

Java is case sensitive and is platform independent

Package name convention: all lower case ,If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

Class ,interface name convention: always start with upper case

Variable = start with lower case , place where data is stored,If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName,requiredSalaryStub

Constant= always uppercase letter,

method= start with lowercase letter,If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

compilation error is known as syntax errors or declaration errors in the code and run time error is known as logic error i.e error in input

Comment = // or /* */

Main method = public static void main(String args[])

To print : System.out.println()

To get input from user = Scanner in(any name) =new Scanner(System.in); Also import scanner class.

int a(variable name)=in.nextInt();

scanner.close();

Programming convention:

- Initialize what we know
- Get what we don't know
- Check the condition
- Perform the process

if() - contains a condition, This condition must resolve to a boolean value.

If else() -If a certain situation occurs, do something. Otherwise, do something else.

Nested If Statements- If a certain situation occurs, check for the next situation.

This else statement goes with the 2nd if statement. Even though there's two if statements here, the else will go with the immediate if that it follows.

```
import java.util.Scanner;

/*
 * NESTED IFS:
 * To qualify for a loan, a person must make at least $30,000
 * and have been working at their current job for at least 2 years.
 */
public class LoanQualifier {

    public static void main(String args[]){

        //Initialize what we know
        int requiredSalary = 30000;
        int requiredYearsEmployed = 2;

        //Get what we don't
        System.out.println("Enter your salary:");
        Scanner scanner = new Scanner(System.in);
        double salary = scanner.nextDouble();

        System.out.println("Enter the number of years with your current employer:");
        double years = scanner.nextDouble();

        scanner.close();

        //Make decision
        if(salary >= requiredSalary){
            if(years >= requiredYearsEmployed){
                System.out.println("Congrats! You qualify for the loan");
            }
            else{
                System.out.println("Sorry, you must have worked at your current job "
                    + requiredYearsEmployed + " years.");
            }
        }
        else{
            System.out.println("Sorry, you must earn at least $"
                + requiredSalary + " to qualify for the loan");
        }
    }
}
```

The If-Else-If _Statement

If situation A occurs, do something.

Else if situation B occurs, do something else.

Else if situation C occurs, do something else.

The if-else-if statement is used when there are more than two possible paths.

Character Data Type

The char data type holds exactly one character. So, it's similar to a String, meaning it holds some type of text. However, it's only one character of text. char data types use the single quotation marks; double quotation marks are for String data type, and single is for char.

```
char grade;

    if(score < 60){
        grade = 'F';
    }
    else if(score < 70){
        grade = 'D';
    }
    else if(score < 80){
        grade = 'C';
    }
    else if(score < 90){
        grade = 'B';
    }
    else{
        grade = 'A';
    }

    System.out.println("Your grade is " + grade);
```

if-else-if, only one of these will be invoked – the first one that is true

Once this is done, it will not even evaluate the rest of these cases. It will go on to the rest of the program same with switch statement

The Switch Statement

If situation A occurs, do something.

Else if situation B occurs, do something else.

Else if situation C occurs, do something else.

The switch statement solves a problem in the same way that the if-else-if does. So, it's ideal for cases when you have more than two paths.

The difference between the if-else-if and the switch statements is that the if-else-if checks the condition to be true; whereas the switch statement checks for equality.

the `Scanner` does not have a method to get just a character. The closest we have is a `String` which is also used for text. Even though it's going to be only one letter we can still hold it in a `String` variable.

So, let's use `next` as there is no `nextString`. The `next` method by default, we see that it returns to `String`.

it's a good practice to also have a `default` case that handles anything else.

```
switch(grade){
    case "A":
        message = "Excellent job!";
        break;
    case "B":
        message = "Great job!";
        break;
    case "C":
        message = "Good job!";
        break;
    case "D":
        message = "You need to work a bit harder";
        break;
    case "F":
        message = "Uh oh!";
        Break;
    default:
        message = "Error. Invalid grade";
        break;
}
```

Let's say that we had some cases where multiple conditions evaluated to true. Only the first one that is evaluated to be true will be executed. It would never look at the rest of

the cases, because these are essentially *elses*. If this is not true, then consider these.

If it finds one of the cases to be true, it will not evaluate the following ones.

However, the *switch* statement allows what's called a *fall through*.

Fall throughs occur when you eliminate the *break* statement at the end of a case.

```
switch(grade){  
    case "A":  
        message = "Excellent job!";  
    case "B":  
        message = "Great job!";  
        break;  
}
```

Let's say the value of grade was A. It would execute case A, however, because case A does not have a break statement, it would continue on to execute case B, even though grade doesn't even equal B! In fact, it will keep going through all the cases until it reaches a break statement.

Operators:

Relational Operators

> greater than
< less than
>= greater than or equal
<= less than or equal
== equals
!= not equal

There will be cases where you want to compare Strings.

You may think to use the == or != for this, however that doesn't necessarily do what you think it does. These two operators will compare the memory locations of both of the Strings, not the values themselves.

If you want to check to see if a String's value is equal to another String's value, you need to use the equals method of the String.

To check if two String values are not equal, you add the not symbol (!) at the beginning of the condition.

If you don't care about the case of the Strings, then you can use the equalsIgnoreCase

Examples:

```
if(string1.equals(string2))  
if(!string1.equals(string2))  
if(string1.equalsIgnoreCase(string2))
```

Logical Operators:

Logical operators are used to combine two separate conditions in order to get one resulting boolean value.

Let's look at the first one. It's two ampersands && and this means "AND". In order for this boolean value to result to true, both of the conditions must be true.

Next, we have the "OR" operator ||. When this combines two conditions, at least one of those conditions must be true in order for the entire condition to be true.

Then finally the "NOT" operator !. The condition itself must be false in order for the resulting condition to be true.

While Loop

- While
- Do While
- For

Now it's up to you as the programmer to make sure that your loops do not run infinitely.

The way you do that is by using, like we've done here, a variable that will be updated at some point within the loop. This is called a **sentinel**.

Sentinels

A sentinel is a variable used within the condition that controls the loop.. It's very important that somewhere in your loops there's an opportunity for the sentinel to be updated. Otherwise, this loop will run infinitely.

```
//Validate input
while(hoursWorked > maxHours) {
    System.out.println("Invalid entry. Your hours must be between 1 and 40. Try again.");
    hoursWorked = scanner.nextDouble();
}
double gross = rate * hoursWorked;
System.out.println("Gross pay: $" + gross)
```

This will allow them a second chance to enter a value for how many hours they've worked. Once they've entered this, it will exit this loop, come back to the top while(hoursWorked > maxHours) and it will test this condition again.

So, hoursWorked has been updated, now its new value will be compared with the maxHours.

- It's controlled by a condition and will continue to run while that condition remains true.
- The condition is tested before the loop is entered, so there's a chance that this loop will never execute.
- It's good to use the While loop when you may or may not need to run the loop, based on the state of the condition.

Steps of while loop

1. First the condition is evaluated to determine if the loop needs to be executed.
2. If that's true, the statements inside of the loop are executed.
3. Then the condition is rechecked to determine if to run the loop again.

Remember to update the sentinel inside of your loop.

Do while loop:

We're going to run this at least one time, and that's the purpose of the *do while* loop.

```
do{  
    } while(/*condition here*/);
```

- Just like the While Loop, it's also controlled by a condition.
- That condition is tested after the completion of the loop. So, the loop will always run at least once.
- So, it's good to use the *do while* loop when the loop should run at least one time, and based on the situation, they now need to repeat.

Steps of do while loop:

1. First the statements inside of the loop are executed. Again, be sure to update the sentinel here.
2. Then the condition is checked to determine if to run the loop again.

```

public class AddNumbers {

    public static void main(String args[]){

        Scanner scanner = new Scanner(System.in);

        boolean again;

        do{
            System.out.println("Enter the first number");
            double num1 = scanner.nextDouble();

            System.out.println("Enter the second number");
            double num2 = scanner.nextDouble();

            double sum = num1 + num2;

            System.out.println("The sum is " + sum);

            System.out.println("Would you like to start over?");
            again = scanner.nextBoolean();

        } while(again);

        scanner.close();
    }
}

```

For loops:

So far, the loops we've reviewed have been condition controlled.

There's one more loop, the *For Loop*, which is *count-controlled* meaning it loops a given number of times.

```

for(int i=0; i<quantity; i++){

}

```

- It's count-controlled, meaning it will run a specified number of times.
- The sentinel is expressed within a condition which is tested before the loop is entered.

- It's best to use when you know how many times the loop should be executed.

Steps of the For Loop

1. First, the sentinel is initialized to a starting value and the ending value is also specified.
2. Then the statements inside of the loop are executed
3. And the condition is rechecked to determine if to run the loop again.

```
public class Cashier {  
  
    public static void main(String args[]){  
  
        //Get number of items to scan  
        System.out.println("Enter the number of items you would you like to scan:");  
        Scanner scanner = new Scanner(System.in);  
        int quantity = scanner.nextInt();  
  
        double total = 0;  
  
        //Create loop to iterate through all of the items and accumulate the costs  
        for(int i=0; i<quantity; i++){  
  
            System.out.println("Enter the cost of the item:");  
            double price = scanner.nextDouble();  
  
            total = total + price;  
        }  
  
        scanner.close();  
  
        System.out.println("Your total is $" + total);  
    }  
}
```

Break statement

Sometimes you may need to get out of a loop regardless of what the condition is, and we can use the break statement for this.

We're going to create a program that searches a String to determine if it contains the letter A.

```

public class LetterSearch {

    public static void main(String args[]){

        //Get text
        System.out.println("Enter some text:");
        Scanner scanner = new Scanner(System.in);
        String text = scanner.next();
        scanner.close();

        boolean letterFound = false;

        //Search text for letter A
        for(int i=0; i<text.length(); i++){
            char currentLetter = text.charAt(i);
            if(currentLetter == 'A' || currentLetter == 'a'){
                letterFound = true;
                break;
            }
        }

        if(letterFound){
            System.out.println("This text contains the letter 'A'");
        }
        else{
            System.out.println("This text does not contain the letter 'A'");
        }
    }
}

```

Methods

Access Modifier : public,private,protected

Having an access modifier is not required. If one is not specified, then it means that this method is only accessible by classes within the same package.

Non access modifier : static,final,abstract,synchronized

```

public static void greetUser(String name){

}

```

The methods need to be called in the main method

```
public class Greetings {

    public static void main(String args[]){

        System.out.println("Enter your name:");
        Scanner scanner = new Scanner(System.in);
        String name = scanner.next();
        greetUser(name);
    }

    public static void greetUser(String name){

        System.out.println("Hi there, " + name);
    }
}

public class InstantCreditCheck {

    static int requiredSalary = 25000;
    static int requiredCreditScore = 700;
    static Scanner scanner = new Scanner(System.in);

    public static void main(String args[]){

        double salary = getSalary();
        int creditScore = getCreditScore();
        scanner.close();

        //Check if the user is qualified
        boolean qualified = isUserQualified(creditScore, salary);

        //Notify the user
        notifyUser(qualified);
    }

    public static double getSalary(){
        System.out.println("Enter your salary:");
        double salary = scanner.nextDouble();
        return salary;
    }

    public static int getCreditScore(){
        System.out.println("Enter your credit score:");
        int creditScore = scanner.nextInt();
        return creditScore;
    }
}
```

```

    }

    public static boolean isUserQualified(int creditScore, double salary){
        if(creditScore >= requiredCreditScore && salary >= requiredSalary){
            return true;
        }
        else{
            return false;
        }
    }
}

    public static void notifyUser(boolean isQualified){
        if(isQualified){
            System.out.println("Congrats! You've been approved.");
        }
        else{
            System.out.println("Sorry. You've been declined");
        }
    }
}

```

Access modifiers

public is an access modifier, meaning that it is used to set the level of access. You can use access modifiers for classes, attributes, and methods.

For classes, the available modifiers are public or default (left blank), as described below:

public: The class is accessible by any other class.

default: The class is accessible only by classes in the same package.

The following choices are available for attributes and methods:

default: A variable or method declared with no access control modifier is available to any other class in the same package.

public: Accessible from any other class.

protected: Provides the same access as the default access modifier, with the addition that subclasses can access protected methods and variables of the superclass (Subclasses and superclasses are covered in upcoming lessons).

private: Accessible only within the declared class itself.

Getters and setters:

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as **private** (only accessible within the same class)
- provide public **setter** and **getter** methods to access and update the value of a **private** variable

For example:

```
Class Example{  
    int x;  
}
```

x is public. You can change x directly.

```
Example ex = new Example();
```

```
ex.x = 4;
```

```
Class Example{  
    private int x;  
    //setter --> setX  
    //getter --> getX  
}
```

x is private. Now it is not possible to change x directly. You have to use setX (or getX to get the value of x).

Encapsulation is one part of OOP. You hide your variables and with setters and getters you get more control.

Example:

```
class Clock {  
  
    String time;  
  
    void setTime (String t) {  
  
        time = t; }  
  
    String getTime() {
```

```

return time; } }

class ClockTestDrive {

public static void main (String [] args) {

    Clock c = new Clock;

    c.setTime("12345") String tod = c.getTime();

    System.out.println(time: " + tod); } }

```

Example:

```

public class myClass{
public static void main(String args[]) {
myClass b=new myClass();
        b.seta(2);
        System.out.println(b.geta());

}}
class myClass{

private int a;

void seta(int a) {
        this.a=a;
    }
int geta() {
        return a;
    }
}

```

Why Encapsulation?

- Better control of class attributes and methods
- Class variables can be made **read-only** (if you omit the **set** method), or **write-only** (if you omit the **get** method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Passing Object as Parameter

```
class Rectangle {
    int length;
    int width;

    Rectangle(int l, int b) {
        length = l;
        width = b;
    }

    void area(Rectangle r1) {
        int areaOfRectangle = r1.length * r1.width;
        System.out.println("Area of Rectangle : "
                           + areaOfRectangle);
    }
}

class RectangleDemo {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(10, 20);
        r1.area(r1);
    }
}
```

1. We can pass Object of any class as parameter to a method in java.

Java returning Objects :

```
class Test
{
    int a;

    Test(int i)
    {
        a = i;
    }

    Test incrByTen()
```

```

    {
        Test temp = new Test(a+10);
        return temp;
    }
}

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

        Test obj1 = new Test(2);
        Test obj2;

        obj2 = obj1.incrByTen();

        System.out.println("obj1.a : " + obj1.a);
        System.out.println("obj2.a : " + obj2.a);

        obj2 = obj2.incrByTen();

        System.out.println("obj2.a after second increase : " + obj2.a);

    }
}

```

Constructors:

Constructors are special methods invoked when an object is created and are used to initialize them.

A **constructor** can be used to provide initial values for object attributes.

- A **constructor** name must be same as its class name.

- A **constructor** must have no explicit return type.

A **constructor** can also take parameters to initialize attributes.

```

public class Vehicle {
    private String color;
    Vehicle(String c) {

```



```
    color = c;
}
}
```

The **constructor** is called when you create an object using the **new** keyword.

```
public class MyClass {
    public static void main(String[] args) {
        Vehicle v = new Vehicle("Blue");
    }
}
```

A single class can have multiple constructors with different numbers of parameters. we can use the constructors to create objects of our class.

```
public class Vehicle {
    private String color;
```

```
    Vehicle() {
        this.setColor("Red");
    }
```

```
    Vehicle(String c) {
        this.setColor(c);
    }
```

```
//color will be "Red"
```

```
Vehicle v1 = new Vehicle();
```

```
//color will be "Green"
```

```
Vehicle v2 = new Vehicle("Green");
```

Java automatically provides a default constructor, so all classes have a constructor, whether one is specifically defined or not.

Static:

we can call methods that are *static* from another class without instantiating them. When I have *static* methods within a class, there is no need to instantiate that class in order to access them. I can access them simply by using the class name and the dot operator.

When you declare a variable or a *method* as *static*, it belongs to the class, rather than to a specific *instance*. This means that only one *instance* of a *static* member exists, even if you create multiple objects of the class, or if you don't create any. It will be shared by all objects.

The **COUNT** variable will be shared by all objects of that class.

Example:

```
public class Counter {  
    public static int COUNT=0;  
    Counter() {  
        COUNT++;  
    }  
}  
  
public class MyClass {  
    public static void main(String[] args) {  
        Counter c1 = new Counter();  
        Counter c2 = new Counter();  
        System.out.println(Counter.COUNT);  
    }  
}  
//Outputs "2"
```

You can also access the *static* variable using any object of that class, such as **c1.COUNT**.

It's a common practice to use upper case when naming a static variable, although not mandatory.

Example:

```

public class Vehicle {
    public static void horn() {
        System.out.println("Beep");
    }
}

public class MyClass {
    public static void main(String[] args) {
        Vehicle.horn();
    }
}

```

Method Overloading:

Within a class, you can have multiple methods that have the same name but different signatures. And this is known as overloading.

Example:

```

public class Month {

    public static String getMonth(int month){
        switch(month){
            case 1: return "January";
            case 2: return "February";
            case 3: return "March";
            case 4: return "April";
            case 5: return "May";
            case 6: return "June";
            case 7: return "July";
            case 8: return "August";
            case 9: return "September";
            case 10: return "October";
            case 11: return "November";
            case 12: return "December";
            default: return "Invalid month. Please enter a value between 1 and 12.";
        }
    }

    public static int getMonth(String month){
        switch(month){
            case "January": return 1;
            case "February": return 2;
            case "March": return 3;

```

```

        case "April": return 4;
        case "May": return 5;
        case "June": return 6;
        case "July": return 7;
        case "August": return 8;
        case "September": return 9;
        case "October": return 10;
        case "November": return 11;
        case "December": return 12;
        default: return -1;
    }
}

}

public class MonthConverter {

    public static void main(String args[]){
        System.out.println(Month.getMonth(2));
        System.out.println(Month.getMonth("January"));
    }
}

```

Arrays:

Arrays are special objects or containers which can hold multiple values but all same data type only.

Syntax:

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

```

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);

```

To change an array value refer to the index number.
cars[0]="Suzuki";

To find out how many elements an array has, use the `length` property:
System.out.println(cars.length)

Loop for an array:

```
for (int i = 0; i < cars.length; i++) {
    System.out.println(cars[i]);
}
```

There is also a "for-each" loop, which is used exclusively to loop through elements in arrays:

Syntax:

```
for (type variable : arrayname) {
    ...
}
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
    System.out.println(i);
}
```

Multi dimensional Arrays:

To access the elements of the myNumbers array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x);
```

We can also use a **for loop** inside another **for loop** to get the elements of a two-dimensional array (we still have to point to the two indexes):

```
public class MyClass {
    public static void main(String[] args) {
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
        for (int i = 0; i < myNumbers.length; ++i) {
            for(int j = 0; j < myNumbers[i].length; ++j) {
                System.out.println(myNumbers[i][j]);
            }
        }
    }
}
```

```
int[] lottoticket = new int[3];
```

```
lottoticket[0]=10;
```

```
lottoticket[1]=12;
```

```
lottoticket[2]=25;
```

OR

```
int[] lottoticket ={10,12,25}
```

- `int` - the first part is the data type that the array holds and while the array is capable of holding multiple values, all values must be of the same data type
- `[]` - the square brackets indicate that this is not just an `int` data type, but this will be an *array of "ints"*. The bracket can appear after the data type or after the variable name itself
- `lottoTicket` - speaking of name, that's the next part of the declaration. We name arrays just as we would any other variable
- `=` - the name is followed by an equal sign
- `new` - and then we use the `new` keyword
- `int` - followed by the data type again
- `[3]` - followed by a number inside of the brackets. This number inside of the brackets represents the length of the array, meaning how many values can this array store. Once the length is declared, it's fixed — meaning it cannot be lengthened to hold more values.

Notice that the indices begin with 0; and index 0 represents the first element of the array.

```

import java.util.Random;

public class LotteryTicket {

    private static final int LENGTH = 6;
    private static final int MAX_TICKET_NUMBER = 69;

    public static void main(String[] args){

        int[] ticket = generateNumbers();
        printTicket(ticket);
    }

    public static int[] generateNumbers(){

        int[] ticket = new int[LENGTH];
        Random random = new Random();

        for(int i=0; i< LENGTH; i++){
            ticket[i] = random.nextInt(MAX_TICKET_NUMBER) + 1;
        }

        return ticket;
    }

    public static void printTicket(int ticket[]){
        for(int i=0; i<LENGTH; i++){
            System.out.print(ticket[i] + " | ");
        }
    }
}

```

Final keyword

Use the final keyword to mark a variable constant meaning the value assigned to the variable should never change, so that it can be assigned only once.

Methods and classes can also be marked final. This serves to restrict methods so that they can't be overridden and classes so that they can't be subclassed.

These concepts will be covered in the next module.

```

class MyClass {
    public static final double PI = 3.14;
    public static void main(String[] args) {
        System.out.println(PI);
    }
}

```

```
}  
}
```

++i will pre - increment and i++ is post increment

- **++i will increment the value of i, and then return the incremented value.**
`i = 1; j = ++i; (i is 2, j is 2)`
- **i++ will increment the value of i, but return the original value that i held before being incremented.**
`i = 1; j = i++; (i is 2, j is 1)`

Examples:

```
public class poojaarray {
```

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

```
        int[] array1= {10,11,12}; //integer  
        String[] array2= {"pooja","sirisha","cyril"}; //string  
        //int array1[]=new int[3];
```

```
        System.out.println("Second name is " +array2[1]);//prints sirisha  
        array2[1]="prashanth";//changing array value  
        System.out.println("Second name is " +array2[1]);//prints prashanth
```

```
        for(int i=0;i<array2.length;i++)//printing a specific array  
        {  
            System.out.println("The contents of the array are" +array2[i]);  
        }
```

```
        int[][] arraymulti= {{10,11,58},{66,96,78}};//multi dimensional integer
```

array

```
        System.out.println("two dimen is " +arraymulti[0][2]);//printing a specific
```

array

```
        for(int i=0;i<arraymulti.length;i++) //printing all elements of integer array
```



```

    {
        for(int a=0;a<arraymulti[i].length;a++) {
            System.out.println("Print is" +arraymulti[i][a]);
        }
    }
    String[][] arraystr1= {"af","dg"},{"rrr"};//multi dimensional string array

    for(int i=0;i<arraystr1.length;i++) //printing all elements of string array
    {
        for(int j=0;j<arraystr1[i].length;j++) {
            System.out.println("string is" + arraystr1[i][j]);
        }
    }

    char[] arraychar= {'a'}; //char array
    System.out.println("character is" +arraychar[0]);
}
}

```

Enhanced for loop:

```

for(declaration : expression) {
    // Statements
}

```

OR

```

for(data_type variable : array | collection){}

```

- **Declaration** – The newly declared block variable is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression** – This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

```

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(",");
}
}
}

```

```

1. class EnhancedForLoop {
2.     public static void main(String[] args) {
3.         String languages[] = { "C", "C++", "Java", "Python",
4.             "Ruby"};
5.         for (String sample: languages) {
6.             System.out.println(sample);
7.         }
8.     }
9. }

```

Data Types:

Dynamically typed programming languages determine the data type of a variable at runtime — meaning while the program is running. Therefore, in these types of languages the programmer doesn't need to specify the data type of a variable.

Java is a statically typed programming language — which means, it expects its variables to be declared before they can be assigned values because, in statically typed languages the data type is checked at compile time.

Local Variable – Type Inference

While Java is a statically typed language, in version 10 of Java they added support for type inference — which means we can declare a variable as `var` and Java will infer the data type based on what is assigned to the variable.

Example:

```
var grid= getGridArray();
```

With var, Java will infer that whatever is the data type of the value returned from the getGridArray() is what grid should be declared as. This is really nice, and it helps remove boilerplate code when working with more complex data types.

Rules:

Type inference variables are only allowed for local variables. You cannot use var for declaring a global variable.

You must initialize these variables at the time of declaration, otherwise, the compiler is unable to infer what the data type should actually be.

Type inference is not allowed in the headers of methods or constructors. You must still explicitly declare the data type of parameters as well as the return types of methods.

Wrapper Classes:

All primitive data types can also be expressed as objects. These objects are known as wrapper classes.

Primitive data type - Wrapper Classes

byte -Byte

short-Short

int-Integer

long-Long

float-Float

double-Double

boolean- Boolean

char-Character

These wrapper classes exist for all primitive data types.

```
int number1= 5;
```

```
Integer number2=12;
```

Only number1 has a data type of int, which is primitive.

And number2 has a data type of Integer, which is a wrapper class for the primitive data type int. This makes number2 an object, so that you can utilize convenience methods that are available in the Integer class.

Refer Oracle Javadocs:

<https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>

Example:

```
public static void convertstringToInt(){  
    double [] numbers={1.2,45.5};  
    for(Double num : numbers){  
        System.out.println(num.intValue());  
    }  
}
```

We have an array of numbers declared as primitive type double. Yet, in the for loop, we have changed their type to its corresponding wrapper class denoted by the capital D in Double.

Now that this is an object, we can use the dot operator and make use of the methods, such as intValue() which will convert that decimal number into a whole number.

Some of them in the Oracle Javadocs are static, meaning they can be called with just the Integer class; Integer.bitCount() for example. And then some of them can be used from the instantiation of the object.

We have some useful ones here like compare() where you can pass in two numbers.

There's doubleValue(), floatValue(), things like this to change it to a different data type in long value.

We have max() and min() where you can pass into numbers and get the max or the min of those two numbers.

Then also you can turn an integer into a String by calling toString().

Strings:

A String is an object and it's a sequence of characters. But much like the wrapper classes, the String class provides a lot of helpful methods for dealing with them.

- A popular one such as `charAt()` will allow you to get a specific character within the String by its index. This is how you would iterate a String as you would an array. So, you won't be able to use the brackets like we use for other arrays. You would use the `charAt()` method within a loop, for example. Or if you wanted to get the very first element of the String, you can do a `charAt(0)`.
- There's also this one, `contains()`, which is really useful. This will allow you to see if a String contains a certain sequence of characters.
- `equals()` we've used before, as well as `equalsIgnoreCase()`.
- `endsWith` is a good one if you want to know if something is at the end of a String. There's also `startsWith()`.
- And then this `format()` one we've talked about a little bit in previous chapters, where you can use placeholders and then add items back into the String.
- There's `indexOf()` where you can pass in some substring of the String and it'll tell you what position it is within the String.
- `isEmpty()` can be used if you want to know if this String has data in it or not.
- `length()` lets us know how many characters are in the String and we can use this just like we use on the arrays.
- There's also this `matches()` one where you can pass in a regular expression and see if the String matches that regular expression.
- `replace()` and `replaceAll()` are useful if there's something within a String that you want to change to something else. You can use these replace methods.
- `split()` is also a good one where you have a String and you want to divide it by some delimiter.
- Here's `substring()` where you can get part of the String — you can give it a beginning and ending index, and just get that portion of the String.
- There's a `toLowerCase()` which is really useful and `toUpperCase()` as well.
- `trim()` is used to get off the white spaces at the beginning or the end of a String.
- `valueOf()` lets you pass in any data type and get the String value of it.

Example:

```
public class TextProcessor {  
  
    public static void main(String[] args){  
        countWords("I love Test Automation University");  
    }  
  
    /**
```

```

    * Splits a String into an array by tokenizing it.
    * Counts words and prints them
    * @param text Full string to be split
    */
    public static void countWords(String text){
        var words = text.split(" ");
        int numberOfWords = words.length;

        String message = String.format("Your text contains %d words:",
numberOfWords);
        System.out.println(message);

        for(int i=0; i<numberOfWords; i++){
            System.out.println(words[i]);
        }
    }
}

```

We're going to create a method that prints a given String backwards

We are using `text.length()` however, `text.length()` is going to give us the number of characters, but remember, indexes start at 0. So, we have to say `text.length() - 1` and that'll give us the index of the very last character.

We'll use `i>=0` for our condition. So that means it's gotten to the beginning of the array by this point. And instead of `i++` we are going to use `i--`.

Inside of the loop, we can simply print out.

Notice we're not going to do `println()` because we want it all on the same line. So we used `print()`.

We can do our `text` — and remember I told you, you cannot use the brackets for this, right? While this essentially is an array of characters, it is still an object, so we need to `text.charAt()` and then give it the index.

```

public class TextProcessor {

    public static void main(String[] args){

```

```

        reverseString("Hello TAU!");
    }

    /**
     * Prints a String in reverse order
     * @param text String to reverse
     */
    public static void reverseString(String text){
        for(int i=text.length()-1; i>=0; i--){
            System.out.print(text.charAt(i));
        }
    }
}

```

StringBuilder Class-wrapper class

append
 insert
 delete
 reverse

The String object is immutable, which means it does not allow for manipulation of the actual String itself. And while there are lots of great methods for Strings, in order to do things like insert or delete characters from a String, you need to use the StringBuilder class.

As we're looking at each character of this String, we want to determine if it is an uppercase. If it is an uppercase, then we need to insert a space into this String.

And as I said, Strings are immutable so we wouldn't be able to do this with just this regular String object of text.

That's why we needed to use the StringBuilder which will allow us to modify a String.

So, I want to say "if i is not equal to zero", meaning this is not the first letter of the String because we don't want to add a leading space to the beginning of the String — AND (&&) — then I'm going to use the Character wrapper class so that I can use these nice methods available.

I want to use isUppercase() so I can then pass in this specific character by using the charAt().

So, I'm saying if this character is uppercase, then I know to add a space here.

Then I can say, take this modifiedText and insert.

This will take a position, so, the position is i (wherever we are right now), and insert a space.

Once I've gotten here, I want to increment `i` so that we move past the space that we've just entered and on to the character, which then once it comes back here, will go on to the next character.

```
public class TextProcessor {

    public static void main(String[] args){
        addSpaces("HeyWorld!It'sMeAngie");
    }

    /**
     * Adds spaces before each uppercase letter
     * @param text jumbled text
     */
    public static void addSpaces(String text){

        var modifiedText = new StringBuilder(text);

        for(int i=0; i< modifiedText.length(); i++){
            if(i!=0 && Character.isUpperCase(modifiedText.charAt(i))){
                modifiedText.insert(i, " ");
                i++;
            }
        }

        System.out.println(modifiedText);
    }
}
```

Character wrapper class

- isUpperCase
- isLowerCase
- isDigit
- isWhitespace

String wrapper class

- startsWith
- endsWith

indexOf
charAt
substring
split
trim
replace
concat

Inheritance:

Inheritance is where one class becomes an extension of another class, therefore inheriting the members of that class.

There are two parties involved in an inheritance — a parent and a child. The parent is known as the superclass, or sometimes also referred to as the parent class or base class. The child is known as the subclass, or sometimes referred to as the child class or the derived class.

When an inheritance relationship is created between these two classes, then the child class inherits the members of the parent class.

This allows classes to reuse data that already exists within other classes.

Subclasses should be a more specialized form of the superclass that they inherit from.

Example:

```
Public class Person{}  
public class Employee extends Person{}  
Employee is the child class  
Person is the parent or superclass
```

Constructors in Inheritance:

When we create a new instance of the subclass, it's going to make a call to the superclass's constructor before it executes the subclass's constructor.

```
public class Person {  
    public Person(){  
        System.out.println("In Person default constructor");  
    }  
}  
  
public class Employee extends Person {
```

```

public Employee(){
    System.out.println("In Employee default constructor");
}
}

```

```

Void main(){
Employee employee=new Employee();
}

```

Output:

```

In Person default constructor
In Employee default constructor
-----

```

Another Constructor:

```

public Person (String name){

S.o.p(" Second constructor ");

}

```

If we go inside Employee and we wanted that constructor to be called instead of the Person default constructor, then we can explicitly state that by using a call to super which will call into the constructor of the superclass.

If we call `super` without anything in the parentheses, this will call the default constructor of the `Person` class. However, if we wanted to call into any other constructors, we would just pass the required arguments.

```

public Employee{

super("Angie")
S.o.p(" Employee constructor ");
}

```

Output:

```

Second Constructor
Employee constructor

```

Explicit Calls to Superclass Constructors Must Come First

Note that if you're going to make an explicit call to the superclass's constructor, then that call must be the very first line of the subclass's constructor.

Another interesting thing is that if the superclass does not have a default constructor, then the subclass must explicitly call one of its other constructors.

Just to reiterate the rules about constructors in inheritance:

- A superclass's constructor is run before the subclass's constructor.
- Explicit calls can be made to superclass's constructor from one of the subclass's constructors by using `super()`.
- Explicit calls to the superclass's constructor must be the first statement in the subclass's constructor.
- If the superclass does not have a default constructor, the subclass must explicitly call one of its other constructors.

Overriding Inherited Methods

a subclass inherits the members of its parent, however, a subclass may want to change the functionality of a method that it inherited.

This is allowed by *overriding* the inherited method.

```
public class Rectangle {  
  
    protected double length;  
    protected double width;  
    protected double sides = 4;  
  
    public double getLength() {  
        return length;  
    }  
  
    public void setLength(double length) {  
        this.length = length;  
    }  
}
```

```

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getSides() {
        return sides;
    }

    public void setSides(double sides) {
        this.sides = sides;
    }

    public double calculatePerimeter(){
        return (2 * length) + (2 * width);
    }
}

public class Square extends Rectangle {

    @Override
    public double calculatePerimeter(){
        return sides * length;
    }
}

```

Override Annotation

It's encouraged that you use the override annotation, which is the @ symbol followed by the word Override — `@Override`. This is not required, but it's strongly encouraged. This lets Java know that your intention is to override the method that you inherited from your super class.

Overloading Inherited Methods

Overloading methods is when you have multiple methods with the exact same name but with different parameter lists.

we looked at overloading methods within the same class. When dealing with a subclass, we can overload a method that we have inherited from a superclass, even though that method lives in another class.

Let's update the `Rectangle` and `Square` classes to demonstrate how this is done.

I'm going to create a new method in the `Rectangle` class that simply prints out a statement.

```
public void print(){  
    System.out.println("I am a rectangle");  
}
```

And then in the `Square` class, I'm going to overload this by creating a method with the same name, but it's going to take a `String`.

```
public void print(String what){  
    System.out.println("I am a " + what);  
}
```

Notice, when I attempt to call the `print` method, there's only 1 method available in this `Rectangle` class called `print`.

However, when I attempt to call `print` on the `Square` class, there's two. There's the one that we inherited, and also the overloaded one.

Overloading - same method name and body but different parameter lists

Overriding - same method name and parameter but different body

Access Limitations

When a subclass inherits from a superclass, not everything is inherited.

constructors are not technically members of a class and therefore they are not inherited.

All the public and protected methods and fields in a superclass, those are indeed inherited, but the private methods and fields are not.

Also, if there are any final methods, meaning methods that have the word `final` in the header, these are inherited but cannot be overridden.

Let's look at this.

In the `Square` class, we were able to use the fields, `side` and `length`, because we inherit them from `Rectangle`.

The reason we were able to inherit them is because they're marked as `protected`.

So, if they're marked as `protected` or `public`, or no marking, then we'll be able to inherit them.

However, let's make one of these as `private`: we would get error

If the superclass has a `public` access modifier, so when we inherited this, we had to override it, we had to declare it as `public`.

If we were to try to declare it as anything stricter, we would get an error.

For a `public` superclass method on overriding the method has to be `public`, if `protected` or `private` throws error

let's say in the superclass instead of this being `public`, it was `protected`. So, when we try to override here, we had the `protected` here in the parent class but in the subclass, we've made it `public`. This is fine and it's allowed.

Chain of Inheritance

Java classes can only directly inherit from one superclass. However, a superclass can also inherit from another class, thus forming a chain of inheritance where the subclass inherits from their ancestor classes as well.

Person -> Woman -> Mother

Person -> Employee

Although we have `Woman` that inherits from `Person`, and we also have `Employee` that inherits from `Person`, there is no relationship at all between `Employee` and `Woman`. Even though they both have the same parent, don't think of them as siblings or anything of the sort. They have no connection.

Example:

```
public class Person {  
  
    private String name;  
    private int age;  
    private String gender;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public String getGender() {  
        return gender;  
    }  
  
    public void setGender(String gender) {  
        this.gender = gender;  
    }  
}  
  
public class Woman extends Person {  
  
    public Woman(){  
        setGender("female");  
    }  
}  
  
public class Mother extends Woman {
```

```
}
```

So, we'll create an instance of `Mother` and notice that we can set the name on `mom` even though that's not in the `Mother` class nor is it in the `Woman` but it's in the `Person` class and we've inherited through that ancestor.

```
public class InheritanceTester {
```

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

```
        Mother mom = new Mother();
```

```
        mom.setName("Glenda");
```

```
        System.out.println(mom.getName() + " is a " + mom.getGender());
```

```
    }
```

```
}
```

Polymorphism:

Polymorphism is the ability to take multiple forms. Specifically, in object-oriented programming, polymorphism is where a subclass can define their own unique behaviors, and yet share some of the same behaviors of their superclass.

An example of polymorphism is where an object has a superclass type but is an instance of a subclass.

Here we have an object called `dog` and it's defined as type `Animal` but it is an instance of `Dog`.

```
Animal dog = new Dog();
```

We're able to do this because `Dog` inherits from `Animal`, and therefore `dog` is a `Animal`.

Animal.java

```
public class Animal {
```

```
    public void makeSound(){
```

```
        System.out.println("unknown animal sound");
```

```
    }
```

```
}
```

Dog.java


```

public class Dog extends Animal {

    @Override
    public void makeSound(){
        System.out.println("woof");
    }

    public void fetch(){
        System.out.println("fetch is fun!");
    }
}

```

Cat.java

```

public class Cat extends Animal {

    @Override
    public void makeSound(){
        System.out.println("meow");
    }

    public void scratch(){
        System.out.println("I am a cat. I scratch things.");
    }
}

```

Zoo.java

```

public class Zoo {

    public static void main(String[] args){
        Dog rocky = new Dog();
        rocky.fetch();
        rocky.makeSound(); // Here makesound() is gonna call the dog class method as it
        // overridden. On overriding the overridden method is called.
    }
}

// This is where polymorphism comes into place

```

I can specify this type as Animal, and let's call this dog sasha. And I'm going to instantiate sasha as a Dog.

```

Animal sasha = new Dog();
sasha.makeSound();

```

```
sasha = new Cat();  
sasha.makeSound();
```

So, `sasha` is of type `Animal` but an instance of `Dog`. Now when I do this, `sasha` can make a sound even though `sasha` is of type `Animal`, because we overrode this `makeSound()` method in the `Dog` class. So, that is the one that will be executed.

Let's continue on, we're going to actually change Sasha's type — this is *real* polymorphism.

I can say `sasha` now is a `Cat`. So, I've changed Sasha from an instance of `Dog` to an instance of `Cat`.

Why is this legal?

Because they are both of type `Animal`, and since I've specified `sasha` as type `Animal`, then this is legal. So now we can make a sound. And now we should see two different sounds.

Now, because `sasha` is of type `Animal` and not of type `Cat`, then `sasha` does not have access to the `scratch()` method. Because the `scratch()` method belongs to `Cat`, and yet `sasha` is of type `Animal` so we cannot make a direct call of `sasha.scratch()`.

However, we can still get around this by casting.

Type Casting

Casting is the act of converting an object's type into a different type.

If we did `sasha.scratch()`, notice here Sasha has been cast to type `Cat`.

```
Animal sasha = new Dog();  
sasha.makeSound();
```

```
sasha = new Cat();  
sasha.makeSound();  
((Cat) sasha).scratch();
```

So, this is not changing the overall object — `sasha` is still of type `Animal`. But it's saying in this specific call go ahead and make `sasha` of type `Cat`, just so that we can execute this method.

Now let's create a new method inside of this `Zoo` class to feed the animals.

Now notice that this method `feed()` takes an `Animal`.

Technically `rocky` is a `Dog`, but yet we could still pass `rocky` into this `feed()` method.

The reason we're able to do that again is because of polymorphism. Because it accepts this superclass of `Animal`, that means we can pass in an `Animal` object or any subclass of `Animal`.

Let's do the same for `sasha`.

Now within this method of `feed()`, let's say that I want to do different things based on the type. If it's actually a dog, I might want to give it dog food, and if it's a cat I want to give it cat food.

Even though our parameter is of type `Animal`, there's a way that we can still figure out exactly what the subclass is by doing a check.

I can use an operator that's called `instanceof`

instanceof Operator

`instanceof` will do a check to see if whatever's on the left side is actually an instance of whatever you specify on the right side.

So, I can check to see if this `Animal` is an instance of `Dog`. And this will return a boolean value. If true, then I know that this object was instantiated as type `Dog`.

Zoo.java

```
public class Zoo {  
  
    public static void main(String[] args){  
  
        Dog rocky = new Dog();  
        rocky.fetch();  
        rocky.makeSound();  
        feed(rocky);  
  
        Animal sasha = new Dog();  
        sasha.makeSound();  
        feed(sasha);  
    }  
}
```

```

        sasha = new Cat();
        sasha.makeSound();
        ((Cat) sasha).scratch();
        feed(sasha);
    }

    public static void feed(Animal animal){

        if(animal instanceof Dog){
            System.out.println("here's your dog food");
        }

        else if(animal instanceof Cat){
            System.out.println("here's your cat food");
        }
    }
}

```

- We see that the first call was to `fetch()`, and we got “fetch is fun” from the `Dog` class.
- “woof” is from the overwritten `makeSound` method of the `Dog` class.
- When passing `rocky` to `feed()`, we see that it did find that this was an instance of `Dog`.
- For `sasha` who's of type `Animal` but an instance of `Dog`, we also see the `makeSound()` prints “woof”
- It also feeds `sasha` dog food when we did the `instanceof` check.
- When `sasha` became a cat, the `makeSound` prints “meow”.
- Then we called the `scratch()`.
- This — “here’s your cat food” — was printed out because `instanceof` shows that it is a `Cat`.

Key Points about Polymorphism

- An object can have a superclass type and a subclass instance.

- Polymorphic objects can only access the methods of their type, not of their instance, unless you use casting.
- If a method is overwritten by the subclass, the polymorphic object will execute the overwritten method at runtime.
- The `instanceof` operator is used to determine if an object is an instance of a certain class.

Abstraction:

In Java, we have the reserved word, `abstract` — which is a non-access modifier that can be used on classes and methods. It is used when you want to define a template for a class or a method, but do not intend for it to be used as is.

An `abstract` class is not designed to be instantiated. It's designed to be extended, thus pushing the burden of implementation onto the subclass.

So, this class cannot be instantiated, but its purpose is to serve as a parent class for more specific subclasses to extend from. An `abstract` method has no body, only the signature of the method is defined, and it is not designed to be run as is. It's designed to be overridden by a subclass.

We would use the reserved word, `abstract`, which is a non-access modifier, and then we would give the return type, and the method name, followed by the parameter list that we like to have (in this case, we don't need anything), and then a semicolon.

That's it, no curly braces, no body to this method.

Now because we have an `abstract` method in this class, we are required to make the class itself `abstract` as well. We can make a class abstract by simply writing the word `abstract` before class.

An `abstract` class doesn't only have to contain `abstract` methods.

It can contain implemented methods as well. So, let's create a print method.

```
public abstract class Shape {

    abstract double calculateArea();
}

public void print(){
    System.out.println("I am a shape");
}
```

So, notice the `print` method is not abstract and the `calculateArea()` method is.

Now, let's go to the `Rectangle` class, and we are going to extend from the `Shape` class.

```
public class Rectangle extends Shape{

}
```

Right away, we get a compilation error, and it's saying that this class must either be declared abstract or implement the abstract method `calculateArea()`. For example, we didn't want to provide the implementation, then we don't have to do it. But this class itself would then need to be declared as abstract because it contains abstract methods. Technically, because they were inherited.

Let's go ahead and implement the abstract methods that `Rectangle` inherited.

`Rectangle.java`

```
public class Rectangle extends Shape {

    private double length;
    private double width;

    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        this.length = length;
    }

    public double getWidth() {
        return width;
    }
}
```

```

    }

    public void setWidth(double width) {
        this.width = width;
    }

    public Rectangle(double length, double width){
        setLength(length);
        setWidth(width);
    }

    @Override
    double calculateArea() {
        return length * width;
    }
}

```

Now let's go to another class, ShapeTester.

We're going to test out what we made here.

So, let's make a Shape object and call it rectangle. Then we'll attempt to make it a new instance of Shape.

Notice the compilation error. That's because you cannot instantiate an abstract class.

```

public class ShapeTester {

    public static void main(String[] args){

        Shape rectangle = new Rectangle(5, 7);
        rectangle.print();
        System.out.println(rectangle.calculateArea());
    }
}

```

#Abstraction Key Points

- Abstract classes and methods are templates that are meant to be implemented by their subclasses.
- The classes and methods are declared abstract by using the abstract keyword.
- If a subclass inherits from an abstract class, it must implement its abstract methods or the class itself must be declared as abstract.

- Abstract classes cannot be instantiated. They are only to be used as a superclass. If even one method in a class is abstract, then the entire class must be declared abstract as well.

Interfaces:

An *interface* is similar to an abstract class, except that in an abstract class there can be some methods that are implemented.

However, an interface consists of all abstract methods. Because all of the methods of an interface are abstract by nature, there's no need to declare the methods as abstract. It's just a given.

The other way interfaces are different from abstract classes is that interfaces are implemented, not extended.

Any class that implements an interface must implement all of its methods, or it must declare itself as an abstract class.

Now inside of here, just like the abstract classes we have method definitions with no body, except we don't have to write the keyword of `abstract`.

We're going to create a `Product` interface and then we're going to create a `Book` class that implements the product.

Product.java

```
public interface Product {  
  
    double getPrice();  
    void setPrice(double price);  
  
    String getName();  
}
```



```

    void setName(String name);

    String getColor();
    void setColor(String color);

}

```

Now notice, even though we have the getter and setter methods here, we do not have the fields defined themselves.

That's because if you define a field within an interface, then that field has to be `final`. Because it's `final`, it's essentially a constant, and therefore you have to also give it a value.

If the purpose of the interface is so that others can inherit from this, and this is serving as just a template or a blueprint, then you don't want to add fields where the values can change based on the implementation of this interface.

So, we'll just provide the methods.

Book.java

```

public class Book implements Product {

    private double price;
    private String name;
    private String color;
    private String author;
    private int pages;
    private String isbn;

    @Override
    public double getPrice() {
        return price;
    }

    @Override
    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String getName() {

```

```

        return name;
    }

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String getColor() {
        return color;
    }

    @Override
    public void setColor(String color) {
        this.color = color;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public int getPages() {
        return pages;
    }

    public void setPages(int pages) {
        this.pages = pages;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}

Customer.java
package chapter11;

public class Customer {

```

```
public static void main(String[] args){  
  
    Product book = new Book();  
    book.setPrice(9.99);  
}  
}
```

Default methods in interfaces

Like regular interface methods, **default methods are implicitly public** — there's no need to specify the *public* modifier.

Unlike regular interface methods, they are **declared with the *default* keyword at the beginning of the method signature**, and they **provide an implementation**.

Let's see a simple example:

```
public interface Vehicle {  
  
    String getBrand();  
  
    String speedUp();  
  
    String slowDown();  
  
    default String turnAlarmOn() {  
        return "Turning the vehicle alarm on.";  
    }  
  
    default String turnAlarmOff() {  
        return "Turning the vehicle alarm off.";  
    }  
}
```

The reason why *default* methods were included in the Java 8 release is pretty obvious.

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface,

all the implementations will be forced to implement them too. Otherwise, the design will just break down.

Default interface methods are an efficient way to deal with this issue. They **allow us to add new methods to an interface that are automatically available in the implementations**. Thus, there's no need to modify the implementing classes.

In this way, **backward compatibility is neatly preserved** without having to refactor the implementers.

And let's write the implementing class:

```
public class Car implements Vehicle {

    private String brand;

    // constructors/getters

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The car is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

Lastly, let's define a typical *main* class, which creates an instance of *Car* and calls its methods:

```
public static void main(String[] args) {
    Vehicle car = new Car("BMW");
    System.out.println(car.getBrand());
    System.out.println(car.speedUp());
    System.out.println(car.slowDown());
}
```

```
        System.out.println(car.turnAlarmOn());
        System.out.println(car.turnAlarmOff());
    }
```

Please notice how the *default* methods *turnAlarmOn()* and *turnAlarmOff()* from our *Vehicle* interface are **automatically available in the *Car* class**.

Multiple Interface Inheritance Rules

Default interface methods are a pretty nice feature indeed, but with some caveats worth mentioning. Since Java allows classes to implement multiple interfaces, it's important to know **what happens when a class implements several interfaces that define the same *default* methods**.

To better understand this scenario, let's define a new *Alarm* interface and refactor the *Car* class:

```
1
2  public interface Alarm {
3
4
5
6      default String turnAlarmOn() {
7
8          return "Turning the alarm on.";
9
10     }

    default String turnAlarmOff() {

        return "Turning the alarm off.";
    }
```

```
}
```

```
}
```

With this new interface defining its own set of *default* methods, the *Car* class would implement both *Vehicle* and *Alarm*:

```
1
2 public class Car implements Vehicle, Alarm {
3
4     // ...
5
6 }
```

In this case, **the code simply won't compile, as there's a conflict caused by multiple interface inheritance** (a.k.a the [Diamond Problem](#)). The *Car* class would inherit both sets of *default* methods. Which ones should be called then?

To solve this ambiguity, we must explicitly provide an implementation for the methods:

```
1
2 @Override
3
4 public String turnAlarmOn() {
5
6     // custom implementation
7
8 }
9
```

```
@Override
```

```
public String turnAlarmOff() {
```

```
// custom implementation
```

```
}
```

We can also **have our class use the *default* methods of one of the interfaces.**

Let's see an example that uses the *default* methods from the *Vehicle* interface:

```
1
2  @Override
3
4  public String turnAlarmOn() {
5
6      return Vehicle.super.turnAlarmOn();
7
8  }
9
```

```
@Override
```

```
public String turnAlarmOff() {

    return Vehicle.super.turnAlarmOff();

}
```

Similarly, we can have the class use the *default* methods defined within the *Alarm* interface:

```
1
2  @Override
3
4  public String turnAlarmOn() {
5
6      return Alarm.super.turnAlarmOn();
7
8  }
9
```

```
@Override

public String turnAlarmOff() {

    return Alarm.super.turnAlarmOff();

}
```

Furthermore, it's even **possible to make the *Car* class use both sets of default methods**:

```
1
2  @Override
3
4  public String turnAlarmOn() {
5
6      return Vehicle.super.turnAlarmOn() + " " +
7  Alarm.super.turnAlarmOn();
8
9  }
```

```
@Override
```



```
public String turnAlarmOff() {  
  
    return Vehicle.super.turnAlarmOff() + " " +  
    Alarm.super.turnAlarmOff();  
  
}
```

1. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

#Key Points on Interfaces

- Interfaces cannot be instantiated.
- Interfaces are implemented, not extended.
- Any class that implements an interface must implement all of its methods or it must declare itself `abstract`.
- Methods in an interface must be default or abstract (there's no explicit declaration needed for abstract methods)

Also notice that we didn't put an access modifier on the methods in an interface, either. That's because by default they're all `public`.

While a class can only extend one class, it can implement multiple interfaces.

Collection frameworks:

Java provides a collections framework which consists of interfaces, classes and methods to store and manipulate aggregate data.

`Collection` itself is an interface and is the root of the hierarchy. Java does not provide any direct implementations of the `Collection` interface, but there are other interfaces which inherit from `Collection`

Set

`Set` is a collection which cannot contain duplicate elements.

An example use case for `Set` would be a standard deck of 52 playing cards, where each card would be an element in the collection and each of them are unique.

#List

`List` is a collection where the elements are ordered.

`List` can contain duplicate elements.

An example of where `List` might be used would be the phone numbers from your call history. They are listed in order and some numbers may appear more than once.

#Queue

`Queue` is a collection of ordered elements, which is typically used to hold items that are going to be processed in some way.

An example use case would be the people in the checkout line at a supermarket. People who are new to the queue are added at the end and the processing of the queue is handled from the beginning. This type of processing is well known as *first in-first out (FIFO)*.

#Map

Then finally we have `Map`, which is not a true collection, meaning it does not inherit from the `Collection` interface. However, it contains collection-like operations, which enable them to be used as collections.

Maps are used to hold key/value pairs.

An example for this would be a list of bank transactions where each `Map` entry has a unique transaction ID serving as the key and the value would be the actual transaction object.

Example of Set:

```
1 Set fruit = new HashSet();
2 fruit.add("apple");
3 fruit.add("lemon");
4 fruit.add("banana");
5 fruit.add("orange");
6 fruit.add("lemon");
7
8 System.out.println(fruit.size()); //4
9 System.out.println(fruit); //[banana, orange, apple, lemon]
```

Again, a `Set` is an unordered collection of unique objects. Because `Set` is an interface, it cannot be instantiated, but it can be an object's type. This is demonstrated on line 1.

Some implementations of `Set` are `HashSet`, `LinkedHashSet` and `TreeSet`. I won't go into details for these; however, I have placed a link in the resources section should you want to go deeper on this topic.

Notice in our example here, we use the `add` method to place items into the set. Note that if I did attempt to add a duplicate to the set, like we're doing with "lemon" on line 6, no error would occur. However, the duplicate elements simply would not be stored.

When we call `fruit.size()` on line 8, we're returned with the number 4.


When I print out the set on line 9, the order is not the same order that I added the elements to the set. Since the set is unordered, we have no control over where our elements will be placed.

Please note that while I'm demonstrating this using Strings, collections can hold any type of object.

Here are the methods available for sets.

| Method | Description |
|----------|---|
| add | Adds an object to the collection. |
| clear | Removes all objects from the collection. |
| contains | Returns true if a specified object is an element within the collection. |
| isEmpty | Returns true if the collection has no elements. |
| iterator | Returns an Iterator object for the collection, which may be used to retrieve an object. |
| remove | Removes a specified object from the collection. |
| size | Returns the number of elements in the collection. |

Example if Lists:



```
1 List fruit = new ArrayList();
2 fruit.add("apple");
3 fruit.add("lemon");
4 fruit.add("banana");
5 fruit.add("orange");
6 fruit.add("lemon");
7
8 System.out.println(fruit.get(2)); //banana
9 System.out.println(fruit.size()); //5
10 System.out.println(fruit); //[apple, lemon, banana, orange, lemon]
```

Some implementations of lists are `ArrayList`, `LinkedList`, `Stack` and `Vector`. There's a link to the Javadoc in the Resources section.

Lists are ordered, so the elements can be accessed by their position in the lists as shown on line 8. Just as with arrays, the index starts at 0.

A `List` can also contain duplicate elements. We added "lemon" on line 3 and line 6 and they both were stored in the array as demonstrated by `fruit.size()` returning 5 on line 9, and then also on line 10 we see lemon appear twice.

Also, note on line 10 when we print the list, we see that the elements stayed in the order that we added them.

In addition to the methods inherited from `Collection`, here are some additional methods available for lists.

| Method | Description |
|--|--|
| <code>add(int index, Object obj)</code> | Inserts <code>obj</code> into list at position of <code>index</code> . |
| <code>addAll(int index, Collection c)</code> | Inserts all elements of <code>c</code> into the list at position of <code>index</code> . |
| <code>get(int index)</code> | Returns the object stored at the specified <code>index</code> within the invoking collection. |
| <code>indexOf(Object obj)</code> | Returns true if the collection has no elements. |
| <code>lastIndexOf(Object obj)</code> | Returns the index of the first instance of <code>obj</code> in the list. |
| <code>listIterator()</code> , <code>listIterator(int index)</code> | Returns the index of the last instance of <code>obj</code> in the list. |
| <code>remove(int index)</code> | Removes the element at position <code>index</code> and returns deleted element. |
| <code>set(int index, Object obj)</code> | Assigns <code>obj</code> to the location specified by <code>index</code> within the invoking list. |
| <code>subList(int start, int end)</code> | Returns a list containing elements from <code>start</code> to <code>end</code> . |

Example of Queues:

```

1 Queue fruit = new LinkedList();
2 fruit.add("apple");
3 fruit.add("lemon");
4 fruit.add("banana");
5 fruit.add("orange");
6 fruit.add("lemon");
7
8 System.out.println(fruit.size()); //5
9 System.out.println(fruit); //[apple, lemon, banana, orange, lemon]
10
11 fruit.remove();
12 System.out.println(fruit);//[lemon, banana, orange, lemon]
13
14 System.out.println(fruit.peek());//lemon

```

The elements are ordered, and the collection can contain duplicates.

The most common implementations of `Queue` are `LinkedList` and `PriorityQueue`, which you can read more about in the Javadocs.

Queues typically follow the first in-first out algorithm – which means elements are processed in the order in which they are entered. For example, on line 11 we call `fruit.remove()`. We didn't have to specify which object to remove, it will remove the first one in the queue.

The first element of the `Queue` is known as the head. By calling `remove()`, we removed “apple” from the queue, which was the head.

The last element of a `Queue` is known as the tail.

Sometimes we may want to know what's the next item to be processed before we actually do anything. For that you can use `peek()`, which is on line 14. This returns the head of the queue.

| Method | Description |
|----------------------|--|
| <code>add</code> | Adds element to the tail of the queue. |
| <code>peek</code> | Used to view the head of the queue without removing it. Returns false if queue is empty. |
| <code>element</code> | Similar to <code>peek()</code> but throws exception if queue is empty. |
| <code>remove</code> | Removes and returns the head of the queue. Throws exception if queue is empty. |
| <code>poll</code> | Similar to <code>remove()</code> , but returns null if queue is empty. |

Examples of Maps:

```
1 Map fruitCalories = new HashMap();
2 fruitCalories.put("apple", 95);
3 fruitCalories.put("lemon", 20);
4 fruitCalories.put("banana", 105);
5 fruitCalories.put("orange", 45);
6 fruitCalories.put("lemon", 17);
7
8 System.out.println(fruitCalories.size()); //4
9 System.out.println(fruitCalories); //{banana=105, orange=45, apple=95, lemon=17}
10
11 System.out.println(fruitCalories.get("lemon")); //17
12
13 System.out.println(fruitCalories.entrySet()); //[banana=105, orange=45, apple=95, lemon=17]
14
15 fruitCalories.remove("orange");
16 System.out.println(fruitCalories); //{banana=105, apple=95, lemon=17}
```

You can access an element in a `Map` by its key. While the keys are required to be unique, the values are not.

Popular implementations of the `map` interface are `HashMap`, `TreeMap` and `LinkedHashMap`, which you can read more about in the Javadocs.

Let's say that we wanted to add calorie information for each of the fruits. We can use the fruit name as a key and the calories as the value.

Remember, a `Map` is not technically a `Collection`, meaning it does not inherit from the `Collection` interface.

`Map` does not have access to the `add` method that we've seen in the other data structures.

`Map` has a `put` method, which is used to add the elements. The `put` method takes two arguments — a key and a value. We are using the fruit's name as the key and the calories as the value.

This example uses `String` and `Integer` as the key in value, but you can use any objects here.

Notice that on line 8, when we get the size of this map, it is 4. But we attempted to add 5 values, yet there's only 4 because the entry on line 6 is not a unique key.

However, when we look at the value that's printed out for key "lemon" on line 11, we see that it did indeed store that entry. That's because, if you call `put` passing in a key that already exists, the map will update that entry with the new value that was passed in. So, what we did on line 3 was technically an update.

There is another method called `putIfAbsent()`, which you can call to prevent yourself from unintentionally overriding something that's already in the map.

Calling `entrySet()`, like on line 13, will get you a `Set` object which technically then is a `Collection`.

Then line 15 shows how you remove an element by its key.

Here are some of the methods provided by the `Map` interface.

| Method | Description |
|--|---|
| <code>clear()</code> | Removes all key/value pairs from the map. |
| <code>containsKey(Object key)</code> | Returns true if the map contains an element that has key. |
| <code>containsValue(Object value)</code> | Returns true if the map contains an element that has value. |
| <code>entrySet()</code> | Returns a <code>Set</code> that contains the entries in the map. |
| <code>get(Object key)</code> | Returns the value associated with the key. |
| <code>isEmpty()</code> | Returns true if the map is empty. Otherwise, returns false. |
| <code>keySet()</code> | Returns a <code>Set</code> that contains the keys in the map. |
| <code>put(Object key, Object value)</code> | Puts an entry in the map, overwriting any previous value associated with the key. |
| <code>putAll(Map m)</code> | Puts all the entries from m into this map. |
| <code>putIfAbsent(Object key, Object value)</code> | Puts an entry in the map if the key does not already exist. |
| <code>remove(Object key)</code> | Removes the entry whose key equals key. |

In the examples so far, we've used the `add()` method to add elements to collections.

There's an easier way to add elements, and that's by using this `of()` method.

As you see here on line 1, we are adding elements to this list by using `List.of()` and passing in all of the elements we want to store within the list.

```
1 List unchangeableList = List.of("apple", "lemon", "banana");
2 unchangeableList.add("orange"); //UnsupportedOperationException
3 unchangeableList.remove(1); //UnsupportedOperationException
```

While this offers a great convenience in setting up this collection, it does come with a cost.

By setting a collection in this way, the collection becomes immutable, meaning it cannot change.

For example, calling the `add()` and `remove()` methods on this collection after using `List.of()` will throw an `UnsupportedOperationException`.

The same is true for Sets as well as Maps.

In the next section I'll show you how to loop through items in collections.

In this section, I'll show you a couple of ways to loop through Collections and Maps.

Collections.java

```
import java.util.*;
```

```
public class CollectionsDemo {
```

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

```
        setDemo();
```

```
        listDemo();
```

```
        queueDemo();
```

```
mapDemo();  
}  
}
```

Let's go into the setDemo() method. I want to show you a couple of ways that you can loop through Collections.

#Collection: Iterator

The first way is using what's called an iterator.

The Collection interface provides an iterator for you, in order to loop through a collection. For example, the Set is an unordered collection so there are no methods on Set that will allow us to get a certain item by its index, for example.

But we can use this iterator and let's just assign this to, we'll call it i. The iterator will allow you to go through each of the items in this collection.

In order to use it, you'll go through a loop.

```
public static void setDemo(){  
    Set<String> fruit = new HashSet();  
    fruit.add("apple");  
    fruit.add("lemon");  
    fruit.add("banana");  
    fruit.add("orange");  
    fruit.add("lemon");  
  
    var i = fruit.iterator();  
    while(i.hasNext())  
        System.out.println(i.next());  
}
```

This is a boolean expression — meaning while we have another element in the set, this will return true as long as we're not at the end of the collection, and once we've gone through all of them it will return false.

Once we're inside of the loop, let's just print them out. We can use `i.next()` which will return an object and it will be whatever type of object is inside of this particular collection.

That's one way to loop through a collection and this works on any collection. Let me show you another way.

#Collection: Enhanced For Loop

We can use the *enhanced for loop*.

Every item in this `Set` is a `String`, so we can say `String item`, and we want to iterate over `fruit`.

```
for(String item : collection){  
    System.out.println(item);  
}
```

#Collection: *forEach*

There's one more approach that I haven't introduced you to yet, and that's using the *forEach method*.

On our collection, we can do a call to `forEach()` and this for each will take a lambda expression.

Lambda expressions are kind of an advanced topic, we won't get into it in this course, however, I have provided a link for you in the Resource section if you'd like to learn more about it.

In the `forEach()`, we're going to give a generic name to whatever item we're on. It's going through, it's looping through each of these items. You give some name, you don't have to give a data type or anything like that. It's best to keep these names short because they are for this one-time use. They can't be used outside of this method call.

We'll say `x` and then you give an action — you do a dash, followed by a greater than sign to make an arrow and you give some action. What is it that you want this to do? I want it to print this out.

```
fruit.forEach(x -> System.out.println(x));
```

Running each of these (Iterator, enhanced for loop, for each) shows the exact same output. It's just a matter of preference of which one that you're most comfortable with.

For this lambda expression, you can even give this a little shortcut — what they call “syntactic sugar” where you can clean this up by using shorthand.

Instead of declaring this variable and using the arrow, you can just use a `System.out` and then put double colon and the method that you want to use

```
fruit.forEach(System.out::println);
```

What this is saying is, call this method and pass in whichever item we're on.

I prefer this way to iterate. It's a one liner, and it's nice and clean.

Map: Enhanced For Loop

Now let me show you the `mapDemo()` method.

```
public static void mapDemo(){  
  
    Map<String,Integer> fruitCalories = new HashMap();  
  
    fruitCalories.put("apple", 95);  
    fruitCalories.put("lemon", 20);  
    fruitCalories.put("banana", 105);  
    fruitCalories.put("orange", 45);  
    fruitCalories.put("lemon", 17);  
}
```

We can't use those exact same methods on the `Map` because, again, the `Map` is not a `Collection`.

To iterate over the `Map` we can use an *enhanced for loop*, but we can't just iterate over the `Map` itself.

We can, however, say give us the `entrySet` — which will give us a `Set` object. Once we have this `Set` object, then we can print out the value from the set.

```
for(var entry : fruitCalories.entrySet()){  
    System.out.println(entry.getValue());  
}
```

We used `entry.getValue()`, however this is red and it's saying it doesn't know what this is. This is a `Set` of a `Map` and `entry` is now a `Map`, but again, it's type is just `Object`.

We can specify the specific type with the diamond operator.

For a `Map` you need to provide two data types:

1. One for the key, so our key is a `String`.
2. The other for the value. The value is an `int`.

Because `Map` takes two objects, we can't just specify the `int` as the primitive data type, we have to use the wrapper class `Integer` which would then be an object.

Now, notice `getValue()` works because it knows that this `entry` is a `Map` of a `String` and an `Integer`.

Map: *forEach*

I want to show you one more way to iterate over a `Map`.

A `Map` also has access to the `forEach()` method. We can say `fruitCalories.forEach()` and inside of here we can pass a key and a value.

Let me put this on another line.

```
fruitCalories.forEach(  
    (k,v)->System.out.println("Fruit: " + k + ", Calories: " + v));
```

I've just given it a little short-term name (*k* for the key, and *v* for the value). We don't need a data type because, again, this is only known inside of this `forEach()`. We give it the little dash and the arrow and then we're going to say the action is to `println()`.

Exceptions :

```
public class ExceptionHandling {  
  
    public static void main(String args[]){  
        numbersExceptionHandling();  
    }  
  
    public static void numbersExceptionHandling(){  
        File file = new File("resources/numbers.txt");  
        Scanner fileReader = null;  
        try{  
            fileReader = new Scanner(file);  
  
            while(fileReader.hasNext()){  
                double num = fileReader.nextDouble();  
                System.out.println(num);  
            }  
        }catch(FileNotFoundException | InputMismatchException e){  
            e.printStackTrace();  
        }finally{  
            fileReader.close();  
        }  
    }  
}
```

You can add however many more exceptions you want, just separate them with this pipe symbol.

Try with Resources

That's just one way to handle this. We could also handle this without using a `finally` block.

There is an option that's referred to as *try with resources*, which will allow you to add a parenthesis after the word `try` and inside of the parentheses, you can initialize the resource there.

For example, the `Scanner` object is a resource, so we can define it within these parenthesis, and get rid of the other places where we declared and initialized it.

```
File file = new File("resources/numbers.txt");

try(Scanner fileReader = new Scanner(file)){

    while(fileReader.hasNext()){
        double num = fileReader.nextDouble();
        System.out.println(num);
    }
}
```

Once we have the resource declared at this level, we no longer need the `finally` clause to close the resource. *Try with resources* allows you to specify a resource and Java will automatically close this resource on your behalf once done with the try/catch.

This only works with classes that implement the [Closable](#) or [AutoClosable](#) interfaces, and `Scanner` happens to be one of those.

So that's an option when you're working with resources and you were only going to use `finally` to close the resource.

Otherwise, you may have other cases where `finally` is a good approach to execute some other statements, and that's fine as well. So you have multiple options.

If your method has code that has the potential of throwing an exception, you must either catch that exception or *rethrow* it.

#Rethrowing Exceptions

When a method throws an exception, it is assumed that the exception will be caught and handled, or rethrown by any calling methods.

Let's look at an example where we're going to write a program that creates a new file but does not catch the exceptions.

Let's make a new method and called `createNewFileRethrow`.

And inside of here, instead of doing try/catch we can say, "Listen, I know that this might throw an exception, but I don't want to handle that exception. I would like to just rethrow the exception."

You can do this by adding the `throws` word to the methods header, and then specifying the `Exception` that will be thrown.

```
package chapter13;

public class ExceptionHandling {

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

    public static void createNewFileRethrow() throws IOException{
        File file = new File("resources/nonexistent.txt");
        file.createNewFile();
    }
}
```

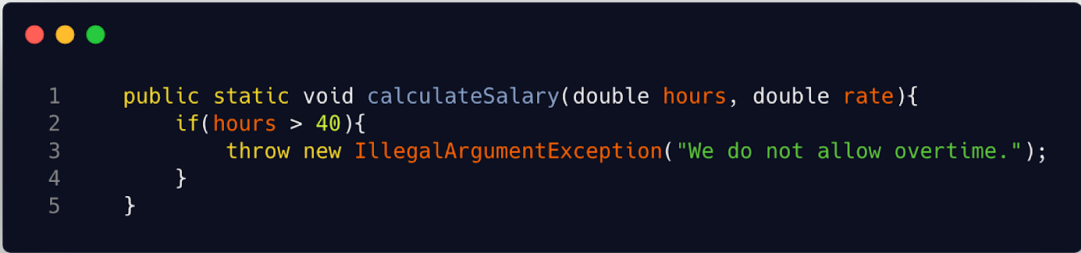
So, we know that this one throws an `IOException`, so we'll put `IOException`.

Polymorphism also works here, so you could say just `Exception` if you didn't know exactly what exceptions might be thrown. Now, if any method calls this method, then they are now on the hook for handling the exception.

Throwing Exceptions

Your method can initiate the throwing of an exception by using the word `throw` inside of the method's body and instantiating an exception.

Notice here, this method accepts `hours` and `rate`, but perhaps you don't allow overtime.



```
1 public static void calculateSalary(double hours, double rate){
2     if(hours > 40){
3         throw new IllegalArgumentException("We do not allow overtime.");
4     }
5 }
```

There's no Java exception that will be automatically called for such a thing, as this is only an error from the programmer's perspective.

So, the programmer themselves can throw an exception.

They can throw one that already exists, like this `IllegalArgumentException`, and just pass in a custom error message.

Or they can even define their own custom exception class, which extends from `Exception`. And they can throw that instead. Perhaps something like `NoOvertimeAllowedException`.