JAVA FUNDAMENTALS

Rafał Roppel

software
development
academy

HISTORY OF JAVA

1991

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project
- The small team of sun engineers called Green Team
- Firstly, it was called "Greentalk" and file extension was .gt
- After that, it was called Oak and was developed as a part of the Green project

1992

- Suns's Green Team creates interactive, handheld home-entertainment device that provides first glimpse of the potential for processor-independent programming language
- But it was too advanced for the digital cable television industry at the time
- It also featured a smart agent called “Duke,” who would later go on to become the mascot of Java

1993

- Sun's Green Project becomes FirstPerson, a wholly owned subsidiary of Sun Microsystems that focuses on building technology for highly interactive devices
- The group is later rolled back into Sun and engineers change their focus from embedded systems in electronic appliances like set-top boxes to online services

1994

- After brainstorming, the team refocused the platform on the WWW, and figured that the Internet could evolve into the same highly interactive medium that they had envisioned for cable TV
- Patrick Naughton and Jonathan Payne used the Oak to write WebRunner (an homage to the movie Blade Runner) later named HotJava
- Sun changed the name of the Oak language to Java (from Java coffee), after a trademark dispute from Oak Technology

1995

- Sun Microsystems released the first public implementation as JDK Alpha and Beta
- It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms
- Fairly secure and featuring configurable security, it allowed network- and file-access restrictions

1996

- The first version was released on January 23
- At the first-ever JavaOne developer conference, more than 6,000 attendees gather to learn more about Java technology
- Sun licenses java to operating systems vendors, including Microsoft, Apple, IBM, and others

1997

- Enterprise JavaBean (EJB) and Java Fundation Classes (JFC) are announced
- JDK 1.1 is released; it includes JavaBeans API and Java Database Connectivity (JDBC)
- Also extensive retooling of the AWT event model and "inner classes" added to the language

1998

- JDK 1.1 tops 2 million downloads
- JDK 1.2 is released and branded Java 2, and the version name changed do J2SE (Java 2 Standard Edition)
- Swing 1.0 and EJB 1.0 are released

1999

- Sun announces a redefined architecture for the Java platform
- HotSpot 1.0 is released and becomes the default Sun JVM in Java 1.3
- Sun announces
  - Java 2 Platform, Standard Edition (J2SE)
  - Java 2 Platform, Enterprise Edition (J2EE)
  - Java 2 Platform, Micro Edition (J2ME)

2000

- J2SE 1.3, codename: Kestrel bundled with HotSpot JVM, Java Sound, Java Naming and Directory Interface (JNDI), and Java Platform Debugger Architecture (JPDA)
- Sun unveils Java Web Start, which enables Java applications to be launched through a Web browser by simply clicking on a link to download and run the application
- Netbeans.org launches

2002

- JDK 1.4 (Merlin) is released
- Major changes included
  - Assertion
  - Regular Expression
  - XML processing
  - Cryptography and Secure Socket Layer (SSL)
  - Non-blocking I/O (NIO)
  - Logging

2003

- Java.net community and site is launched
- Java's new cofee cup logo debuts
- J2EE 1.4 is released

2004

- Java 2 Platform, Standard Edition 5.0 (Project Tiger) is released
- New features
  - Generics
  - Autoboxing/unboxing
  - Enhanced for
  - Static imports
  - Annotation/Metadata
- Java is verified for mobility

2005

- Java celebrates its tenth anniversary with huge celebrations at JavaOne
- The Java Champions program is launched to recognize leaders in the Java developer community
- Sun launches the GlassFish Project - an application server project for the Java EE platform

2006

- Java SE 6 (Mustang) is released. Replaced the name from J2SE to Java SE and dropped the .0 from the version number
- Main features
  - Web Services
  - Scripting
  - Security & Monitoring and Management
  - Compiler Access
  - Pluggable Annotations
- Java is Open Sourced

2009

- Project Coin is launched to enhence the Java programming language with an assortment of small changes
- The Java EE 6 is released with simplified development and deployment model, RESTful Web services, and the Java EE Web Profile
- The NetBeans IDE is developer.com's Product of the Year

2010

- The JCP approves Java 7 and Java 8 roadmaps
  - JSR-336 for Java 7
  - JSR-337 for Java 8
- The Java standard will progress through the JCP while the open source reference implementation will be delivered through the OpenJDK project

2011

- Java SE 7 (Dolphin) is released
- Feature additions for Java 7 include
  - JVM support for dynamic languages
  - Small language changes (Project Coin)
  - Concurrency utilities under JSR 166
  - New file I/O library to enhance platform independence
  - Fork/Join
- Lambda, Jigsaw, and part of Coin were dropped from Java 7

2014

- The most significant top-to-bottom changes to the Java language appear in Java SE 8 (Spider)
- New features
  - Language-level support for lambda expressions
  - Project Nashorn, a JavaScript runtime
  - Date and Time API
  - Streams

2017

- JDK 9 is released
- Designing and implementing a standard module system for the Java SE platform
- jshell - the Java Shell (a Java REPL)
- Reactive Streams

DESIGN GOALS OF THE JAVA

DESIGN GOALS OF THE JAVA

1. Simple, Object Oriented, and Familiar
2. Robust and Secure
3. Architecture Neutral and Portable
4. High Performance
5. Interpreted, Threaded, and Dynamic

Source: http://www.oracle.com/technetwork/java/intro-141325.html

BASIC ASSUMPTIONS OF LANGUAGE

1. Architecture neutral
2. Distributed
3. Dynamic
4. High Performance
5. Interpreted
6. Multithreaded
7. Object-Oriented
8. Platform independent
9. Portable
10. Robust
11. Secured
12. Simple

ARCHITECTURE NEUTRAL

- The compiler will generate an architecture-neutral object file meaning that compiled Java code (bytecode) can run on many processors given the presence of a Java runtime
- It's no need to recompile Java source code for 32-bit or 64-bit

DISTRIBUTED

- It is possible to create distributed applications in Java
- Programs can be design to run on computer networks
- Support for TCP, UDP, and basic Socket communication is excellent and getting better
- Also RMI and EJB are used for creating distributed applications

DYNAMIC

- Compiler doesn’t understand which method to called in advance
- JVM decide which method to called at run time
- All Java objects are dynamically allocated

HIGH PERFORMANCE

- Java is faster than traditional interpretation since byte code is "close" to native code
- Java uses Just-In-Time compiler - a computer program that turns Java byte codes into instructions that can directly be sent to compilers

INTERPRETED

- Java is a compiled programming Language which compiles the Java program into Java byte codes
- This JVM is then interpreted to run the program

MULTITHREADED

- A thread in Java refers to an independent program, executing concurrently
- We can write Java programs that deal with many tasks at once by defining multiple threads
- The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area

OBJECT-ORIENTED

- The code is organized as a combination of different types of objects which have data and behaviour
- Basic concepts of OOP: class, object, abstraction, encapsulation, inheritence, polimorphism

PLATFORM INDEPENDENT

- Java code can be run on multiple platforms (Windows, Linux, Mac/OS etc)
- When we compile Java code then .class file is generated by javac compiler
- These codes are readable by JVM and every operating system have its own JVM
- So, Java is platform independent but JVM is platform dependent

PORTABLE

- Output of a Java compiler is Non Executable Code i.e Bytecode
- Bytecode is executed by Java run-time system - Java Virtual Machine (JVM)

ROBUST

- Java uses strong memory management
- There are lack of pointers that avoids security problem
- There is exception handling and type checking mechanism
- There is automatic garbage collection

SECURED

- Java program is executed by the JVM
- The JVM prevent java code from generating side effects outside of the system
- Classloader: adds security by separating the package for the classes of the local file system from those that are imported from network sources
- Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects
- Security Manager: determines what resources a class can access such as reading and writing to the local disk

SIMPLE

- There is no confusing rarely used features: explicit pointers, operator overloading, multiple inheritance etc
- Syntax is based on C++
- Garbage Collection - no need to remove unreferenced objects

JAVA ENVIRONMENT

- **JDK** (Java Development Kit) - the software for programmers who want to write Java programs
- **JRE** (Java Runtime Environment) - the software for consumers who want to run Java programs
- **IDE** (Integrated Development Environment) - a software application which enables users to more easily write and debug Java programs

FIRST JAVA PROGRAM

HELLO, WORLD!

HELLO, WORLD!

```
public class Application {

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

```
$ javac Application.java
```

```
$ java Application
Hello, World!
```

DATA TYPES

DATA TYPES

- Java is strongly typed language
- Every variable must have a declared type
- There are eight primitive types
  - four are integer types
  - two are floating-point number types
  - one is character type char for individual characters
  - one is a boolean type for truth values

DATA TYPES - EXAMPLE

- 56 - int literal
- 523342.5432 - double literal
- 'g' - char literal
- true - boolean literal

INTEGER TYPES

- The integer types are for numbers without fractional parts
- Negative values are allowed
- Literal integers are implicitly ints
- In most situations, the int type is the most practical
- Integer expressions always result in an int-sized result

INTEGER TYPES

type	size	range	wrapper
byte	1 byte	-128 to 127	Byte
short	2 bytes	-2^{15} to $2^{15}-1$	Short
int	4 bytes	-2^{31} to $2^{31}-1$	Integer
long	8 bytes	-2^{63} to $2^{63}-1$	Long

INTEGER LITERALS

- Decimal (base 10)
- Octal (base 8)
- Hexadecimal (base 16)
- Binary (base 2)

DECIMAL LITERALS - EXAMPLE

- 343
- -4533
- 1000000
- 1_000_000

OCTAL LITERALS - EXAMPLE

Octal integers use only the digits 0 to 7. Octal form needs placing a zero in front of the number.

- 07 (decimal 7)
- 010 (decimal 8)
- -045 (decimal -37)
- 011000 (decimal 4608)
- 011_000 (decimal 4608)

HEXADECIMAL LITERALS - EXAMPLE

Hexadecimal (hex) numbers are constructed using 16 distinct symbols: 0 1 2 3 4 5 6 7 8 9 a b c d e f. Letters can be uppercase or lowercase. Hex form needs to be started by 0x or 0X.

- 0X0001 (decimal 1)
- -0x0101 (decimal -257)
- 0x7fffff decimal 2147483647)
- 0xCAFEBABE
- 0xDEAD_C0DE

BINARY LITERALS - EXAMPLE

Binary literals can use only the digits 0 and 1. Binary literals must start with either 0B or 0b

- 0b10101010 (decimal 170)
- 0B00000011 (decimal 3)
- 0b11111111 (decimal 255)
- -0b11111111 (decimal -255)
- 0B1100_0000 (decimal 192)

FLOATING-POINT TYPES

- The floating-point types denote numbers with fractional parts
- Negative values are allowed
- Floating-point numbers are implicitly doubles
- The name double refers to the fact that these numbers have twice the precision of the float type
- Floating-point numbers are not suitable for financial calculations in which roundoff errors cannot be tolerated

FLOATING-POINT TYPES

type	size	range	wrapper
float	4 bytes	IEEE754	Float
double	8 bytes	IEEE754	Double

FLOATING-POINT LITERALS - EXAMPLE

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d. The floating point types can also be expressed using E or e (for scientific notation)

- 123.4
- 1.234e2
- -34.623F
- 3.14_15F
- 3.14d (d is optional)

CHAR TYPE

- A char literal is represented by a single character in single quotes
- You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u
- Unicode code units can be expressed as hexadecimal values that run from \u0000 to \uFFFF
- Characters are just 16-bit unsigned integers under the hood

CHAR TYPE

type	size	range	wrapper
char	2 bytes	\u0000 to \uFFFF	Character

CHAR TYPE - EXAMPLE

- 'a'
- 'A'
- '\u004E' (letter 'N')
- '\u005D' (sign ']')

SPECIAL CHARACTERS

escape sequence	name	unicode value
\b	backspace	\u0008
\t	tab	\u0009
\n	linefeed	\u000a
\r	carriage return	\u000d
\"	double quote	\u0022
\'	single quote	\u0027
\\	backslash	\u005c

BOOLEAN TYPE

- A boolean value can be defined only as true or false
- It is used for evaluating logical conditions
- Wrapper class for boolean values is Boolean

DATA TYPES - EXERCISES

- integer types
- float-pointing types
- characters
- booleans

DATA TYPES

WHAT HAVE WE LEARNED ABOUT NUMERAL DATA TYPES

- Java is a strongly typed language
- Every variable must have a declared type
- There are eight primitive types, like
  - four integers
  - two floating-points
  - one character
  - one boolean

- Integers
  - byte, short, **int**, long
- Floating-points
  - float, **double**
- Integer literals representation
  - Decimal, Octal, Hexadecimal, Binary

VARIABLES

VARIABLES

- A variable is a storage location in a computer program
- Each variable has a name and holds a value
- In Java, every variable has a type
- Good practice - use a short, descriptive, meaningful variable name!
- There are four types of variables in java
  - block
  - local
  - instance
  - static

VARIABLES

Declaration

```
int width;
boolean done;
double factor;
```

Declaration with initialization

```
int width = 1920;
done = false;
double factor = 4.127;
```

VARIABLES

- A variable name must begin with a letter and must be a sequence of letters or digits
- A letter is defined as 'A'–'Z', 'a'–'z', '_', '$', or any Unicode character that denotes a letter in a language
- Similarly, digits are '0'–'9' and any Unicode characters that denote a digit in a language
- The first letter should be lowercase, and then normal CamelCase rules should be used
- All characters in the name of a variable are significant and case is also significant

JAVA KEYWORDS

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

VARIABLES - EXERCISES

- declarations
- initializations

VARIABLES

WHAT HAVE WE LEARNED ABOUT VARIABLES

- A variable is a storage location in a computer program
- Each variable has a name, a type and holds a value
- There are four types of variables in java
  - block, local, instance, static
- All characters in the name of a variable are significant and case-sensitive
- You cannot use java keywords as names for variables (or as any other identifiers)

OPERATORS

OPERATORS

- Data in Java is manipulated using operators
- Java operators produce new values from one or more operands
- Operands are the things on the right or left side of the operator
- The result of most operations is either a boolean or numeric value

ASSIGNMENT OPERATORS

=, +=, -=, *=, /=

RELATIONAL OPERATORS

<, <=, >, >=, ==, !=

ARITHMETIC OPERATORS

+, -, *, /, %, ++, --

LOGICAL OPERATORS

&, |, ^, !, &&, ||

BITWISE OPERATORS

&, |, ^, ~

BIT-SHIFTING OPERATORS

<<, >>, >>>

CONDITIONAL OPERATOR

? :

INSTANCEOF OPERATOR

`instanceof`

OPERATORS - EXERCISES

- Assignment Operators
- Relational Operators
- Arithmetic Operators
- Logical Operators
- Bitwise Operators
- Bit-Shifting Operators
- Conditional Operator
- instanceof Operator

OPERATORS

WHAT HAVE WE LEARNED ABOUT OPERATORS

- Relational operators always result in a `boolean` value (`true` or `false`)
- When comparing reference variables, == returns `true` only if both references refer to the same object.
- `instanceof` is for reference variables only; it checks whether the object is of a particular type

- The `instanceof` operator can be used only to test objects (or `null`) against class types that are in the same class hierarchy
- Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others

- If either operand is a `String`, the + operator concatenates the operands
- If both operands are numeric, the + operator adds the operands
- Logical operators work with two expressions (except for `!`) that must resolve to boolean values

CASTS

CASTS

- Numeric conversions are possible in Java
- Conversions in which loss of information is possible are done by means of casts
- The syntax for casting is to give the target type in parentheses, followed by the variable name

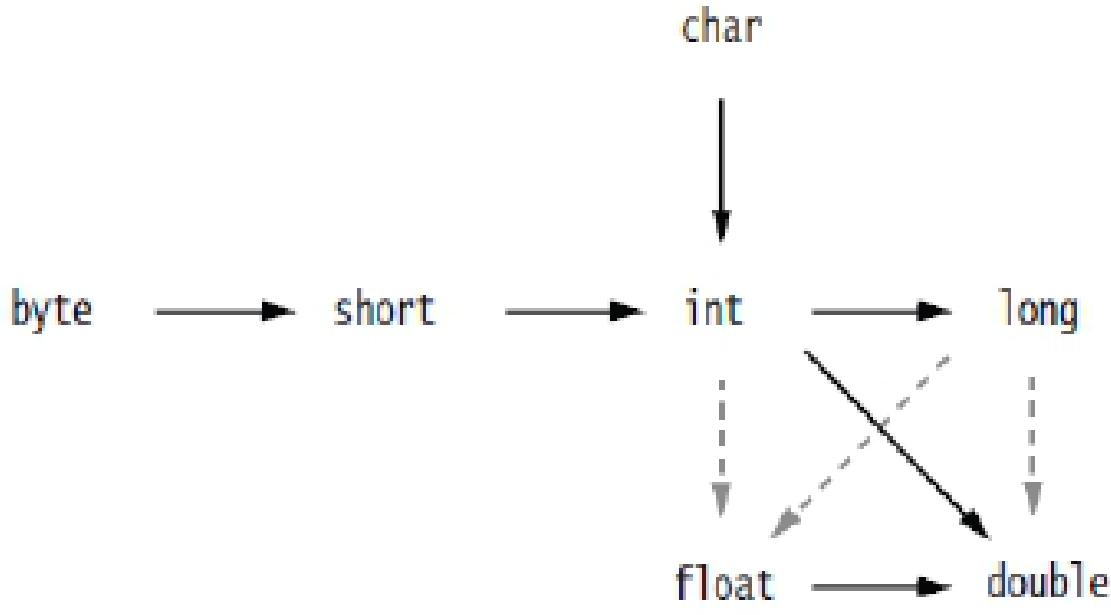
```
double y = 89.832;
int x = (int) y;
```

CASTS

primitive type	reference type
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character
void	Void

CASTS

Legal conversions between numeric types



OPERATOR - PRECEDENCE

- Operator precedence defines how an expression evaluates when several operators are present
- Multiplication and division happen before addition and subtraction
- Use parentheses to make the order of evaluation explicit
- If no parentheses are used, operations are performed in the hierarchical order indicated
- Operators on the same level are processed from left to right, except for those that are right-associative

OPERATOR - PRECEDENCE

operators	associativity
[], ., () (method call)	left to right
!, ~, ++, --, + (unary), - (unary), () (cast), new	right to left
+, -, *, /, %, <<, >>, >>>, <, <=, >, >=	left to right
==, !=, instanceof, &, ^, |, &&, ||	left to right
?:, =, +=, -+, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	right to left

CASTS - EXERCISES

- casts
- precedence

CASTS

WHAT HAVE WE LEARNED ABOUT CASTS

- Numeric conversions are possible in Java
- Conversions in which loss of information is possible are done by means of casts
- Boxing, unboxing
- Legal conversions between numeric types
- Operators precedence

STRINGS

STRINGS

- A string is a sequence of characters
- Java strings are sequences of Unicode characters
- Java does not have a built-in string type, strings are objects
- The standard library contains a predefined class called String
- Strings are immutable objects!

STRINGS

```
String a = "abc";
String b = new String("abc");
```

equality

```
System.out.println(a == b);     \\ false
System.out.println(a.equals(b); \\ true
```

STRINGS OPERATIONS - SUBSTRINGS

```
String text = "Hello, World!";
String s = text.substring(0, 5);
```

creates a new string consisting of the characters "Hello"

```
String t = text.substring(7);
```

creates a new string consisting of the characters "World!"

STRINGS OPERATIONS - CONCATENATION

```
String h = "Hello";
String w = "World!";
String text = h + ", " + w;
```

creates a new string consisting of the characters "Hello, World!"

USEFUL METHODS FROM API

- char charAt(int index)
- int compareTo(String other)
- boolean endsWith(String suffix)
- boolean equals(Object other)
- boolean equalsIgnoreCase(String other)
- int indexOf(String str)
- int lastIndexOf(String str)

USEFUL METHODS FROM API

- int length()
- String replace(CharSequence oldString, CharSequence newString)
- boolean startsWith(String prefix)
- String substring(int beginIndex)
- String toLowerCase()
- String toUpperCase()
- String trim()

STRINGBUILDER

- Should be used when you have to make a lot of modifications to strings of characters
- Every time you concatenate strings, a new String object is constructed
- This is time-consuming and wastes memory - use StringBuilder class to avoid this problem
- Prefer StringBuilder to StringBuffer

STRINGS - EXERCISES

- operations
- useful methods
- StringBuilder

STRINGS

WHAT HAVE WE LEARNED ABOUT STRINGS

- A Java string is a sequence of Unicode characters
- Strings are immutable objects!
- String equality (== vs `equals`)
- String operations
- String API
- StringBuilder

LOOPS

LOOPS

Loops let repeat a block of code as long as some condition is `true`, or for a specific number of iterations.

- `while` loop
- `do while` loop
- `for` loop
- enhanced `for` loop

WHILE LOOP

- The `while` loop executes a block or statement as long as some condition is `true`
- Loop will never execute if the condition is `false` at the outset

```
while (expression) {
    // statement or block of code
}
```

WHILE LOOP - EXAMPLE

```
int x = 3;
while (x > 1) {
    System.out.println(x);
    x--;
}
```

```
while (true) {
    System.out.println("Endless loop...");
}
```

DO WHILE LOOP

- The `do while` loop is quite similar to the `while` loop
- The code in a do loop is guaranteed to execute at least once
- The expression is not evaluated until after the `do` loop's code is executed

```
do {
  // statement or block of code
} while (expression);
```

DO WHILE LOOP - EXAMPLE

```
do {
  System.out.println("Greetings from do while loop!");
} while (false);
```

```
do {
  System.out.println("Endless loop...");
} while (true);
```

FOR LOOP

- Is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block
- Has three main parts
  - Declaration and initialization of variables
  - The boolean expression (conditional test)
  - The iteration expression

```
for (initialization; condition; iteration) {
    // statement or block of code
}
```

FOR LOOP - EXAMPLE

```
for (int x = 0; x < 10; x++) {
    System.out.println("x is " + x);
}
```

```
for ( ; ; ) {
    System.out.println("Endless loop...");
}
```

ENHANCED FOR LOOP

- The enhanced for loop is a specialized for loop that simplifies looping through an array or a collection
- Has two main parts
  - Declaration - the newly declared block variable
  - Expression - must evaluate to the array or collection (instance of `java.lang.Iterable`)

```
for (declaration : expression) {
    // statement or block of code
}
```

ENHANCED FOR LOOP - EXAMPLE

```
for (Animal a : animals) {
    System.out.println(a);
}
```

```
int[] arrayOfInts = {1, 2, 3, 4, 5, 6};
for (int n : arrayOfInts) {
    System.out.println(n);
}
```

LOOP CONTROL FLOW

code in loop	behaviour
`break`	execution jumps immediately to the first statement after the loop
`continue`	stops just the current iteration
`return`	execution jumps immediately back to the calling method
`System.exit()`	all program execution stops; the VM shuts down

LOOPS - EXERCISES

- `while` loop
- `do while` loop
- `for` loop
- enhanced `for` loop
- loop control flow
- nested loops

LOOPS

WHAT HAVE WE LEARNED ABOUT LOOPS

- Loops let repeat a block of code as long as we want
- There are four types of loops
  - `while`, `do while`, `for`, enhanced `for`
- Loop control flow
  - `break`, `continue`, `return`, `System.exit()`
- Loop in loop - nested loops

CONTROL FLOW

IF STATEMENT

- The `if` statement is commonly referred to as decision statements
- Rules for using `else` and `else if`
  - You can have zero or one `else` for a given `if`, and it must come after any `else ifs`.
  - You can have zero to many `else ifs` for a given `if` and they must come before the (optional) `else`
  - Once an `else if` succeeds, none of the remaining `else ifs` nor the `else` will be tested

IF STATEMENT

```
if (booleanExpression) {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else if (booleanExpression) {
    // statement or block of code
  ...
} else {
    // statement or block of code
}
```

IF STATEMENT - EXAMPLE

```
if (points >= 100) {
    System.out.println("You win!");
}
```

```
if (age < 18) {
    System.out.println("You are teenager!");
} else {
    System.out.println("You are adult!");
}
```

```
if (age < 18) {
    System.out.println("You are teenager!");
} else if (age > 100) {
    System.out.println("You are very old!");
} else {
    System.out.println("You are adult!");
}
```

SWITCH STATEMENT

- The `if/else` construct can be cumbersome when you have to deal with multiple selections with many alternatives
- The `switch` statement provides a cleaner way to handle complex decision logic

```
switch (expression) {
    case constant1:
        // statement or block of code
    case constant2:
        // statement or block of code
    ...
    default:
        // statement or block of code
}
```

SWITCH STATEMENT - EXAMPLE

```
switch (direction) {
    case 'n':
        System.out.println("You are going North");
        break;
    case 's':
        System.out.println("You are going South");
        break;
    case 'e':
        System.out.println("You are going East");
        break;
    case 'w':
        System.out.println("You are going West");
        break;
    default:
        System.out.println("Bad direction!");
```

SWITCH STATEMENT

- A `switch`'s expression must evaluate to a `char`, `byte`, `short`, `int`, an `enum`, and a `String`
- You won't be able to compile if you use types of `long`, `float`, and `double`
- A `case` constant must evaluate to the same type that the `switch` expression can use
- A `case` constant must be a compile-time constant!
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value

CONTROL FLOW - EXERCISES

- if statement
- switch statement

CONTROL FLOW

WHAT HAVE WE LEARNED ABOUT CONTROL FLOW

- The only legal expression in an `if` statement is a `boolean` expression
- `switch` statements can evaluate only to `enums` or the `byte`, `short`, `int`, `char`, and `String` data types
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value

ARRAYS

ARRAYS

- Arrays are the fundamental mechanism in Java for collecting multiple values
- Arrays can hold primitives or objects, but the array itself is always an object
- You access each individual value through an integer *index*
- Arrays are indexed beginning with zero
- An `ArrayIndexOutOfBoundsException` occurs if you use a bad index value
- Arrays have a `length` attribute whose value is the number of array elements

ARRAYS - EXAMPLES

declaration

```
dataType[] array; // recommended
dataType []array;
dataType array[];
```

```
int[] arrayOfInts;
String[] arrayOfStrings;
```

ARRAYS - EXAMPLES

declaration and instantiation

```
dataType[] array = new dataType[size]; // recommended
dataType []array = new dataType[size];
dataType array[] = new dataType[size];
```

```
int[] arrayOfInts = new int[5];
// initialization
arrayOfInts[0] = 10;
arrayOfInts[1] = 15;
arrayOfInts[2] = 20;
arrayOfInts[3] = 25;
arrayOfInts[4] = 30;

String[] arrayOfStrings = new String[2];
// initialization
arrayOfStrings[0] = "Tree";
arrayOfStrings[1] = "Forest";
```

ARRAYS - EXAMPLES

declaration, instantiation and initialization

```
dataType[] array = new dataType[]{el1, el2, ... , eln};
dataType[] array = {el1, el2, ... , eln};
```

```
int[] arrayOfInts = new int[]{10, 15, 20, 25, 30};
int[] arrayOfInts = {10, 15, 20, 25, 30};

String[] arrayOfStrings = new String[]{"Tree", "Forest"};
String[] arrayOfStrings = {"Tree", "Forest"}
```

ARRAYS - EXAMPLES

accessing

```
int[] arrayOfInts = {10, 15, 20, 25, 30};
System.out.println(arrayOfInts[0]);    // print 10
System.out.println(arrayOfInts[2]);    // print 20
System.out.println(arrayOfInts[4]);    // print 30

for (int i = 0; i < arrayOfInts.length; i++) {
    System.out.print(arrayOfInts[i] + " ");
}
// print 10 15 20 25 30

for (int i : arrayOfInts) {
    System.out.print(i + " ");
}
// print 10 15 20 25 30
```

ARRAYS - EXERCISES

- declaration
- instantation
- initialization
- accessing
- multidimensional arrays

ARRAYS

WHAT HAVE WE LEARNED ABOUT ARRAYS

- Arrays can hold primitives or objects, but the array itself is always an object
- When you declare an array, the brackets can be to the left or to the right of the variable name
- It is never legal to include the size of an array in the declaration
- An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array.

OBJECT-ORIENTED PROGRAMMING

CLASS

A class is the template or blueprint from which objects are made. Describes the behavior/state that the object of its type support.

OBJECT

Objects have states and behaviors.

Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating.

An object is an instance of a class.

OBJECT'S STATE

How does the object react when you invoke those methods?

OBJECT'S BEHAVIOR

What can you do with this object, or what methods can you apply to it?

OBJECT'S IDENTITY

How is the object distinguished from others that may have the same behavior and state?

CLASS DECLARATION

Class declarations can include these components, in order:

1. Modifiers such as *public*, *private* (if any), and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The class body, surrounded by braces, {}.

CLASS DECLARATION - EXAMPLE

```
class Bicycle {
    // class body
}
```

or

```
public class Bicycle {
    // class body
}
```

or

```
private class Bicycle {
    // class body
}
```

OBJECT-ORIENTED PROGRAMMING

WHAT HAVE WE LEARNED ABOUT OBJECT-ORIENTED PROGRAMMING

- Class vs Object
- State, behavior, identity
- Class declaration

FIELDS, METHODS, CONSTRUCTORS, PACKAGES, IMPORTS

ACCESS MODIFIERS

There are four *access controls* (levels of access) but only three *access modifiers*.

- `public` - visible to the world
- `protected` - visible to the package and all subclasses
- default - visible to the package
- `private` - visible to the class only

VARIABLE SCOPE

The scope of a variable is the part of the code in which you can access it. There are four basic variable scopes

- static - they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM
- instance - they are created when a new instance is created, and they live until the instance is removed
- local - they live as long as their method remains on the stack
- block - live only as long as the code block is executing

MEMBER VARIABLES

There are several kinds of variables:

- Member variables in a class — called *fields*.
- Variables in a method or block of code — called *local variables*.
- Variables in method declarations - called *parameters*.

FIELDS

The `Bicycle` class uses the following lines of code to define its fields:

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
}
```

FIELDS

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
}
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers.
2. The field's type.
3. The field's name.

FIELDS

- You can use any access modifier, but it is good practice to use `private` for fields
- All variables must have a type. You can use primitive types such as `int`, `float`, `boolean`, etc. Or you can use reference types, such as strings, arrays, or objects.
- All variables should have meaningful names.

CONSTRUCTORS

Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

CONSTRUCTORS - EXAMPLE

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
    }
}
```

INSTANTIATING A CLASS

The `new` operator instantiates a class by allocating memory for a new object and returning a reference to that memory.

The `new` operator also invokes the object constructor.

"instantiating a class" means the same thing as "creating an object."

INSTANTIATING A CLASS

The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The `new` operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type

INSTANTIATING A CLASS - EXAMPLE

```
Bicycle bike = new Bicycle(75, 2, 20);
```

or

```
Cat garfield = new Cat("Gerfield");
```

or

```
Integer age = new Integer(34);
```

METHODS

Methods are fundamental building blocks of Java programs. Each Java method is a collection of statements that are grouped together to perform an operation.

METHODS DECLARATION

Method declarations have six components, in order:

1. Modifier - it defines the access type of the method and it is optional to use.
2. Return type – method may return a value.
3. Method name.
4. Parameter list in parenthesis - it is the type, order, and number of parameters of a method.
5. Method body - defines what the method does with the statements.

METHODS DECLARATION - EXAMPLE

```
public int sum(int a, int b) {
    // return a + b;
}
```

```
void draw(String s) {
    // perform some draw functions
}
```

```
private boolean isNew() {
    // return true or false according to some rules
}
```

METHODS CALLING - EXAMPLE

Consider

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
    }

    public int getCadence() {
        return cadence;
    }
}
```

METHODS CALLING - EXAMPLE

```
Bicycle bike = new Bicycle(75, 2, 20);
bike.getCadence();   // should return 75
System.out.println(bike.getCadence();   // should print 75

int cadence = bike.getCadence();
System.out.println("Cadence is: " + cadence);
```

PACKAGES

Packages are used in Java in order

- to prevent naming conflicts
- to control access
- to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

PACKAGES

Some of the existing packages in Java are

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for reading and writing (input and output)
- **java.util** - contains the collections framework, date and time facilities, internationalization, and miscellaneous utility classes

PACKAGES CREATING

While creating a package, you should choose a name for the package and include a `package` statement along with that name at the top of every source file that contains types that you want to include in the package.

PACKAGES CREATING - EXAMPLE

```
package vehicle;

public class Bicycle {
    // class body
}
```

or

```
package ro.sdacademy.animals.mammals;

public class Cat {
    // class body
}
```

IMPORTS

If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, `java.util.Scanner` refers to the `Scanner` class in the `java.util` package:

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

IMPORTS - EXAMPLE

Instead, you can import a name with an `import` statement:

```
import java.util.Scanner;
```

or import all classes from the `java.util` package

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

and then you can write:

```
Scanner in = new Scanner(System.in);
```

IMPORTS

An automatic import `java.lang.*;` statement has been placed into every source file.

You don’t need to import other classes in the same package.

SOURCE FILE DECLARATION RULES

- There can be only one `public` class per source code file.
- If there is a `public` class in a file, the name of the file must match the name of the `public` class. For example, a class declared as

```
public class Cat { }
```

  must be in a source code file named `Cat.java`.
- A file can have more than one nonpublic class.

SOURCE FILE DECLARATION RULES

- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If the class is part of a package, the `package` statement must be the first line in the source code file, before any `import` statements that may be present.

SOURCE FILE DECLARATION RULES

- If there are `import` statements, they must go between the `package` statement (if there is one) and the class declaration. If there isn't a `package` statement, then the `import` statement(s) must be the first line(s) in the source code file.
- If there are no `package` or `import` statements, the class declaration must be the first line in the source code file.

SOURCE FILE DECLARATION RULES

- `import` and `package` statements apply to **all** classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages or use different imports.
- Files with no `public` classes can have a name that does not match any of the classes in the file.

FIELDS, METHODS, CONSTRUCTORS, PACKAGES, IMPORTS

WHAT HAVE WE LEARNED ABOUT FIELDS, METHODS, CONSTRUCTORS, PACKAGES, IMPORTS

- Levels of access vs access modifiers
- Variable scope
- Fields, constructor, method
- Packages & imports
- Source file declaration rules

STATIC FIELDS, METHODS AND IMPORTS

STATIC FIELDS AND METHODS

The keyword `static` indicates that the particular member belongs to a type itself, rather than to an instance of that type.

This means that only one instance of that static member is created which is shared across all instances of the class.

STATIC FIELDS

- Static variable in Java is variable which belongs to the class and initialized only once at the start of the execution
- It is a variable which belongs to the class and not to object (instance)
- A single copy to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn’t need any object

STATIC FIELDS - DECLARATION

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    static int count = 0;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
        count++;
    }
}
```

STATIC FIELDS - ACCESS

```
Bicycle bike = new Bicycle(75, 2, 20);
System.out.println(Bicycle.count);   // should print 1

Bicycle anotherBike = new Bicycle(80, 4, 25);
System.out.println(Bicycle.count);   // should print 2

// should prints true in both cases
System.out.println(Bicycle.count == bike.count);
System.out.println(bike.count == anotherBike.count);
```

STATIC FIELDS CONSTANTS

- Java constants are created by marking variables `static` and `final`
- They should be named using uppercase letters with underscore characters as separators

```
static final double PI = 3.141592653589793;
```

- The variable is marked `final` means that - once initialized - it can never change
- The variable is marked `static` so that it doesn't need an instance of class

STATIC METHODS

- It is a method which belongs to the class and not to the object (instance)
- A static method can access only static data. It can not access non-static data (instance variables)
- A static method can call only other static methods and can not call a non-static method from it.
- A static method can be accessed directly by the class name and doesn’t need any object

STATIC METHODS - DECLARATION

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private static int count = 0;

    public Bicycle(int cadence, int gear, int speed) {
        // fields assignment omitted for brevity
        count++;
    }

    public static int getCount() {
        return count;
    }
}
```

STATIC METHODS - ACCESS

```
Bicycle bike = new Bicycle(75, 2, 20);
System.out.println(Bicycle.getCount());   // should print 1

Bicycle anotherBike = new Bicycle(80, 4, 25);
System.out.println(Bicycle.getCount());   // should print 2

// should prints true in both cases
System.out.println(Bicycle.getCount() == bike.getCount());
System.out.println(bike.getCount() == anotherBike.getCount());
```

SPECIAL METHOD

Used to start a Java application

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

or

```
public static void main(String... args) {
}
```

COMMAND-LINE PARAMETERS

It is possible to pass some information into a program when run it. This is accomplished by passing command-line arguments to `main()` method.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to `main()` method.

COMMAND-LINE PARAMETERS

The following program displays all of the command-line arguments

```
public class CommandLine {

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

COMMAND-LINE PARAMETERS

When we execute this program as below

```
$java CommandLine 1 2 Tom Java
```

This will produce the following output

```
args[0]: 1
args[1]: 2
args[2]: Tom
args[3]: Java
```

STATIC IMPORTS

- The static import declaration is analogous to the normal import declaration
- Where the normal import declaration imports classes from packages, allowing them to be used without package qualification, the `static import` declaration imports static members from classes, allowing them to be used without class qualification

STATIC IMPORTS

- So when should you use static import? **Very sparingly**!
- use it when you require frequent access to static members from one or two classes.

STATIC IMPORTS - EXAMPLE

```
import static java.lang.System.*;
import static java.lang.Math.PI;

public class Application {

    public static void main(String args[]) {
        out.println("Hello");    // Now no need of System.out
        out.println("PI: " + PI);// No need Math.PI
    }
}
```

STATIC FIELDS, METHODS AND IMPORTS

WHAT HAVE WE LEARNED ABOUT STATIC FIELDS, METHODS AND IMPORTS

- The `static` indicates that the particular member belongs to a type itself
- A static variable can be accessed directly by the class name and doesn’t need any object
- Java constants are created by marking variables `static` and `final`
- `static` method belongs to the class, not to the object (instance)

- Command-line parameters
  - `public static void main(String[] args)`
- the `static import` declaration imports static members from classes

VARARGS

ARGUMENTS VS PARAMETERS

- **arguments** - things you specify between the parentheses when you're invoking a method

```
go("abc", 12);  // "abc" and 12 are arguments
```

- **parameters** - things in the method's signature that indicate what the method must receive when it's invoked

```
void go(String s, int n) {}  // s and n are parameters
```

VARARGS

- Varargs allows the method to accept zero or muliple arguments
- There can be only one variable argument in a method
- Variable argument (varargs) must be the last argument

VARARGS

- We use three dots (...) in the method signature to make it accept variable arguments
- We don't have to provide overloaded methods so less code
- In fact varargs parameter behaves like an array of the specified type

VARARGS - EXAMPLE

```
int sum(int... elements) {
    int result = 0;
    for (int i : e) {
        result += i;
    }
    return result;
}

System.out.println(sum(1, 2, 3, 4);  // 10
System.out.println(sum(1);           // 1
System.out.println(sum();            // 0
```

VARARGS - EXERCISES

VARARGS

WHAT HAVE WE LEARNED ABOUT VARARGS

- Methods can declare a parameter that accepts from zero to many arguments
- A var-arg parameter is declared with the syntax `type... name`
- A var-arg method can have only one var-arg parameter
- In methods with normal parameters and a var-arg, the var-arg must come last

DATE, TIME

DATE, TIME

- There are two basic ways to represent time
  - represents time in human terms/human time, such as year, month, day, hour, minute and second
  - machine time, measures time continuously along a timeline from an origin, called the epoch, in nanosecond resolution
- Some classes in the Date-Time API are intended to represent machine time, and others are more suited to representing human time

DATE, TIME - LEGACY WAY

- `java.util.Date` - represents a specific instance in time, with millisecond precision

DATE, TIME - LEGACY WAY

- `java.util.Calendar` - is an abstract class that provides methods for converting between a specific instant in time and for manipulating the calendar fields. An instant in time can be represented by a millisecond value that is an offset from the Epoch, January 1, 1970 00:00:00.000 GMT (Gregorian)

DATE, TIME - LEGACY WAY

- `java.util.TimeZone` - represents a time zone offset, and also figures out daylight savings

DATE, TIME - LEGACY WAY - EXAMPLE

```
Date now = new Date();
// or
long millis = System.currentTimeMillis();
Date now = new Date(millis);
System.out.println(now);  // Sun Dec 31 21:31:49 CET 2017

Calendar cal = Calendar.getInstance();
Date date = cal.getTime();  // convert Calendar to Date
System.out.println(date);   // Sun Dec 31 21:31:49 CET 2017

cal.setTime(now);  // convert Date to Calendar
System.out.println(cal.get(Calendar.YEAR));          // 2017
System.out.println(cal.get(Calendar.DAY_OF_YEAR));   // 365
System.out.println(cal.get(Calendar.WEEK_OF_YEAR));  // 52
```

JAVA.TIME.LOCALDATE

Represents a date in ISO format (yyyy-MM-dd) without time

```
LocalDate localDate = LocalDate.now();
LocalDate.of(2017, 12, 31);
LocalDate.parse("2017-12-31");
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDate previousMonthSameDay = LocalDate.now()
    .minus(1, ChronoUnit.MONTHS);
DayOfWeek sunday = LocalDate.parse("2017-12-31")
    .getDayOfWeek();
boolean leapYear = LocalDate.now().isLeapYear();
boolean isAfter = LocalDate.parse("2016-06-12")
    .isAfter(LocalDate.parse("2014-06-21"));
```

JAVA.TIME.LOCALTIME

Represents time without a date

```
LocalTime now = LocalTime.now();
LocalTime sixThirty = LocalTime.parse("06:30");
LocalTime sixThirty = LocalTime.of(6, 30);
LocalTime sevenThirty = LocalTime.parse("06:30")
    .plus(1, ChronoUnit.HOURS);
int six = LocalTime.parse("06:30").getHour();
boolean isbefore = LocalTime.parse("06:30")
    .isBefore(LocalTime.parse("07:30"));
```

JAVA.TIME.LOCALDATETIME

Represent a combination of date and time

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);
LocalDateTime.parse("2015-02-20T06:30:00");
localDateTime.plusDays(1);
localDateTime.minusHours(2);
localDateTime.getMonth();
```

JAVA.TIME.ZONEDDATETIME

The ZoneId is an identifier used to represent different zones. There are about 40 different time zones

```
ZoneId zoneId = ZoneId.of("Europe/Paris");
Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
ZonedDateTime zonedDateTime =
    ZonedDateTime.of(localDateTime, zoneId);
ZonedDateTime.parse("2015-05-03T10:15:30+01:00[Europe/Paris]")
```

JAVA.TIME.PERIOD

Represents a quantity of time in terms of years, months and days

```
LocalDate initialDate = LocalDate.parse("2007-05-10");
LocalDate finalDate = initialDate.plus(Period.ofDays(5));
int five = Period.between(finalDate, initialDate).getDays();
int five = ChronoUnit.DAYS.between(initialDate , initialDate);
```

JAVA.TIME.DURATION

Represents a quantity of time in terms of seconds and nano seconds

```
LocalTime initialTime = LocalTime.of(6, 30, 0);
LocalTime finalTime = initialTime.plus(Duration.ofSeconds(30))
int thirty = Duration
    .between(finalTime, initialTime).getSeconds();
int thirty = ChronoUnit.SECONDS
    .between(finalTime, initialTime);
```

JAVA.TIME.INSTANT

Is used to work with machine readable time format, it stores date time in unix timestamp

```
Instant now = Instant.now();
Instant fromUnixTimestamp = Instant.ofEpochSecond(1262347200);
Instant fromEpochMilli = Instant.ofEpochMilli(12623472000001);
Instant fromIso8601 = Instant.parse("2010-01-01T12:00:00Z");
long toUnixTimestamp = now.getEpochSecond();
long toEpochMillis = now.toEpochMilli();
Instant nowPlusTenSeconds = now.plusSeconds(10);
```

DATE, TIME - SUMMARIZE

Class	Year	Month	Day	Hour	Minute	Second	Zone Offset	Zone ID
Instant						X		
LocalDate	X	X	X					
LocalDateTime	X	X	X	X	X	X		
ZonedDateTime	X	X	X	X	X	X	X	X
LocalTime				X	X	X		
MonthDay		X	X					
Year	X							
YearMonth	X	X						
Month		X						
OffsetDateTime	X	X	X	X	X	X	X	
OffsetTime				X	X	X	X	
Duration						X		
Period	X	X	X					

DATE, TIME - EXERCISES

- Date, Calendar, TimeZone
- LocalDate
- LocalTime
- LocalDateTime
- ZonedDateTime
- Period, Duration
- Instant

DATE, TIME

WHAT HAVE WE LEARNED ABOUT DATE, TIME

- Date, Time - legacy
  - `java.util.Date`, `java.util.Calendar`
- `java.time` package
  - `LocalDate`, `LocalTime`, `LocalDateTime`
  - `Period`, `Duration`, `Instant`
  - `Year`, `YearMonth`, `Month`

REGULAR EXPRESSIONS

REGULAR EXPRESSIONS

- A regular expression defines a search pattern for strings
- The search pattern can be anything from a simple character, a fixed string or a complex expression containing special characters describing the pattern
- The pattern defined by the regex may match one or several times or not at all for a given string
- Regular expressions can be used to search, edit and manipulate text

Source: https://docs.oracle.com/javase/tutorial/essential/regex/index.html

JAVA REGEX API

Java Regex API provides an interface and three classes in `java.util.regex` package

- `MatchResult` interface
- `Matcher` class
- `Pattern` class
- `PatternSyntaxException` class

MATCHER

It implements MatchResult interface. Is the engine that interprets the pattern and performs match operations against an input string

boolean matches()	test whether the regular expression matches the pattern
boolean find()	finds the next expression that matches the pattern
boolean find(int start)	finds the next expression that matches the pattern from the given start number
String group()	returns the matched subsequence.
int start()	returns the starting index of the matched subsequence
int end()	returns the ending index of the matched subsequence
int groupCount()	returns the total number of the matched subsequence

PATTERN

Is a compiled representation of a regular expression

static Pattern compile(String regex)	compiles the given regex and return the instance of pattern
Matcher matcher(CharSequence input)	creates a matcher that matches the given input with pattern
static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern
String[] split(CharSequence input)	splits the given input string around matches of given pattern
String pattern()	returns the regex pattern

PATTERN AND MATCHER - EXAMPLE

```
System.out.println(Pattern.matches(".s", "as"));     // true
System.out.println(Pattern.matches(".t", "dt"));     // false
System.out.println(Pattern.matches(".d", "odt"));    // false
System.out.println(Pattern.matches(".d", "oodt"));   // false
System.out.println(Pattern.matches("..t", "odt"));   // true
```

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
System.out.println(m.matches());  // true
```

REGULAR EXPRESSIONS

characters

x	The character x
\\	The backslash character
\t	The tab character
\n	The newline (line feed) character
\r	The carriage-return character
\f	The form-feed character

REGULAR EXPRESSIONS

character classes

[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

REGULAR EXPRESSIONS

predefined character classes

.	Any character
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

REGULAR EXPRESSIONS

boundary matches

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

REGULAR EXPRESSIONS

quantifiers

greedy	reluctant	possessive	meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

REGULAR EXPRESSIONS - EXERCISES

REGULAR EXPRESSIONS

WHAT HAVE WE LEARNED ABOUT REGULAR EXPRESSIONS

- Regex is short for regular expressions, which are the patterns used to search for data within large data sources
- The `Pattern` and `Matcher` classes have Java's most powerful regex capabilities