COE318 Software Design Java:

Lecture 1:

What is Java:

Java programming is a high-level, object-oriented programming language known for its simplicity, portability, and versatility. It provides a stable and secure platform for developing a wide range of applications, from desktop software to mobile apps and enterprise systems. Java's key features include its platform independence, strong standard library, automatic memory management, and support for multi-threading, making it a popular choice among developers worldwide.
Java Course: https://youtu.be/xk4_1vDrzzo
Extra practice: https://www.w3schools.com/java/default.asp

Key features of Java:

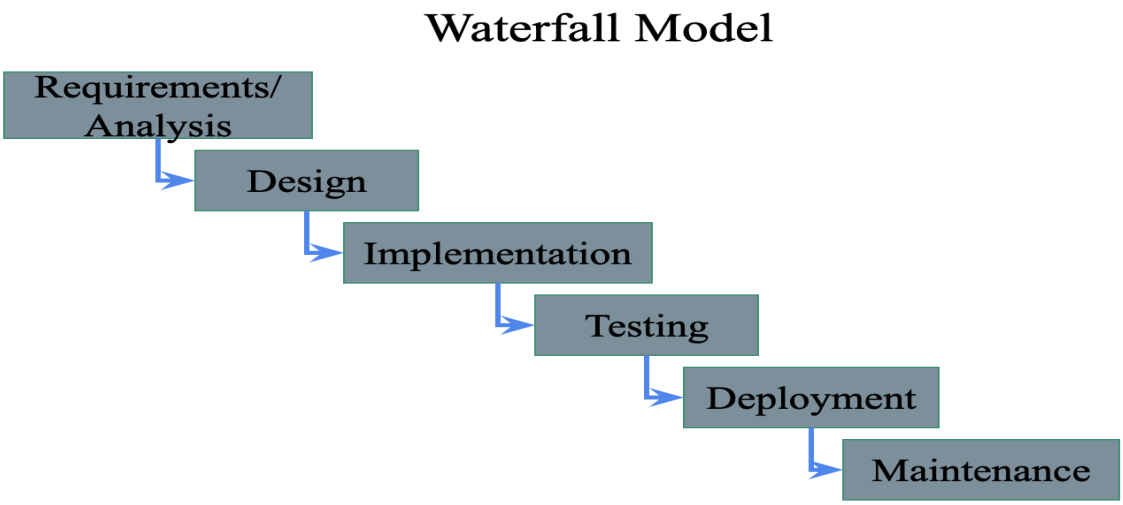
- **Simple:**
  - Easy-to-learn offers straightforward syntax and a large library of pre-built functions
- **Object-oriented:**
  - Supports modular and reusable code through classes and objects.
- **Platform independence:**
  - Programs can run on any platform with a compatible JVM.
- **Automatic memory management:**
  - Garbage collection prevents memory leaks and overflows.
- **Strong standard library:**
  - Provides pre-built classes and APIs for various tasks.
- **Security:**
  - Built-in features for protection against vulnerabilities and secure communication.
- **Multi-threading:**
  - Supports concurrent programming for improved performance and responsiveness.

SDLC Models:

(Software Development Life Cycle) models provide a structured approach to software development. These models define a series of phases and activities that guide the development process from start to finish.

- **Waterfall model:**

    - The linear approach, sequential phases, emphasizes planning and documentation.
- **Iterative model:**
    - Small iterations, repeated cycles of development, testing, and feedback.
- **Agile model:**
    - Adaptability, collaboration, customer involvement, frequent iterations.
- **V Model:**
    - Corresponding testing phases, early and continuous testing, verification, and validation focus.
- **RAD model:**
    - Rapid prototyping, iterative development, user involvement, quick feedback.
- **Spiral model:**
    - Iterative cycles, risk analysis and mitigation, flexibility, and risk management.

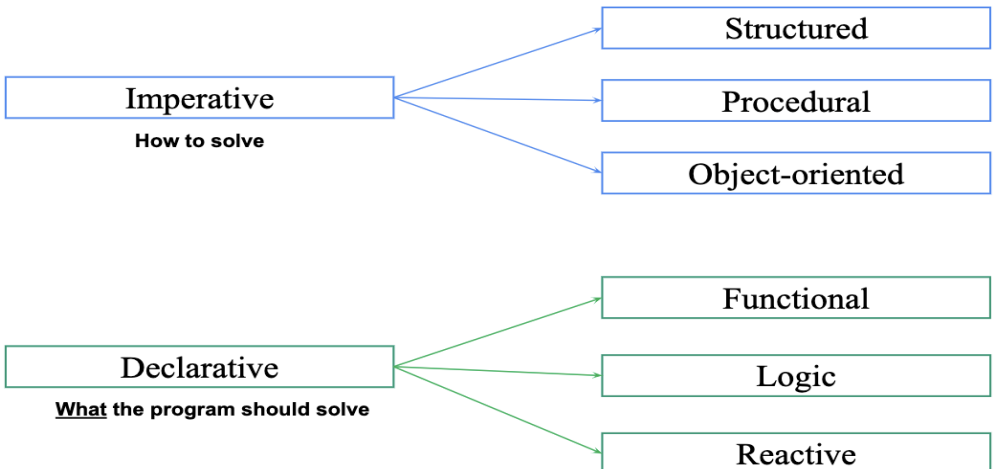


https://www.youtube.com/watch?v=Fi3_BjVzpqk

➔ Helps explain SDLC

Language Paradigms:

➔ **Java** is Imperative Language, a sequence of statements to specify the steps for the computer to execute and achieve a desired result. It provides control structures like loops and conditional statements to control the flow of execution. While it has some declarative features, its core emphasizes explicit control flow and step-by-step instructions.



- ➔ Languages like HTML and CSS are Declarative
- ➔ Declarative programming focus on describing the desired outcome or logic of a program while leaving the implementation details to the language or framework being used.

Imperative programming example (using Java):

```
List<Employee> filteredEmployees = new ArrayList<>();
for (Employee employee : employees) {
    if (employee.getAge() > 30) {
        filteredEmployees.add(employee);
    }
}
```

- ➔ imperative Java code, we explicitly iterate over a collection of employees, check each employee's age, and manually add the matching employees to a new list. The code specifies the exact steps to perform the filtering.
- ➔ declarative example focuses on the desired outcome (selecting names) while leaving the implementation details to the database engine.

Declarative programming example (using SQL):

```
SELECT name FROM employees WHERE age > 30;
```

- ➔ In this declarative SQL query, we specify the desired outcome (selecting names of employees above a certain age) without explicitly defining the steps to achieve it.

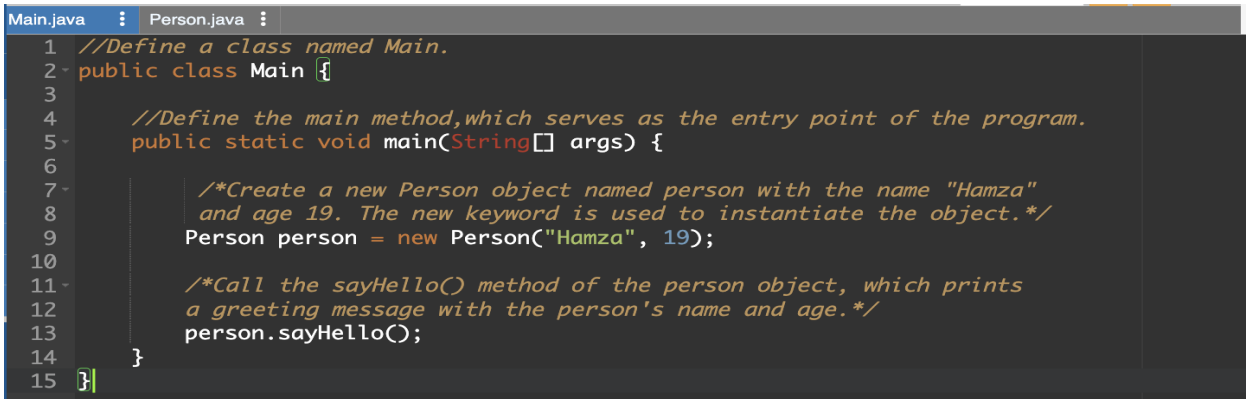
Java Code Structure:

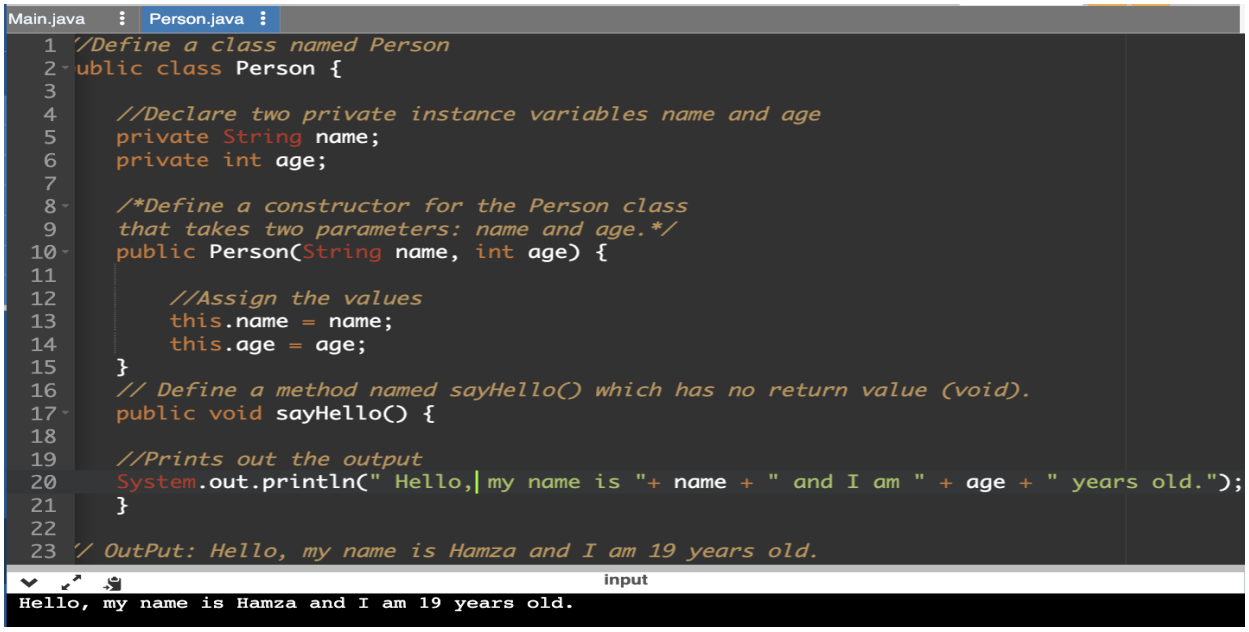
- Project File
  - Name Of the Project, Ex. Lab1ComplexNumber
- Source File
  - A file containing human-readable Java source code with a .java extension.
  - Ex. ComplexNumber.java
- Package
  - A way to organize related classes and interfaces in a hierarchical structure.
  - Ex. package coe318.lab1;
- Class File
  - A compiled version of a Java source file with a .class extension, containing bytecode. A class is a definition, Contains variables and methods

    - public class ComplexNumber { }
- Class Declaration
    - Defines a new class, specifying its name, access modifiers, inheritance, and class body.
- Class
    - A “template” that defines the type of an object.
    - Can contain - variables and methods.
- Main Method
    - The entry point for a Java program, executed first when the program starts.
- Object
    - To use a class, you need to create an object of that specific class type
    - Objects are called instances of a class, thus variables in the class are called instance variables/methods
    - variable name = new variable name
    - Have a main() method for the object, this method can be placed in the class definition itself (not recommended), or have a Helper/Tester class just for the main() method
    - To access an object use the “.” operator, object.variable (for instance variable), object.method (for instance method)
    - For each instance variable you want accessible, you must have a getter or setter methods
    - You can create multiple objects in the same class
- Variables
    - Used to store data, with specific data types (int, double, String) and names.
        - float = any number with 6-7 total digits
        - double = any number with decimals
        - Int = any integer value
        - String = “anyWord”
        - Boolean = true/false
- Data Types
    - Specify the kind of data that can be stored in variables (primitive and reference types).
- Methods
    - Blocks of code that perform specific tasks, defined within a class and callable when needed
    - Have return types and parameters
    - **Constructures:** a special method that is used when creating objects

- Statements
  - Individual instructions that perform actions and control the flow of execution.
- Import Statements:
  - Used to bring in classes or packages from other sources into the current source file.
- Comments
  - Explanatory or descriptive text within the code, ignored by the compiler.
    - /* */ (for multiple lines)
    - // (for single lines)
- Private
  - When a variable or method is declared as private, it can only be accessed and used within the class where it is defined.
- Access modifiers
  - Determine the accessibility or visibility of classes, methods, variables, and other members in Java. There are four access modifiers in Java:
    - Public: The public access modifier allows unrestricted access to the class, method, or variable from any other class or package.
    - Protected: The protected access modifier allows access within the same package and subclasses (even if they are in different packages).
    - Private: The private access modifier restricts access to only within the same class. Other classes cannot access private members directly.
    - Default (No Modifier): If no access modifier is specified, it is known as the default access modifier. It allows access within the same package but not from outside the package.
- Common Errors
  - Java is Case Sensitive most of the keywords are lowercase.
    - Ex. “class”, “public”, “main”
  - Java source file name should be the public class name plus
    - “.java”. E.g. “HelloWorld.java”
  - Semicolon at the end of a statement.
  - Braces come in matching pairs. { and }

Example:





Main.java:

```
public class Main {
   public static void main(String[] args) {
      Person person = new Person("Hamza", 19);
      person.sayHello();
   }
}
```

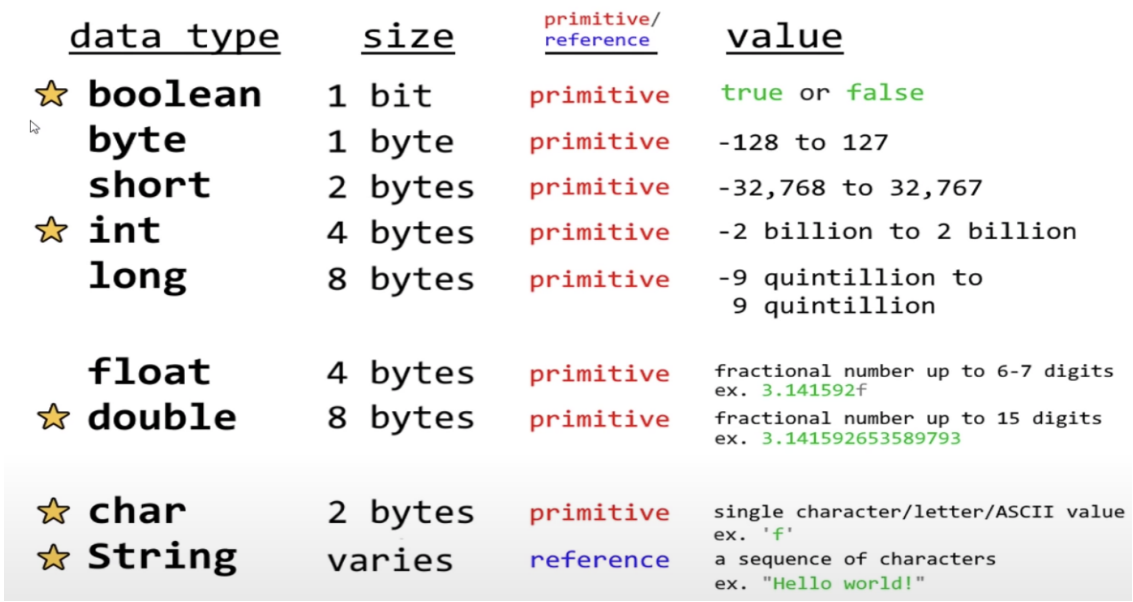
Person.java:

```
public class Person {
   private String name;
   private int age;
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
```

```
    }
    public void sayHello() {
    System.out.println(" Hello, my name is "+ name + " and I am " + age + " years old.");
    }
}
```

Lecture 2:

Primitive data types



data type	size	primitive/ reference	value
☆ boolean	1 bit	primitive	true or false
byte	1 byte	primitive	-128 to 127
short	2 bytes	primitive	-32,768 to 32,767
☆ int	4 bytes	primitive	-2 billion to 2 billion
long	8 bytes	primitive	-9 quintillion to 9 quintillion
float	4 bytes	primitive	fractional number up to 6-7 digits ex. 3.141592f
☆ double	8 bytes	primitive	fractional number up to 15 digits ex. 3.141592653589793
☆ char	2 bytes	primitive	single character/letter/ASCII value ex. 'f'
☆ String	varies	reference	a sequence of characters ex. "Hello world!"

- Byte: 1 signed byte. Each byte is 8 bits . - 128 to 127
- Char: 2 unsigned bytes unicode, 0 to 65535
- Short: 2 signed bytes -32768 to 32767
- Long: 8 signed bytes
- float = any number with 6-7 total digits (4 signed bytes)
- double = any number with decimals (8 signed bytes)
- Int = any integer value (4 signed bytes)
- String = “any word”

Arithmetic Operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Remainder or modulus: % (e.g. 8%3 is 2)

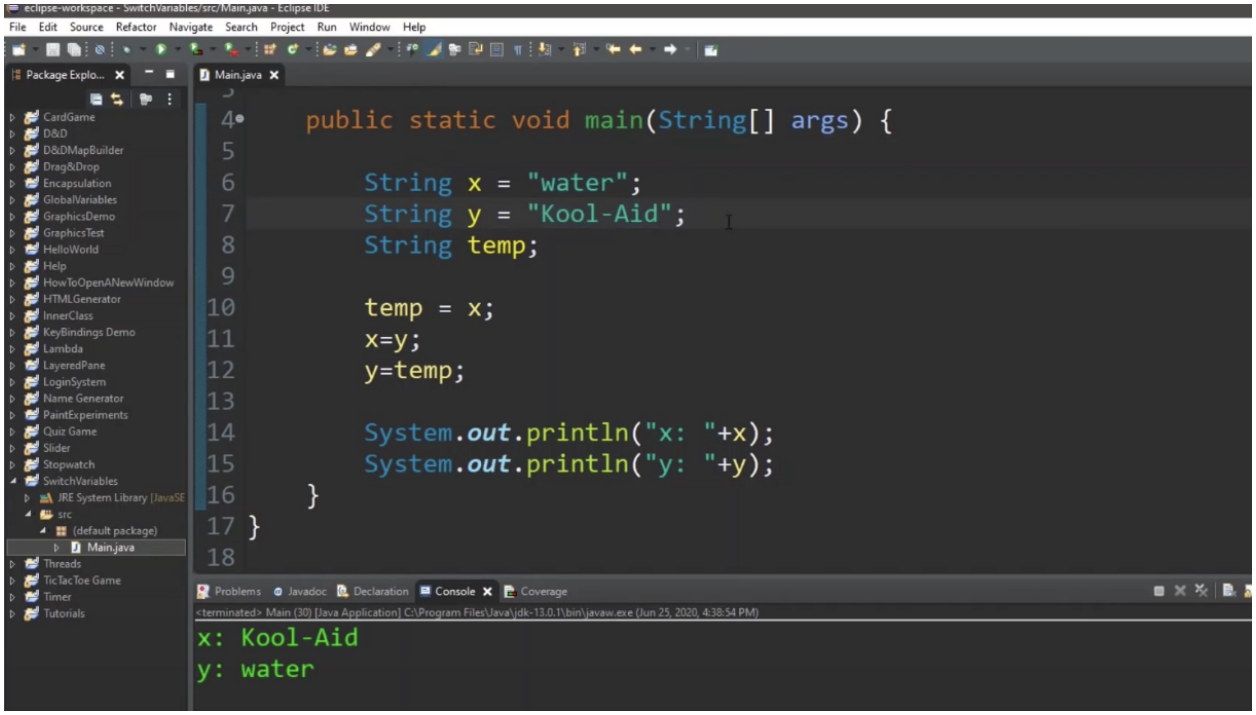
Main.java

```java
public class Main{

    public static void main(String[] args) {

        byte a = 127;
        int b = 50004;
        short c = -32435;
        long d = 4385349535L;
        float e = 1.78f;
        double f = 15609670.89;
        char g ='@';
        boolean h = true ;
        String pancho = "Hamza Malik" ;

         System.out.println("my name is " +pancho);
         System.out.println("I hold $" + c + " in the bank!");
         System.out.println("Im Broke Bitch!");
    }
}
```

input

```
my name is Hamza Malik
I hold $-32435 in the bank!
Im Broke Bitch!
```

Swap Two Variables



The “overloaded” + operator

- a+b adds the variables, if both the variables are stings, it connects the strings

```
System.out.println("1"+2+3);
System.out.println(1+2+ "3");
System.out.println("1"+ (2+3));
```

```
"123"
"33"
"15"
```

```
2+3                 Evaluates as   5
"Hello"+"There"     Evaluates as   "HelloThere"
"Hello"+2           Evaluates as   "Hello2"
```

Equality operators

- == means x equals y
- != means x does not equal y
- > means “greater than”
- < means “less than”
- >= means “greater than or equal to”
- <= means “less than or equal to”

Component Assignment Operator

Operator	Example	Meaning
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Increment/Decrement operator

- Prefix Increment eg: ++i, Increase i by 1, then use the new value of i to evaluate the expression that i resides
- Postfix Increment eg: i++ Use the current value of i to evaluate the expression that i resides, then increase i by 1
- Prefix Decrement eg: - - i, Decrease i by 1, then use the new value of i to evaluate the expression that i resides
- Postfix Decrement eg: i- -, Use the current value of i to evaluate the expression that i resides, then decrease i by 1

Conditional statements:

- Conditional statements allow you to execute different blocks of code based on specified conditions. The two main types of conditional statements in Java are:
- The if statement is used to execute a block of code only if a certain condition is true. It can be followed by an optional else statement to execute a different block of code if the condition is false.

Switch Statement

- The switch statement allows you to perform different actions based on the value of a variable or an expression. It provides a convenient way to handle multiple cases.
- Evaluates the expression and attempt to match the result from several cases
- An optional default case can be added as the last case if no other cases match the requirements
- A break statement must be used at the end of each case, if not then the operation continues to the next case until a break statement is found

```
int day = 2;
switch (day) {
    case 1:
        System.out.println("Sunday");
        break;
    case 2:
        System.out.println("Monday");
        break;
    case 3:
        System.out.println("Tuesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

In this example, if the value of the variable `day` is 2, it will print "Monday."

Given the following switch statement where x is an int,

```
switch (x)
{
    case 3 : x = x+1;
    case 4 : x = x+2;
             break;
    case 5 : x = x+3;
    case 6 : x=x+1;
    case 7 : x = x+2;
    case 8 : x=x-1;
    case 9 : x=x+1 ;
}
```

For x=3 we continue no break ← x=3+1=4; x=4+2=6; break; Skip; x=7+2=9; 9-1=8; 8+1=9

What is the final value of x if initially:

(1) x = 4 ? → 6

(2) x = 3 ? → 6

(3) x = 7 ? → 9

Loops (3 Types)

- While loop:
  - The while loop repeatedly executes a block of code as long as a specified condition is true. The condition is checked before each iteration. If the condition is false initially, the block of code will not be executed at all.

```
public class Loopy {
    public static void main (String[] args) {
      int x = 1;
      System.out.println("Before the Loop") ;
      while (x < 4) {
            System.out.println("In the loop");
            System.out.println("Value of x is " + x) ;
            x = x + 1;
      }
      System.out.println("This is after the loop");
    }
}
```

Output:

```
Before the Loop
In the loop
Value of x is 1
In the loop
Value of x is 2
In the loop
Value of x is 3
This is after the loop
```

- For loop:
  - The for loop is a control flow statement that allows you to repeatedly execute a block of code a fixed number of times. It consists of three parts:
  - initialization, condition, and increment/decrement.

```
public class Loopy {
    public static void main (String[] args) {
      int x = 1;
      System.out.println("Before the Loop") ;
      for (x=1;x < 4;x++) {
            System.out.println("In the loop");
            System.out.println("Value of x is " + x) ;

      }
      System.out.println("This is after the loop");
    }
}
```

Output:

```
Before the Loop
In the loop
Value of x is 1
In the loop
```

Value of x is 2
In the loop
Value of x is 3
This is after the loop

- Do-while loop
  - The do-while loop is similar to the while loop, but it checks the condition after executing the block of code. Body of the loop will always be executed at least once, regardless of the condition.

```
public class Loopy {
    public static void main (String[] args) {
      int x = 1;
      System.out.println("Before the Loop") ;
      do{
            System.out.println("In the loop");
            System.out.println("Value of x is " + x) ;
            x =  1;
      }while(x!=1);}}
```

Output:
Before the Loop
In the loop
Value of x is 1

```
public class Loopy {
    public static void main (String[] args) {
      int x = 1;
      System.out.println("Before the Loop") ;
      while(x!=1) { //code never enters the loop!
            System.out.println("In the loop");
            System.out.println("Value of x is " + x) ;
            x =  1;
      }}}
```

Output:
Before the Loop

Examples:

Given the following code, where x = 0, what is the resulting value of x after the for-loop terminates?

```
int x=0;
for (int i=0;i<5;i++)
      x = x+i;
```

How many times will the following loop iterate?

```
int x = 10;
while (x > 0){
     System.out.println(x);
     x=x-1;
}
```

```
for (x=1;x < 40;x++) {
     System.out.println("In the loop" + x);
}
```

```
int x=3;
while (x<40) {
     System.out.println("In the loop" + x);
}
```

```
int x=3;
while (x<40) {
     System.out.println("In the loop" + x);
          x=x+1;
}
```

```
int x=50;
do {
     System.out.println("In the loop" + x);
} while (x<40);
```

```
int x,y;
for (x=1;x < 40;x++) {
     for(y=1;y<40;y++){
          System.out.println("In the loop");
     }
}
```

Conditional Operator

- concise way to make decisions based on a condition and choose between two values or expressions. It provides a shorthand alternative to if-else statements, allowing you to express conditional logic in a compact form.
- Condition ? Do_if_true : Do_if_false
- It can be used instead of the if-else statement
- 3 operands
    - Boolean expression, conditional expression (true), conditional expression (false)

```
int number = 7;
String message = (number % 2 == 0) ? "Even" : "Odd";
System.out.println("The number is " + message);
```

In the given example, the variable `number` is assigned a value of 7. If the condition `(number % 2 == 0)` is true (which it is not), "Even" is assigned to `message`; otherwise, "Odd" is assigned. The resulting value "Odd" is printed to the console.

The general form of a class definition

```
class classname
{
   //declare variables
   type var1;
   type var2;
   ….
   //declare methods
   type method(parameters)
   {
      //body of method, consists of statements
    }
   type method2(parameters)
   {
      //body of method, consists of statements
   }
   ….
}
```

The general form of a method

```
ReturnType methodName(parameter-list) {
    //body of method
}
```

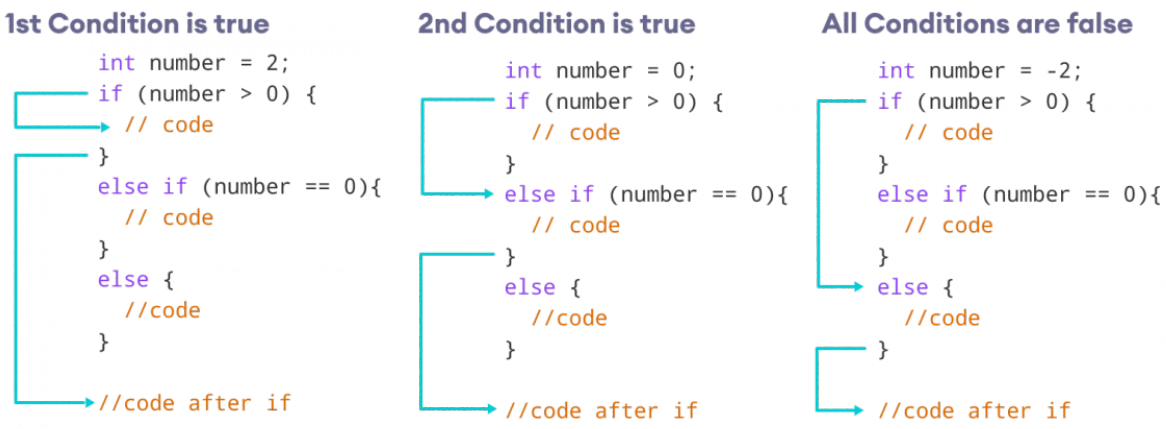
- methodName is the name of the method as a string with no space. Capitalize the first letter of each word except the first
- ReturnType is any primitive type or object reference. If nothing is returned from the method to the method caller, a "void" is declared
- Parameter-list is a comma-separated list of type and variable pair
- The body of the method consists of statements

Getter and Setter methods

```
public double getFuel() {
  return fuel;
}
public void setFuel(double val) {
  fuel=val;
}
```

- Getters and setters are used to protect your data, particularly when creating classes. A getter method returns its value for each instance variable while a setter method sets or updates its value.

If, else, and else if Statements



Quiz 1:

Quiz 1: Obtained 100% For Lectures 1 and 2

Q1

Time Limit: 0:15:00 Time Left:0:05:04 Hamza Malik: Attempt 1

Page 1: 1 2 3

Question 1 (1 point) Saved

Consider the following code fragment

```
1  public class Dog {
2      public void bark() {
           System.out.println("B"); }
3      public static void main (String[] args) {
4          bark(); }}
```

Which line will result in a compile error?

- ○ Line 1
- ○ Line 2
- ○ Line 3
- ◉ Line4

Question 2 (1 point) Saved

What will be the output of the following program?

```
public class Test {
    public static void main(String[] args){
        System.out.println(2 + 5 + "1"+ (2+3));
    }
}
```

715

Question 3 (1 point) Saved

Given the following switch statement where x is an int,

```
switch (x)
{
   case 3 : x = x+1;
   case 4 : x = x+2;
            break;
   case 5 : x = x+3;
   case 6 : x=x+1;
   case 7 : x = x+2;
   case 8 : x=x-1;
   case 9 : x=x+1 ;
}
```

What is the final value of x if initially:

1. **x = 5**

2. **x = 6**

11

9

Question 1 Answer:

The line that will result in a compile error is line 4: `bark();`.

In the given code fragment, the `main` method is declared as `static`. Static methods can only directly call other static methods or access static variables. However, the `bark` method in the `Dog` class is an instance method, not a static method. Therefore, attempting to call `bark()` directly from the `main` method without creating an instance of the `Dog` class will cause a compile error.

To fix the error, you would need to create an instance of the `Dog` class and then call the `bark()` method on that instance. For example:

java Copy code

```java
public static void main(String[] args) {
    Dog dog = new Dog();
    dog.bark();
}
```

This code creates a new instance of the `Dog` class called `dog` and then calls the `bark()` method on that instance.

Question 2 Answer:

The output of the program will be:

Copy code

```
715
```

Let's break down the expression:

1. `2 + 5` evaluates to `7`.
2. `"1"` is a string literal.
3. `(2 + 3)` evaluates to `5`.

The expression `2 + 5 + "1" + (2 + 3)` is evaluated from left to right, following the operator precedence rules:

1. `2 + 5` results in `7`.
2. `"1"` is concatenated with the previous result, resulting in `"71"`.
3. `(2 + 3)` results in `5`.
4. Finally, `"71"` is concatenated with `5`, resulting in the final string `"715"`.

Therefore, the program will output `715`.

Lecture 3

Method Overloading

- Allows different methods to have the same name, but different parameters signatures (number of input parameters, types of parameters)
- Overloading: varying return types does not qualify, parameters must be different

```
public class Calculator {
   public int add(int a, int b) {
      return a + b;
   }

   public int add(int a, int b, int c) {
      return a + b + c;
   }
}
```

Mutable vs Immutable Class

- Mutable class: Object state can be changed after creation (using getter and setter or a constructor)
- Immutable class: Set the instance variables only during the creation
- To convert an immutable class to a mutable class add a constructor

```
public class Person {
    private String name;
    private boolean isMale;
    public String getName() {
        return name;
    }
    public boolean isMale() {
        return isMale;
    }
}
```

Immutable class example

Constructor

- The method is the same name as the class
- Used to initialize a new object
- Does not return a value, meaning it has no return type, not even void
- You only need to define a constructor when you assign initial values to instance variables of each created object
- If a class has no constructor the compiler generates a default constructor
- Constructors can be overloaded too (you can have multiple constructors)
- Sets multiple instance variables (variable name = new variable name)

```
public class Person {
    private String name;
    private boolean isMale;

    public Person(String n, boolean x) {
        name = n;
        isMale = x;
    }
    public String getName() {
        return name;
    }
    public boolean isMale() {
        return isMale;
    }
}
```

The “this.” keyword

- “This” refers to the current object
- Allows the same instance variables to be used as parameters

Types of Variables

- Based on data type:
    - Primitive data types (int a, short b, etc)
    - Reference variables: any data of type object (car, miniVan, String name, etc)
- Based on usage
    - Instance variables
    - Parameters (of methods)
    - Local variables

```
public class C {
   private int i; //instance variable
   public void f(int j) { //j is a parameter
        int k; //k is a local variable
        k = 2*j;
        i = i + k;
   }
    public void g() {
        int k; //another local variable named k
    }
}
```

Class as Data type

- Once you define a class, it becomes a "data type"
- You can declare variables as the class type
- You can have methods return the class type
- Can be a parameter type in a method

Conversion between primitive types

- Widening conversions: from a small data type to a larger one

byte → short → int → long → float → double

- Narrowing conversions: Can lose information because they tend to go from a large data type to a smaller one

Casting

- Both types of conversions can be achieved by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted

```
int total, count;
float result = (float) total / count;
int total1 = (int) result;
```

- Casting is a must when dividing integers by integers and you want a floating point. Without casting, the result will throw out the numbers after the decimal point

Variable Initialization

- Local variables are not automatically initialized; using them before they are set causes a compilation error
- Instance variables are automatically initialized to 0 (for numbers), the character '\0' for chars, and null for references

***Null: (a keyword in Java) can represent a reference to any type of object

Data Storage in Memory

- Primitive variables: have their own size
- Reference variables: The variables itself require 4 bytes (or 8 bytes in a 64-bit machine) and any memory required for the instance variables of the object
- Heap: Any memory required by an object goes into the special section of the memory
- Stack: Local variables and parameters, however, reside elsewhere: on a data structure that can grow and shrink dynamically

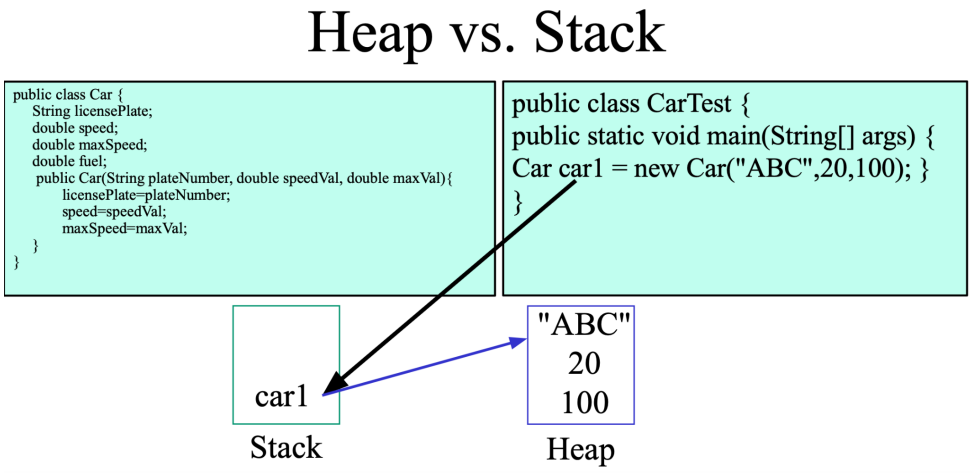
Heap vs. Stack

Data storage in memory - Heap

- Any memory required by an object goes into The special section of the memory called the heap

Data storage in memory - Stack

- Local variables and parameters, however, reside elsewhere: on a data structurethat can grow and shrink dynamically called the Stack



Variable Timelife

- When a method is invoked, the parameters are first pushed onto the stack. When the method is entered, additional space on the stack is reserved for all local variables. The method's code is then executed
- When the method finishes, it releases the space on the stack occupied by its local variables and parameters automatically, meaning they no longer exist

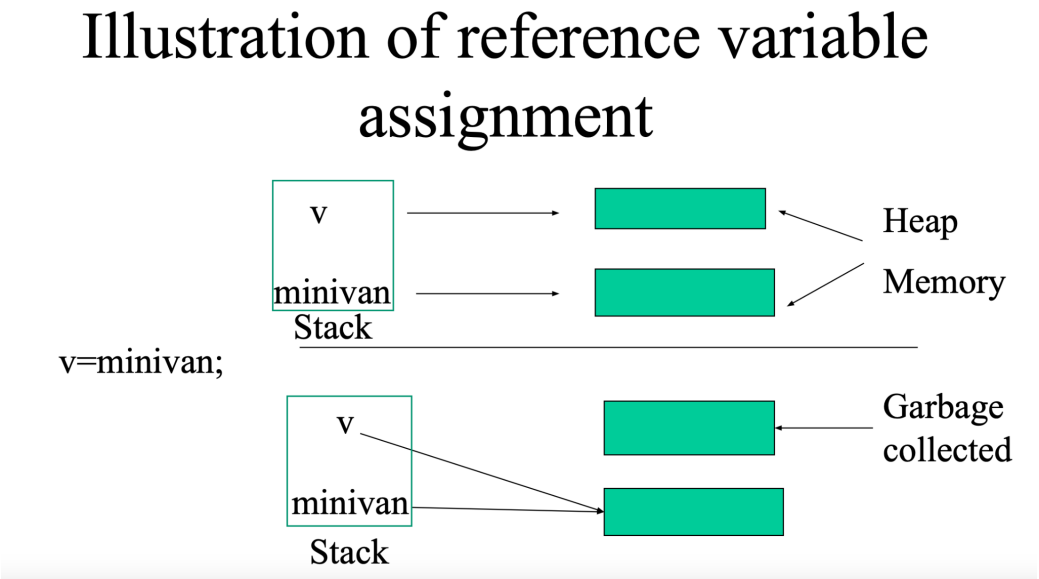
Garbage Collection

- To manage heap memory, Java maintains a reference count of how many reference variables point to it
- Reference drop count to zero → The object is eligible for garbage collection which reclaims the heap space used by the object
- Garbage collection relieves the programmer from the responsibility of releasing dynamically allocated heap memory

Reference variable and assignment

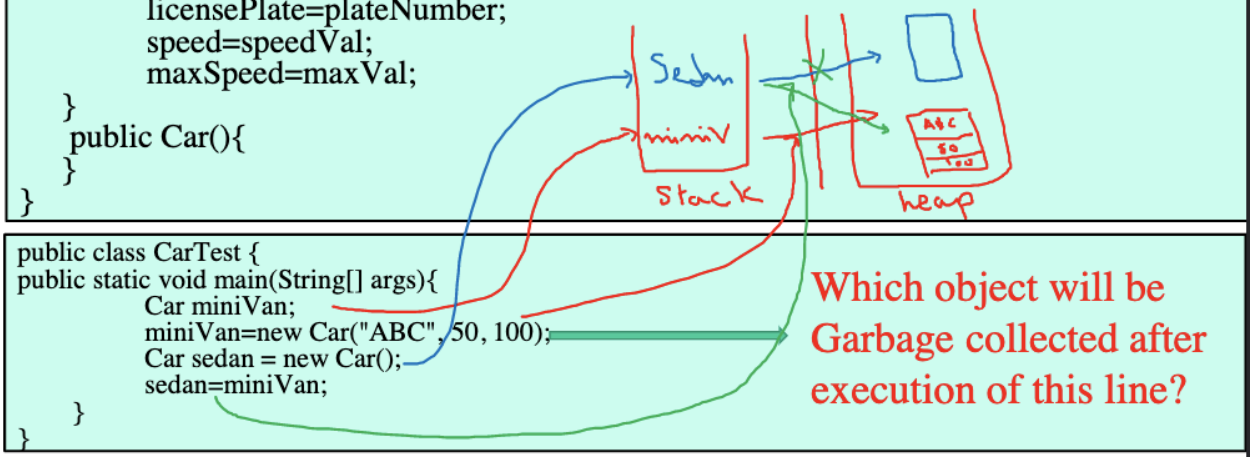
- When you assign one reference variable to another reference variable, the second variable begins to reference the object the first reference variable is referencing to
- If no more object variables refer to the memory block, it will be collected by the garbage collector

Illustration of reference variable Assignment



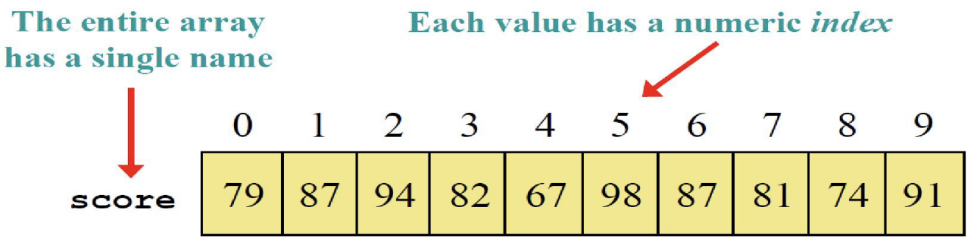
```
public class Car {
    String licensePlate;
    double speed;
    double maxSpeed;
    double fuel;
     public Car(String plateNumber, double speedVal, double maxVal){
            licensePlate=plateNumber;
            speed=speedVal;
            maxSpeed=maxVal;
    }
     public Car(){
    }
}
```

```
public class CarTest {
public static void main(String[] args){
            Car miniVan;
            miniVan=new Car("ABC", 50, 100);
            Car sedan = new Car();
            sedan=miniVan;
    }
}
```



Arrays (3 elements)

- An array stores multiple values of the same type
- That type can be primitive types of objects
- An array of size N is indexed from zero to N-2



- Declaring Arrays:
    - Defines the variable type with square brackets []

Declaring Arrays

```
int [] scores;
scores=new int[10];
```

```
int [] scores = new int[10];
```

```
int scores[10];
```

- Initializing Arrays:
  - Assigning values to a new array

Initializing Arrays

```
int[] units = {147, 323, 89, 933, 540, 269, 97, 114, 298, 476};
char[] letterGrades = {'A', 'B', 'C', 'D', 'F'};
```

- Accessing Arrays Elements:
  - An index used in an array reference must specify a valid element
  - The index value must be in bounds (0 to N-1), where N is the length
  - Java interpreter throws an ArrayIndexOutOfBoundsException error if an array is out of bounds

Accessing Array elements

```
int scores[3];
scores[0] = 79;
scores[1] = 87;
scores[2] = 92;
mean = (scores[0] + scores[1]+scores[2])/3;
System.out.println ("Average = " + mean);
```

The length Property

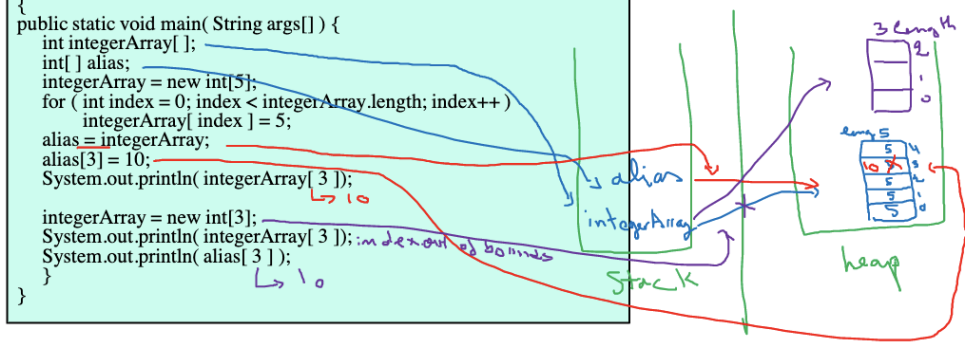
- Each array object has a public constant called length that stores the size of the array

```
public class Primes
{
public static void main (String[] args)
  {
        int [] primeNums = {2, 3, 5, 7, 11, 13, 17, 19};
        System.out.println ("Array length: " + primeNums.length);
        for (int i=0; i< primeNums.length; i++)
                System.out.println (primeNums[i]);
  }
}
```

Arrays in Memory

- the reference types in Java are stored in heap area

```
public class ArrayExamples
{
public static void main( String args[] ) {
    int integerArray[ ];
    int[ ] alias;
    integerArray = new int[5];
    for ( int index = 0; index < integerArray.length; index++ )
          integerArray[ index ] = 5;
    alias = integerArray;
    alias[3] = 10;
    System.out.println( integerArray[ 3 ]);

    integerArray = new int[3];
    System.out.println( integerArray[ 3 ]);
    System.out.println( alias[ 3 ] );
    }
}
```

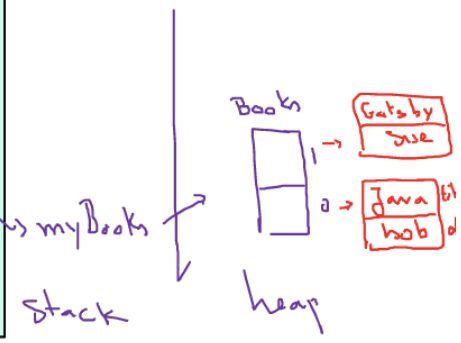


Arrays of Objects

- The element of an array can be the object themselves
- When an array is created, memory is not automatically allocated for the objects
- Each array element will have to be individually allocated memory explicitly (initially contains null)

FIND OUTPUT

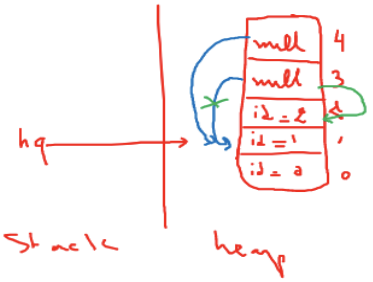
```
Class Books {
      String title;
      String author;
}
class BooksTestDrive {
   public static void main (String [] args) {
        Books [] myBooks = new Books[2];
         int x=0;
         myBooks[0] = new Books();
         myBooks[1] = new Books();
         myBooks[0].title = "The Grapes of Java ";
         myBooks[0].author = "bob";
         myBooks[1].title = "The Java Gatsby ";
         myBooks[1].author = "sue";
         While (x<2) {
                System.out.print (myBook[x].title);
                System.out.print (" by ");
               System.out.println (myBooks [x].author);
                x = x+1;
} } }
```



FIND OUTPUT

```
class HeapQuiz {
   int id = 0;}

class HeapQuizTest{
public static void main (String [] args) {
      int x = 0;
      HeapQuiz [] hq = new HeapQuiz[5];
      while (x < 3) {
               hq[x] = new HeapQuiz();
                hq[x].id = x;
                x =x +1; }
       hq[3] =hq[1];
       hq[4] =hq[1];
       hq[3] =hq[2];
       for (int j=0; j<5; j++)
             System.out.println ("hq["+j +"]:"+hq[j].id);
       }
   }
}
```

0 1 2 2 1



Lecture 4

The String class

- One of the many examples of Java built-in classes

String object

```
String licensePlate;
licensePlate= new String ("ABC 41");
```

String literal

```
String licensePlate="ABC 41";
String licensePlateTwo="ABC 41";

//A special case only for Strings
//new memory is not allocated if same string, existing //string is used
```

Comparing Variables

Floats

- Floats need to be matched exactly
  - Not possible sometimes due to approximations on the hardware
  - To fix this, define a variable tolerance with a small value. Example:

```
if(Math.abs(a-b)<TOLERANCE){
}
```

Chars

- 16-bit Unicode for all characters
- 38885 characters currently are defined with Unicode values
- The low 8 bits can be used for ASCII

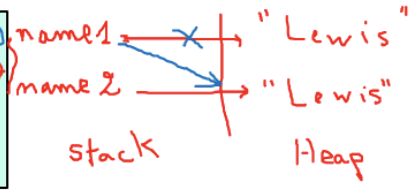
Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

```
if('0'<'L'){
}
```

Strings

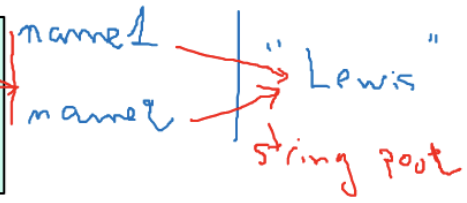
- Remember, String variables are references
- References can be compared with the "= =" operator

```
String name1 = new String("Lewis");
String name2 = new String("Lewis");
if(name1==name2){
   //condition will be evaluated to false

}
```




```
String name1 = new String("Lewis");
String name2 = new String("Lewis");
name1=name2;
if(name1==name2){
   //condition will be evaluated to true

}
```

```
String name1 = "Lewis";
String name2 = "Lewis";
if(name1==name2){
   //condition will be evaluated to true
   //String literals use String pool, no new //memory
allocated if the strings are the same
}
```




```
String name1 = new String("Lewis");
String name2 = new String("Lewis");
if(name1.equals(name2)){
   //condition will be evaluated to true

}
```

The equals() method

Comparing Objects

- you can compare objects with '==' or 'equals()'

Comparing objects with ==

```
public class CarTest {
public static void main(String[] args){
        Car miniVan = new Car("ABC", 50, 100);
        Car sedan=new Car("ABC", 50, 100);
        if(sedan==miniVan){
        //condition will be evaluated to false
        }
    }
}
```

Comparing objects with equals()

```
public class CarTest {
public static void main(String[] args){
        Car miniVan = new Car("ABC", 50, 100);
        Car sedan=new Car("ABC", 50, 100);
        if(sedan.equals(miniVan)){
      //condition will STILL be evaluated to false
        }
    }
}
```

Parameters Passing in Java methods

- If the parameter is of primitive data type, the parameter is passed by value
    - Changing the parameter value inside the method has no effect on the original value

Parameter Passing by Value:

```
public static void main(String [] args){
        double d = 2.0;
        changeMe(d);
        System.out.println(d);
}
public static void changeMe(double d)
{
    //this has no effect on d outside of this method!
    d = 345.0;
}
```

Parameter Passing by Reference:

- If the parameter is an object, the parameter is passed by reference
    - Changing the parameter value inside the method changes the original value

```
public static void main(String[] args){
      Car miniVan = new Car("ABC", 50, 100);
      changeParameters(miniVan);
      System.out.println(miniVan.speed);


}
public static void changeParameters(Car c)
{
    //changes the car's speed outside this method!
     c.speed=30;
}
```

Linking Objects

- A variable can hold a reference to another object. When one object contains a variable that refers to another object, we think of the objects as being "linked" by the reference.
- Any variable that can contain a reference to an object can also contain the special value null, which refers to nowhere.

```
class EmployeeTest{
    public static void main(String[] args){
                    Employee emp, emp1;
                    emp1 = new Employee("Stacey", null);
                    emp = new Employee("Gavin", emp1);
                    if ( emp.supervisor == null) {
                                System.out.println(emp.name + " is the CEO!");}
                    else {//Since we are checking emp, this line will be printed
                System.out.println( "The supervisor of " + emp.name + " is " + emp.supervisor.name ); }
} }
```

Quiz 2

Quiz 2: Obtained … For Lectures 3 and 4

Q2

Hamza Malik: Attempt 1

Page 1:

1 2 3 4

```
        maxSpeed=maxSpeed;
    }

  // Call the constructor
  public static void main(String[] args) {
    Main myObj = new Main("ABC", 5, 213);
    System.out.println("Value of x = " + myObj.licensePlate);
    System.out.println("Value of speed = " + myObj.speed);
    System.out.println("Value of maxVal = " + myObj.maxSpeed);
  }
}
```

null

0.0

0.0

Question 2 (1 point) *Saved*

For each of the following statements, please answer True or False.

In the same class, two method declarations with the same names and only one difference in the number of parameters are examples of different versions of an overloaded method.

- (•) True
- ( ) False

Question 3 (1 point) *Saved*

Inside the same class, two method declarations with the same names and only one difference in return types are examples of different versions of an overloaded method.

- ( ) True
- (•) False

Question 4 (1 point) *Saved*

In the same class, two method constructors with different names and different parameter numbers are examples of different versions of overloaded constructors.

- ( ) True
- (•) False

Submit Quiz *4 of 4 questions saved*

Midterm outline

- **Approx 6-7 questions**
- **true/false**
- **Finding output**
- **Fill in the blanks**
- **Finding errors**

Lecture 5

Java Packages

- Packages are used in Java, in order to avoid name conflicts
- A package can be defined as a group of similar types of classes and interface
- Using Java packages, it becomes easier to locate the related classes
- Built-in package: Existing Java package. Example: java.lang, java.util, etc
- User-defined-package: Java package created by the user to categorize classes
- Java uses a file system directory to store packages

```
package coe318;
public class Lab3 {
        int labNum=3;

}
```

Declaring a package
with the "package" keyword
(NetBeans does it automatically)

```
//this code is in another package
import coe318.*;
public static void main (String[] args) {
        Lab3 l=new Lab3();
}
```

Using the class OUTSIDE the same package requires use of the "import" keyword
Option 1: import everything
Can make code slow if package is big

```
//this code is in another package
import coe318.Lab3;
public static void main (String[] args) {
        Lab3 l=new Lab3();
}
```

Option 2: import specific class

Encapsulation

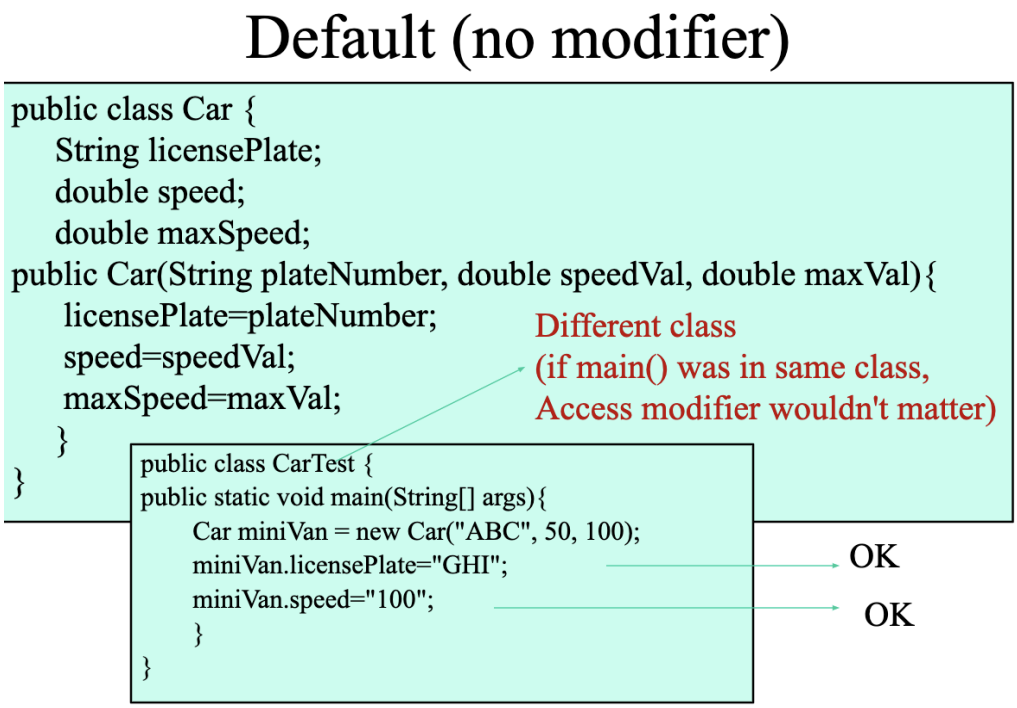
- A program or object may use methods from another object
- To protect the instance variables, we should make it difficult for a main() or object to access another object's variables directly
- Encapsulation: Hiding the variables within the class, and making it available only through the methods

Visibility/Access Modifiers

- We accomplish encapsulation through the appropriate use of visibility modifiers
- Java has four visibility modifiers: public, private, default (package), and protected
- The protected modifier involves inheritance, which we will discuss later
- If no modifier is mentioned, default access modifier: variables accessible by any class in the same package

Public vs Private

- Instance variables of a class that are declared with public visibility can be referenced anywhere
- Methods and variables that are declared private can only be accessed within the declared class itself



Public and private

```
public class Car {
    private String licensePlate;
    public double speed;
    public double maxSpeed;
public Car(String plateNumber, double speedVal, double maxVal){
     licensePlate=plateNumber;
     speed=speedVal;
     maxSpeed=maxVal;
    }
}
```

```
public class CarTest {
public static void main(String[] args){
      Car miniVan = new Car("ABC", 50, 100);
      miniVan.licensePlate="GHI";
      miniVan.speed=100;
      }
}
```

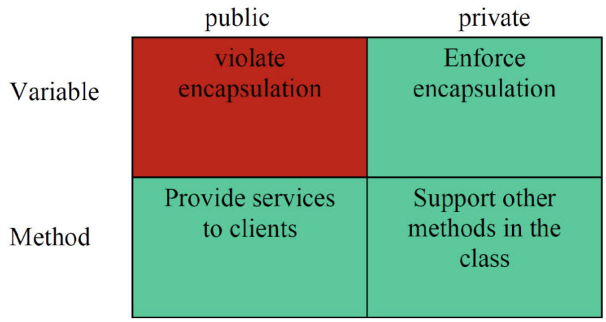
Illegal!

OK

```
class Modifiers {
   public int pubData;
   private int privData;

   private void privMethod( ) {}

   public void pubMethod( ) {
        privMethod( ); //private method is called from within the same class, OK
   }
}
class ModifierDemo {
     public static void main( String[ ] args ) {
          Modifiers mod = new Modifiers( );
          mod.pubData = 1; // Okay: pubData is public
          mod.pubMethod( ); // Okay: pubMethod is public
          mod.privData = 1; // Illegal! privData is private
          mod.privMethod( ); // Illegal! privMethod is private
           }
}
```

Rule of thumb for visibility Modifiers

- As a general rule, no object‘s instance variables should be declared with public visibility
- Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly
- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public methods are also called service methods
- Typically methods are public
- A private method created simply to assist a service method is called a support method



Accessor (getter) and Mutator (setter)

- Because the instance variable is private, a class usually provides methods to access and modify instance variables values
- An accessor (getter) method returns the current value of a variable
- A mutator (setter) method changes the value of a variable

Encapsulation Enforced

- A combination of access modifiers, constructors, and getter/setter methods can help only exposing the instance variables that a developer wants to be exposed
- The rest can be set through constructors

Encapsulated Car class from Lecture 2 (only speed has getter/setter)

```
public class Car {
    private String licensePlate;
    private double speed;
    private double maxSpeed;

    public Car(String plateNumber, double speedVal, double maxVal){
            licensePlate=plateNumber;
            speed=speedVal;
            maxSpeed=maxVal;
    }
    public void setSpeed (double speedVal) {
            speed=speedVal;
    }
    public double getSpeed () {
            return speed;
    }
}
```

Scope of a variable: Instance variables have class scope

- The instance variables of a class have class scope. Class scope begins at the opening left brace and ends at the closing right brace of the class definition
- Class scope allows any methods in the class to directly access any instance variable
- In effect, instance variables are "global variables" within a class

Scope of a variable: Local variables have block scope

- A block is a compound statement and consists of all the statements between an opening and closing brace
- Local variables defined within a block have block scope; not visible outside the block
- A local variable can have the same name as an instance variable. In this case, the instance variable is hidden from the method by the local (hence the "this" keyword)

Arrays as parameters

- Passing entire array = pass by reference - same behavior as passing any object

```
void aProc() {
    int[] xyz = new int[12];
    theProc(xyz);
    System.out.println(xyz[3]);
}
void theProc(int[] mno)
{
    mno[3] = 15;
 }
```

- Passing individual array element = pass by value - same behavior as passing a primitive

```
void aProc() {
    int[] xyz = new int[12];
    theProc(xyz[3]);
    System.out.println(xyz[3]);
}
void theProc(int mno)
{
    mno = 15;
}
```

```
public class ArrayTest {
    public static void main(String[] args) {
       int[] test = new int[2];
       test[0] = 10;
        test[1] = 5;
       System.out.println(test[0] + "," + test[1]);
       fiddle(test, test[1]);
       System.out.println(test[0] + "," + test[1]);
   }
    static void fiddle(int[] test, int element) {
       ___ = 15;
       ___ = 19;
       _____ = 12;
        System.out.println(___+ "," + ___ + ","
        +    element);
        test = new int[2];
        test[0] = ___;
        test[1] = ___;
        System.out.println(test[0] + "," + test[1]);
    }}
```

Given the following program output, Fill in the blanks

10, 5

19, 15

15, 19,12

20, 21

Two Dimensional (2D) Arrays

- In Java, a two-dimensional array is an array of arrays
- If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns
- In reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints
- An array element is referenced using two index values: int value = A[1][1]

Initializing 2D Arrays

```
int[][] A = { { 1, 0, 12, -1 },
{ 7, -3, 2, 5 },
{ -5, -2, 2, 9 }
};
```

or

```
int[][] A = new int[3][4];
A.length: the number of rows of A.
A[0].length: the number of columns in A
A[1].length : the number of columns in A
A[2].length : …..
```

2D arrays of objects

```
public class 2DArray
{
    public static void main (String[] args)
    {
      String [][] A = {
            {"Hello", "World" },
            {"Guten", "Welt"}
      };

      for (int row=0; row < A.length; row++){
         for (int col=0; col < A[row].length; col++)
            System.out.println (A[row][col]);
            }
    }
}
```

Interaction - taking input from the user

- Java has a nice built-in class scanner that can simplify taking input from the user
- The scanner class is part of java.util class library (package), and must be imported into a program to be used

Instantiating Scanner

```
import java.util.Scanner;
….
……
Scanner s= new Scanner(System.in);
```

If you want to read from console

```
import java.util.Scanner;
….
……
Scanner s= new Scanner(new FileReader("myfile")
```

If you want to read from file

Reading a line with a scanner

```
import java.util.Scanner;
class Echo {
    public static void main (String[] args) {
    String message;
    Scanner scan = new Scanner (System.in);
    System.out.println ("Enter a line of text:");
    message = scan.nextLine(); //app waits here for user input
    System.out.println ("You entered:" + message);
    }}
```

Looping input with stop condition

```
import java.util.Scanner;
class Echo {
    public static void main (String[] args) {
    String message="";
    Scanner scan = new Scanner (System.in);
   while(!message.Equals("quit")){
        System.out.println ("Enter a line of text:");
         message = scan.nextLine();
         System.out.println ("You entered:" + message);
         }
   }
}
          //can you make this code case-insensitive? (hint:
          toLowerCase())
```

Reading input as tokens

- A scanner breaks its input into elements (tokens) using white space. The resulting tokens may then be converted into values of different types using various methods already defined in the scanner class

```
import java.util.Scanner;
class Age {
    public static void main (String[] args) {
    String message;
    Scanner scan = new Scanner (System.in);
    System.out.println("Enter your first name & year of birth");
    String firstName=scan.next(); //reads string till finds a space
    int yob=scan.nextInt(); //reads int till finds a space
    System.out.println("Hello "+firstName + "," + "you are " +
    (2021-yob)+ " years old!");
   }
}
```

- Scanner methods that read numeric data throw exception (error) if the next value isn't what the method expects
- Boolean methods can help to check ahead

Method	*Returns*
`boolean hasNextLine()`	Returns `true` if the scanner has another line in its input; `false` otherwise.
`boolean hasNextInt()`	Returns `true` if the next token in the scanner can be interpreted as an `int` value.
`Boolean hasNextFloat()`	Returns true if the next toke in the scanner can be interpreted as a `float` value.

Lecture 6

Enhanced for loop

- Instead of using the index explicitly in a loop, enhanced for loop can:

```
int[] arrayOfInts = { 32, 87, 3, 12, 8};
for (int element : arrayOfInts)
{
    System.out.print(element + " ");
}
```

The (built-in) object class

- Defined in the java.lang package of the Java standard class library
- All classes are (implicitly) derived from the object class

Typecasting objects

- One object reference can be typecast into another object reference. The cast can be to one of its subclass or superclass types
- The casting of object references depends on the relationship of the classes involved in the same hierarchy
- Any object reference can be assigned to a reference variable of the type object because the object class is a superclass of every Java class
- When we cast a reference along the class hierarchy in a direction from the subclasses toward the root, it is an upcast. We don't need to use a cast operator in this case

```
public class Vehicle{
   public void motor() { // ... }
}
```

```
public class Car extends Vehicle {
   public void motor() { // ... }
   public void wheel() {// ... }
}
```

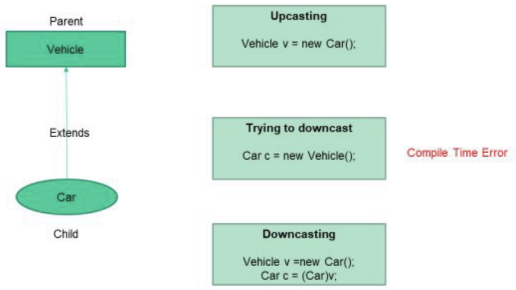
1. create an object of *Car* class and assign it to the reference variable of type *Car*:

```
Car miniVan = new Car();
```

2. assign it to the reference variable of type *Animal*

```
Vehicle animal = car;
```

- When we cast a reference along the class hierarchy in a direction from the root class towards the children or subclasses, it is a downcast
- If it is performed directly, the compiler gives an error as ClassCastException at runtime



Typecasting objects

```
Object obj;
Student st;
Object obj = st; //upcast, no need to explicitly cast
Student st = (Student) obj; //downcast, must be explicit
```

ArrayList

- The problem with arrays is that the length must be mentioned when being created and can't be changed later
- The ArrayList class is part of the java.util package
- Like an array, it can store a list of values and reference each one using a numeric index
- Unlike an array, an ArrayList object grows and shrinks as needed

Force ArrayList to accept only one kind of Object

```
ArrayList <Car> myList = new ArrayList <Car> ();
```

Command-line arguments

- We have been using: public static void main(String[] args)

- String[] args: command line arguments (array of strings) that you can use to pass arguments to the main method

NetBeans can be configured to pass arguments

```
public class HelloWorld {
    public static void main (String[] args) {
      for (int i = 0; i < args.length; i++)
          System.out.println(args[i]);
    }
}
```

Variable length parameters

- Suppose we wanted to create a method that processed a different amount of data from one run to the next
- The type of parameter can be any primitive or object type
- A method that accepts a variable number of parameters can also accept the other parameters
- There can be only one variable number of parameters in a method
- The varying length must come last in the formal arguments

```
public void test (int count, String name, double ... nums)
{ }
```

Static Variables and methods

- Each object of a class has a new instance for each variable
- A method or instance variable that is declared static exists as part of the class and no objects of the class need to be created to access any static variables or method

```
public class Car {
    String licensePlate;
    double speed;
    double maxSpeed;
    double fuel;
    static String navigator;
     public Car(String plateNumber,
double speedVal, double maxVal){
     licensePlate=plateNumber;
     speed=speedVal;
     maxSpeed=maxVal;}}
```

```
public class CarTest {
public static void main(String[] args){
   Car miniVan=new Car("ABC", 50,
100);
   Car sedan = new Car("DEF", 60, 200);
   Car.navigator="Google Maps";
//no need for object  to access static var
System.out.println(sedan.navigator);} }
//notice same value for both objects
```

- Static variables and methods are also called class variables and methods (as opposed to instance variables and methods)

- Main is always static: public static void main(String[] args)
    - This is because Java needs an entry point before any object is created
- Only static variables/methods can be invoked directly without an instance of the class, as they only exist with the class, not with each instance separately
- A common Java error is “non-static variable cannot be referenced from a static context”
- If a class has a static method, it cannot access non-static variables without initializing an instance of the class explicitly

When to use static variables

- When a variable does not need to be set specifically for each object, but should be “global” among all objects, for example, keeping count of objects

When to use static methods

- When a method solely depends on the arguments passed to it (for example, performing a calculation) and does not depend on object-specific instance variables
- Example: Java has “utility” classes like math, where all methods are static, and can be called without instantiating a math object, ex: Math.random(), Math.abs()

The “final” keyword

- When a variable is declared as “final,” the value can't be changed once initialized
- Used for constant values
- Final variables must be initialized right away or must be initialized in the constructor. If multiple constructors, initialize every time

```
public class Car {
    String licensePlate;
    double speed;
    final double maxSpeed;
    double fuel;
    static String navigator;
     public Car(String plateNumber,
double speedVal){
      licensePlate=plateNumber;
      speed=speedVal;
            maxSpeed=120;
\\ Compile error if not initialized }}
```

```
public class CarTest {
public static void main(String[] args){
   Car miniVan=new Car("ABC", 50);
   miniVan.maxSpeed=200; \\ERROR
 }}
```

- The “Final” keyword can be used for method parameters as well, for security (the arguments value can't be changed inside the method)

Static final variables

- Variables can be declared “static final: if only one constant value should exist across the class, regardless of objects. For example, Java uses it in the math class for PI.

- Must be initialized when declared

```
Class Math{
public static final double PI = 3.141592653589793;
……
}
```

Order of modifiers

- In Java, the order of modifiers does not matter
- Could write “private static final double c” or “static final private double c” or “final static private double c”
- However, recommended is: “private static final double c”

Lecture 7

Testing

- Testing is the process of running a program on a set of test cases and comparing the actual results with the expected results
- A test case tests the response of a single method to a particular set of inputs
- Unit testing: A test of a single class (ex. calculatioins)
- Integration Testing: a test of how well classes work together. Test the combination of two or more classes
- Unit testing should always precede integration testing

JUnit

- A simple tool for unit testing in Java
- Integrated with Netbeans

How to Write Unit Tests

- Unit tests typically test each component of the class to see if they behave as expected
- Step to write Unit tests:
    - Identify the test cases: Think about different scenarios or inputs
    - Write test methods: Create separate test methods for each test case you identified
    - Set up the test environment: If required, prepare the necessary objects or data that the method under test might need
    - Execute the method: call the method being tested with the provided inputs and store the results

- Verify the result: Use assertions to check if the returned value matches the expected results
- Verify for other test cases: continue writing methods for other scenarios

How to write Unit tests
We will use the following example class

```
public class Counter {
      int count = 0;
      public int increment () {
            return ++count;
      }
      public int decrement() {
          return --count;
      }
}
```

```
public class CounterTest {
   public CounterTest() {   }
   @BeforeAll
   public static void setUpClass() {   }

   @AfterAll
   public static void tearDownClass() {   }

   @BeforeEach
   public void setUp() {   }

   @AfterEach
   public void tearDown() {   }

   /**
    * Test of increment method, of class Counter.
    */
   @Test
   public void testIncrement() {
      System.out.println("increment");
      Counter instance = new Counter();
      int expResult = 1;
      int result = instance.increment();
      assertEquals(expResult, result);}

   /**
    * Test of decrement method, of class Counter.
    */
   @Test
   public void testDecrement() {
      System.out.println("decrement");
      Counter instance = new Counter();
      int expResult = -1;
      int result = instance.decrement();
      assertEquals(expResult, result);}}
```

public void setUp()

Creates a test by creating and initializing objects and values, for example, open a network connection.

public void tearDown()

Releases any system resources used by the test, for example, close a network connection.

public void testIncrement()
public void testDecrement()

These methods contain tests for the Counter methods *increment()*, *decrement()*.

@BeforeAll and **@AfterAll** – Methods with these annotations run once per class

@BeforeEach and **@AfterEach** – Methods with these annotations run once per test

Different ways of conducting the test

- Within a test
  - Call the method being tested and get the actual result
  - Assert what the correct result should be with one of the provided assert methods and fail method
- Types of testing
  - static void assertTrue(boolean test): Asserts that a condition is true
  - static void assertFalse(boolean test): Asserts that a condition is false
  - assertEquals(expected, actual): This method is heavily overloaded. Expected and actual must be both objects or both of the same primitive type
  - fail()
  - fail(String failResponse)

public static void **assertEquals**(boolean expected, boolean actual)
Asserts that two booleans are equal.

public static void **assertEquals**(Object expected, Object actual)
Asserts that two objects are equal. If they are not an AssertionFailedError is thrown.

public static void **assertEquals**(String expected, String actual)
Asserts that two Strings are equal.

public static void **assertEquals**(char expected, char actual)
Asserts that two chars are equal.

public static void **assertEquals**(int expected, int actual)
Asserts that two ints are equal.

void **assertArrayEquals**(expectedArray, resultArray);
The assertArrayEquals() method will test whether two arrays are equal to each other.

Fail methods

(1) `fail()`

public static void **fail**()
Fails a test with no message.

(2) **`fail(String failResponse)`**

It immediately cause the test method to fail and the **`failResponse`** string will be displayed.

```
public void testIncrement() {
if (counter1.increment() != 1) {
      fail("The test  is failed.");
}
}
```

Inheritance

- Put common codes in a superclass (parent class)
- Subclass (child class) inherits from the superclass, and reuses codes from superclass
- An inheritance relationship means a subclass inherits the members (both instance variables and methods) of a superclass
- A subclass can add new methods and variables of its own
- Subclass can override the methods it inherits from superclass
- You cannot extend from multiple classes in Java (no multiple inheritances)

IS-A relationship

- Inheritance is not just for reusing code
- A subclass should be a specific kind of superclass
- Superclass: Contains all the common instance variables and all the common methods with a default implementation

@Override

- The @Override part is optional, however, it is better to use it because then the Java compiler internally checks if you are actually Overriding a method. Easy to make spelling mistakes
- If the method names don't match, the base method won't be overridden
- It also makes the code more readable
- The method of the most specific class/object is called
- If a method is not overridden in a subclass, the superclass method is called
- If a method is overridden in a subclass, the subclass is called

Things to watch out for

- Only public and protected methods can be overridden
- You can have a private method with the exact same name and signature as a private method in the superclass, but you are not overriding the private method in the superclass, you are just declaring a new private method in the subclass
- The access level cannot be more restrictive than the overridden method access level
- Only subclasses have access to protected variables (make the instance variables protected)

Constructors and Inheritance

- When you instantiate a subclass, the default constructor of the superclass is implicitly called

- However, if you have a non-default constructor in the superclass, and no default constructor, it won't be implicitly called. You must call them explicitly (otherwise compile error)

The “super” keyword

- The super keyword is used to “call” the constructor of the superclass
- It must be the first statement in the constructor of the subclass
- Without an explicit call to the super() constructor, the default constructor is still called
- If only non-default constructor exists, the super() call is must. Otherwise, compile error
- You can use the super keyword to call superclass public or protected methods as well
- You can only go one level up with super, no such thing as super.super

Lecture 8

Inheritance and the “final” keyword

- A good way to protect methods
    - A class declared as final cannot have any subclasses
    - A “final” method cannot be overridden
    - You cannot extend from multiple classes in java (no multiple inheritance)

```
class A {
  protected int one;
  public A (int a) {
    one = a;
 }
 public void hi() {
    System.out.println ("Hi
" + one);
  }
}
```

```
class B extends A {
  protected int two;
  public B (int a, int b) {
    super (a);
    two = b;
 }
  public void hello() {
     super.hi();
    System.out.println("Hello "
+ one + " " + two);
  }
}
```

```
class C extends B {
   protected int three;
   public C (int a, int b, int c) {
      super (a, b);
      three = c;
   }
  public void salut() {
      super.hi();
      super.hello();
      System.out.println ("Salut " +
super.one + " " + super.two + " "
+ three);
    }
}
```

Output?

```
public class Tester {
   public static void main
(String[] args) {
      C c = new C (5, 6, 7);
      c.salut();
   }
}
```

Design of class inheritances

- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Use parent classes to represent general concepts that lower classes have in common
- Look for objects that have common states and behaviors
- Design a class that represents the common state and behavior
- Decide if a subclass needs behaviors that are specific to that particular subclass type
- Look for more opportunities to use abstraction, by finding two or more subclasses with common behavior

Abstract class

- We can declare a class as abstract to make sure no object of this class type can be instantiated
- An abstract class has no use, no value, unless it is extended
- Instances of a subclass of the abstract class are doing the real work at run time

Abstract methods

- An abstract method has no body
- An abstract method can only exist in an abstract class, they can't be private
- In a subclass, implementing an abstract method is just like overriding a method. But you must override it, otherwise error

Abstract class - constructors

- The constructor behavior of superclasses still prevails for abstract classes:
    - “Super” keyword to call constructor, the first statement in the subclass constructor
    - If no explicit call, the default constructor is called implicitly

Interface

- Interfaces are not classes, although they look very similar, it is a new reference type
- All public methods in the interface must be abstract
- A class that implements an interface must implement all its abstract methods
- An interface cannot implement any methods, whereas an abstract class can
- A class can implement many interfaces but can have only one superclass
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface. An interface defines a protocol of behavior that can be implemented by any class

Interface restrictions

- No constructor
- Any instance variable in an interface is constant. Ex: implicitly public, static, and final, whether you declare it as so or not
- All methods are implicitly public, abstract
- Must initialize the variable when you declare it, otherwise compile the error
- There is no restriction on how many interfaces one class can implement, because interfaces are not part of the class hierarchy

Two interfaces with the same method name and signature?
Doesn't matter, you can just implement one, no conflict
since interface methods don't have a body

```
public interface A{
public abstract void m();
}
```

```
public interface B{
public abstract void m();
}
```

```
public class C implements A,B{
 public void m(){}
 }
```

Interface + abstract class

- You can declare a class as an abstract is you want to implement an interface but leave it up to the subclasses to actually implement the method

Extending an interface

- An interface can extend another interface, just like a class can extend another class

Interfaces in the Java standard class library

- The Java standard class library contains many helpful interfaces
- The comparable interface contains one abstract method called compareTo, which is used to compare two objects

interfaces are probably used for testing

The Comparable Interface

The Comparable Interface

Method Summary

int	**compareTo**(Object o) Compares this object with the specified object for order.

Method Detail

compareTo

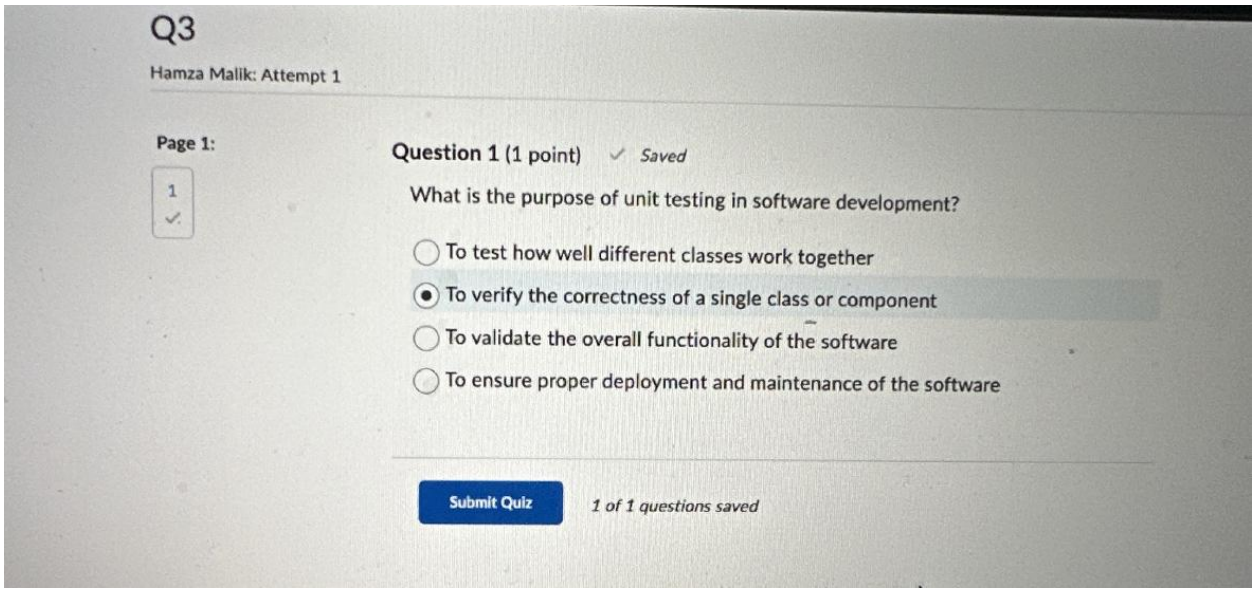
public int **compareTo**(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

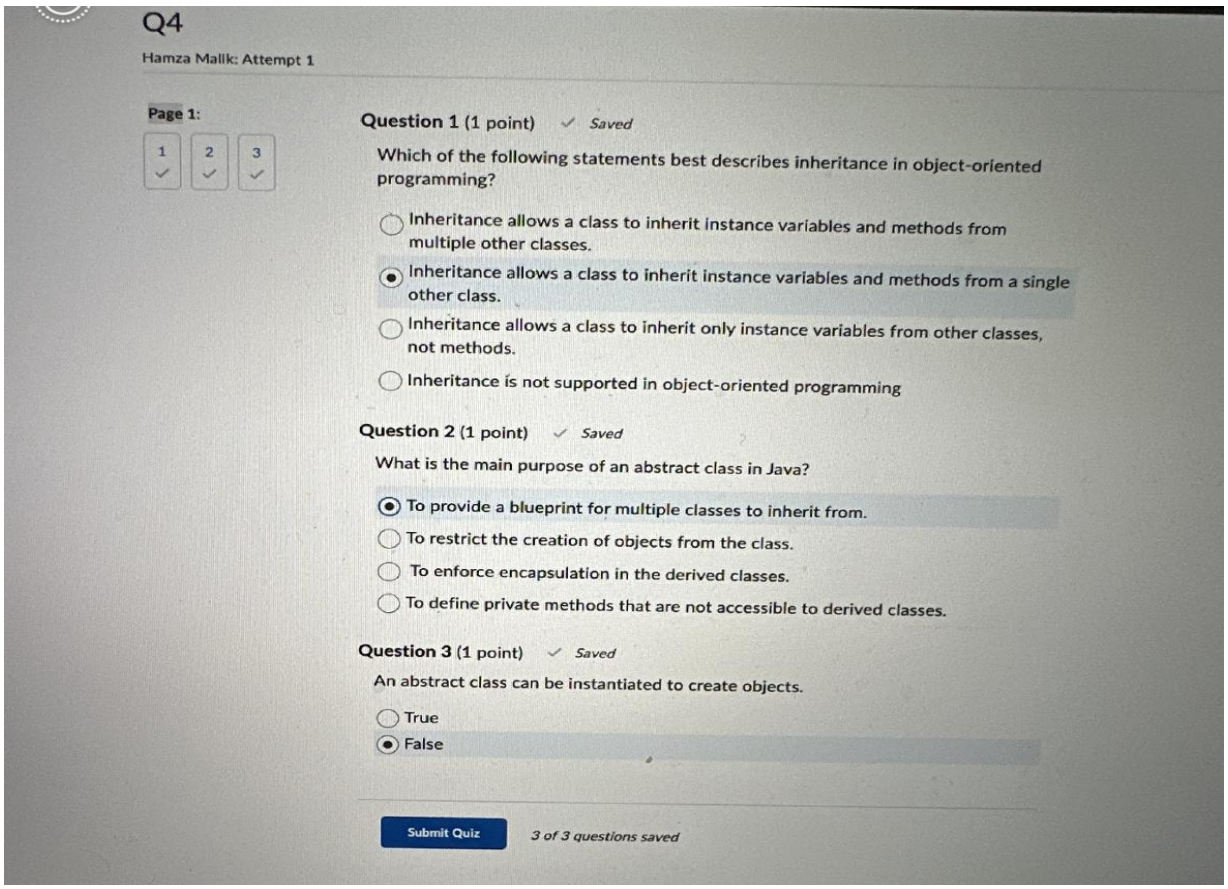
The comparable interface

- Any class can implement comparable to provide a mechanism for comparing obnjects of that type
- The value returned from compareTo should be:
    - Negative if obj1 < obj2
    - 0 if they are equal
    - And positive if obj1 > obj2
- It's up to the programmer to determine what makes one object lesser (greater) than the other

Quiz 3


Q3
Hamza Malik: Attempt 1
Page 1:
1
Question 1 (1 point) Saved
What is the purpose of unit testing in software development?
To test how well different classes work together
To verify the correctness of a single class or component
To validate the overall functionality of the software
To ensure proper deployment and maintenance of the software
Submit Quiz
1 of 1 questions saved

Quiz 4


Q4
Hamza Malik: Attempt 1
Page 1:
1
2
3
Question 1 (1 point) Saved
Which of the following statements best describes inheritance in object-oriented programming?
Inheritance allows a class to inherit instance variables and methods from multiple other classes.
Inheritance allows a class to inherit instance variables and methods from a single other class.
Inheritance allows a class to inherit only instance variables from other classes, not methods.
Inheritance is not supported in object-oriented programming
Question 2 (1 point) Saved
What is the main purpose of an abstract class in Java?
To provide a blueprint for multiple classes to inherit from.
To restrict the creation of objects from the class.
To enforce encapsulation in the derived classes.
To define private methods that are not accessible to derived classes.
Question 3 (1 point) Saved
An abstract class can be instantiated to create objects.
True
False
Submit Quiz
3 of 3 questions saved

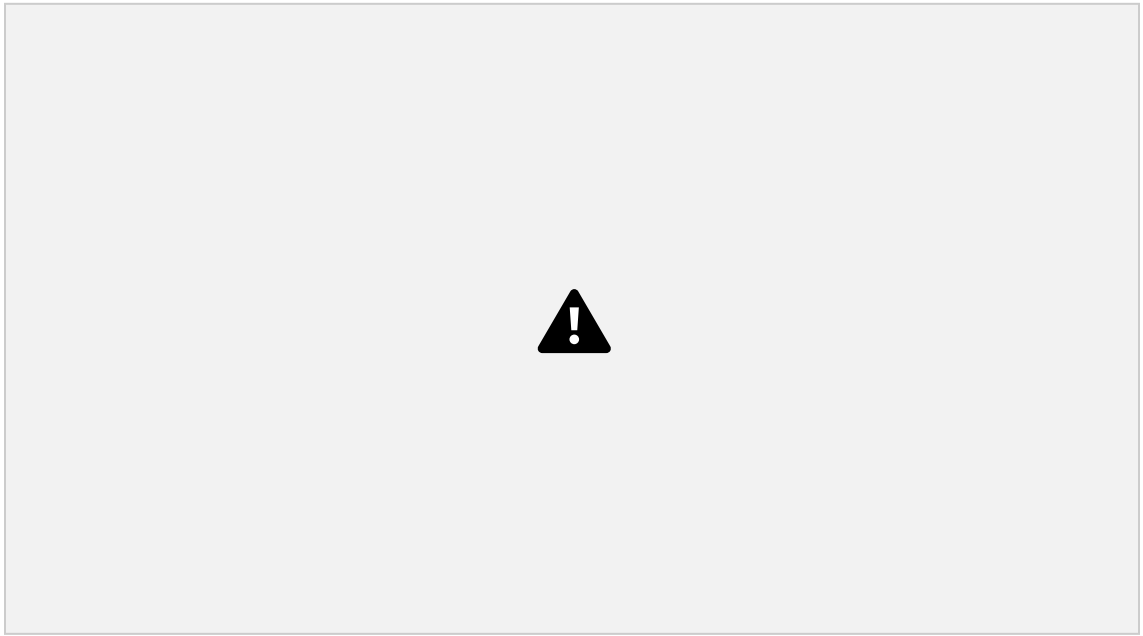
Lecture 9

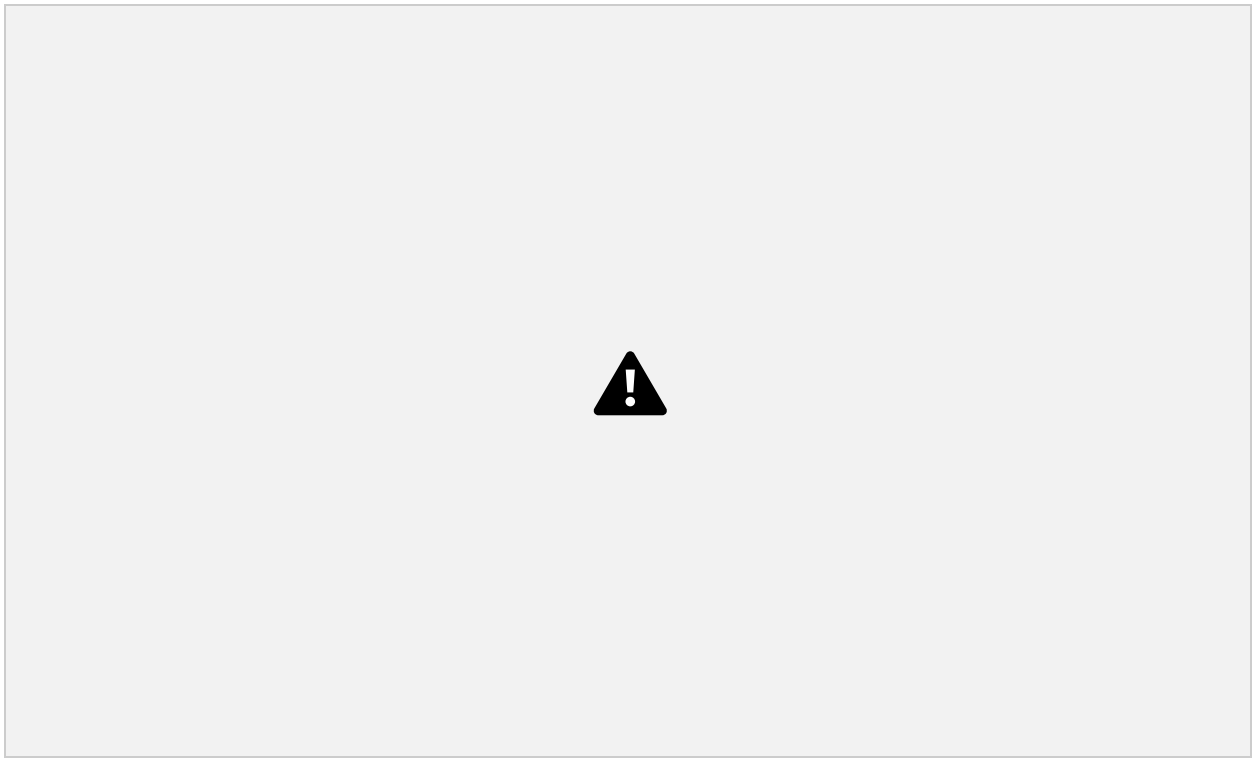
Polymorphism

- Polymorphism means having many forms
- Java actually decides which method to call based on the current type of object it is being called from. It is decided during runtime, not during compilation
- Type can be changed with casting/changing references of the object
- This process is called late binding
- When you call one of the parent class methods (that have been overridden in the child classes), it is the type of the object being referenced, not the reference type, that determines which method is invoked

Polymorphism and Type Casting objects

- When we cast a reference along the class hierarchy in a direction from the subclasses toward the root, it is an upcast. We need not use an explicit cast in this case
- Upcasting is perfectly fine with polymorphism
- When we cast a reference along the class hierarchy in a direction from the root class towards the children or subclasses, it is a downcast
- Downcast is not valid (compiles, but runtime error)
- Downcast only works if it can succeed during runtime





- Also remember, instance variables do not get overridden. Having an instance variable with the same name in a subclass simply redefines it, and does not override.

Late binding with interfaces

- You can also use an interface as a reference type, and instantiate an object of a concrete class that implements the interface
- However, by doing so, you can only call methods that exist in the interface definition

Object-Oriented Design

- Design is the process of developing a software solution from a customer specification
- Many steps in the design process:
    - Requirements analysis
    - Use case/user stories
    - Flow charts
    - Specific diagrams for software development such as class diagrams, sequence diagram

Unified Modeling Language (UML)

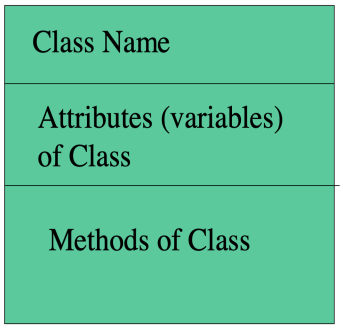
- Standardized general-purpose modeling language used to specify, visualize, construct, and document the design of an object-oriented system

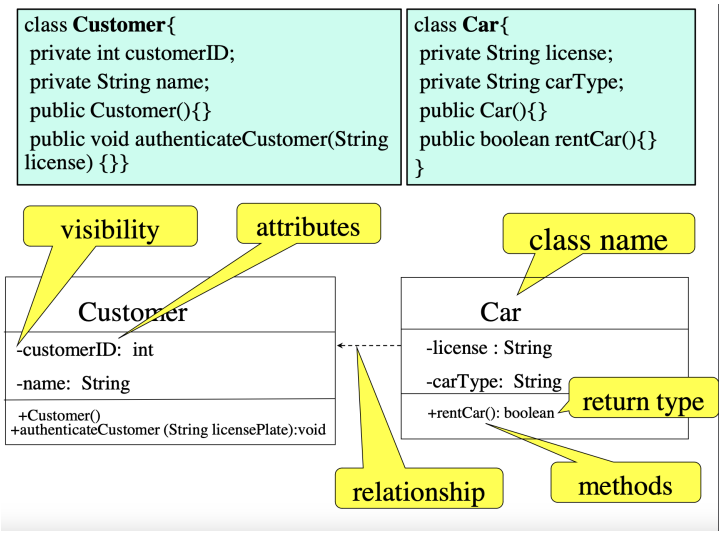
- Used to be a very strict standard that all software development processes have to adhere to. Nowadays a little more relaxed because of agile development principles

Class diagram

- Describe the structure of the system in terms of classes and objects
- The primary purpose is to create a vocabulary that is used by developers, analysts, and users

UML Representation of Class



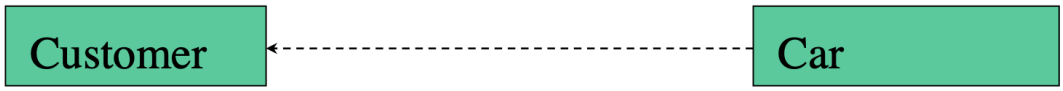


Visibility of Attributes and Operations

Visibility	Symbol	Accessible To
Public	+	All objects within your system.
Protected	#	Instances of the implementing class and its subclasses.
Private	-	Instances of the implementing class.

Relationships among classes

- Represents a connection between multiple classes or a class and itself
- 3 categories
    - Dependency relationships
        - A dependency exists when one class relies on another in some way usually by invoking the methods of the other
        - The dotted line with an open arrow



Customer Car

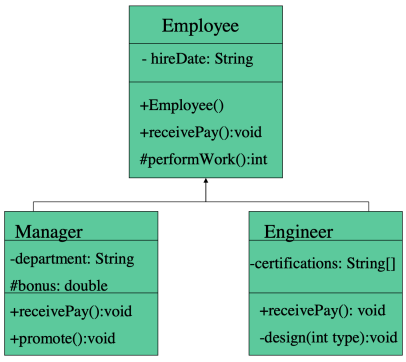
    - Generalized relationships
        - Shows subclasses and superclasses' relation
        - Represented by IS-A relationship
        - Solid line with a closed arrow

Generalization Relationship

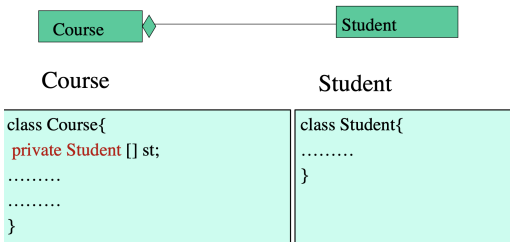
```
class Employee{
 private String hireDate;
 public Employee(){}
 public void receivePay(){}
 protected int performWork(){}
}
```

```
class Manager extends Employee{
 private String department;
 protected double bonus;

 public void receivePay(){}
 public void promote(){}}
```

```
class Engineer extends Employee{
 private String[] certifications;
 protected double bonus;

 public void receivePay(){}
 private void design(int type){}}
```



    - Aggregation relationships
        - Specialized relationship in which a whole is related to its part(s)
        - Represented by a-part-of relationship
        - Denoted by placing a diamond nearest the class representing the aggregation



- Interfaces
  - The dotted line with a closed arrow

Deriving class diagrams from scenarios (text description)

- Analyze the text in the description
- Identify classes and their relationships
- Draw the diagram
- There could be multiple answers, the target is to be as precise as possible

Class Diagram

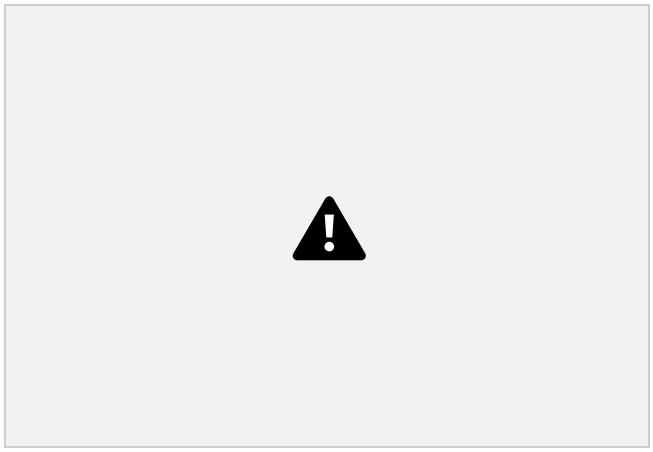
- Ensure that the classes are both necessary and sufficient to solace the underlying problem
  - No missing attributes or methods in each class
  - No extra or unused attributes or methods in each class
  - No missing or extra classes

Discarding unnecessary and incorrect classes

- Redundant classes
- Irrelevant classes
- Vague classes
- Attributes
- Operations
- Roles
- Implementation constructs

Types of classes

- One found during the analysis
  - people, places, events, and things about which the system will capture information
  - Ones found in the application domain
- One found during the design
  - Specific objects like windows and forms are used to build the system



*** Lecture 9, Slides 46 - 56 example***

Lecture 10

Exception

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
    - The Java language uses exceptions to provide error-handling capabilities for its programs
    - Exceptions are represented as classes in Java
- A program can deal with an exception in one of three ways
    - Ignore it
        - The program will terminate and produce an appropriate message
    - Handle it where it occurs
        - The exception is represented as an object in Java, They are thrown by a program and may be caught and handled by another part of the program
        - The try-and-catch statement
            - To process an exception when it occurs, the line that throws the exception is executed within a try block
            - A try block is followed by one or more catch clauses, which contain code to process an exception
            - Each catch clause has an associated exception type. When an exception occurs, processing continues at the first catch clause that matches the exception type

```java
try {
    // Code that might generate exceptions
} catch (Type1 id1) {
    // Handle exceptions of Type1
} catch (Type2 id2) {
    // Handle exceptions of Type2
} catch (Type3 id3) {
    // Handle exceptions of Type3
}
// etc...
```

- Handle it in another place in the program

Exception handling

- Exception handling is very important in software development to make programs foolproof. Instead of the program crashing, you can show a readable message
- All Java exceptions are part of the hierarchy
- You can catch “Exception” for all kinds of exceptions

```java
try {
    // Code that might generate exceptions
} catch (SpecificExceptionType exception) {
    // Handle specific exception
} catch (AnotherSpecificExceptionType exception) {
    // Handle another specific exception
} catch (GeneralExceptionType exception) {
    // Handle general exception
}
```

- If you want to catch a specific exceptional you can first look at the type of exception being thrown, then write the code for that

The finally block

- A try statement can have an optional clause designated by the reserved word finally
- A final block is where you put code that must run regardless of an exception (whether you catch it or not)

```java
try {
    // Code that might generate exceptions
} catch (ExceptionType exception) {
    // Handle exception
} finally {
    // Code that must run regardless of an exception
}
```

- The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that the resource is always recovered

"Throwing" an exception

- Sometimes, you do not want to handle an exception within a method. You would rather "throw" it to whoever called the method so that its someones else's problem
- Makes code cleaner, all the exception handling can be in the main() method
- A method can throw an exception with the "throws" keyword

Checked and unchecked exceptions

- Checked exceptions
    - Subclasses of exception, an exception that is checked at compile time
    - Many exceptions, including IOException and most exceptions you will declare, are checked
- When a method calls another method that can throw a checked exception, it has just two choices:
    - The exception must be declared in the header of the method, using a throws clause

```java
public void readFile() throws IOException {
    // Code that reads a file
}
```

- Or the code that might cause the exception to be thrown must be inside a try block with a catch clause for that exception. Otherwise, there will be a compilation error

```java
public void readFile() {
    try {
        // Code that reads a file
    } catch (IOException e) {
        // Handle the IOException
    }
}
```

- Unchecked exceptions
  - Are subclasses of RunTimeException
  - Unchecked at compile time and detected only when the code is executed at runtime
  - Runtime exceptions represent problems that the programmer cannot reasonably be expected to recover from them or handle them in any way
  - Java does not have to catch or specify runtime exceptions
  - You don’t need a throws clause to throw a RunTimeException
  - RunTimeExceptions also do not need to be caught, the program will compile fine

Throw and Catch Exception:

Throwing and catching exceptions is a way to deal with errors or unexpected situations that can happen during a program's execution, allowing you to handle them appropriately.

Throwing a Checked Exception (must catch)

```
class Age {
    static void throwingDemo() throws IllegalAccessException {
        throw new IllegalAccessException("Just a demo");
    }

    public static void main(String[] args) {
        try {
            throwingDemo();
        } catch (IllegalAccessException e) {
            System.out.println(e.toString());
        }
    }
}
```

Runtime Exception:

A runtime exception is an unexpected error that occurs during the execution of a program, indicating a problem that prevents normal program flow.

Throwing a RuntimeException (no need to catch)

```
class Age {
    static void throwingDemo() {
        // "throws" clause is optional
        throw new RuntimeException("Just a demo");
    }

    public static void main(String[] args) {
        throwingDemo(); // try-catch not required,
        //although there will be a runtime error
    }
}
```

Exception toString()

- The toString() method in exception and its subclasses can be used to communicate an appropriate message.

Defining your own exceptions

- You can define your own exception, done commonly in software development

Exception Hierarchy

- Notice that the custom exception is also extended from the exception superclass
- We can create a further hierarchy of exception

```
class RideRestrictionException extends Exception {
    public RideRestrictionException(String message) {
        super(message);
    }
}
```

```
class RideChecker {
    public void checkRestriction(int age, double height) throws
    RideRestrictionException {
        if (age < 6 || height < 3.2)
            throw new RideRestrictionException("Age or height is too low");
    }

    public static void main(String[] args) {
        RideChecker c = new RideChecker();
        try {
            c.checkRestriction(3, 5.0);
        } catch (RideRestrictionException r) {
            System.out.println(r.toString());
        }
```

- Since exception classes are subclasses of each other, be careful when catching multiple types of exception
- Always catch the most specific exception first, then go up to the superclass
- If you go the other way, the superclass will catch everything

Exception Rules

- You cannot have a catch or finally without a try

```
void go() {
    Foo f = new Foo();
        f.foof();
  catch(FooException ex) { }
```

- You cannot put code between the try and catch

```
try {
      x.doStuff();
     }
  int y = 43;
} catch(Exception ex) { }
}
```

- A try must be followed by either a catch or a finally

```
try {
      x. doStuff ();
} finally {//cleanup}
```

- A try with only a finally (no catch) must still throw the exception

```
void go() throws FooException {
 try{  x.doStuff();
} finally { } }
```

Exception: Handle it in another place in the program

- If it is not appropriate to handle the exception where it occurs, it can be handled at a higher level
- Exception propagate up through the method calling hierarchy until they are caught and handled or until they reach the outermost level

```
public class Propagation {
        static public void main (String[] args)         {
                ExceptionScope demo = new ExceptionScope();
                demo.level1();
        }
public class ExceptionScope{
        public void level1(){
                try {  level2();
                }catch (ArithmeticException problem){
                        System.out.println ("The exception message");
                }}
        public void level2(){
                level3 ();     }
        public void level3 (){
                int numerator = 10, denominator = 0;
                int result = numerator / denominator;
    }
}
```

output

↳ Arithmetic Exception

Inner try blocks

- You can have inner try blocks for specialized exceptions, and reserve the outermost try block for the exception handling to use if all the previous attempts fail

---
