

Java 101: Core language

Agenda

- Java language overview
- Language basics
- Object oriented programming
- Annotations
- Javadoc

Java language overview

The Java programming language

- Object oriented
- Promises “write once, run everywhere”
- One of the most widely adopted languages at present
- Has a community of > 10 million developers

Java vs C/C++

- Syntax is similar to the one of C/C++
- Provides automatic memory management
- Lacks operator overloading and multiple inheritance
- Everything resides in classes
- Other differences related to language constructs, OOP and SDKs

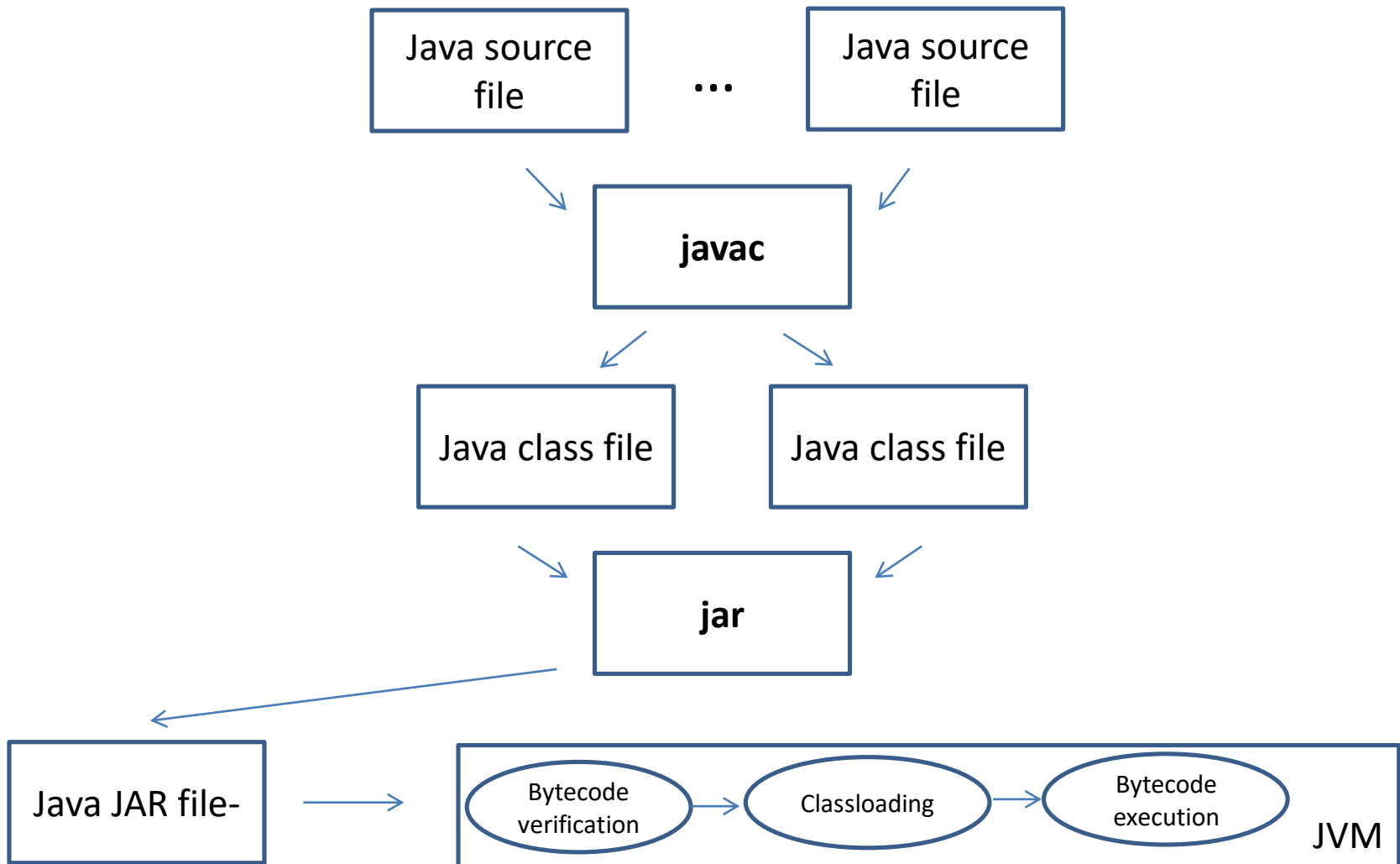
A bit of a history ...

Release	When
JDK 1.0	January, 1996
JDK 1.1	February, 1996
J2SE 1.2	December, 1998
J2SE 1.3	May, 2000
J2SE 1.4	February, 2002
J2SE 5.0	September, 2004
Java SE 6	December, 2006
Java SE 7	July, 2011
Java SE 8	March, 2014
Java SE 9	September, 2017
Java SE 10	March, 2018
Java SE 11	September, 2018
Java SE 12	March, 2019
Java SE 13	September 2019
Java SE 14	March, 2020

Vendors

- Not only Oracle builds OpenJDK releases ...
- Many others do:
 - Red Hat (build of OpenJDK)
 - SAP (SapMachine)
 - Azul (Zulu)
 - Amazon (Corretto)
 - Alibaba (Dragonwell)
 - Java community (AdoptOpenJDK)

JVM execution model



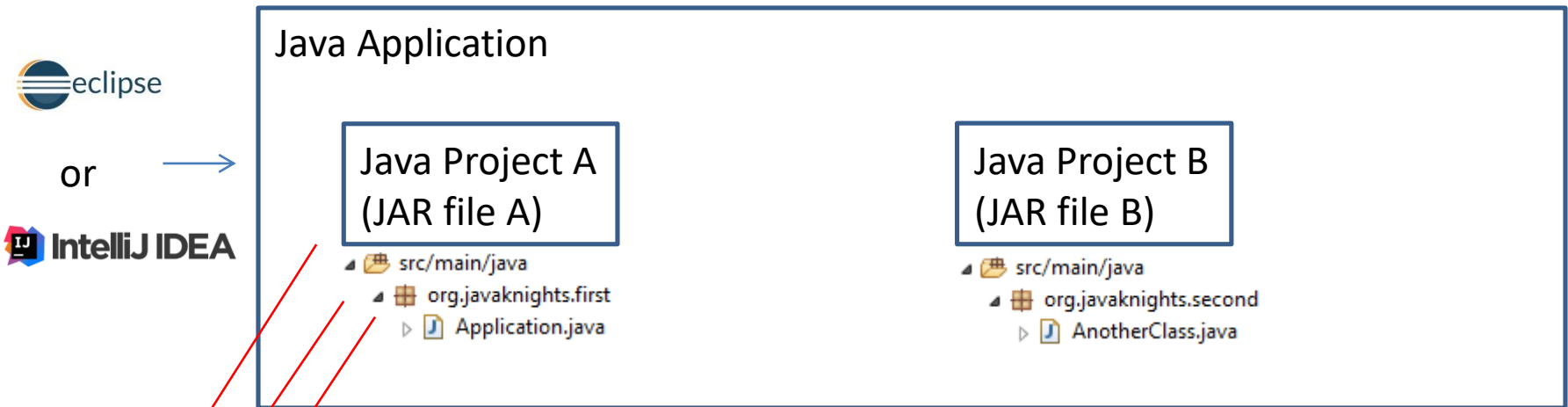
Language basics

Java applications

- The JVM loads Java class files or JAR files (as a collections of class files)
- A Java project is a concept implied by Java IDEs (integrated development environments)
- The IDE or another build tool is used to generate typically a JAR file out of that project

Java applications

- A Java application is a combination of one more Java projects



Each package might have one or more Java source files

Each source folder might have one or more packages

Each project might have one more source directories

Application entrypoint

- The entrypoint to a Java application is the **main** method

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

- The args array is a list of the command line arguments passed to the Java application
- It is the starting point of the application's execution by the JVM

```
java -cp app.jar java_class_with_main_method
```

Application logic

- A Java application is a collection of classes
- Each class has fields and methods
- The methods of the class supply its logic
- Let's investigate the syntax of the Java language in terms of the logic that can be implemented in the methods ...

Data types

- The Java language has 8 primitive data types:
 - byte – 8 bit
 - short – 16 bit
 - int – 32 bit
 - long – 64 bit
 - float – single precision 32-bit floating point
 - double – double precision 64-bit floating point
 - boolean – **true** or **false**
 - char – single 16-bit Unicode character

Literals

- For each primitive data types literal values can be provided in source code
- Literal values provide fixed values for the particular data type

```
byte a = 7;  
short b = 100;  
int c = 5000;  
long d = 12000;  
float e = 13.4f;  
double f = 13.4f;  
boolean g = true;  
char h = 'x';
```

Wrapper types

- In some cases (such as Java generics) primitive types cannot be used directly
- Objects are required instead
- For that reason for each of the primitive types there is an equivalent object type
- The Java compiler provides automatic conversion between primitive types and their object equivalent

Wrapper types

boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Type casting

- You can also cast between the primitive objects:
 - widening (converting from a smaller to a large data type)
byte -> short -> char -> int -> long -> float -> double
 - narrowing (converting a larger data type into a smaller one)
double -> float -> long -> int -> char -> short -> byte

```
float x = 10.2f;  
double y = x;
```

Strings

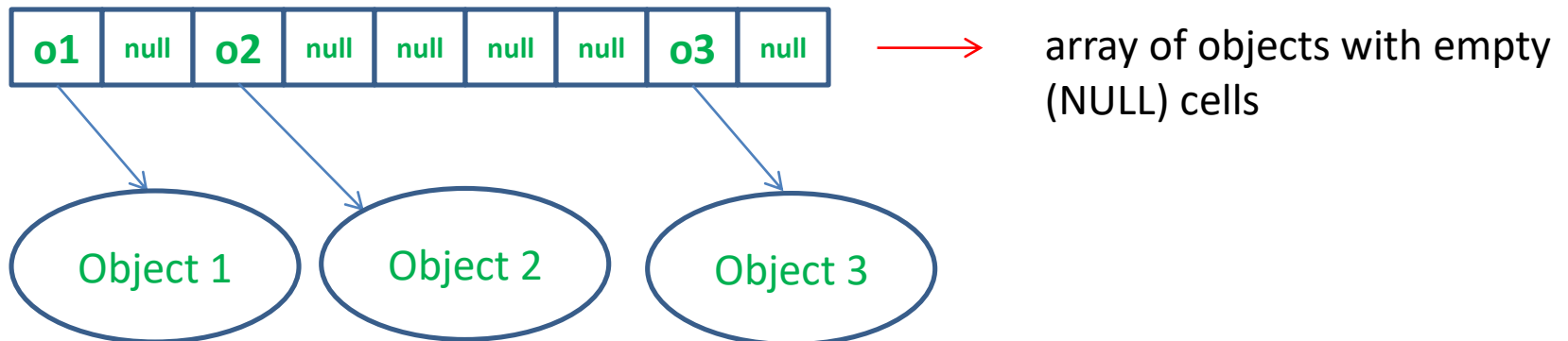
- Strings are special objects in Java that represent a sequence of characters
- Strings are stored in a dedicated area in the JVM heap called the **string pool**

```
String first = "text";  
String second = "text";  
String third = new String("text");  
  
System.out.println(first.equals(second)); // true  
System.out.println(third.equals(second)); // false
```

Arrays

- Arrays are sequence of elements stored contiguously in the JVM heap

```
int[] a = new int[10];  
Object[] b = new Object[20];  
double [][] c = new double[20][30];  
String [] d = {"one", "two", "three"};
```



Variables

- There are two types of variables:
 - primitive – can have one of the 8 data types we defined earlier. Primitive variables inside a method are allocated on the **stack**
 - reference – point to an object. Objects are allocated on the **heap**

Variables

- Variables can be specified as:
 - local variables: store the temporary state of a method. Primitive local variables are allocated on the **stack**
 - parameters: convey data to a method in the form of parameters
 - instance variables: fields of a class (unique for each object of a class)
 - static variables: static fields of a class (shared across all objects of a class)

The heap and stack

- The heap is the area of the JVM memory that is used for object allocation and its internal data structures
- The Xms and Xmx flags on JVM startup specify initial and maximum heap memory allocation

```
java -Xms256m -Xmx2048m
```

The heap and stack

- The stack (also called **thread stack**) is an area of the JVM memory
- The stack is used to store local variables, partial results, and data for method invocations and returns
- The stack consists of stack frames that are created on method invocation

Legal identifiers

- Not only variable names but also names of classes, methods and packages are subject to constraints
- There are certain reserved words in the Java language (such as **abstract** or **while**) that cannot be used as identifier names
- Full list:
<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/keywords.html>

Comments

- You can specify source comments in two favors:

- Single line

```
// this is a single line comment
```

- Multiline

```
/*  
This is a  
multiline comment  
*/
```

- A special type of comments are called Javadoc that are used to generate source code documentation

Operators

- Operators in Java are used to perform operations on a number of operands
- Depending on the operator there might be a certain number of operands that can be specified
- Some operators have higher precedence than others
- Unlike C++ there is no operator overloading built in Java

Operators

Operators	Usage (precedence from top to bottom)
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

Expressions, Statements and Blocks

- Expressions are a combination of operators applied on certain operands that lead to a single value

```
x + y + 5
```

- Statements are the execution units of a Java program, delimited by ; such as method invocation or variable assignment

```
int z = x + y + 5;  
validateCount();  
Table t = new Table();
```

Expressions, Statements and Blocks

- Blocks are a collection of one more statements
- Blocks are related to the concept of **variable scope**: that is the places where a particular variable can be accessed
- Block scope is created by putting statements in brackets
{...}

```
{  
    int z = x + y + 5;  
    validateCount();  
}
```

- The method body is an example of a block scope

Control flow

- There are certain statements that can be used to determine the control flow in a Java application:
 - If/else statements to provide conditional logic;
 - switch statements to provide conditional logic based on a certain value;
 - loops to execute a block multiple times based on a condition;
 - branching statements such as **break**, **continue** and **return**.

If..else

- “if a certain condition is met execute a particular block otherwise execute the **else** block”

```
if(x > 10) {  
    y = 10;  
} else {  
    y = x;  
}
```

- The expression of the **if** clause must be a boolean expression

If any of the block has a single statement the brackets can be omitted but that is a bad practice as it can introduce subtle bugs !

switch

- “if a certain value is provided then execute a particular block of code”

```
switch(x) {  
    case 10:  
        // some logic  
        break;  
    case 20:  
        // some logic  
        break;  
    default:  
        // some logic  
}
```

- A switch statement works with the **byte, short, char, int, enum** and **String** (as of JDK 7) types

switch

As of JDK 13 you can also use switch statement as expressions (that return a particular value)

```
int y = switch(x) {  
    case 10 -> 100;  
    case 20 -> 120;  
    default -> 1000;  
}
```

while loop

- “while a particular condition is true execute a block of statements”

```
while(x > 10) {  
    y = y + 1;  
}
```

do..while loop

- “execute a block of statements while a particular condition is true”

```
do {  
    y = y + 1;  
} while(x > 10);
```

for loop

- “execute a block of statements a particular number of times based on initial value, condition and step”

```
for(int i = 1; i <= 100; i++) {  
    x = x + i/2;  
}
```

- Any of the three **for**-loop statements can be omitted

```
for(;;) {  
    if(x > 50) {  
        break;  
    } else {  
        x++;  
    }  
}
```

break/continue

- **break** can be used to break from a loop
- **continue** can be used to continue with next loop iteration (skipping logic after **continue**)

```
for(int i = 1; i <= 100; i++) {  
    x = x + i/2;  
    if(x > 1000) {  
        break;  
    }  
    if(i % 17 == 0) {  
        continue;  
    }  
}
```

break/continue

- **break** and **continue** also have a labeled form that can be applied in case of nested loops

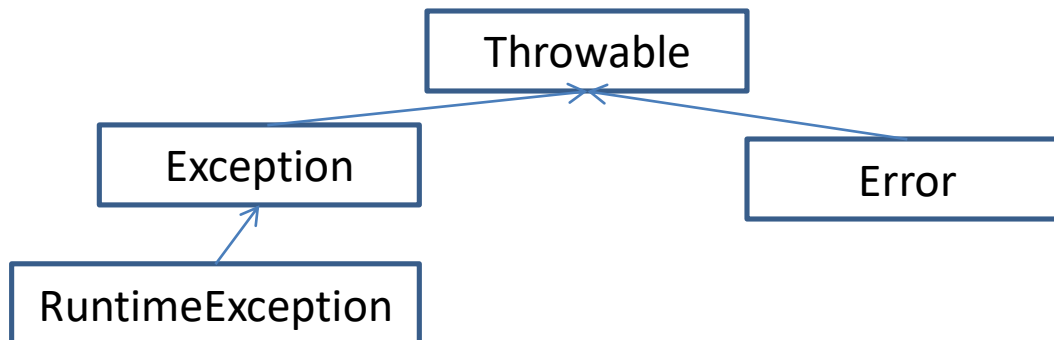
```
outer:
    for (int i = 1; i <= 100; i++) {
        for (int j = 1; j <= 100; j++) {
            x = x + i / 2;
            if (x > 1000) {
                break outer;
            }
            if (i % 17 == 0) {
                continue outer;
            }
        }
    }
```

Exceptions

- Exceptions are used to handle program errors and other exception events in a running application
- All exceptions extend the **java.lang.Exception** class
- Critical errors can also be represented by a **java.lang.Error** class
- Both of them extend the **java.lang.Throwable** class

Exceptions

- Exceptions can be either:
 - **checked**: must be either caught or declared in the calling method
 - **unchecked**: may not be declared or caught; extend the `java.lang.RuntimeException` class



Throwing an exception

- An exception can be thrown using the **throws** keyword

```
throw new NullPointerException();
```

```
throw new MyCustomException();
```

- If a method throws a checked exception or some of the methods it calls do and the method cannot handle it then it must be declared

```
public void move(Location location)
    throws MyCustomException {
    // logic here
}
```

Throwing an exception

- A method can handle an exception in a try-catch-finally block:
 - The **try** block calls methods or some logic that can throw an exception
 - The **catch** block provides logic to be executed in case the exception is caught
 - The **finally** block performs logic to be executed in case an exception is thrown

Handling an exception

```
public class MyCustomException extends Exception {  
}
```

```
try {  
    throw new MyCustomException();  
} catch (MyCustomException e) {  
    System.out.println("Exception caught: " +  
        e.getMessage());  
    e.printStackTrace();  
} finally {  
    // close resources, i.e. DB connections  
}
```

Object oriented programming

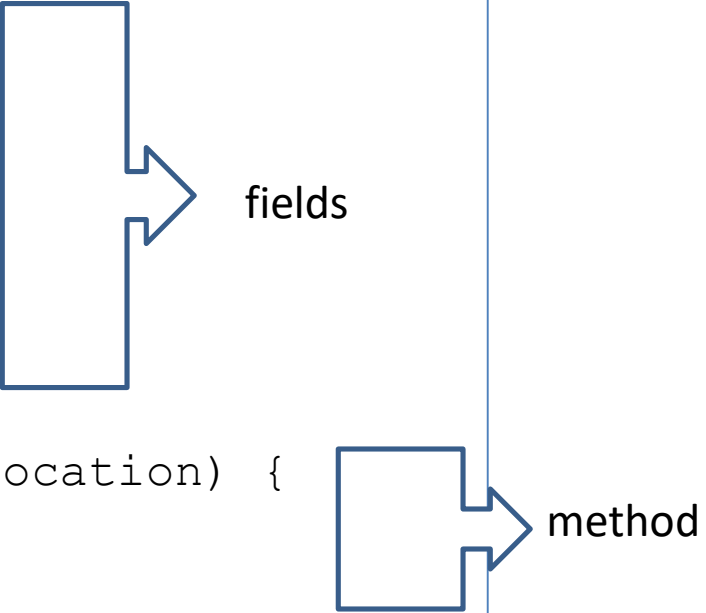
Declarations

- The fundamental element of an object-oriented programming language like Java is the **class**
- A class is a collection of fields and methods
- The fields provide different state for the instances (objects) of the class
- The methods provide the logic of the class

Declarations

```
public class Table
{
    private String color;
    private int numberOfLegs;
    private double weight;
    private Location location;

    public void move(Location location) {
        ...
    }
}
```



fields

method

Each public class in Java must reside in its own Java file
(the above must be in **Table.java**) !

Object creation

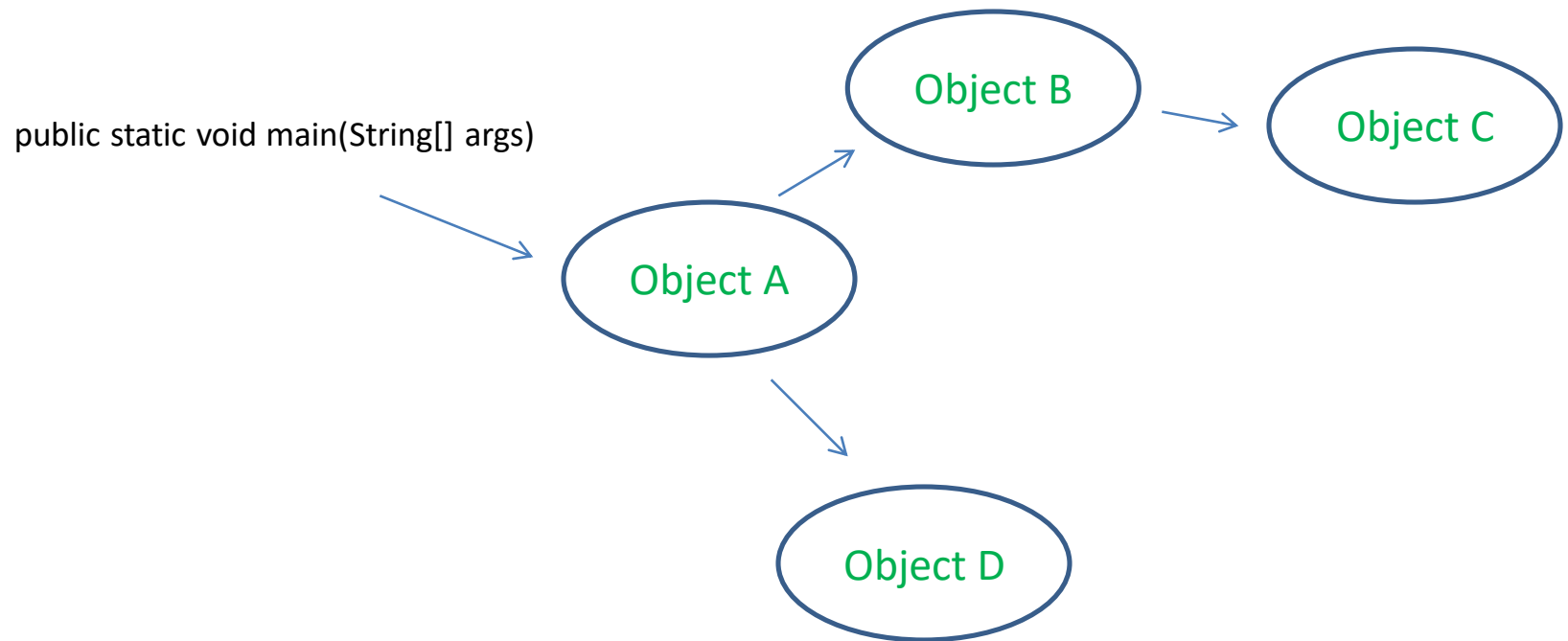
- Each class may have multiple instances also called **objects**
- An object of a class is created using the **new** operator

```
Table green = new Table();
```

- You can also pass parameters for the object at creation time by defining special types of methods in the class called **constructors**

Object creation

- At runtime the application memory in the JVM heap forms an object graph ...



Class methods

```
public class Table
{
    public void move(Location location) {
        ...
    }
}
```

access modifier

return type

method name

method parameters

The entire method declaration is also called **method signature** !

Access modifiers

- There are four access levels (via 3 access modifiers and 1 default) that can be used for classes, methods or fields:
 - **public**: can be used anywhere
 - **protected**: can be used in current package or child classes
 - **private**: can be used only in current class
 - package-level (default, in case no modifier is specified): can be accessed in classes of the same package

Method return type

- The method return type specifies the type of instance returned by the method as a result
- In case the method does not return any result then the **void** keyword can be specified

Method parameters

- Method parameters are specified separate by comma
- Unlike other languages (such as Python) Java does not support default values for method parameters
- There is a special type of variable argument (vararg) parameter that can be specified as the last parameter of a method and has ... (three dots) after the type

```
public static int sum(int... values) {  
    ...  
}
```

```
sum(1);  
sum(7, 15);  
sum(4, 12, 18);
```

Constructors

```
public class Table
{
    public Table(String color) {
        ...
    }
}
```

- The constructor is a special method that:
 - does not have a return type
 - has the same name as the class
 - is used to specify additional parameters during object creation

```
Table green = new Table("green");
```

Init blocks

- Initialization logic can be specified not only in the constructor but in initialization blocks as well:

```
public class Table
{
    {
        // logic here
    }
}
```

Init blocks

- A static init block is used to specify initialization logic once the class is loaded by the JVM:

```
public class Table
{
    static {
        // logic here
    }
}
```


Method overloading

- The same method can be defined multiple times in a class with different parameters
- This is called method overloading

```
public class Table
{
    public void move(int distance) {
    }

    public void move(long distance) {
    }
}
```

Class fields

- Instance fields of a class have the following default values:

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Inheritance

- In most OOP languages classes can form a hierarchy
- The hierarchy establishing by creating classes that inherit from other (parent) classes
- In some languages (like C++) one class can inherit from more than one parent class but that is not possible in Java

Inheritance

- Child classes inherit members (fields, methods and nested classes) of the parent class
- All classes in Java extend from the `java.lang.Object` class
- ...

Inheritance

```
public class Table
{
    public void move(Location location) {
        ...
    }
}
```

```
public class RoundTable extends Table
{
}
```

Method overriding

- A child class can **override** (redefine) a method of the parent class
- If the method wants to call the one from the parent class it can use the **super** reference

```
public class RoundTable extends Table {  
  
    public void move(Location location) {  
        super.move(location);  
        // logic here  
    };  
  
}
```

Abstract classes

- If the class wants to force child classes to implement certain methods we can mark them as **abstract** and mark the class **abstract** as well
- Abstract classes cannot be instantiated directly

```
public abstract class Chair {  
    public abstract void move(Location location);  
}
```

Interfaces

- An interface is a reference type like a class
- Can contain method signatures or constants
- An interface can be implemented by classes
- A class can implement more than one interface

As of JDK 8 interfaces can have default and static methods with concrete implementation. They provides extensibility of the interface without breaking existing implementation.

A particular type of interfaces are the ones without any methods ! Also called **marker** interfaces – we will talk more about these when we cover design patterns

Interfaces

```
public interface MovableTable {  
  
    public int REFERENCE_ID = 1;  
  
    public void move(Location location);  
  
    public default void resetLocation() {  
        // logic here  
    }  
  
    public static int getReference() {  
        return REFERENCE_ID;  
    }  
  
}
```

```
public class Table implements MovableTable {  
    public void move(Location location) {  
        // logic here  
    }  
  
}
```

Final classes

- If you want to prevent a class from being extended you can mark it as **final**

```
public final class Table  
{  
    ...  
}
```

Enums

- Enums are special data types that allow a variable to have a predefined constant
- Enums can also have instance members, constructors and methods similar to a class

```
public enum Color {  
    RED, GREEN, BLUE  
}
```

Packages

- Classes, interfaces, enums and annotations can be logically grouped in packages
- By conventions packages in Java should follow the reference naming conventions

```
com.company.project.package
```

- Packages are typically organized either by purpose (utilities, entities, services etc.) or by domain (marketing, accounting, sales etc.)

Annotations

Annotations

- Annotations are used to provide additional metadata for certain elements of the source code
- Annotations are applied in a vast number of use cases
- Annotation can be applied over:
 - classes
 - fields
 - methods
 - other elements (such as method parameters)

As of JDK 8 annotations can also be applied on the use of types such as instance creation, type casting, implements clause and throws declaration

Annotations

- Annotations can be available either only at compile time or at runtime as well
- By default they are available at compile time
- Annotations can have attributes

As of JDK 8 annotations can be applied multiple times (also called **repeatable annotations**). That was not possible in earlier JDK versions.

Annotation declaration

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Furniture {

    String value();

    int id() default -1;
}
```


Annotation usage

```
@Furniture("TABLE")  
public class Table  
{  
    ...  
}
```

```
@Furniture(value = "TABLE", id = 5)  
public class Table  
{  
    ...  
}
```

Javadoc

Javadoc

- Javadoc is a document generator used to generate documentation from source code
- Documentation is typically generated as HTML pages (although other formats are supported)
- Distributed as part of the JDK
- Defined in a special type of comment

```
/**  
 * Javadoc here  
 */
```

Javadoc

```
/**
 * Provides a representation of a table.
 *
 * @author Martin
 */
public class Table {

    /**
     * Moves the current table to a new location
     * @param location The new location for the table
     */
    public void move(Location location) {
        // method logic
    }
}
```

Javadoc

- Javadoc can also be specified at the package level in a **package-info.java** file
- The package-info.java file can also specify package-level annotations

```
/**  
 * Javadoc here  
 */  
com.company.project.package
```

Questions ?