,Month.APRIL,1,07,14);
ZonedDateTime londonTime = ZonedDateTime.of(someTime, london);
ZonedDateTime laTime = londonTime.withZoneSameInstant(la);
```

- The `java.time.ZoneId` class defines time zones.
- Timezone can be set as:

```
ZoneId.of("America/Los_Angeles");
ZoneId.of("GMT+2");
ZoneId.of("UTC-05:00");
ZoneId.systemDefault();
```

Represent Languages and Countries

Java provides APIs to make your application adjustable to different languages and locations around the world.

- The `java.util.Locale` class represents languages and countries.
- The ISO 639 language and ISO 3166 or country codes or UN M.49 area codes are used to set up locale objects.
- Locale can represent just language or a combination of language plus country or area.
- Variant is an optional parameter, designed to produce custom locale variations.
- Language tag string allows constructing locales for various calendars, numbering systems, currencies, and so on.

```
                                                Language Country Variant
Locale uk = new Locale("en","GB");          // English  Britain
Locale uk = new Locale("en","GB","EURO"); // English  Britain Euro (custom variant)
Locale us = new Locale("en","US");          // English  America
Locale fr = new Locale("fr","FR");          // French   France
Locale cf = new Locale("fr","CA");          // French   Canada
Locale fr = new Locale("fr","029");         // French   Caribbean
Locale es = new Locale("fr");               // French
Locale current = Locale.getDefault();      // current default locale
// Example constructing locale that uses Thai numbers and Buddhist calendar:
Locale th = Locale.forLanguageTag("th-TH-u-ca-buddhist-nu-thai");
```

Format and Parse Numeric Values

The `java.text.NumberFormat` class is used to parse and format numeric values.

- Works with any Java Number, including primitives, wrapper classes and BigDecimal objects

```
BigDecimal price = BigDecimal.valueOf(2.99);
Double tax = 0.2;
int quantity = 12345;
Locale locale = new Locale("en","GB");
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
String formattedPrice = currencyFormat.format(price);
String formattedTax = percentageFormat.format(tax);
String formattedQuantity = numberFormat.format(quantity);
```

value initializations

locale initialization

formatter initializations

formatting values

- Method `parse` return type is `Number` and it can be casted to numeric primitive wrappers or `BigDecimal` types

```
£2.99 20% 12,345
```

formatted results

```
BigDecimal newPrice = (BigDecimal)currencyFormat.parse("£1.75");
Double newTax = (Double)percentageFormat.parse("12%");
int newQuantity = numberFormat.parse("54,321").intValue();
```

parsing values

```
1.75 0.12 54321
```

parsed values

Format and Parse Date and Time Values

The `java.time.format.DateTimeFormatter` class is used to parse and format date and time values.

- You can set up custom format pattern or use standard format patterns defined by `java.time.format.FormatStyle` enum.

```
LocalDate date = LocalDate.of(2019,Month.APRIL,1);
Locale locale = new Locale("en","GB");
DateTimeFormatter format = DateTimeFormatter.ofPattern("EEEE dd MMM yyyy",locale);
String result = date.format(format);
```

value initialization
locale initialization
formatter initialization
format value
formatted result

```
Monday 01 Apr 2019
```

```
date = LocalDate.parse("Tuesday 31 Mar 2020",format);
locale = new Locale("ru");
format = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM).localizedBy(locale);
result = date.format(format);
```

parse value
reset locale
reset formatter
format value
formatted result

```
31 мар. 2020 г.
```

Localizable Resources

Resource bundles contain localizable resources.

- Resource bundle can be represented as plain text file with the extension `.properties`.
- Resources, for example, user-visible messages, are placed into resource bundles as `<key>=<value>`.
- Bundles may contain messages or message patterns with substitution parameters.
  (value substitutions are explained later)
- The `java.util.ResourceBundle` class loads bundles and retrieves resources.
- Default bundle can be used if no locale is specified `ResourceBundle.getBundle(<bindle name>)` or if the resource you're trying to get is not present in another (language and country specific) bundle.

```
Locale locale = new Locale("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages",locale);
String helloPattern = bundle.getString("hello");
String otherMessage = bundle.getString("other");
```

```
resources          (package folder)
  messages.properties
  messages_en_GB.properties
  messages_ru.properties
```

```
hello=もしもし {0}
product={0}, 価格 {1}, 分量 {2}, 賞味期限は {3}
other=他に何か
```

default bundle, can be in any language

```
hello=Hello {0}
product={0}, price {1}, quantity {2}, best before {3}
```

```
hello=Привет {0}
product={0}, цена {1}, количество {2}, годен до {3}
```

Format Message Patterns

Formatter classes parse and format messages, numbers, date and time values.

- The `java.text.MessageFormat` class substitutes values into message patterns.
- Message patterns can be stored in recourse bundles. For example, `resources/messages_en_GB.properties` containing product message pattern:

```
product={0}, price {1}, quantity {2}, best before {3}
```

- After all required values are formatted, they can be substituted into the message:

```
Locale locale = new Locale("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages",locale);
// assume following values are already formatted:
String name = "Cookie",
String price = currency.format(price);
String quantity = number.format(quantity);
String bestBefore = date.format(dateFormatter);

String pattern = bundle.getString("product");
String message = MessageFormat.format(pattern, name, price, quantity, bestBefore);
```

initialize locale and bundle

prepare formatters and values

get pattern

substitute values

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

formatted result

Formatting and Localization: Example

resources/messages_en_GB.properties

```
product={0}, price {1}, quantity {2}, best before {3}
```

```
String name = "Cookie";
BigDecimal price = BigDecimal.valueOf(2.99);
LocalDate bestBefore = LocalDate.of(2019,Month.APRIL,1);
int quantity = 4;

Locale locale = new Locale("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages",locale);

NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("dd MMM yyyy",locale);

String fPrice = currencyFormat.format(price);
String fQuantity = numberFormat.format(quantity);
String fBestBefore = dateFormat.format(bestBefore); // or bestBefore.format(dateFormat);

String pattern = bundle.getString("product");
String message = MessageFormat.format(pattern, name, fPrice, fQuantity, fBestBefore);
```

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

Non static variable cannot be accessed via static method

```
2  public class StaticDemo {
3      int a=100;
4      static int b=10;
5       void seta() {
6          a=1000;
7          System.out.println(b);
8      }
9       static void  show() {
0           System.out.println(a);
1       }
2       static {
3           System.out.println("hello");
4       }
```

static variable can be accessed via non-static method

```
2 public class StaticDemo {
3     int a=100;
4     static int b=10;
5      void seta() {
6         a=1000;
7         System.out.println(b);
8     }
9
```

1.Guess the answer

```
1
2  public class StaticBlockDemo {
3
4      static int a=10;
5
6      public static void main(String[] args) {
7
8          System.out.println(a);
9      }
10
11 }
12
```

2.Guess the answer

```
2 public class StaticBlockDemo {
3
4       int a=10;
5
6     public static void main(String[] args) {
7
8         System.out.println(a);
9     }
10
11 }
12
```

2.answer

```
2  public class StaticBlockDemo {
3
4      int a=10;
5
6      public static void main(String[] args) {
7
8          System.out.println(a);
9      }
10
11 }
12
```

Problems | Javadoc | Declaration | Console

<terminated> StaticBlockDemo [Java Application] C:\Users\matam\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (Sep 20, 2021, 7:45:57 PM –

```
xception in thread "main" java.lang.Error: Unresolved compilation problem:
        Cannot make a static reference to the non-static field a

        at StaticBlockDemo.main(StaticBlockDemo.java:8)
```

3.Guess the answer

```
1
2 public class StaticBlockDemo {
3
4     int a=10;
5
6     public static void main(String[] args) {
7
8         StaticBlockDemo obj= new StaticBlockDemo();
9         System.out.println(obj.a);
0
1     }
2
}
```

4.Guess the answer

```
public class StaticBlockDemo {

    int a=10;

    static void show() {
        System.out.println("show method in my class");
    }

    public static void main(String[] args) {

        StaticBlockDemo obj= new StaticBlockDemo();
        System.out.println(obj.a);
        StaticBlockDemo.show();
    }

}
```

5.Guess the answer

```
public class StaticBlockDemo {

    int a=10;
    static void show() {
        System.out.println("static show method in my class");
    }

    static {
        System.out.println("hello am static block");

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

        StaticBlockDemo obj= new StaticBlockDemo();
        StaticBlockDemo.show();
    }

}
```

6.Guess the answer

```
public class StaticBlockDemo {

	int a=10;

	static {
		System.out.println("hello am static block");
	}

	{
		a=100;
	}

    public static void main(String[] args) {
		StaticBlockDemo obj= new StaticBlockDemo();
		System.out.println(obj.a);

	}

}
```

Constructor

1.Guess the answer

```
2
3  public class ConstructorDemo {
4      int a=10;
5      ConstructorDemo(){
6          a=100;
7          System.out.println(a);
8      }
9      void construtorDemoMethod() {
10         System.out.println(a+a);
11     }
12     public static void main(String[] args) {
13         ConstructorDemo obj= new ConstructorDemo();
14     }
15 }
16
```

```
class InitDemo {
static int y;
int x;
//Static Initialization Block
static {
     y=10;
 }
// Instance Initialization Block
{
      x=20; }
}
```

1. Which variables can be accessed via class reference ?
2. Can we access non static variables inside static method ?
3. Static variables can be accessed inside non static method ?
4. Can we access Static variable inside static method ?
5. Can we access non static variable inside non static method ?
6. When does static block executes ?
7. Static method can be accessed via class reference ?

Instance initialization block code runs right after the call to **super()** in a constructor, in other words, after all super constructors have run.

The order in which initialization blocks appear in a class matters. If a class has more than one they all run in the order they appear in the class file.

Some rules to remember:

- Initialization blocks execute in the order they appear.
- Static Initialization blocks run once when the class is first loaded.
- Instance Initialization blocks run every time a class instance is created.
- Instance Initialization blocks run after the constructor's call to **super().**

Instance initializer block:

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Rules for instance initializer block :

There are mainly three rules for the instance **initializer** block. They are as follows:

1.The instance **initializer** block is created when instance of the class is created.

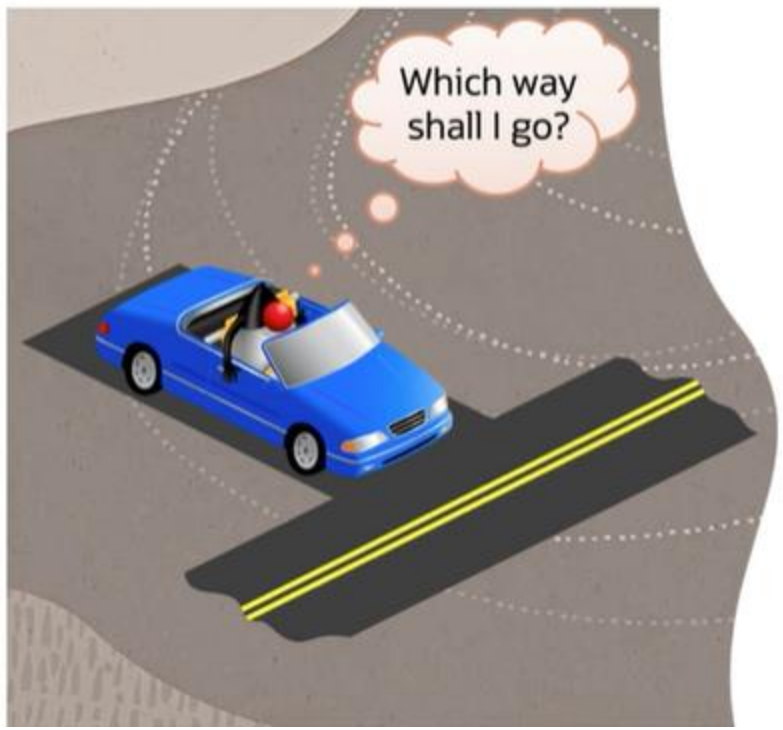
2.The instance **initializer** block is invoked after the parent class constructor is invoked (i.e. after **super()** constructor call).

3.The instance **initializer** block comes in the order in which they appear

Objectives

1. Create a simple `if/else` statement.
2. Describe the purpose of an array.
3. Declare and initialize a `String` or an `int` array.
4. Access the elements of an array.
5. Explain the purpose of loops.

Control Program Flow with the `if/else` Statement



boolean expression

```
if ( <some condition is true> ) {

   // do something

}
else {

       // do something different
       // the else block is optional

}
```

Common Conditional Operators

Operation	Operator	Example

If one condition AND another condition	&&	`int i = 2;`
		`int j = 8;`
		`((i < 1) && (j > 6))`

If either one condition OR another condition	||	`int i = 2;`
		`int j = 8;`
		`((i < 1) || (j > 10))`

NOT	!	`int i = 2;` `(!(i < 3))`

Examples

```
int attendees = 4;
boolean largeVenue = false;
int fee = 42;

if (attendees >= 5 && fee > 25){
        largeVenue = true;
}
//else {
//      largeVenue = false;
//}

// same outcome with less code
largeVenue = (attendees >= 5 && fee > 25);
```

Assign a boolean by using an `if` statement.

Assign the boolean directly from the boolean expression.

Ternary Conditional Operator

```
int x = 2, y = 5, z = 0;
```

Operation	Operator	Syntax
If `condition` is true, assign the value of `value1` to the result. Otherwise, assign the value of `value2` to the result.	**? :**	`condition ? value1 : value2`

Ternary Conditional Operator

```
int x = 2, y = 5, z = 0;
```

Operation	Operator	Syntax
If `condition` is true, assign the value of `value1` to the result. Otherwise, assign the value of `value2` to the result.	**? :**	`condition ? value1 : value2`

Equivalent constructs

```
if (y < x) {
    z = x;
} else {
    z = y;
}
```

```
z = (y < x) ? x : y;
```

Handling Flow Control with a `switch` Statement

The `switch` statement:

- Is easier to read and maintain
- Offers better performance

```
double price = 1.99;
double discount = 0;
String condition = "Used";
switch (condition) {
  case "Damaged":
    discount = price*0.1;
    break;
  case "Used":
    discount = price*0.2;
    break;
  default:
    discount = price;
}
```

Quiz

What is the purpose of the `else` block in an `if/else` statement?

a. To contain the remainder of the code for a method

b. To contain code that is executed when the expression in an `if` statement is false

c. To test if an expression is false

Answer

What is the purpose of the `else` block in an `if/else` statement? (b)

a. To contain the remainder of the code for a method

b. To contain code that is executed when the expression in an `if` statement is false

c. To test if an expression is false

Exercise 4-1: Using `switch` Statements

1. Add String property size to the Customer class:
   - Property name: `size`
   - Property Type: `String`
2. Use Customer in the `main` method of `ShopApp`.
3. In the `main` method of the ShopApp class, set size property for the customer, Pinky.
   - Place this logic immediately after the initialization of customer name.
   - Set customer `size` to be `"S"`.
4. Create a `measurement` variable of type `int` and set it to the value of `3`.
5. Add the `switch` statement that derives customer `size` based on a `measurement` value.

Arrays

Introduction to Arrays

index (int)	value (String)
0	Shirt
1	Jacket
2	Scarf

```
// Without an array
String itemDesc1 = "Shirt";
String itemDesc2 = "Trousers";
String itemDesc3 = "Scarf";

// Using an array
String[] items = {"Shirt","Trousers","Scarf"};
```

Not realistic for many items!

A more elegant solution!

Array: Examples

Array of int values

index (int)	value (int)
0	25
1	27
2	48

Array of String values

index (int)	value (String)
0	Mary
1	Bob
2	Pinky

```
String[] names = {"Mary", "Bob", "Pinky"};
int[] ages = {25, 27, 48};
```

Array: Contents

```
int[] ages1 = new int[3]; // all elements have a value of 0
ages1[0] = 19;
ages1[1] = 42;
ages1[2] = 92;

String[] names1 = new String[3]; // all elements have a null value
names1[0] = "Mary";
names1[1] = "Bob";
names1[2] = "Pinky";

// alternative approach
// Array is declared, initialized and populated at the same time
int[] ages2 = {19, 42, 92};
String[] names2 = {"Mary", "Bob", "Pinky"};
```

Accessing Array Elements

- Use values from the `ages` array:

```
int[] ages = {25, 27, 48};
int myAge = ages[0];
int muchOlder = ages[0] + 10;
```

- Use values from the `names` array:

```
String[] names = {"Mary", "Bob", "Pinky"};
String name = names[0];
int length = names[0].length();
names[2] = "Brain";
```

Using the `args` Array in the `main` Method

- Parameters can be typed on the command line:

```
> java ArgsTest Hello World!

First Parameter Hello
Second Parameter World!
```

Using the `args` Array in the `main` Method

- Parameters can be typed on the command line:

```
> java ArgsTest Hello World!

First Parameter Hello
Second Parameter World!
```

- Code for retrieving the parameters:

```
public class ArgsTest {
      public static void main (String[] args) {
            System.out.println("First Parameter " + args[0]);
            System.out.println("Second Parameter " + args[1]);
      }
}
```

Quiz

Why does the following code fail to compile?

```
int[] lengths = {2, 4, 3.5, 0, 40.04};
```

a. `lengths` cannot be used as an array identifier.

b. Not all elements of the array have been assigned a value.

c. The array was declared to hold `int` values. `double` values are not allowed.

Answer

Why does the following code fail to compile? (c)

```
int[] lengths = {2, 4, 3.5, 0, 40.04};
```

a. `lengths` cannot be used as an array identifier.

b. Not all elements of the array have been assigned a value.

c. The array was declared to hold `int` values. `double` values are not allowed.

Quiz

Given the following array declaration, which statements are true? (Choose all that apply.)

```
int[] classSize = {5, 8, 0, 14, 194};
```

a. `classSize[0]` is the reference to the first element in the array.

b. `classSize[5]` is the reference to the last element in the array.

c. There are five integers in the `classSize` array.

d. `classSize.length == 5`

Answer

Given the following array declaration, which statements are true? (a, c, d)

```
int[] classSize = {5, 8, 0, 14, 194};
```

a. `classSize[0]` is the reference to the first element in the array.

b. `classSize[5]` is the reference to the last element in the array.

c. There are five integers in the `classSize` array.

d. `classSize.length == 5`

Loops and Iteration

Control Program Flow with Loops

Loops are used in programs to repeat blocks of statements:

- Until an expression is false:
  - Processing all records until there are no more
- For a specific number of times:
  - Processing each element of an array

Control Program Flow with Loops

Loops are used in programs to repeat blocks of statements:

- Until an expression is false:
  - Processing all records until there are no more
- For a specific number of times:
  - Processing each element of an array


```
while (notThereYet) {

       read book;
  argue with sibling;
  ask, "Are we there yet?";

}

Woohoo!;
Get out of car;
```

Types of Loops

- A `while` loop repeats *while* a Boolean expression is true.
- A `do/while` loop executes at *least once*.
- A `for` loop simply repeats a *set number* of times.
- A `for each` is the enhanced `for` loop, which *automatically iterates* through a collection of values.

Note: Only `for`, and `for each` loops are covered in this course.

Coding a Standard `for` Loop

The standard `for` loop repeats its code block a set number of times using a counter.

- Syntax:

```
for(<type> counter = n; <boolean_expression>; <counter_increment>){
        code_block;
 }
```

- Example:

```
for(int i = 1; i < 5; i++){
     System.out.print("i is " +i +" ");
 }
```

Output: `i is 1, i is 2, i is 3, i is 4,`

Standard `for` Loop Compared to an Enhanced `for` Loop

index (int)	value (String)
0	Mary
1	Bob
2	Pinky

```
String [] names =  "Mary", "Bob", "Pinky";
//standard for loop
for (int idx = 0; idx < names.length; idx++) {
  System.out.println(names[idx]);
}
// enhanced for loop
for(String name: names) {
  System.out.println(name);
}
```

Using `break` with Loops

Terminate the loop using `break` when there is no need to proceed with remaining iterations:

```
int passmark = 12;
boolean passed = false;
int[] scores = {4,6,2,8,12,35,9};
for (int unitScore : scores) {
  if (unitScore >= 12) {
    passed = true;
    break;
  }
}
System.out.println("At least one passed? " +passed);
```

Quiz

Given:

```
int[] sizes = {4, 18, 5, 20};
  for (int size : sizes){
      if (size > 16){break;}
      System.out.println("Size: "+size + ", ");
  }
```

What is the output?

a. `Size: 4,`

b. `Size: 4`

c. `Size: 4,`
   `Size: 5,`

d. There is no output.

Answer

Given:

```
int[] sizes = {4, 18, 5, 20};
  for (int size : sizes){
      if (size > 16){break;}
      System.out.println("Size: "+size + ", ");
  }
```

What is the output? (a)

a. Size: 4,

b. Size: 4

c. Size: 4,
   Size: 5,

Will it work ?

```
11          int[] array1= new int[3];
12
13          array1[0]=100;
14          array1[1]=200;
15          array1[2]=300;
16          array1[3]=400;
17
18
19
20
```

```java
public class IntegerArrayDemo {

    public static void main(String[] args) {
        int[] array= {10,20,30,40,50,60,70,80};
        System.out.println(array.length);
        int[] array1= new int[3];
        array1[0]=100;
        array1[1]=200;
        array1[2]=300;
        array1[3]=400;
```

Problems | Javadoc | Declaration | Console | Progress

<terminated> IntegerArrayDemo [Java Application] C:\Users\matam\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (Sep 28, 2021, 7:50:24 PM – 7

```
8
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 o
        at arraysdemo.IntegerArrayDemo.main(IntegerArrayDemo.java:14)
```

```java
package arraysdemo;

public class StringArray {

    public static void main(String[] args) {

        String[] strArray= {"kk","csk","rcb","pkb","dc"};
        System.out.println(strArray.length);
        for(String s:strArray) {
            System.out.println(s);
        }
```

Problems | Javadoc | Declaration | Console | Progress | Coverage

<terminated> StringArray [Java Application] C:\Users\matam\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (Sep 28, 2021, 7:54:59 PM

```
5
kk
csk
rcb
pkb
dc
```

ORACLE
University

Java Basics

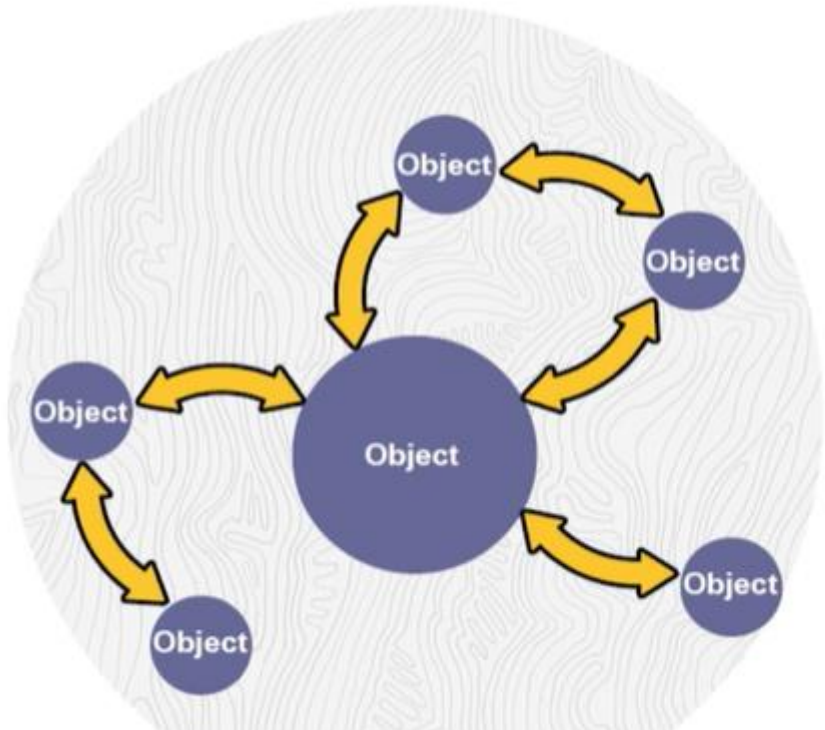
Defining Classes and Creating Objects

Objectives

1. List the characteristics of an object
2. Define an object as an instance of a class
3. Instantiate an object and access its fields and methods
4. Instantiate an array of objects
5. Describe how an array of objects is stored in memory

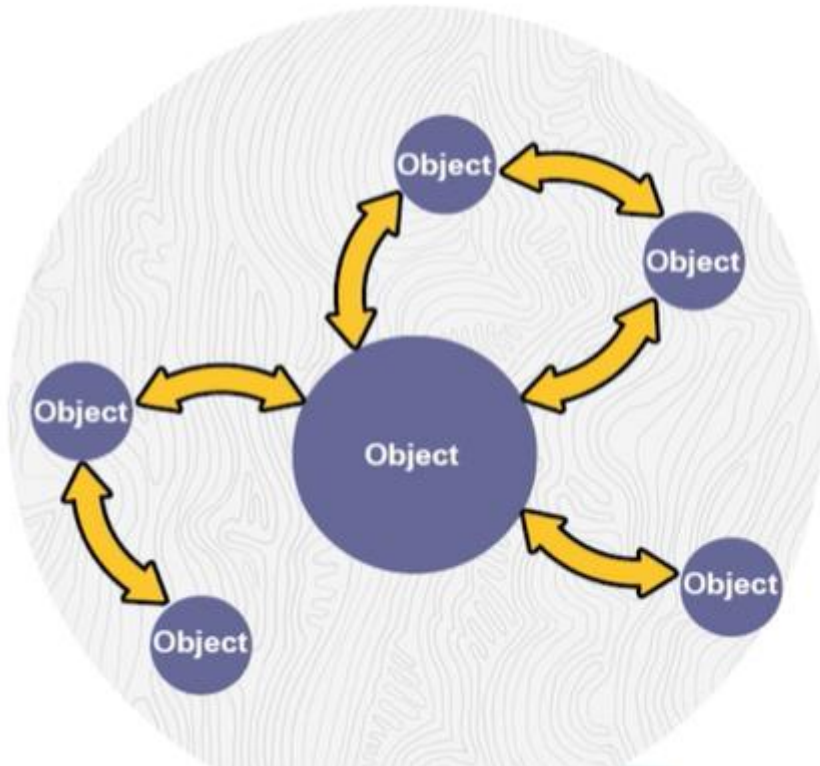
Java Is an Object-Oriented Language

- Interaction of objects
- No prescribed sequence



Java Is an Object-Oriented Language

- Interaction of objects
- No prescribed sequence
- Benefits:
  - Modularity
  - Information hiding
  - Code reuse
  - Maintainability



Classes and Instances

A class:

- Is a blueprint or recipe for an object
- Describes an object's properties and behavior
- Is used to create object instances

Class

- Properties
- Behaviors

Classes and Instances

A class:

- Is a blueprint or recipe for an object
- Describes an object's properties and behavior
- Is used to create object instances

Class
- Properties
- Behaviors

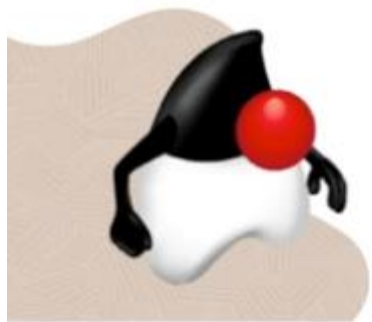
An object:

- Is an instance of a specific class

Objects

Thought Question

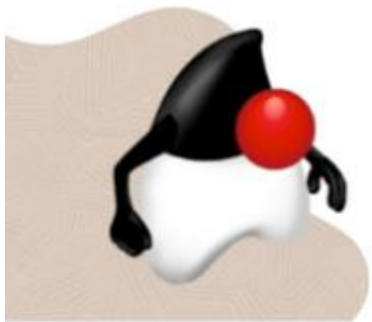
Can you think of appropriate properties and behaviors for the `Customer` class?



Properties	Behaviors

Possible Answer

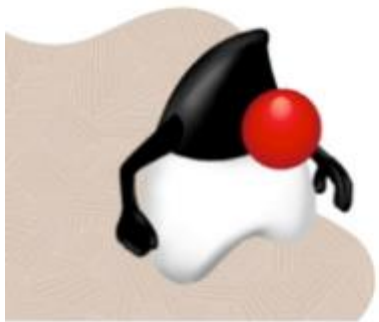
Can you think of appropriate properties and behaviors for the `Customer` class?

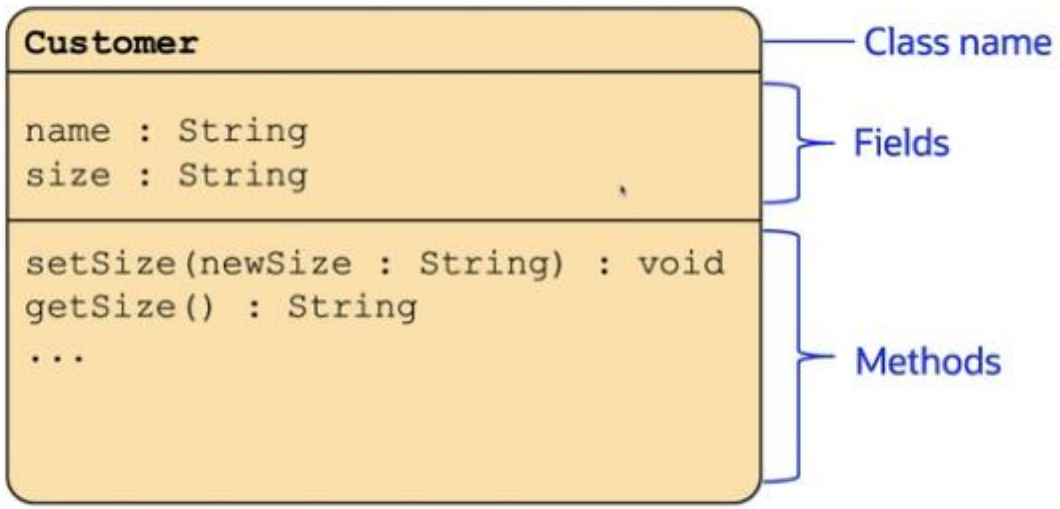


Properties	Behaviors
name	getName
address	setAddress
size	setSize
	inspectWardrobe

Modeling Properties and Behaviors

Classes are modeled using Unified Modelling Language (UML) diagrams.





The actual UML Model would contain descriptions of data types, method parameters, and so on.

Components of a Class

Class declaration

```
class Customer {

    String name;
    String size = "S";

    void setSize(String newSize) {
        size = newSize;
    }
    String getSize() {
        return size;
    }
}
```

Field definitions

Method definitions

Describe Methods

Methods provide a reusable unit of logic, which:

- Enables objects communications
- Distributes work performed by the program
- Implements required object behaviors, such as:
  - Provide access to information
  - Calculate or validate values

```
<access modifier> <return type> <method name> (<parameter list>) {
  /* method body */
  /* return statement */
}
```

Create Methods

Components of a method definition:

Access modifier

Name

```
public void printWishlist () {
  String[] list = {"Golden Hat", "Iron Boots"};
  for (String idea: list) {
    System.out.println(idea);
  }
}

public double getTotal(double price, int quantity) {
  double total = price*quantity*tax;
  return total;
}
```

Method body

Parameters

Return type

return statement

Method Parameters and Return Types

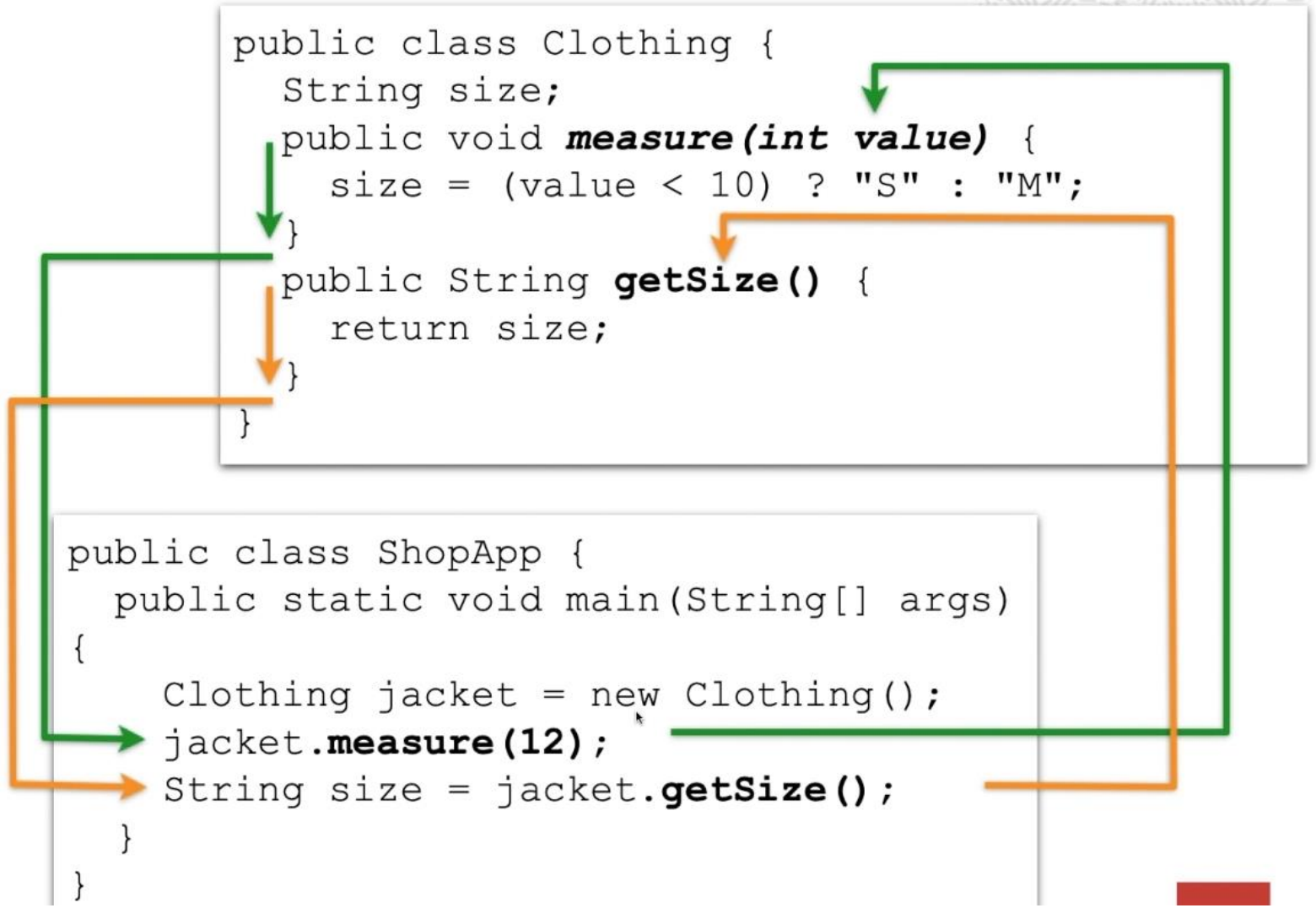
- Any type can be used to describe a method parameter or return a value
- A Method can accept zero or more parameters.
- A Method with void return type does not return a value.
- You must return a value of appropriate type if it is not void.

```
public class Clothing {
  String size;
  public boolean fit(Customer customer) {
   return size.equals(customer.size);
  }
  public String getSize () {
   return size;
  }
  public void setSize (String newSize) {
    size = newSize;
  }
}
```

Invoking Methods

When calling a method invoker:

- Qualifies the method name
- Supplies parameter values
- Waits for the method to complete
- Uses a return value if it is provided



Resolving Method Calls

- Method invocation is resolved based on a method name and types of supplied parameters.
- No two methods with the same signature can exist in a class, even if they have different return types.
- Parameter names are irrelevant.

```
public class ShopApp {
  public static void main(String[] args) {
    Clothing jacket = new Clothing();
    jacket.setSize("M");
    jacket.fit("S"); // which method to
invoke?!
  }
}
```

```
public class Clothing {
  String size;
  public void setSize(String newSize) {
    size = newSize;
  }
  public boolean fit(String sizeToCompare) {
    return size.equals(sizeToCompare);
  }
   public String fit(String otherSize) {
    return (size.equals(otherSize)) ? "Fit" : "Unfit";
  }
}
```

Using Method Overloading

Class can define several methods that:

- Have the same name
- Have different signatures
  - The **number** of parameters
  - The **types** of parameters

Overloading is convenient for invokers.

```
public class ShopApp {
  public static void main(String[] args){
    Clothing jacket = new Clothing();
    jacket.fit(6);
    jacket.fit("S");
    jacket.fit(167,60);
  }
}
```

```
public class Clothing {
  public boolean fit(int size) { ... }
  public boolean fit(String size) { ... }
  public boolean fit(int height, int width) { ... }
}
```

Quiz

Given:

```
myPerson.printValues(100, 147.7, "lavender");
```

Which method declaration correctly corresponds to this method call?

a. `public void printValues(int i, double d)`

b. `public void printValues(i, d, s)`

c. `public void printValues(int i, double d, String s)`

d. `public void printValues(double d, String s, int i)`

Answer

Given:

```
myPerson.printValues(100, 147.7, "lavender");
```

Which method declaration correctly corresponds this method call? (c)

a. `public void printValues(int i, double d)`

b. `public void printValues(i, d, s)`

c. `public void printValues(int i, double d, String s)`

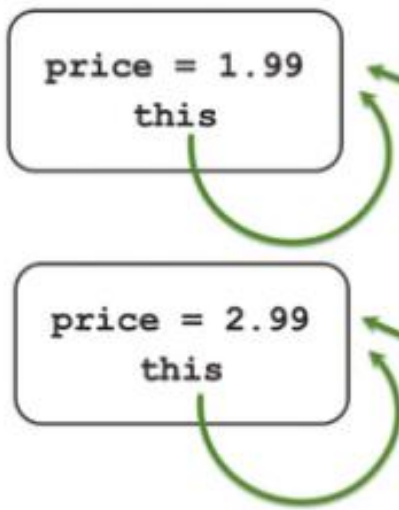
d. `public void printValues(double d, String s, int i)`

Variable Scopes

- Instance variables store information for the overall object.
- The `price` variable is an instance variable and is visible throughout the entire `Clothing` object.
- Local variables are visible only within a given method.
- Method parameters are essentially local variables.
- Variables `fee` and `newPrice` are visible only inside the `setPrice` method.

```
public class Clothing {
    double price;

    public void setPrice (String newPrice) {
        double fee = 10;
        price = newPrice+fee;

        if (price > 20) {
            double discount = 0.5;
            price = price - fee * discount;
        }

    }

    public double getPrice() {
        price = price - fee;
        return price;
    }
}
```

Variable Shadowing




```
public class Clothing {
  double price;
  public void setPrice(String price) {
    this.price = price;
  }
}
```

```
public class Shop {
  public static void main(String[] args) {
    Clothing item1 = new Clothing();
    Clothing item2 = new Clothing();
    item1.setPrice(1.99);
    item2.setPrice(2.99);
  }}
```

What Are Access Modifiers?

- Access modifiers are:
  - Used to restrict access (control visibility scope)
  - Applicable to classes, variables, and methods
  - Not applicable to local variables, because they are not visible outside of a given method anyway

Access Modifier	Visibility Scope
`public`	All classes
`protected`	Classes in the same package and subclasses
default (no access modifier)	Classes in the same package
`private`	Same Class

- O

Apply Access Modifiers to Enforce Encapsulation

```
public class Clothing {
  private double price;
  public final double MIN_PRICE = 1.0;
  public final double TAX = 0.2;
  public void setPrice(double newPrice) {
    price = (newPrice > MIN_PRICE) ? newPrice : MIN_PRICE;
  }
  public double getPrice() {
    return price+price*TAX;
  }
}
```

Hiding variables and allowing access to data through Methods only is called **Encapsulation**.

```
public class Shop {
  public static void main(String[] args) {
    Clothing c = new Clothing();
    c.setPrice(0.99);
    double total = c.getPrice();
    c.price = 0.99;
    c.MIN_PRICE = 2.0;
  }
}
```

Quiz

Given:

```
public class Clothing {
   private double price;                    //line n0

   public double getPrice(){
      return price;                         //line n1
   }
   public void setPrice(double price){
      price = price;                        //line n2
   }
   public double findTax(){
      double price = 10.99;                 //line n3
      double taxRate = 0.05;
      return taxRate * price;               //line n4
   }
}
```

Which line uses the same version of `price` that is declared in `line n0`?

a. `line n1`

b. `line n2`

c. `line n3`

d. `line n4`

Answer

Given:

```
public class Clothing {
    private double price;                    //line n0

    public double getPrice(){
        return price;                        //line n1
    }
    public void setPrice(double price){
        price = price;                       //line n2
    }
    public double findTax(){
        double price = 10.99;                //line n3
        double taxRate = 0.05;
        return taxRate * price;              //line n4
    }
}
```

Which line uses the same version of `price` that is declared in `line n0`? (a)

a. `line n1`

b. `line n2`

c. `line n3`

d. `line n4`

Quiz

Given:

```
public class Store {
   public static void main(String[] args){
      Customer cust = new Customer();
      //line n1
   }
}
```

```
public class Customer {
   private Clothing item;
   …
   public Clothing getItem(){
      return shirt;
   }
}
```

```
public class Clothing {
   private double price;
   …
   public double getPrice(){
      return price;
   }
```

How would you attempt to get the Item's price from `line n1`?

a. `getPrice();`

b. `cust.getPrice();`

c. `cust.item.getPrice();`

d. `cust.getItem().price;`

e. `cust.getItem().getPrice();`

Given:

```
public class Store {
   public static void main(String[] args){
      Customer cust = new Customer();
      //line n1
   }
}
```

```
public class Customer {
   private Clothing item;
   …
   public Clothing getItem(){
      return shirt;
   }
}
```

```
public class Clothing {
   private double price;
   …
   public double getPrice(){
      return price;
   }
```

Answer

How would you attempt to get the item's price from `line n1`? (e)

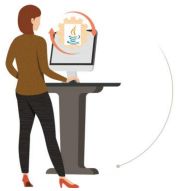
a. `getPrice();`

b. `cust.getPrice();`

c. `cust.item.getPrice();`

d. `cust.getItem().price;`

e. `cust.getItem().getPrice();`



Customer Instances

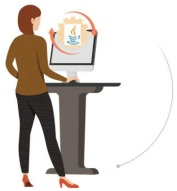
```
public class ShopApp {
  public static void main(String[] args) {

    Customer customer01 = new Customer();
    Customer customer02 = new Customer();

     customer01.name = "Duke";
     customer02.name = "Marvin";

    customer01.getName();
    customer02.producePayment();
  }
}
```

Create new instances (instantiate).

Access fields.

Invoke methods.

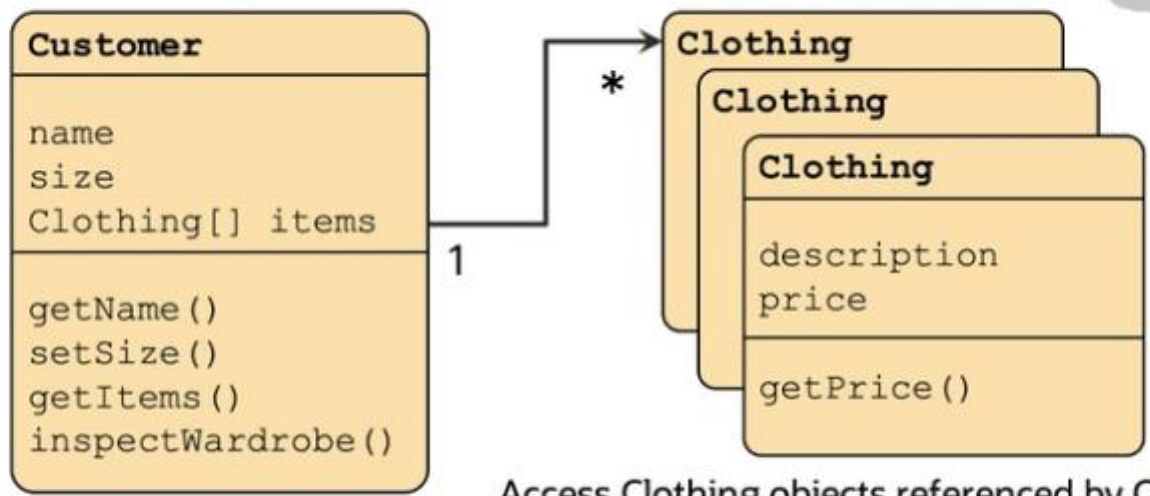
```
class Customer {
// field and method definitions
}
```



Object Instances and Instantiation Syntax

```
public static void main(String[] args){
    // create an instance
    new Customer();

  // declare variable to reference an instance
  Customer customer01;

  // assign instantiate reference to variable
   customer01   = new Customer();

   // do all together.
  Customer customer02 = new Customer();
   }
}
```

Object Referencing Other Objects as a Property



Access Clothing objects referenced by Customer

```
Customer customer01 = new Customer();
for (Clothing item: customer01.getItems())
   {
  item.getPrice();
}
```

```java
package demo;

public class ConstructorDemo {
    int a=10;
    ConstructorDemo(){
        a=100;
        System.out.println(a);
    }
```

Will constructor have return type

Can constructor be overloaded?

```
3  public class ConstructorDemo {
4      int a=10;
5      ConstructorDemo(){
6          a=100;
7          System.out.println(a);
8      }
9      ConstructorDemo(int b){
10         a=100;
11         System.out.println(a+b);
12     }
13     ConstructorDemo(int b,int c){
14         a=100;
15         System.out.println(a+b+c);
16     }
17
```

When constructor will be invoked?

```
3  public class ConstructorDemo {
4      int a=10;
5      ConstructorDemo(){
6          a=100;
7          System.out.println(a);
8      }
9      ConstructorDemo(int b){
0          a=100;
1          System.out.println(a+b);
2      }
3      ConstructorDemo(int b,int c){
4          a=100;
5          System.out.println(a+b+c);
6      }
7
8      public static void main(String[] args) {
9          ConstructorDemo obj= new ConstructorDemo();
0          ConstructorDemo obj1= new ConstructorDemo(20);
1          ConstructorDemo obj2= new ConstructorDemo(20,30);
2
3      }
```

```
3  public class ConstructorDemo {
4      int a=10;
5      ConstructorDemo(){
6          a=100;
7          System.out.println(a);
8      }
9      ConstructorDemo(int b){
10         a=100;
11         System.out.println(
12     }
13     ConstructorDemo(int b,int c){
14         a=100;
15         System.out.println(a+b+c);
16     }
```

Default constructor

Parameterized constructor

Method Overloading

```
1  package demo;
2
3  public class ArithmeticOperations {
4
5
6      void add(int a, int b) {
7          System.out.println("Addition of two numbers is " + (a+b));
8      }
9
10     void add(int a, int b, int c) {
11         System.out.println("Addition of 3 numbers is " + (a+b+c));
12     }
13
14 }
15
```

In Setters and Getters which will return a value?

Super Keyword in Java

sno	This keyword	Super keyword
1	Can invoke current class method	Can invoke parent class method
2	Can invoke current class variable	Can invoke parent class variables
3	JVM never put automatically this() keyword like super() in Java.	By default JVM automatically put the super() keyword at first line inside the constructor.

- Interface doesn’t have method definition(java7), but interface support defualt method definition from java8

```
1  package demo;
2
3  public interface IAddition {
4
5      float PI=3.14F;
6
7      void add();
8
9      default void addition() {
10         PI =4.75F;
11     }
12
13     default void  substraction() {
14
15     }
16
17 }
18
```

Interface can implement multiple interfaces

IAddition.java | Arithematic... | IMultiplica... | Rushwanth.java | Karthik.java | Arvind.java | King.java | Maths.java

```
1  package demo;
2
3  public interface IMyinterface extends IAddition,ISubstraction,IMultiplication {
4
5
6
7  }
8
```

Class can implement multiple interfaces

IAddition.java | Arithematic... | IMultiplica... | Rushwanth.java | Karthik.java | Arvind.java | King.java

```
1  package demo;
2
3  public class Arithematic implements IAddition,ISubstraction,IMultiplication {
4
5      int a=100;
6      int b=200;
7      @Override
```

```java
package demo;

public class King extends Arvind{

    public static void main(String args[]) {
        King obj= new King();
        obj.multiplication(100, 100);

    }

}

```

Class can extend only single class

Interface cannot create an instance

Can interface have default method?

IAddition.java | Arithematic... | Rushwanth.java | Karthi

```
1  package demo;
2
3  public interface IAddition {
4
5      void add();
6
7      default void addition() {
8
9      }
10
11     default void  substraction() {
12
13     }
14
15 }
16
```

Objectives

1. Describe Exceptions
2. Describe the effect an exception has on program flow of control
3. Use a `try/catch` construct to handle exceptions

What Are Exceptions?

Java handles unexpected situations using exceptions.

- Something unexpected happens in the program.
- Java doesn't know what to do. Therefore, it:
  - Creates an exception object containing useful information
  - Throws the exception to the code that invoked the problematic method

Exception Type	Actions that Produce Exception
`ArrayIndexOutOfBoundsException`	Access a nonexistent array index
`NullPointerException`	Use an uninitialized object reference
`ArithmeticException`	Integer divided by zero
And many others types, including custom exceptions that you can define	

The `try/catch` Block

Examine the exception to find out the exact problem, so you can recover cleanly.

You do not need to catch every exception.

A programming mistake should not be handled. It must be fixed.

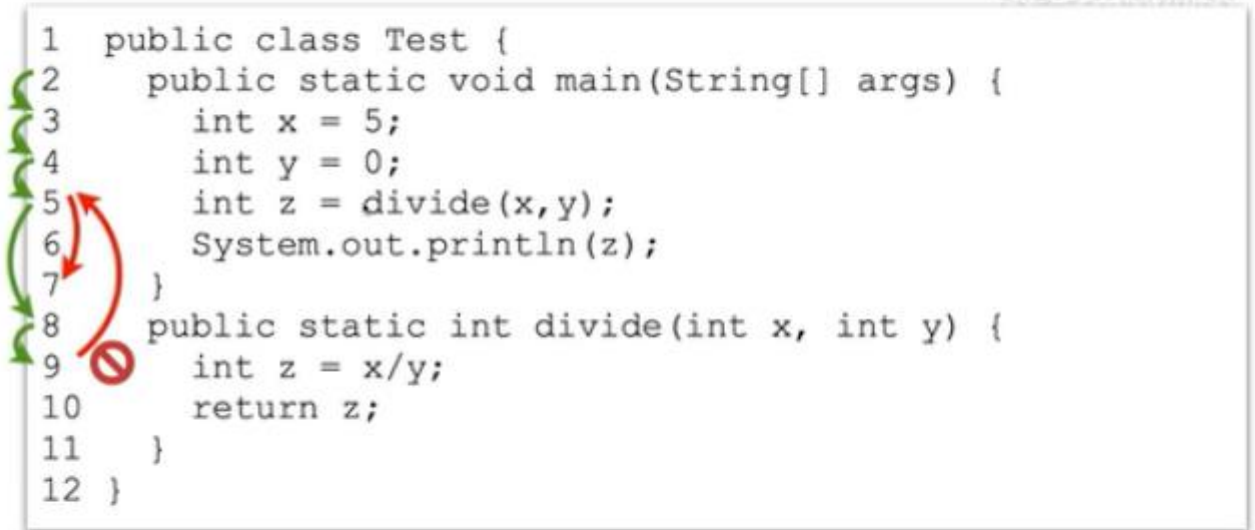
```
try {
  Clothing[] items = new Clothing[10];
  items[0].description = "Pink Boiler Suit";
} catch (NullPointerException e) {
  String errMsg = e.getMessage();
  e.printStackTrace();
} catch (Exception e) { ... }
```

`try` block contains logic that can potentially result in exceptions.

`catch` blocks contain logic that handles exceptions.

Exception Stack Trace

- Exception terminates the normal flow.

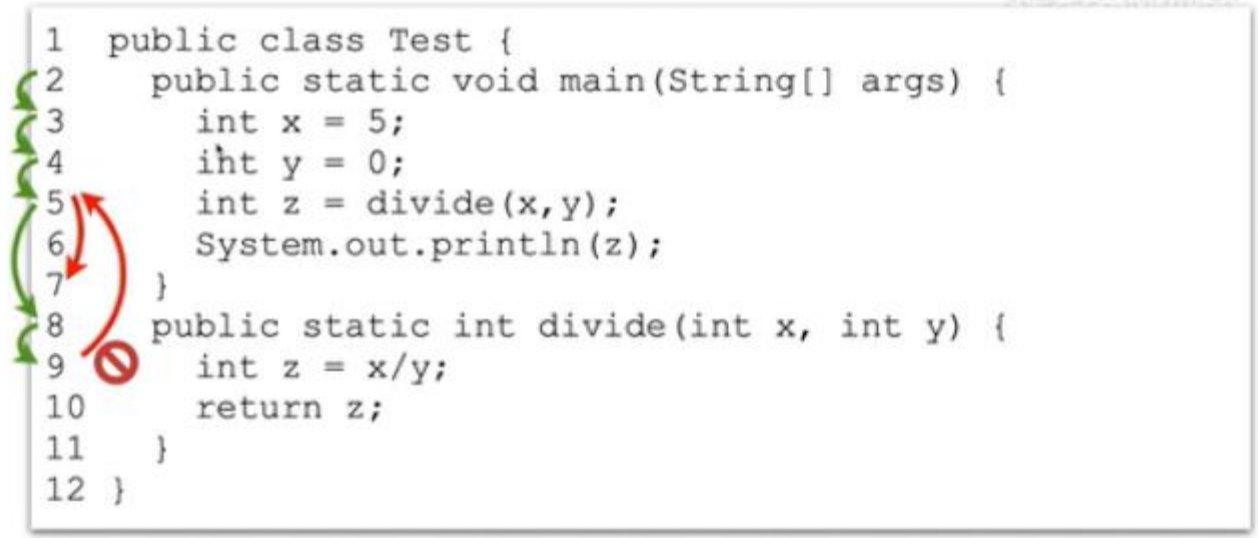


```
1  public class Test {
2    public static void main(String[] args) {
3      int x = 5;
4      int y = 0;
5      int z = divide(x,y);
6      System.out.println(z);
7    }
8    public static int divide(int x, int y) {
9      int z = x/y;
10     return z;
11   }
12 }
```

```
java Test
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.divide(Test.java:9)
        at Test.main(Test.java:5)
```

Exception Stack Trace

- Exception terminates the normal flow.
- Program retraces its path in search of an exception handler.
- If no exception handler is found, the program will print the stack trace and exit.



```
1  public class Test {
2    public static void main(String[] args) {
3      int x = 5;
4      int y = 0;
5      int z = divide(x,y);
6      System.out.println(z);
7    }
8    public static int divide(int x, int y) {
9      int z = x/y;
10     return z;
11   }
12 }
```

```
java Test
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.divide(Test.java:9)
        at Test.main(Test.java:5)
```

Quiz

Given:

```
try {
    int k = 10/0;
}
catch (NullPointerException e) {
    System.out.println("Null");
}
catch(ArithmeticException e){
    System.out.println("Arithmetic");
}
catch (Exception e) {
    System.out.println("Exception");
}
System.out.println("Everything is perfect!");
```

Which statements are printed?
(Choose all that apply.)

a. Null

b. Arithmetic

c. Exception

d. Everything is perfect!

How to connect to DB from java?
Steps to connect to database from java.

Difference between statement vs preparedstatement

What is sql injection how you will avoid it?

PreparedStatement Update

```
public static void main(String[] args) throws SQLException, IOException {

    String database_connection_url = "jdbc:postgresql://127.0.0.1:5432/test";

    String database_user_name = "postgres";

    String database_user_password = "root";

    Connection conn = DriverManager.getConnection(database_connection_url, database_user_name,
            database_user_password);
    System.out.println(conn);

    System.out.println("You are successfully connected to the PostgreSQL database server.");

    String query = "update employee set emp_name=?  where emp_name=?";

    System.out.println(query);
    PreparedStatement pst = conn.prepareStatement(query);

    InputStreamReader r = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(r);
    System.out.println("Enter your name :");
    String myname = br.readLine();

    System.out.println("Enter your updated name :");
    String uname = br.readLine();
    pst.setString(1,uname);
    pst.setString(2, myname);

    int i = pst.executeUpdate();
    System.out.println(i+" rows got updated !");
}
```

Collections Framework in Java

Root interface in collections framework?

The primary advantages of a collections framework are that it:

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

The collections framework consists of:

- **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- **General-purpose implementations**. Primary implementations of the collection interfaces.
- **Legacy implementations**. The collection classes from earlier releases, `Vector` and `Hashtable`, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**. Implementations designed for highly concurrent use.
- **Wrapper implementations**. Add functionality, such as synchronization, to other implementations.
- **Convenience implementations**. High-performance "mini-implementations" of the collection interfaces.
- **Abstract implementations**. Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms**. Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure**. Interfaces that provide essential support for the collection interfaces.
- **Array Utilities**. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

Where does the collection package reside in java

Which classes implements List interface?

Which collection allows duplicate values ?

What is the root class In java?

What design patterns do you know in java?
How many design patterns are there ?

Why serializable interface is used?

How you will avoid getting variable serialized?

What is the difference b/w multithreading and multitasking?

What is the difference b/w final ,finally

What is try with resources?

```java
public class ReadDataExample {

    public static void main(String args[]) throws IOException {


        try(BufferedReader br= new BufferedReader(new InputStreamReader(System.in));) {
            String str =br.readLine();
            System.out.println(str);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }finally {
            System.out.println("checking try with resources ");
        }
    }

}

```

Markers | Properties | Servers | Data Source Explorer | Snippets | Console

<terminated> ReadDataExample [Java Application] C:\Users\matam\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149\jre\bin\javaw.exe (Nov 10, 2021

```
try with resources
try with resources
111
```

Youtube link to java Videos