

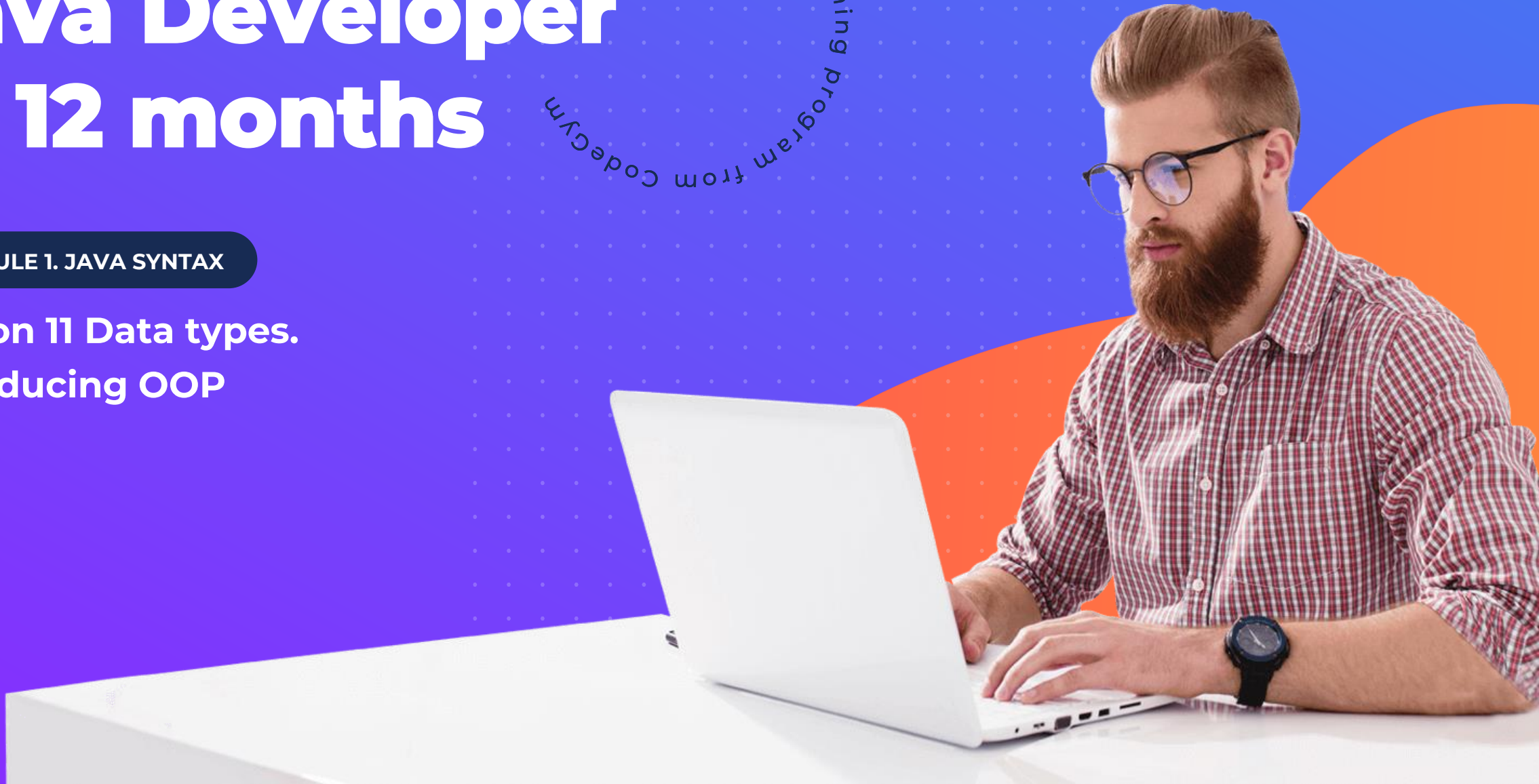


Mentor-supported training
program from CodeGym

Java Developer in 12 months

MODULE 1. JAVA SYNTAX

Lesson 11 Data types.
Introducing OOP



Lesson plan

- Primitive types
- Reference variables
- Objects
- Classes
- Introducing the principles of OOP



Java's primitive data types are the foundation for everything

Java has 8 basic primitive types. They are called primitive because the values of these types are not objects and are stored directly inside variables.

Type	Size in bytes	Value range:	Default value	Description
byte	1	-128 .. 127	0	The smallest integer type is a single byte
short	2	-32,768 .. 32,767	0	Short integer, two bytes
int	4	$-2 \cdot 10^9 \dots 2 \cdot 10^9$	0	Integer, 4 bytes
long	8	$-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$	0L	Long integer, 8 bytes
float	4	$-10^{38} \dots 10^{38}$	0.0f	Floating-point number, 4 bytes
double	8	$-10^{308} \dots 10^{308}$	0.0d	Double-precision floating point number, 8 bytes
boolean	1	true, false	false	Boolean type (only true and false)
char	2	0 .. 65,535	'\u0000'	Characters, 2 bytes, all non-negative



Default value

By the way, here's an important nuance. If you declare an instance variable (field) or a static class variable and do not immediately assign any value to it, then it is initialized with a default value. The table presents a list of these values.

Infinity

Floating-point numbers have another interesting feature: **they can store a special value denoting infinity**. And you can represent positive infinity and negative infinity.

Code	Note
<code>System.out.println(100.0 / 0.0);</code>	Infinity
<code>System.out.println(-100.0 / 0.0);</code>	-Infinity
<code>double a = 1d / 0d;</code> <code>double b = a * 10;</code> <code>double c = b - 100;</code>	<code>a == Infinity</code> <code>b == Infinity</code> <code>c == Infinity</code>

Not a Number (NaN)

Any operations involving infinity yield infinity. Well, most but not all.

Floating-point numbers can store another special value: NaN.

In mathematics, if you divide infinity by infinity, the result is undefined. In Java, if you divide infinity by infinity, the result is NaN.

Code	Note
<code>System.out.println(0.0 / 0.0);</code>	NaN
<code>double infinity = 1d / 0d;</code> <code>System.out.println(infinity / infinity);</code>	NaN
<code>double a = 0.0 / 0.0;</code> <code>double b = a * 10;</code> <code>double c = b - 100;</code> <code>double d = a + infinity;</code>	<code>a == NaN</code> <code>b == NaN</code> <code>c == NaN</code> <code>d == NaN</code>



Any operation with
NaN yields NaN.

Char type

Char values are stored in memory as numbers between 0 and 65536, representing Unicode characters. It looks approximately like this:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00A0		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï

Unicode is a special table (encoding) that contains the characters of nearly every written language in the world. And each character has its own number.

Code	Description
<code>char a = 'A';</code>	The a variable will contain the Latin letter A.
<code>char a = 65;</code>	The a variable will contain the Latin letter A. Its code is 65.
<code>char a = 0x41;</code>	The a variable will contain the Latin letter A. Its code is 65, which equals 41 in the hexadecimal system.
<code>char a = 0x0041;</code>	The a variable will contain the Latin letter A. Its code is 65, which equals 41 in the hexadecimal system. The two extra zeros don't change anything.
<code>char a = '\u0041';</code>	The a variable will contain the Latin letter A. Another way to define a character by its code.

Typecasting

There is free space inside a nesting doll. The larger the nesting doll, the more space there is.

We can easily put a smaller int nesting doll inside of the larger long nesting doll. It easily fits and there's nothing extra we need to do.



long



int

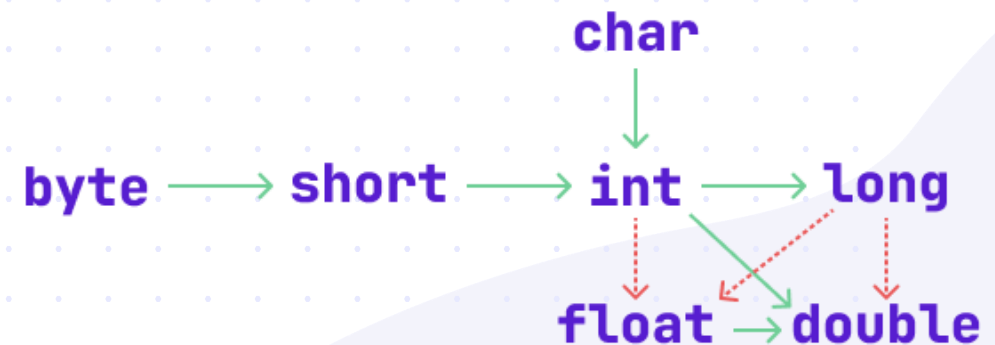


short



byte

Allowed type conversions



Solid lines indicate conversion without the loss of information.

Conversions that may involve information loss are indicated using **dashed lines**.

Widening type conversions

Variable stored in smaller baskets can always be assigned to variables stored in larger baskets.

Int, short and byte variables can be assigned to long variables.
Short and byte variables can be assigned to int variables.
And byte variables can be assigned to short variables.

Code	Description
<pre>byte a = 5; short b = a; int c = a + b; long d = c * c;</pre>	This code will compile just fine.

Narrowing type conversions

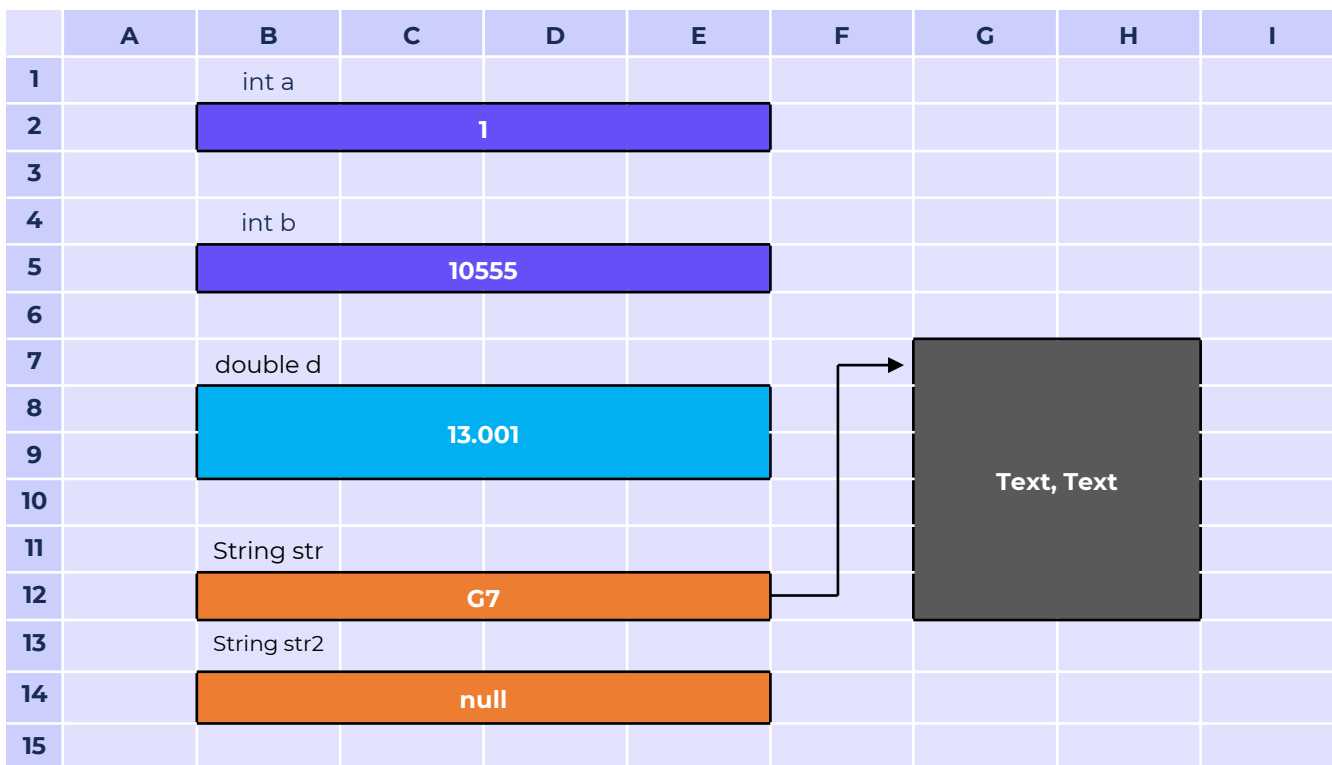
To make a narrowing conversion, you must explicitly specify the type to which you want to cast the value.

The typecast operator (which "casts types") is used for this.

Code	Description
<pre>long a = 1L; int b = (int) a; short c = (short) b; byte d = (byte) c;</pre>	Each time the typecast operator must be indicated explicitly
<pre>public static void main(String[] args) { int bigNumber = 10_000_000; short littleNumber = 1_000; littleNumber = (short) bigNumber; System.out.println(littleNumber); }</pre>	<p>We explicitly indicated that we want to force the int value into the short variable and that we accept responsibility for the results.</p> <p>Output: -27008</p>

Reference variables

Reference variables are variables of any type other than the primitive ones. Such variables contain only the address of the object (a reference to the object).



If a reference variable is declared but not initialized, then its value is **null**

Objects

Everything in Java is an object.

To create an object of a certain class, you need to use the **new** operator.

```
Class variable = new Class(arguments);
```

Where **Class** is the name of the class of the **variable** variable as well as the name of the class of the object to be created. And **variable** is a variable that stores a reference to the created object.

And **arguments** is a placeholder for a comma-delimited list of arguments passed to the constructor.

Code	Description
<code>String s = new String("Hello");</code>	Create a <code>String</code> object
<code>Scanner scanner = new Scanner("");</code>	Create a <code>Scanner</code> object
<code>int[] data = new int[10];</code>	Create an <code>int[]</code> : a container of 10 <code>int</code> elements

The created objects are called **objects of the class** or **instances of the class**, while the class is called the **type of the object**.

For example, the `s` variable stores a reference to a **String** object.

Classes

A class is essentially a template for an object. It defines what an object will look like and what functions the object will have.

Every object is an object of some class.

```
public class Cat {  
    String name;  
    int age;  
  
    public static void main(String[] args) {  
        Cat tom = new Cat();  
        tom.age = 3;  
        tom.name = "Tom";  
  
        System.out.println("Cat name: " + tom.name + ", age: " + tom.age);  
    }  
}
```

We created the Cat class and gave it two variables: String name and int age. These instance variables are called fields.

Basically, this is a template for all the cats that we will create in the future.

Classes are a response to increasingly larger program size.
A large program becomes less complex when it is broken up into classes.

Introducing the principles of OOP

The 4 main features that together form the object-oriented programming paradigm:

Abstraction

Encapsulation

Inheritance

Polymorphism

Abstraction

Abstraction means to identify the main, most significant characteristics of something with respect to the program, while simultaneously discarding anything minor and insignificant.

We're creating a filing system for company employees. To create "employee" objects, we wrote an Employee class. What characteristics are important to describe them in the company filing system?

Name, date of birth, SSN, and employee ID.

In the Employee class, we declare the following variables: String name, int age, long socialSecurityNumber, and long employeeId. And **we abstract away** unnecessary information like eye color.

```
public class Employee {  
    String name;  
    int age;  
    long socialInsuranceNumber;  
    long employeeId;  
}
```

Encapsulation

The goal of encapsulation is to improve interactions by making objects simpler. And the best way to simplify something is to hide anything complicated from prying eyes.

Encapsulation is exemplified by access modifiers (private, public, etc.), as well as setters and getters.

If you don't **encapsulate** the **Cat** class's age field, then anyone can write:

```
Cat.age = -1000;
```

The encapsulation mechanism lets us protect the **age** field with a setter method, where we can ensure that age cannot be set to a negative number.

Inheritance

A mechanism that lets you describe a new class based on an existing (parent) class. In doing so, the new class borrows the properties and functionality of the parent class.

Non-private fields and methods declared in parent classes can be used in descendant classes.

If all types of cars have **10** common fields and **5** identical methods, you just need to move them into the **Auto parent class**. And then you can use them in descendant classes without copying common code into each of the classes.



Polymorphism

This is the ability to work with several types as if they were the same type. Moreover, the objects' behavior will be different depending on their type.

For example, take the abstract Auto class, which is inherited by two concrete classes — SportsCar and Truck.

Both sports cars and trucks will have common characteristics and will perform actions common to all cars. These commonalities are specified in the abstract parent class, but their specific implementations may be different.

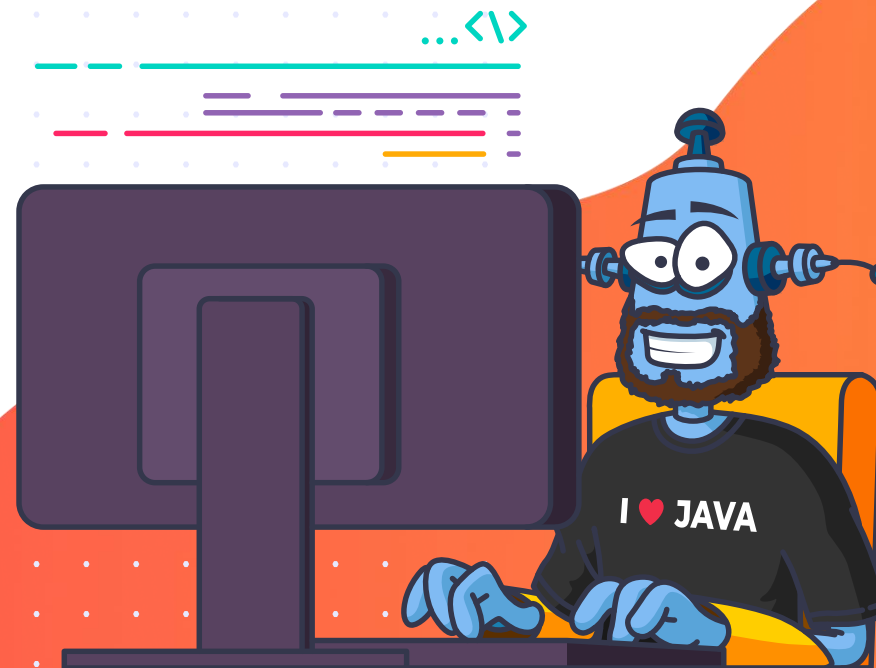
For example, the "start engine" action common to all automobiles could be implemented for a sports car by pressing a button, and for a truck — by using a key. The same result from different solutions. This is what polymorphism is all about.

Polymorphism is the ability to work with different data types in a consistent manner.

Homework

MODULE 1. JAVA SYNTAX

Complete Level 12



Answers to questions

