JAVA FUNDAMENTALS

INTRODUCTION TO THE JAVA PROGRAMMING LANGUAGE

AGENDA

1. What is java?
2. Who created Java?
3. Why choose Java?
4. The Java Environment

WHAT IS JAVA ?

- Java is a computer programming language.
- **It enables programmers to write computer instructions using English-based commands instead of having to write in numeric codes.**
- It's known as a **high-level language** because it can be read and written easily by humans.
- Like English, Java has a set of rules that determine how the instructions are written.
- These rules are known as its **syntax**.
- Once a program has been written, the high-level instructions are translated into numeric codes that computers can understand and execute.

1. WHO CREATED JAVA ?

- In the early nineties, Java was created by a team led by James Gosling for Sun Microsystems.
- It was originally designed for use on digital mobile devices, such as cellphones.
- However, when Java 1.0 was released to the public in 1996, its focus had shifted to use on the internet.
- On April 20, 2009, Sun and Oracle Corporation announced that they had entered into a definitive agreement under which Oracle would acquire Sun.
- On January 27, 2010, Oracle announced that it had completed its acquisition of Sun Microsystems, making Sun a wholly owned subsidiary of Oracle.

2. WHY CHOOSE JAVA ?

Java was designed with a few key principles in mind:

- **Ease of Use:** The fundamentals of Java came from a programming language called C++.
  - Although C++ is a powerful language, it is complex in its syntax and inadequate for some of Java's requirements.
  - Java built on and improved the ideas of C++ to provide a programming language that was powerful and simple to use.
- **Reliability:** Java needed to reduce the likelihood of fatal errors from programmer mistakes.
  - With this in mind, **object-oriented programming** was introduced. When data and its manipulation were packaged together in one place, Java was robust.

3. WHY CHOOSE JAVA ?

- **Security:** As Java was originally targeting mobile devices that would be exchanging data over networks, it was built to include a high level of security.
  - Java is probably the most secure programming language to date.
- **Platform Independence:** Programs need to work regardless of the machines it is being executed on.
  - Java was written to be a portable language that doesn't care about the operating system or the hardware of the computer or device it is running on.

4. THE JAVA ENVIRONMENT

- **JDK (Java Development Kit)**
  - The software for programmers who want to write Java programs.
- **JVM (Java Virtual Machine)**
  - JVM is the **heart** of java programming language. When we run a program, JVM is responsible for converting byte code to the machine specific code.
  - Provides core java functions like memory management, garbage collection, security etc.
  - JVM is called virtual because it provides an interface that does not depend on the underlying operating system and machine hardware. **This independence from hardware and operating system is what makes java program write-once run-anywhere**.
- **JRE (Java Runtime Environment)**
  - The software for consumers who want to run Java programs.
  - JRE is the implementation of JVM, it provides platform to execute java programs.
  - JRE consists of JVM and java binaries and other classes to execute any program successfully. JRE doesn't contain any development tools like java compiler or debugger.

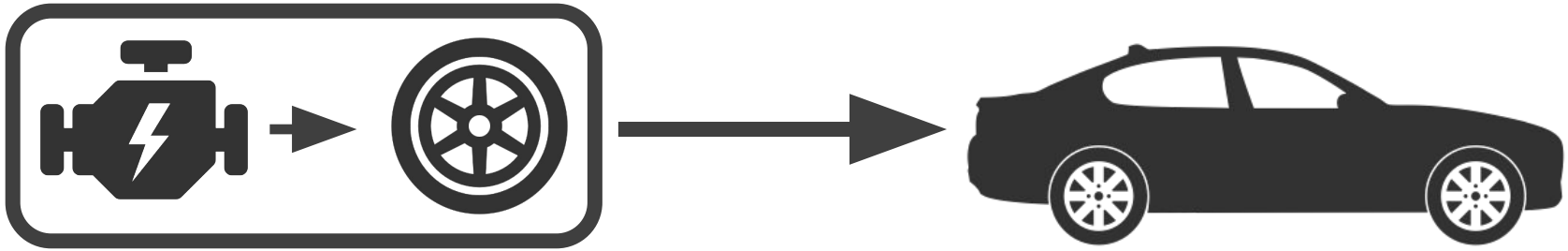
GETTING STARTED WITH THE JAVA PROGRAMMING LANGUAGE

AGENDA

1. Object oriented programming definition
2. Java class structure
3. The main() method
4. Primitive data types
5. Declaring and initializing variables
6. Default initialization of variables
7. Variable scope
8. Exercises

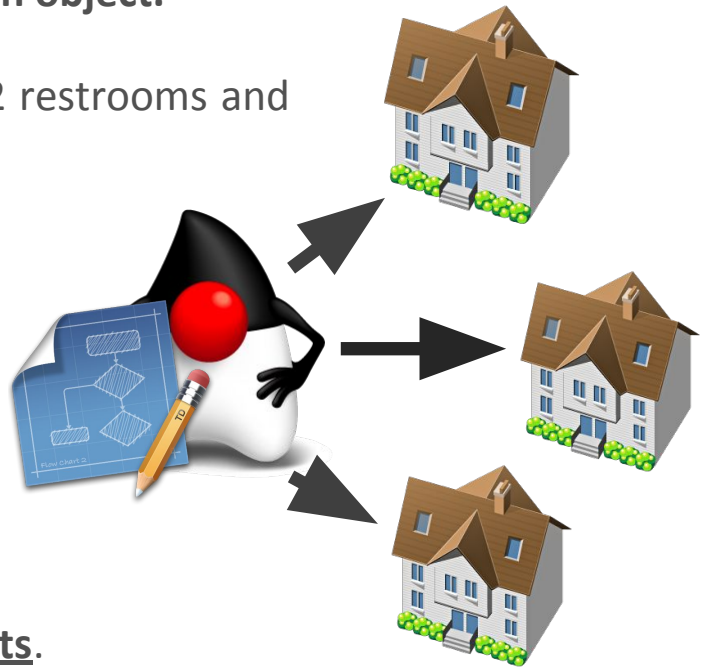
1. OBJECT ORIENTED PROGRAMMING DEFINITION

- Object Oriented Programming is a programming **paradigm** based on the concept of objects and how they interact with each other.
- A paradigm is a **synergy** of abstract concepts.
- The abstract concepts are: **Abstraction, Encapsulation, Inheritance and Polymorphism**.
- Don’t worry we will cover them in detail through the course.
- As we can see in the example presented below, the interaction between the engine object and the wheel object has the result of putting a third object, the car, in motion.



2. JAVA CLASS STRUCTURE | WHAT IS A CLASS ?

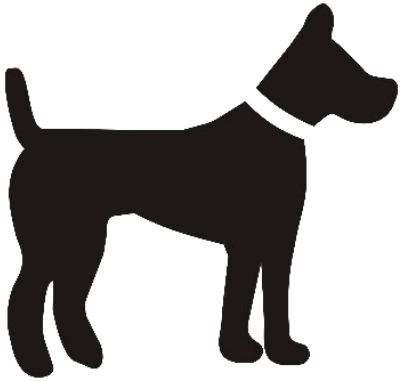
- In Java programs, classes are the basic building blocks.
- **A class is a <u>blueprint</u> from which individual objects are created. A class can contain fields and methods to describe the states and behaviors of an object.**
- To use most classes, we must create objects.
- Let's say we want to build a house with 3 bedrooms, 2 restrooms and one awesome living room.
- The first step in building our dream house is finding an architect who will design it. Luckily for us, we know Duke, so we kindly ask him for help.
- After Duke designs out house, we will take the plan to the construction company to develop it.
- But since Duke is also a very skilled entrepreneur, he realizes that he has the blueprint for our house, and he can use it to develop an entire neighborhood.
- **<u>The blueprint is the class and the houses are the objects</u>**.



2. JAVA CLASS STRUCTURE | WHAT IS AN OBJECT ?

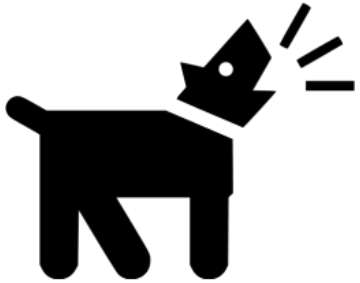
- **An object is an instance of a class.**
- Objects are characterized by **states** (fields) and **behaviors** (methods).

STATES

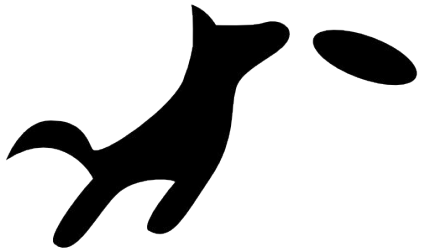


Name: Max
Color: Black

BEHAVIOURS



Barking



Playing

2. JAVA CLASS STRUCTURE | WHAT IS A VARIABLE ?

- Java classes contain two primary elements: behaviors which are known as methods, and states (fields), which are more generally known as **variables**.
- Variables hold the state of the program, and methods operate on that state.
- If the change is important to remember, a variable stores that change.
- A variable is just like a **bucket**.
- With a bucket you can carry water. After you dropped the water you can use the same bucket to carry flowers. After you finished with the flowers, your cat might want to get inside the bucket just because it can.
- **The contents of the bucket is what's varying**.

2. JAVA CLASS STRUCTURE | WHAT IS A VARIABLE ?

- In Java, every line of code must be written inside a class. The most basic class looks like the one from the right.
- In Java, a word with a special meaning is called a **keyword**.
- The **public** keyword on line 1 means that the class can be used by other classes.
- The **class** keyword indicates we're defining a class.
- **Dog** gives the name of the class.
- This class doesn't look very interesting, so let's add our first field.
- On line 2, we defined a variable named ***color***.
- We also defined the **type** of that variable to be a **String**.
- A **String** is a type that we can put text into, such as **"Hi! My name is Max"**.
- As we said that variables are like buckets, **in Java variables are specialized buckets**. Unfortunately, we can not carry both flowers and cats with the same bucket. Each of them can be carried only by special created buckets for flowers or for cats.
- **The type of a variable specifies what content it can hold.**

```
1. public class Dog {
2. }
```

```
1. public class Dog {
2.   String color;
3. }
```

2. JAVA CLASS STRUCTURE | WHAT IS A METHOD ?

- On lines 4-6, we've defined our first method.
- A method is **an operation that can be called.**
- **void** means that no value at all is returned.
- On lines 8-12 is another method.
- This method requires information be supplied.
- This information is called a **parameter**.
- eat has one parameter named nrOfSteaks, and it is of type **int**.
- This means the caller should pass in one **int** parameter and expect nothing to be returned.

```
1.  public class Dog {
2.
3.    String color;
4.
5.    public static void bark(){
6.       System.out.println("Woof-Woof");
7.    }
8.
9.    public static void eat(int
10.                              nrOfSteaks){
11.      System.out.println("Eating " +
12.               nrOfSteaks + " steaks");
13.   }
14. }
```

2. JAVA CLASS STRUCTURE | CLASSES VS. FILES

- Usually, each Java class is defined in its own ***.java** file.
- It is usually **public**, which specifies that any code can call it.
- The fun part is that Java does not require the class to be **public**, so this class is just fine.

```
1. class Dog {
2. }
```

- We can even put two classes in the same file.
- **When we do so, at most one of the classes in the file can be public.**

- That means a file containing the following is also fine:

```
1. public class Dog {
2.   String color;
3. }
4. class Cat {
5.   String name;
6. }
```

- **If we do have a public class, it needs to match the name of the file.**
- **public class Cat** would not compile in a file named `Dog.java`.
- We will discuss later what nonpublic access means.

3. THE MAIN METHOD

- In any Java program, the main method is the entry point of the program.
- Its syntax is always: `public static void main(String[] args)`.

```
1. public static void main(String[] args){
2.    System.out.println("Hello World!");
3. }
```

- Let's analyze the main method closely and try to understand each of its parts:
- **public**
  - Is the **access modifier** of the main method.
  - It must be **public** so that java can execute this method.
  - Don't worry, we will cover access modifiers in detail later.
- **static**
  - The keyword static binds a method to its class so it can be called by just the class name.
  - It must be **public** so that java can execute this method.
  - For example, `Test.main()`. Java doesn't have to create an object to call the `main()` method - which is nice since we haven't learned about creating objects just yet.
  - When the java runtime starts, there is no object of the class present. That's why main method must be **static**.

3. THE MAIN METHOD

- **`void`**
  - The keyword **`void`** represents the return type of the method.
  - **Used for methods that don't return anything.**
  - It's a good practice to use **`void`** for methods that change an object's state.
  - Considering this, **the `main()` method changes the program state from started to finished**.
- **`main`**
  - This is the name of java `main` method.
  - **It's fixed!**
  - When we start a java program, the JVM looks for the `main` method.

- **`String[] args`**
  - Java `main` method accepts a single argument of type `String` array.
  - This is also called as **command line arguments**.

```
public static void main(String[] args){
  System.out.println("Hello " + args[0]);
  System.out.println("Hello " + args[1]);
}
```

- Look at how relaxed Duke is!
- We will also be so relaxed as Duke, once we master the java programming art.

4. PRIMITIVE DATA TYPES

TYPE	CONTAINS	DEFAULT	SIZE	RANGE
`boolean`	true or false	false	1 bit	NA
`char`	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

- **`byte`**, **`short`**, **`int`**, and **`long`** are used for numbers without decimal points.
- **`float`** and **`double`** are used for floating-point values.
- A **`float`** requires the letter **f** following the number, so Java knows it is a **`float`**.
- Each numeric type uses twice as many bits as the smaller similar type.
- For example: **`short`** uses twice as many bits as **`byte`** does.

4. PRIMITIVE DATA TYPES

- There are a few more things we need to know about primitives.
- When a number is present in the code, it is called a **literal**.
- By default, Java assumes we are defining an **int** value with a literal.
- In this example, the number listed is bigger than what fits in an **int**.

```
1. // DOESN'T COMPILE
2. long variable = 7685940345;
```

Integer number is too large

- Java complains the number is out of range. And it is for an **int**. But wait… we don't have an **int**.
- The solution is to add the character **L** at the end of the number:

```
1. // NOW JAVA KNOWS IT'S A LONG
2. long variable = 7685940345L;
```

- Another way to specify numbers is to change the **base**.
- When we learned how to count, we used the digits 0–9.
- This numbering system is called **base 10** since there are 10 digits.
- It is also known as the **decimal number system**.
- Java allows us to specify digits in several other formats.

4. PRIMITIVE DATA TYPES

- **Binary (digits 0-1)**
  - Uses the digit `0` followed by `b` or `B` as a prefix.
  - Example: `0b10`.
- **Octal (digits 0-7)**
  - Uses the digit `0` as a prefix.
  - Example: `015`.
- **Hexadecimal (digits 0-9 and letters A-F)**
  - Uses the digit `0` followed by `x` or `X` as a prefix.
  - Example: `0x2A`.

```
1. System.out.println(21);   //21
2. System.out.println(0b10); //10
3. System.out.println(015);  //13
4. System.out.println(0x2A); //42
```

- The last thing we need to know about numeric literals is a feature added in Java 7.
- We can have underscores in numbers to make them easier to read:

```
1. int million1 = 1000000;
2. int million2 = 1_000_000;
```

- We can add underscores anywhere except at the beginning of a literal, at the end of a literal, right before a decimal point, right after a decimal point.

```
1. double notAtStart = _1000.00;
2. double notAtEnd = 1000.00_;
3. double notByDecimal = 1000._00;
4. double uglyButLegal = 1_00_0.0_0;
```

5. DECLARING AND INITIALIZING VARIABLES

- **Academic definition**: **A variable is a name for a piece of memory that stores data.**
- When we declare a variable, we need to state the variable type along with giving it a name.
- For example, the following code declares two variables:

```
1. String address;
2. int age;
```

- One is named `address` and is of type `String`.
- The other is named age and is of type **`int`**.
- Now that we've declared a variable, we can give it a value.
- This is called **initializing a variable**.
- To initialize a variable, we just type the variable name followed by an equal sign, followed by the desired value:

```
3. address = "221B Baker Street";
4. age = 30;
```

- Since we often want to initialize a variable right away, we can do so in the same statement as the declaration.
- For example, here we merge the previous declarations and initializations into more concise code:

```
1. String address = "221B Baker Street";
2. int age = 30;
```

5. DECLARING AND INITIALIZING VARIABLES

- We can also declare and initialize multiple variables in the same statement::

```
1. String s1, s2;
2. String s3 = "dog", s4 = "cat";
```

- Four `String` variables were declared: `s1`, `s2`, `s3`, and `s4`.
- We can declare many variables in the same declaration **if they all have the same type**.
- We can also initialize any or all those values in one line.
- In example presented above, we have two initialized variables: `s3` and `s4`.
- The other two variables are declared but are not yet initialized.

- This is where it gets a bit tricky:

```
1. int i1, i2, i3 = 50;
```

- As we should expect, three variables were declared: `i1`, `i2`, `i3`.
- However, only one of those values was initialized: `i3`.
- The other two remain declared but not initialized.
- That's the trick: **Each snippet separated by a comma is a little declaration of its own.**
- The initialization of `i3` only applies to `i3`.
- It doesn't have anything to do with `i1` or `i2` despite being in the same statement.

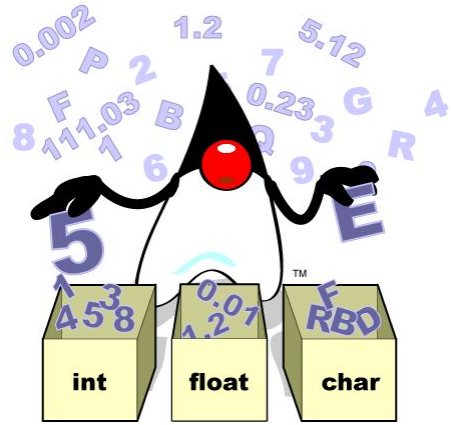
5. DECLARING AND INITIALIZING VARIABLES

- We should not be surprised that Java has precise rules about identifier names.
- Luckily for us, the same rules for identifiers apply to anything we are free to name, **like variables, methods, classes, and fields**.
- There are only three rules that we need to know:
    1. The name must begin with a letter or the symbol $ or _.
    2. Subsequent characters may also be numbers.
    3. You cannot use the same name as a Java reserved word. As we might imagine, a reserved word is a keyword that Java has reserved so that you are not allowed to use it.
- Remember that Java is case sensitive, so we can use versions of the keywords that only differ in case. But in practice it's not recommended.

5. DECLARING AND INITIALIZING VARIABLES

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

- Don’t worry, we won’t need to memorize the full list of reserved words.
- The compiler will tell us if we want to use a reserved keyword.



6. DEFAULT INITIALIZATION OF VARIABLES

- Before we can use a variable, it needs a value.
- Some types of variables get this value set automatically, and others require the programmer to specify it.
- Java has 3 types of variables: **local variables, instance variables and class variables.**
- **LOCAL VARIABLES**
  - A local variable is a variable defined within a method or a block of code.
  - **Local variables must be initialized before use.**
  - They do not have a default value and contain garbage data until initialized.
  - The compiler will not let us read an uninitialized value.

```
1.  public void notValid(){
2.    int a = 10;
3.    int b;
4.    // DOESN'T COMPILE
5.    int result = a + b;
6.  }
```

```
1.  public void valid(){
2.    int a = 10;
3.    int b; // b is declared here
4.    b = 3; // and initialized here
5.    int result = a + b;
6.  }
```

6. DEFAULT INITIALIZATION OF VARIABLES

- **INSTANCE AND CLASS VARIABLES**
  - Variables that are not local variables are known as instance variables or class variables.
  - **Instance variables are also called fields.**
  - **Class variables are shared across multiple objects.**
  - We can tell a variable is a class variable because it has the keyword `static` before it.
  - We are not required to initialize instance and class variables.
  - As soon as we declare these variables, they are given a default value.
  - Don't worry if these concepts are a bit fuzzy now. We will cover them in more detail later.

VARIABLE TYPE	DEFAULT INITIALIZATION VALUE
`boolean`	false
`byte, short, int, long`	0 (in the type's bit-length)
`float, double`	0.0 (in the type's bit-length)
`char`	'\u0000' (NUL)
All object references (everything else)	null

7. VARIABLE SCOPE

- How many local variables are in this example ?

```
1.  public void sleep(int bedTime){
2.    int nrOfStoriesToRead = 2;
3.  }
```

- There are two local variables in this method.
- **nrOfStoriesToRead** is declared inside the method.
- **bedTime** is called a method parameter. It is also local to the method.
- **Both variables have the scope local to the method.**
- **This means they cannot be used outside the method.**
- Local variables can never have a scope larger than the method they are defined in.

- However, they can have a smaller scope.

```
1.  public void sleepIfTired(boolean
2.                               isTired){
3.    if(isTired == true){
4.       int nrOfHours = 8;
5.    } // nrOfHours goes out of
6.     //  scope
7.    // DOESN'T COMPILE
8.    System.out.println(nrOfHours);
9.  }
```

- **isTired** has the scope of the entire method.
- **nrOfHours** has a smaller scope. It is only available for use in the if statement because it is declared inside of it.

7. VARIABLE SCOPE

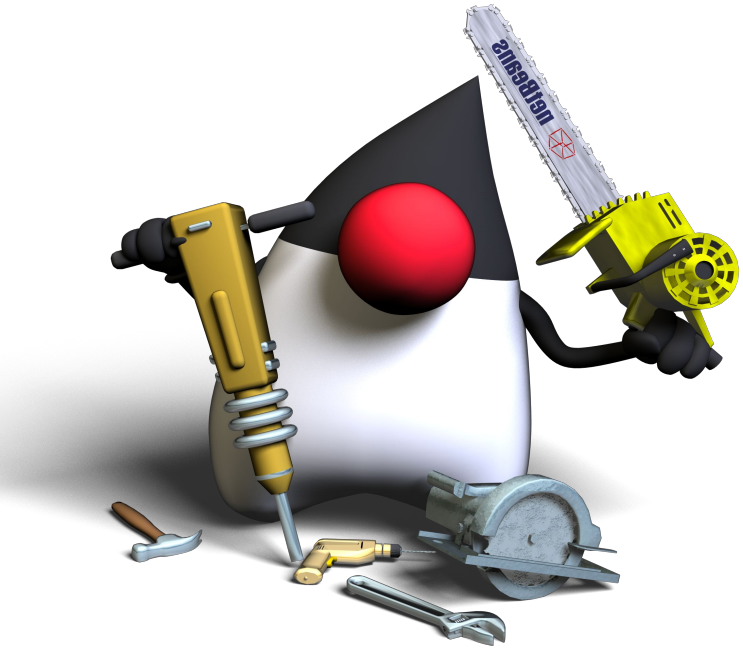
- Remember that a code block can contain other code blocks. The smaller blocks can use variables defined in the larger scoped blocks, but not vice versa.

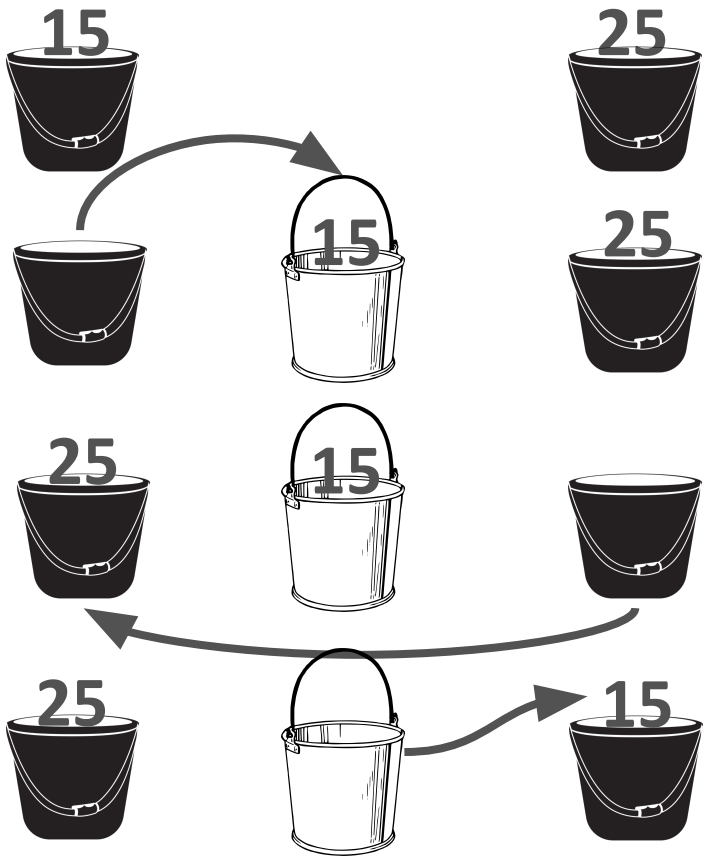
```
1.  public void sleepIfTired(boolean isTired){
2.    if(isTired == true){
3.    int nrOfHours = 8;
4.      {
5.        int nrOfStories = 2;
6.        System.out.println(nrOfStories);
7.        System.out.println(nrOfHours);
8.      }
9.      // DOESN'T COMPILE
10.     System.out.println(nrOfStories);
11.     System.out.println(nrOfHours);
12.   }
13. }
```

- The variable defined on line 3 is in scope until the block ends on line 11.
- The variable defined on line 5 is in scope until the block ends on line 7.
- Using it on line 10 is not allowed.
- If we understand what is the scope of a variable, we can say that we are almost as his bike.

8. EXERCISE

- Write a Java program that swaps two variables.
  - Duke's here ready to help us.
  - What a nice guy.





JAVA OPERATORS AND STATEMENTS

AGENDA

1. Understanding operators
2. Arithmetic operators
3. Numeric promotions
4. Unary operators
5. Binary operators
6. Java statements
7. Exercises

1. UNDERSTANDING OPERATORS

- A Java operator is **a special symbol** that can be applied to a set of variables or values - referred to as operands - and that returns a result.
- Three types of operators are available in Java: **unary, binary, and ternary.**
- These types of operators can be applied to one, two, or three operands, respectively.
- Java operators are not necessarily evaluated from left-to-right order.
- For example, the following Java expression is evaluated from right-to-left given the specific operators involved:

```
1. int y = 5;
2. double x = 2 + 2 * --y;
```

- In this example, we would first decrement y to 4, and then multiply by 2, and finally add 2.
- The value will then be automatically converted from 10 to 10.0 and assigned to the variable x.
- The final values of x and y will be 10.0 and 4.
- Unless overridden with parentheses, Java operators follow **order of operation**, by decreasing **order of operator precedence**.
- If two operators have the same level of precedence, then **Java guarantees left-to-right evaluation.**

1. UNDERSTANDING OPERATORS

ORDER OF PRECEDENCE

OPERATOR	SYMBOLS AND EXAMPLES
Post-unary operators	expression++, expression--
Pre-unary operators	++expression, --expression
Other unary operators	+, -, !
Multiplication/Division/Modulo	*, /, %
Addition/Subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	boolean expression ? expression1 : expression2
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

2. ARITHMETIC OPERATORS

OPERATOR	USE	DESCRIPTION
+	op1 + op2	Adds op1 and op2; also used to concatenate strings
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

- In the order of precedence table, the multiplicative operators (*, /, %) have a higher order of precedence than the additive operators (+, -).
- Given: `int x = 5 * 2 + 4 * 2 - 5; //What value will be assigned to x ?`
- Given: `int y = 5 * ((2 + 4) * 2 - 5); //What value will be assigned to y ?`

2. ARITHMETIC OPERATORS

- The modulo, or remainder operator (%) , is simply the remainder when two numbers are divided.
- For example, 9 divided by 3 divides evenly and has no remainder - therefore, the remainder, or 9 % 3, is 0.
- On the other hand, 11 divided by 3 does not divide evenly - therefore, the remainder, or 11 % 3, is 2.

```
1. System.out.println(9 / 3);  // 3
2. System.out.println(9 % 3);  // 0
3. System.out.println(10 / 3); // 3
4. System.out.println(10 % 3); // 1
5. System.out.println(11 / 3); // 3
6. System.out.println(11 % 3); // 2
7. System.out.println(12 / 3); // 4
8. System.out.println(12 % 3); // 0
```

- Note that the division results only increase when the value on the left-hand side goes divides equally with the divider.
- Whereas the modulo remainder value increases by 1 each time the left-hand side is increased until it wraps around to zero.
- For a given divider y, which is 3 in these examples, the modulo operation results in a value between 0 and (y - 1) .
- This means that the result of a modulo operation is always 0, 1, or 2.

3. NUMERIC PROMOTIONS

- We should keep in mind 4 rules when applying operators to data types:
  1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
  2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value data type.
  3. Smaller data types, namely **`byte`**, **`short`**, and **`char`**, are first promoted to **`int`** any time they're used with a Java binary arithmetic operator, even if neither of the operands is **`int`**.
  4. After all promotions has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.
- What is the data type of `x * y` ?
  - `int x = 12; long y = 5;`
  - Since one of the values is **`long`** and the other is **`int`**, and **`long`** is larger than **`int`**, then the **`int`** value is promoted to a **`long`**, and the resulting value is **`long`**.
- What is the data type of `x + y` ?
  - `double x = 14.17; float y = 5.4;`
  - **This code will not compile!** Floating-point literals are assumed to be **`double`**, unless postfixed with an **f**, like `5.4f`.

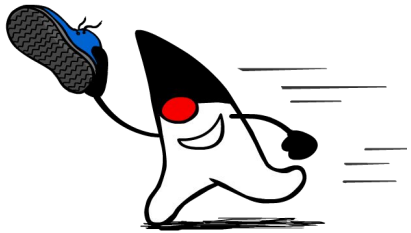
3. NUMERIC PROMOTIONS

- What is the data type of `x * y / z` ?
  - `short x = 4; float y = 3f; double z = 2;`
  - Following the third rule, `x` will be promoted to **`int`** because it is a **`short`** and it is being used in an arithmetic binary operation.
  - The promoted `x` value will then be promoted to a **`float`** so that it can be multiplied with `y`, according to the second rule.
  - The result of `x * y` will then be automatically promoted to a **`double`**, so that it can be multiplied with `z`, as per the first rule.

- Not that hard, right ?
- The Java programming language is based on rules.
- We only need to follow them, and we will be just as confident as Duke is.

4. UNARY OPERATORS

- By definition, a unary operator requires exactly one operand, or variable to function.
- As shown in the following table, they often perform simple tasks, such as increasing a numeric variable by one, or negating a `boolean` value.
- Seems like Duke is having fun smashing bugs. If we want to help him, the first thing we should do is learn Java's rules and practice them, so that we won't produce any more bugs.



UNARY OPERATOR	DESCRIPTION
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a `boolean` value

4. UNARY OPERATORS | LOGICAL COMPLEMENT AND NEGATION

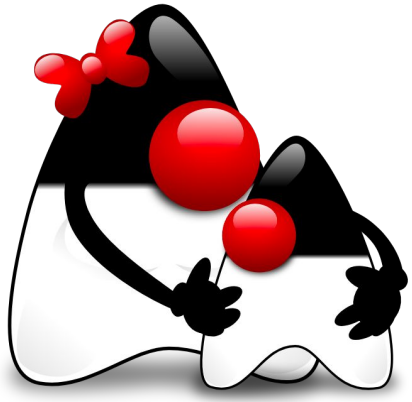
- The logical complement operator, **!**, flips the value of a **boolean** expression.
- For example, if the value is **true**, it will be converted to **false**, and vice versa.
- Let's look at the following example:

```
1. boolean x = false;
2. System.out.println(x);   //false
3. x = !x
4. System.out.println(x);   //true
```

- The negation operator, **-**, reverses the sign of a numeric expression.
- Let's look at the following example:

```
1. double x = 9.5;
2. System.out.println(x);   //9.5
3. x = -x
4. System.out.println(x);   //-9.5
5. x = -x
6. System.out.println(x);   //9.5
```

- Yes, these are the same rules we all learned when we were at school.



4. UNARY OPERATORS | INCREMENT AND DECREMENT

- Increment and decrement operators, ++ and --, can be applied to numeric operands and have the higher order or precedence than binary operators.
- **We need to be careful while using the increment or decrement operators because they alter the result of an expression.**
- If the operator is placed before the operand, pre-increment or pre-decrement operator, then the operator is applied first and the value returned is the new value of the expression.
- If the operator is placed after the operand, post-increment or post-decrement operator, then the original value of the expression is returned, with operator applied after the value is returned.

```
1. int x = 0;
2. System.out.println(x);     //0
3. System.out.println(++x);   //1
4. System.out.println(x);     //1
5. System.out.println(x--);   //1
6. System.out.println(x);     //0
7. int a = ++x + x-- + x++ - --x;
8. System.out.println(a);     //2
```

- In practice you should avoid using multiple pre and post unary operators in a single expression, because they add unnecessary complexity and can lead to bugs.

4. UNARY OPERATORS | ASSIGNMENT OPERATOR

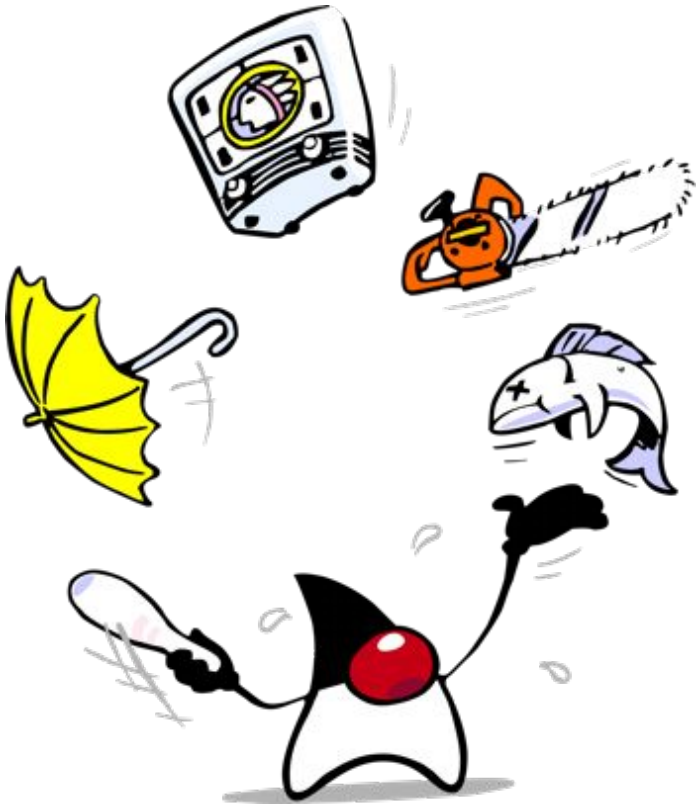
- The assignment operator is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side.
- This statement assigns x the value of 10.

```
1. int x = 10;
```

- Java will automatically promote from smaller to larger data types, as we already saw, but it will throw a compiler exception if it detects we are trying to convert from larger to smaller data types.

```
1. int x = 3.0;
2. short y = 1912123;
3. int z = 23f;
```

- We will also juggle with all these rules, just like Duke. We only need to learn and practice them.



5. BINARY OPERATORS | CASTING PRIMITIVES

- We can fix the examples in the previous slide by casting the results to a smaller data type.
- **Casting primitives is required any time we are going from a larger numerical data type to a smaller numerical data type or converting from a floating-point number to an integral value.**

```
1. int x = (int)3.0;
2. short y = (short)1912123;
3. System.out.println(y); //11579
4. int z = (int)23f;
```

- The examples are now compiling, although there's a cost.
- The second value, 1.912.123, is too large to be stored as a **short**, so numeric **overflow** occurs, and it becomes 11.579.
- **Overflow** is when a number is so large that it will no longer fit within the data type, so the system **"wraps around"** to the next lowest value and counts from there.
- There's also an analogous **underflow**, when the number is too low to fit in the data type.

```
1. //-2147483648
2. System.out.println(2147483647+1);
```

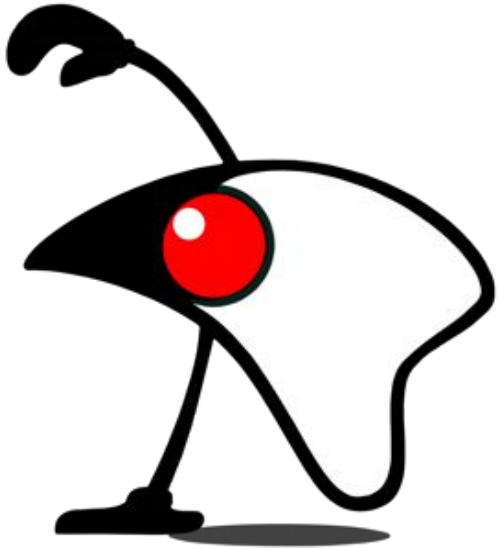
- Since 2147483647 is the maximum **int** value, adding any strictly positive value to it will cause it to wrap to the next negative number.

5. BINARY OPERATORS | COMPOUND ASSIGNMENT OPERATORS

- Besides the simple assignment operator, =, there are also numerous compound assignment operators.
- **Complex operators are just fancy forms of the simple assignment operator, with a logic operation.**

```
1. int x = 2, y = 3;
2. //Simple assignment operator
3. x = x * y;
4. //Compound assignment operator
5. x *= y;
```

- The left-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable.
- In the previous example, if x was not already defined, then the expression x *= y would not compile.



5. BINARY OPERATORS | RELATIONAL OPERATORS

- Relational operators compare two expressions and return a **boolean** value.
- If the two numeric operands are not of the same data type, the smaller one is promoted in the manner as previously discussed.

```
1. int x = 7, y = 15, z = 7;
2. System.out.println(x < y); //true
3. System.out.println(x <= y); //true
4. System.out.println(x >= z); //true
5. System.out.println(x > z); //false
```

- Notice that the last example outputs **false**, because although x and z are the same value, x is not strictly greater than z.

OPERATOR	DESCRIPTION
<	Strictly less than
<=	Less than or equal to
>	Strictly greater than
>=	Greater than or equal to

5. BINARY OPERATORS | LOGICAL OPERATORS

- The logical operators, (&) and (|), may be applied to both numeric and **boolean** data types.
- When they're applied to **boolean** data types, they're referred to as logical operators.
- Alternatively, when they're applied to numeric data types, they're referred to as bitwise operators, as they perform bitwise comparisons of the bits that compose the number.
- We should familiarize with the truth tables, where x and y are assumed to be **boolean** data types.

X & Y (AND)

	y = true	y = false
x = true	true	false
x = true	false	false

X | Y (OR)

	y = true	y = false
x = true	true	true
x = true	true	false

5. BINARY OPERATORS | LOGICAL OPERATORS

- The **short-circuit operators** are nearly identical to the logical operators, & and |, **except that the right-hand side of the expression may never be evaluated if the result can be determined by the left-hand side of the expression.**
- For example, consider the following statement:
  - `boolean x = true || (y < 4);`
  - According to the truth tables, x can only be **`false`** if both sides of the expression are **`false`**.
  - Since we know the left-hand side is **`true`**, there's no need to evaluate the right-hand side, since no value of y will ever make the value of x anything other than **`true`**.

```
1. int x = 7;
2. boolean y = (x >= 7) || (++x <= 8)
3. System.out.println(x); //7
```

- Because `x >= 7` is **`true`**, the increment operator on the right-hand side of the expression is never evaluated, so the output is 7.
- Usually people say "Use double AND and it will work" because they don't know this subtle difference.
- **But we will be professionals.**
- **Professionals know their stuff.**

5. BINARY OPERATORS | EQUALITY OPERATORS

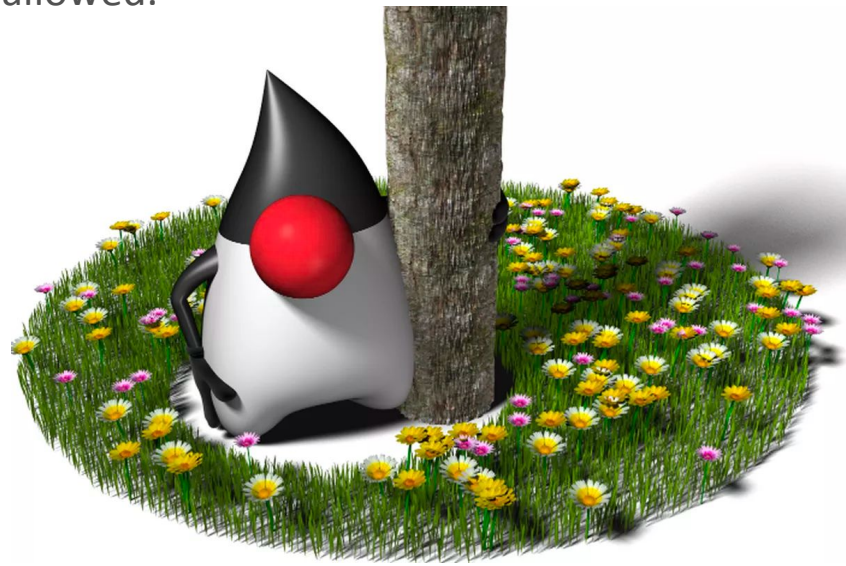
- The equals operator `==` and not equals operator `!=`.
- Like relational operators, they compare two operands and return a **`boolean`** value about whether the expressions or values are equal, or not equal.
- The equality operators are used in one of three scenarios:
  1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted. For example, `3 == 3.00` returns **`true`** since the left side is promoted to a **`double`**.
  2. Comparing two **`boolean`** values.
  3. Comparing two objects, including **`null`** and `String` values.
- The comparisons for equality are limited to these three cases, so we cannot mix and match types.
- For example, each of the following would result in a compiler error:

```
1. boolean x = true == 7;
2. boolean y = false != "Dog";
3. boolean z = 3 == "Dog";
```

6. STATEMENTS

- Java operators allow us to create a lot of complex expressions, but they're limited in the way they can control the program flow.
- For example, imagine we want a section of code to only be executed under certain conditions.
- Or suppose we want a segment of code to repeat once for every item in some list.
- A Java statement is a **complete unit of execution**, terminated with a semicolon (`;`).
- Control flow statements break up the flow of execution by using decision making, looping, and branching, allowing the application to selectively execute blocks of code.

- A block of code in Java is a group of zero or more statements between balanced braces, (`{}`), and can be used anywhere a single statement is allowed.



6. STATEMENTS | IF-THEN STATEMENT

- Often, we only want to execute a block of code under certain circumstances.
- The if-then statement, accomplishes this by allowing our application to execute a block of code if and only if a **boolean** expression evaluates to **true** at runtime.

```
1. if (temperature > 100){
2.  System.out.println("The water" +
3.                          will boil");
4. }
```

- The block allows multiple statements to be executed based on the if-then evaluation.

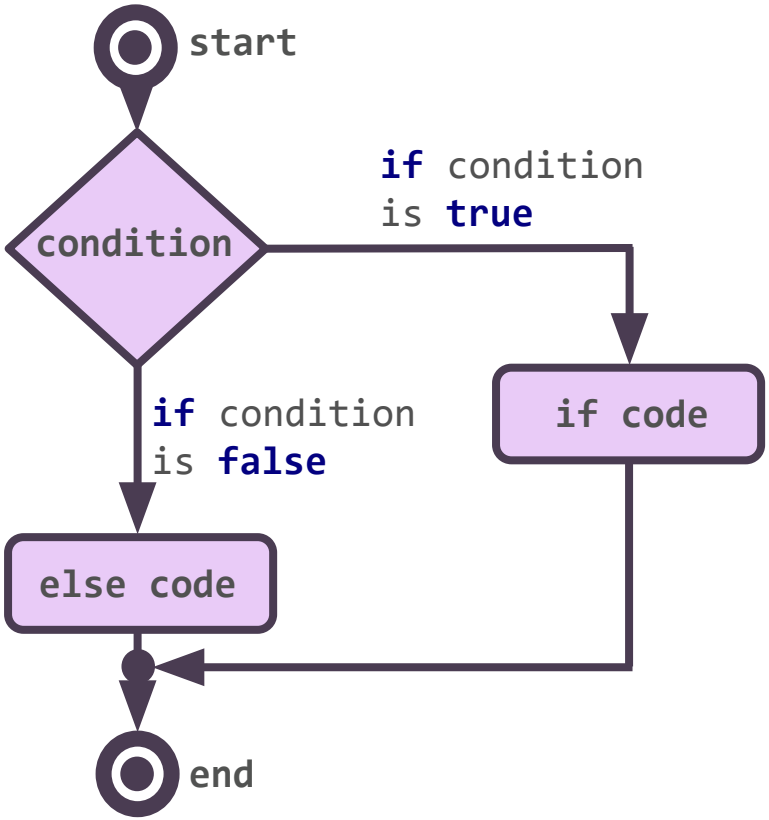
```
1. if (boolean expression){
2.     // the code from here will
3.     // get executed if the
4.     // expression is true
5. }
```

Curly braces are required for blocks or multiple statements but they are optional for a single statement

6. STATEMENTS | IF-THEN-ELSE STATEMENT

- An **if** statement can be followed by an optional **else** statement, which executes when the **boolean** expression is **false**.

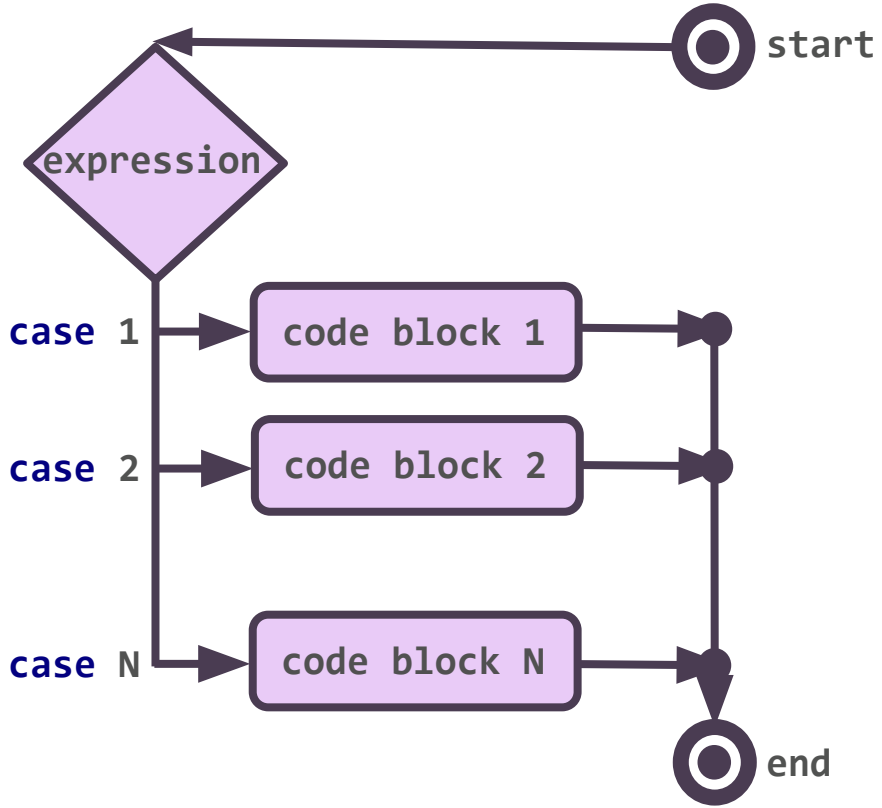
```
1.  if (temperature > 100){
2.    System.out.println("The water " +
3.                       will boil");
4.  } else {
5.    System.out.println("The water " +
6.                       isn't boiling);
7.  }
8.  if (temperature > 100){
9.    System.out.println("The water " +
10.                      will boil");
11. } else if(temperature < 50){
12.   System.out.println("The water " +
13.          is getting hotter);
14. }
```



6. STATEMENTS | SWITCH STATEMENT

- A **switch** statement allows a variable to be tested for equality against a list of values.
- Each value is called a **case**, and the variable being switched on is checked for each **case**.

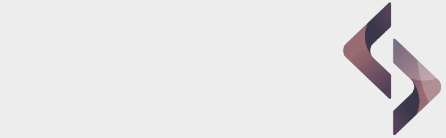
```
1.  switch (expression){
2.      case value1:
3.       // statements
4.      break; // optional
5.      case value2:
6.       // statements
7.      break; // optional
8.  // we may have any number of cases
9.      default: // optional
10.      // statements
11.  }
```



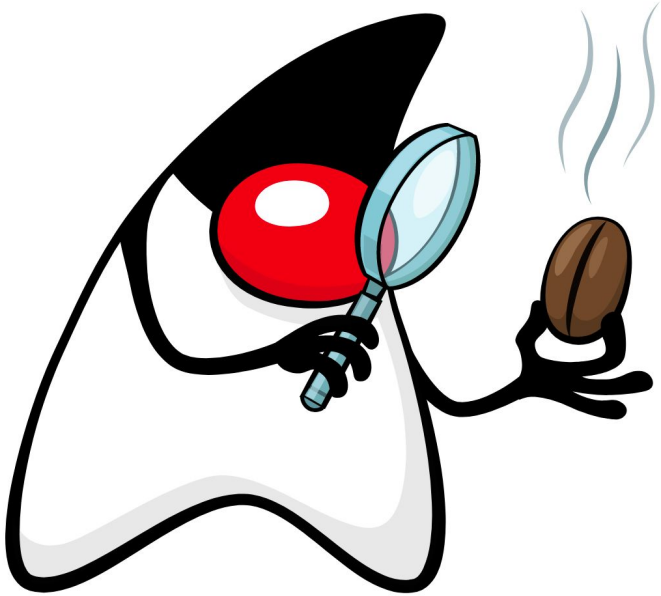
6. STATEMENTS | SWITCH STATEMENT

- The following rules apply to a **`switch`** statement:
  1. The variable used in a **`switch`** statement can only be integers, convertible integers (**`byte`**, **`short`**, **`char`**), `Strings` and `Enums`.
  2. We can have any number of **`case`** statements within a **`switch`**. Each **`case`** is followed by the value to be compared.
  3. The value for a **`case`** must be the same data type as the variable in the **`switch`** and it must be a constant or a literal.
  4. When the variable being switched on is equal to a **`case`**, the statements following that **`case`** will execute until a **`break`** statement is reached.
  5. When a **`break`** statement is reached, the **`switch`** terminates, and the flow of control jumps to the next line following the **`switch`** statement.
  6. Not every case needs to contain a **`break`**. If no **`break`** appears, the flow of control will fall through to subsequent cases until a **`break`** is reached.
  7. A **`switch`** statement can have an optional **`default case`**, which must appear at the end of the **`switch`**. The **`default case`** can be used for performing a task when none of the cases is **`true`**. No **`break`** is needed in the **`default case`**.

6. STATEMENTS | SWITCH STATEMENT



- Ok, let's see if we got it.
- What do you think the following code will print ?
- Duke already started investigating



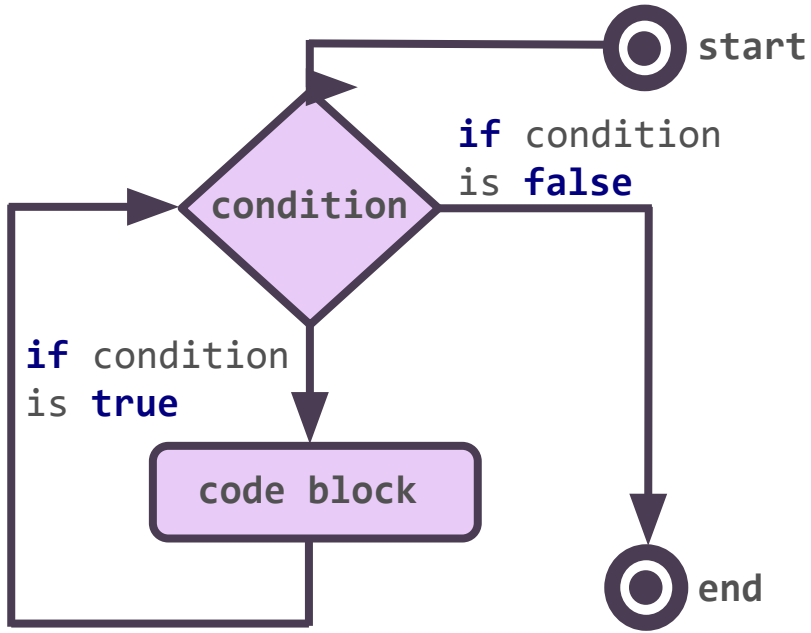
```
1.  int grade = 5;
2.  switch (grade){
3.    case 10:
4.       System.out.println("Super Star!");
5.       break;
6.    case 9:
7.    case 7:
8.       System.out.println("Hm, Not bad!");
9.       break;
10.   case 5:
11.      System.out.println("You passed");
12.   case 4:
13.      System.out.println("You failed");
14.      break;
15.   default:
16.      System.out.println("Invalid " +
17.                                 grade");
18. }
```

6. STATEMENTS | WHILE LOOP STATEMENT

- A **while** loop is a statement that is repeatedly executes a target statement as long as a given condition is **true**.

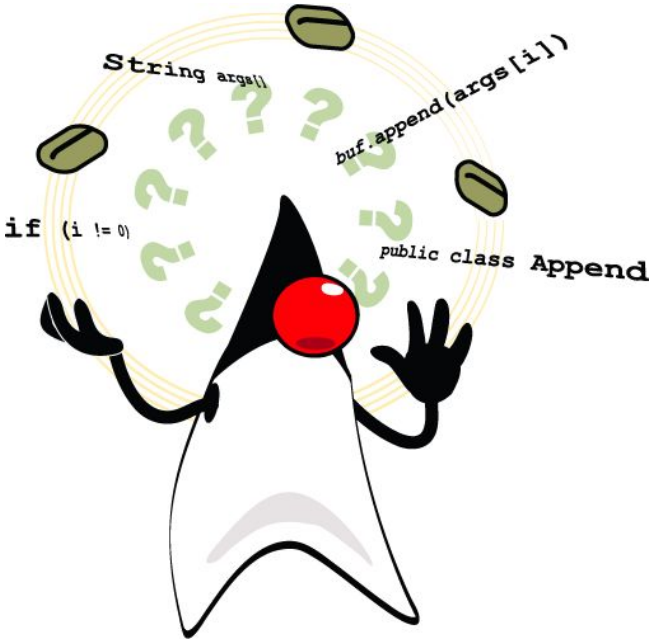
```
1. while (boolean condition){
2.    // statements
3. }
```

- If the **boolean** expression is **true**, then the actions inside the loop will be get executed repeatedly until the expression result is **false**.
- When the condition becomes false, the program jumps to the line immediately following the loop.
- The tricky part is that the loop might not ever run or might never end, depends on the **boolean** expression.



6. STATEMENTS | WHILE LOOP STATEMENT

- Ok, let's see if we got it.
- What do you think the following code will print ?




```
1.  int x = 10;
2.  while (x < 20){
3.     System.out.println("x is: " + x);
4.     x++;
5.  }
```

- The output should look something like this:

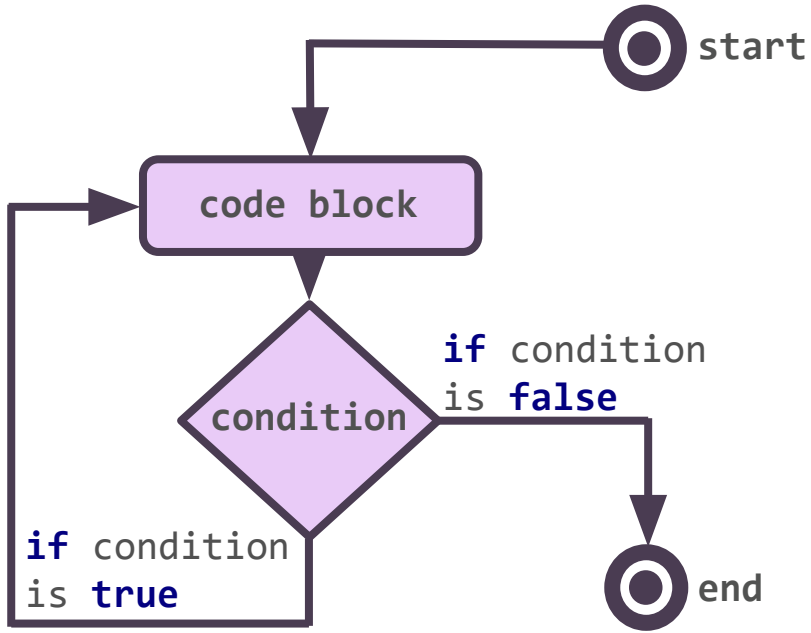
```
1.  x is: 10
2.  x is: 11
3.  x is: 12
4.  x is: 13
5.  x is: 14
6.  x is: 15
7.  x is: 16
8.  x is: 17
9.  x is: 18
10. x is: 19
```

6. STATEMENTS | DO-WHILE LOOP STATEMENT

- A **do-while** loop is like a while loop, except that the **do-while** loop is guaranteed to execute **at least once**.

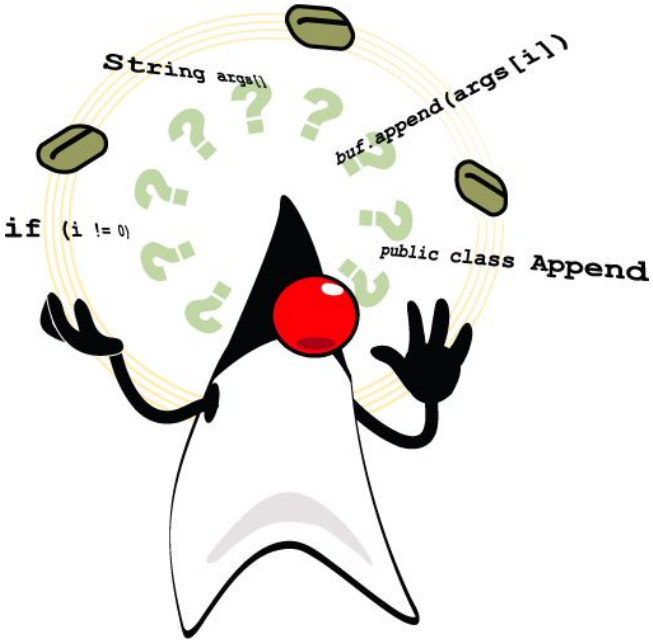
```
1. do {
2.    // statements
3. } while(boolean condition);
```

- Notice that the **boolean** expression appears at the end of the loop, so the statements in the loop are executed **once before the boolean is tested.**
- If the **boolean** expression is **true**, the control jumps back up to the **do** statement, and the statements inside the loop are executed again.
- This process repeats until the **boolean** expression is **false**.



6. STATEMENTS | DO-WHILE LOOP STATEMENT

- Ok, let's see if we got it.
- What do you think the following code will print ?




```
1.  int x = 10;
2.  do {
3.     System.out.print("x is: " + x);
4.     x++;
5.     System.out.print("/n);
6.  } while (x < 20);
```

- The output should look something like this:

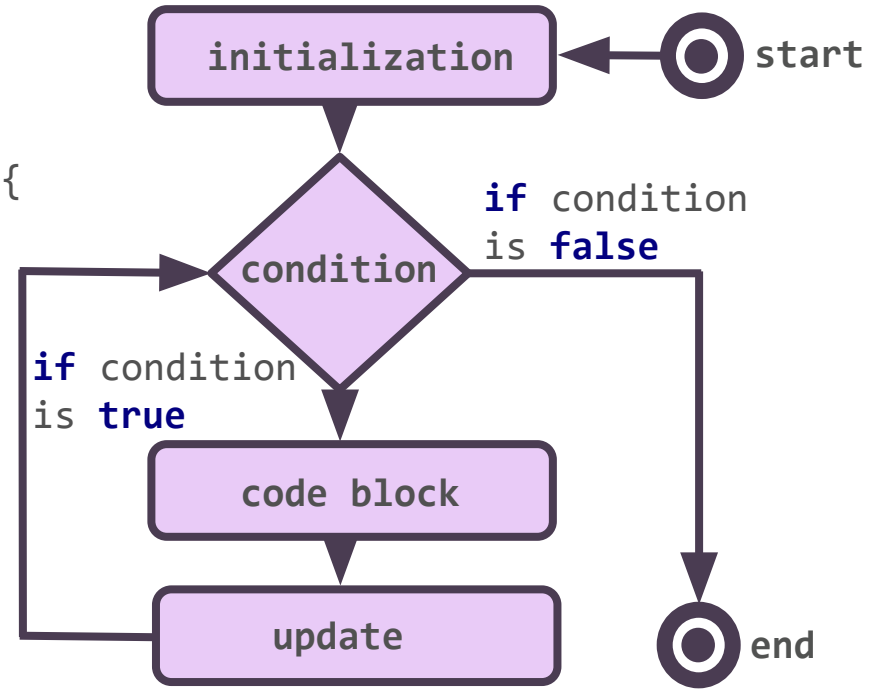
```
1.  x is: 10
2.  x is: 11
3.  x is: 12
4.  x is: 13
5.  x is: 14
6.  x is: 15
7.  x is: 16
8.  x is: 17
9.  x is: 18
10. x is: 19
```

6. STATEMENTS | FOR LOOP STATEMENT

- A **for** loop is a repetition structure that allows us to efficiently write a loop that needs to be executed **a specific number of times**.

```
1. for(initialization; boolean
2.                  condition; update){
3.     // statements
4. }
```

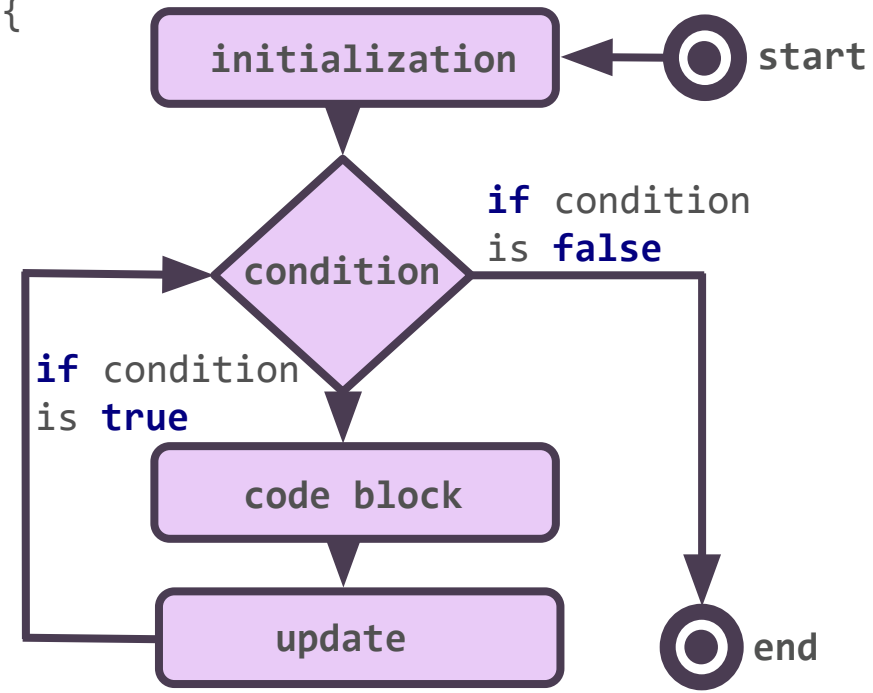
- The **initialization** step is executed first, and only once. This step allows us to declare and initialize variables that will be used in the loop and this step ends with a semicolon (;).
- Next, the **boolean** condition is evaluated. If it is **true**, the **body** of the loop is executed. If it is **false**, the **body** of the loop will not be executed, and the program jumps to the next statement after the **for** loop.



6. STATEMENTS | FOR LOOP STATEMENT

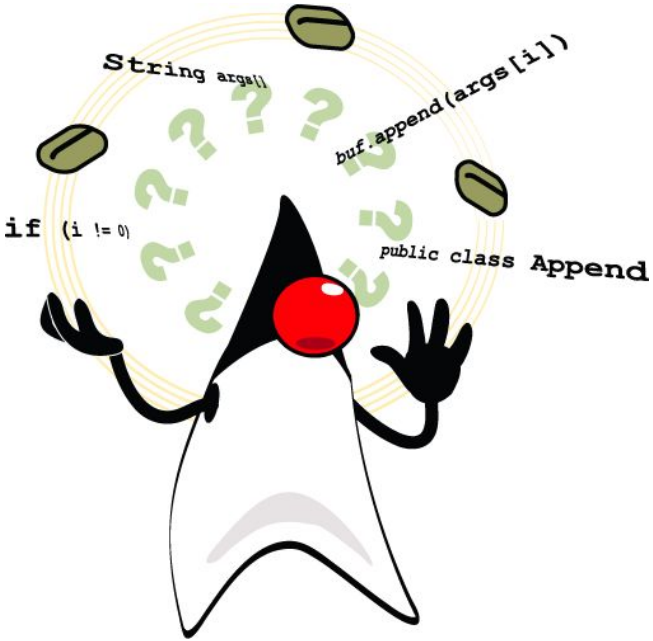
```
1. for(initialization; boolean
2.                  condition; update){
3.     // statements
4. }
```

- After the **body** of the for loop gets executed, the control jumps back up to the **update** part. This part allows us to **update** variables. This statement can be left blank with a semicolon at the end.
- The **boolean** expression is now evaluated again. If it is **true**, the loop executes and the process repeats **(body of loop, then update step, then boolean condition)**.
- After the **boolean** expression is **false**, the **for** loop terminates.



6. STATEMENTS | FOR LOOP STATEMENT

- Ok, let's see if we got it.
- What do you think the following code will print ?




```
1. for(int x = 10; x < 20; x++) {
2.     System.out.println("x is: " + x);
3. }
```

- The output should look something like this:

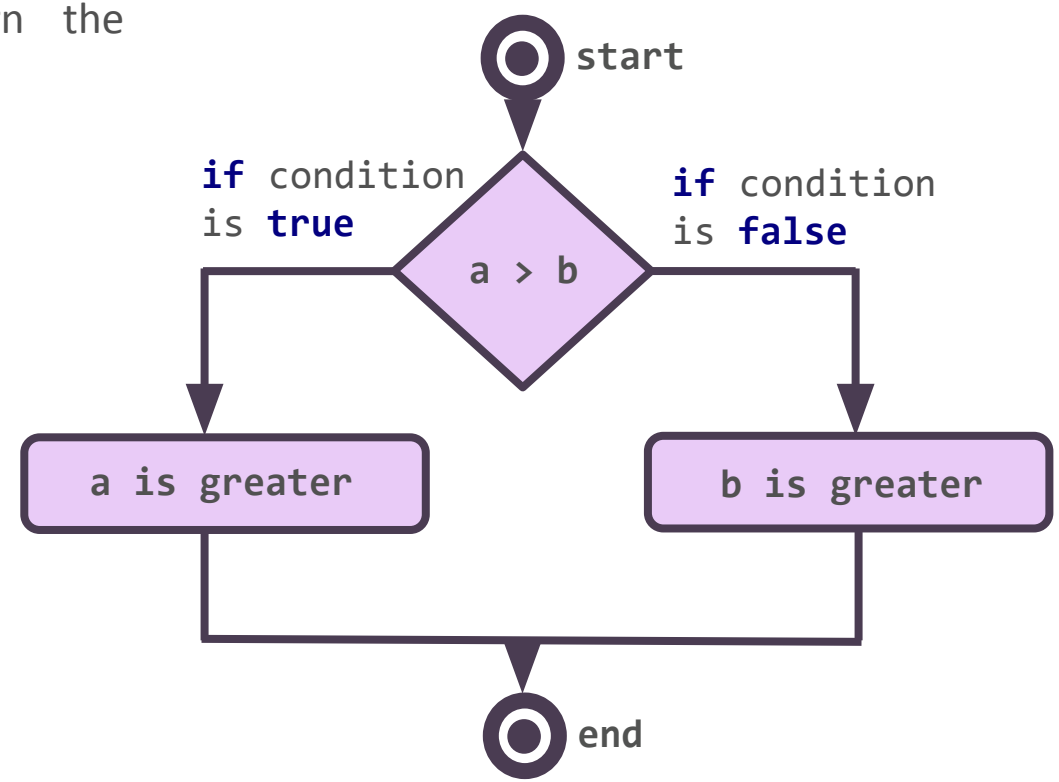
```
1.  x is: 10
2.  x is: 11
3.  x is: 12
4.  x is: 13
5.  x is: 14
6.  x is: 15
7.  x is: 16
8.  x is: 17
9.  x is: 18
10. x is: 19
```

7. EXERCISES

1. Compare two integers and return the greater one.

ALGORITHM

1. Take two integer variables, say A and B.
2. Assign values to the variables.
3. Compare variables **if** A is greater than B.
4. If **true** print A is greater than B.
5. If **false** print A is not greater than B.

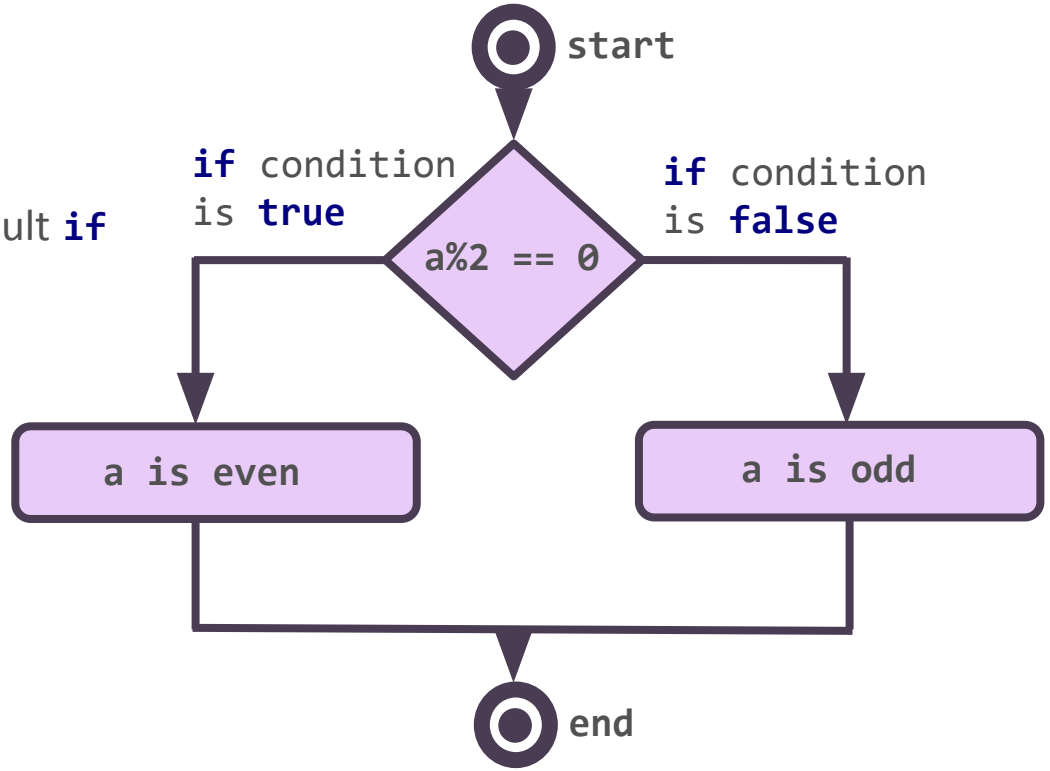


7. EXERCISES

2. Check if a number is odd or even.

ALGORITHM

1. Take integer variable A.
2. Assign value to the variable.
3. Perform A modulo 2 and check result `if` output is `0`.
4. If `true` print A `"is even"`.
5. If `false` print A `"is odd"`.



7. EXERCISES

3. Write a Java program to find if a year is a leap year or not ?
   - A leap year is a year with 366 days which is 1 extra day than normal year.
   - This extra day comes in month of February and on leap year Feb month has 29 days than normal 28 days.
   - If a year is multiple of 400 or multiple of 4 but not multiple of 100 then its a leap year.

ALGORITHM

1. Take an integer variable year.
2. Assign a value to the variable.
3. Check `if` year is divisible by 400 or is divisible by 4, but not 100, display `"is leap year"`.
4. Otherwise, DISPLAY `"not leap year"`.

- Until now, Duke knew that leap years need only to be divisible by 4.
- He took of to let all his friends know this new cool fun fact.

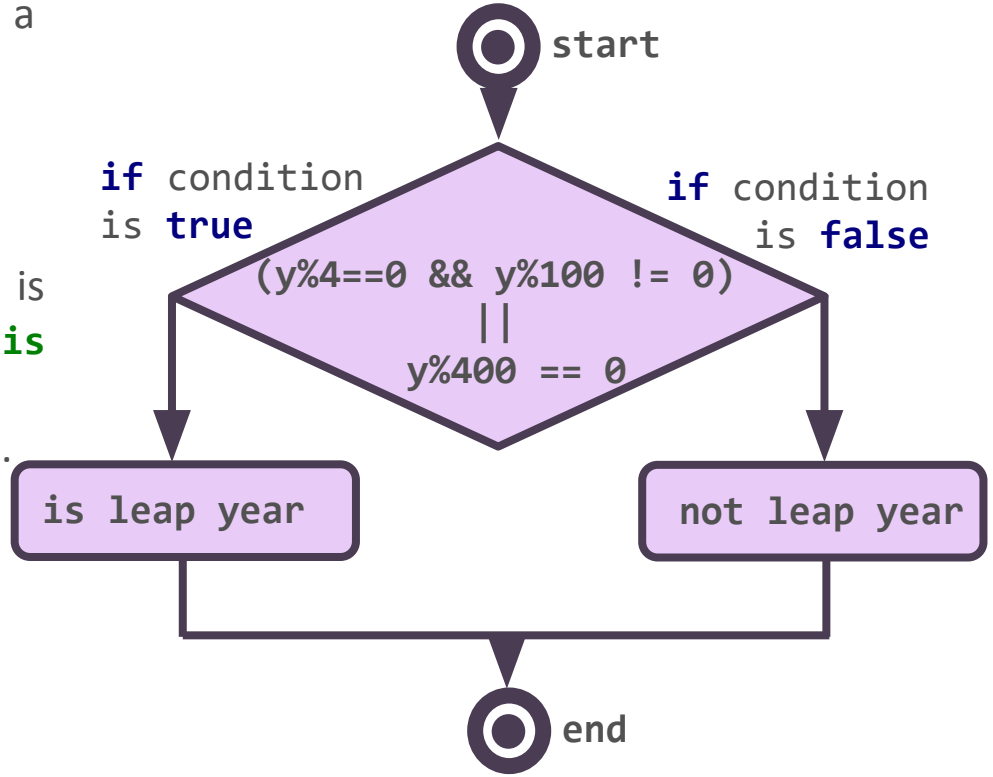
7. EXERCISES

3. Write a Java program to find if a year is a leap year or not ?

ALGORITHM

1. Take an integer variable year.
2. Assign a value to the variable.
3. Check **if** year is divisible by 400 or is divisible by 4, but not 100, display **"is leap year"**.
4. Otherwise, DISPLAY **"not leap year"**.

- The common perception of a leap year is that it only needs to be divisible by 4.
- A few years that are divisible by 4 but are not leap years: 1800, 1900, 2100, 2200 and 2300. **More info here**.



7. EXERCISES

4. Write a program which prints `"fizz"` `if` the number is a multiplier of 3, prints `"buzz"` `if` its multiplier of 5 and prints `"fizzbuzz"` `if` the number is divisible by both 3 and 5.
   - This exercise is called **The Fizz Buzz Problem**.
   - It's quite a common at interviews.

JAVA ARRAYS

AGENDA

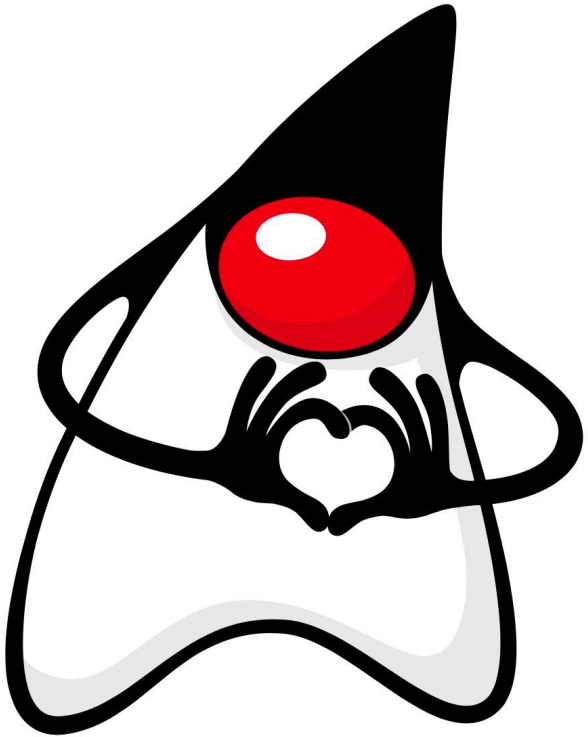
1. Declaring array variables
2. Creating an array
3. Memory model introduction
4. Creating an array
5. Processing an array
6. Multidimensional arrays
7. Exercises

2. DECLARING ARRAY VARIABLES

- **An array is a data structure, which stores a fixed-size sequential collection of elements of the same type.**
- Instead of declaring individual variables, such as `number0`, `number1`, `...`, and `number99`, we declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and `...`, `numbers[99]` to represent individual variables.
- In other words, an array is an ordered list.
- It can contain duplicates.
- Syntax for declaring an array:

```
1. int[] numbers;
```

- Cool, right ? That's why Duke loves arrays.



2. CREATING AN ARRAY

- The most common way to create an array looks like this:

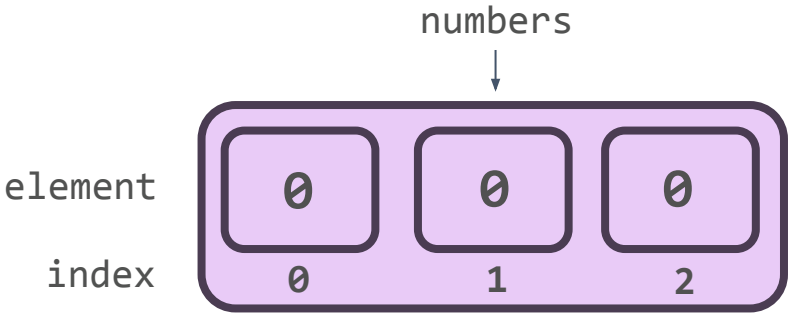
type of array

```
1.  int[] numbers = new int[3];
```

array symbol — size of array

  - When using this form to **instantiate** an array, all elements will be set to the default value for that type.
  - The default value of an `int` is `0`. **We remember this, right ?**
  - Since `numbers` is a **reference variable**, it **points** to the array object.

- **Also, the indexes start with `0`.**



- Hmm, there are a few words on this slide we didn't saw before: the **`new`** keyword, instantiation and reference variable.
- **Let's see what's the deal with them.**

3. MEMORY MODEL INTRODUCTION

- The **new** keyword constructs a **new** object in the **heap memory**.
- Hmm, heap memory ?
- From an abstract point of view, the JVM has 2 main memory zones: **the stack, and the heap.**
- On the stack we can find method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.
- The stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

- Ok, let's sum this up.
- **On the stack memory the JVM puts method calls, primitives and reference variables.**
- Aham, reference variables ??
- **Reference variables are pointers to objects that are stored in the heap**

Here we can see the variable x is declared as an int primitive, so x will be stored on the stack.

```
int x = 3;
```

Here we can see the new keyword. This will create a Dog object on the heap, that we will be able to access using the d1 reference from the stack.

```
Dog d1 = new Dog();
```

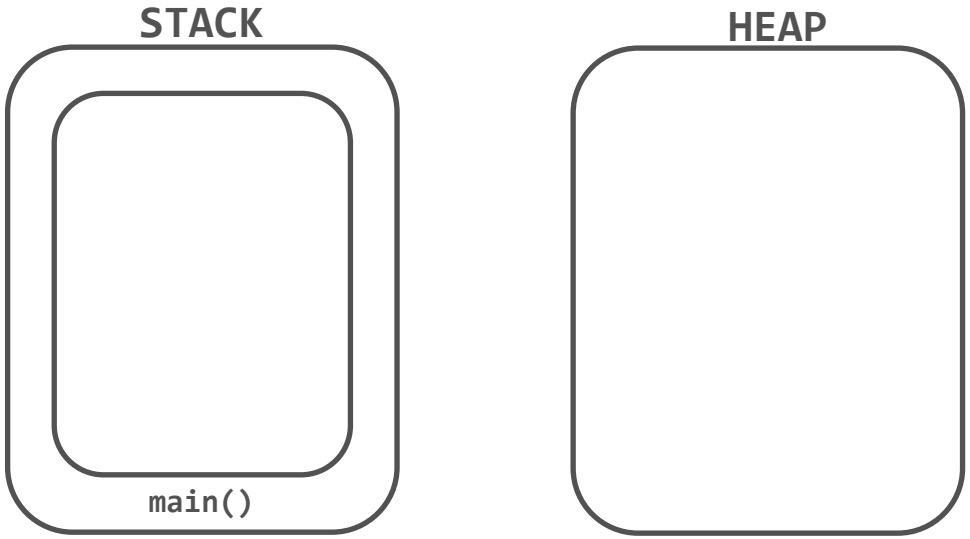
3. MEMORY MODEL INTRODUCTION

- Let's try to draw the memory map in order to visualize the big picture.

```
1. public static void
2.  main(String args[]){
3.   int x = 10;
4.   Dog d1 = new Dog();
5.   x = x + 5;
6.   Dog d2 = new Dog();
7.   d1 = d2;
8.   d2 = new Dog();
9. }
```

- When we start or Java program, the JVM starts executing the main method, from top to down.

- When we start or Java program, the JVM starts executing the main method, from top to down.

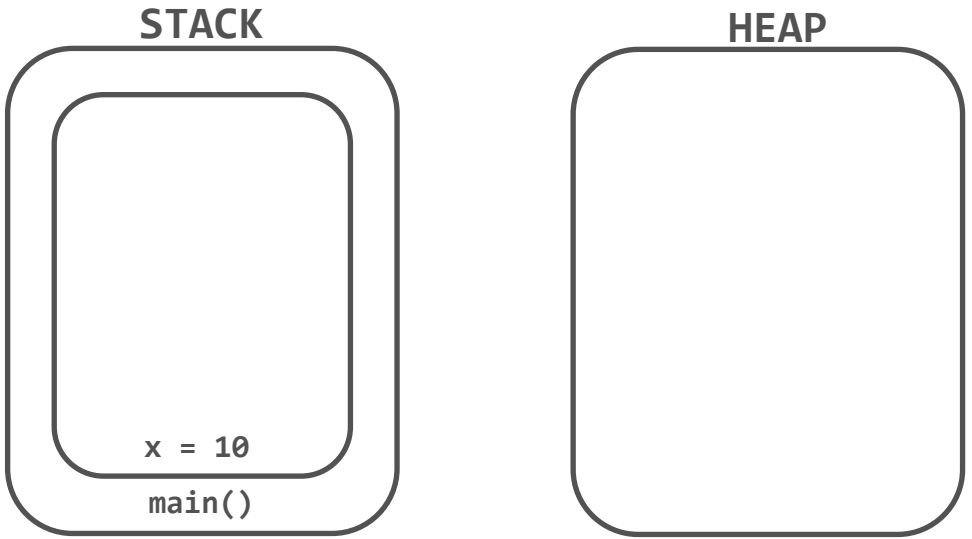


- A memory block will be created on the stack that will be used by the primitives and references created in the `main` method.

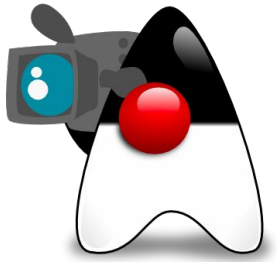
3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 3, the memory map looks like this.
- x is a **primitive** variable of type **int**, so it goes on the stack.
- Nothing fancy yet, right ?



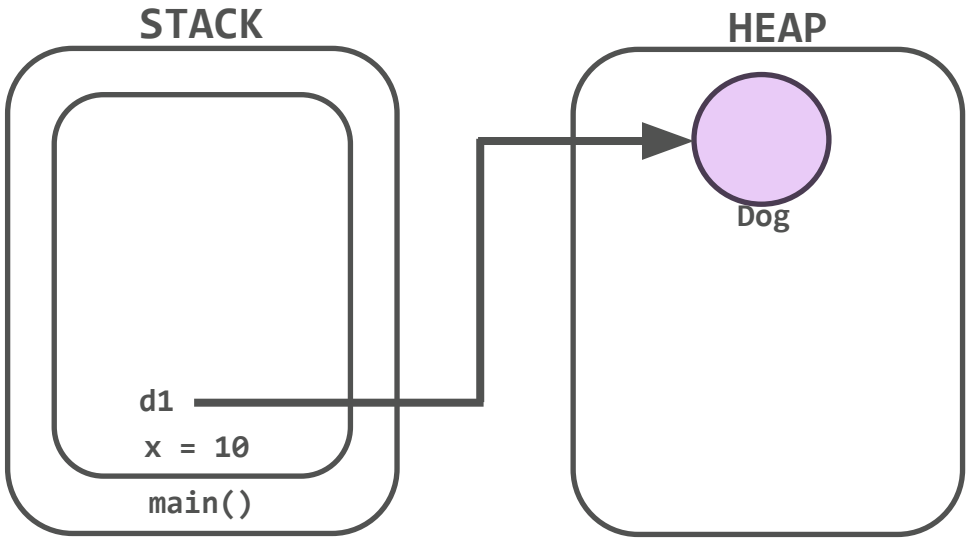
- Duke is recording these examples, because they are very important, every Java Developer needs to know what happens behind the scenes.



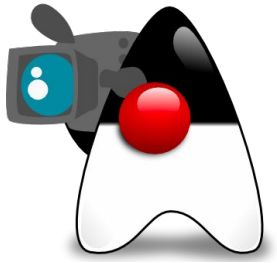
3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 4, the memory map looks like this.
- A `new` Dog object is created on the heap, which has the `d1` reference pointing to it.



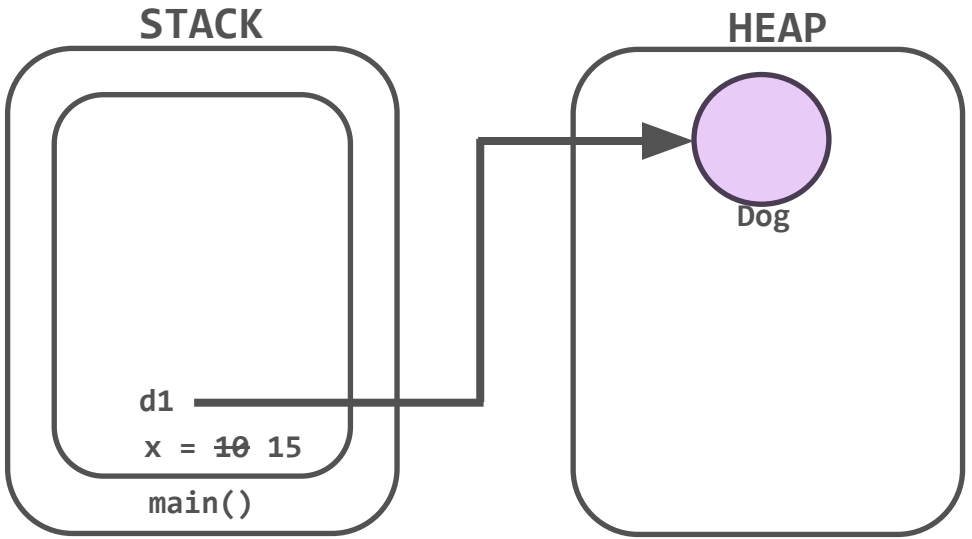
- Duke is still recording.
- It starts to become interesting.



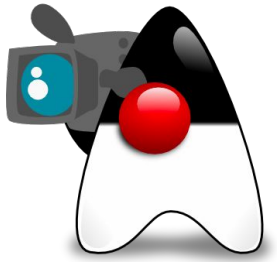
3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 5, the memory map looks like this.
- We update variable's x value.
- Since x is a primitive of type **int**, it's updated value will still be on the stack.



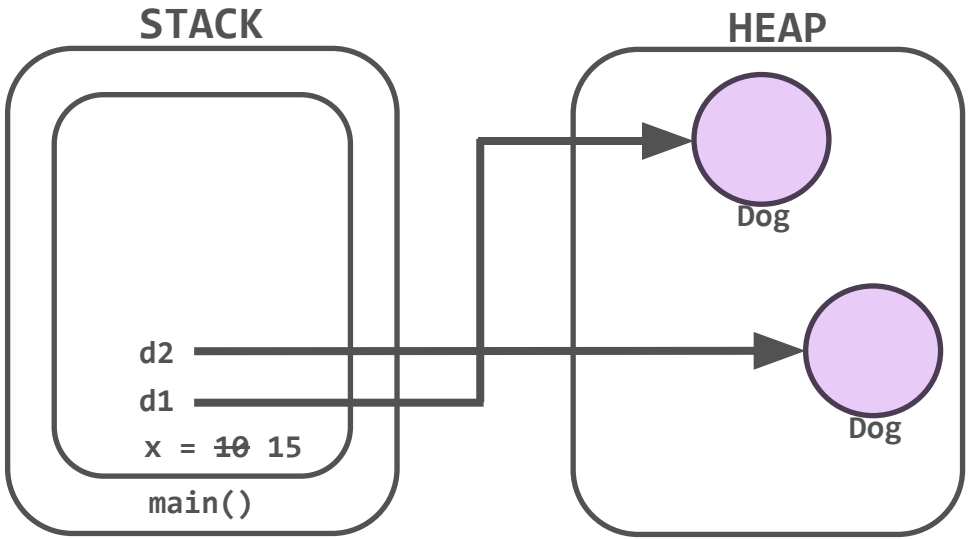
- Duke is still recording.
- Ok, we are ready for some action.



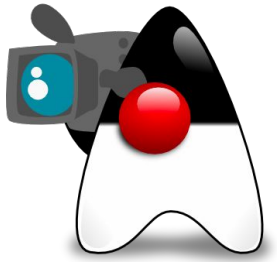
3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 6, the memory map looks like this.
- A **new** Dog object is created on the heap, which has the d2 reference pointing to it.



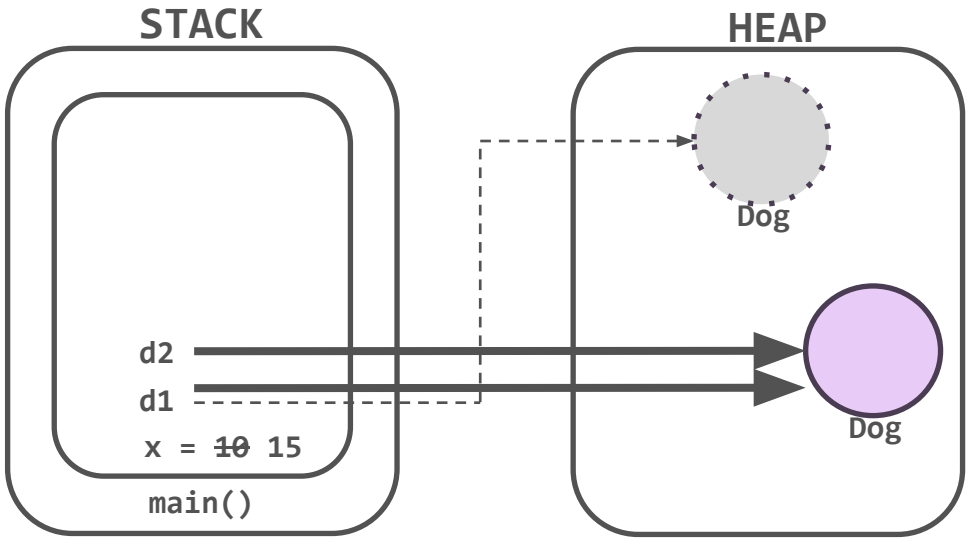
- Duke is still recording.
- Cool, a new character appeared.
- Feels like a plot twist will come soon.



3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 7, the memory map looks like this.
- **The d1 reference will point to the object to which d2 is pointing at.**
- **But a reference can not point at 2 objects at once, so the d1's current pointer will get dropped.**

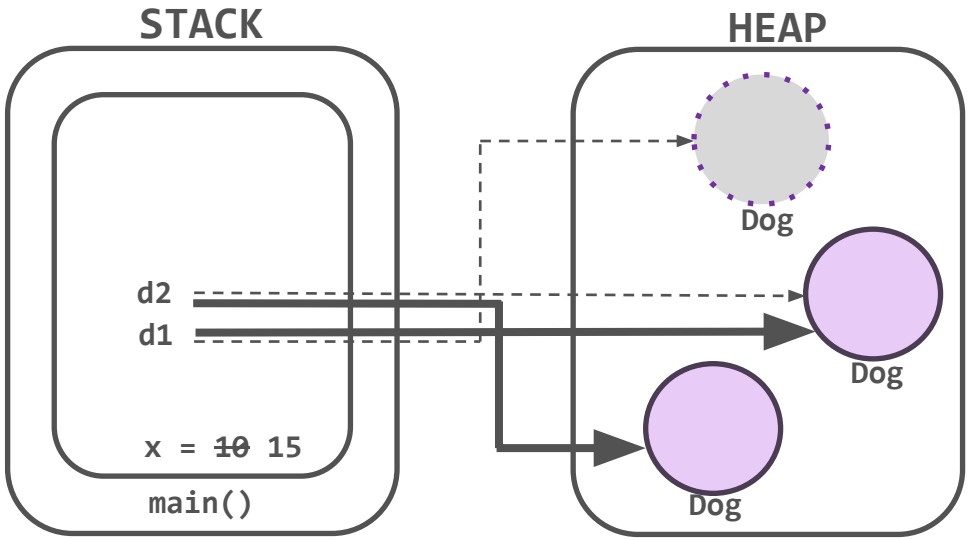


- The dashed arrow is present on the map just to see the previous step. In the memory it got dropped. We'll talk about what happens to that object later (spoiler: it gets turned into garbage).
- Now d1 and d2 point to the same object.

3. MEMORY MODEL INTRODUCTION

```
1.  public static void
2.   main(String args[]){
3.    int x = 10;
4.    Dog d1 = new Dog();
5.    x = x + 5;
6.    Dog d2 = new Dog();
7.    d1 = d2;
8.    d2 = new Dog();
9.  }
```

- Moving to line 8, the memory map looks like this.
- A `new` Dog object is created on the heap, and the existing `d2` reference will point to it. So `d2`'s current pointer gets dropped.



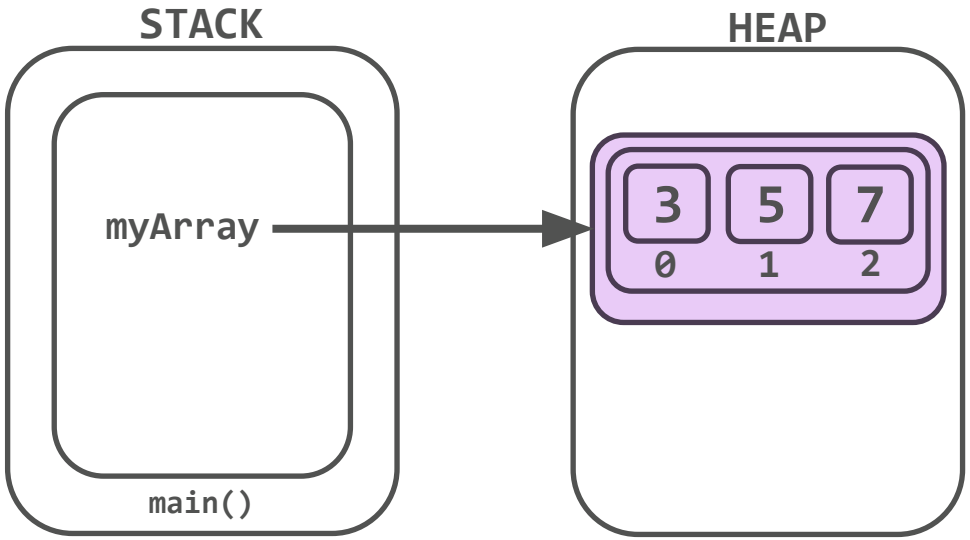
- The dashed arrows is present on the map just to see the previous steps. In the memory they got dropped.
- We'll talk about what happens to that object later.
- Spoiler: This one doesn't turn into garbage.

4. CREATING AN ARRAY

- Another way to create an array is to specify all the elements it should start out with:

```
int[] myArray = new int[] {3, 5, 7};
```

- In this example, we also create an int array of size 3.
- This time, we specify the initial values of those three elements instead of using the defaults.
- We can see that an array is created using the **new** keyword.
- **This means that an array is an object !**



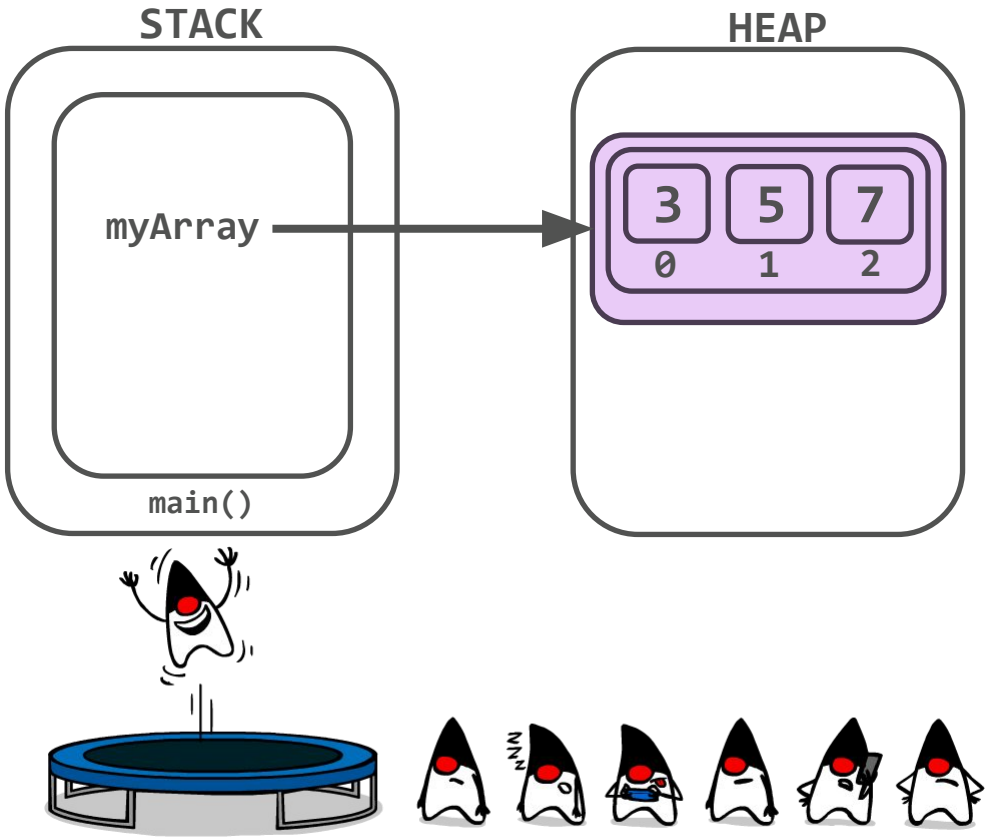
- It goes on the heap, and its access using a reference from the stack.
- Duke's here to measure how much cool stuff we learned so far.
- It looks like he is satisfied.

4. CREATING AN ARRAY

- Since the previous example of creating an array is a bit long and redundant since we specify the type two times, Java offers us a shortcut:

```
int[] myArray = {3, 5, 7};
```

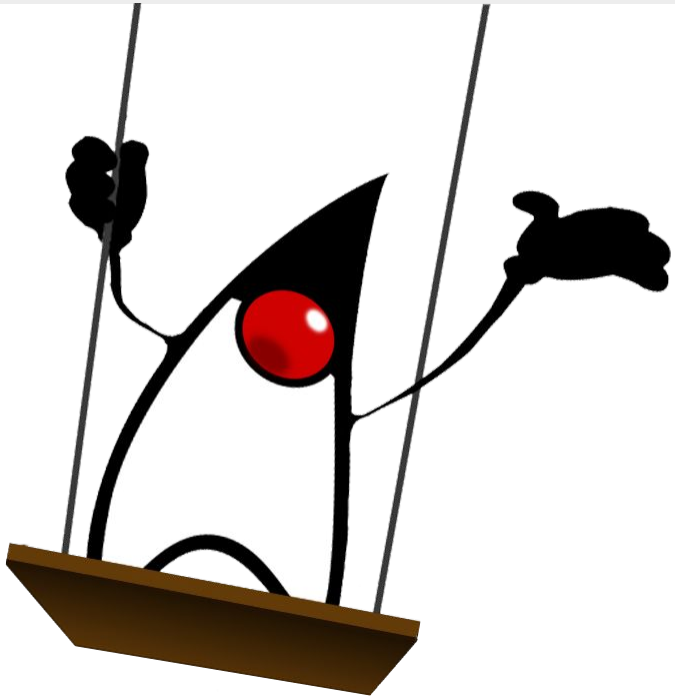
- This approach is just a short version of the previous example.
- Don't get tricked that the **new** keyword is not present anymore.
- **In Java, an array is an object.**
- Even if we use this approach, the JVM will create a **new** object**.**
- Seems like Duke and his friends are ready to process some arrays.



5. PROCESSING AN ARRAY

- When processing array elements, we usually use the **for** loop.

```
1.  public static void main(String args[]){
2.    double[] myArray = {3, 2.5, -8, 11};
3.    // PRINT ALL ELEMENTS
4.    for(int i=0; i < myArray.length; i++){
5.     System.out.print(myArray[i] + " ");
6.    }
7.    // SUMMING ALL ELEMENTS
8.    double total = 0;
9.    for(int i=0; i < myArray.length; i++){
10.    total = total + myArray[i];
11.   }
12.   System.out.print("Total is: " + total);
13. }
```



5. PROCESSING AN ARRAY

- To understand what a code snippet does, a cool trick is to create a table that tracks the evolution of variables.

```
1.  public static void main(String args[]){
2.    double[] myArray = {3, 2.5, -8, 11};
3.    // FIND LARGEST ELEMENT
4.    double max = myArray[0];
5.    for(int i=1; i < myArray.length; i++){
6.       if(myArray[i] > max){
7.          max = maxArray[i];
8.       }
9.    }
10.   System.out.print("Max is: " + max);
11. }
```

i	myArray[i]	max
0	3	3
1	2.5	3
2	-8	3
3	11	11

- In the head of the table we can find all the variables that change during our program.
- The first row represents the initial state.
- The next row represent the new values after each iteration.

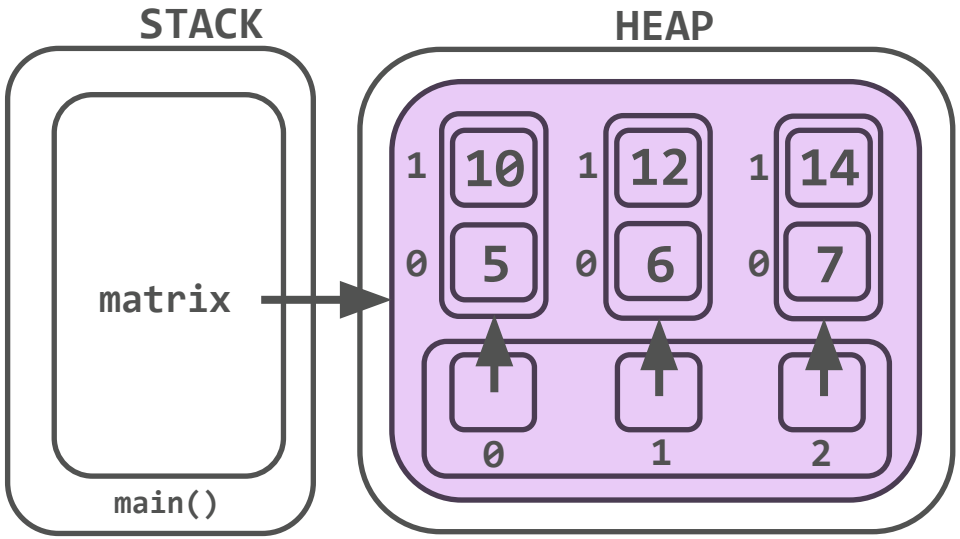
6. MULTIDIMENSIONAL ARRAYS

- We just need to put more array symbols in the declaration to create multi dimensional arrays.

```
int[][] matrix;  // 2D array
int[][] multi[]; // 3D array
```

- We can give our multidimensional array a size directly in the declaration.

```
int[][] matrix = new int[3][2];
matrix[0][0] = 5;
matrix[0][1] = 10;
matrix[1][0] = 6;
matrix[1][1] = 12;
matrix[2][0] = 7;
matrix[2][1] = 14;
```



- As we can see in the memory map, a matrix is an array of arrays.
- Every element of the first array is in fact another array.

7. EXERCISES

1. Write a Java program to calculate the average value of array elements: [1, 7, 3, 10, 9].
   Output: 6
2. Write a Java program to print all odd numbers from an array: [1, 7, 3, 10, 9].
   Output: 1, 7, 3, 9.
3. Write a Java program to reverse an array of integer values: [1, 7, 3, 10, 9]
   Output: [9, 10, 3, 7, 1].
4. Write a Java program to find the number of even and odd integers from an array of integers: [1, 7, 3, 10, 9].
   Output: Odd=4; Even=1.
5. Write a Java program to find the duplicate values of an array of integer values: [1, 7, 3, 7, 10, 1, 9].
   Output: 1, 7.
6. Write a Java program to find the second largest element in an array: [1, 7, 3, 10, 9].
   Output: 9.
7. Write a Java program to find all pairs of elements in an array whose sum is equal to a specified number: [1, 2, 7, 3, 10, 2, 9] with sum 4.
   Output: 1-3; 2-2

DEEPER IN JAVA

AGENDA

1. Packages
2. Creating objects
3. Destroying objects

1. PACKAGES | IMPORTS

- When we installed the JDK, we got a lot of classes that help us in developing application.
- Those classes need a way to organize them.
- In Java, those classes are organized like a cabinet of file cabinets.
- Yes, Java is very organized.

- This approach sounds much better rather than searching by our name through all school's students.
- Java works the same way. It needs us, as developers, to tell it in which packages to look in order to find a specific class.

- Let's imagine we are at school and we go to the administrative office to get our grades.
  - The first question of the secretary is in which year of study are we. She goes to the cabinet of our year of study.
  - Then she asks us in which class are we in, and she finds the cabinet of our class.
  - And lastly, she asks for our name and she finally finds our file.

1. PACKAGES | IMPORTS

- Let's look at this example.

```
1.  public class DeeperInJava{
2.   public static void main(String args[]){
3.    Random someNr = new Random();
4.    System.out.print(someNr.nextInt(47));
5.   }
6.  }
```

- Hmm, we didn't create a class named `Random`.
- So what's with that `Random` ?
- It's a class that's in the JDK, but we need to specify where it is located.
- Now we should get an error like this one: `Random cannot be resolved to a type`.
- This error appears because we didn't tell the JVM where is the `Random` class.

- In order to tell the JVM from where we need the `Random` class, we need to use the **`import`** statement.

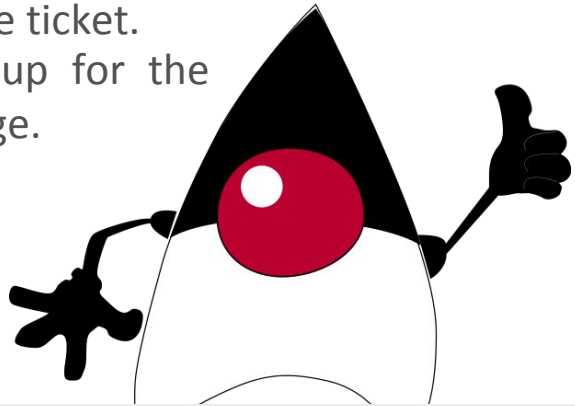
1. PACKAGES | IMPORTS

- Let's look add the **import** statement.

```
1. import java.util.Random;
2. public class DeeperInJava{
3.  public static void main(String args[]){
4.   Random someNr = new Random();
5.   System.out.print(someNr.nextInt(47));
6.  }
7. }
```

- Now that we added the **import** statement, the JVM will know from where to load the `Random` class.
- The JVM will go into the `java` folder, then to the `util` folder, and it will take the `Random` class from there.
- This code compiles successfully and outputs a random number between `0` and `46`.

- Hmm, cool stuff.
- Who is up for a challenge ?
- Using this cool new class, `Random`, we can implement a lottery system.
- We need to create a ticket of 6 numbers.
- Extract 46 random numbers and compare them with the numbers from the ticket.
- Duke's up for the challenge.



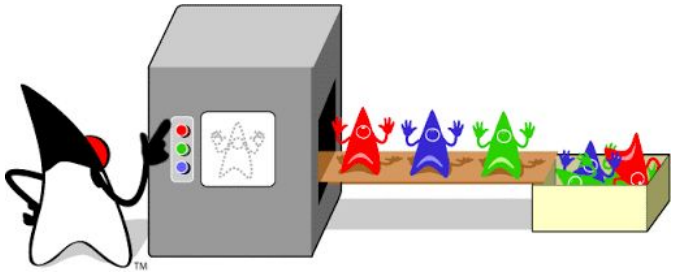
1. PACKAGES | WILDCARDS

- If we need to **import** multiple classes that are in the same package, we can do it in a single line.

```
1.  /* Now we will import the Random class,
2.   * and all other classes that are placed
3.   * in the java.util package.
4.   */
5.  import java.util.*;
6.  public class DeeperInJava{
7.   public static void main(String args[]){
8.    Random someNr = new Random();
9.    System.out.print(someNr.nextInt(47));
10.  }
11. }
```

- The * is called a wildcard.
- It matches all classes in the package.

- We can now use every class in the `java.util` package.
- **The wildcard doesn't import any child packages, fields, or methods, it imports only classes.**
- Hmm, ok.
- So we now have a bunch of classes that we don't need. Is it ok ?
- In practice it is recommended to **import** only the classes we need.



1. PACKAGES | REDUNDANT IMPORTS

- Ok, but we used `System.out.print` without an **`import`** until now.
- That's because the `System` class is part of a special package called `java.lang`.
- **The reason that this package is special, is because it's imported automatically in every class.**

```
1.  import java.lang.*;
2.  import java.lang.System;
3.  import java.util.*;
4.  import java.util.Random;
5.  public class DeeperInJava {
6.   public static void main(String args[]){
7.    Random someNr = new Random();
8.    System.out.print(someNr.nextInt(47));
9.   }
10. }
```

- In this code snippet we can see that there are 3 redundant **`import`** statements.
- Lines 1 and 2 are redundant because every class present in the `java.lang` package is imported by default.
- Line 3 is also redundant because `Random` is already imported from in line 4. If line 4 wasn't present, `java.util.*` wouldn't be redundant. Java looks for the most specific **`import`** first.
- Importing a class into a class from the same package is also redundant. Java scans automatically the current package for other classes.

1. PACKAGES | NAMING CONFLICTS

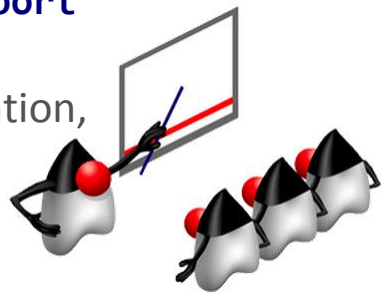
- A cool reason of using packages, is that the name of the classes aren't required to be unique.
- Yes, we can have the scenario that we have 2 different classes with the same name. A common example of this is the `Date` class.
- Java provides implementations of `java.util.Date` and `java.sql.Date`.

```
1. public class NameConflict {
2.  Date today;
3. }
```

- You can write either `import java.util.*;` or `import java.util.Date`.

```
1. import java.util.*;
2. import java.sql.*;
3. public class NameConflict {
4.  Date today;
5. }
```

- When a class is found in multiple packages, Java will throw a compile error, since it doesn't know which one to use.
- In our example is easy, we can just remove one of the **`import`** statements.
- Yes, understanding **`import`** statements is a must.
- Please pay close attention, just like the Dukes.



1. PACKAGES | NAMING CONFLICTS

- But what do we do if we need a classes from both packages ?

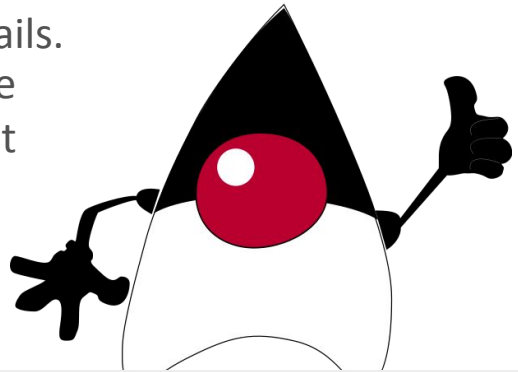
```
1. import java.util.Date;
2. import java.sql.*;
3. public class NameConflict {
4.  Date today;
5. }
```

- If we explicitly **import** a class, it takes precedence over any wildcards.
- Java thinks, “Ok, Sure! You really want the `java.util.Date` class.”
- But sometimes we really need to use two Date objects from two different packages.

- When we encounter this situation, we can use the fully qualified class name when declaring the variable, without using any **import** statements.

```
1. public class NameConflict {
2.  java.util.Date dateFromUtil;
3.  java.sql.Date dateFromSql;
4. }
```

- Yes, a professional knows all these subtle details.
- We all want to be professionals, right ?
- Duke sure wants.



2. CREATING OBJECTS | CONSTRUCTORS

- **We remember that an object is an instance of a class, right ?**
- In order to create an instance of a class, all we need to do is us the **new** keyword.
- `Random someNr = new Random();`
- First, we specify the type of the object that we'll be creating (`Random`) and give the reference a name (`someNr`).
- Then we write **new** `Random()` to create object on the heap.
- `Random()` looks like a method call since it is followed by parentheses.
- **It's called a constructor, which is a special type of method that creates a new object.**

- Let's define our own constructor.

```
1. public class Mouse {
2.  public Mouse(){
3.    System.out.println("Constructing"+
4.    "a Mouse");
5.  }
6. }
```

- **We need to know that a constructor is a special method because:**
  1. **The name of the constructor matches the name of the class.**
  2. **The constructor doesn't have a return type.**

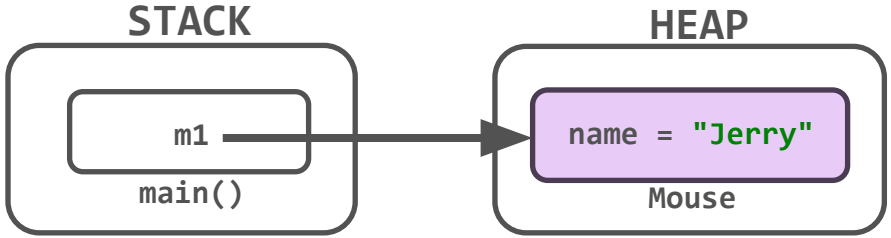
2. CREATING OBJECTS | CONSTRUCTORS

- Hmm, so, why do need constructors ?
- **We need them to create objects on the heap and to initialize their states.**

```
1.  public class Mouse {
2.   String name;
3.   public Mouse(){
4.    name = "Jerry";
5.   }
6.  }
```

```
1.  public class Home {
2.   public static void
3.    main(String args[]){
4.     Mouse m1 = new Mouse();
5.     System.out.println(m1.name);
6.   }
7.  }
```

- Let's look at the memory map when we execute the `main` method from the `Home` class.



1. A reference `m1` of type `Mouse` gets placed on the stack.
2. The constructor of the `Mouse` class is called using the **`new`** keyword and creates an object on the heap.
3. Because of the `=` operator, `m1` will point to that object.
4. The constructor initializes the state name with **`"Jerry"`**.

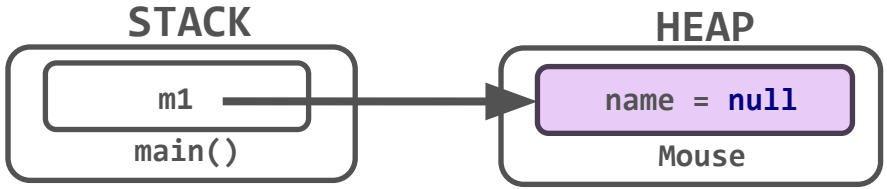
2. CREATING OBJECTS | OBJECT STATES

- We can initialize the object's states using the reference.
- **Let's use the default constructor. We will talk more about this one later. For now, just note that if we don't supply a construct, Java will provide one for us.**

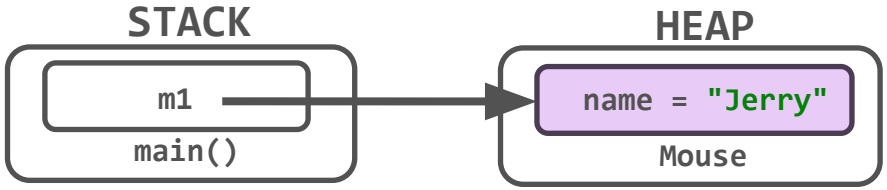
```
1. public class Mouse {
2.  String name;
3. }
```

```
1. public class Home {
2.  public static void
3.   main(String args[]){
4.    Mouse m1 = new Mouse();
5.    m1.name = "Jerry";
6.  }
7. }
```

- At line 4 we create a reference `m1` on the stack that will point to a mouse object from the heap. Because the state `name` is of type String, its default value is **`null`**.



- At line 5, using the reference `m1`, from the stack we initialize the state `name`, of the object to which is `m1` is pointing to.



2. CREATING OBJECTS | OBJECT REFERENCES

- Ok, let's sum up everything we learned about references until now.
- A reference type refers to an object (an instance of a class).
- **Unlike primitive types that hold their values on the stack, references are placed on stack, but they do not hold the value of the object they refer to.**
- **A reference from the stack points to an object that is placed on the heap.**
- If we want to access an object from the heap, we **must** use a reference from the stack that points to it.

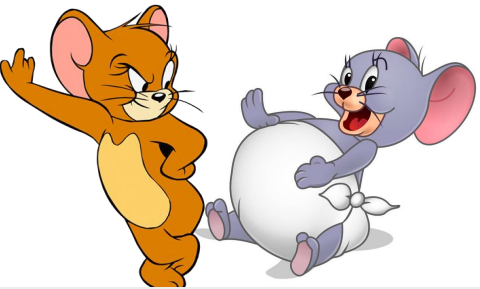
- A reference can point to an object only if:
  1. The object from the heap already exists and has the same type as the reference.
  2. The object is newly created using the `new` keyword.
- Ok, Duke's ready for some more examples.

2. CREATING OBJECTS | OBJECT REFERENCES

- Let's use these code snippets to help us understand how references work.

```
1.  public class Mouse {
2.   String name;
3.  }
```

- This class is a blueprint to create mice.
- We all know Jerry, Nibbles and Mickey, right ?

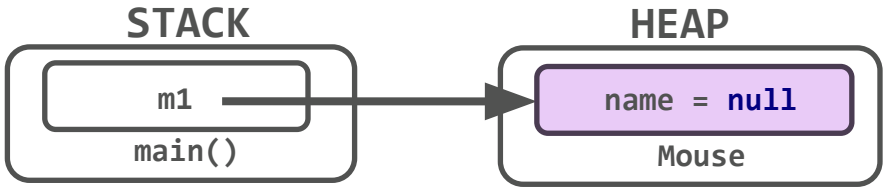


```
1.  public class Home {
2.   public static void main(String args[]){
3.     Mouse m1 = new Mouse();
4.     m1.name = "Jerry";
5.     Mouse m2 = new Mouse();
6.     m2.name = "Nibbles";
7.     m2 = m1;
8.     Mouse m3 = new Mouse();
9.     m3.name = "Mickey";
10.    m1 = m3;
11.    System.out.println(m1.name);
12.    System.out.println(m2.name);
13.    System.out.println(m3.name);
14.   }
15.  }
```

2. CREATING OBJECTS | OBJECT REFERENCES

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m2 = m1;
9.     Mouse m3 = new Mouse();
10.    m3.name = "Mickey";
11.    m1 = m3;
12.    System.out.println(m1.name);
13.    System.out.println(m2.name);
14.    System.out.println(m3.name);
15.  }
16. }
```

- The easiest way to understand what this code snipped does is to draw the memory map.
- At line 4, a reference named `m1` of type `Mouse` gets placed on the Stack, and it will point to a **new** `Mouse` object that gets created on the Heap.

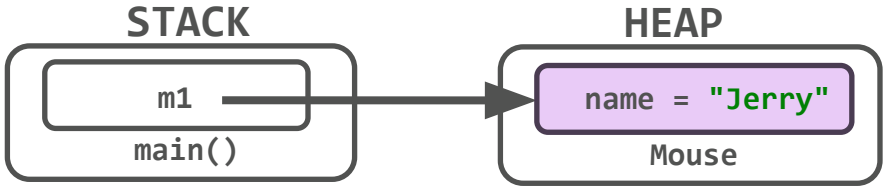


- Why is the **name** **null** ?
- Because we used the default constructor to create that **new** `Mouse` object.
- When we use the default constructor, all instance variables (states) are initialized with their default value. In this case **name** is of type `String`, which is an object. The default value for all objects is **null**.

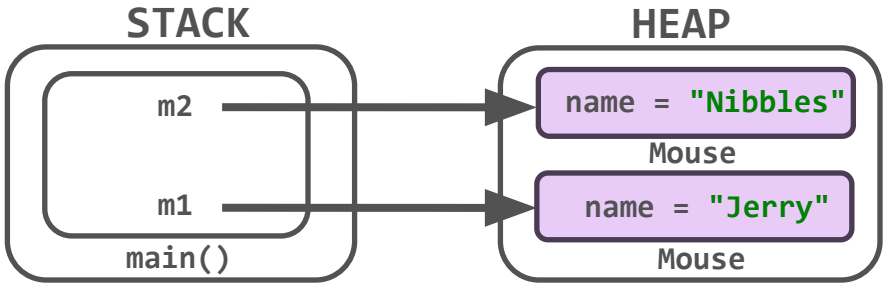
2. CREATING OBJECTS | OBJECT REFERENCES

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m2 = m1;
9.     Mouse m3 = new Mouse();
10.    m3.name = "Mickey";
11.    m1 = m3;
12.    System.out.println(m1.name);
13.    System.out.println(m2.name);
14.    System.out.println(m3.name);
15.  }
16. }
```

- At line 5, using the reference `m1`, we access the state of the object, `m1` is pointing to and we change it.



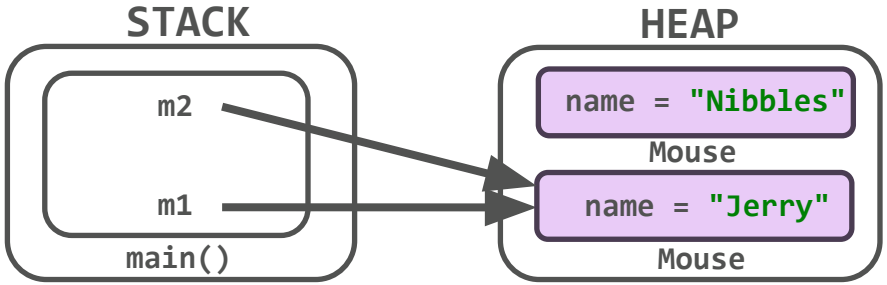
- Lines 6 and 7, create a reference `m2` of type `Mouse` on the Stack, which will point to a **new** `Mouse` object that gets placed on the Heap.



2. CREATING OBJECTS | OBJECT REFERENCES

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m2 = m1;
9.     Mouse m3 = new Mouse();
10.    m3.name = "Mickey";
11.    m1 = m3;
12.    System.out.println(m1.name);
13.    System.out.println(m2.name);
14.    System.out.println(m3.name);
15.  }
16. }
```

- At line 8, the reference `m2` will point to the object to which `m1` is pointing to.

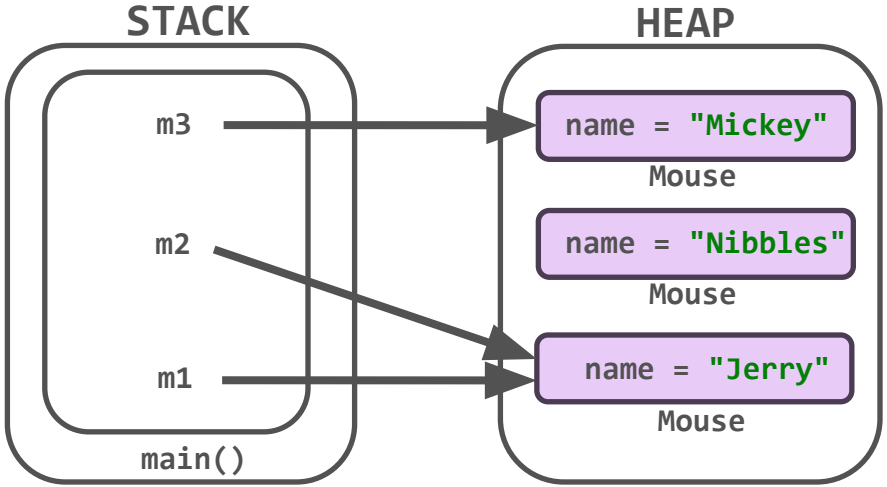


- Hey, what happens to Nibbles now ?
- For now let's just say that Nibbles becomes eligible for ***garbage collection***. We'll talk more about this in a few slides.
- Code snippets like this are common at technical interviews. Best way to tackle them is to draw the memory map just like we are doing now.

2. CREATING OBJECTS | OBJECT REFERENCES

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m2 = m1;
9.     Mouse m3 = new Mouse();
10.    m3.name = "Mickey";
11.    m1 = m3;
12.    System.out.println(m1.name);
13.    System.out.println(m2.name);
14.    System.out.println(m3.name);
15.  }
16. }
```

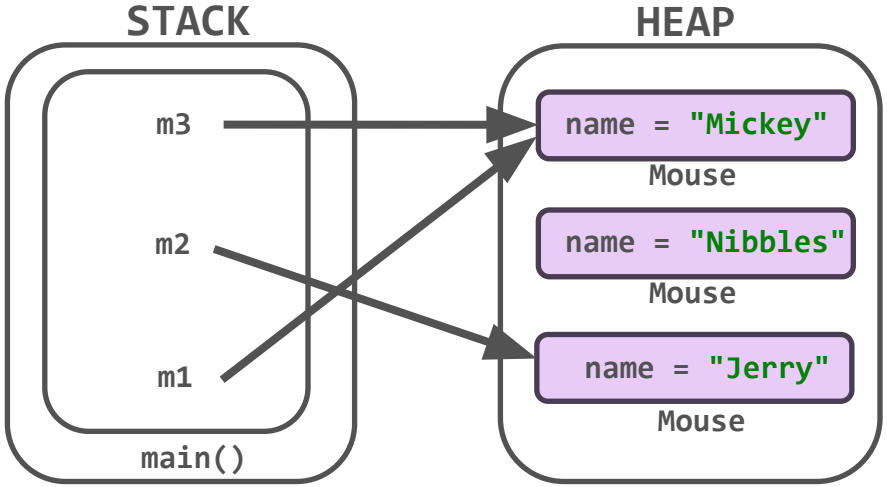
- Lines 9 and 10, create a reference `m3` of type `Mouse` on the Stack, which will point to a **`new`** `Mouse` object that gets placed on the Heap and initializes the object's **`name`** state with **`"Mickey"`**.



2. CREATING OBJECTS | OBJECT REFERENCES

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m2 = m1;
9.     Mouse m3 = new Mouse();
10.    m3.name = "Mickey";
11.    m1 = m3;
12.    System.out.println(m1.name);
13.    System.out.println(m2.name);
14.    System.out.println(m3.name);
15.  }
16. }
```

- At line 11, the reference `m1` will point to the object to which `m3` is pointing to.



- In order to find out the output, just follow the pointers (the lines from the references to the objects), and it is: Mickey, Jerry, Mickey.

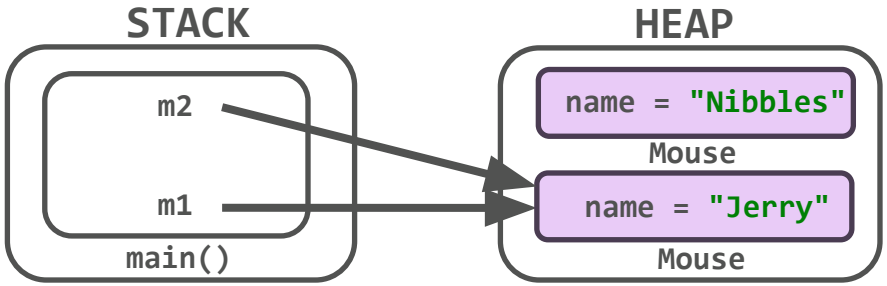
3. DESTROYING OBJECTS

- **Hey, let's pause for a moment here.**
- **So, what happened to little Nibbles ?**
- We've seen how easy is for us to create objects, right ? Just use the **new** keyword.
- We can create as many objects that we want, if we have enough memory on the heap.
- Yes, the memory is limited, so we should not abuse it.
- Luckily, Java takes care of the memory for us.
- We said that Nibbles became eligible for garbage collection. Let's see what this means.

- **Garbage collection is the process that automatically frees memory from the heap by deleting those objects which are no longer reachable by our program.**
- The garbage collector comes automatically, when he decides to.
- We have a method, `System.gc()` with which we can **suggest** to the garbage collector that now would be a good time to start freeing up some memory.
- **But the garbage collector is free to ignore our suggestion.**

3. DESTROYING OBJECTS

- Java waits patiently until it's sure we don't need that object and only than calls the garbage collector.
- **An object will remain on the heap as long as it's reachable from the stack.**
- **An object from the heap is no longer reachable from a reference from the stack when:**
  1. The object no longer has any references pointing to it.
  2. All references to the object have gone out of scope.
- In our previous example, at line 8, the reference `m2` started pointing to Jerry instead of Nibbles.



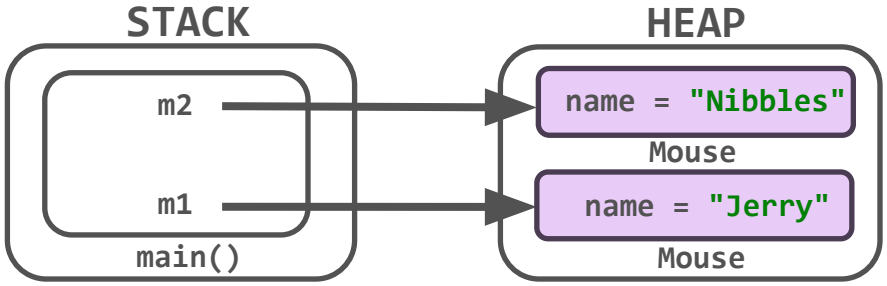
- Because Nibbles doesn't have any references pointing to him, we say that he became **eligible** for garbage collection.
- We say eligible, because the garbage collector won't necessarily come at that exact point in time.
- The garbage collector might wait and come once for multiple objects, not just for one.

3. DESTROYING OBJECTS

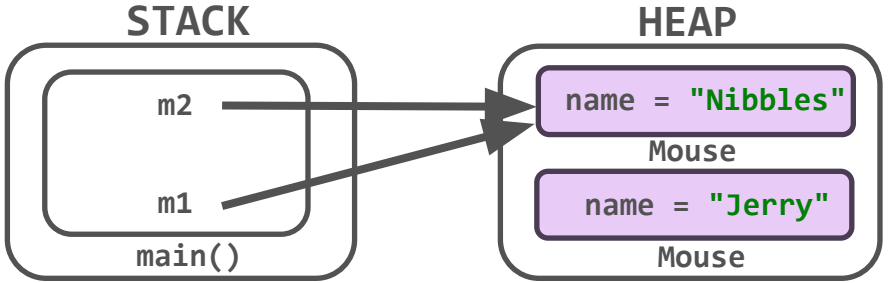
- Cool stuff. Let's see if we can figure out when each object becomes **eligible** for garbage collection.

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.      Mouse m1 = new Mouse();
5.      m1.name = "Jerry";
6.      Mouse m2 = new Mouse();
7.      m2.name = "Nibbles";
8.      m1 = m2;
9.      Mouse m3 = m1;
10.     m1 = null;
11.   }
12.  }
```

- Line 4-7 are straight forward, right ?



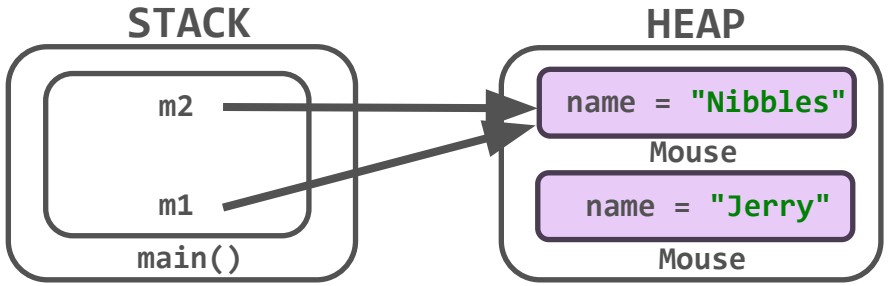
- At line 8, the `m1` reference will point to the object to which `m2` is pointing to.



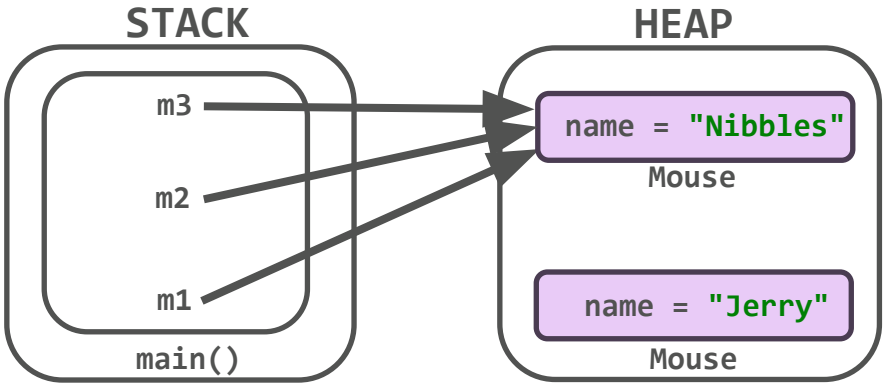
3. DESTROYING OBJECTS

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.      Mouse m1 = new Mouse();
5.      m1.name = "Jerry";
6.      Mouse m2 = new Mouse();
7.      m2.name = "Nibbles";
8.      m1 = m2;
9.      Mouse m3 = m1;
10.     m1 = null;
11.   }
12.  }
```

- Now we can see that Jerry doesn't have any references pointing to him, so at line 8 Jerry becomes **eligible** for garbage collection.



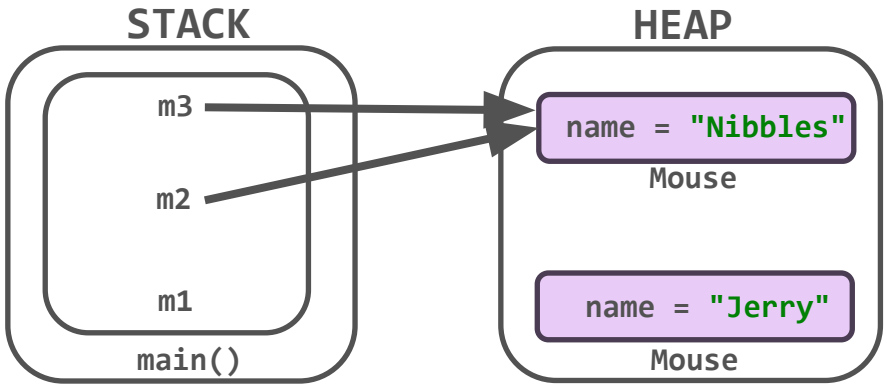
- At line 9, a new reference `m3` is placed on the stack, and it will point to to the object to which the `m2` reference is pointing to.

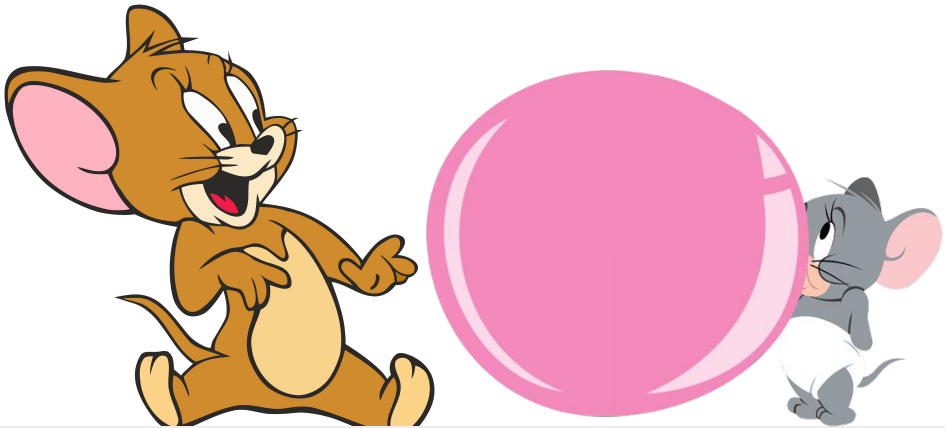


3. DESTROYING OBJECTS

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.      Mouse m1 = new Mouse();
5.      m1.name = "Jerry";
6.      Mouse m2 = new Mouse();
7.      m2.name = "Nibbles";
8.      m1 = m2;
9.      Mouse m3 = m1;
10.     m1 = null;
11.   }
12.  }
```

- At line 10 the reference m1 will point to **null**. We can illustrate this by just removing m1's pointer from the memory map.

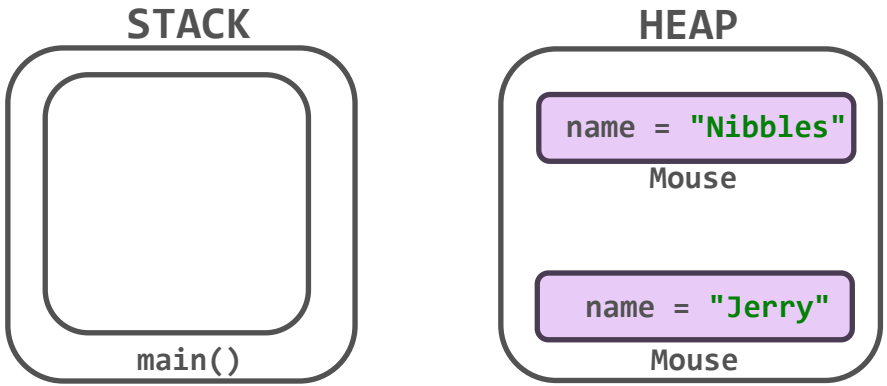




3. DESTROYING OBJECTS

```
1.  public class Home {
2.   public static void main(String
3.                              args[]){
4.     Mouse m1 = new Mouse();
5.     m1.name = "Jerry";
6.     Mouse m2 = new Mouse();
7.     m2.name = "Nibbles";
8.     m1 = m2;
9.     Mouse m3 = m1;
10.    m1 = null;
11.   }
12.  }
```

- At line 11 the body of the main method ends, and all reference go out of scope, since they were all local variables.



- So, now Nibbles also becomes **eligible** for garbage collection, since it doesn't have any references pointing to him.
- Not that hard, right ?
- Everything's there on the memory map, we just need to keep in mind a few rules and we won't encounter any issues.

STRINGS IN JAVA

AGENDA

1. String
2. StringBuilder
3. Equality

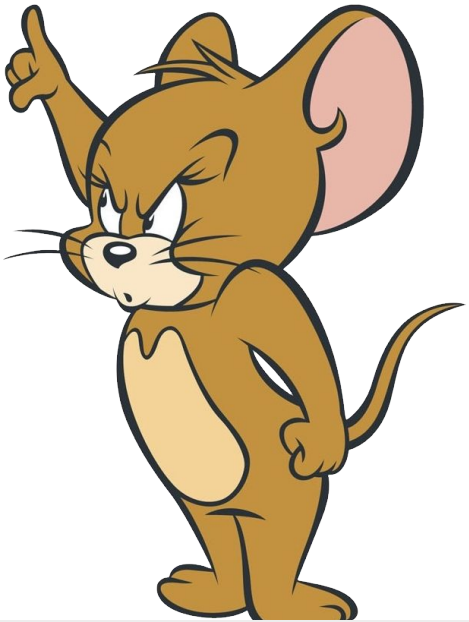
1. STRING

- A string is just a sequence of characters.
- In Java, we can create Strings using either of these approaches:

```
1. String s1 = "Jerry";
2. String s2 = new String("Jerry");
```

- Well, the second one is clear. We know that `String` is a class, and in order to use it, we must instantiate it using the **`new`** keyword.
- Both approaches will give us a reference pointing to the `String` object **`"Jerry"`**.
- There is a subtle, but very important difference between the two approaches.
- We will talk in a few slides in detail about them.

- **For now, please note that the `String` class in very special, and we don't need to instantiate it using the `new` keyword.**
- `Strings` are a common topic at interviews, not just because they are very used but **due to these subtleties that can separate amateurs from professionals**.



1. STRING | CONCATENATION

- We've already learned how to add numbers using the + operator.
- But what will be the result here:

```
1. String s1 = "1" + "2";
```

- It's `"12"`.
- Combining two `Strings` together is called **concatenation**.
- Regarding the process of concatenation, we need to keep in mind only these 3 rules:
  1. If both operands are numeric, + means numeric addition.
  2. If one operand is a String, + means concatenation.
  3. The expression is evaluated left to right.

```
1. System.out.println(1 + 2);
2. System.out.println("a" + "b");
3. System.out.println("a" + "b" + 3);
4. System.out.println(1 + 2 + "c");
```

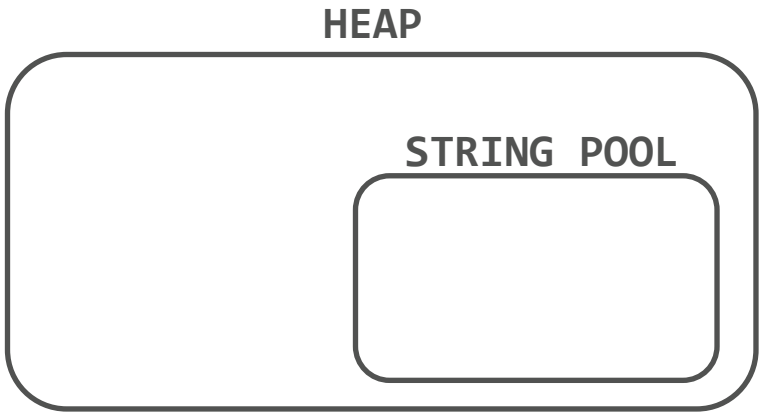
- Line 1 follows the first rule. Both operands are numbers, so Java uses numeric addition.
- The second example is `String` concatenation, following the second rule.
- Lines 3 and 4 combine the second and third rules. Since we start from the left, Java evaluates `"a" + "b"` to `"ab"`. Then the remaining expression is `"ab" + 3`. Since one of the operands is a `String`, we will perform concatenate.

1. STRING | STRING POOL

- Strings are very used in Java.
- In some production application they might use up to 50% of the memory.
- A study revealed that most of the string get duplicated.
- Java tackles this issue by introducing the **STRING POOL**, a special memory zone inside the HEAP, that **re-utilizes strings**.
- In order to place a String in the **String Pool**, we must create the String using a **literal**.

```
1. // s1 will go in the String Pool
2. String s1 = "Jerry";
3. // s2 will go in the Heap
4. String s2 = new String("Jerry");
```

- **When we will draw the memory map, we will draw the String Pool as a separated box for readability, but please keep in mind that the String Pool is part of the Heap.**
- This is who it really looks like:

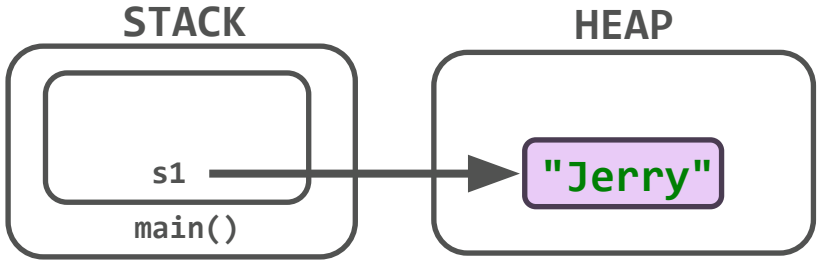


1. STRING | STRING POOL

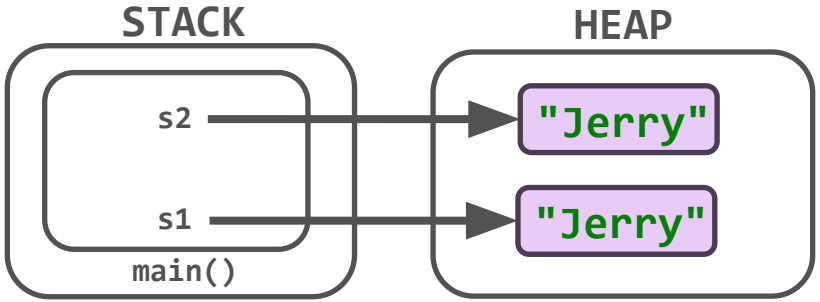
- Ok, how does the String Pool reutilize strings ? Great question !
- Let's draw the memory map for the following code snippet.

```
1. String s1 = new String("Jerry");
2. String s2 = new String("Jerry");
3. String s3 = "Tom";
4. String s4 = "Tom";
```

- The first line creates a reference s1 on the stack that will point to a String object Bob from the **heap**.
- That object gets placed in the heap, because it got created using the **new** keyword.
- **The new keyword, creates a new object on the HEAP, every time, no exception.**



- Line 2, again creates a **new** String object on the heap.

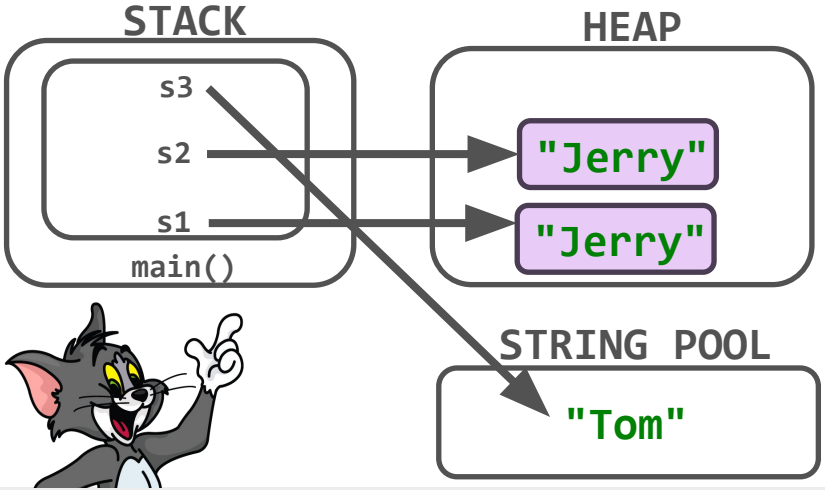


- Now we have two **"Jerry"** objects in the heap.

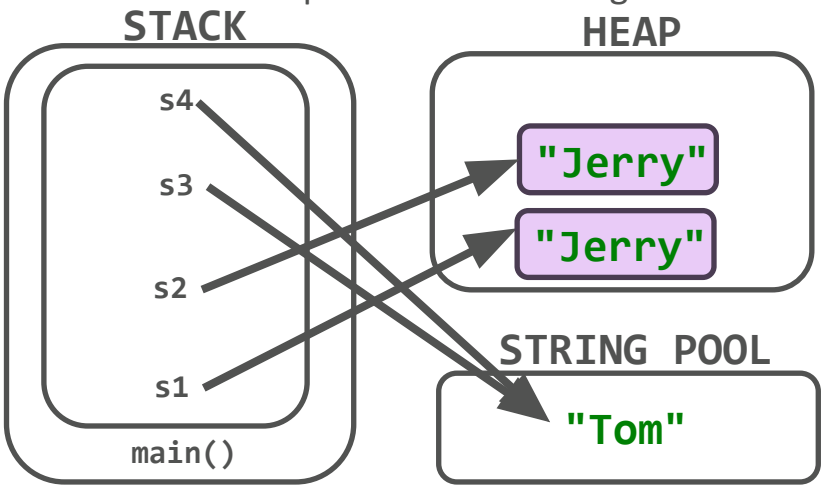
1. STRING | STRING POOL

```
1. String s1 = new String("Jerry");
2. String s2 = new String("Jerry");
3. String s3 = "Tom";
4. String s4 = "Tom";
```

- Moving to line 3, we can see that to the reference `s3`, a `String` literal is assigned.
- **So, it will go to the STRING POOL.**



- Line 4, assigns again a literal `"Tom"` to `s4`.
- Since it's a literal, the JVM looks in the String Pool to see if there is not already a `"Tom"` present.
- The JVM will find the `"Tom"` literal created at line 3 and won't create another `"Tom"`.
- It will make `s4` point to the existing `"Tom"`.



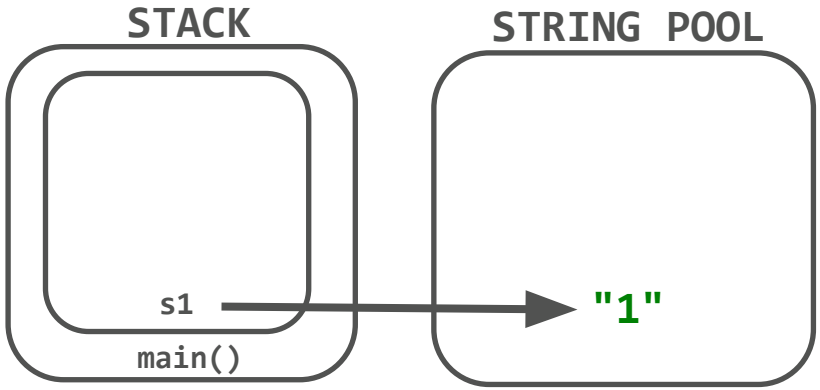
1. STRING | IMMUTABILITY

- **Another reason why `String` is a special class it's because it's immutable.**
- This means that once we create a String, we can't change it.
- This topic is **very** important.
- Here is a code snippet that my pop out at a regular technical interview. What will it print ?

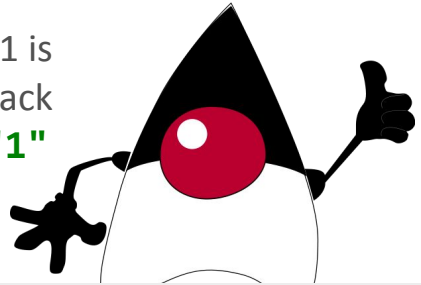
```
1. String s1 = "1";
2. String s2 = s1 + "2";
3. s2 + "3";
4. System.out.println(s2);
```

- **The easiest way to tackle these challenges is to draw the memory map we are already used to.**

- **We see that those Strings are created using literals, so this means they are placed in the String Pool.**



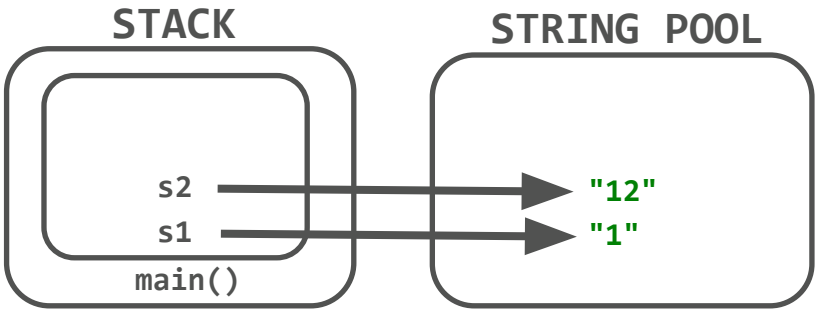
- At line 1, a reference s1 is placed on the stack which will point to `"1"` from the String Pool.



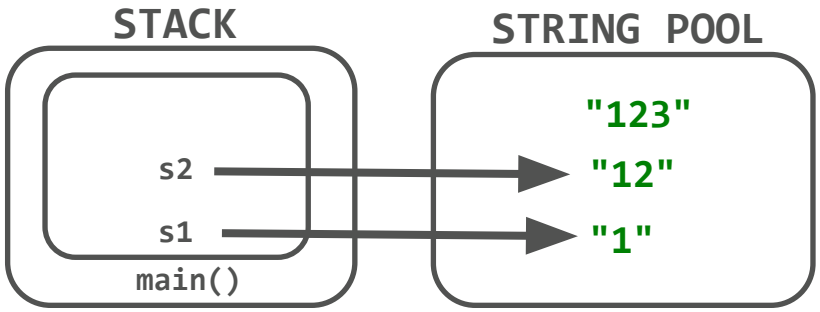
1. STRING | IMMUTABILITY

```
1. String s1 = "1";
2. String s2 = s1 + "2";
3. s2 + "3";
4. System.out.println(s2);
```

- Line 2 creates a reference `s2` on the stack.
- **But in the String Pool, it will create a new String `"12"`!**
- Because the String class is **immutable**, the object to which `s1` is pointing to, **is not altered** !



- At line 3, `s2` is a reference of type `String` which is immutable, so the object which is pointing to, will never get changed.
- So a new `String`, `"123"` will get placed in the String Pool.



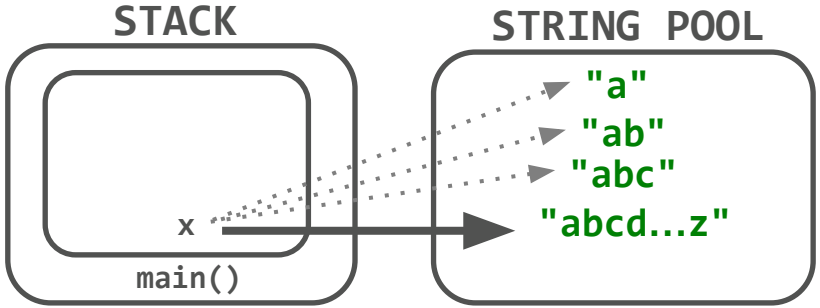
- The `"123"` `String` does not have a reference pointing to it, so this means it's **eligible for garbage collection**.
- So, the final output will be `"12"`.

2. STRINGBUILDER

- A program can create a lot of String objects very quickly.
- Let's look at the following code snippet.

```
1. String x = "";
2. for(char y = 'a'; y <= 'z'; y++){
3.     x += y;
4. }
```

- At line 1 we send the empty `String` to the String Pool and we assign the reference x to it.
- At line 3, we assign x to `'a'`, so the empty `String` becomes eligible for garbage collection.
- At the next iteration x will point to `'ab'`, `'a'` and will again become eligible for garbage collection.
- And this process repeats for all the letters.



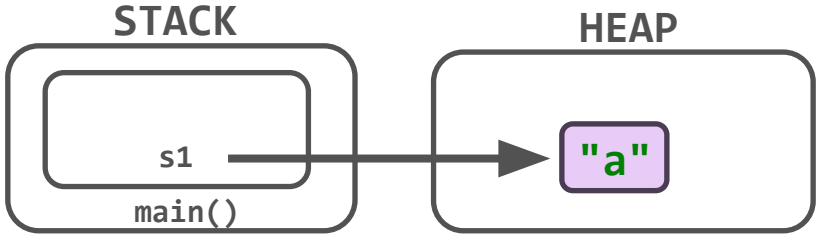
- We can see that in order to construct out desired String that contains the entire alphabet, another 25 String objects got created, which became eligible for garbage collection as soon as they got created.
- Java has a solution for this problem !
- It's called `StringBuilder`.

2. STRINGBUILDER

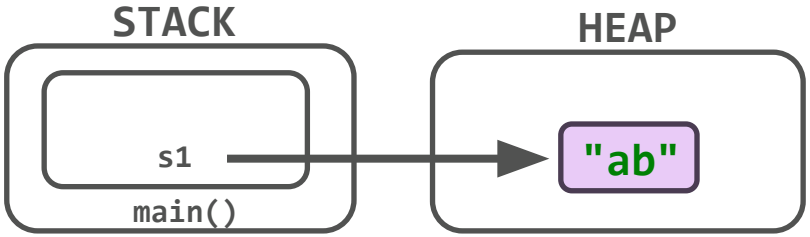
- `StringBuilder` is a **mutable** class.
- This means that we can modify an object after we create it.

```
1. StringBuilder x = new StringBuilder("");
2. for(char y = 'a'; y <= 'z'; y++){
3.     x.append(y);
4. }
```

- At line 1 we create a reference `x` that will point to a `StringBuilder` object that is placed on the **heap**. We know it's on the **heap**, because it was created using the **new** keyword.
- On the right we can see the memory map after the first iteration.



- Let's see the second iteration:



- Because the `StringBuilder` is **mutable**, the same object from the memory is changed.
- There are no other objects created or garbage collected.

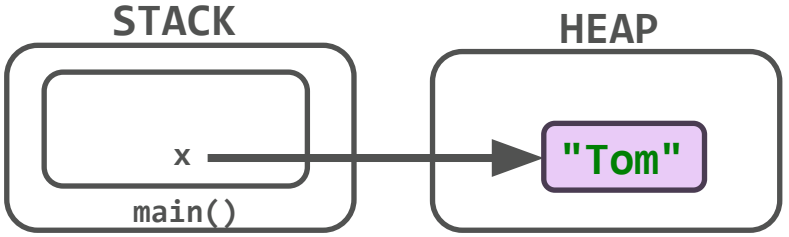
2. STRINGBUILDER

- Cool stuff, right ?
- Let's see if we got it. What will this code snippet print ?

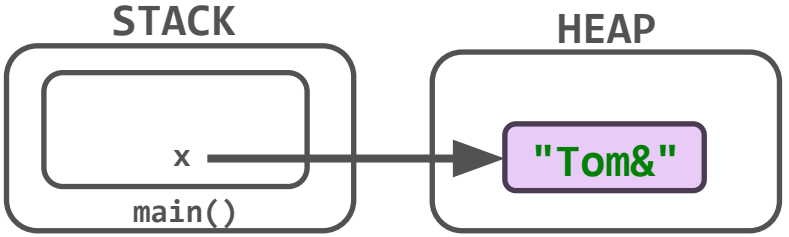
```
1.  StringBuilder x =
2.                  new StringBuilder("Tom");
3.  x.append("&");
4.  StringBuilder y = x.append("Jerry");
5.  System.out.println(x);
6.  System.out.println(y);
```

- Well, we can tackle this challenge easily using the memory map.
- This can easily appear at a technical interview.

- At line 1 we have:



- At line 3, we append `"&"` to the object to which the x reference is pointing to.



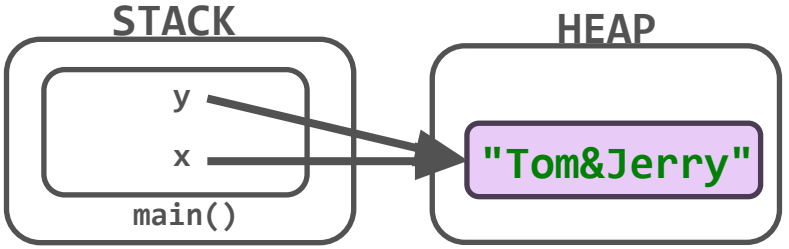
- We remember that `StringBuilder` is **mutable**, so we alter the existing object.

2. STRINGBUILDER

```
1.  StringBuilder x =
2.                  new StringBuilder("Tom");
3.  x.append("&");
4.  StringBuilder y = x.append("Jerry");
5.  System.out.println(x);
6.  System.out.println(y);
```

- On line 4, we have a new reference, y.
- Which will point to the object to which x is pointing to.
- But on x we append **"Jerry"**.

- So at line 4 we have:



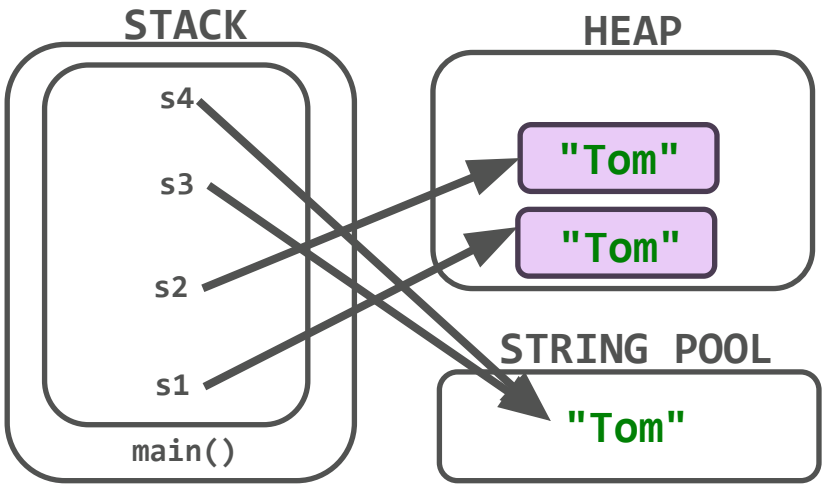
- So the final output will be **"Tom&Jerry"** twice.

3. EQUALITY

- In the previous chapters we learned how to use == to compare numbers.
- We use == check if references point to the same object.
- To understand this better, let's analyze the following code snippet:

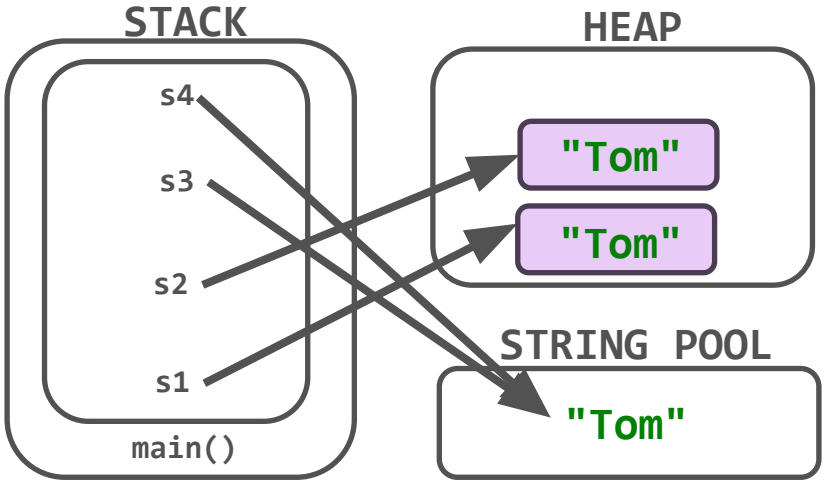
```
1. String s1 = new String("Tom");
2. String s2 = new String("Tom");
3. String s3 = "Tom";
4. String s4 = "Tom";
5. System.out.println(s1 == s2);
6. System.out.println(s1 == s3);
7. System.out.println(s3 == s4);
```

- First thing we need to do is draw the memory map.

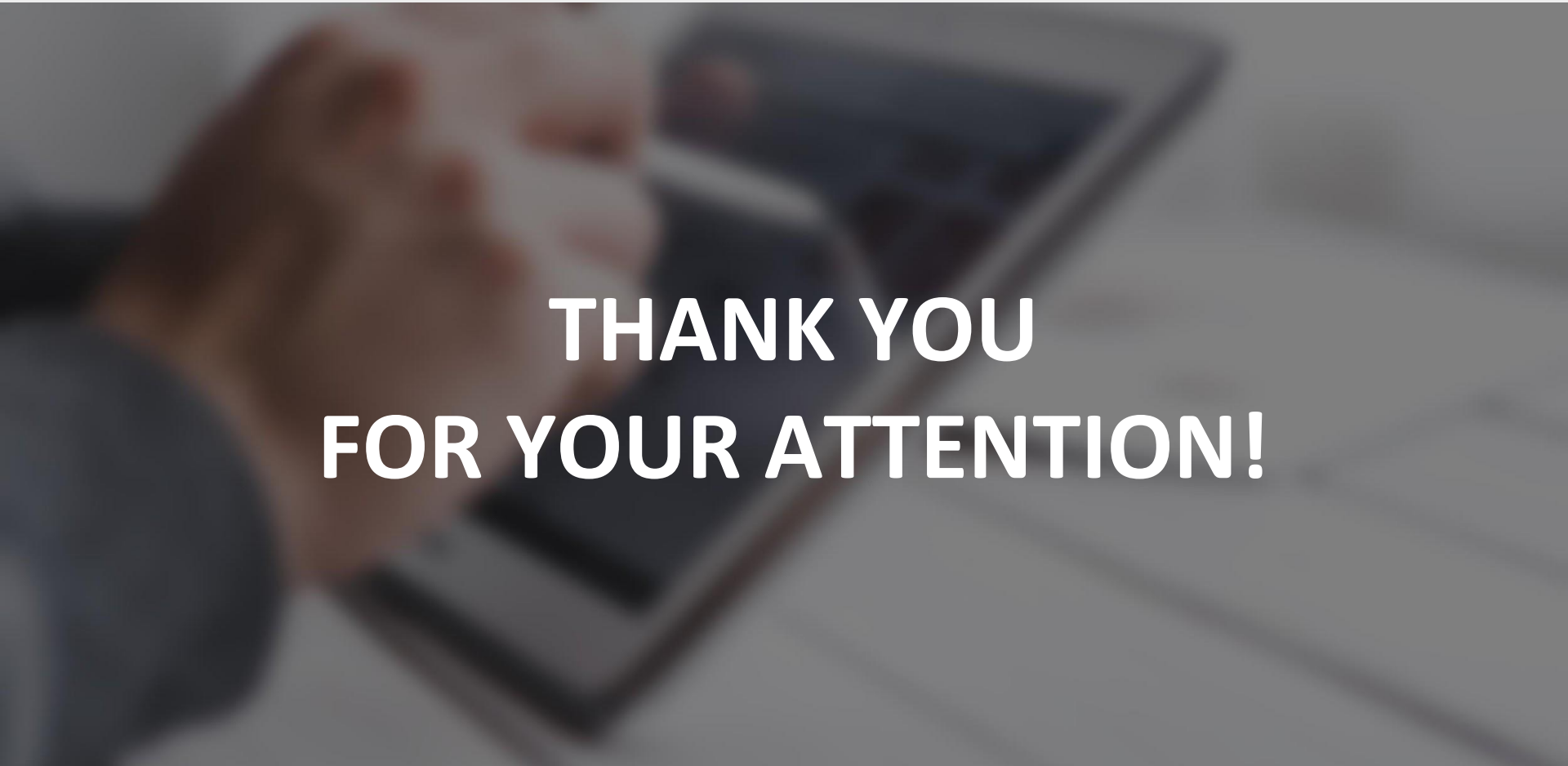


- So, == checks if the references are pointing to the same objects.

3. EQUALITY



- `s1` and `s2` are pointing to two different objects even if they both contain **`"Tom"`**.
- So `s1 == s2` will output **`false`**.
- Moving on to `s1 == s3`, we can see that `s1` is pointing to an object from the heap, and `s3` is pointing to the String Pool.
- So `s1 == s3` will output **`false`**.
- And finally, we can see that both `s3` and `s4` are pointing to the same object from the String Pool.
- So `s3 == s4` will output **`true`**.
- It gets easy once you get the hang of the memory map, right ?
- If we need to check if two strings have equal content, we should use the **`equals`** method.
- `s1.equals(s2)` will output **`true`**.
- `s1.equals(s3)` will output **`true`**.
- `s3.equals(s4)` will output **`true`**.



THANK YOU
FOR YOUR ATTENTION!