AN INTRODUCTION TO BACKEND FOR BEGINNERS

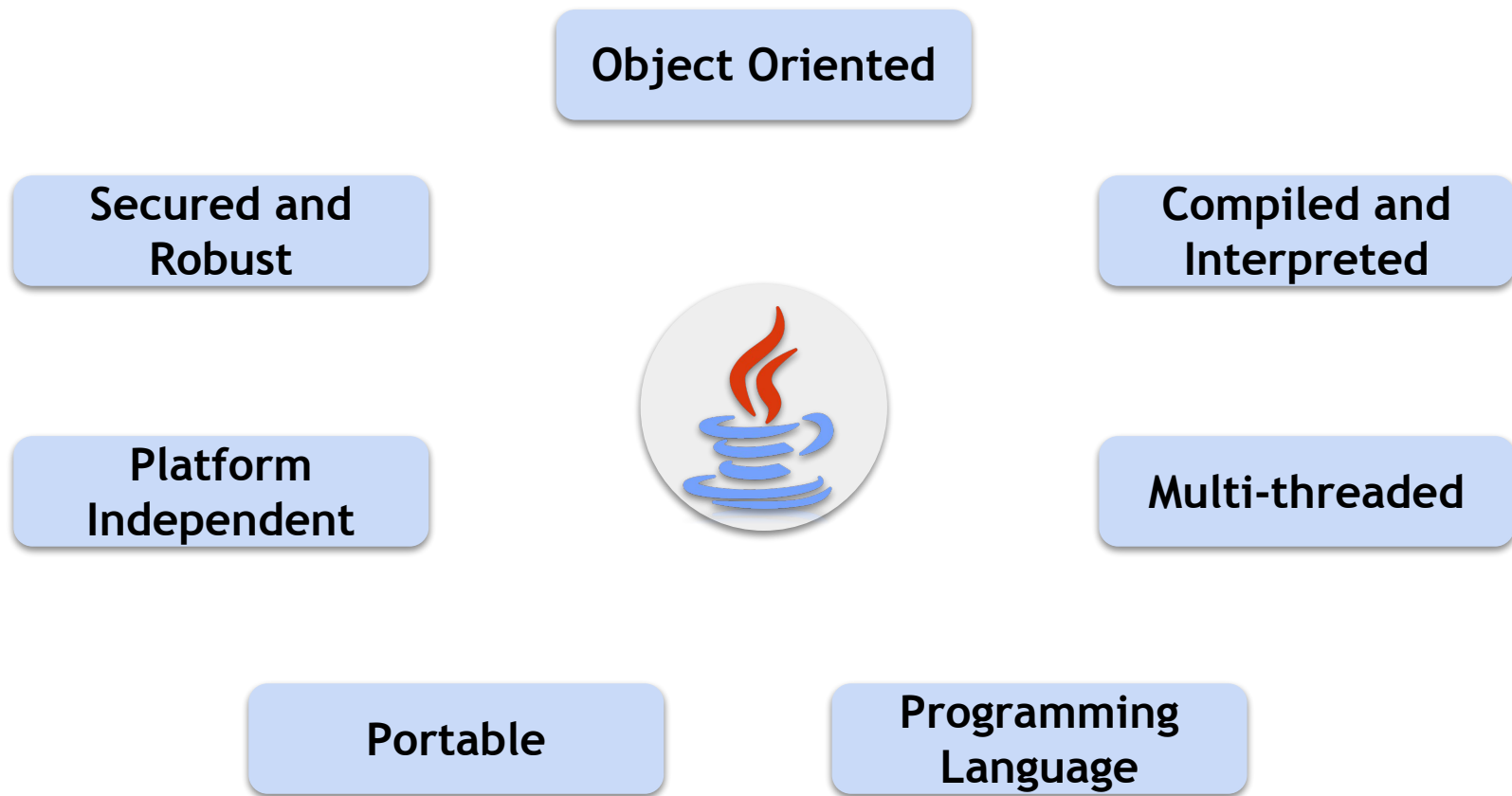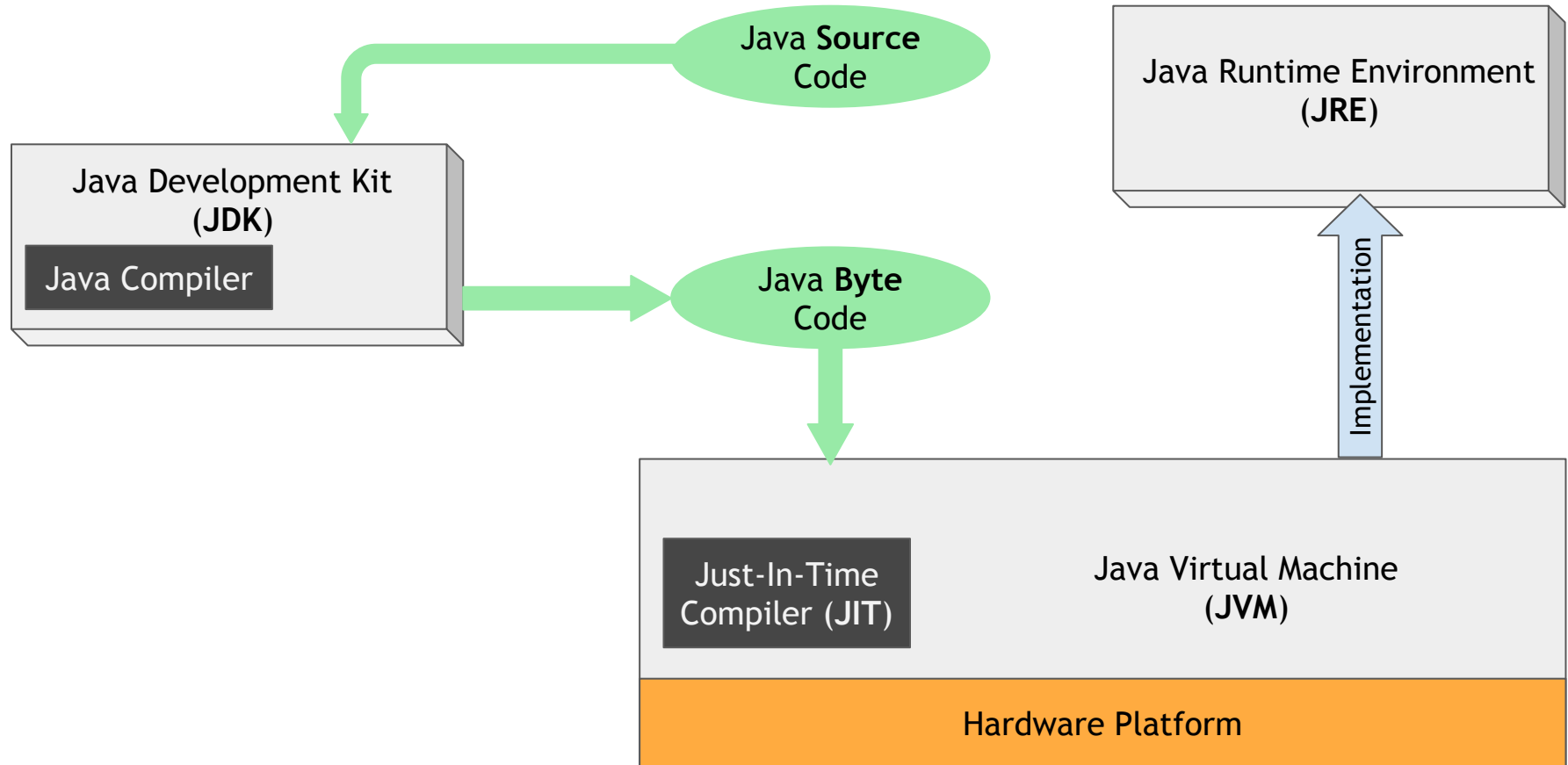## MODULO 2 – JAVA SE

Matteo Colombo Speroni
Armando Esposito

Objectives:

- Java basics
- Object Oriented Programming
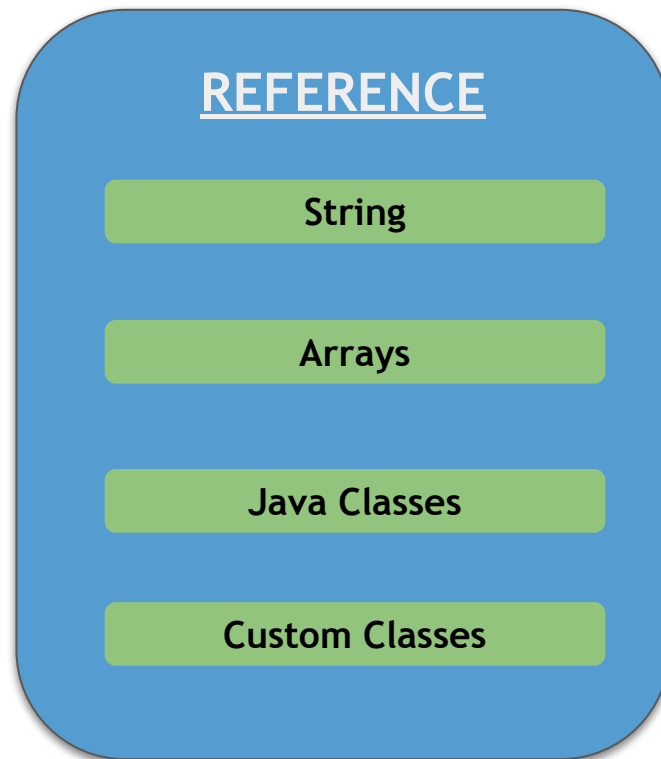- Learn how to modelling a complex object structure

What is Java?

Object Oriented

Secured and Robust

Compiled and Interpreted

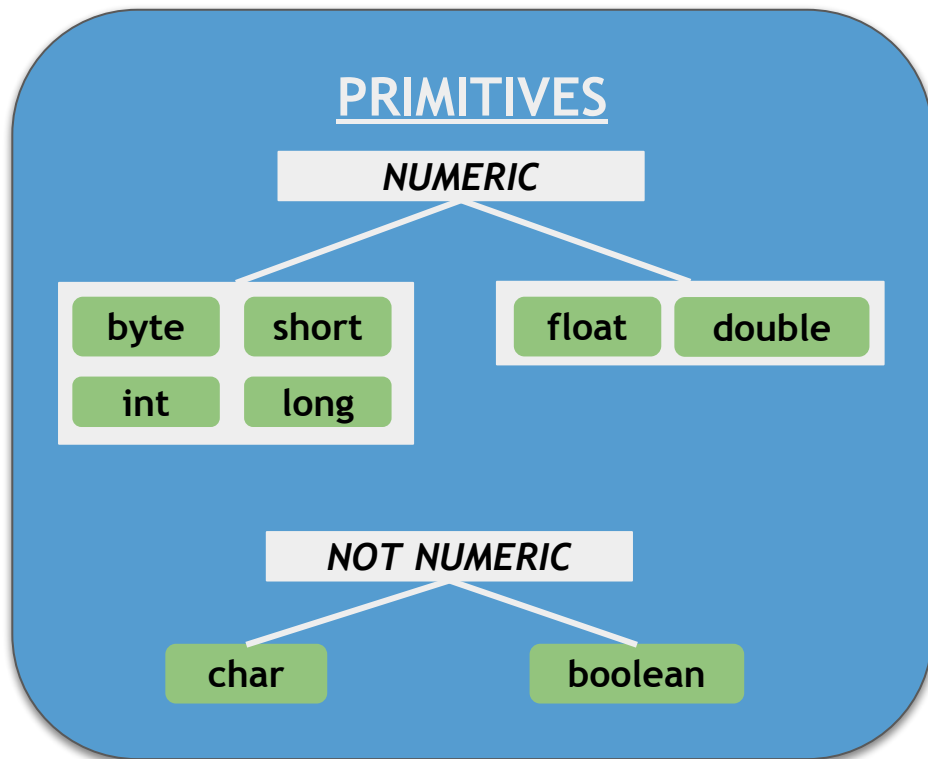Platform Independent

Multi-threaded

Portable

Programming Language

JDK

Data Types

Class and Objects

A Class is a blueprint

An object is an instance created from a blueprint

Objects belonging to same type has same attributes but different attributes values
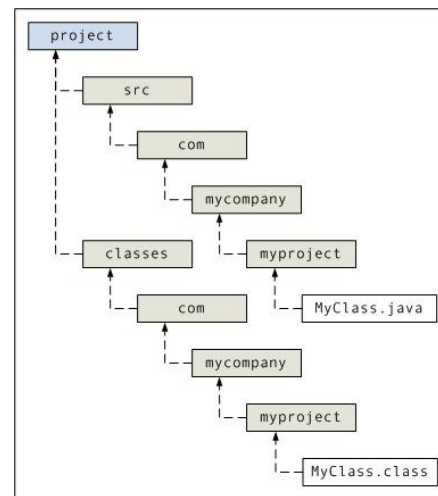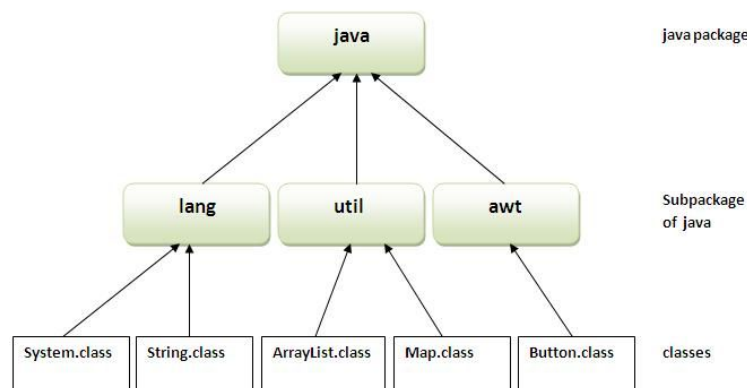
Packages

A package is a container for classes.

A package is a **grouping** of related types(classes and interfaces) providing **access protection** and **namespace** management **avoiding name collisions.**

Packages are divided into two types they are **built-in packages** and **user defined** packages

Class Structure

Access modifier   class keyword   Class Name   Imports   package

Instance variables

Arguments. Can be void.

```java
package slidecode;

import java.lang.String;

public class Cat {

    private String colour;
    private int age;

    public Cat(String colour, int age) {
        this.colour=colour;
        this.age=age;
    }

    protected String talk() {
        int local = 1;
        return "meow!";
    }

    public static String talk2() { return "zzzzz!"; }
}
```

Constructor. If not exist default applied

Instance method

Return type. Can be void

Class Method

} Local scope

Enum

```java
enum Color {

    RED(1), BLUE(2), GREEN(3);

    private int value;

    Color(int value) {

        this.value = value;

    }

    public int getValue() { return value;}

}

Color color = Color.RED;

color.name()      // RED

color.ordinal()     // 0

color.toString()    // RED

color.values()  // [RED, BLUE, GREEN]

color == COLOR.RED   //true
```

Enum in java is type safe

Enum constants are implicitly static and final

Enum can implement an interface

Can be threatened as static classes encapsulating fields.

Can be used inside switch like int and String

You should use enum types to represent fixed set of constants

Scope

| Variable	Scope	Lifetime

static	`class Example {`
	
	    `//static variable`
	
	    `static int const=100;`
		Exists for **as long as the class** it belongs to is loaded in the *JVM*.

instance	`//instance variable`
	
	    `int data=50;`
	
	    `void method(){`
		Exists for **as long as the instance** of the class it belongs to.

local	`//local variable`
	
	        `int n=90;`
	
	    `}`
	
	`}`
		Exists **until the method** it is declared in **finishes executing.**

Memory Organization

Stack: portion of memory containing local (block) variables, primitives, ...

Heap: portion of memory dynamic containing objects

Static Area: portion of memory containing global and static variables

```java
public class Student {

    String name;
    String gender;
    static String COLLEGE = "MIT";

    public Student(String name, String gender) {
        this.name=name;
        this.gender=gender;
    }

    public static void main(String[] args) {
        System.out.println(Student.COLLEGE); // MIT

        Student s1 = new Student( name: "John",  gender: "M");
        Student s2 = new Student( name: "Jane",  gender: "F");

        System.out.println(s1.name); // John
        System.out.println(s2.name); // Jane
        s1.COLLEGE = "BHO!";
        System.out.println(Student.COLLEGE); // BHO!
        System.out.println(s2.COLLEGE); // BHO!
    }
}
```
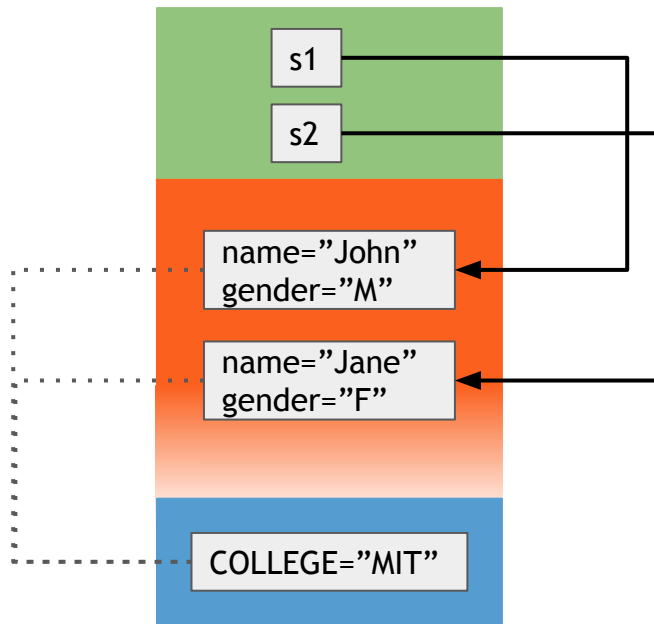
Call by Value vs by Reference

Pass by Value: The method parameter values are copied to another variable and then the copied object is passed. All primitives types are passed by value.

Pass by Reference: An alias or reference to the actual parameter is passed to the method, that's why it's called pass by reference. PAY ATTENTION: also the reference to the object passed as parameter is passed by value.

```
void method(int num) {
    num = 200;
}

int a = 100;
method(a);

System.out.println("a"); // 100
```

a	num
100	200

```
void method(Person pers) {
    pers.setName = "new Name";
}

Person p = new Person();
p.setName("Pippo");
method(p);

System.out.println(p.getName()); // new Name
```

p → Real Object ← pers

Keywords

1. Java Files (3)
```
class
interface
enum (1.5)
```

2. Data Types (8+1)
```
byte
short
int
long
float
double
char
boolean

void
```

3. Memory Location (2)
```
static
new
```

4. Control Statements (11)
    4.1 conditional
```
if
else
switch
case
default
```
    4.2 loop
```
while
do
for
```
    4.3 transfer
```
break
continue
return
```

5. Accessibility Modifiers (3)
```
private
protected
public
```

6. Modifiers (8)
```
static
final
abstract
native
transient
volatile
synchronized
strictfp
```

7. Object Representation (3)
```
this
super
instanceOf
```

8. Inheritance Relationship (2)
```
extends
implements
```

9. Package (2)
```
package
import
```

10. Exception Handling (5+1)
```
try
catch
finally
throw
throws

assert (1.4)
```

11. Unused Keywords (2)
```
const
goto
```

These are not keywords

Default Literals
1. referenced literal   null

2. boolean literals
    true
    false

Java is an **Object Oriented Programming** Language

Object
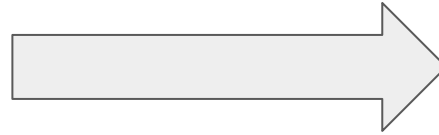
Object

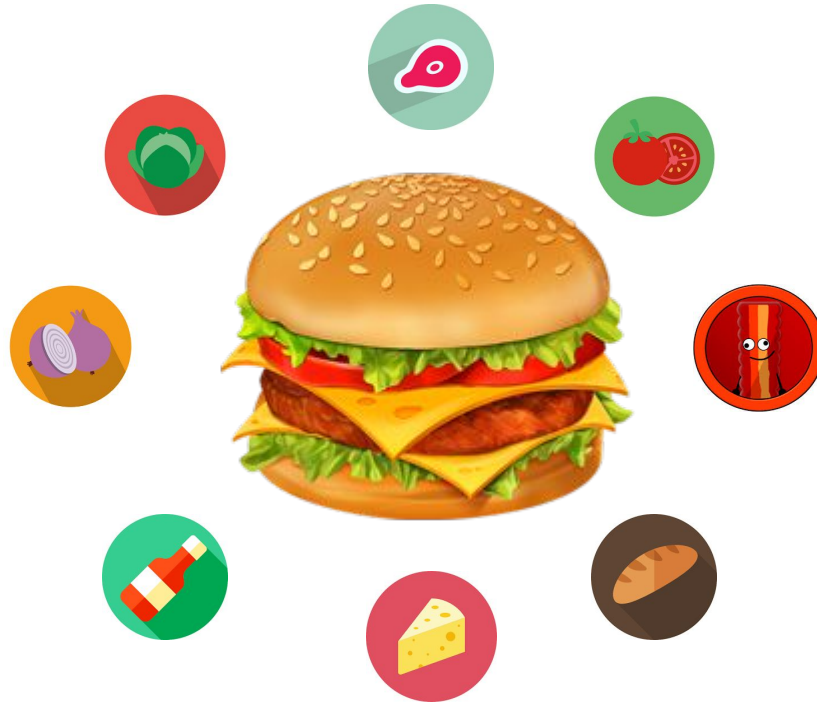Object

Object

Object

Object

Class

OOP - Principles

Encapsulation

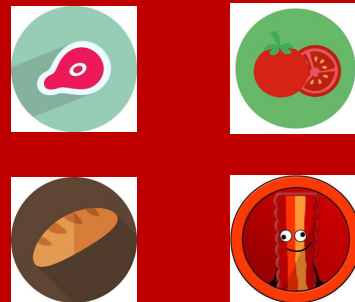Abstraction

Inheritance

Polymorphism

OOP - Encapsulation

Deny access to state - Wrapping

Other Object

+ **addSauce**
+ **removeIngredient**


```
class Hamburger {
    private List<Ingredient> ingredients = new ArrayList<>();
    ...

    public void removeIngredient(Ingredient ing){
        ingredients.add(ing);
    }

    public void addSauce(Sauce sauce){
        ingredients.add(sacue);
    }
}
```
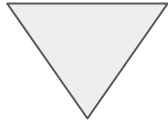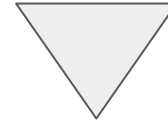
Visibility

Class	Different class but same package	Different package but subclass	Unrelated class and different package
package p1; class A {			
private int var1;	🟥	🟥	🟥
int var2;	🟩	🟥	🟥
protected int var3;	🟩	🟩	🟥
public int var4;	🟩	🟩	🟩
}			

OOP - Abstraction
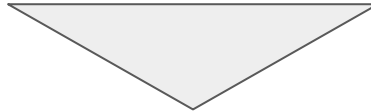
Abstract vs Interface

IS A — ABSTRACT Beverage

Eatable   **Drinkable**   CAN DO — INTERFACE

IS A — ABSTRACT Food

```java
abstract class Food {

    abstract void abstractMethod();

    void concreteMethod(){
        // code here
    }
}

interface Eatable {
    void eat();
}

class Hamburger
        extends   Food
        implements Eatable {

    @Override
    void abstractMethod() {
        //code here
    }

    @Override
    public void eat() {
        //code here
    }
}
```

Annotations

*"Annotation-based development lets us avoid **writing boilerplate** code under many circumstances by enabling tools to **generate** it **from annotations** in the source code. This leads to a **declarative programming style** where the programmer **says what should be done** and **tools emit the code** to do it."*

Oracle Definition

```
public @interface Author {
    String name();
    int year();
}
```
declaration →

```
@Author(name = "Brett Whiskers",
        year = 2008)
class MyClass {}
```
usage →

Program metadata – decorations on ordinary Java code.

Like javadoc comments, but with syntax and strong types.

Meant to be both human- and machine-readable.

OOP - Inheritance

PARENT

1°LEVEL CHILD

2°LEVEL CHILD

Object class - super - this

```java
class Hamburger {

    int meat;
    int tomato


    public Hamburger(){
        this(1,1);
    }

    public Hamburger(int meat, int tomato){
        this.meat = meat;
        this.tomato = tomato;
    }
}
class Cheeseburger extends Hamburger {
    int cheese;
    public MyCustomBurger(int cheese) {
        super(2,0);
        this.cheese = cheese;
    }
}
```
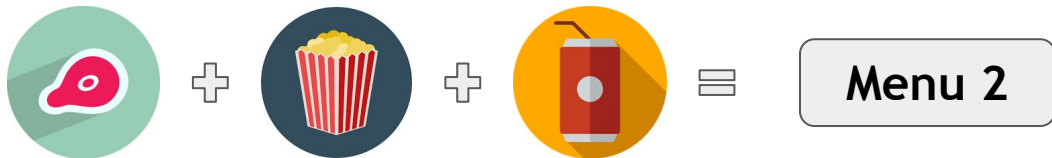
A class can inherit **only from a superclass**,
in the hierarchy tree the parent of all java classes is
OBJECT.

SUPER: **explicit reference to superclass constructor, method or field**.
In case of constructor if not explicitly called the default one or the one without parameters is called

THIS: **explicit reference to instance constructor, method or field**.

OOP - Polymorphism

Overload and Override

Overloading: same method name with different argument and perhaps return type.

Override: same method name with identical signature in a child class.

```
public void eat(){
…
}

public void eat(Food f) {

…

}

public void eat(Food f, Beverage b)
{
…
}
```

```
public class Menu {
        public void eat(){

        …
        }
}
```

```
public class MegaMenu
extends Menu{
        @Override
        public void eat(){

        …
        }
}
```

```
public class SimpleMenu
extends Menu{
        @Override
        public void eat(){

        …
        }
}
```

Final Keyword

```java
class Bike {
      final int speedlimit = 90;//final variable

      void run() {
            speedlimit =  400;
      }

      final void runFinal() { // final method
      }
}

class Ducati extends Bike {
      @Override
      void runFinal() {
      }
}

final class Ducati999 extends Ducati {
}

class DucatiNew extends Ducati999 {
}
```
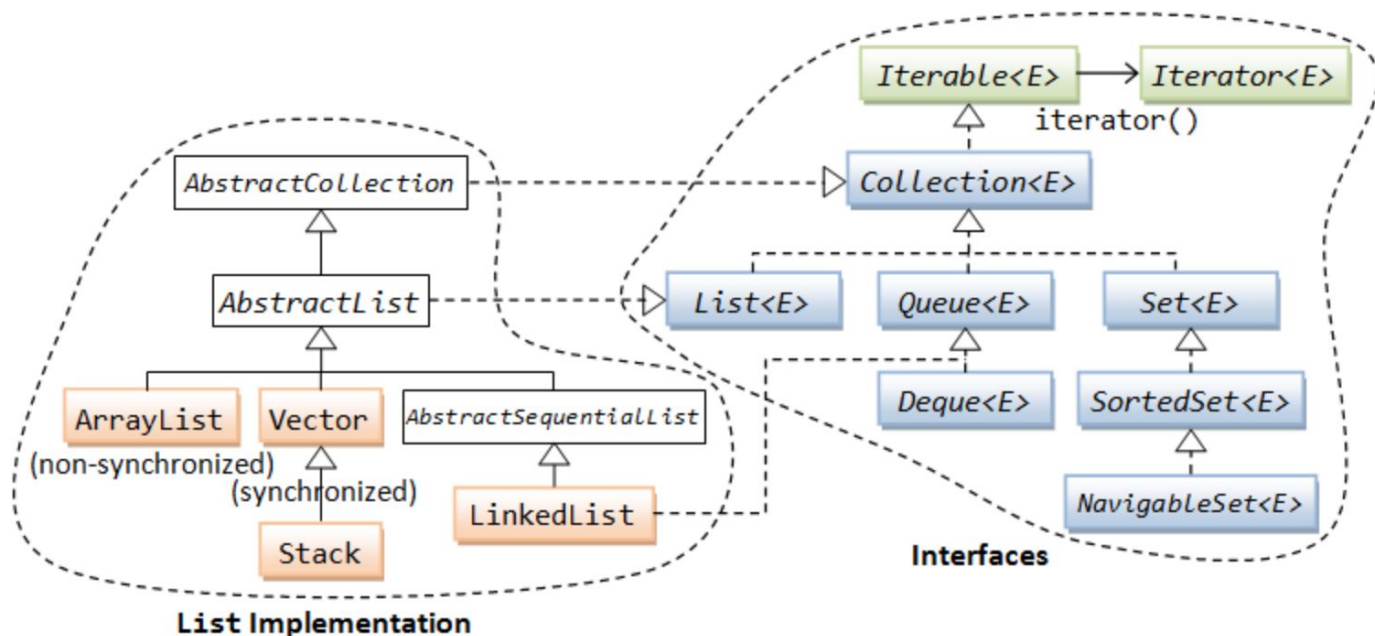
Value Change

Method Overriding

Inheritance

Collections

Java collections framework:
- a **hierarchy of <u>interface</u> types and <u>classes</u> for collecting objects**
- each interface type is implemented by one or more classes

Hash - Equals contract

Equals and Hashcode must be ovverriden together

If not overridden the java.lang.Object method are invoked

```java
public native int hashCode();
public boolean equals(Object var1) {
    return this == var1;
}
```

First Value	Second Value	==	equals()
int i = 10	int j = 10		
Integer i = new Integer(10)	Integer i = new Integer(10)		
Person p1 = new Person("John")	Person p2 = new Person("John")		
Integer i = new Integer(10)	int j = 10		

Sorting - Comparable - Comparator

```java
class Student implements Comparable<Student> {

    @Override

    public int compareTo(Student student) {

        return this.surname.compareTo(student.getSurname());

    }

}
```

```java
List<Student> classroom= new ArrayList<>();

classroom.add(new Student("Rossi", 7.5d));

classroom.add(new Student("Bianchi", 5.5d));

Collections.sort(classroom); // Bianchi, Rossi

Collections.sort(classroom, new Comparator<Student>() {

    @Override

    public int compare(Student o1, Student o2) {

        return o1.getAverage().compareTo(o2.getAverage());

    }

});
```

Meant for **natural sorting** order
Returns: - **Negative obj1 < obj2** - **Zero   obj1 == obj2** - **Positive   obj1 > obj2**
Frequently implemented in java class String, Date, Calendar, ecc

Meant for **customized sorting** order
Same as comparable
To be implemented if necessary. Implemented in a separate Class

Generics - Casting

```java
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Test {

    public class Pair<K extends Number, V> {
        K key;
        V value;

        public void set(K k, V v) {
            key = k;
            value = v;
        }

        <T extends Calendar> void test(T arg) {
            if(arg instanceof GregorianCalendar) {
                //do something
            }
        }
    }

    Pair<Integer,String> pair1 = new Pair<>();
    pair1.set(1, "Test");

    Pair<Integer,Object> pair2= new Pair<>();
    pair2.set(2, new Object());

    Calendar c = new GregorianCalendar();
    pair1.test(c);

}
```

Introduced with Java 1.5 a generic Type is a formal Type that is not clearly defined at compilation time.

The advantages on using Generics are:
- Type-safety
- Type casting is not required
- Compile-Time Formal Checking

```java
// Without generics
List list = new ArrayList();
list.add(new Integer(0));
Integer x = (Integer) list.get(0); // Explicit downcast
list.add("abc");
Integer y = (Integer) list.get(1); // Run -time exception

// With Generics
List<Integer> list = new ArrayList<> ();
list.add(new Integer(0));
Integer x = list.get(0);
list.add("abc"); // Compiler Error – Expected Integer
```
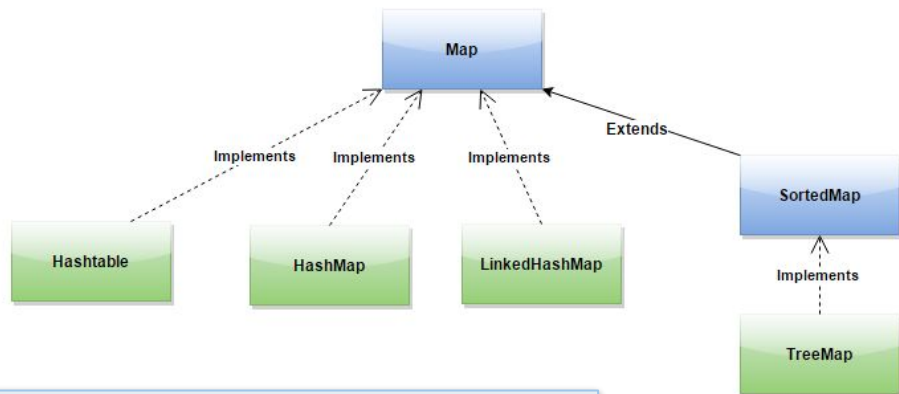
Maps

NOT a Collection.

Map<K,V> is an **interface** representing a data structure that allows you to store and search an element associated to a Key.

The **key** must be **unique**. If a second element is stored with an existing key it will **replace** the first.


```java
Map<String, Person> persons = new HashMap<>();
Person johnDoe = new Person("John","Doe");
Person janeDoe = new Person("Jane","Doe");
users.put("XXX", johnDoe);
users.put("XXX", janeDoe);
persons.get("XXX); // gets janeDoe

Set<String> keySet = persons.keySet(); // set of all the keys
Collection<Person> values = persons.values(); // collection of all values
Set<Map.Entry<String, Person>> entries = persons.entrySet()); // set of (K,V)
```

Exceptions
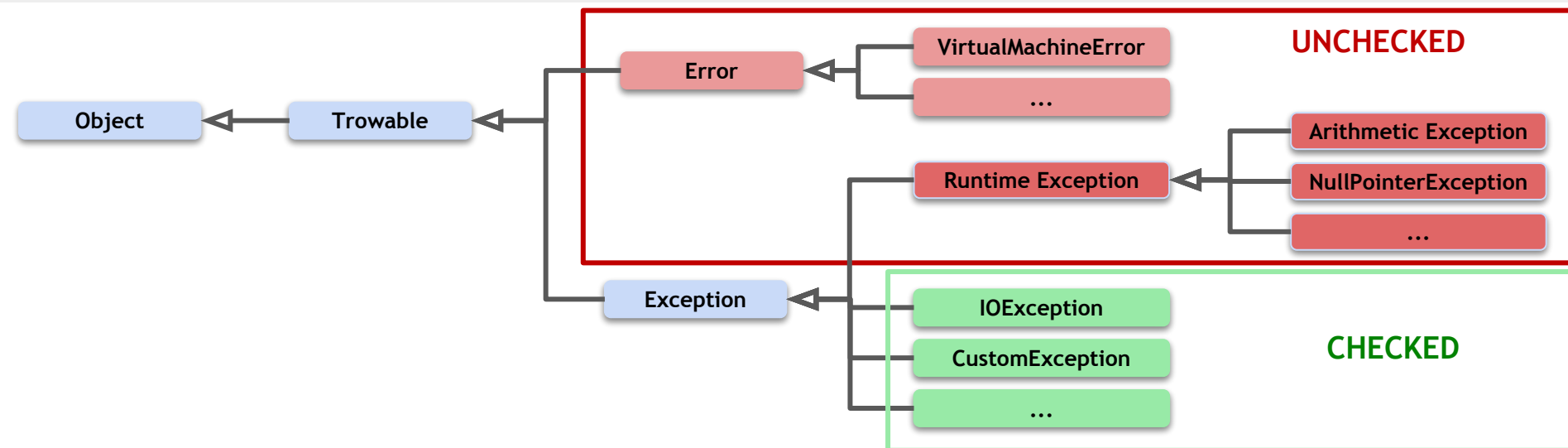
UNCHECKED: Tipically represents serius errors or logic errors. Catching is optional. Handled by JVM.

CHECKED: Tipically represents operation errors. Catching or declaration required.

Exceptions Handling

```java
public static void main(String[] args){
  try {
      methodCallExample();
  } catch(CustomException e) {
      // do something
  }
}
void methodCallExample() throws CustomException  {
  try {
     //do something
     throw new CustomException("message");
  } catch(NullPointerException npe) {
     // do something
  } catch(RuntimeException e) {
     //do something
  } finally {
     // do something
  }
}
class CustomException extends Exception {
  public CustomException() {}
  public CustomException(String message) {
    super(e);
  }
}
```

TRY: Encloses all statements that may throw an exception

CATCH: Catches exception matching type. Only the catch that match is executed, the evaluation is made by order priority.

FINALLY: Optional block. Always executed after try - catch block.

THROW: used to throw an exception

THROWS: used at method signature level to declare methods could throw an exception

Useful Links

Java:
[JavaDocs](#)
[Collections 1](#) - [Collections 2](#)
[Oracle Java SE tutorial](#)

Java 8:
[Java 8 in practice - Streams 1](#)
[Java 8 in practice - Streams 2](#)
[Java 8 in practice - Optional](#)
[Java 8 in practice - Virtual Extension Method](#)

Github project course repository and Lessons documentation:
https://github.com/mcolombosperoni/an-introduction-to-backend-for-beginners