JAVA BOOTCAMP

INTRODUCTION TO JAVA

TOPICS:

- Review of pre-work
- IntelliJ walkthrough
- Intro to Java

ABOUT JAVA

JAVA

- Introduced in 1995 by James Gosling
- Object-Oriented
- Derives much of its syntax from C and C++, but has fewer low-level facilities than either of them

JAVA TIMELINE

Year	Release/Event
1996	Java Development Kit 1.0 (JDK 1.0)
1997	Java Development Kit 1.1 (JDK 1.1)
1998	Java 2 Platform with version 1.2 of the Software Development Kit (SDK 1.2)
1999	Java 2 Platform, Standard Edition (J2SE) Java 2 Platform, Enterprise Edition (J2EE)
2000	J2SE with version 1.3 of the SDK
2002	J2SE with version 1.4 of the SDK
2004	J2SE 5.0 with version 1.5 of the JDK
2006	Java SE 6 with version 1.6 of the JDK
2010	Oracle buys Sun Microsystems
2011	Java SE 7 with version 1.7 of the JDK
2014	Java SE 8

JAVA EDITIONS

JAVA EDITIONS - JAVA SE

Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

JAVA EDITIONS - JAVA SE

In addition to the core API, the Java SE platform consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits commonly used in Java technology applications.

JAVA EDITIONS - JAVA EE

The Java EE platform is built on top of the Java SE platform.

The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.

JAVA EDITIONS - JAVA ME

The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development. Java ME applications are often clients of Java EE platform services.

JAVA EDITIONS - JAVA FX

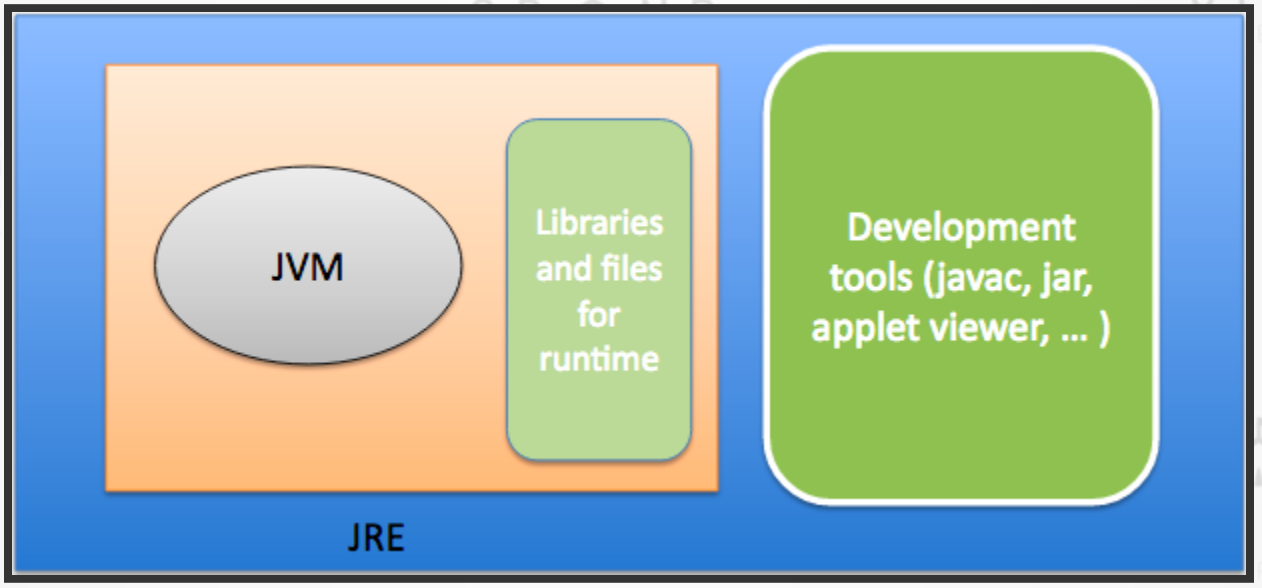
JavaFX is a platform for creating rich internet applications using a lightweight user-interface API. JavaFX applications use hardware-accelerated graphics and media engines to take advantage of higher-performance clients and a modern look-and-feel as well as high-level APIs for connecting to networked data sources. JavaFX applications may be clients of Java EE platform services.

OPERATING SYSTEMS SUPPORTED BY JAVA

- Windows (XP, Vista, 7)
- Linux
- Solaris
- Macintosh OSX

ABOUT JAVA CONT...

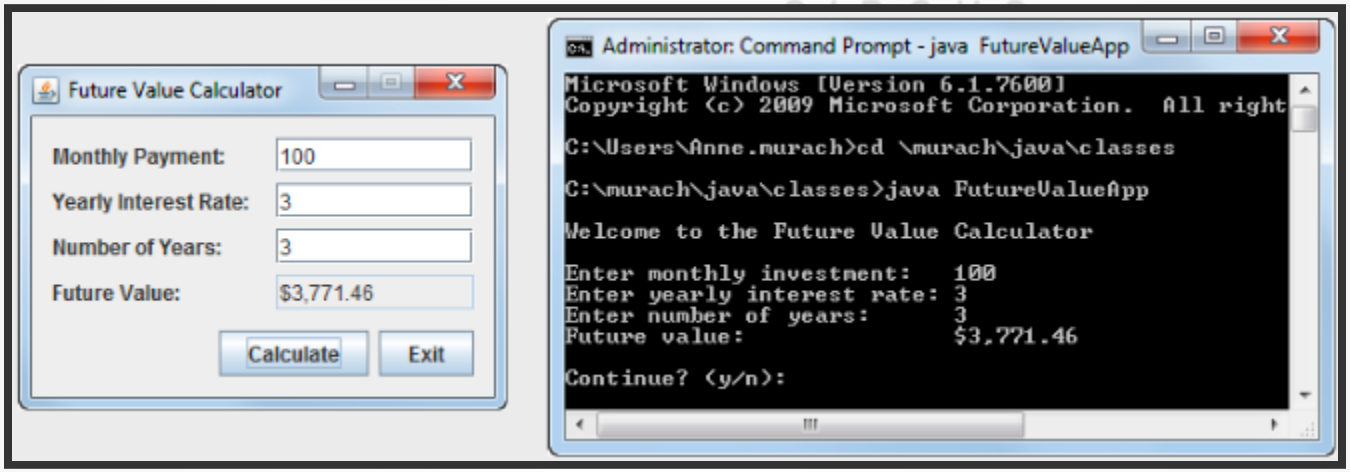
JAVA COMPONENTS



JAVA COMPARED TO C++ AND C

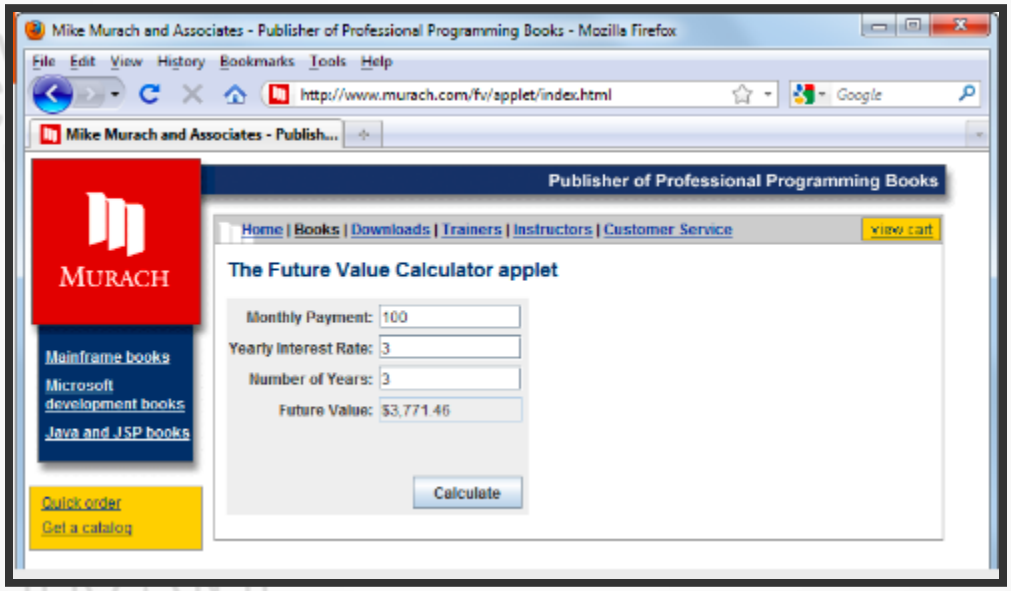
Feature	Description
Syntax	Java syntax is similar to C++ and C# syntax.
Platform	Compiled Java code can be run on any platform that has a Java interpreter. Similarly, compiled C# code (MSIL) can be run on any system that has the appropriate interpreter. Currently, only Windows has an interpreter for MSIL. C++ code must be compiled once for each type of system that it is going to be run on.
Speed	C++ and C# run faster than Java, but Java is getting faster with each new version.
Memory	Both Java and C# handle most memory operations automatically, while C++ programmers must write code that manages memory.

GUI VS CONSOLE APPLICATION

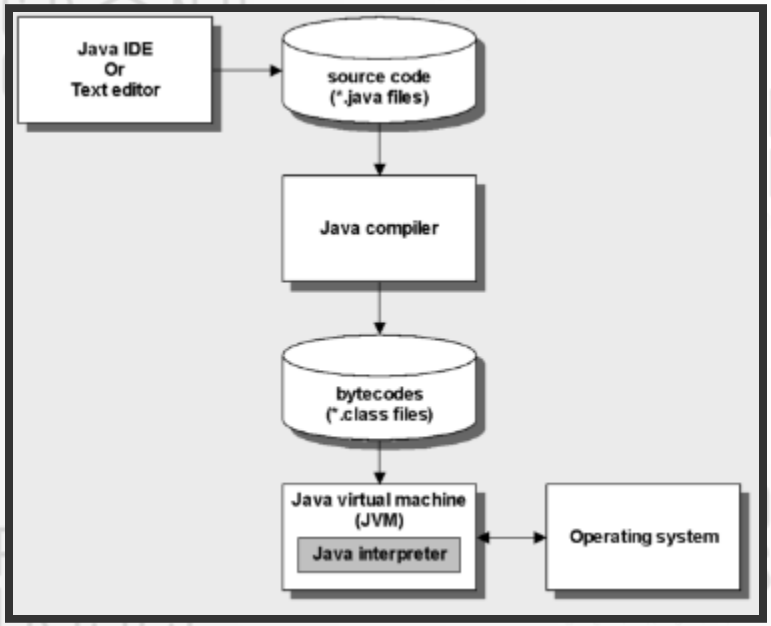


Both are desktop applications that operate directly on the computer.

APPLET VS. SERVLET



HOW JAVA COMPILES AND INTERPRETS CODE



"write once, run anywhere" (WORA)

INTEGRATED DEVELOPMENT ENVIRONMENTS - NET BEANS

This is a free, open-source IDE that runs on most operating systems. It includes features like code completion, a compiler, and a debugger.

INTEGRATED DEVELOPMENT ENVIRONMENTS - WE'RE USING INTELLIJ!

Your instructor will guide you through an introduction to IntelliJ.

INTRO TO JAVA

INTRO TO JAVA

DECLARING THE MAIN METHOD

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

INTRO TO JAVA - STATEMENTS

- In Java, *statements* direct the operation of the program.
- Statements end with a semicolon (;), but blocks of code between curly braces ({}) simply end with the right curly brace.
- It is best practice to use indentation and spacing to align statements and blocks of code. This makes your code much easier to read.

INTRO TO JAVA - COMMENTS

- Comments are used to document what the code does. This is useful not only for other programmers who may need to maintain your code, but future you as well!
- A *single-line comment* begins with two slashes (//).
- A *block comment* starts with a slash and an asterisk (/*) and ends with the same two symbols in reverse (*/).

INTRO TO JAVA - IDENTIFIERS

Any name created in Java is called an identifier. Identifiers may be used to name classes, methods, variables, and so on.

INTRO TO JAVA - KEYWORDS

A keyword is one that is reserved by the Java language. It may not be used as an identifier.

INTRO TO JAVA - NAMING AN IDENTIFIER

- Start each identifier with a letter, underscore, or dollar sign. Use letters, dollar sign, underscores, or digits for subsequent characters.
- Use no more than 255 characters.
- Don't use Java keywords.

PRINTING DATA TO THE CONSOLE

println(data)

Prints the data argument followed by a new line character to the console.

print(data)

Prints the data to the console without starting a new line.

GETTING INPUT FROM THE CONSOLE - THE SCANNER CLASS

The Scanner class allows us to get data input that the user enters into the console. Each entry the user makes is called a token and these can be separated with whitespace.

GETTING INPUT FROM THE CONSOLE - METHODS

next(), *nextInt()*, and *nextDouble()* return the next token stored in the scanner as a String, int, double respectively. *nextLine()* returns any remaining input on the current line as a String and advances the scanner to the next line.

DECLARING AND INITIALIZING VARIABLES

We use variables to store data. For each variable we use, we must declare it (specifying its data type):

type variableName;

In order to initialize a variable, we have to assign it a value:

variableName = value;

DECLARING AND INITIALIZING VARIABLES

One Line

type variableName = value;

Two Lines

type variableName;

variableName = value;

INTRO TO JAVA CONT...

DECLARING AND INITIALIZING VARIABLES

Constants

A *constant* stores a value that cannot change as the program executes. Declare a constant by preceding the normal initialization with the keyword *final* and capitalizing all letters in the name of the variable.

Primitive and reference types

1. Primitive types:
   - Variables have a value
   - Space allocated on stack
2. Reference types:
   - Variable is just a reference allocated on the stack
   - Reference is a “type-safe” pointer
   - Data space allocated on heap

Primitive Types

Type	Bytes	Use
byte	1	Very short integers from -128 to 127
short	2	Short integers from -32,768 to 32,767
int	4	Integers from -2,147,483,648 to 2,147,483,647
long	8	Long integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	Single-precision, floating-point numbers from -3.4E38 to 3.4E38 with up to 7 significant digits
double	8	Double-precision, floating-point numbers from -1.7E308 to 1.7E308 with up to 16 significant digits
char	2	A single Unicode character that's stored in two bytes
boolean	1	A *true* or *false* value

Reference Types

- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.

ARITHMETIC OPERATORS

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+	Positive sign
-	Negative sign

ARITHMETIC EXPRESSIONS - ORDER OF PRECEDENCE

1. Increment and decrement
2. Positive and negative
3. Multiplication, division, and remainder
4. Addition and subtraction

ASSIGNMENT OPERATORS

Operator	Name
=	Assignment
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
%=	Modulus

ASSIGNMENT STATEMENTS - ASSIGNING A VALUE

An *assignment statement* consists of a variable, an equals sign, and an expression. When the assignment statement is executed, the value of the expression is determined and the result is stored in the variable.

CASTING

Implicit

Assigning a less precise data type to a more precise data type will cause Java to automatically convert the less precise data type to the more precise data type. This is also called a *widening conversion*.

CASTING

Explicit

We can code an assignment statement that assigns a more precise data type to a less precise data type. In this case, we must use parentheses to specify the less precise data type. This is also called a *narrowing conversion*.

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT:

- Know what Java is and why Java is important.
- Know Java timeline and editions.
- How Java is compared to other programming languages.
- Types of applications developed using Java.
- Know common Java IDEs
- How to write a Hello World program using Java.
- Add comments to your code
- Keywords in Java

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT:

- Keywords in Java
- How to do input and output at the console
- Data types in Java
- Arithmetic operators and expressions
- Using assignment operators
- Do implicit and explicit casting

CONTROL STATEMENTS AND LOOPS

CONTROL STATEMENTS AND LOOPS

TOPICS

- Control Statements
- Loops

CONTROL STATEMENTS

BOOLEAN EXPRESSIONS - BOOLEAN EXPRESSIONS AND CONTROL STATEMENTS

Recall that a *boolean expression* is one that evaluates to either *true* or *false*. We can use these expressions to determine the flow of our control statements.

BOOLEAN EXPRESSIONS - RELATIONAL OPERATORS

The six *relational operators* (listed in following slide) are used to compare operands which are primitive data types. Operands may be literals, variables, arithmetic expressions, or keywords.

RELATIONAL OPERATORS

Operator	Name
==	Equality
!=	Inequality
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal

BOOLEAN EXPRESSIONS

Examples

```
discountPercent == 2.3  // equal to a numeric literal
letter == 'y'           // equal to a char literal
isValid == false        // equal to the false value

subtotal != 0           // not equal to a numeric literal

years > 0               // greater than a numeric literal
i < months              // less than a variable
```

BOOLEAN EXPRESSIONS

Examples cont..

```
subtotal >= 500          // greater than or equal to a numeric literal

quantity <= reorderPoint  // less than or equal to a variable

isValid                  // isValid is equal to true
!isValid                 // isValid is equal to false
```

Operator	Name	Description
&&	And	Returns a true value if both expressions are true. This operator only evaluates the second expression if necessary.
||	Or	Returns a true value if either expression is true. This operator only evaluates the second expression if necessary.
&	And	Returns a true value if both expressions are true. This operator always evaluates both expressions.
|	Or	Returns a true value if either expressions is true. This operator always evaluates both expressions.
!	Not	Reverses the value of the expression.

SELECTION STATEMENTS

- If statement
- Switch statement

- An **if** statement identifies which statement to run based on the value of a Boolean expression.
- Example:

```
bool condition = true;
if (condition) {
  System.out.println("Variable is set to true.");
} else {
  System.out.println("Variable is set to false.");
}
```

SWITCH STATEMENT

The switch statement is a control statement that selects a switch section to execute from a list of candidates.

SWITCH STATEMENT

```
int caseSwitch = 1;
switch (caseSwitch) {
  case 1:
    System.out.println("Case 1");
    break;
  case 2:
    System.out.println("Case 2");
    break;
  default:
    System.out.println("Default case");
    break;
}
```

LOOPS

LOOPS

- The while loop
- The do loop
- The for loop
- The enhanced loop

WHILE LOOP

- A while loop statement in java programming language repeatedly executes a target statement as long as a given condition is true.
- Syntax:

```
while(Boolean_expression)
{
  //Statements
}
```

DO LOOP

- A do loop is similar to a while loop, except that a do loop is guaranteed to execute at least one time.
- Syntax:

```
do
{
        //Statements
}
while(Boolean_expression);
```

FOR LOOP

- A for loop allows you to write a loop that needs to execute a specific number of times. A for loop is useful when you know how many times a task is to be repeated.
- Syntax:

```
for(initialization; Boolean_expression; update)
{
                //Statements
}
```

THE ENHANCED FOR LOOP

- As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.
- Syntax:

```
for(declaration : expression)
{
        //Statements
}
```

THE ENHANCED FOR LOOP

```
public class Test {

   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

MORE ON CONTROL STATEMENTS

CONTROL STATEMENTS

BREAK STATEMENTS

The *break statement* is used to exit the current loop. In the case of multiple nested loops, a *labeled break statement* may be used to differentiate.

CONTROL STATEMENTS

CONTINUE STATMENTS

The *continue statement* is used to skip any remaining statements in the current loop and jump to the top of the current loop. A *labeled continue statement* may be used to jump to the top of a labeled loop.

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT

- How to evaluate Boolean expressions.
- Know how to use logical and relational operators.
- Use if and switch statements
- Use for, while, do while, and enhanced for loop.
- Comparing numeric variables.
- Comparing string variables.
- How to use continue and break statements.

METHODS

METHODS

TOPICS

- Creating Methods
- Recursion

INTRO TO METHODS

METHODS

DESCRIPTION

- To allow other classes to access a method, use the *public keyword*. To prevent other classes from accessing it, use the *private* keyword.
- Methods that don't return any data should have the *void* keyword as the return type.
- Names of methods should typically start with verbs. Two very commonly used examples of this are methods which either set or return the values of instance variables, called *setters* and *getters* respectively.

METHODS

Syntax

```
public returnType methodName([parameterList])
{
        //the statements of the method
}
```

METHODS

Example

```
public String getCode() {
    String code="Java 101";
    return code;
}
```

METHODS

OVERLOADING

- When you create two or more methods with the same name but with different:
  1. Parameter type
  2. Parameter order
  3. Parameter count
- Example: println method.
- Exercise: Overloaded methods

CALLNG METHODS

DESCRIPTION

- When calling a method with no arguments, we include an empty set of parentheses after the method name.
- If a method returns a value, we can code an assignment statement to assign the return value to a variable. The data type of that variable must be compatible with the data type of the return value.

CALLING METHODS

SYNTAX

```
objectName.methodName(argumentList)
```

EXAMPLES

```
product.printToConsole();
product.setCode(productCode);
double price = product.getPrice();
```

PASSING PARAMETERS

PASSING PARAMETERS TO METHODS

- Pass by value
  - Send the method copies of the variables. Objects are never passed, but a copy of the reference to that object is passed to the method.
  - Strings are like primitive types in the way they are passed.
  - Example on sending objects as parameters.

RECURSION

RECURSION - WHAT IS RECURSION?

- Concept that aims to solve a problem by dividing it into smaller chunks in what is called divide and conquer.
- The idea is that a method can call itself, but with parameters that represent a smaller instance of the problem.

RECURSION - WHAT IS RECURSION?

- You need to specify a stop condition to end the recursion.

RECURSION

EXAMPLE:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Write in Java!

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT:

- What are methods and why are they important
- Syntax for writing methods
- How to do method overloading
- How to call methods
- Passing parameters to methods
- Doing recursion

STRINGS

STRINGS

TOPICS:

- String type
- String functions
- StringBuilder and StringBuffer classes

INTRO TO STRINGS

STRING VARIABLES

- A string consists of letters, numbers, and special characters strung together.
- Use “String” to declare a string variable because it is a class.

STRING VARIABLES

- Strings are immutable!
- This means that once they are created, they cannot be changed.

EXAMPLE

```
String word = "Hello!";
```

STRING VARIABLES

- You can initialize a string with an array of characters.
- Example:

```
char[] helloArray = { 'H', 'e', 'l', 'l', 'o'};
String helloWord = new String(helloArray);
```

JOINING STRINGS

CONCATENATING

A string may be joined with another string by using the plus symbol (+). However, this will convert any other data type to a string.

JOINING STRINGS

APPENDING

Another way is to use the *concat* method of a string variable.

COMPARING STRINGS

METHODS VS. OPERATORS

Because strings are not primitive data types but objects, we must use methods to compare them rather than the relational operators.

COMPARING STRINGS

EQUALS(STRING)

Compares the current String object with the String object specified as the argument and returns a true value if they are equal. This method takes a case-sensitive comparison.

COMPARING STRINGS

EQUALSIGNORECASE(STRING)

Works like the *equals* method, but is not case-sensitive.

COMPARING STRINGS - EXAMPLES

```
firstName.equals("Frank")    // equal to a string literal
firstName.equalsIgnoreCase("Frank”)       // equal to a string literal
firstName.equals("")         // equal to an empty string

!lastName.equals("Jones")    // not equal to a string literal
!code.equalsIgnoreCase(productCode)     // not equal to another string variable

firstName == null            // equal to a null value
firstName != null            // not equal to a null value
```

COMPARING STRINGS

EXAMPLES

```
double discountPercent = 0.0;
String shippingMethod = "";
String customerType="G";
if (customerType.equals("R")) {
    discountPercent = .1;
    shippingMethod = "UPS";
} else if (customerType.equals("C") {
    discountPercent = .2;
    shippingMethod = "Bulk";
} else {
    shippingMethod = "USPS";
}
```

STRING FUNCTIONS

EXAMPLES ON STRING FUNCTIONS

- **char charAt(int index)**:Gets the character at the specified index of the string.
- **boolean endsWith(String suffix)**: Checks if the string ends with the a certain suffix.
- **int indexOf(String str)**: Returns the index of the first occurrence of a certain substring. If the substring is not found, the function returns -1

STRING FUNCTIONS

EXAMPLES ON STRING FUNCTIONS

- **int lastIndexOf(String str)**: Returns the index of the rightmost occurrence of the certain substring.
- **int length()**: Gets the length of a string.
- **String replace(char oldChar, char newChar)**: Returns a copy of the string that has oldChar replaced with newChar.

STRING FUNCTIONS

EXAMPLES ON STRING FUNCTIONS

- **String[] split(String regex)**: Splits the string around matches of the given regular expression, and returns the words as an array of strings.
- **boolean startsWith(String prefix)**: Tests if a string starts with a certain prefix.
- **String substring(int beginIndex)**: Returns a new a substring that starts at a specified index

STRING FUNCTIONS

EXAMPLES ON STRING FUNCTIONS

- **String toLowerCase()**: Returns a string that has all lower case chars.
- **String toUpperCase()**: Returns a string that has all upper case chars.
- **String trim()**: Omits leading and trailing whitespaces.

STRINGBUFFER

STRINGBUFFER IS MUTABLE

- Strings can leave many unused objects in the memory when you do a lot of operations on them, as a new copy is made after each operation.
- It is better to use StringBuffer when you do a lot of string operations.
- Unlike Strings, objects of type StringBuffer are mutable, so they can be modified.

EXAMPLE

STRINGBUFFER

```
StringBuffer strBuff = new StringBuffer("String Buffer test!");
strBuff.append("\t Super!");
System.out.println(strBuff);
```

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT:

- What are strings
- How to define and initialize strings
- Joining strings
- Comparing strings
- String functions
- Mutable strings (StringBuffer)

EXCEPTIONS AND ERROR HANDLING

EXCEPTIONS AND ERROR HANDLING

TOPICS

- Exception handling
- Data validation
- Basic testing & debugging

EXCEPTIONS

HANDLING EXCEPTIONS

HOW EXCEPTIONS WORK

- An exception is *thrown* when the application can't perform an operation. An exception is an object created from one of several different Exception classes.

HANDLING EXCEPTIONS

HOW EXCEPTIONS WORK

- An application is said to *catch* any thrown exceptions and handle them, which may involve simply informing the user they need to provide valid input or more complicated action.

CATCHING EXCEPTIONS DESCRIPTION

In order to catch an exception, we use blocks of statements, called the *try statement* (which contains code that may throw an exception) and *exception handler* (which details the appropriate response to an exception).

CATCHING EXCEPTIONS

SYNTAX

```
try { statements }
catch (ExceptionClass exceptionName) {statements}
```

HANDLING EXCPTIONS

EXAMPLE

```
double subtotal = 0.0;
try
{
    System.out.print("Enter subtotal:    ");
    subtotal = sc.nextDouble();
}
catch (InputMismatchException e)
{
    sc.next();  // discard the incorrectly entered double
    System.out.println("Error! Invalid number. Try again.\n");
}
```

RUNTIME ERRORS

CHECKED EXCEPTIONS

Checked exceptions are checked by the compiler, so we must write code to handle them before we can compile our code.

RUNTIME ERRORS

UNCHECKED EXCEPTIONS

Unchecked exceptions do not get checked by the compiler, but can occur at runtime. If one occurs that is not handled by our code, the program will terminate.

RUNTIME ERRORS

TRY BLOCK

We can code a *try block* around any statements which might throw an exception.

```
try { statements }
```

```
try {
        in = new RandomAccessFile(path, "r");
                // may throw FileNotFound
        String line = in.readLine();
                // may throw IOException
        return line;
    }
```

RUNTIME ERRORS

MULTI-CATCH BLOCK

The *multi-catch* block allows us to use a single catch block for multiple exceptions at the same level in the inheritance hierarchy.

```
catch(ExceptionType | ExceptionType [| ExceptionType]… e)  { statements }

catch(FileNotFoundException | EOFException e) {
        System.err.println(e.toString());
        return null;
```

RUNTIME EXCEPTIONS

THROW STATEMENTS

We can use a throw statement to throw any object created from a subclass of the Throwable class.

```
throw throwableObject;

throw new IllegalArgumentException("Interest rate must be > 0.");
```

VALIDATING DATA

SCANNER CLASS

Method	Description
hasNext()	Returns true if the scanner contains another token
hasNextInt()	Returns true if the scanner contains another token that can be converted to an int value
hasNextDouble()	Returns true if the scanner contains another token that can be converted to a double value
nextLine()	Returns any remaining input on the current line as a String object and advances the scanner to the next line

VALIDATING DATA

EXAMPLE

```
Enter subtotal:   $100
Error! Invalid number. Try again.

Enter subtotal:
```

PREVENTION AND DEBUGGING

VALIDATING DATA

PREVENT EXCEPTIONS

We will generally wish to prevent exceptions being thrown due to invalid data. We can use *data validation* to display an error message when the user inputs invalid content until all entries are valid.

VALIDATING DATA

NUMERIC ENTRIES

- Make sure the entry has a valid numeric format
- Make sure that the entry is within a valid range. This is known as *range checking*.

TESTING AND DEBUGGING

TESTING

To test an application is to run it to make sure it works correctly, trying every possible combination of input data and user actions to be certain it work in every case.

TESTING AND DEBUGGING

DEBUGGING

Debugging an application means actually fixing the errors (or bugs) identified by testing the it. The app must then be tested again to make sure the fix hasn't caused more errors.

TESTING

PHASES

1. Check the user interface to make sure that it works correctly.
2. Test the application with valid input data to make sure the results are correct.
3. Test the application with invalid data or unexpected user actions. Try everything you can think of to make the application fail.

TESTING

TYPES OF ERRORS

- *Syntax errors* violate the rules for how Java statements must be written.
- *Runtime errors* don't violate syntax rules, but cause exceptions to be thrown which stop the execution of the application.
- *Logic errors* don't cause syntax or runtime errors, but produce incorrect results.

SYNTAX ERRORS

COMMON SYNTAX ERRORS

- Misspelling keywords
- Forgetting to declare a data type for a variable
- Forgetting an opening or closing parenthesis, bracket, or comment character
- Forgetting to code a semicolon at the end of a statement
- Forgetting an opening or closing quotation mark

LOGIC ERRORS

PROBLEM WITH VALUES

- Not checking that a value is the right data type before processing it
- Using one equals sign instead of two when testing numeric and Boolean values for equality
- Using two equals signs instead of the *equals* or *equalsIgnoreCase* methods to test two strings for equality

LOGIC ERRORS

FLOATING-POINT ARITHMENTIC

- The double data type uses floating-point numbers that can lead to arithmetic errors.
  - Ex. 0.2 + 0.7 = 0.8999999999999999
- One way around this is to use the BigDecimal class.

LOGIC ERRORS

TRACING CODE EXECUTION

- We can *trace* the execution of our apps by adding statements to our code that display messages or variable values at key points.
- Messages printed to the console can indicate what code is being executed or display the current value of variables.

LOGIC ERRORS

TRACING CODE EXECUTION

- When an incorrect value is displayed, there is a good chance the app contains a logic error between the current println statement and the previous one.

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT

- How exceptions occur.
- How to handle exceptions and runtime errors.
- How to validate input.
- How to test and debug your code.
- How to fix syntax errors.
- How to detect logical errors.

DOCUMENTATION

DOCUMENTATION

TOPICS

- Importance of Documentation
- Using javadoc to document a package

JAVADOC

DOCUMENT A PACKAGE WITH JAVADOC

JAVADOC

The JDK includes a utility named *javadoc* that makes it easy to generate HTML-based documentation for your classes.

DOCUMENT A PACKAGE WITH JAVADOC

HOW TO ADD JAVADOC COMMENTS TO A CLASS

A *javadoc comment* begins with /* and ends with */. With a *javadoc comment*, any additional asterisks are ignored. Because of that, asterisks are commonly used to set the comments off from the rest of the code.

DOCUMENT A PACKAGE WITH JAVADOC

DESCRIPTION

- You can use *javadoc comments* to describe a class and its public and protected fields, constructors, and methods.
- A comment should be placed immediately above the class or member it describes. For a class that means that the comment must be placed after any import statements.

DOCUMENT A PACKAGE WITH JAVADOC

HOW TO USE HTML AND JAVADOC TAGS IN JAVADOC COMMENTS

To help format the information that's displayed in the documentation for a class, you can include HTML and javadoc comments.

The HTML tag you're most likely to use is the *CODE tag*. This tag can be used to display text in a monospaced font.

DOCUMENT A PACKAGE WITH JAVADOC

COMMON HTML TAG USED TO FORMAT JAVADOC COMMENTS

HTML tag	Description
< code> </code>	Displays the text between these tags with a monospaced font.

DOCUMENT A PACKAGE WITH JAVADOC

COMMON JAVADOC TAGS

Javadoc tag	Description
@author	Identifies the author of the class. Not displayed by default.
@version	Describes the current version of the class. Not displayed by default.
@param	Describes a parameter of a constructor or method.
@return	Describes the value that's returned by a method.

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT

- Importance of documenting your code
- How to use Javadoc to document your code

CODE BEST PRACTICES

CODE BEST PRACTICES

TOPICS

- Coding best practices
- Code refactoring

BEST PRACTICES

BEST PRACTICES

CLEAN CODE

A lot of time is wasted due to poorly written and organized code. If we take the time to clean up our code from the beginning, we can avoid negative consequences later on.

BEST PRACTICES

STRATEGIES FOR WRITING CLEAN CODE

- Give things meaningful names
- Write comments only when necessary (make the code so clear it speaks for itself)
- Create small functions
- Follow a specific formatting structure

BEST PRACTICES

CODE REVIEW

One way to evaluate the quality of a feature we've coded is to submit it for *code review*. This is a systematic examination of the software.

BEST PRACTICES

TYPES OF CODE REVIEW

- Over-the-shoulder
- Email pass-around
- Pair programming
- Tool-assisted

BEST PRACTICES

REFACTORING CODE

When we *refactor* our code, we go back and restructure it without changing its behavior. As far as the user's interaction with the application is concerned, nothing changes.

RECAP

RECAP - WHAT YOU SHOULD KNOW AT THIS POINT

- How a clean code looks like.
- How to write clean code.
- How to do code reviews.
- The types of code review.
- How to refactor code.

ARRAYS

ARRAYS

TOPICS

- Arrays
- Multi-Dimension Arrays

THE ARRAY OBJECT

THE ARRAY OBJECT

WHAT IS AN ARRAY?

An *array* is an object that contains one or more items called *elements*. All elements of an array must be of the same type.

THE ARRAY OBJECT

ARRAY SIZE

Arrays in Java have a fixed *length* or size that indicates the number of elements that it contains. You can code the number of elements as a literal, a constant, or a variable of type int.

CREATING ARRAYS

CREATING AN ARRAY

To create an *array* you must declare a variable of the correct type and instantiate an array object that the variable refers to.

CREATING ARRAYS

ASSIGNING VALUES

The elements of an array are referred to by their *index*. Indexing begins with 0, so an *index* of 0 refers to the first element; an *index* of 3 refers to the fourth element.

CREATING ARRAYS

INITIAL VALUES

Depending on the data type for the array, each element will be given an initial value.

- *0* for numeric arrays
- *false* for booleans
- *'\0'* for characters
- *null* for objects

DECLARING AND INSTANTIATING ARRAYS

ONE LINE

```
type[ ] arrayName = new type[length];
```

DECLARING AND INSTANTIATING ARRAYS

TWO LINES

```
type[ ] arrayName; OR type arrayName[ ];
arrayName = new type[length];
```

DECLARING AND INSTANTIATING ARRAYS

ONE LINE

```
String[ ] titles = new String[3];
```

TWO LINES

```
double[ ] prices;
prices = new double[4];
```

DECLARING AND INSTANTIATING ARRAYS

AN ARRAY OF PRODUCT OBJECTS

```
String[ ] titles = new String[3];
```

CODE THAT USES CONSTANT

```
final int TITLE_COUNT = 100;
String[ ] titles = new String[TITLE_COUNT];
```

DECLARING AND INSTANTIATING ARRAYS

CODE THAT USES A VARIABLE

```
Scanner sc = new Scanner(System.in);
int titleCount = sc.nextInt();
String[ ] titles = new String[titleCount];
```

ARRAY ELEMENTS

ONE LINE

```
arrayName[index]
```

REFER TO THE 3RD ELEMENT

```
myArray[2]
```

ASSIGNING VALUES TO ARRAYS

ONE LINE

```
type[ ] arrayName = {value1, value2, value3, …};
```

TWO+ LINES

```
type[ ] arrayName;      OR    type arrayName[ ];
arrayName = new type[length];
arrayName[0] = value1;
arrayName[1] = value2;
arrayName[2] = value3;
…
```

FOR LOOPS WITH ARRAYS

FOR LOOPS WITH ARRAYS

FOR LOOPS

For loops are commonly used to process the elements in an array one at a time by incrementing an index variable. You can use the length field of an array to determined how many elements are defined for the array.

FOR LOOPS WITH ARRAYS

SYNTAX

```
for (int i = 0; i < arrayName.length; i++)
{
        statements
}
```

FOR LOOPS AND ARRAYS

CODE THAT PRINTS AN ARRAY OF PRICES TO THE CONSOLE

```
double[ ] prices = {14.95, 12.95, 11.95, 9.95};
for (int i = 0; i < prices.length; i++)
{
        System.out.println(prices[i]);
}
```

FOR LOOPS AND ARRAYS

CODE THAT COMPUTES THE AVERAGE OF THE ARRAY OF PRICES

```
double sum = 0.0;
for (int i = 0; i < prices.length; i++)
{
        sum += prices[i];
}
double average = sum / prices.length;
```

ENHANCED FOR LOOPS

An **enhanced for loop** is sometimes called a **foreach loop** because it lets you process each element of an array. The **enhanced for loop** simplifies the code required to loop through arrays.

```
for (type variableName : arrayName)
  {
        statements
  }
```

FOR LOOPS AND ARRAYS

CODE THAT PRINTS AN ARRAY OF PRICES TO THE CONSOLE

```
double[ ] prices = {14.95, 12.95, 11.95, 9.95};
for (double price : prices)
{
        System.out.println(price);
}
```

FOR LOOPS AND ARRYAS

CODE THAT COMPUTES THE AVERAGE OF THE ARRAY OF PRICES

```
double sum = 0.0;
for (double price : prices)
{
        sum += price;
}
double average = sum / prices.length;
```

THE ARRAYS CLASS

ARRAYS CLASS

The java.util package contains the *arrays class*. The *Arrays class* contains several static methods that you can use to compare, sort, and search arrays.

METHODS OF THE ARRAY CLASS

STATIC METHODS OF THE ARRAYS CLASS

Method	Description
fill(arrayName, value)	Fills all elements of the specified array with the specified value.
fill(arrayName, index1, index2, value)	Fills elements of the specified array with the specified value from the index1 element to, but not including the index2 element.

METHODS OF THE ARRAY CLASS

STATIC METHODS OF THE ARRAYS CLASS

Method	Description
equals(arrayName1, arrayName2)	Returns a boolean true value if both arrays are of the same type and all of the elements within the arrays are equal to each other.
copyOf(arrayName, length)	Copies the specified array, truncating or padding with default values as necessary so the copy has the specified length.

METHODS OF THE ARRAYS CLASS

METHOD FEATURES

- All accept arrays of primitive data types and arrays of objects for the arrayName argument
- All index arguments must be int types

REFERENCE AND COPY ARRAYS

REFERENCE AND COPY ARRAYS

REFERENCE AN ARRAY

You can create a *reference* to an array by assigning an array variable to an existing array. Both variables will refer to the same array, thus changes to one will be reflected in the other.

REFERENCE AND COPY ARRAYS

COPY AN ARRAY

The easiest way to copy an array is to use the static copyOf or copyOfRange methods. When you copy an array the new array must be of the same type as the source array.

HOW TO CREATE AN ARRAY REFERENCE

CODE THAT CREATES A REFERENCE TO AN ARRAY

```
double[ ] grades = {92.3, 88.0, 95.2, 90.5};
double[ ] percentages = grades;
percentages[1] = 70.2;
System.out.println("grades[1]=" + grades[1]);
```

HOW TO CREATE AN ARRAY REFERENCE

HOW TO CREATE AN ARRAY REFERENCE

```
double[ ] grades = new double[5];
grades = new double[20];
```

HOW TO COPY AN ARRAY

CODE THAT COPIES THE VALUES OF AN ARRAY

```
double[ ] grades = {92.3, 88.0, 95.2, 90.5};
double[ ] percentages = Arrays.copyOf(grades, grades.length);
percentages[1] = 70.2;
System.out.println("grades[1]=" + grades[1]);
```

HOW TO COPY AN ARRAY

CODE THAT COPIES PART OF ONE ARRAY INTO ANOTHER ARRAY

```
double[ ] grades = {92.3, 88.0, 95.2, 90.5};
Arrays.sort(grades);
double[ ] lowestGrades = Arrays.copyOfRange(grades, 0, 2);
double[ ] highestGrades = Arrays.copyOfRange(grades, 2, 4);
```

TWO-DIMENSIONAL ARRAYS

TWO-DIMENSIONAL ARRAYS

ARRAY OF ARRAYS

Two-dimensional arrays use two indexes to store data. Java implements a *two-dimensional array* as an *array of arrays* where each element in the array is at the intersection of a row and column.

TWO-DIMENSIONAL ARRAYS

RECTANGULAR ARRAYS

The simplest type of two-dimensional array is a *rectangular array*, in which each row has the same number of columns.

RECTANGULAR ARRAYS

SYNTAX FOR CREATING A RECTANGULAR ARRAY

```
type[ ] [ ] arrayName = new type[rowCount] [columnCount];
```

RECTANGULAR ARRAYS

SYNTAX FOR REFERRING TO AN ELEMENT OF A RECTANGULAR ARRAY

```
arrayName[rowIndex] [columnIndex]
```

RECTANGULAR ARRAYS

A STATEMENT THAT CREATES A 3X2 ARRAY

```
int[ ] [ ] numbers = new int[3] [2];
```

RECTANGULAR ARRAYS

SYNTAX FOR REFERRING TO AN ELEMENT OF A RECTANGULAR ARRAY

```
arrayName[rowIndex] [columnIndex]
```

JAGGED ARRAYS

A **jagged array** is a two-dimensional array in which the rows contain unequal numbers of columns. When you instantiate a **jagged array**, you specify the number rows, but not the number of columns.

SYNTAX

```
type[ ] [ ] arrayName = new type[rowCount] [ ];
```

JAGGED ARRAYS

CODE THAT CREATES A JAGGED ARRAY OF INTEGERS

```
int [ ] [ ] numbers = new int[3] [ ];
numbers[0] = new int[10];
numbers[1] = new int[15];
numbers[2] = new int[20];
```

JAGGED ARRAYS

CODE THAT CREATES AND INITIATES A JAGGED ARRAY OF STRINGS

```
String [ ] [ ] titles = {{“War and Peace”, “Wuthering Heights”, “1984”,},
                         {“Casablanca”, “Wizard of Oz”, “Star Wars”, “Bir
                         {“Blue Suede Shoes”, “Yellow Submarine”}};
```

RECAP

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT

- What is an array.
- How to declare and instantiate an array.
- How to access and use array elements.
- How to use loops with arrays.
- How to use the Array class.
- How to copy arrays.
- How to create and use multi-dimensional arrays.

COLLECTIONS AND GENERICS

COLLECTIONS

TOPICS

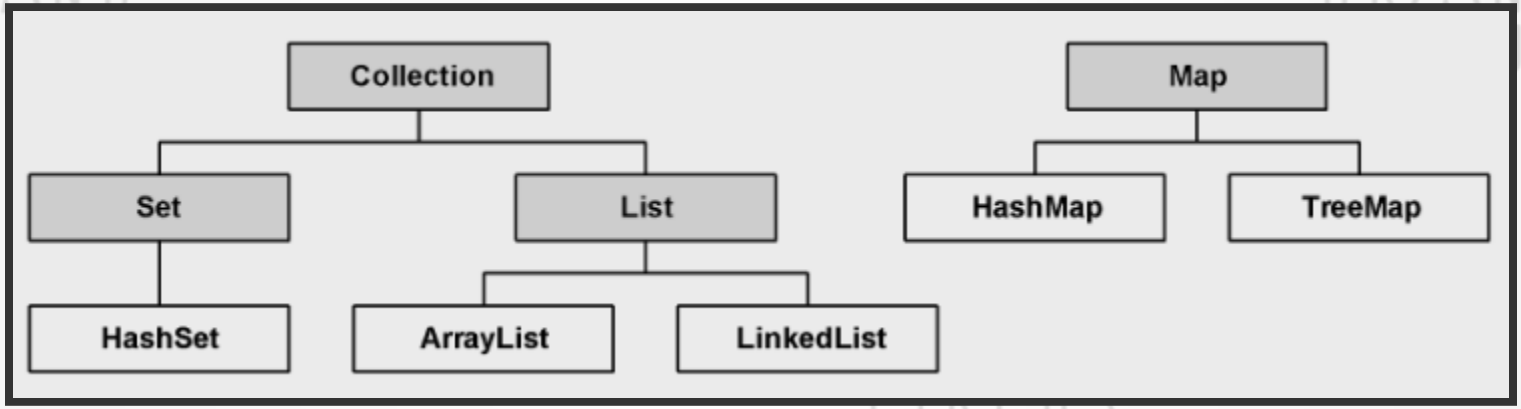
- Java collections
- ArrayList and LinkedList classes
- Maps
- Stacks and Queues

COLLECTIONS IN JAVA

WHAT ARE COLLECTIONS?

- Similar to an array, a *collection* is used to hold other objects.
- However, unlike arrays, collection aren't part of the language itself.
- Rather, they're classes provided with the Java API. They are also more flexible and efficient than arrays.

JAVA COLLECTIONS



JAVA COLLECTIONS

GENERICS

- Generics allows us to create typed collections, which can hold objects of any type.
- To declare a variable that refers to a typed collection, we need to list the type in angle brackets (<>) following the name of the collection class.

JAVA COLLECTIONS

ARRAY LISTS

- An array list is a collection that's similar to an array, but can change its capacity as elements are added or removed.
- We can specify the type of elements to be stored in the array list by naming a type in angle brackets.
- We can specify the initial size of an array list when we create it or. The default size for the array list is 10.

JAVA COLLECTIONS

ARRAY LISTS

- You cannot define an array list with generics using primitive types.
- Example of the wrong way:

```
ArrayList<int> numbers = new ArrayList<int>();
```

JAVA COLLECTIONS

ARRAY LISTS

- Solution: Use the Integer object.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

- Beginning with Java 7, you can omit the type from the angle brackets before the constructor call.

JAVA COLLECTIONS

LINKED LISTS

- Similar to the ArrayList class, the LinkedList class gives us a few more features.
- Rather than using an array to store its elements, a LinkedList uses pointers to refer to adjacent objects.

JAVA COLLECTIONS

METHODS TO ACCESS THE ELEMENTS

- add(object): Adds an object to the end of the list.
- add(index, object): Adds an object at a specific location.
- get(index): Returns the object at the specified index.
- size(): Returns the number of elements in the list.

JAVA COLLECTIONS

MAPS

A map is a specific type of collection that deals with values that are associated with keys.

JAVA COLLECTIONS

MAPS

Examples on using maps

```
Map m1 = new HashMap();
      m1.put("John", "8");
      m1.put("Adam", "31");
      m1.put("Ian", "12");
      m1.put("Daisy", "14");
      System.out.println(m1.get("Zara"));
```

JAVA COLLECTIONS

STACKS AND QUEUES

- A stack is a LIFO (Last in First Out) data structure.
- A Queue is a FIFO (First in First Out) data structure.
- Examples

RECAP

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT

- Know what collections are.
- Know classes and packages that define collections.
- Generics and their role in collections
- Define and use Array lists and Linked lists.
- The difference between an Array list and a Linked list, and when to use them.
- Define and use Maps. Know when to use Maps.
- Definition of stacks and queues. When to use stacks and queues.