# **Codeacademy**

# **Les 1**

# **Java Review: Putting It All Together**

In this lesson, we’ve started writing our first programs in Java.

We’ve also learned rules and guidelines for how to write Java programs:

* Java programs have at least one class and one main() method.
  + Each class represents one real-world idea.
  + The main() method runs the tasks of the program.
* Java comments add helpful context to human readers.
* Java has whitespace, curly braces, and semicolons.
  + Whitespace is for humans to read code easily.
  + Curly braces mark the scope of a class and method.
  + Semicolons mark the end of a statement.
* Java is a compiled language.
  + Compiling catches mistakes in our code.
  + Compilers transform code into an executable class.

**Les 2**

The int data type allows values between -2,147,483,648 and 2,147,483,647, inclusive.

The double primitive data type can help. double can hold decimals as well as very large and very small numbers. The maximum value is 1.797,693,134,862,315,7 E+308, which is approximately 17 followed by 307 zeros. The minimum value is 4.9 E-324, which is 324 decimal places!

The char data type can hold any character, like a letter, space, or punctuation mark.

Char = single quotes in declaratie variabele

String = double quotes in declaratie variabele

# **Review**

Creating and filling variables is a powerful concept that allows us to keep track of all kinds of data in our program.

In this lesson, we learned how to create and print several different data types in Java, which you’ll use as you create bigger and more complex programs.

We covered:

* int, which stores whole numbers.
* double, which stores bigger whole numbers and decimal numbers.
* boolean, which stores true and false.
* char, which stores single characters using single quotes.
* String, which stores multiple characters using double quotes.
* *Static typing*, which is one of the safety features of Java.
* Variable naming conventions.

Practice declaring variables and assigning values to make sure you have a solid foundation for learning more complicated and exciting Java concepts!

**Les 3**

# **Review**

What’s the use of having variables if you can’t do anything with them? We’ve now seen some ways you can operate on variables and compare them. The possibilities are endless!

We covered:

* Addition and subtraction, using + and -
* Multiplication and division, using \* and /
* The modulo operator for finding remainders, %
* Greater than, >, and less than, <
* Equal to, ==, and not equal to, !=
* Greater than or equal to, >=, and less than or equal to, <=
* equals() for comparing Strings and other objects
* Using + to concatenate Strings

Practice some of these concepts here, to make sure you have a solid foundation for learning more complicated and exciting Java concepts!

**Les 4 Classes**

A *class* is the set of instructions that describe how an instance can behave and what information it contains.

Nieuwe class:

Classname (=type) variable name (=name) new Classname()

# **Classes: Instance Fields**

Our last exercise ended with printing an instance of Store, which looked something like Store@6bc7c054. The first part, Store, refers to the class, and the second part @6bc7c054 refers to the instance’s location in the computer’s memory.

We don’t care about memory location, but our instances have no other characteristics!

We’ll add associated data to an object by introducing *instance variables*, or *instance fields*. Instance fields are the state in our objects.

# **Classes: Constructor Parameters**

We’ll use a combination of constructor method and instance field to create instances with individual state.

We need to alter the constructor method because now it needs to access data we’re assigning to the instance.

Our Car constructor now has a *parameter*: String carColor.

public class Car { String color; // constructor method with a parameter public Car(String carColor) { // parameter value assigned to the field color = carColor; } public static void main(String[] args) { // program tasks }}

We need a value for the instance field color, so we’ve added String carColor as a parameter.

Parameters specify the type and name of data available for reference within a method’s scope.

We’ve already seen a parameter in the main() method: String[] args, but this is the first time we’re using the parameter value within a method body.

The parameter carColor references the value passed in during a method call:

new Car("blue");// carColor references "blue" inside constructornew Car("yellow");// carColor references "yellow" inside constructor

Within the constructor, we assign the parameter value to the instance field.

Instance fields are available for assignment inside the constructor because we declared them within the class.

STATE = variabele die je in de class zelf declareert. Die kun je gebruiken om instances met een eigen state te maken. Deze state krijgt de nieuwe class via de aanroep van de constructor in de main method. De state bestaat in feite uit de properties/value-paren die een instance heeft gekregen.

Classname nieuweclassnaam = new Classname(aanroep met waarde per parameter)

**Behaviour = methods**

**JAVA: INTRODUCTION TO CLASSES**

# **Classes: Review**

Java is an object-oriented programming language where every program has at least one class. Programs are often built from many classes and objects, which are the instances of a class.

Classes define the state and behavior of their instances. Behavior comes from methods defined in the class. State comes from instance fields declared inside the class.

Classes are modeled on the real-world things we want to represent in our program. Later we will explore how a program can be made from multiple classes. For now, our programs are a single class.

public class Dog { // instance field String breed; // constructor method public Dog(String dogBreed) { /\* value of parameter dogBreed assigned to instance field breed \*/ breed = dogBreed; } public static void main(String[] args) { /\* create instance: use 'new' operator and invoke constructor \*/ Dog fido = new Dog("poodle"); /\* fields are accessed using: the instance name, `.` operator, and the field name. \*/ fido.breed; // "poodle" }}

**Instructions**

The text editor contains a Dog class. Play around with the code!

Try to add and remove instance fields. Create instances with different values. Access and print different fields.

**Les 5 Methods**

Void = als een method niets retourneert als deze wordt aangeroepen!

Je moet wel aangeven wat een method retourneert,

public int numberOfTires() { int tires = 4; return tires; }

Met de ToString method

public String toString(){

return "This store sells " + productType + " at a price of " + price +".";

}

Kun je de waarden van een object uitprinten

Gebruik je hem niet, dan krijg je alleen de instances te zien, zoals ze zijn opgeslagen in de vorm van een soort ID:

Store@2aae9190Store@2f333739

# **Review**

Great work! Methods are a powerful way to abstract tasks away and make them repeatable. They allow us to define behavior for classes, so that the Objects we create can do the things we expect them to. Let’s review everything we have learned about methods so far.

* *Defining a method* : Methods have a method signature that declares their return type, name, and parameters
* *Calling a method* : Methods are invoked with a . and ()
* *Parameters* : Inputs to the method and their types are declared in parentheses in the method signature
* *Changing Instance Fields* : Methods can be used to change the value of an instance field
* *Scope* : Variables only exist within the domain that they are created in
* *Return* : The type of the variables that are output are declared in the method signature

As you move through more Java material, it will be helpful to frame the tasks you create in terms of methods. This will help you think about what inputs you might need and what output you expect.

Return is niet per se nodig om een variabele te kunnen benaderen. Je kunt hem in de global space declareren en dan aanpassen met een method. In de main method kun je dan de method aanroepen die de variabele aanpast en deze vervolgens printen met dotnotatie in de instance.

**Les 6 control flow**

**The first** condition to evaluate to true will have that code block run. Here’s an example demonstrating the order:

int testScore = 72;if (testScore >= 90) { System.out.println("A");} else if (testScore >= 80) { System.out.println("B");} else if (testScore >= 70) { System.out.println("C");} else if (testScore >= 60) { System.out.println("D");} else { System.out.println("F");}// prints: C

This chained conditional statement has two conditions that evaluate true. Because testScore >= 70 comes before testScore >= 60, only the earlier code block is run.

**Note:** Only one of the code blocks will run.

# **Review**

Before this lesson, our code executed from top to bottom, line by line.

Conditional statements add branching paths to our programs. We use conditionals to make decisions in the program so that different inputs will produce different results.

Conditionals have the general structure:

if (condition) { // consequent path} else { // alternative path}

Specific conditional statements have the following behavior:

* if-then:
  + code block runs if condition is true
* if-then-else:
  + one block runs if conditions is true
  + another block runs if condition is false
* if-then-else chained:
  + same as if-then but an arbitrary number of conditions
* switch:
  + switch block runs if condition value matches case value

**ins**

# Les 7 **Introduction to Conditional Operators**

**CONDITIONAL OPERATORS**

# **Review**

Conditional operators work on boolean values to simplify our code. They’re often combined with conditional statements to consolidate the branching logic.

Conditional-AND, &&, evaluates to true if the booleans on *both sides* are true.

if (true && false) { System.out.println("You won't see me print!");} else if (true && true) { System.out.println("You will see me print!");}

Conditional-OR, ||, evaluates to true if one or both of the booleans on either side is true.

if (false || false) { System.out.println("You won't see me print!");} else if (false || true) { System.out.println("You will see me print!");}

Logical-NOT, !, evaluates to the opposite boolean value to which it is applied.

if (!false) { System.out.println("You will see me print!");}

**Les** at **8 LEARN JAVA: ARRAYS**

Array heeft { } in Java. Krijgt ter predefinitie []

Bij method geen = ter declaratie

Java print een array niet zomaar uit: [Ljava.lang.String;@2aae9190s

# **Importing Arrays**

When we printed out the array we created in the last exercise, we saw a memory address that did not help us understand what was contained in the array.

If we want to have a more descriptive printout of the array itself, we need a toString() method that is provided by the Arrays *package* in Java.

import java.util.Arrays;

We put this at the top of the file, before we even define the class!

When we import a package in Java, we are making all of the methods of that package available in our code.

The Arrays package has many useful methods, including Arrays.toString(). When we pass an array into Arrays.toString(), we can see the contents of the array printed out:

# **Get Element By Index**

Now that we have an array declared and initialized, we want to be able to get values out of it.

We use square brackets, [ and ], to access data at a certain index:

double[] prices = {13.1, 15.87, 14.22, 16.66};System.out.println(prices[1]);

This command would print:

15.87

Because 15.87 is the item at the 1 index of the array. Remember that arrays start at index 0!

# **Creating an Empty Array**

We can also create empty arrays and then fill the items one by one. Empty arrays have to be initialized with a fixed size:

String[] menuItems = new String[5];

Once you declare this size, it cannot be changed! This array will always be of size 5.

After declaring and initializing, we can set each index of the array to be a different item:

menuItems[0] = "Veggie hot dog";menuItems[1] = "Potato salad";menuItems[2] = "Cornbread";menuItems[3] = "Roasted broccoli";menuItems[4] = "Coffee ice cream";

This group of commands has the same effect as assigning the entire array at once:

String[] menuItems = {"Veggie hot dog", "Potato salad", "Cornbread", "Roasted broccoli", "Coffee ice cream"};

We can also change an item after it has been assigned! Let’s say this restaurant is changing its broccoli dish to a cauliflower one:

menuItems[3] = "Baked cauliflower";

Now, the array looks like:

["Veggie hot dog", "Potato salad", "Cornbread", "Baked cauliflower", "Coffee ice cream"]

The args parameter is another example of a String array. In this case, the array args contains the arguments that we pass in from the terminal when we run the class file. (So far, args has been empty.)

So how can you pass arguments to main()? Let’s say we have this class HelloYou:

public class HelloYou { public static void main(String[] args) { System.out.println("Hello " + args[0]); }}

When we run the file HelloYou in the terminal with an argument of "Laura":

java HelloYou Laura

We get the output:

Hello Laura

The String[] args would be interpreted as an array with one element, "Laura".

When we use args[0] in the main method, we can access that element like we did in HelloYou.

Args is de verbinding tussen terminal input en java output

# **Review**

We have now seen how to store a list of values in arrays. We can use this knowledge to make organized programs with more complex variables.

Throughout the lesson, we have learned about:

* Creating arrays explicitly, using { and }.
* Accessing an index of an array using [ and ].
* Creating empty arrays of a certain size, and filling the indices one by one.
* Getting the length of an array using length.
* Using the argument array args that is passed into the main() method of a class.

Lengte van een nieuwe array bepalen:

double[] mathScores = new double[4];

**Les 9 ARRAYLISTS**

Nodig om waarden te kunnen toevoegen en verwijderen in Java.

# **Introduction**

When we work with arrays in Java, we’ve been limited by the fact that once an array is created, it has a fixed size. We can’t add or remove elements.

But what if we needed to add to the book lists, newsfeeds, and other structures we were using arrays to represent?

To represent dynamic lists, we can use Java’s ArrayLists. ArrayLists allow us to:

* Store elements of the same type (just like arrays)
* Access elements by index (just like arrays)
* Add elements
* Remove elements

Remember how we had to import java.util.Arrays in order to use additional array methods? To use an ArrayList at all, we need to import them from Java’s util package as well:

import java.util.ArrayList;

Let’s learn how to make use of this powerful object…

# **Creating ArrayLists**

To create an ArrayList, we need to declare the type of objects it will hold, just as we do with arrays:

ArrayList<String> babyNames;

We use angle brackets < and > to declare the type of the ArrayList. These symbols are used for *generics*. Generics are a Java construct that allows us to define classes and objects as parameters of an ArrayList. For this reason, we can’t use primitive types in an ArrayList:

// This code won't compile:ArrayList<int> ages;// This code will compile:ArrayList<Integer> ages;

The <Integer> generic has to be used in an ArrayList instead. You can also use <Double> and <Char> for types you would normally declare as doubles or chars.

We can initialize to an empty ArrayList using the new keyword:

// Declaring:ArrayList<Integer> ages;// Initializing:ages = new ArrayList<Integer>();// Declaring and initializing in one line:ArrayList<String> babyNames = new ArrayList<String>();

# Zoekresultaten

## Samenvatting van internet

A **Java** both **int and Integer** are used to store **integer** type data the major **difference** between both is type of **int** is primitive while **Integer** is of class type. ... **int** helps in storing **integer** value into memory. **Integer** helps in converting **int** into object and to convert an object into **int** as per requirement.16 sep. 2019

# **Adding an Item**

Now we have an empty ArrayList, but how do we get it to store values?

ArrayList comes with an add() method that takes an argument to add to the end of the ArrayList:

# **ArrayList Size**

Let’s say we have an ArrayList that stores items in a user’s online shopping cart. As the user navigates through the site and adds items, their cart grows bigger and bigger.

If we wanted to display the number of items in the cart, we could find the size of it using the size() method:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");System.out.println(shoppingCart.size());// 1 is printedshoppingCart.add("Tweed Houndstooth Hat");System.out.println(shoppingCart.size());// 2 is printedshoppingCart.add("Magnifying Glass");System.out.println(shoppingCart.size());// 3 is printed

In dynamic objects like ArrayLists, it’s important to know how to access the amount of objects we have stored.

# **Accessing an Index**

With arrays, we can use bracket notation to access a value at a particular index:

double[] ratings = {3.2, 2.5, 1.7};System.out.println(ratings[1]);

This code prints 2.5, the value at index 1 of the array.

For ArrayLists, bracket notation won’t work. Instead, we use the method get() to access an index:

ArrayList<String> shoppingCart = new ArrayList<shoppingCart>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");System.out.println(shoppingCart.get(2));

This code prints "Magnifying Glass", which is the value at index 2 of the ArrayList.

# **Changing a Value**

When we were using arrays, we could rewrite entries by using bracket notation to reassign values:

String[] shoppingCart = {"Trench Coat", "Tweed Houndstooth Hat", "Magnifying Glass"};shoppingCart[0] = "Tweed Cape";// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth Hat", "Magnifying Glass"]

ArrayList has a slightly different way of doing this, using the set() method:

ArrayList<String> shoppingCart = new ArrayList<shoppingCart>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.set(0, "Tweed Cape");// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth Hat", "Magnifying Glass"]

# **Removing an Item**

What if we wanted to get rid of an entry altogether? For arrays, we would have to make a completely new array without the value.

Luckily, ArrayLists allow us to remove an item by specifying the index to remove:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.remove(1);// shoppingCart now holds ["Trench Coat", "Magnifying Glass"]

We can also remove an item by specifying the value to remove:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.remove("Trench Coat");// shoppingCart now holds ["Tweed Houndstooth Hat", "Magnifying Glass"]

**Note:** This command removes the FIRST instance of the value "Trench Coat".

# **Getting an Item's Index**

What if we had a really large list and wanted to know the position of a certain element in it? For instance, what if we had an ArrayList detectives with the names of fictional detectives in chronological order, and we wanted to know what position "Fletcher" was.

// detectives holds ["Holmes", "Poirot", "Marple", "Spade", "Fletcher", "Conan", "Ramotswe"];System.out.println(detectives.indexOf("Fletcher"));

This code would print 4, since "Fletcher" is at index 4 of the detectives ArrayList.

# **Review**

Nice work! You now know the basics of ArrayLists including:

* Creating an ArrayList.
* Adding a new ArrayList item using add().
* Accessing the size of an ArrayList using size().
* Finding an item by index using get().
* Changing the value of an ArrayList item using set().
* Removing an item with a specific value using remove().
* Retrieving the index of an item with a specific value using indexOf().

Now if only there were some way to move through an array or ArrayList, item by item…

, "table tennis", "beach", "holidays", "frisbee", "tennis

Bij een arraylist hoef je bij uitprinten alleen maar het object (de instance) en met puntnotatie de arraylist te printen:

System.out.println((hobby.lol));

Weet nog niet wanneer ik moet compilen met:

**Javac classname.java**

En wanneer ik kan runnen met:

**Java classname**

En wanneer ik in Codeacademy op:

**Run**

Moet klikken

**Oefenen met de geleerde concepten**

import java.util.ArrayList;

import java.util.Arrays;

class List {

String[] activities;

ArrayList<String> lol = new ArrayList<String>();

public List(String[] activity, ArrayList<String> plezier) {

activities = activity;

lol = plezier;

}

public static void main(String[] args) {

System.out.println("hello world");

String[] weather = {"rain", "sun", "temperature", "clouds", "thunder", "moisture", "heat"};

String[] hobbies = {"soccer", "music", "tennis"};

System.out.println(Arrays.toString(weather));

System.out.println("test");

ArrayList<String> fun = new ArrayList<String>();

fun.add("soccer");

fun.add("tennis");

fun.add("pingpong");

fun.add("vrijworstelen");

System.out.println(fun.size());

List hobby = new List (hobbies, fun);

System.out.println("tesssst");

System.out.println((hobby.lol));

System.out.println((hobby.lol.get(1)));

fun.remove(3);

System.out.println((hobby.lol));

fun.set(2, "basketbal");

System.out.println((hobby.lol) + "derde element moet nu basketbal zijn");

System.out.println(fun.indexOf("tennis"));

}

}

# **Introduction to Loops**

In the programming world, we hate repeating ourselves. There are two reasons for this:

* Writing the same code over and over is time-consuming.
* Having less code means having less to debug.

// Importing the Random library

import java.util.Random;

class LuckyFive {

public static void main(String[] args) {

// Creating a random number generator

Random randomGenerator = new Random();

// Generate a number between 1 and 6

int dieRoll = randomGenerator.nextInt(6) + 1;

// Repeat while roll isn't 5

}

}

# **For-Each Loops**

Sometimes we need access to the elements’ indices or we only want to iterate through a portion of a list. If that’s the case, a regular for loop is a great choice. But sometimes we couldn’t care less about the indices; we only care about the element itself. At times like these, for-each loops come in handy.

*For-each loops* allow you to directly loop through each item in a list of items (like an array or ArrayList) and perform some action with each item. The syntax looks like this:

for (String inventoryItem : inventoryItems) { System.out.println(inventoryItem);}

We can read the : as “in” like this: for each inventoryItem (which should be a String) in inventoryItems, print inventoryItem.

Note that we can name the individual item whatever we want; using the singular of a plural is just a convention. You may also encounter conventions like String word : sentence.

# **Review**

Nice work! Let’s iterate over what you’ve just learned about loops:

* while loops: These are useful to repeat a code block an unknown number of times until some condition is met. For example:

int wishes = 0;while (wishes < 3) { // code that will run wishes++;}

* for loops: These are ideal for when you are incrementing or decrementing with a counter variable. For example:

for (int i = 0; i < 5; i++) { // code that will run}

* For-each loops: These make it simple to do something with each item in a list. For example:

for (String inventoryItem : inventoryItems) { // do something with each inventoryItem}

**String methods**

As you may recall, a String, which is widely used in Java, is an object that represents a sequence of characters. It is a great way to store information.

Wel haakjes zetten!!

System.out.println(tweet.length());

The concat() method concatenates one string to the end of another string. Concatenation is the operation of joining two strings together.

Suppose we have a String called str1 and another String called str2, using str1.concat(str2) would return str1 with str2 appended to the end of it.

For example:

String name = "Code";name = name.concat("cademy");System.out.println(name);

Codecademy would be printed.

In the code block above, we changed the value of name by reassigning it with a new value. However, concat() doesn’t actually change the value of the original string.

Suppose we do something slightly different:

String name = "Code";name.concat("cademy");System.out.println(name);

Code would be printed instead.

# **equals()**

With objects, such as Strings, we can’t use the primitive equality operator == to check for equality between two strings. To test equality with strings, we use a built-in method called equals().

For example:

String flavor1 = "Mango";String flavor2 = "Peach";System.out.println(flavor1.equals("Mango"));// prints trueSystem.out.println(flavor2.equals("Mango"));// prints false

Side note, there’s also a equalsIgnoreCase() method that compares two strings without considering upper/lower cases.

# **indexOf()**

If we want to know the index of the first occurence of a character in a string, we can use the indexOf() method on a string.

Remember that the indices in Java start with 0:

String letters = "ABCDEFGHIJKLMN";System.out.println(letters.indexOf("C"));

This would output 2.

Although C is the third letter in the English alphabet, it is located in the second index of the string.

Suppose we want to know the index of the first occurrence of an entire substring. The indexOf() instance method can also return where the substring begins (the index of the first character in the substring):

String letters = "ABCDEFGHIJKLMN";System.out.println(letters.indexOf("EFG"));

This would output 4, because EFG starts at index 4.

If the indexOf() doesn’t find what it’s looking for, it’ll return a -1.

# **charAt()**

The charAt() method returns the character located at a String‘s specified index.

For example:

String str = "qwer";System.out.println(str.charAt(2));

It would output e because that’s what’s at index 2. (Once again, indices start with 0.)

Suppose we try return the character located at index 4, it would produce an error because index 4 is out of str‘s range.

# **substring()**

There may be times when we only want a part of a string. In such cases, we may want to extract a *substring* from a string.

The substring() method does exactly that. For example:

String line = "The Heav'ns and all the Constellations rung";System.out.println(line.substring(23));

It would output Constellations rung because that’s what begins at index 23 and ends at the end of line. The substring begins with the character at the specified index and extends to the end of the string.

But suppose we want a substring at the middle of the string. We can include two arguments with this string method. For example:

String line = "The Heav'ns and all the Constellations rung";System.out.println(line.substring(23, 38));

It would output Constellations because that’s the substring that begins at index 23 and ends at index 38.

# **toUpperCase() / toLowerCase()**

There will be times when we have a word in a case other than what we need it in. Luckily, Java has a couple String methods to help us out:

* toUpperCase(): returns the string value converted to uppercase
* toLowerCase(): returns the string value converted to lowercase

For example:

String input = "Cricket!";String upper = input.toUpperCase();// stores "CRICKET!"String lower = input.toLowerCase();// stores "cricket!"

A good use of this functionality is to ensure consistency of the data you store in a database. Making sure all of the data you get from a user is lowercase before you store it in your database will make your database easier to search through later.

# **Review**

Congratulations! 🙌

We have learned some of the string methods that come with the String class:

* length()
* concat()
* indexOf()
* charAt()
* equals() / equalsIgnoreCase()
* substring()
* toUpperCase() / toLowerCase()

# **Introducing Inheritance**

When you hear the word “inheritance”, code may not be the first thing that springs to mind; you’re probably more likely to think of inheriting genetic traits, like eye color from your mother or a smile from your grandfather. But inheritance is also an important feature of object-oriented programming in Java.

Suppose we are building a Shape class in Java. We might give it some points in 2D, a method for calculating the area, and another method for displaying the shape. But what happens if we want a class for a triangle that has some triangle-specific methods? Do we need to redefine all of the same methods that we created for Shape?

No! (Phew.) Lucky for us, a Java class can also *inherit* traits from another class. Because a Triangle is a Shape, we can define Triangle so that it inherits fields and methods directly from Shape. The object-oriented principle of inheritance saves us the headache of redefining the same class members all over again.

There are several terms you’ll encounter frequently:

* *Parent class*, *superclass*, and *base class* refer to the class that another class inherits from (like Shape).
* *Child class*, *subclass*, and *derived class* refer to a class that inherits from another class (like Triangle).

# **Inheritance in Practice**

So how do we define a child class so that it inherits from a parent class? We use the keyword extends like this:

class Shape { // Shape class members}class Triangle extends Shape { // additional Triangle class members}

Now Triangle has inherited traits from Shape, meaning it copied over class members from Shape.

Until now, we’ve only been working with one class and one file. However, most Java programs utilize multiple classes, each of which requires its own file. Only one file needs a main() method — this is the file we will run.

Note: the various classes in our Java package — even though they are in different files — will have access to each other, so we can instantiate one class inside of another.

In een parent class kun je een nieuwe instance maken van een child class. Dan moet je de child class wel overerven.

# **Inheriting the Constructor**

Hang on, you might be thinking, if the child class inherits its parent’s fields and methods, does it also inherit the constructor? It does indeed, and sometimes this isn’t quite what we want.

Let’s say Shape has a numSides field that is set by passing an integer into the constructor. If we’re instantiating a Triangle, we would want that number to always be 3, so we’d want to modify the constructor to automatically assign numSides with a value of 3.

Can we do that?

As it happens, Java has a trick up its sleeve for just this occasion: using the super() method which acts like the parent constructor inside the child class constructor:

class Triangle extends Shape { Triangle() { super(3); } // additional Triangle class members}

By passing 3 to super(), we are making it possible to instantiate a Triangle without passing in a value for numSides.

Meanwhile, super() (behaving as Shape()) will shoulder the responsibility of setting numSides to 3 for our Triangle object.

It is also possible to just completely override a parent class constructor by writing a new constructor for the child class:

class Triangle extends Shape { Triangle() { this.numSides = 3; } // additional Triangle class members}

In the above example, numSides is just set to 3 without leveraging the parent class constructor.

Je kunt de constructor in de parent class wijzigen voor een child class zonder dat je dit doet voor alle child classes:

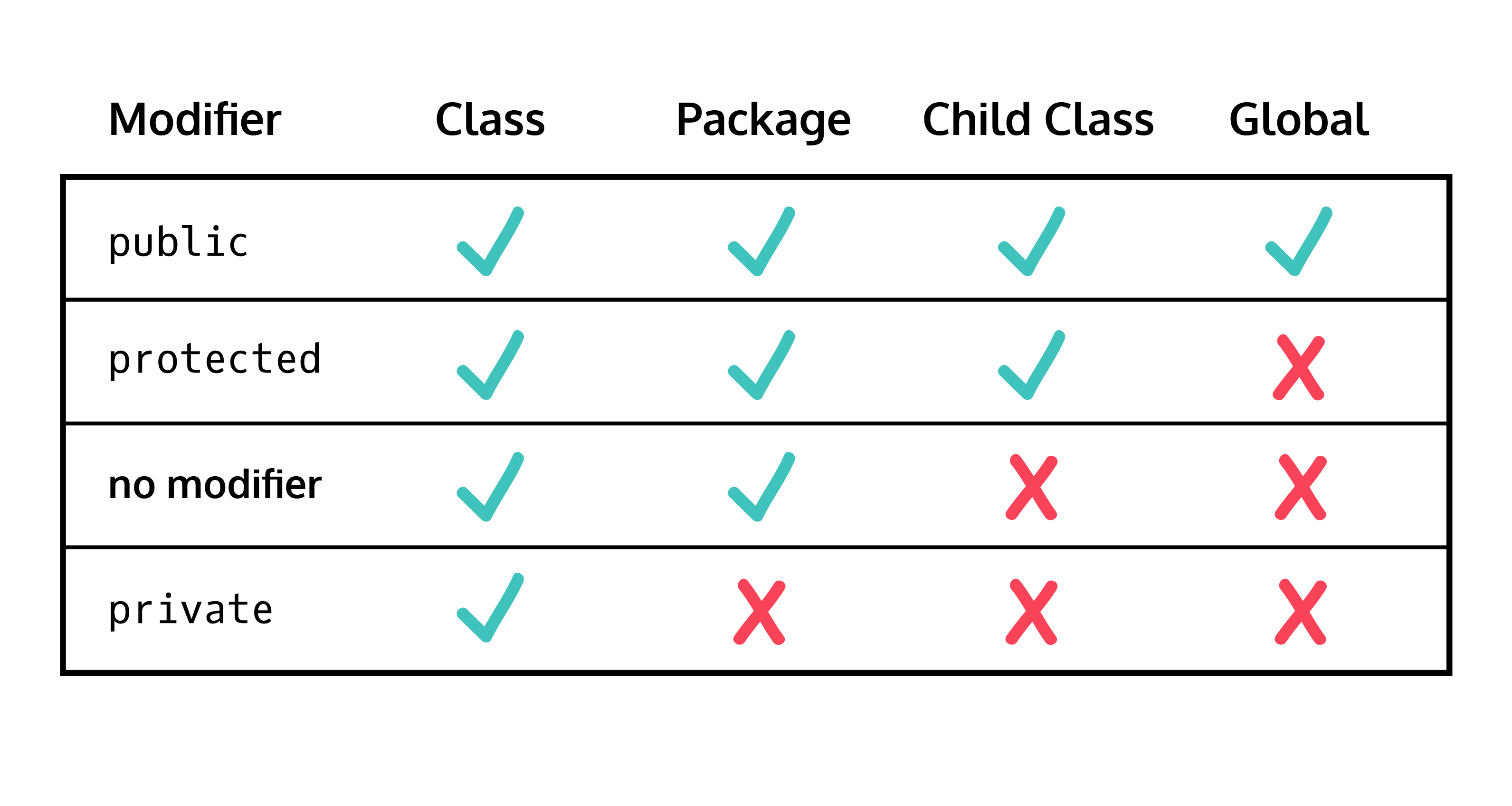
-via this

-via de super()-methode

SUPER IS MET EEN KLEINE s!!!!!!

# **Parent Class Aspect Modifiers**

You may recall that Java class members use private and public access modifiers to determine whether they can be accessed from outside the class. So does a child class inherit its parent’s private members?

Well, no. But there is another access modifier we can use to keep a parent class member accessible to its child classes and to files in the package it’s contained in — and otherwise private: protected.Here’s what protected looks like in use:

class Shape { protected double perimeter;}// any child class of Shape can access perimeter

In addition to access modifiers, there’s another way to establish how child classes can interact with inherited parent class members: using the final keyword. If we add final before a parent class method’s access modifier, we disallow any child classes from changing that method. This is helpful in limiting bugs that might occur from modifying a particular method.

# **Method Overriding**

One common use of polymorphism with Java classes is something we mentioned earlier — *overriding* parent class methods in a child class. Like the + operator, we can give a single method slightly different meanings for different classes. This is useful when we want our child class method to have the same name as a parent class method but behave a bit differently in some way.

Let’s say we have a BankAccount class that allows us to print the current balance. We want to build a CheckingAccount class that inherits the functionality of a BankAccount but with a modified printBalance() method. We can do the following:

class BankAccount { protected double balance; public void printBalance() { System.out.println("Your account balance is $" + balance); }}class CheckingAccount extends BankAccount { @Override public void printBalance() { System.out.println("Your checking account balance is $" + balance); }}

Notice that in order to properly override printBalance(), in CheckingAccount the method has the following in common with the corresponding method in BankAccount:

* Method name
* Return type
* Number and type of parameters

You may have also noticed the @Override keyword above printBalance() in CheckingAccount. This annotation informs the compiler that we want to override a method in the parent class. If the method doesn’t exist in the parent class, we’ll get a helpful error when we compile the program.

# **Using a Child Class as its Parent Class**

An important facet of polymorphism is the ability to use a child class object where an object of its parent class is expected.

One way to do this explicitly is to instantiate a child class object as a member of the parent class. We can instantiate a CheckingAcount object as a BankAccount like this:

BankAccount kaylasAccount = new CheckingAcount(600.00);

We can use kaylasAccount as if it were an instance of BankAccount, in any situation where a BankAccount object would be expected. (This would be true even if kaylasAccount were instantiated as a CheckingAccount, but using the explicit child as parent syntax is most helpful when we want to declare objects in bulk.)

It is important to note here that the compiler just considers kaylasAccount to be any old BankAccount. But because method overriding is handled at runtime, if we call printBalance(), we’ll see something CheckingAccount specific:

Your checking account balance is $600.00

This is because at runtime, kaylasAccount is recognized as the CheckingAccount it is. So, what if CheckingAccount has a method transferToSavings() that BankAccount does not have? Can kaylasAccount still use that method?

Well, no. The compiler believes that kaylasAccount is just a BankAccount that doesn’t have some fancy child class transferToSavings() method, so it would throw an error.

private void makeNoodles(Noodle noodle, String sauce) {

noodle.cook();

System.out.println("Mixing " + noodle.texture + " noodles made from " + noodle.ingredients + " with " + sauce + ".");

System.out.println("Dinner is served!");

}

Methode ontvangt een noodle-object als parameter.

**Lastige opgave om te vatten!**

If you take a look at the Dinner class, you’ll see there’s a makeNoodles() method that accepts a Noodle object as an argument.

In main(), instantiate a BiangBiang object as a Noodle called biangBiang.

Then call makeNoodles() on noodlesDinner with the following arguments:

* biangBiang
* "soy sauce and chili oil"

*Dinner.java*

class Dinner {

private void makeNoodles(Noodle noodle, String sauce) {

noodle.cook();

System.out.println("Mixing " + noodle.texture + " noodles made from " + noodle.ingredients + " with " + sauce + ".");

System.out.println("Dinner is served!");

}

public static void main(String[] args) {

Dinner noodlesDinner = new Dinner();

// Add your code here:

Noodle biangBiang = new BiangBiang();

noodlesDinner.makeNoodles(biangBiang, "soy sauce and chili oil");

}

}

*Noodle.Java*

class Noodle {

protected double lengthInCentimeters;

protected double widthInCentimeters;

protected String shape;

protected String ingredients;

protected String texture = "brittle";

Noodle(double lenInCent, double wthInCent, String shp, String ingr) {

this.lengthInCentimeters = lenInCent;

this.widthInCentimeters = wthInCent;

this.shape = shp;

this.ingredients = ingr;

}

public void cook() {

this.texture = "cooked";

}

}

*BiangBiang.java*

class Noodle {

protected double lengthInCentimeters;

protected double widthInCentimeters;

protected String shape;

protected String ingredients;

protected String texture = "brittle";

Noodle(double lenInCent, double wthInCent, String shp, String ingr) {

this.lengthInCentimeters = lenInCent;

this.widthInCentimeters = wthInCent;

this.shape = shp;

this.ingredients = ingr;

}

public void cook() {

this.texture = "cooked";

}

}

# **Child Classes in Arrays and ArrayLists**

Usually, when we create an array or an ArrayList, the list items all need to be the same type. But polymorphism puts a new spin on what is considered the same type…

In fact, we can put instances of different classes that share a parent class together in an array or ArrayList! For example, let’s say we have a Monster parent class with a few child classes: Vampire, Werewolf, and Zombie. We can set up an array with instances of each:

Monster dracula, wolfman, zombie1;dracula = new Vampire();wolfman = new Werewolf();zombie1 = new Zombie();Monster[] monsters = {dracula, wolfman, zombie1};

We can even iterate through the list of items — regardless of subclass — and perform the same action with each item:

for (Monster monster : monsters) { monster.attack();}

In the code above, we were able to call attack() on each monster in monsters despite the fact that, in the for-each loop, monster is declared as the parent class type Monster.

# **Review of Inheritance and Polymorphism**

Excellent work! You’ve learned quite a bundle about inheritance and polymorphism in Java:

* A Java class can inherit fields and methods from another class.
* Each Java class requires its own file, but only one class in a Java package needs a main() method.
* Child classes inherit the parent constructor by default, but it’s possible to modify the constructor using super() or override it completely.
* You can use protected and final to control child class access to parent class members.
* Java’s OOP principle of polymorphism means you can use a child class object like a member of its parent class, but also give it its own traits.
* You can override parent class methods in the child class, ideally using the @Override keyword.
* It’s possible to use objects of different classes that share a parent class together in an array or ArrayList.

**Debugging**

**DEBUGGING**

# **Introduction to Bugs**

*“First actual case of bug being found.”*

The story goes that on September 9th, 1947, computer scientist [Grace Hopper](https://en.wikipedia.org/wiki/Grace_Hopper) found a moth in the Harvard Mark II computer’s log book and reported the world’s first literal computer bug. However, the term “bug”, in the sense of technical error, dates back at least to 1878 and with Thomas Edison.

On your programming journey, you are destined to encounter innumerable red errors. Some even say that debugging is over 75% of the development time. But what makes a programmer successful isn’t avoiding errors; it’s knowing how to find the solution.

In Java, there are many different ways of classifying errors, but they can be boiled down to three categories:

* **Syntax errors:** Errors found by the compiler.
* **Run-time errors:** Errors that occur when the program is running.
* **Logic errors:** Errors found by the programmer looking for the causes of erroneous results.

Generally speaking, the errors become more difficult to find and fix as you move down the above list.

In this lesson, we will be looking at different errors and different error messages, and we’ll teach you how to think about errors in your code a little differently.

# **Syntax Errors**

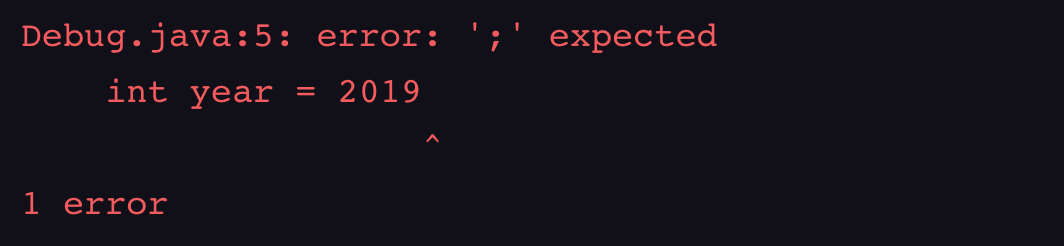
When we are writing Java programs, the compiler is our first line of defense against errors. It can catch syntax errors.

*Syntax errors* represent grammar errors in the use of the programming language. They are the easiest to find and correct. The compiler will tell you where it got into trouble, and its best guess as to what you did wrong.

Some common syntax errors are:

* Misspelled variable and method names
* Omitting semicolons ;
* Omitting closing parenthesis ), square bracket ], or curly brace }

Here’s an example of a syntax error message:



Usually the error is on the exact line indicated by the compiler, or the line just before it; however, if the problem is incorrectly nested braces, the actual error may be at the beginning of the nested block.

# **Run-time Errors**

If our program has no compile-time errors, it’ll run. This is where the fun really starts.

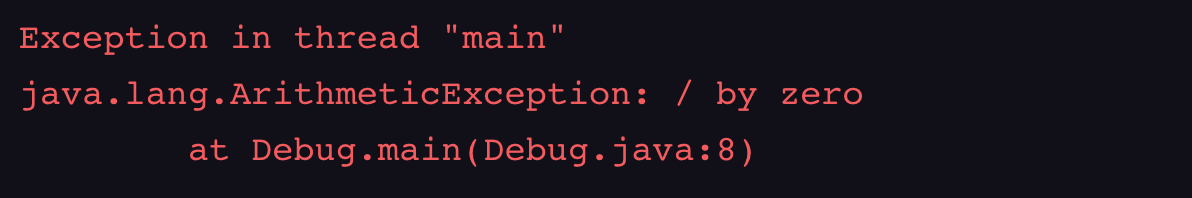
Errors which happen during program execution (run-time) after successful compilation are called run-time errors. *Run-time errors* occur when a program with no compile-time errors asks the computer to do something that the computer is unable to reliably do.

Some common run-time errors:

* Division by zero also known as division error
* Trying to open a file that doesn’t exist

There is no way for the compiler to know about these kinds of errors when the program is compiled.

Here’s an example of a run-time error message:



# **Exceptions**

In the last exercise when we were dealing with run-time errors, you might’ve noticed a new word in the error message: “Exception”.

Java uses *exceptions* to handle errors and other exceptional events. Exceptions are the conditions that occur at runtime and may cause the termination of the program.

When an exception occurs, Java displays a message that includes the name of the exception, the line of the program where the exception occurred, and a *stack trace*. The stack trace includes:

* The method that was running
* The method that invoked it
* The method that invoked that one
* and so on…

Make sure to examine it.

Some common exceptions that you will see in the wild:

* ArithmeticException: Something went wrong during an arithmetic operation; for example, division by zero.
* NullPointerException: You tried to access an instance variable or invoke a method on an object that is currently null.
* ArrayIndexOutOfBoundsException: The index you are using is either negative or greater than the last index of the array (i.e., array.length-1).
* FileNotFoundException: Java didn’t find the file it was looking for.

# **Exception Handling**

Exception handling is an essential feature of Java programming that allows us to use run-time error exceptions to make our debugging process a little easier.

One way to handle exceptions is using the try/catch:

* The try statement allows you to define a block of code to be tested for errors while it is being executed.
* The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs, though you can also catch several types of exceptions in a single block:

try { // Block of code to try} catch (NullPointerException e) { // Print the error message like this: System.err.println("NullPointerException: " + e.getMessage()); // Or handle the error another way here}

Notice how we used System.err.println() here instead of System.out.println(). System.err.println() will print to the standard error and the text will be in red.

You can also chain exceptions together:

try { // Block of code to try} catch (NullPointerException e) { // Code to handle a NullPointerException} catch (ArithmeticException e) { // Code to handle an ArithmeticException}

You can learn more about exceptions and handling them [here](https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html).

# **Logic Errors**

Once we have removed the syntax errors and run-time errors, the program runs successfully. But sometimes, the program still doesn’t do what we want it to do or no output is produced. Hmmm…

These types of errors which provide incorrect output, but appears to be error-free, are called *logic errors*. Logic errors occur when there is a design flaw in your program. These are some of the most common errors that happen to beginners and also usually the most difficult to find and eliminate.

Because logical errors solely depend on the logical thinking of the programmer, your job now is to figure out why the program didn’t do what you wanted it to do.

Some common logic errors:

* Program logic is flawed
* Some “silly” mistake in an if statement or a for/while loop

**Note:** Logic errors don’t have error messages. Sometimes, programmers use a process called [test-driven development (TDD)](https://en.wikipedia.org/wiki/Test-driven_development), a way to give logic errors error messages and save yourself a lot of headaches!

# **Debugging Techniques**

If you have examined the code thoroughly, and you are sure the compiler is compiling the right source file, it is time for desperate measures:

**1. Divide and conquer:** Comment out or temporarily delete half the code to isolate an issue.

* If the program compiles now, you know the error is in the code you deleted. Bring back about half of what you removed and repeat.
* If the program still doesn’t compile, the error must be in the code that remains. Delete about half of the remaining code and repeat.

Tip: In most code editors, one can highlight a block of code and use the keyboard shortcut command + / to comment it out.

**2. Print statements for the rescue:** Use System.out.println() to check variable/return values at various points throughout the program.

A lot of the time with logic errors, there was a flawed piece of logic, a miscalculation, a missing step, etc. By printing out the values at different stages of the execution flow, you can then hopefully pinpoint where you made a mistake.

# **Review**

Finding bugs is a huge part of a programmer’s life. Don’t be intimidated by them… embrace them. Errors in your code mean you’re trying to do something cool!

In this lesson, we have learned about the three types of Java errors:

* **Syntax errors:** Errors found by the compiler.
* **Run-time errors:** Errors found by checks in a running program.
* **Logic errors:** Errors found by the programmer looking for the causes of erroneous results.

Remember, [Google](https://www.google.com/) and [Stack Overflow](https://www.stackoverflow.com/) are a programmer’s best friends. For some more motivation, check out this blog post: [Thinking About Errors in Your Code Differently](https://news.codecademy.com/errors-in-code-think-differently).

Sometimes once you’ve tracked down a bug, you might still be confused on how to fix it! Whenever you want to know more about how Java works and what it can do, the best place to go is documentation. You can find the Java documentation at [Oracle](https://docs.oracle.com/javase/tutorial/index.html).

**Interessant programma voor opdracht**

import java.util.\*;

public class AreaCalculator {

public static void main(String[] args) {

Scanner keyboard = new Scanner(System.in);

System.out.println("Shape Area Calculator");

while(true) {

System.out.println();

System.out.println("-=-=-=-=-=-=-=-=-=-");

System.out.println();

System.out.println("1) Triangle");

System.out.println("2) Rectangle");

System.out.println("3) Circle");

System.out.println("4) Quit");

System.out.println();

System.out.print("Which shape: ");

int shape = keyboard.nextInt();

System.out.println();

if (shape == 1) {

area\_triangle(5,6);

} else if (shape == 2) {

area\_rectangle(4,5);

} else if (shape == 3) {

area\_circle(4);

} else if (shape == 4) {

quit();

break;

}

}

}

public static double area\_triangle(int base, int height) {

Scanner keyboard = new Scanner(System.in);

System.out.print("Base: ");

base = keyboard.nextInt();

System.out.print("Height: ");

height = keyboard.nextInt();

System.out.println();

int A = base \* height \* 0.5;

System.out.println("The area is " + A + ".");

return A;

}

public static int area\_rectangle(int length, int width){

Scanner keyboard = new Scanner(System.in);

System.out.print("Length: ");

length = keyboard.nextInt();

System.out.print("Width: ");

width = keyboard.nextInt();

System.out.println();

int A = length \* width;

System.out.println("The area is " + A + ".");

return A;

}

public static double area\_circle(int radius) {

Scanner keyboard = new Scanner(System.in);

System.out.print("Radius: ");

radius = keyboard.nextInt();

System.out.println();

double A = Math.PI \* radius \* radius;

System.out.println("The area is " + A + ".");

return A;

}

public static String quit() {

System.out.println("GoodBye");

return null;

}

}