Java Programming: Step by Step from A to Z

Introduction

About me

- from Budapest, Hungary
- BSc in physics
- MSc is applied mathematics
- working as a software engineer at the moment
- special addiction to models concerning quantitative finance such as the Black-Scholes model or credit risk

About the course

"Java programming: Step by Step from A to Z"

"First Steps in Java"

About the course

If we think of Java Programming as a big mountain we can say that in the "First Steps in Java" course we went through the peaks of the Great Mountain of Java Knowledge. It was an interesting and useful journey and it gave a good overview for us in this programming language.

About the course

But if we want to understand better Java Programming we have to go deeper in the mountains, down into the valleys to discover the connections between different parts of Java and to expand our knowledge.

About the course

This is the purpose of this course.

Java programming: Step by Step from A to Z

Great Mountain of Java Knowledge

About the content of the course

More about Operators
Wrapper Classes
Garbage Collection
Java Heap & Stack Memory
More about Strings
String Buffer and String Builder
Enums & Dates and Times
More about loops and Conditional Statements
More about Arrays
Varargs & ArrayList
Generics & the Collections Framework
Nested Classes
More about Exceptions
More about Abstraction, Encapsulation, Polymorphism
Serialization
Lambda Expression & Functional Interface
Method Reference
Optional & Stream
Multithreading & Design Patterns

HD

Java Programming: Step by Step from A to Z

MORE ABOUT Operators

Operators

Java provides operators to manipulate variables. Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

- Arithmetic Operators
- Relational Operators
- Assignment Operators

Basic Operators

- Bitwise Operators
- Logical Operators
- Conditional Operator
- Type Comparison Operator

Hint! Previously in: First Steps in Java, lecture 13, "Basic operations"

Operators

Java provides operators to manipulate variables. Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

- Arithmetic Operators
- Relational Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Conditional Operator
- Type Comparison Operator

Basic Operators

Bitwise operator works on bits and performs the bit-by-bit operation

Operators

Java provides operators to manipulate variables. Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

- Arithmetic Operators
- Relational Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Conditional Operator
- Type Comparison Operator

Bitwise operator works on bits and performs the bit-by-bit operation

In the following lectures we will focus on these operators

Arithmetic Operators & Relational Operators

Reminder

Arithmetic

+ (Addition)

- (Subtraction)

* (Multiplication)

/ (Division)

% (Modulus)

++ (Increment)

-- (Decrement)

Relational

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

Assignment Operators

Reminder

Operator	Name	Example	Equivalent
=	Simple assignment	i = 22;	i = 22;
+=	Addition assignment	i += 33;	i = i + 33;
-=	Subtraction assignment	i -= 44;	i = i – 44;
*=	Multiplication assignment	i *= 55;	i = i * 55;
/=	Division assignment	i /= 66;	i = i / 66;
%=	Modulus (Remainder) assignment	i %= 77;	i = i % 77;

Bitwise Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. These operators are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The bitwise **&** operator performs a bitwise AND operation.
The bitwise **^** operator performs a bitwise exclusive OR operation.
The bitwise | operator performs a bitwise inclusive OR operation.
The bitwise **~** operator is unary and has the effect of 'flipping' bits.

Logical Operators

A logical operator (sometimes called a "Boolean operator") returns a Boolean result that's based on the Boolean result of one or two other expressions.

&& (logical AND) If both the operands are non-zero, then the condition becomes true.

| | (logical OR) If any of the two operands are non-zero, then the condition becomes true.

! (logical NOT) Reverse the logical state of the operand.

^ (logical exclusive or XOR) The condition becomes true if any of the operands are true, BUT not both!

Conditional Operator

- conditional operator is also known as the **Ternary Operator.**
- the Java ternary operator functions like a **simplified Java if** statement.
- the ternary operator consists of a condition that evaluates to either true or false, plus a value that is returned if the condition is true and another value that is returned if the condition is false.

The ternary operator syntax is:

```
result = testCondition ? value1 : value2
```

If testCondition is true: result = value1
Otherwise: result = value2

Type Comparison Operator

- the Type Comparison Operator is also known as the **instanceof Operator**.
- the instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof operator syntax is:

```
if (objectReference instanceof type)
```

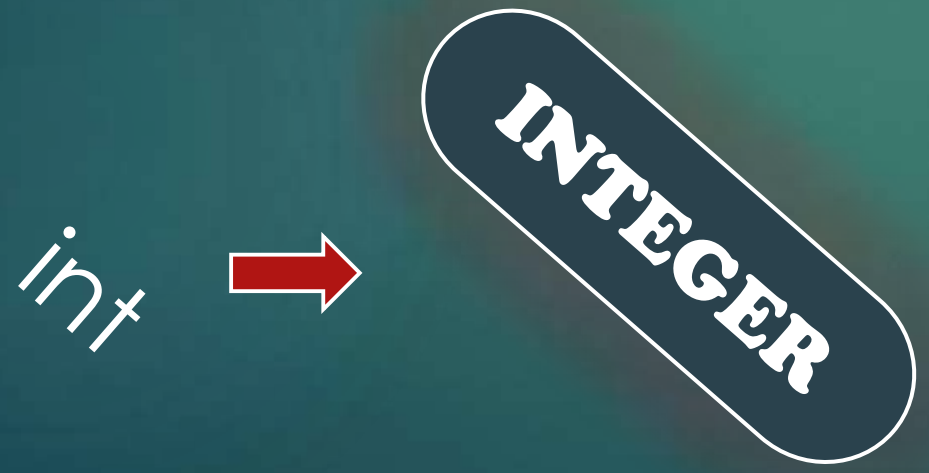
Java Programming: Step by Step from A to Z

Wrapper Classes

Wrapper classes

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself.

Java provides wrapper classes for each of the primitive data types and the mechanism to convert primitive into object and object into primitive.



Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

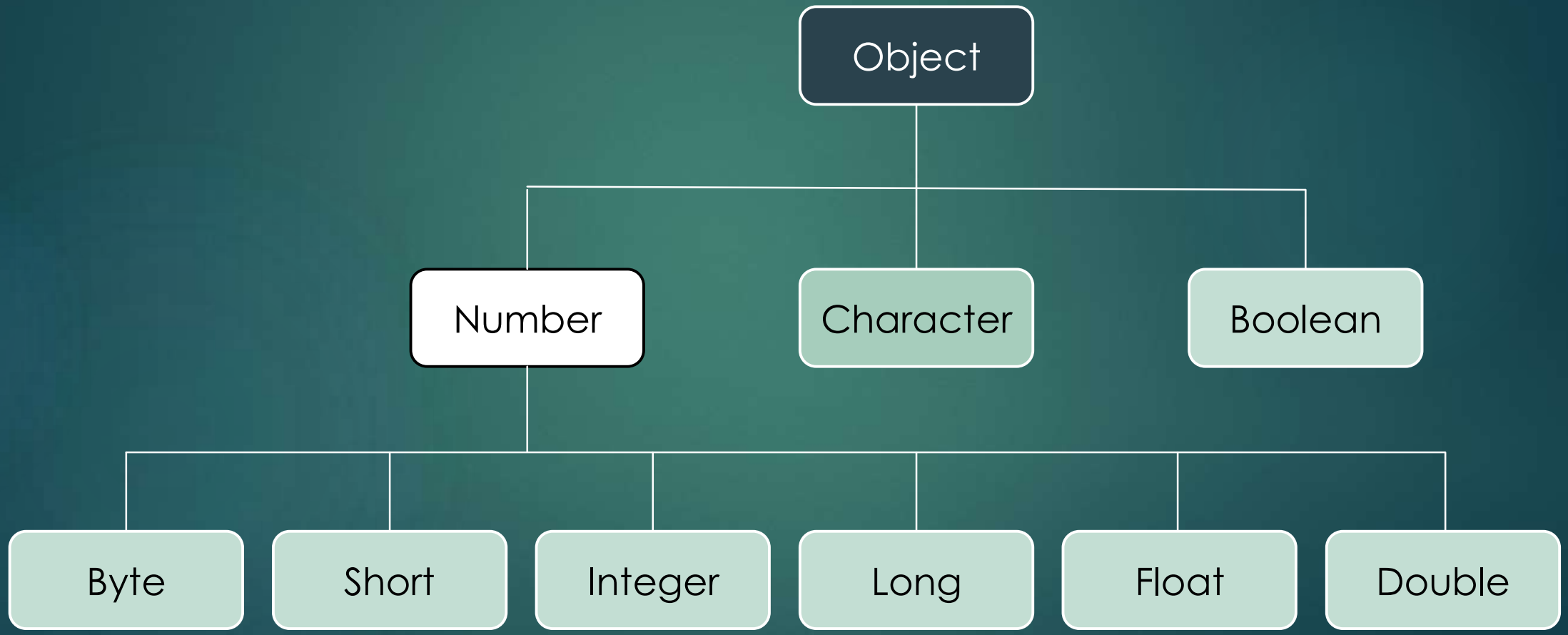
There are eight wrapper classes:

Primitive Data Type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

These wrapper classes are subclasses of the abstract class **Number**

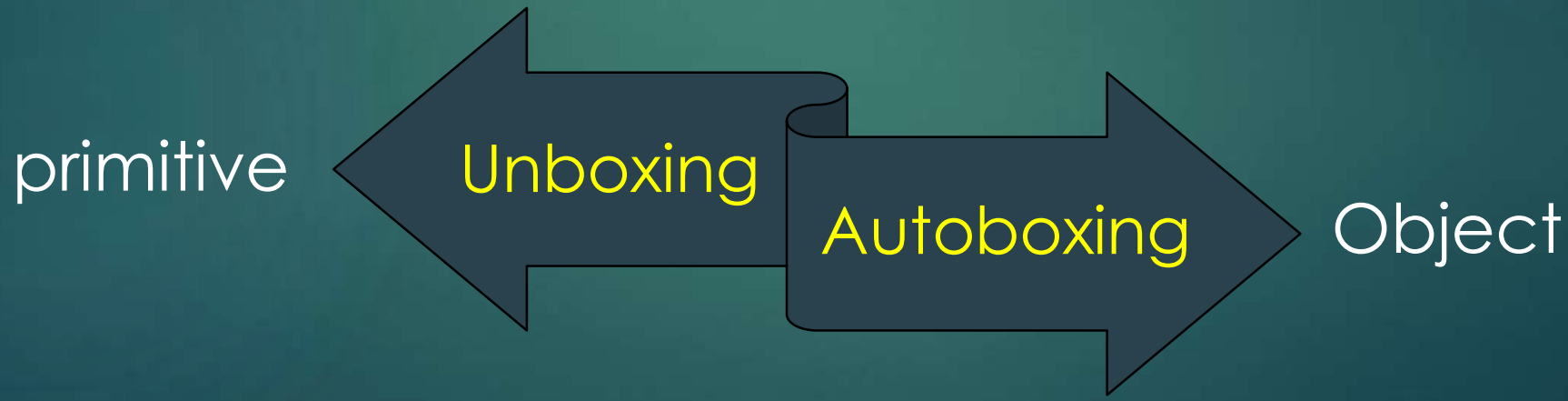
Hint! Previously in: First Steps in Java, lecture 8, „Data Types"

Wrapper class hierarchy:



Autoboxing / unboxing

Autoboxing and **unboxing** feature converts primitive into object and object into primitive automatically.



Autoboxing / unboxing example

Integer intObj = 55;

simplest example of autoboxing

what happens in the background invisible

Integer intObj = Integer.valueOf(55);

int intPrimitive = intObj;

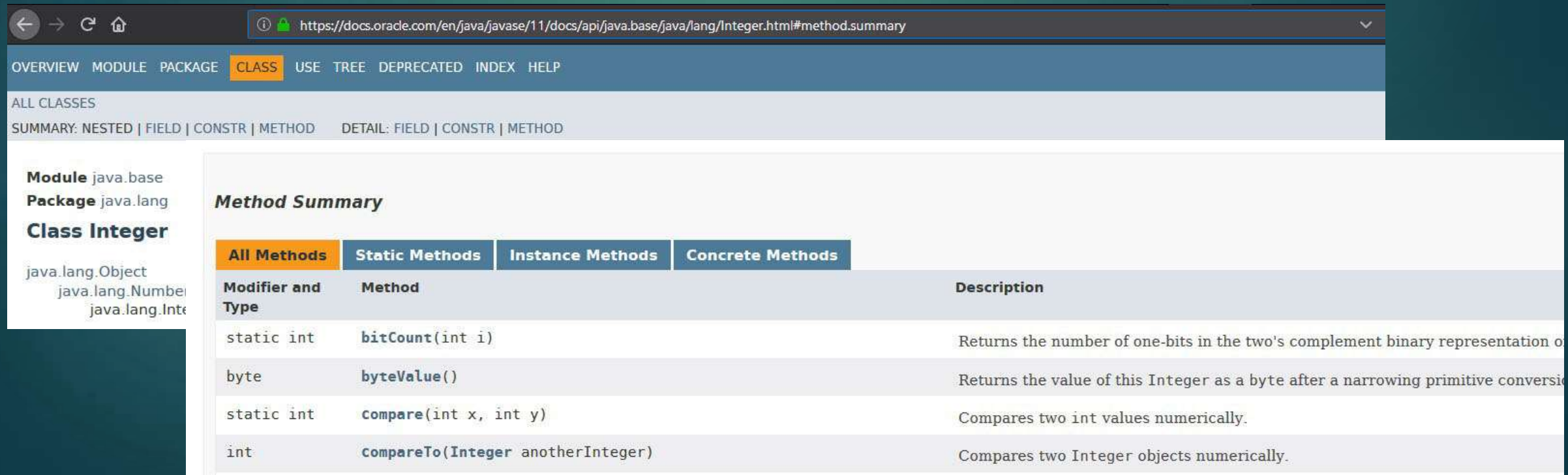
unboxing the Object (intObj)

Methods of the Wrapper classes

One of the objective of the **Wrapper** classes is to define several utility methods which are required for the primitive types.

There are lots of methods for **Wrapper** classes. You can look around on the website of Oracle.

Java API Documentation → **https://docs.oracle.com/en/java/**

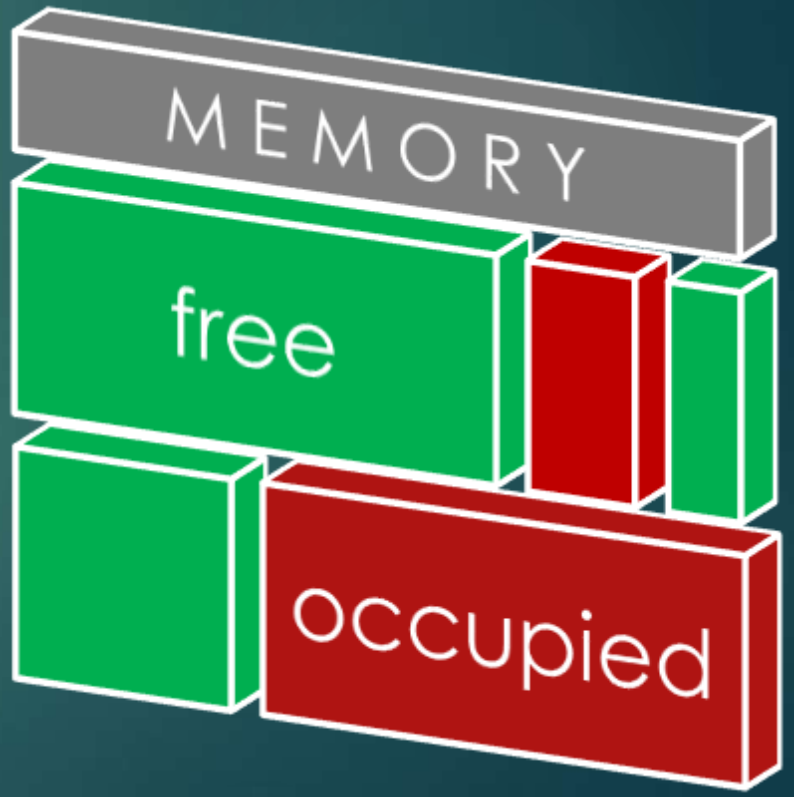


Java Programming: Step by Step from A to Z

Garbage Collection

Garbage Collection overview

In other programming languages like C/C++ the programmer has to deal with memory management manually. In these languages the programmer is responsible for both the creation and the destruction of objects. Usually the programmer neglects destruction of useless objects and this can cause out of memory errors (the memory becomes full).



Garbage Collection overview

In Java we are in a better situation because Java has automatic memory management, called Garbage Collector that works in the background. The Garbage Collector runs on the heap memory (we will discuss heap memory in the next lecture) to free the memory used by objects that doesn't have any reference.

Garbage Collector works in two simple steps:

Mark

The Garbage Collector identifies which pieces of memory are in use and which are not.

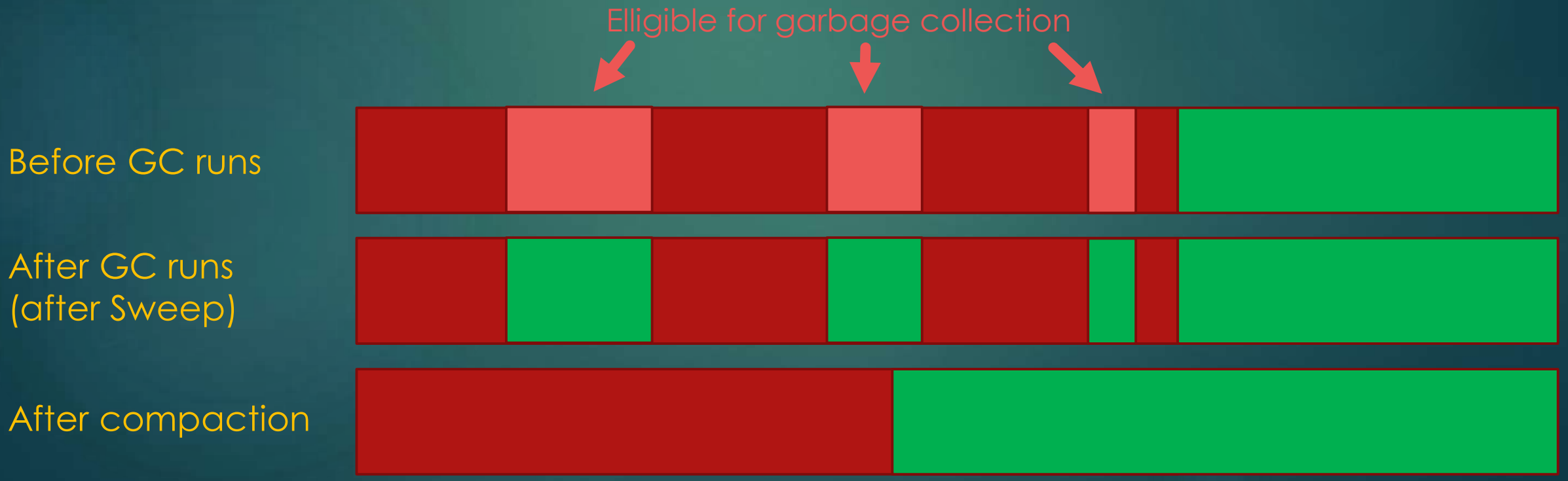
Sweep

The GC automatically removes objects identified during the "mark" phase.

Garbage Collection overview

Once we made an object eligible for garbage collection (Mark), it may not be destroyed immediately by Garbage Collector. When the JVM runs Garbage Collector program, the object will be destroyed (Sweep).

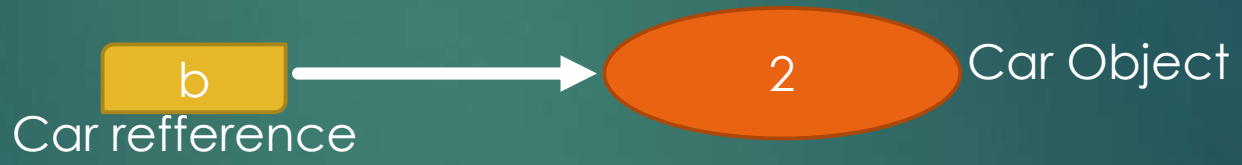
System checks whether there's enough free space in memory, and if there isn't, it runs the GC to free up space. You have no direct control over GC in Java.



How GC works

Car a = new Car();

Car b = new Car();



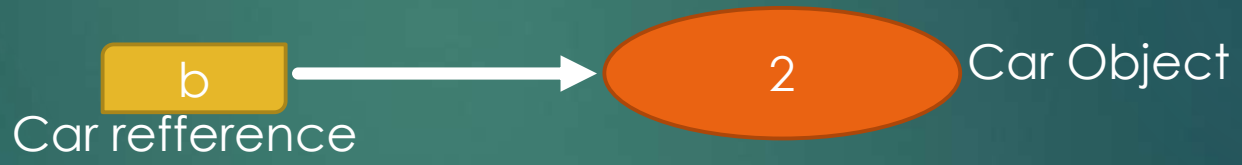
Car c = new Car();

Hint! Previously:
First Steps in Java,
lecture 9,
„Classes and objects"

How GC works

Car a = new Car();

Car b = new Car();



Car c = new Car();

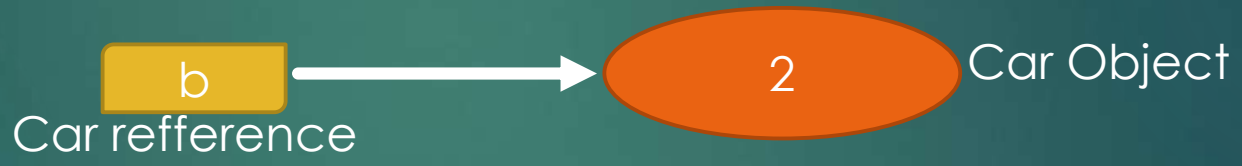
a = null;

Hint! Previously:
First Steps in Java,
lecture 9,
„Classes and objects"

How GC works

Car a = new Car();

Car b = new Car();



Car c = new Car();

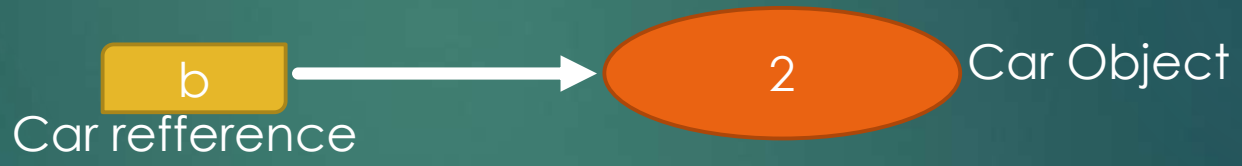
a = null;

How GC works

Don't forget! An objects that doesn't have any reference elligible for garbage collection.

Car a = new Car();

Car b = new Car();



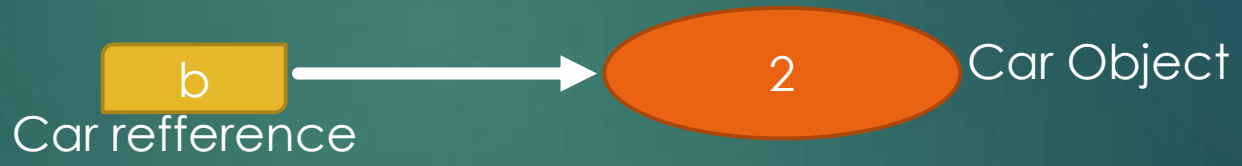
Car c = new Car();

a = null;

How GC works

Car a = new Car();

Car b = new Car();



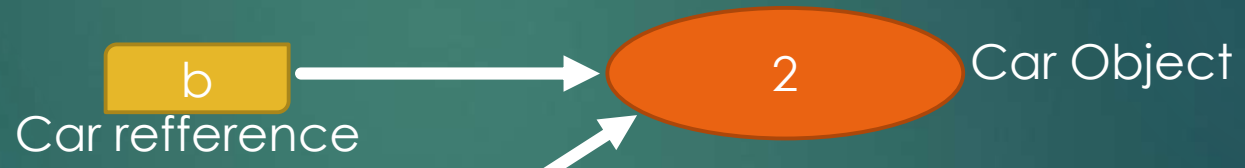
Car c = new Car();

a = null; **c = b;**

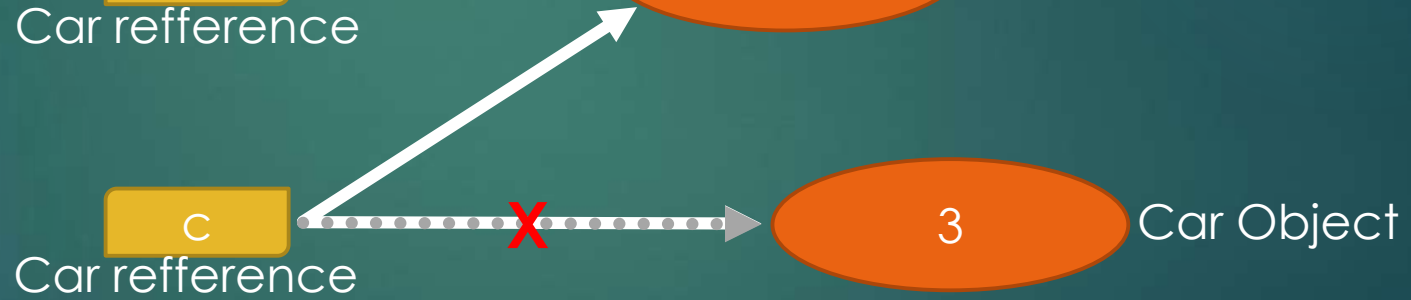
How GC works

Car a = new Car();

Car b = new Car();



Car c = new Car();

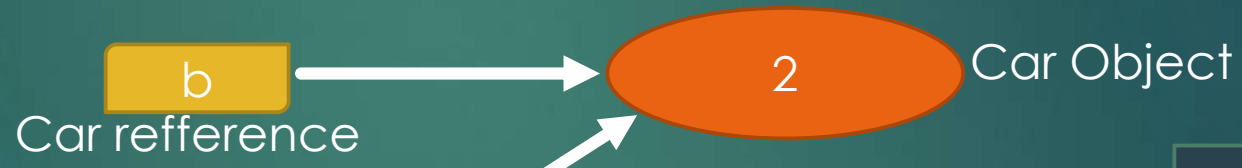


a = null; **c = b;**

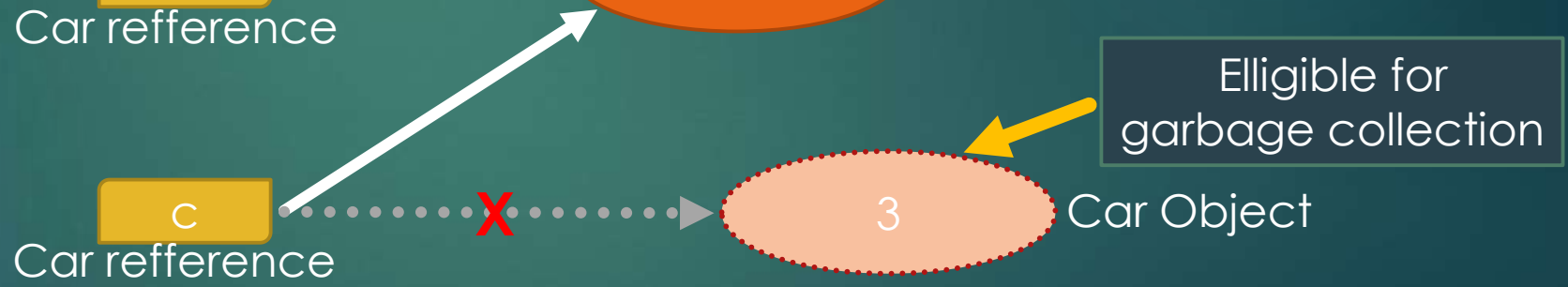
How GC works

Car a = new Car();

Car b = new Car();



Car c = new Car();

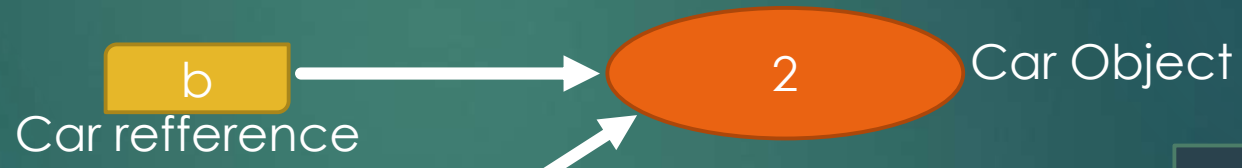


a = null; **c = b;**

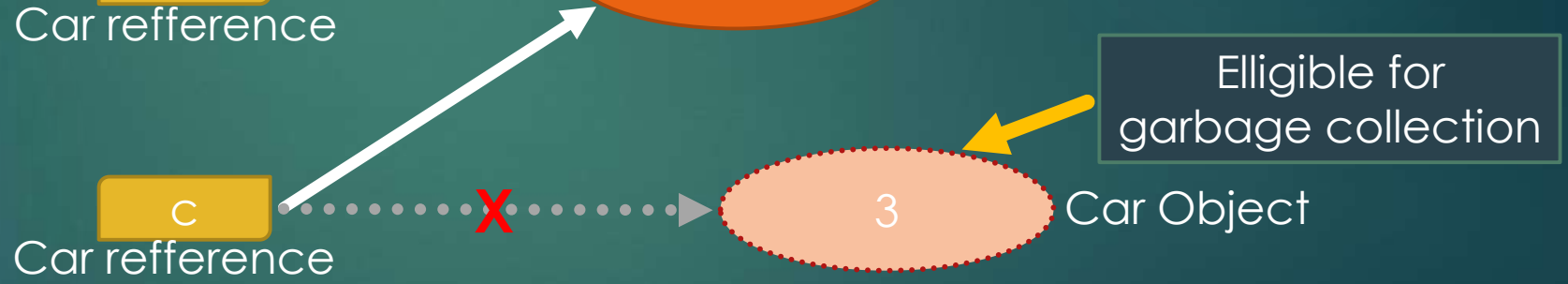
How GC works

Car a = new Car();

Car b = new Car();



Car c = new Car();



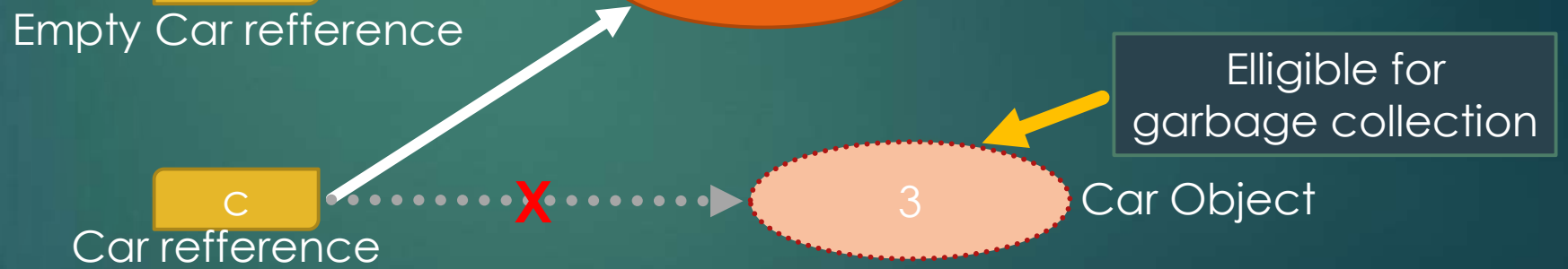
a = null; **c = b;** **b = null;**

How GC works

Car a = new Car();

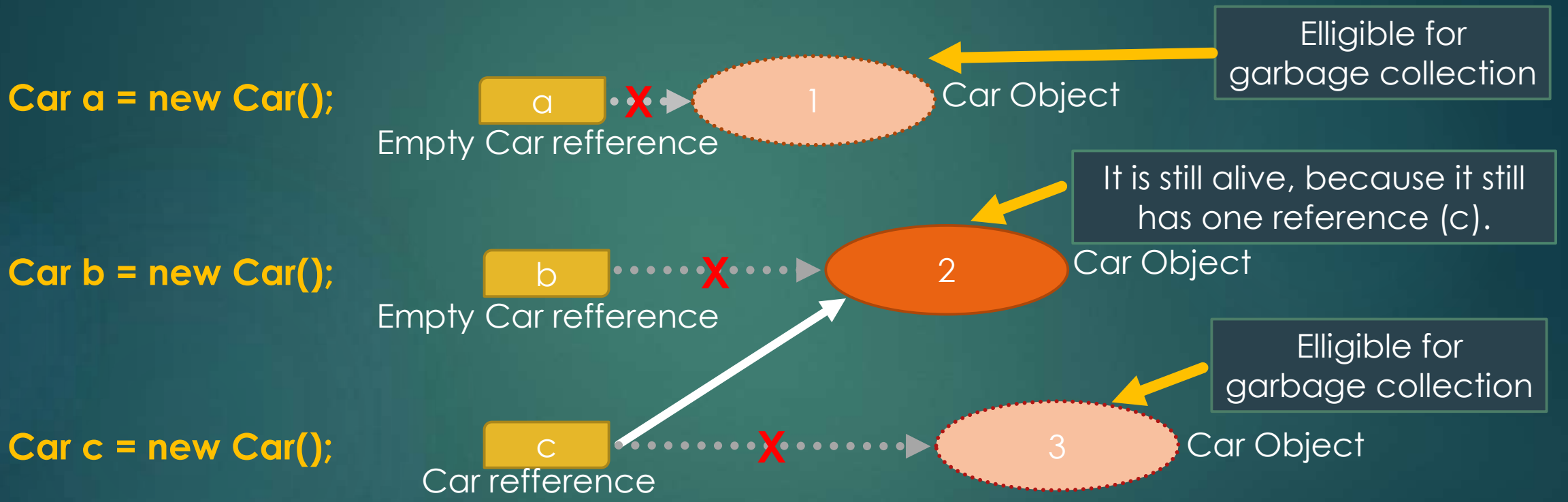
Car b = new Car();

Car c = new Car();

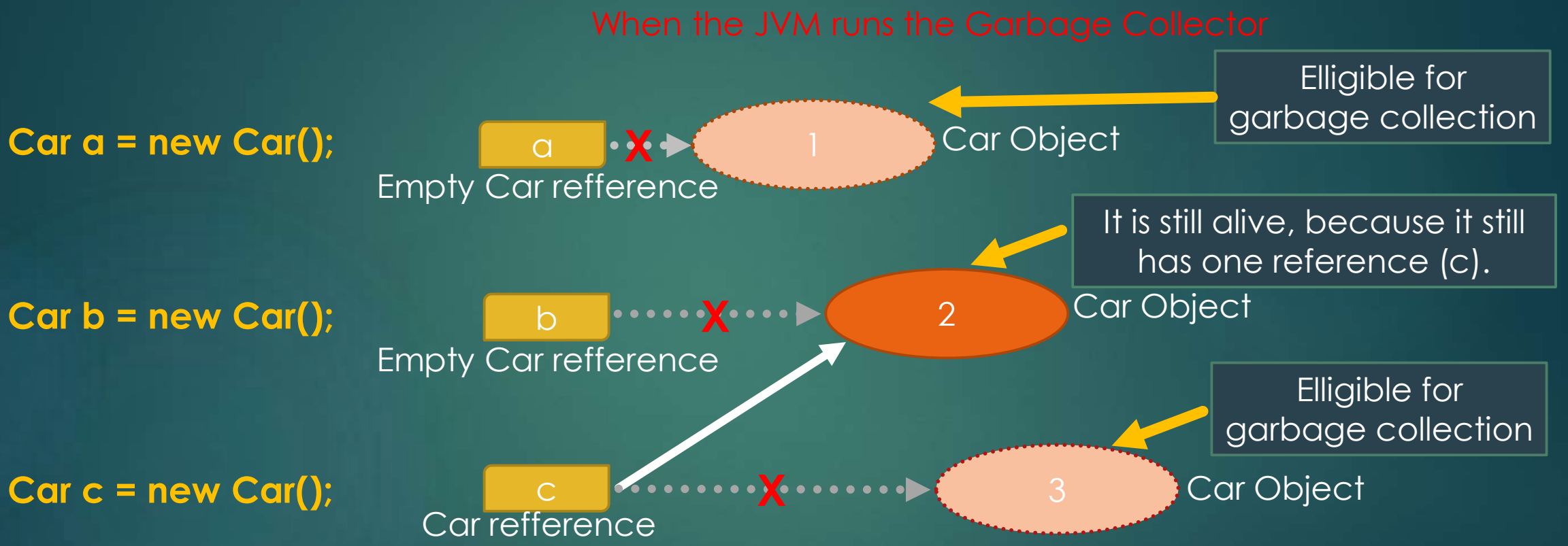


a = null; **c = b;** **b = null;**

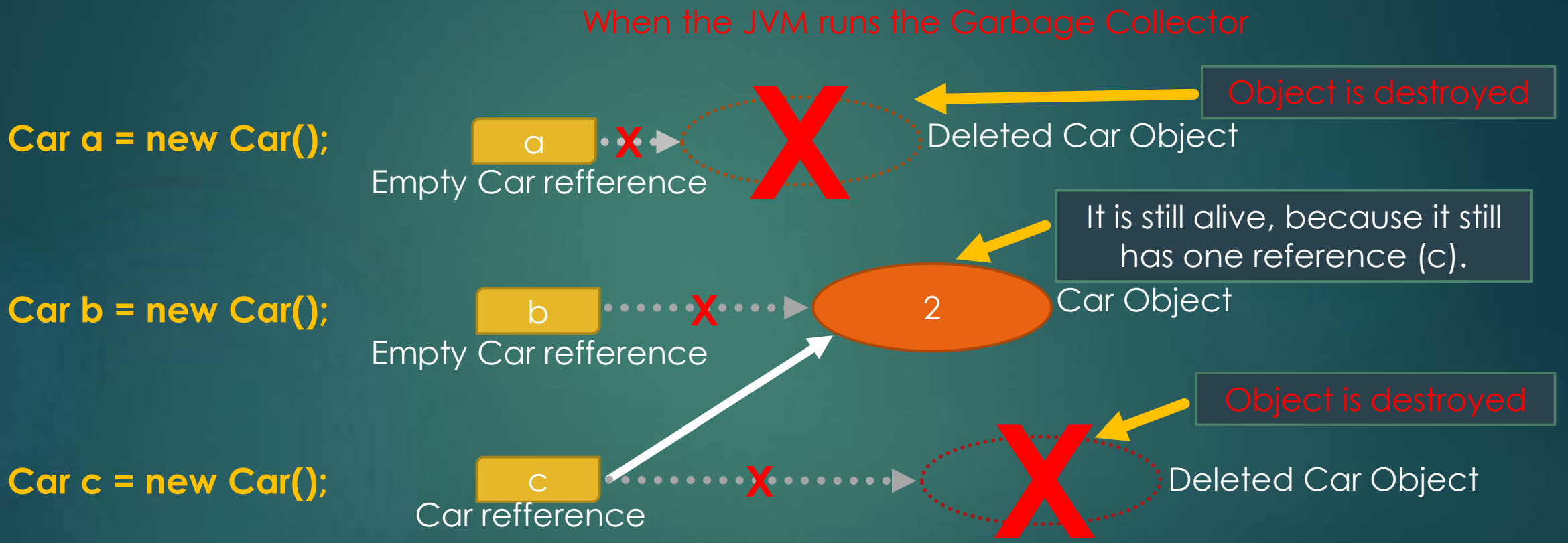
How GC works


Car a = new Car();
a
Empty Car refference
1
Car Object
Elligible for garbage collection
Car b = new Car();
b
Empty Car refference
2
Car Object
It is still alive, because it still has one reference (c).
Car c = new Car();
c
Car refference
3
Car Object
Elligible for garbage collection
a = null;
c = b;
b = null;

How GC works


When the JVM runs the Garbage Collector
Car a = new Car();
a
Empty Car refference
1
Car Object
Elligible for garbage collection
Car b = new Car();
b
Empty Car refference
2
Car Object
It is still alive, because it still has one reference (c).
Car c = new Car();
c
Car refference
3
Car Object
Elligible for garbage collection
a = null;
c = b;
b = null;

How GC works

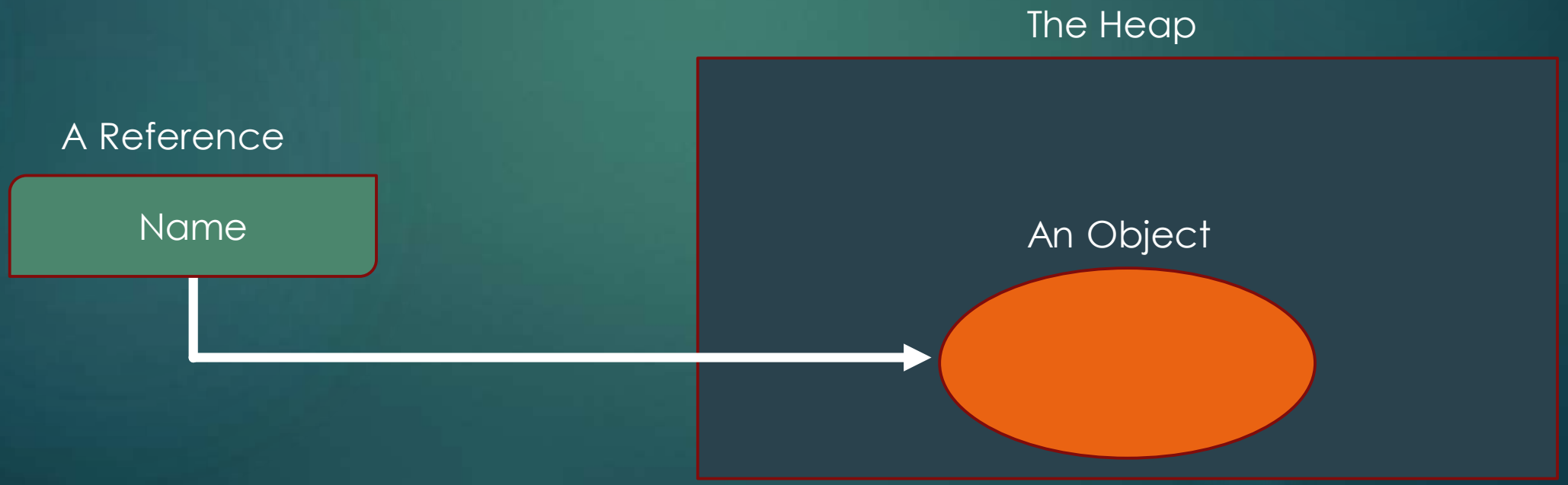

When the JVM runs the Garbage Collector
Car a = new Car();
a
Empty Car refference
Deleted Car Object
Object is destroyed
Car b = new Car();
b
Empty Car refference
2
Car Object
It is still alive, because it still has one reference (c).
Car c = new Car();
c
Car refference
Deleted Car Object
Object is destroyed
a = null;
c = b;
b = null;

Objects vs. References

Look out! Do not confuse a reference with the object that it refers to; they are two different entities.

The reference is a variable that has a name and can be used to access the contents of an object.
An object sits on the heap memory and does not have a name.

Therefore, you have no way to access an object except through a reference.

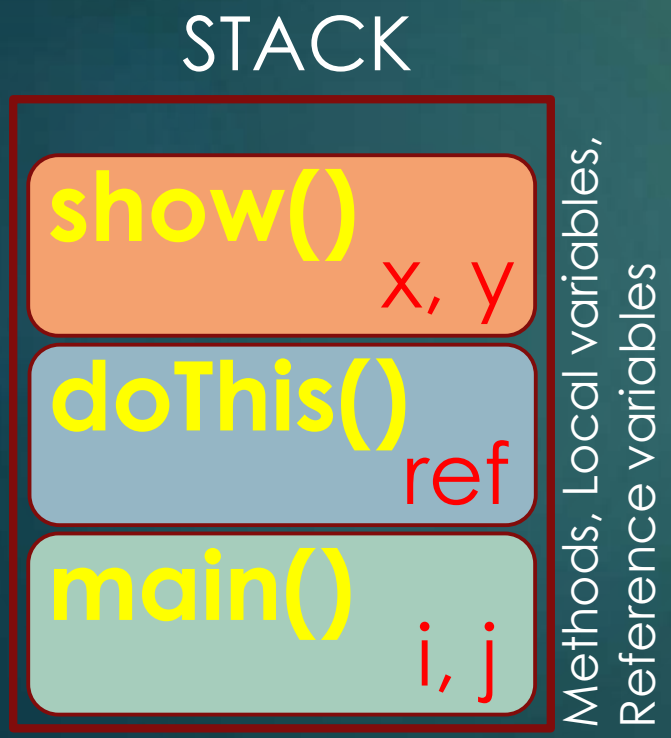


Java Programming: Step by Step from A to Z

Heap & Stack Memory

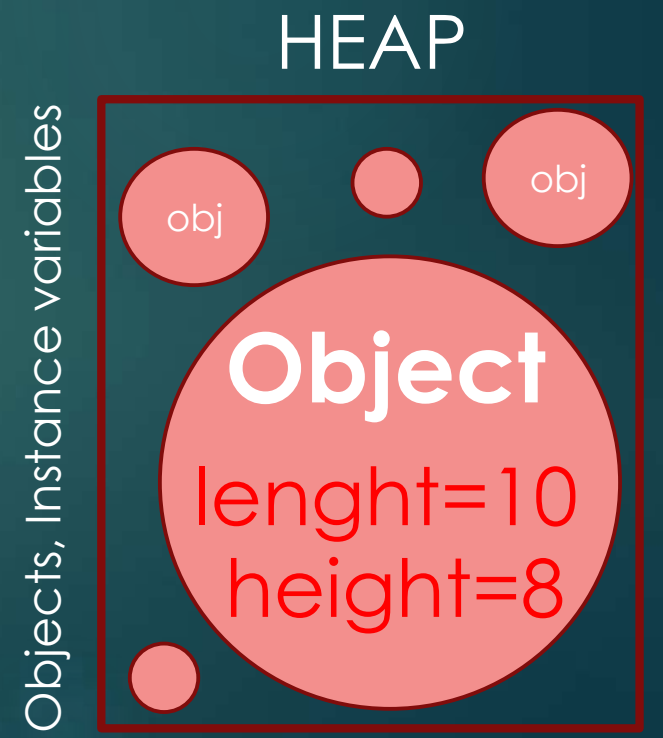
Heap and Stack memory overview

Knowing how memory actually works in Java is important, as it gives you the advantage of writing high-performance and optimized applications. It helps you to understand deeper the scope of variables, the object creation and the memory management. In Java the two main area of the memory are the **Stack** and the **Heap** memory.



Stack Memory contains primitive values that are specific to a method and references to objects that are in a heap, referred from the method.

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space.



Heap and Stack memory example

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


}
```

In the Stack a frame will be created from the main method

STACK

Heap and Stack memory example

```
public static void main(String[] args) {
    double d = 10;

}
```

A local variable d in main method will also be created in the main method's frame in the Stack.

STACK

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){



 }
```

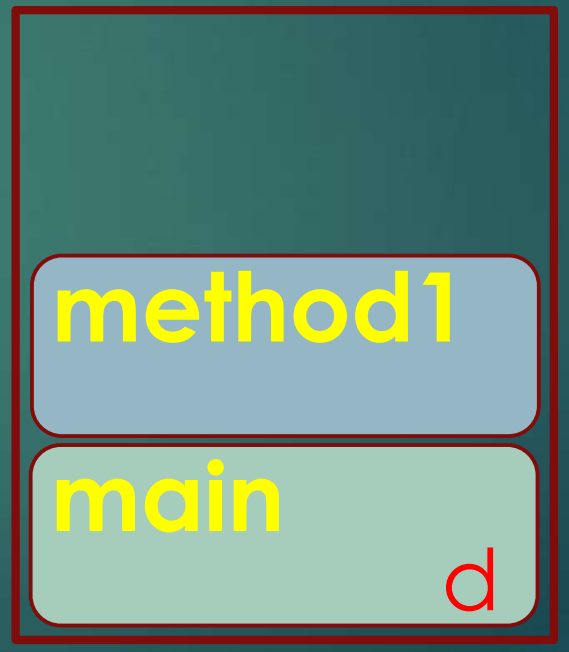
Main method is calling method1

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){


 }
```

Main method is calling method1

In the Stack a new frame will be created for method1 on the top of the main method's frame

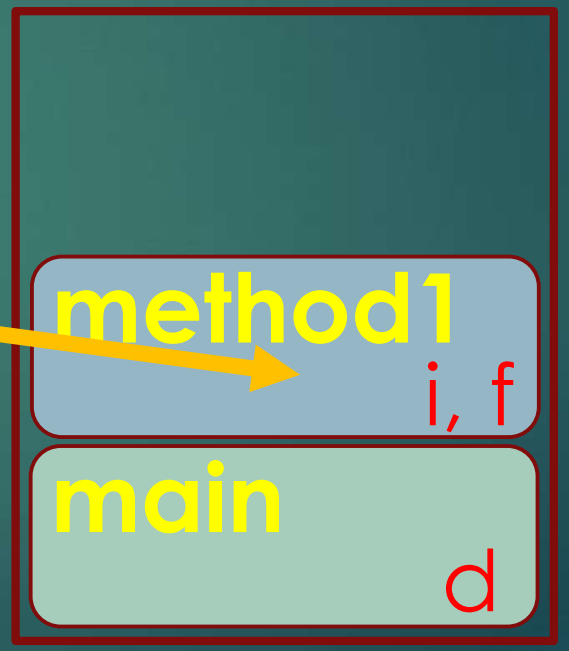


STACK

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;


 }
```

Variable i and f will also be created in the frame of method1 in the Stack

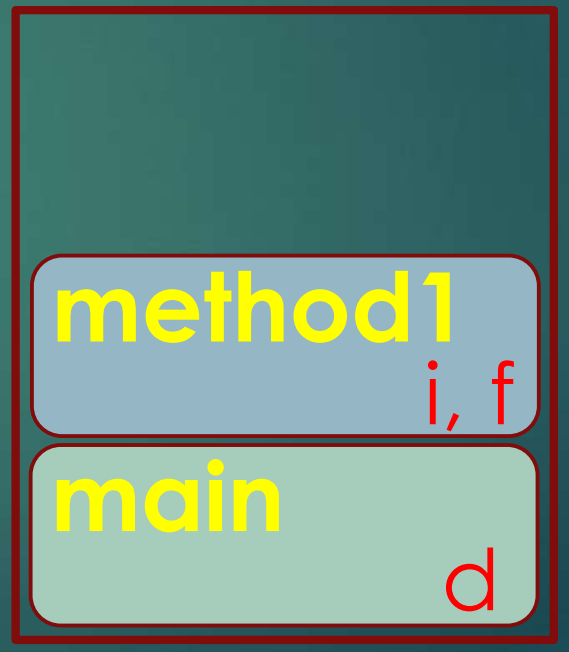


STACK

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){


  }
```

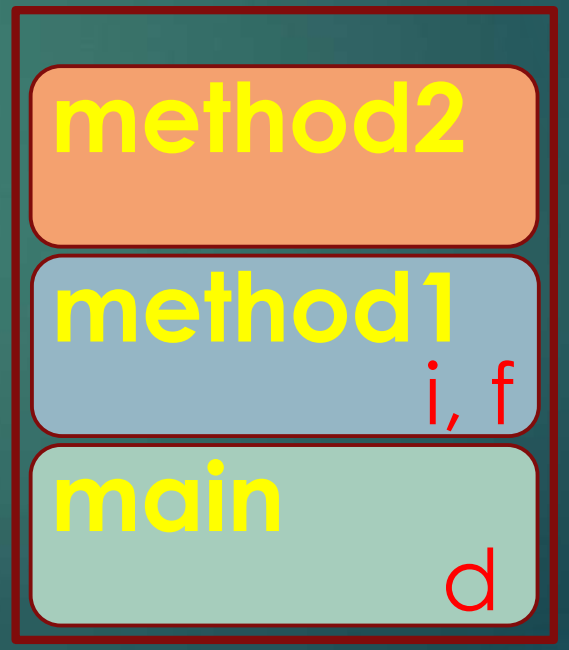
Method1 is calling method2



STACK

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){


  }
```

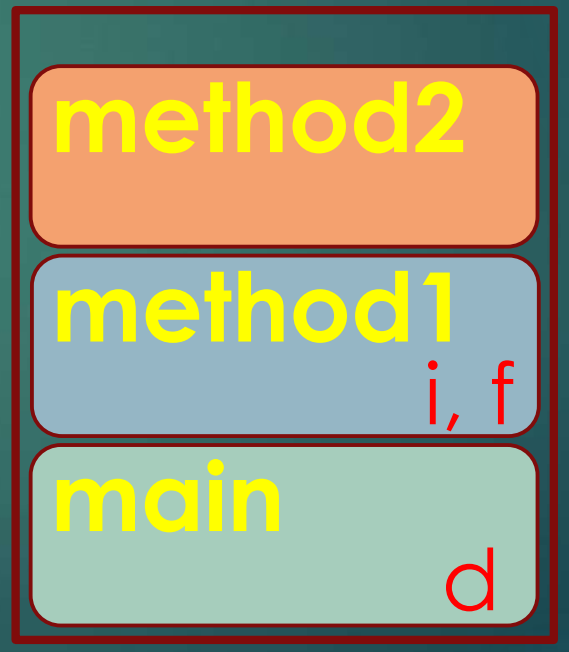


On the top of the Stack a frame for method2 is created.

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef  =  new House();

  }
```

```
public class House{
      int windows;
      int doors;
}
```



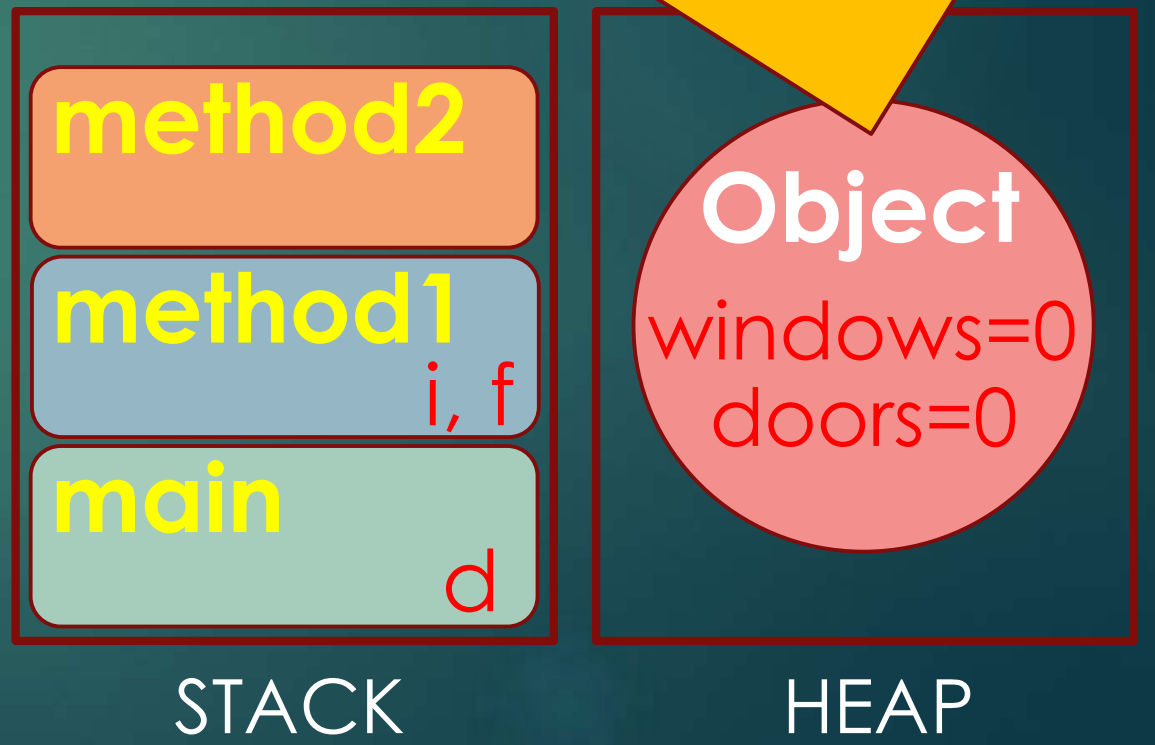
STACK

Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef  =  new House();

  }
```

```
public class House{
      int windows;
      int doors;
}
```

New operator is creating an Object in the Heap memory with the Object's instance variables.

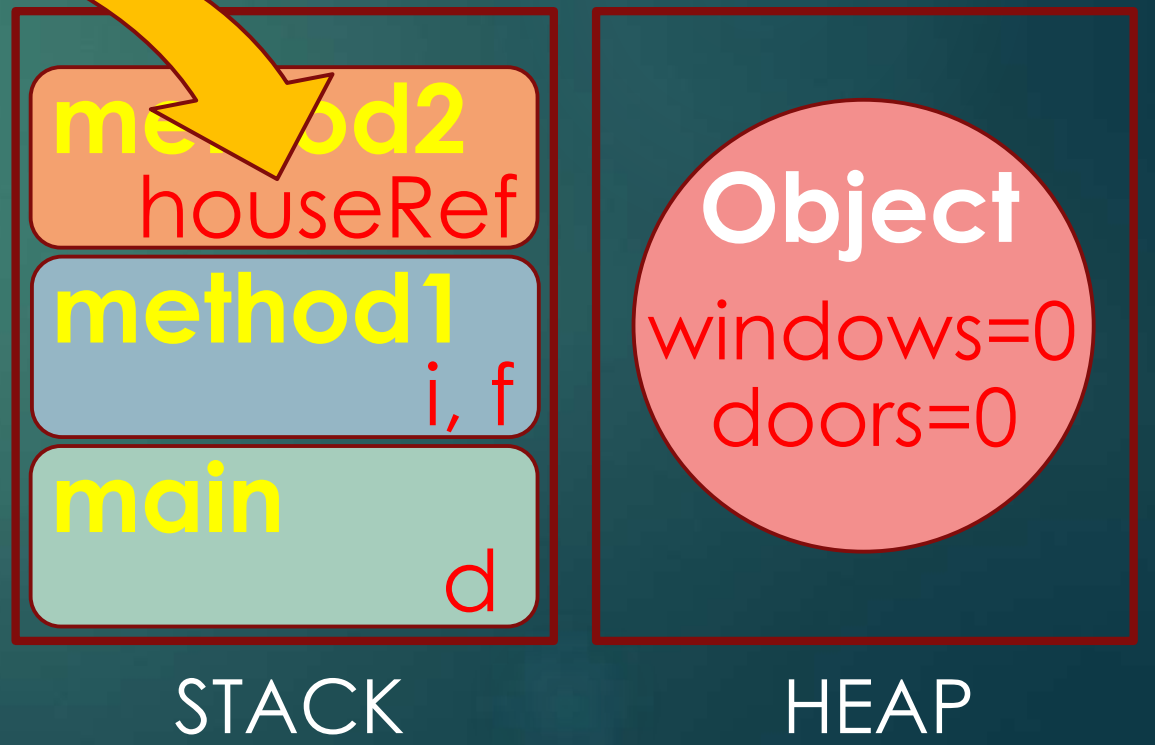


Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef =  new House();

  }
```

```
public class House{
      int windows;
      int doors;
}
```

The reference variable called houseRef is created in the Stack inside the frame method2.

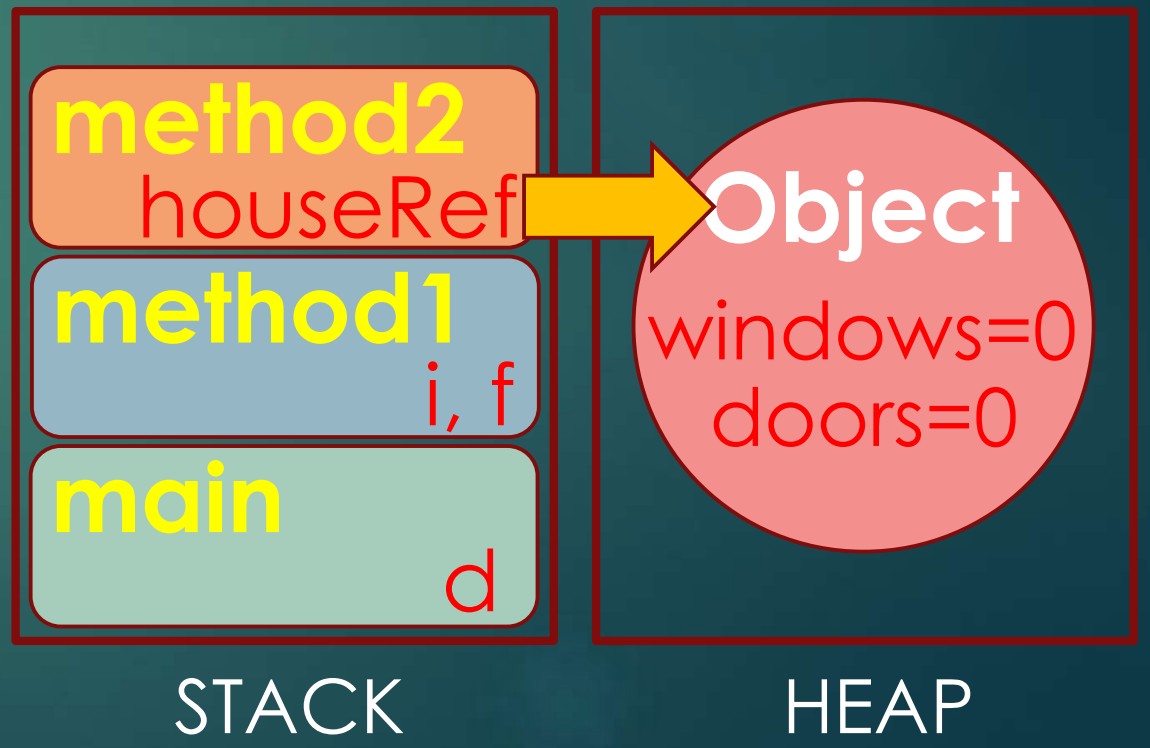


Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef = new House();

  }
```

```
public class House{
      int windows;
      int doors;
}
```

The assignment operator makes the houseRef reference variable to point to the House object in the Heap.



Heap and Stack memory example

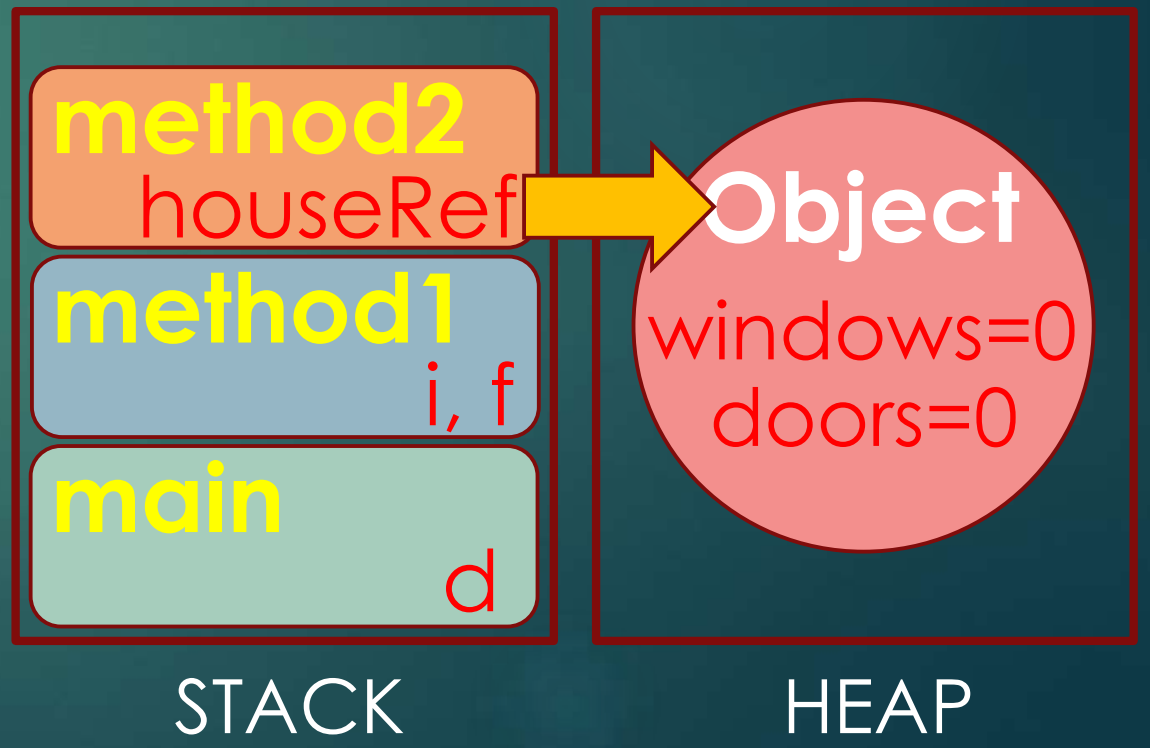
```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}
```

```
public class House{
      int windows;
      int doors;
}
```

```
public void method1(int i){
      float f = 30f;
      // more code here
       method2();
}
```

When method2 execution is completed the flow of the control will go back to the command method1.

```
Public void method2 ){
      House houseRef =  new House();
      // more code here
}
```



Heap and Stack memory example

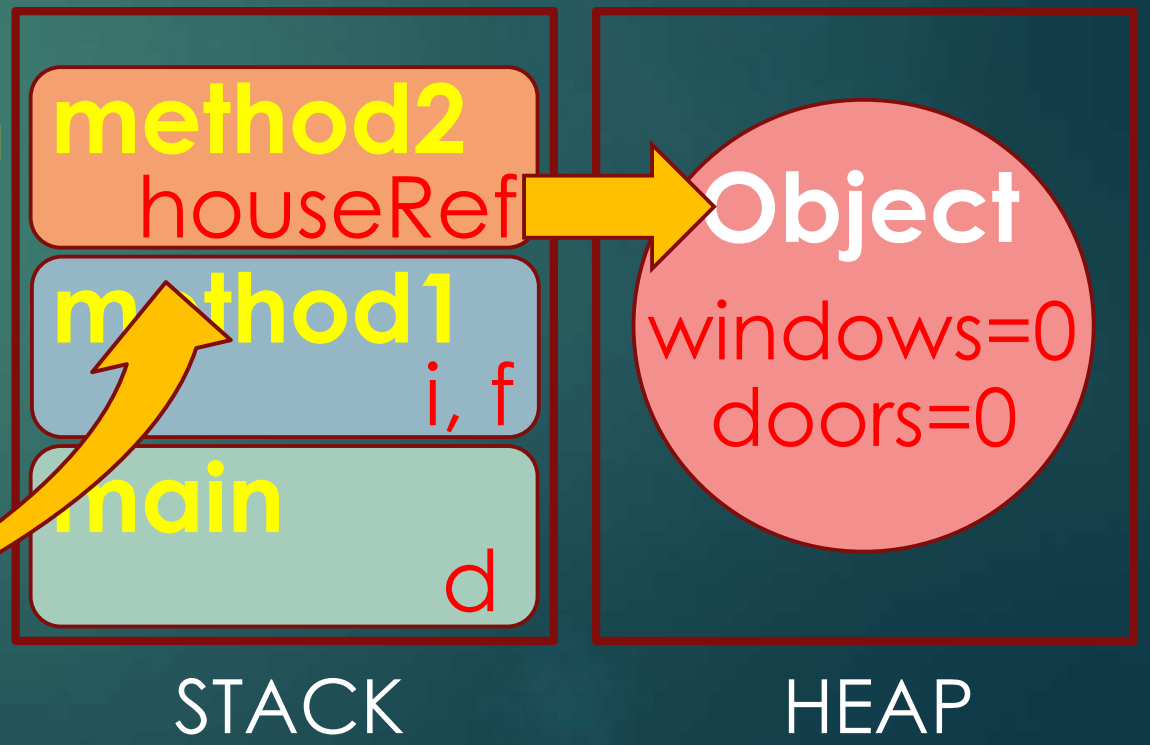
```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}
```

```
public class House{
      int windows;
      int doors;
}
```

```
public void method1(int i){
      float f = 30f;
      // more code here
       method2();
}
```

```
Public void method2 ){
      House houseRef =  new House();
      // more code here
}
```

Because method2 is completed it's flushed out from the Stack.

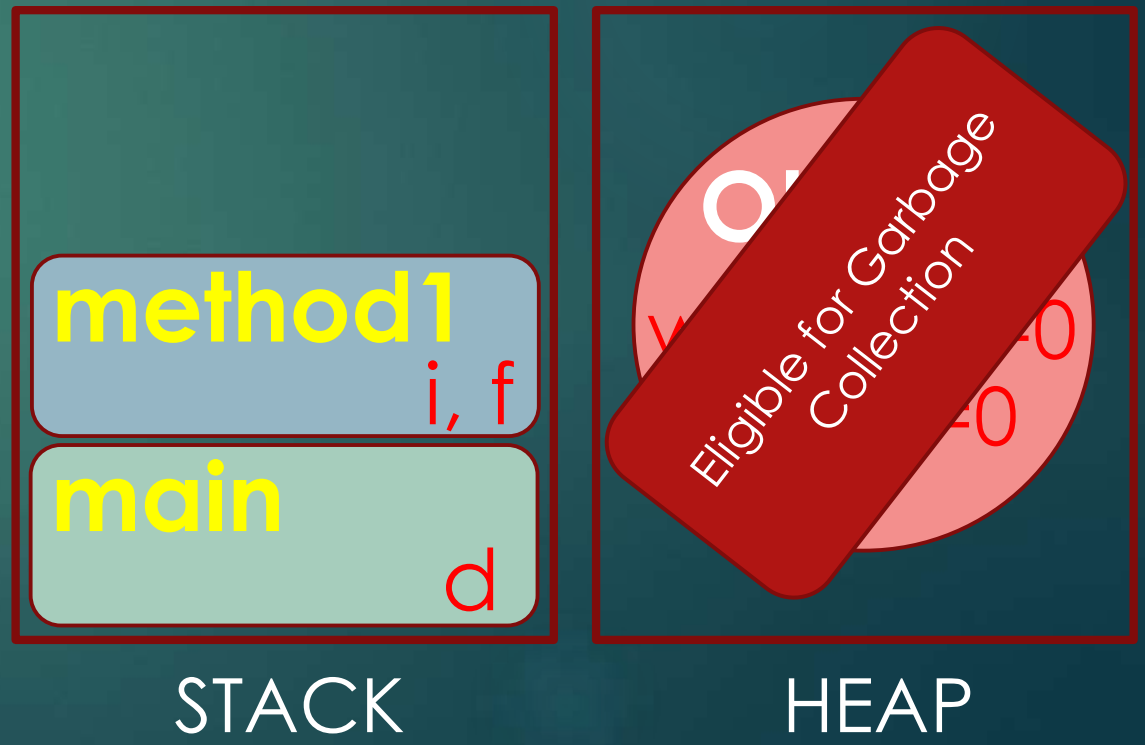


Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef  =  new House();
       // more code here
  }
```

```
public class House{
      int windows;
      int doors;
}
```

Since the houseRef reference variable will no longer be pointing to the Object, it will be Eligible for Garbage Collection

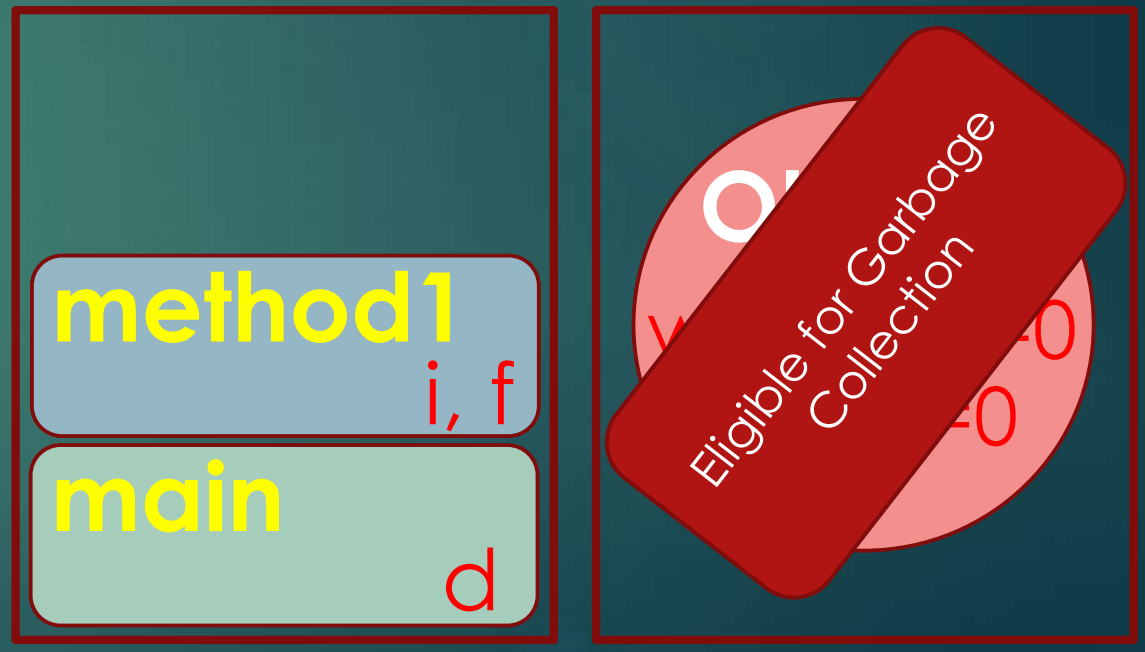


Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
 }

  Public void method2(){
       House houseRef =  new House();
       // more code here
  }
```

```
public class House{
      int windows;
      int doors;
}
```

When the JVM runs the Garbage Collector the Object will be destroyed



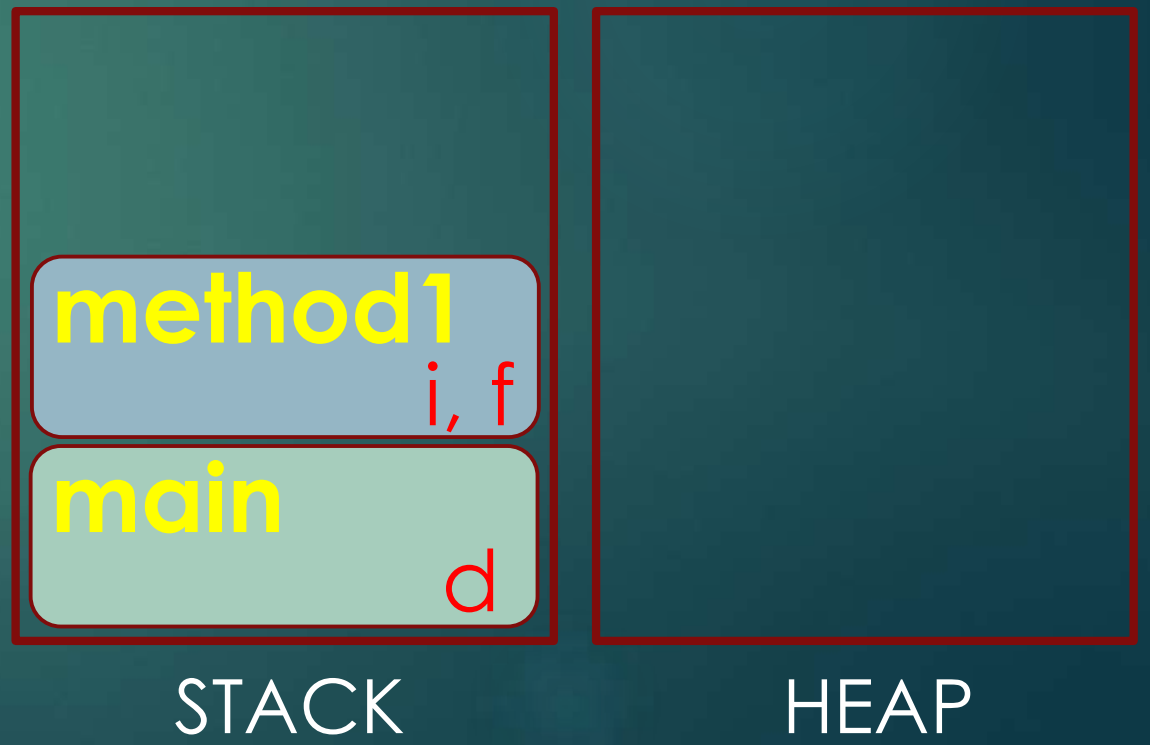
Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

public void method1(int i){
      float f = 30f;
      // more code here
      method2();
}

Public void method2(){
      House houseRef = new House();
      // more code here
}
```

When method1 completed its execution the flow of the control will go back to the command main method.

```
public class House{
      int windows;
      int doors;
}
```

When the JVM runs the Garbage Collector the Object will be destroyed

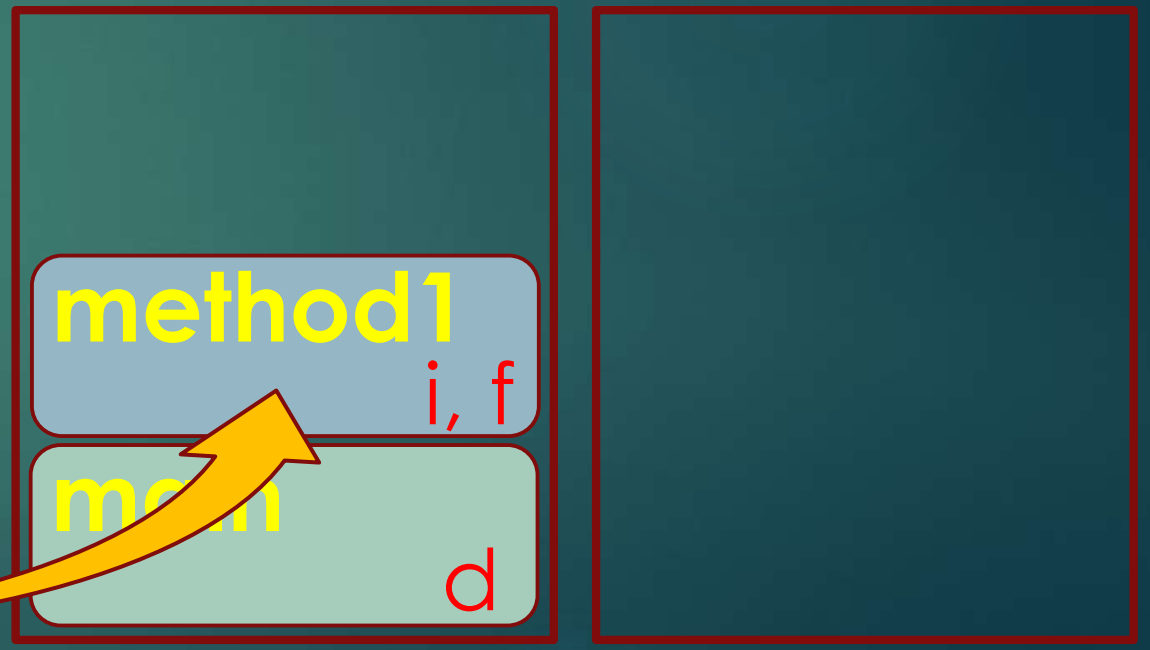


Heap and Stack memory example

```
public static void main(String[] args) {
      double d = 10;
      method1(20);
}

 public void method1(int i){
       float f = 30f;
       // more code here
        method2();
}

 Public void method2(){
       House houseRef =  new House();
       // more code here
 }
```

```
public class House{
      int windows;
      int doors;
}
```

And method1 will pop out from the Stack.

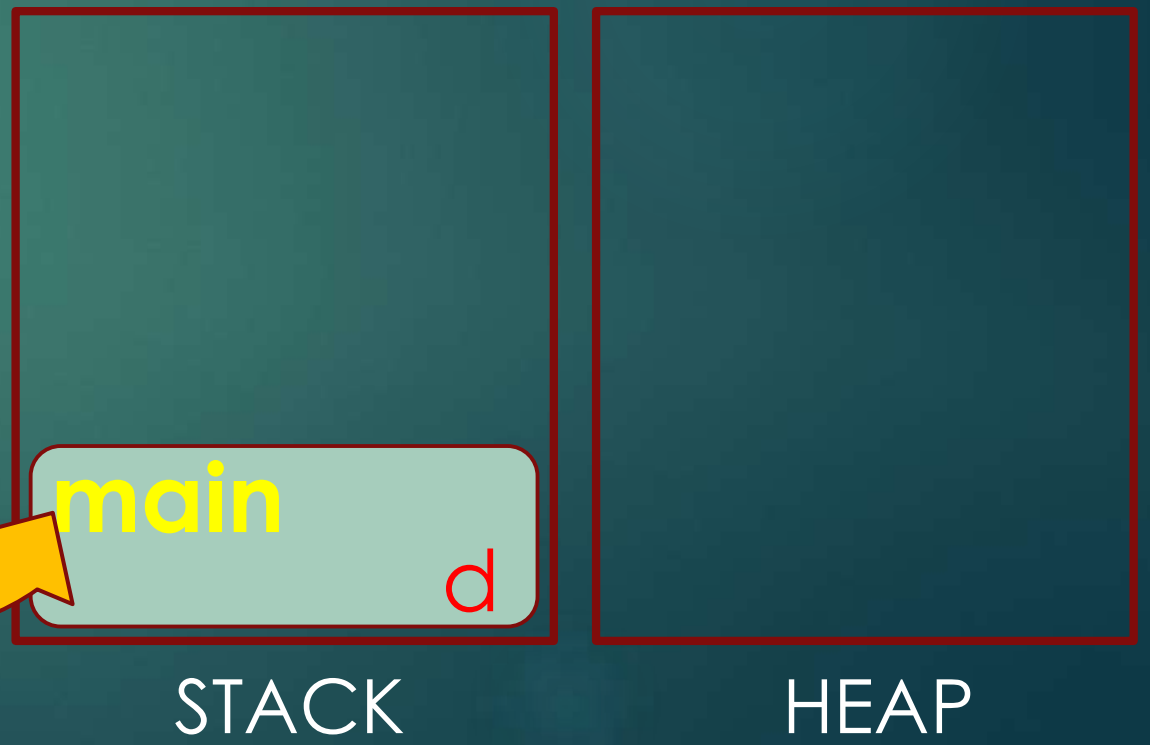


Heap and Stack memory example

```
public static void main(String[] args) {
    double d = 10;
    method1(20);
}

public void method1(int i){
    float f = 30f;
    // more code here
    method2();
}

Public void method2(){
    House houseRef = new House();
    // more code here
}
```

```
public class House{
    int windows;
    int doors;
}
```

When main method completed its execution main method's frame will be poped out from the Stack.



Heap and Stack memory example

```
public static void main(String[] args) {
    double d = 10;
    method1(20);
}

public void method1(int i){
    float f = 30f;
    // more code here
    method2();
}

Public void method2(){
    House houseRef = new House();
    // more code here
}
```

```
public class House{
    int windows;
    int doors;
}
```

When main method completed its execution main method's frame will be poped out from the Stack.

Heap and Stack memory, good to know

Any object created in the **HEAP** space has **global access** and can be referenced from anywhere of the application.

STACK memory is always referenced in **LIFO** (Last-In-First-Out) order. **Push** operation adds an element at the top of the stack, and the **pop** operation removes an element from the top of the stack.

We can use -Xms and -Xmx JVM option to define the **startup size and maximum size** of **heap** memory. We can use -Xss to define the **stack** memory size.

When **stack** memory is full, Java runtime throws java.lang.**StackOverFlowError** whereas if **heap** memory is full, it throws java.lang.**OutOfMemoryError**: Java Heap Space error.

Java Programming: Step by Step from A to Z

MORE ABOUT Constructors

Constructor overloading

Constructors overview

Reminder

As we have learned before the constructor is called when an object of a class is created. The name of the constructor matches the name of the class, it's syntactically similar to a method and there's no return type.

All classes have constructors by default: if you do not create a class constructor yourself, java compiler creates a default constructor for you. The **default constructor** doesn't have any parameters.

```
class ExampleClass {
    .......
    // A Default Constructor
    public ExampleClass() {}
    .......
}
```

Each time an object is created using the **new()** keyword at least one constructor (it could be the default constructor) is invoked to assign initial values to the data members of the same class. If we write a constructor then the compiler does not create a default one.

Hint! Previously in:
First Steps in Java,
lecture 9,
„Classes and objects"

Constructor Types

Reminder

Default constructor
We do not creat a constructor, so Java creates default in the background and it initializes class variables to their default values (null or 0).

```
public class MyClass {
    int x;
}
```

```
MyClass c = new MyClass();
System.out.print(c.x);
```

No argument Constructor
It initializes class variables with fixed values for all objects created by this constructor.

```
public class MyClass {
    int x;
    public MyClass() {
    x = 33;
    }
}
```

```
MyClass c = new MyClass();
System.out.print(c.x);
```

Parameterized Constructor
A constructor that has parameters is known as parameterized constructor. We initialize fields of the class with the constructor.

```
public class MyClass {
    int x;
    public MyClass(int i) {
    x = i;
    }
}
```

```
MyClass c = new MyClass(99);
System.out.print(c.x);
```

Constructor Overloading

Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter list. The compiler differentiates these constructors by the number of parameters in the list and their type.

```
public class Person{

    private String name;

    public Person(String n){
    name = n;
    }

    public Person(){
    name = "unknown";
    }
}
```

```
Person a = new Person("Kevin");
Person b = new Person();

System.out.println(a.name);
System.out.println(b.name);
```

Constructor Chaining

Overloaded constructors often call each other. One common technique is to have each constructor add one parameter until getting to the constructor that does all the work. It's called Constructor Chaining.

```
public class MyClass {
    public int x;
    public int y;
    public int z;
    public MyClass(int x) {
        this(x, 10); // calls constructor with 2 parameters
    }
    public MyClass(int x, int y) {
        this(x, y, 100); // calls constructor with 3 parameters
    }
    // this constructor does all the work
    public MyClass(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

```
MyClass c = new MyClass(3);
System.out.print(c.x +", "+ c.y + ", " + c.z);
```

Java Programming: Step by Step from A to Z

Overriding vs Overloading

Overriding vs Overloading

We have already learned about method Overloading and Overriding, but sometimes they can be confusing for **Java** novice programmers so now we will summarize the differences.

Overriding means having two methods with the same method name and parameters. One of the methods is in the parent class and the other is in the child class.

Method overriding is the example of run time polymorphism.

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Method overloading is the example of compile time polymorphism.

Hint! Previously in:
First Steps in Java, lecture 27, „Override"

Hint! Previously in:
Java programing: Step by Step from A to Z, lecture 17, „Method Overloading"

Overriding vs Overloading

```
public class Kangaroo {

    protected void jump() {
        System.out.println("Kangaroo jump!");
    }
}
```

```
public class GreyKangaroo extends Kangaroo {

    @Override
    protected void jump() {
        System.out.println("GreyKangaroo jump");
    }
}
```

Overriding

Method overriding occurs in two classes that have **IS-A** (inheritance) relationship.
Same method name, parameters must be same.

```
public class Kangaroo {

    public void jump() {
        System.out.println("Kangaroo jump!");
    }

    // Overload
    public void jump(int num) {
        for (int i = 0; i < num; i++) {
            System.out.println("Kangaroo jump");
        }
    }
}
```

Overloading

Method overloading is performed within class.
Same method name, parameters must be different.

Java Programming: Step by Step from A to Z

MORE ABOUT Strings

String's creation - behind the scenes

Strings overview

Reminder

Strings are a sequence of characters, and they are treated as an **object**.
We can create and manipulate strings with the String class.
There are two ways to create String object:

By string literal

```
String name = "Kevin";
```

By new keyword

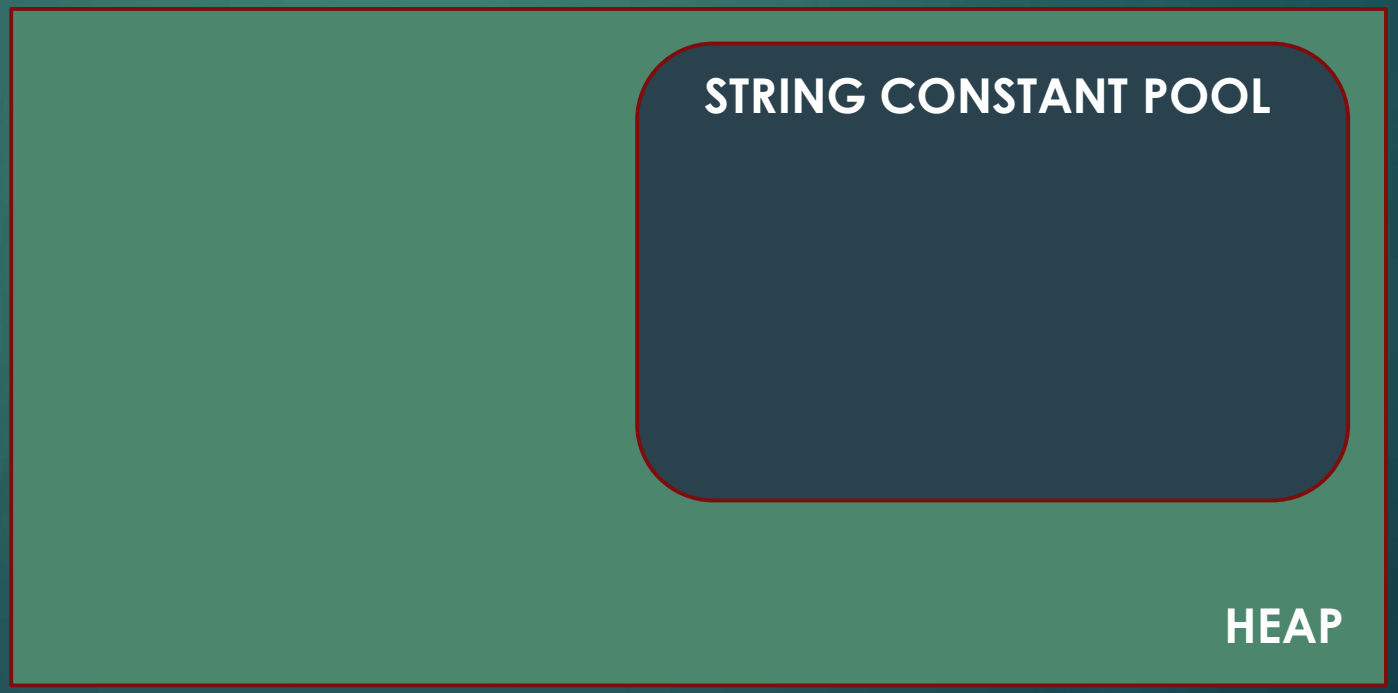
```
String name = new String("Kevin”);
```

Hint! Previously in:
First Steps in Java,
lecture 9,
„Classes and objects"

String constant pool

Since strings are everywhere in Java, they use up a lot of memory. Many strings repeat in the program and to solve this issue it's good idea to reuse common ones. The **string constant pool** (also known as the intern pool), is a location in the JVM, that collects all these strings.

String constant pool is a special memory area in the Heap.



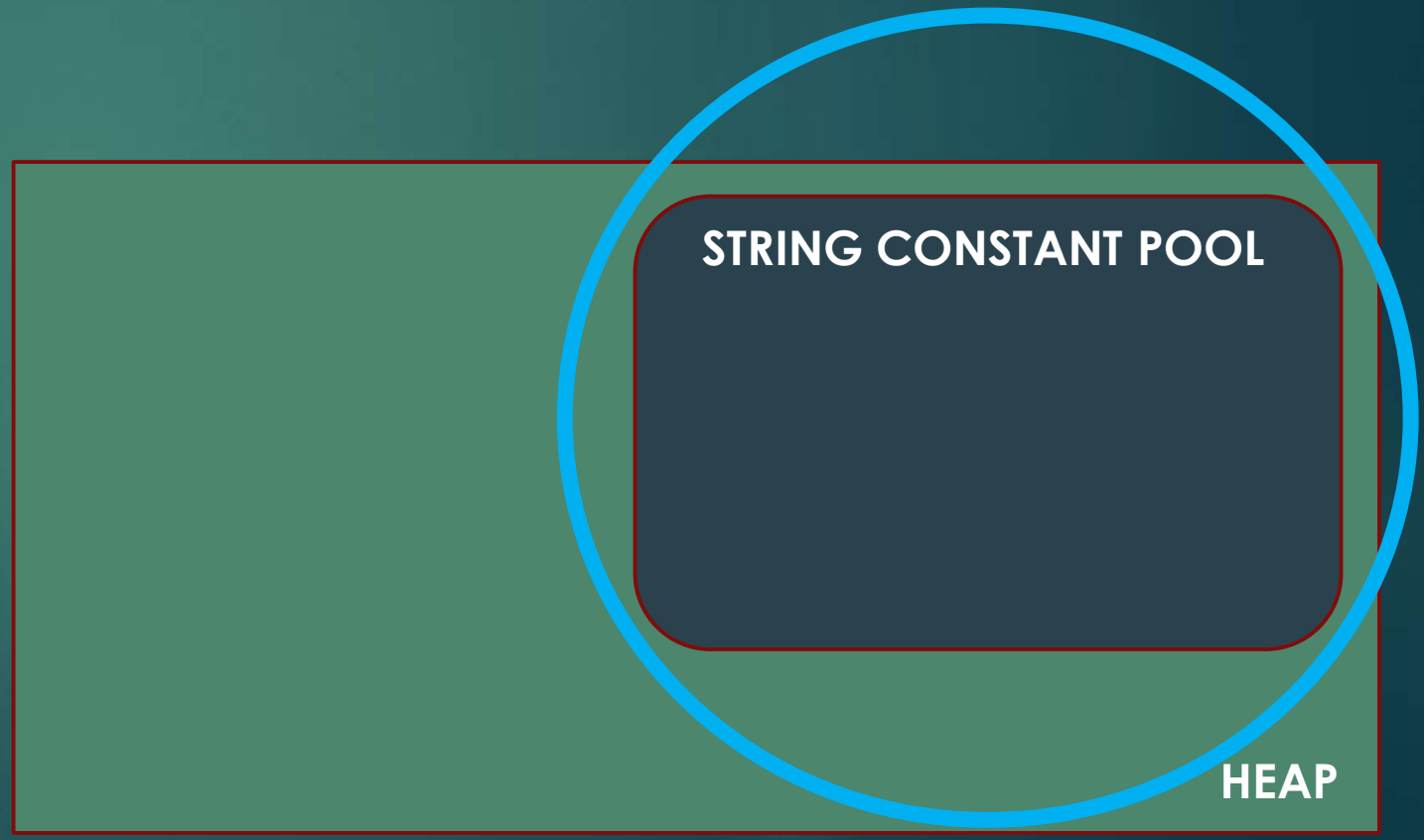
String creation

```
String n1 = "Kevin";
```

reference variable

constant value

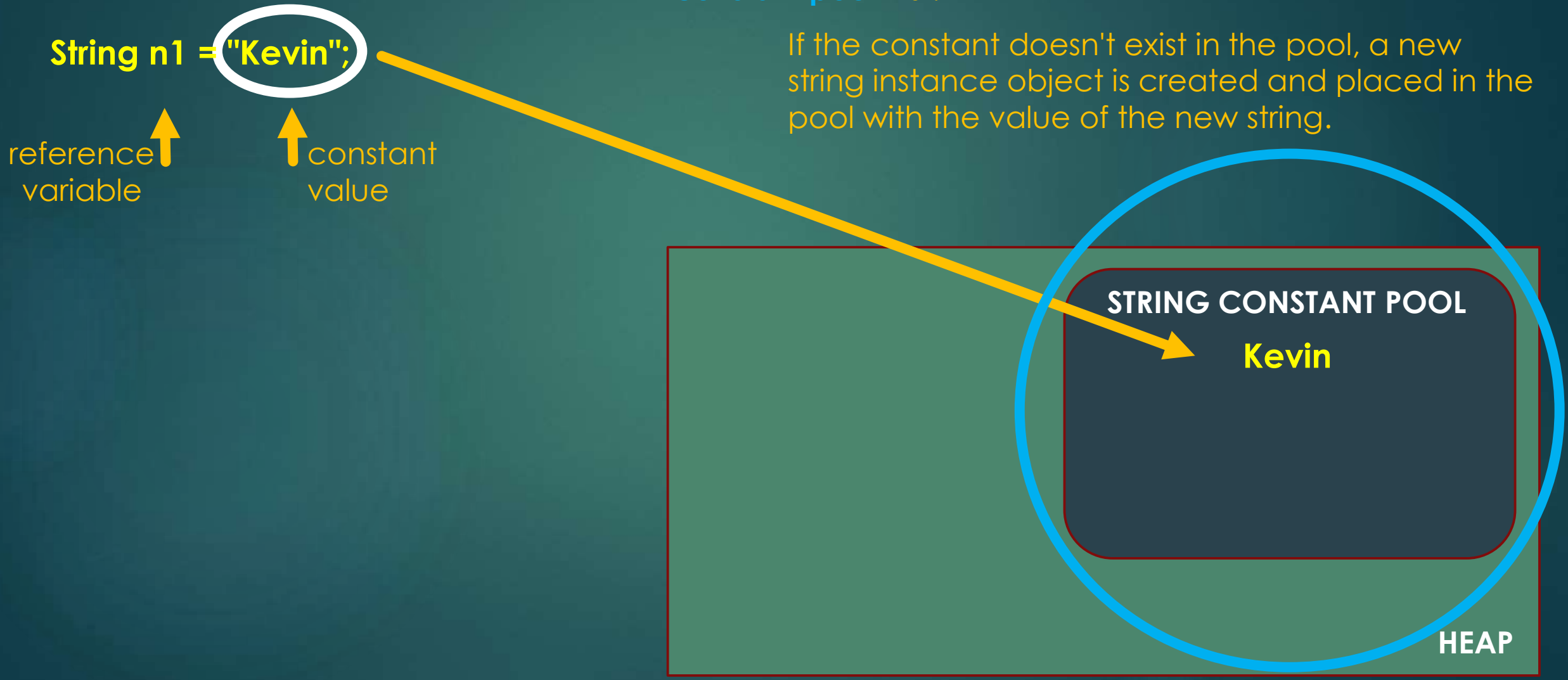
Each time you create a string literal, the JVM checks the **constant value** and the **string constant pool** first.



String creation

Each time you create a string literal, the JVM checks the **constant value** and the **string constant pool** first.

If the constant doesn't exist in the pool, a new string instance object is created and placed in the pool with the value of the new string.



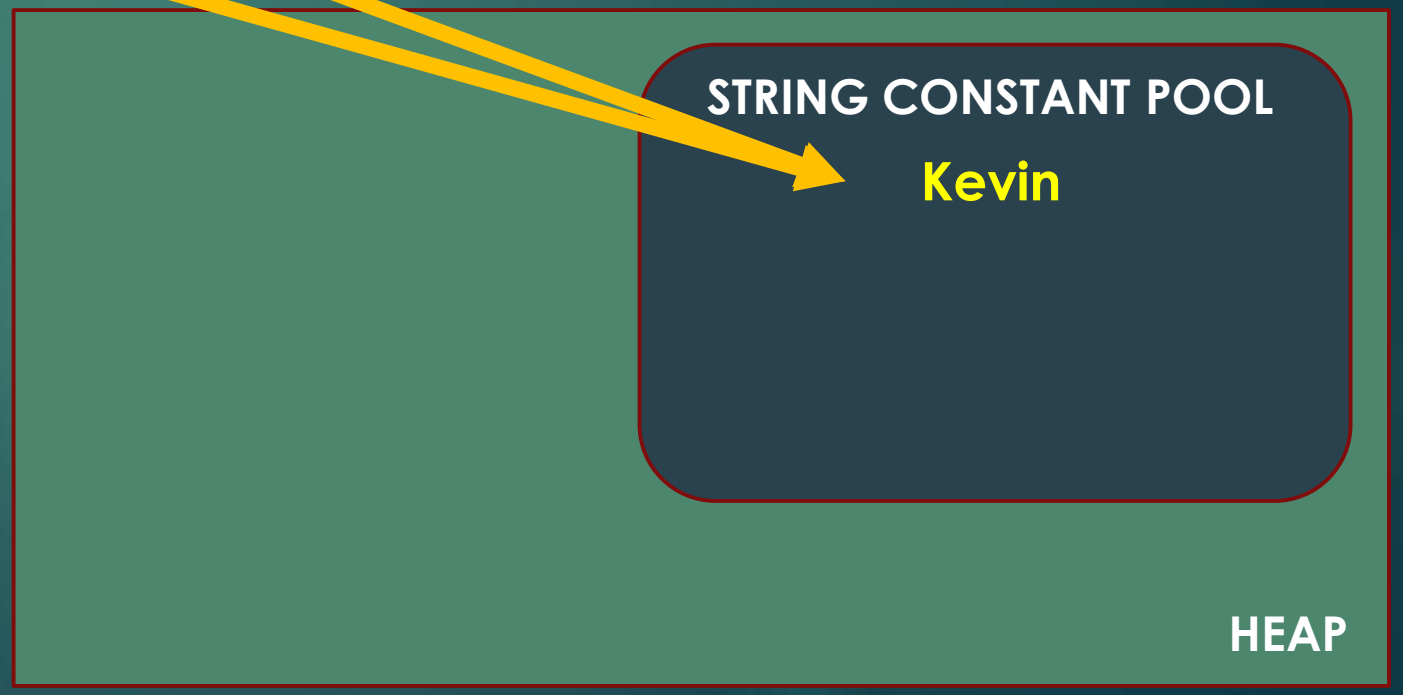
String creation

String n1 = "Kevin";

String n2 = "Kevin";

If the string with the same value already exists in the pool, JVM will not create a new object but will return the reference to the same instance.

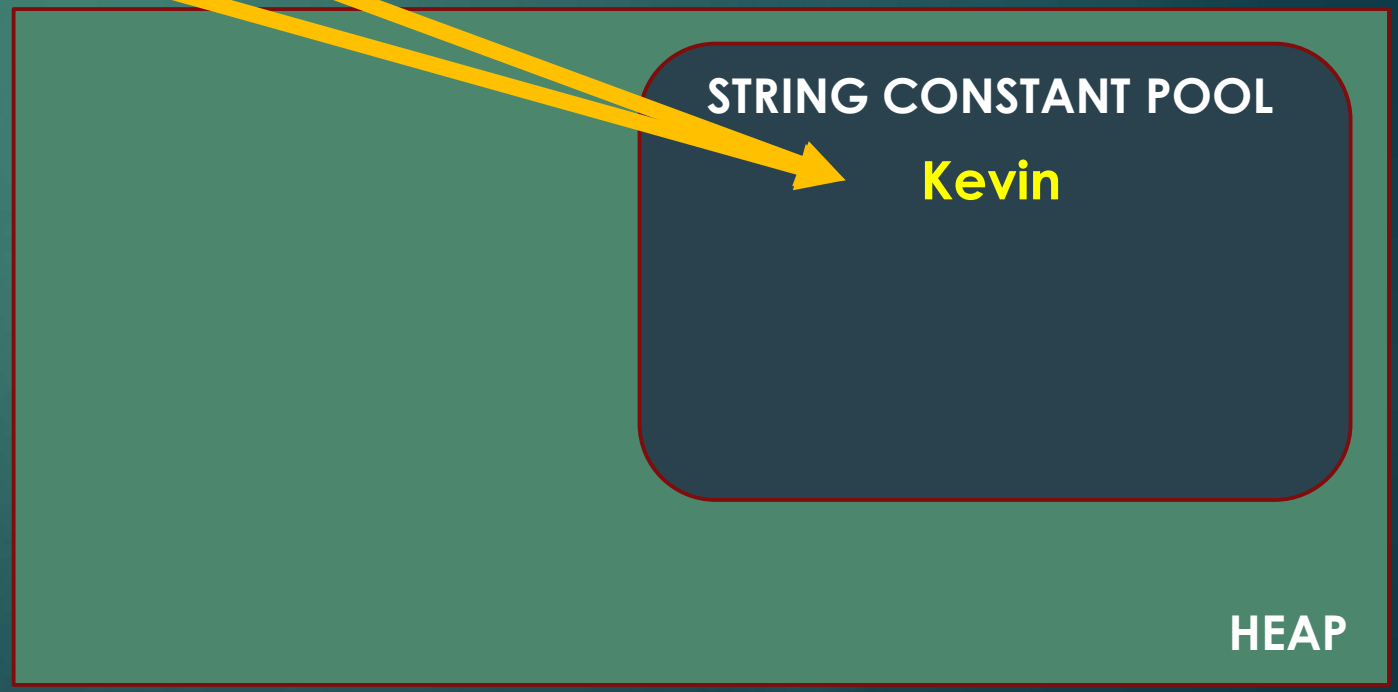
Don't forget! JVM never creates duplicate objects in String Pool.



String creation

This is string creation by string literal

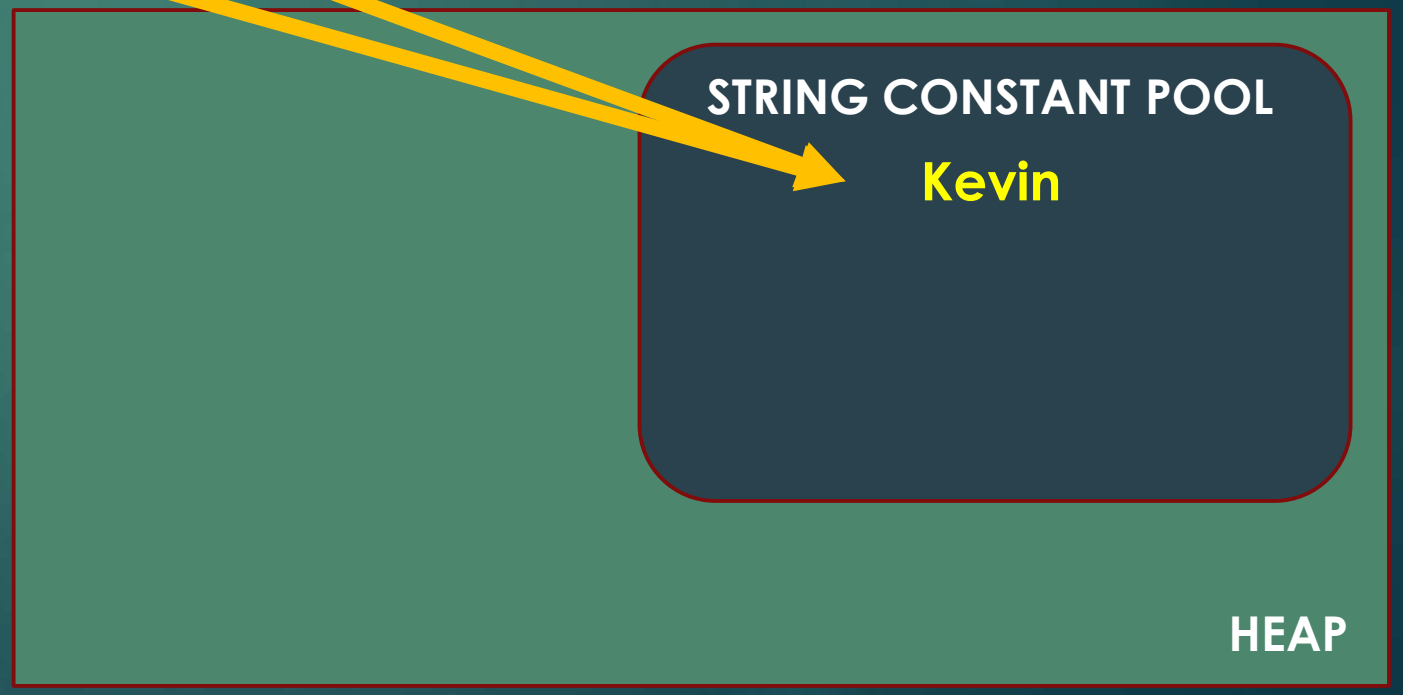
```
String n1 = "Kevin";
String n2 = "Kevin";
```



String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
```

This is string creation by new keyword



String creation

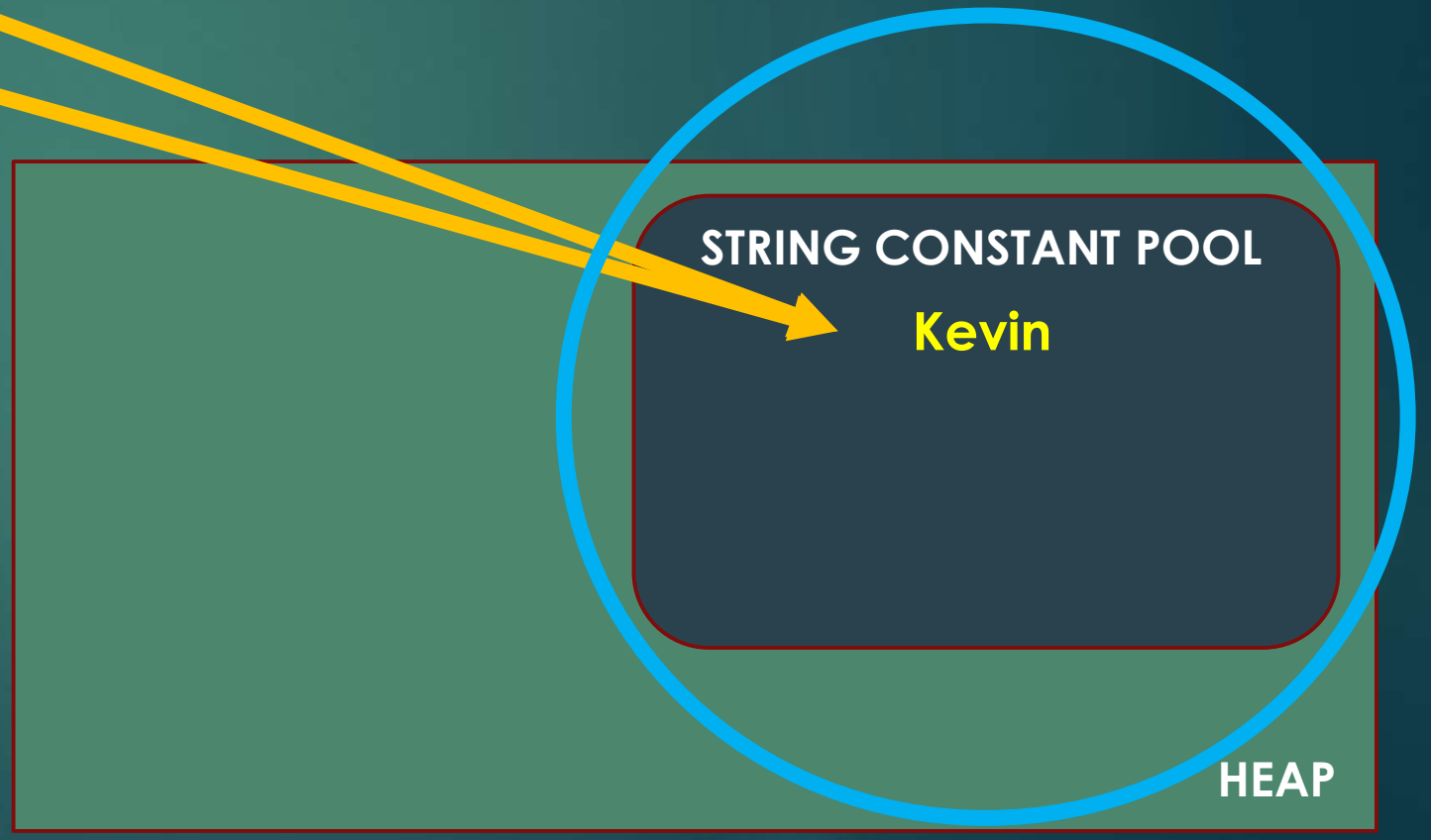
String n1 = "Kevin";

String n2 = "Kevin";

String n3 = new String("Kevin");

The beginning is the same. At compile time the JVM checks the constant value of the new string and the constant pool.

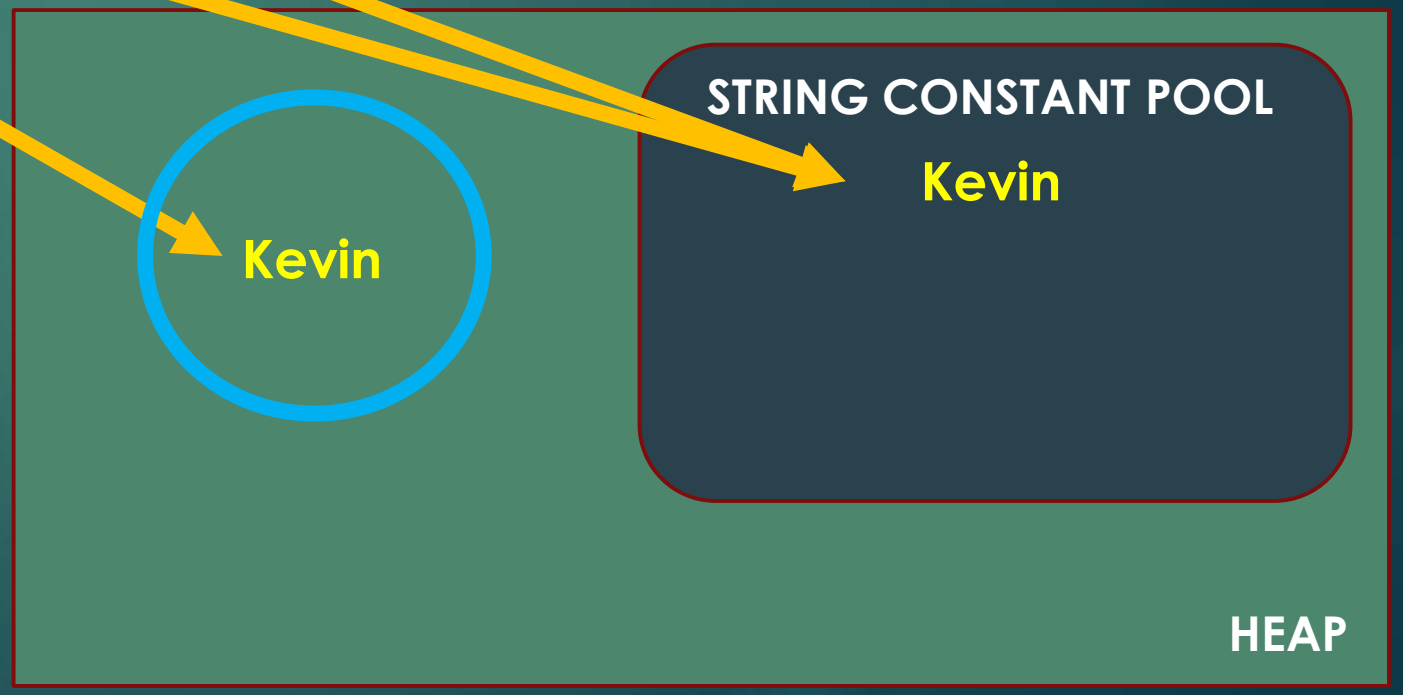
If a string with the same value already exists in the pool (as in this case), JVM will not create a new object there.



String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
```

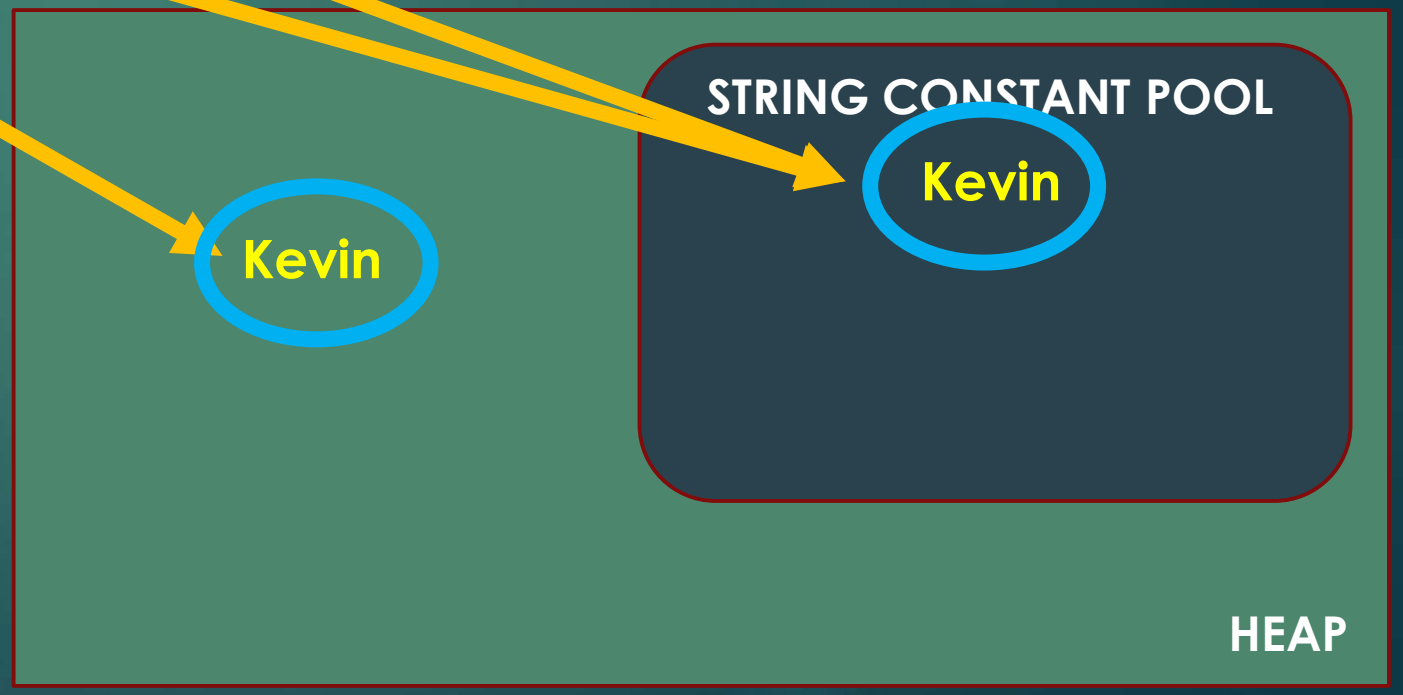
But because of the „**new**" keyword the JVM at runtime creates a **new string object** in the normal (non-pool) heap memory, and the **reference variable** will be pointing to this object.



String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
```

Notice that only the values are the same, but the reference variables are pointing to different objects!



String creation

String n1 = "Kevin";

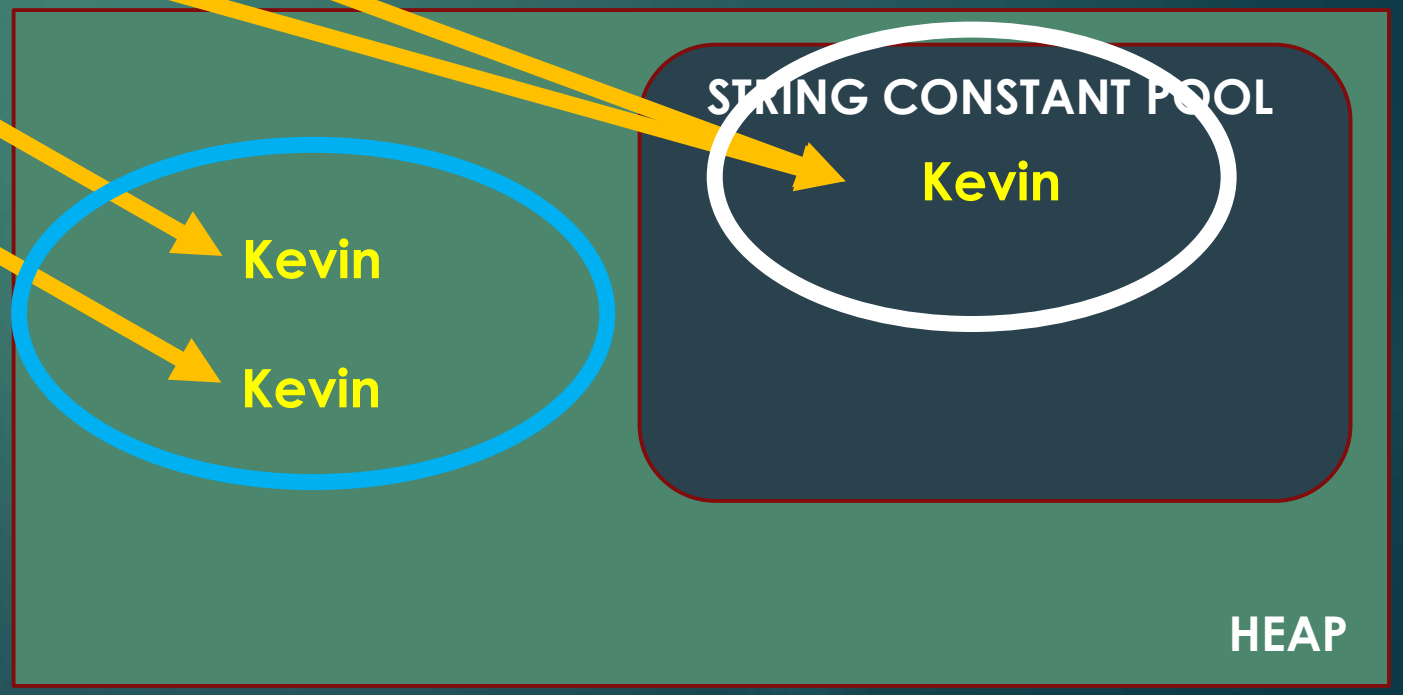
String n2 = "Kevin";

String n3 = new String("Kevin");

String n4 = new String("Kevin");

Because of the „new" keyword the JVM at runtime creates another new string object in the heap memory even though the value is the same.

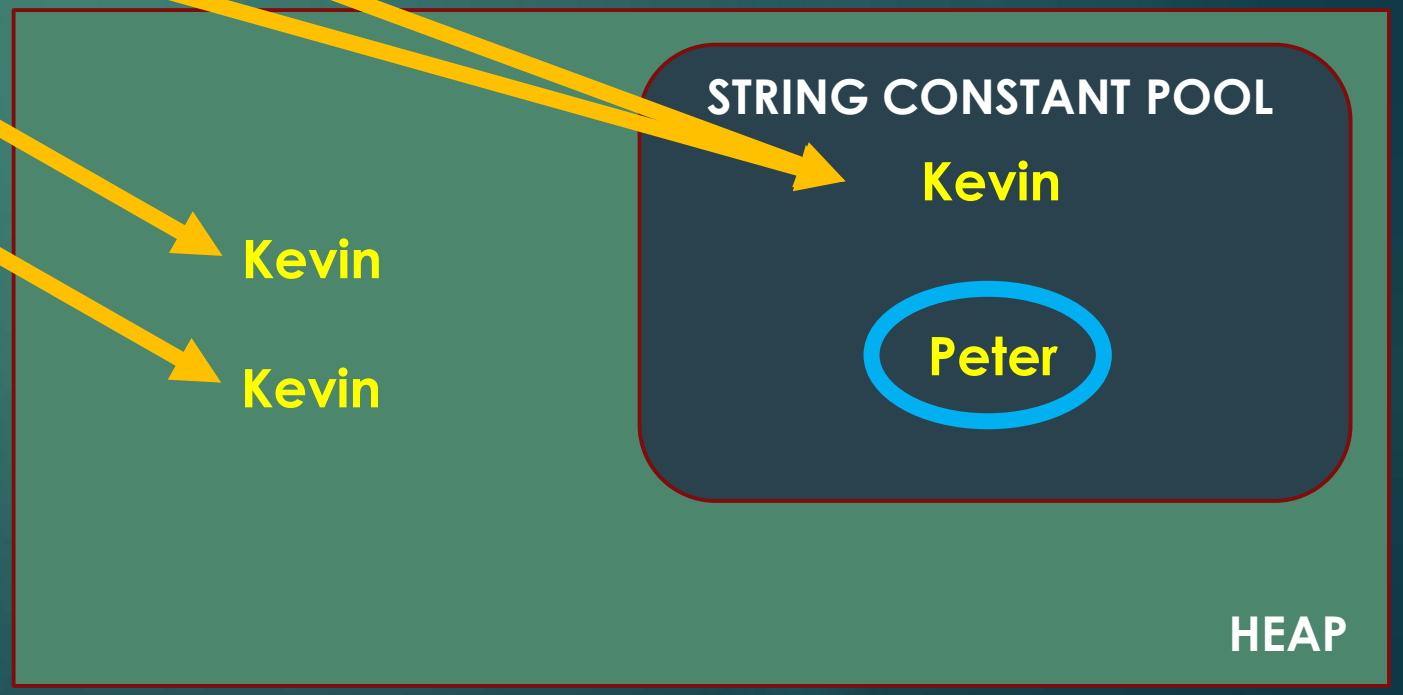
Notice the difference between the literal creation and the creation by new keyword technique.



String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
String n4 = new String("Kevin");
String n5 = new String("Peter");
```

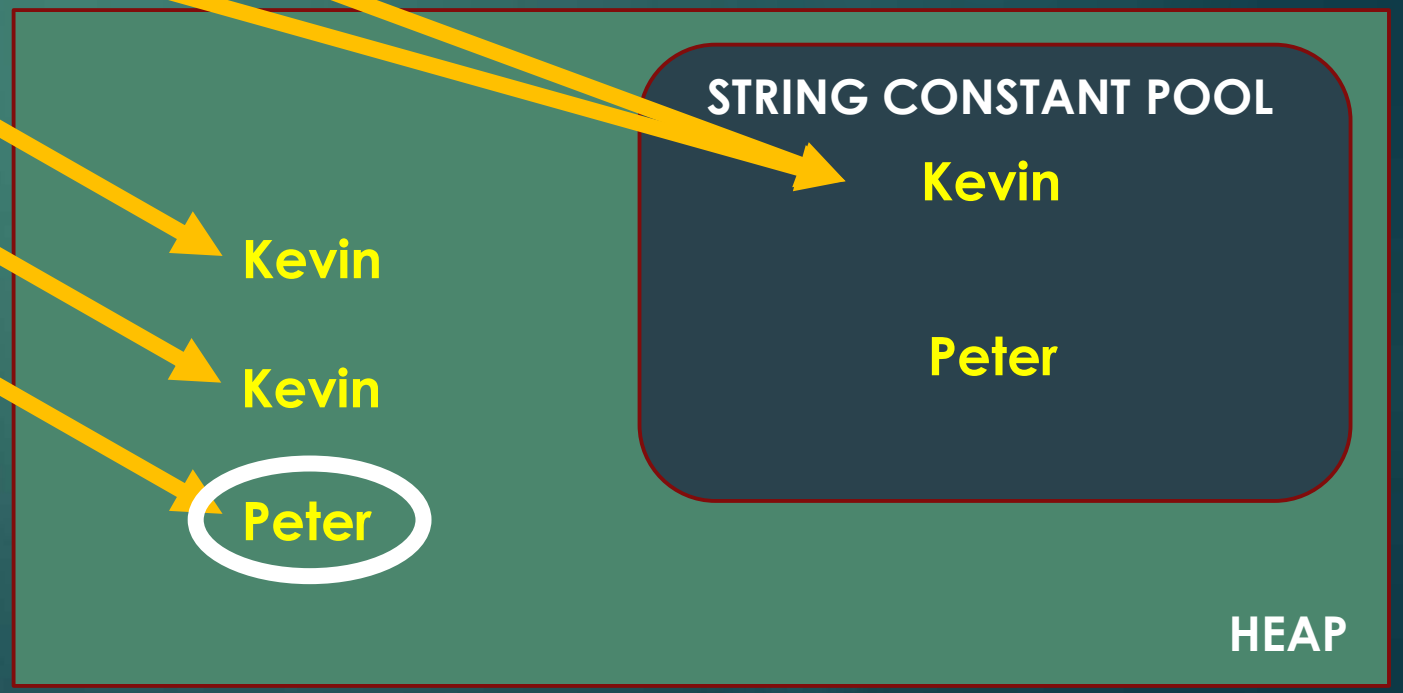
In this case **constant value** of the new string doesn't exist in the pool, so a **new string instance** is created and placed in there.



String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
String n4 = new String("Kevin");
String n5 = new String("Peter");
```

Because of the „new" keyword the JVM will create a **new string object** in the heap memory, and the **reference variable** will be pointing to this object.

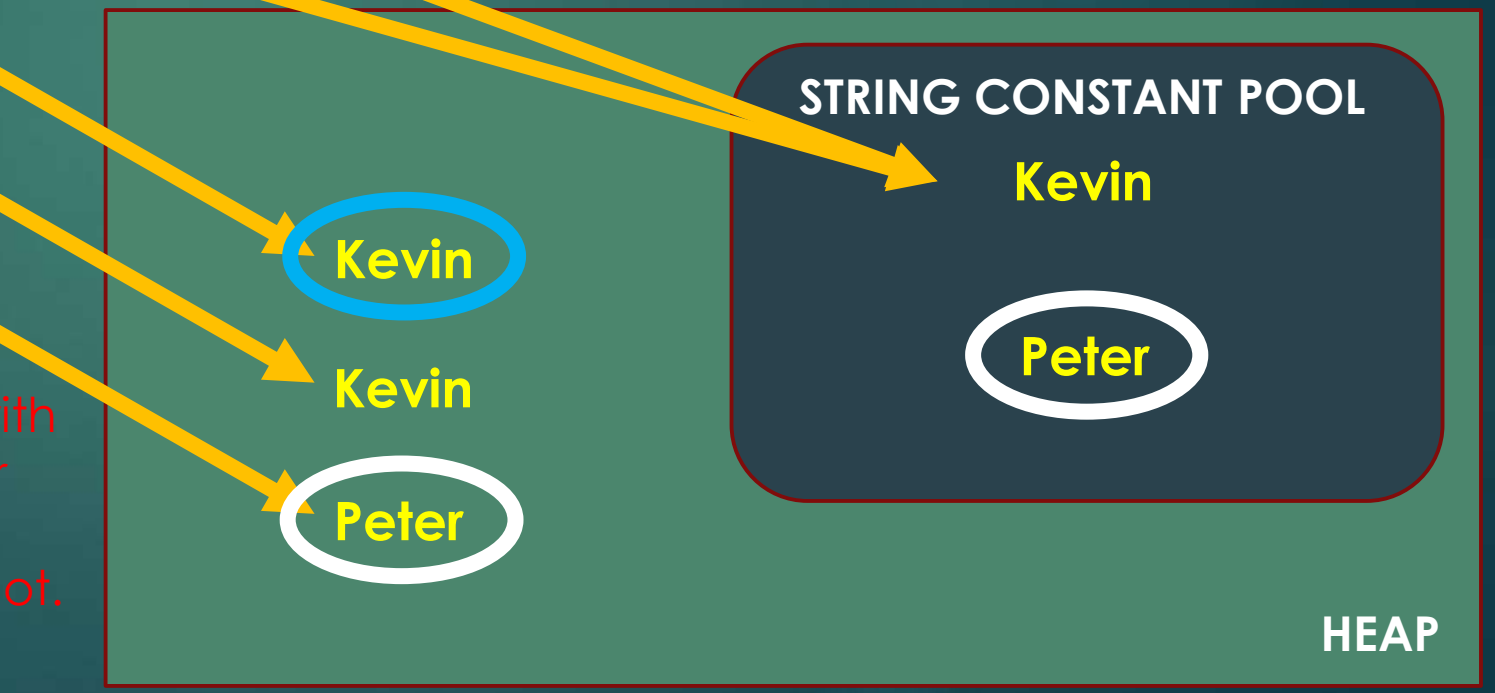


String creation

```
String n1 = "Kevin";
String n2 = "Kevin";

String n3 = new String("Kevin");
String n4 = new String("Kevin");
String n5 = new String("Peter");
```

In the example when the string's value "Kevin" was already in the Pool, there was only one new object created by us in the **Heap**.

In the other example the string with value of "Peter" didn't exist in the Pool, so that's why our process created two objects, one in the **Pool**, and one in the **Heap**.



Notice that when you're creating string with new keyword you will creat one object or two, it depends on whether the value of the string is already in the String pool, or not.

String creation interview question

It is possible that in the future in a java interview, you will be asked a tricky question in the String topic. The question looks simple:

How many string object is getting created in the statement below?

```
String s = new String ("ABC");
```

Many people give the wrong answer that one object is created in the Heap.

But from now on you will know the right answer, that it could be one or two. And it depends on whether the constant value of the new string is already in the string constant pool, or not.

Java Programming: Step by Step from A to Z

MORE ABOUT Strings

String immutability

String's immutability

The String class is immutable, so that once it is created a String object cannot be changed.

The String class has a number of methods, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

String's immutability example

String name = „Alex";

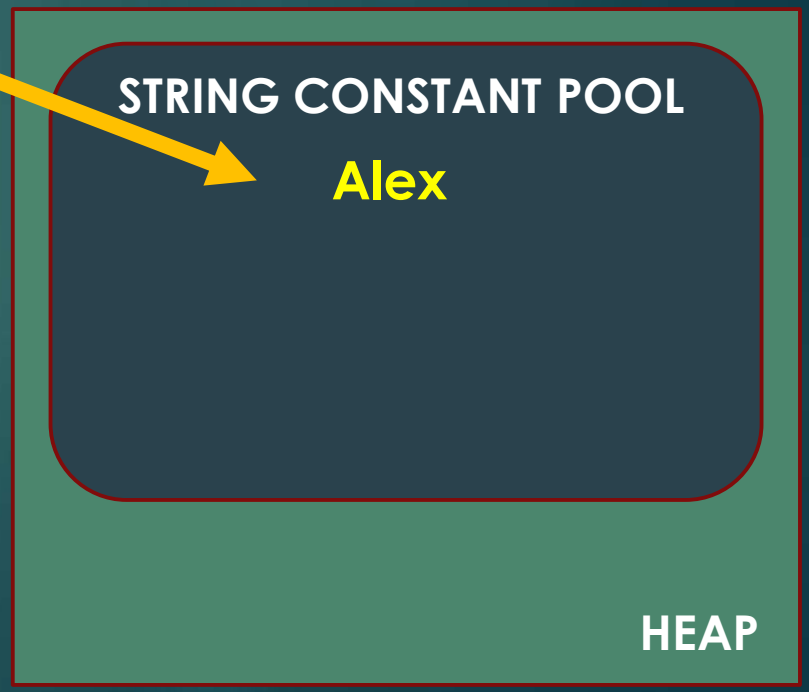
name.concat("ander");

System.out.println(name);

String creation by string literal

concat() method appends the string at the end

?



String's immutability example

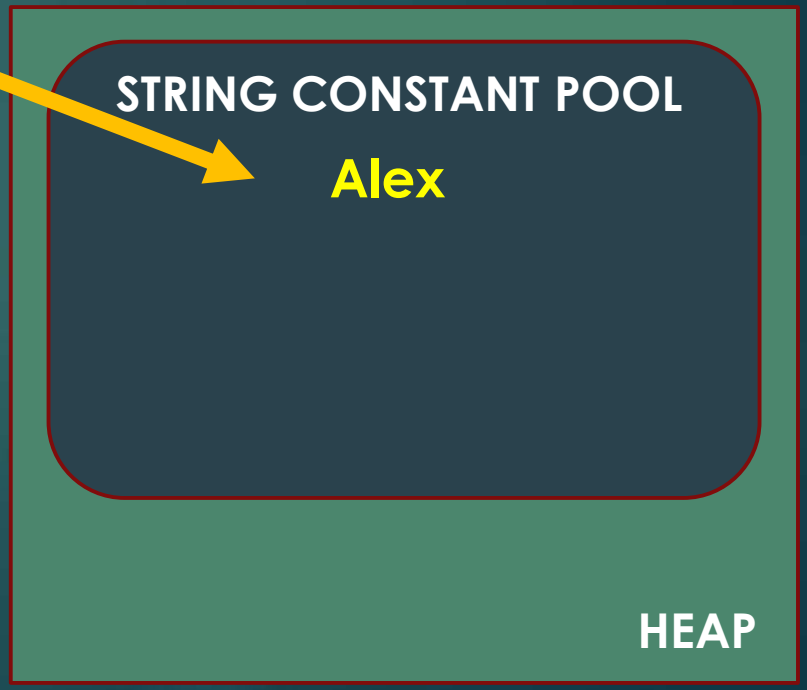
String name = „Alex";

name.concat("ander");

System.out.println(name);

?

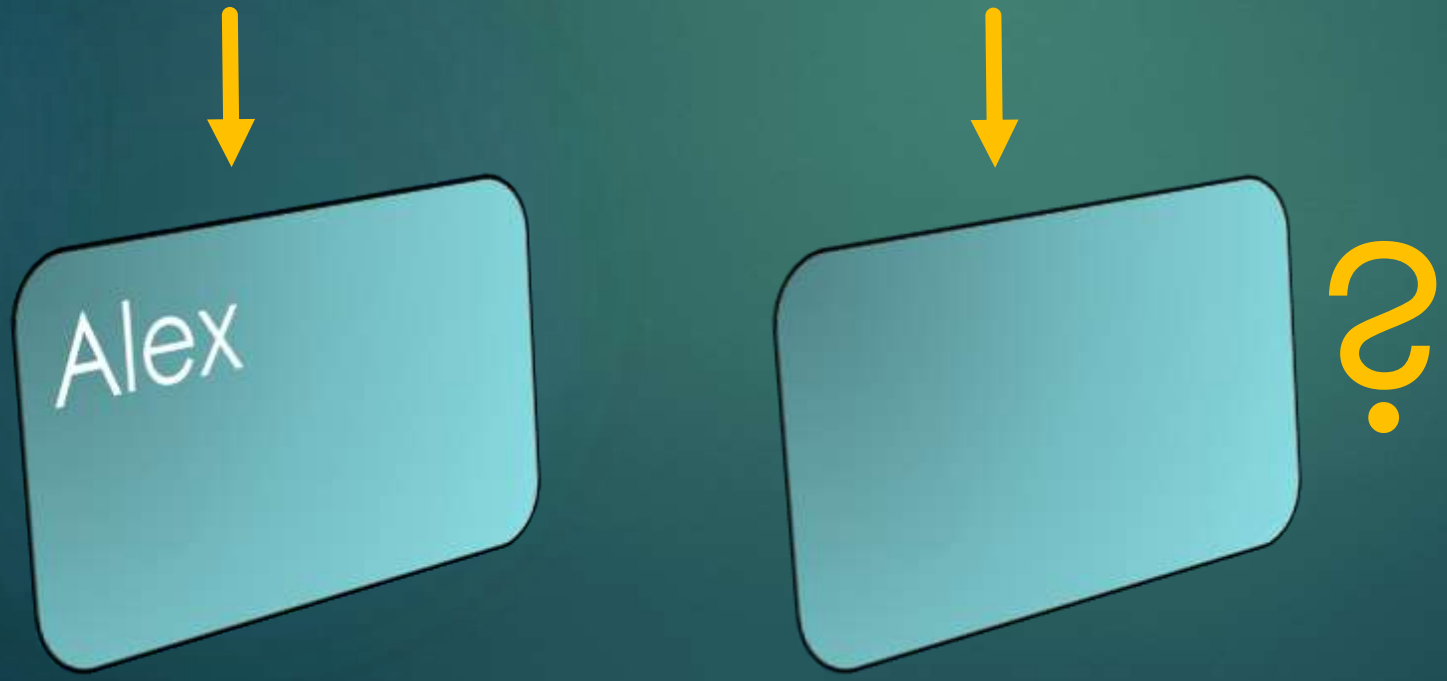
It will print „Alex", not „Alexander" because strings are immutable objects and the „name" reference variable is still pointing to „Alex" in the Pool.



String's immutability example

```
String name = „Alex";

name.concat("ander");

System.out.println(name);
```

```
name = name.concat("ander");

System.out.println(name);
```



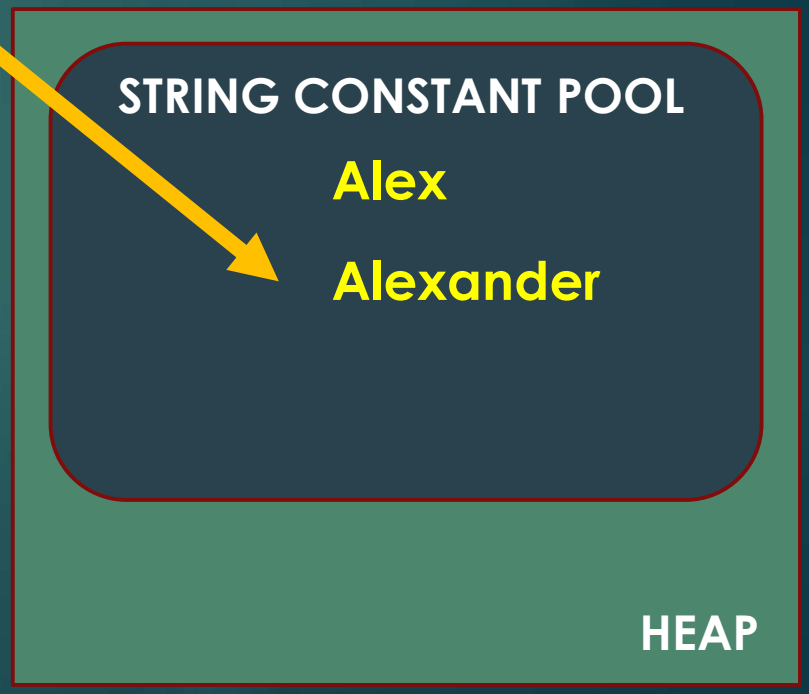
String's immutability example

```
String name = „Alex";

name.concat("ander");

System.out.println(name);
```

This time it will print „Alexander", because JVM creates a new string by string literal in the Pool with this new concatenated value, and the „name" reference variable points to this. Please notice that the „Alex" object is still not modified because it's immutable.

```
name = name.concat("ander");

System.out.println(name);
```

?

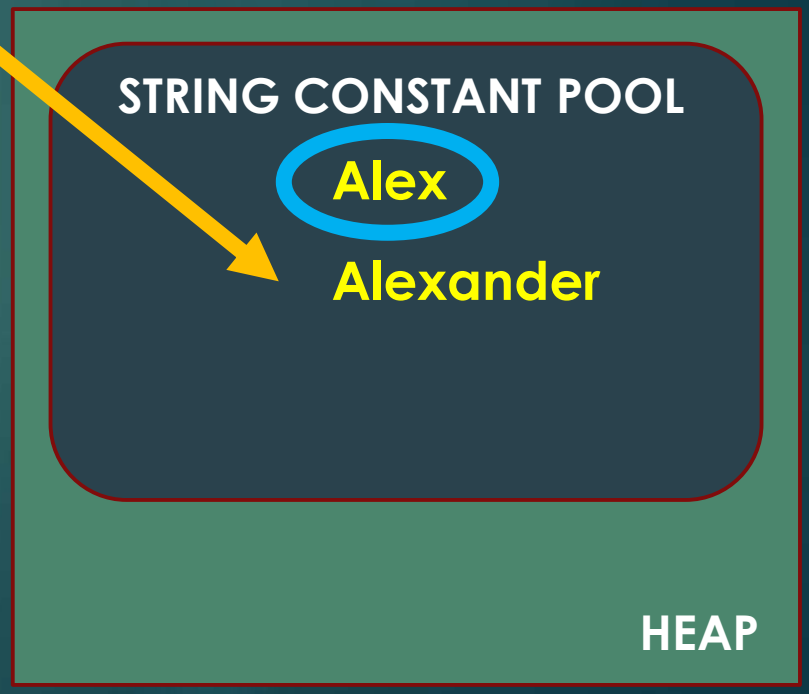


String's immutability example

Because the new String object "Alexander" is assigned to **name**, the old **"Alex" object** has no more reference variable and it becomes eligible for garbage collection.

```
String name = „Alex";

name.concat("ander");

System.out.println(name);
```

```
name = name.concat("ander");

System.out.println(name);
```

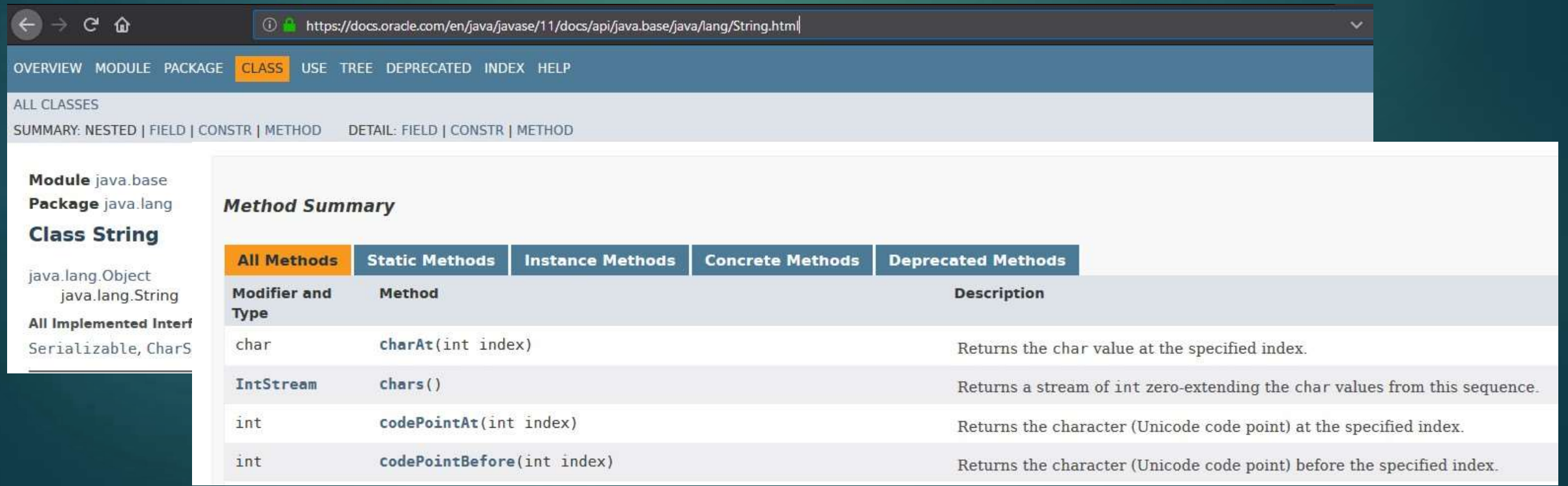


Methods of the String class

In the next lexture we will examine a few main methods of the String class, but it's useful to know, that there are lots of methods for String class.

You can look around at any time on the website of Oracle for these.

Java API Documentation → **https://docs.oracle.com/en/java/**



Java Programming: Step by Step from A to Z

StringBuffer and StringBuilder

String Buffer and String Builder overview

The StringBuffer and StringBuilder classes are used when there is a need to make a lot of modifications to Strings of characters.

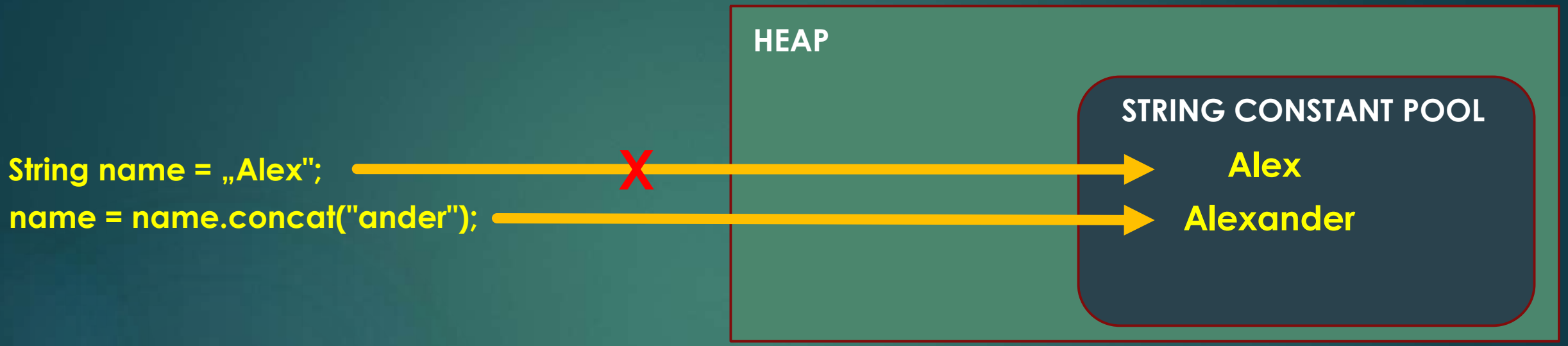
Unlike Strings, objects of type **StringBuffer** and **StringBuilder** can be modified over and over again without leaving behind a lot of new unused objects.

The only difference between **StringBuffer** and **StringBuilder** is that StringBuffer is thread safe (synchronized) whereas **StringBuilder** is not. It is recommended to use **StringBuilder** whenever possible because it is faster than **StringBuffer**.

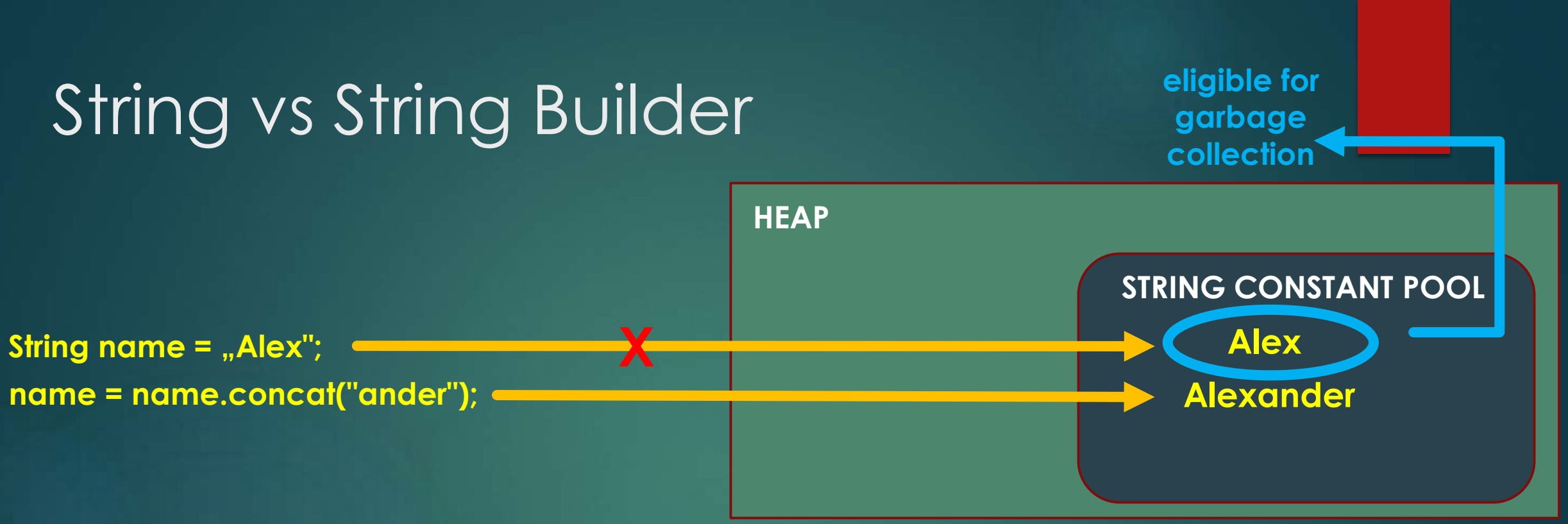
About
thread safety

In an appliaction there can be more threads. In this case thread-safe means that at the same time only one thread can access a StringBuffer object's synchronized code. About threads we will learn later at the Multithreading topic.

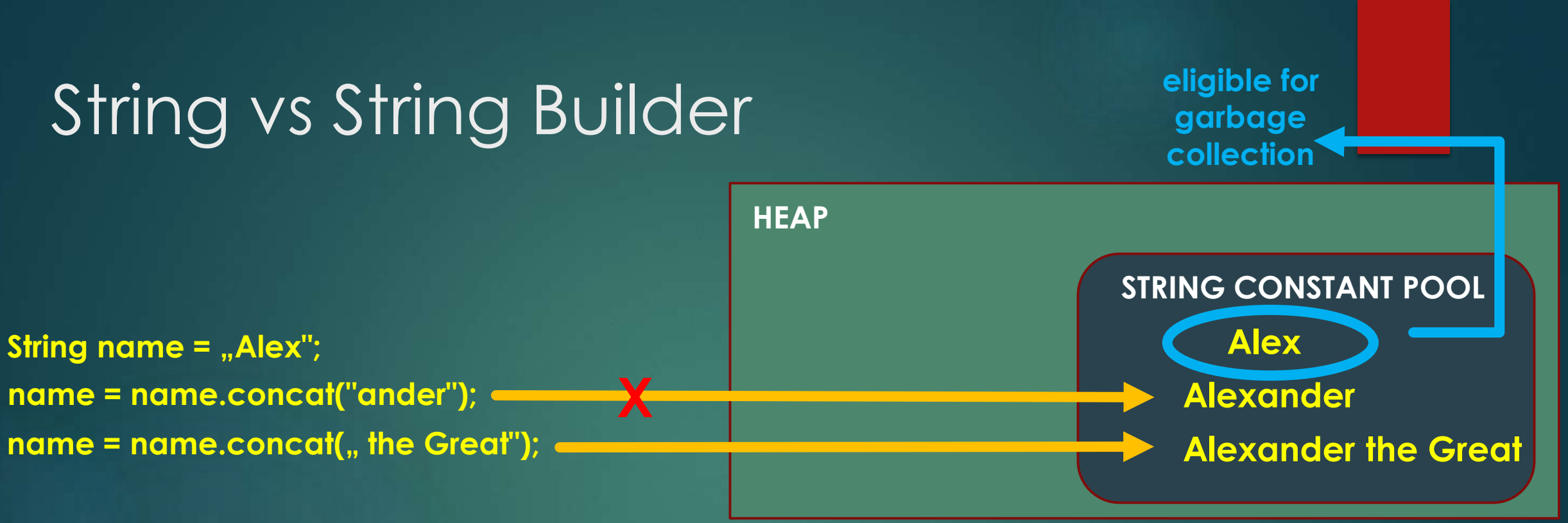
String vs String Builder



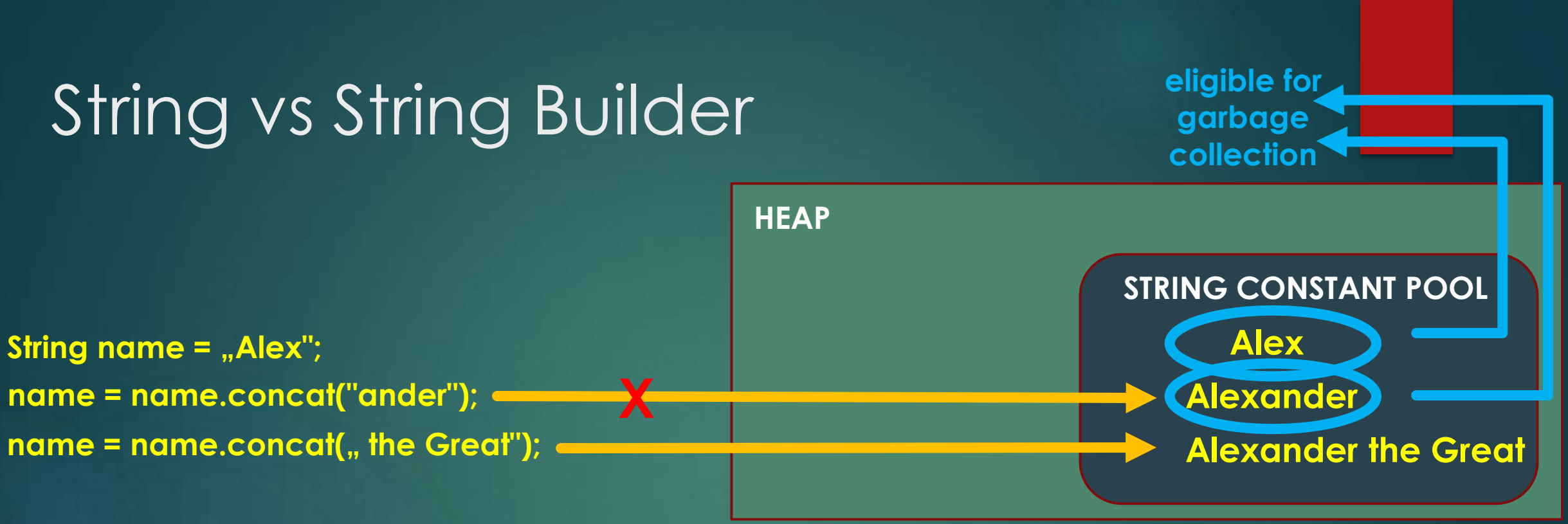
String vs String Builder



String vs String Builder

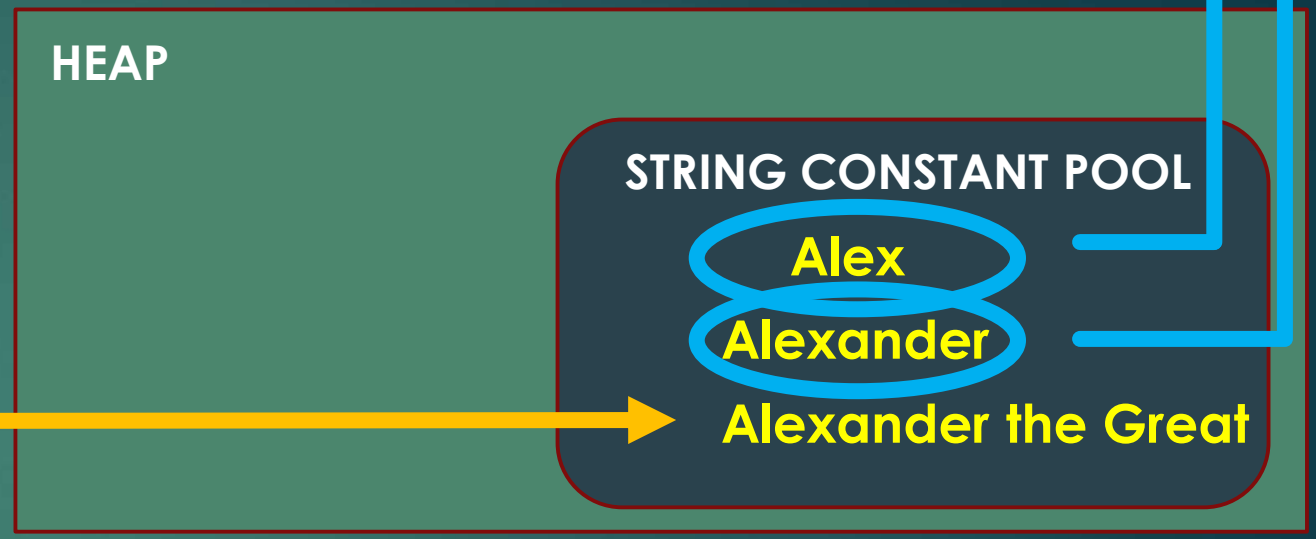


String vs String Builder

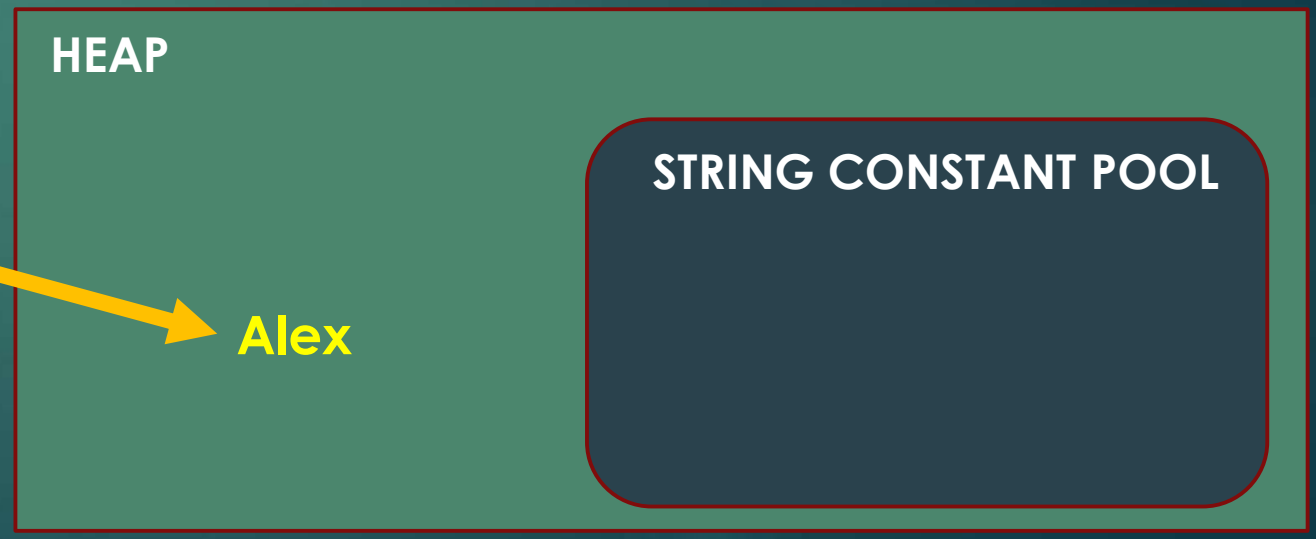


String vs String Builder

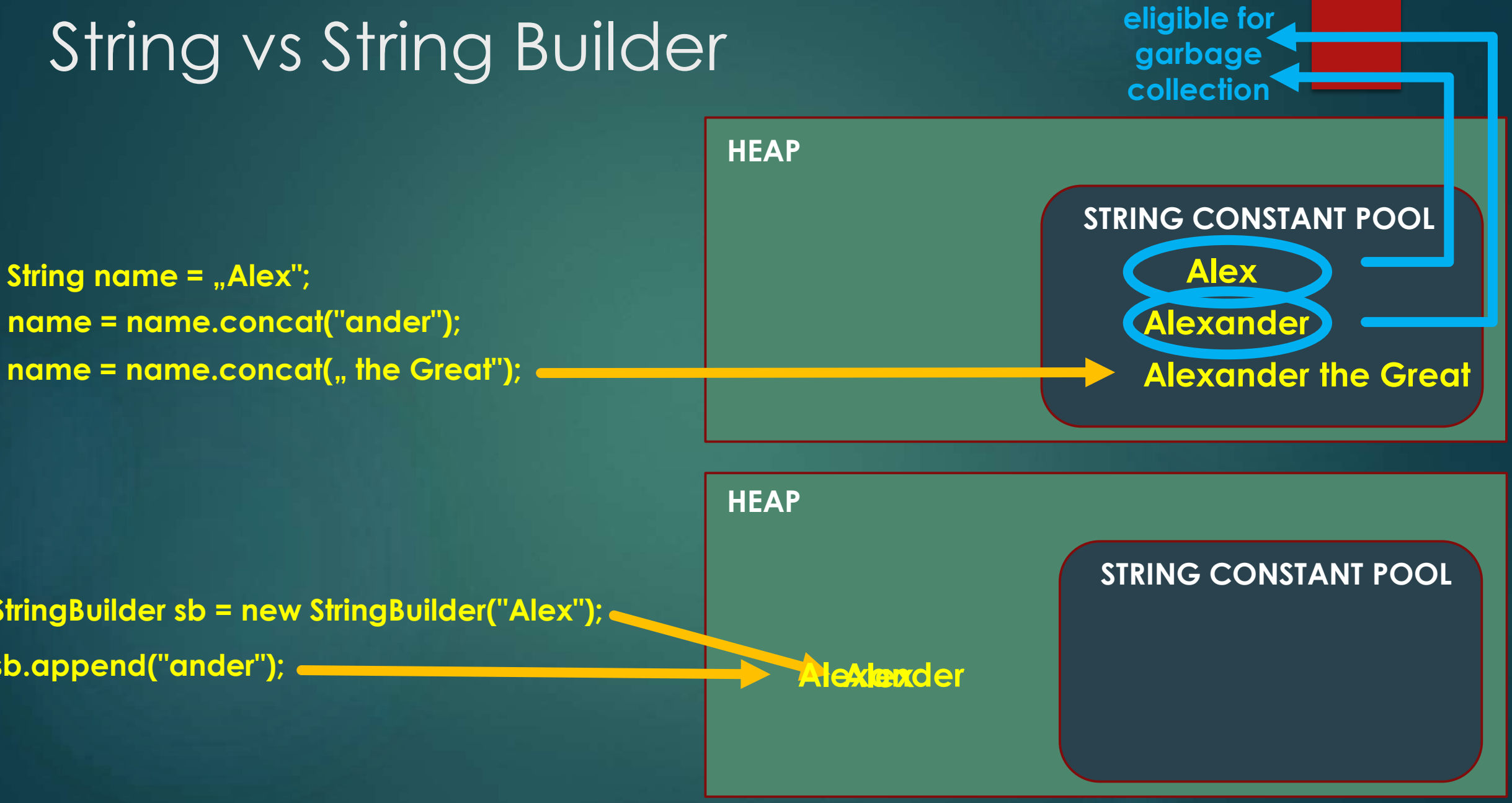
```
String name = „Alex";
name = name.concat("ander");
name = name.concat(„ the Great");
```



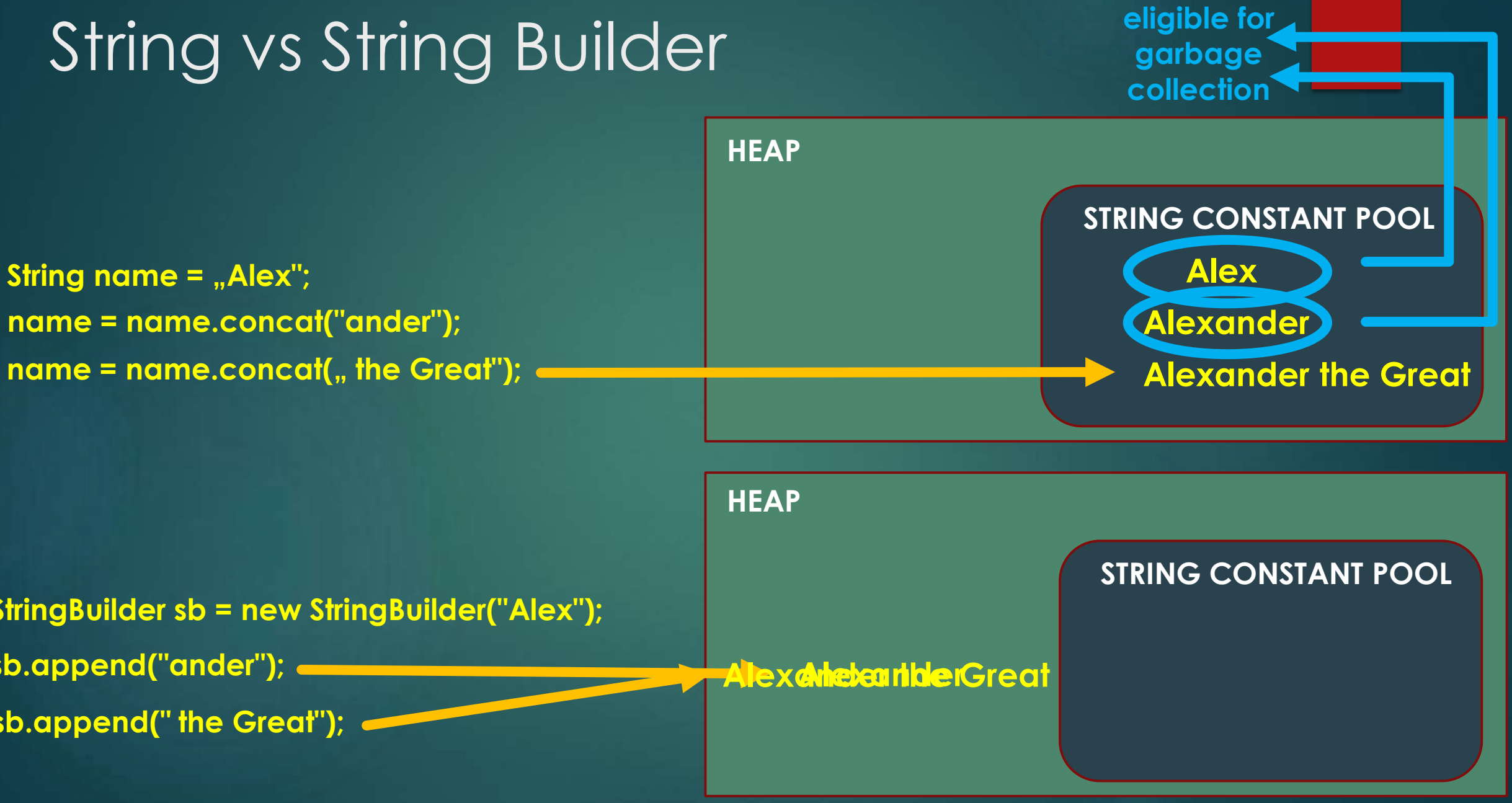

```
StringBuilder sb = new StringBuilder("Alex");
```



String vs String Builder



String vs String Builder



String vs String Buffer vs String Builder conclusion

	String	String Buffer	String Builder
Immutable	Yes	No	No
Speed	Fast	Slow	Fast
Store	String Pool	Heap	Heap
Thread Safe	Yes	Yes	No

- objects of String are immutable, and objects of **StringBuffer** and **StringBuilder** are mutable.
- **StringBuffer** and **StringBuilder** are similar, but **StringBuilder** is faster and preferred over **StringBuffer** for single threaded programs. If thread safety is needed, then **StringBuffer** is used.

Java Programming: Step by Step from A to Z

Java Enums

Enum overview

Enumerations are representing a group of named constants in a programming language. An *enum type* is a special data type, an enum is special "class" that represents a group of constants (unchangeable variables, like final variables). To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Because they are constants, the field names of the enum types are in uppercase letters.

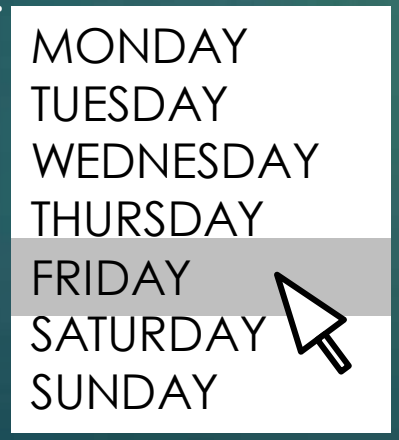
Don't forget! Enum's unchangeable variables must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week (MONDAY, TUESDAY, etc).

Enum overview

Enum code example:

```
public enum Day {
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Day d = Day.



The **d** field is of type Day, therefore the only values that it can be assigned are those defined by this enumeration. (MONDAY, TUESDAY, etc)

Enum overview

Enum code example:

```
public enum Day {
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Day d = Day.FRIDAY;

MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY

The **d** field is of type Day, therefore the only values that it can be assigned are those defined by this enumeration. (MONDAY, TUESDAY, etc)

We can use E nt things too (for example with switch statement). We will see these in our next lectu

Enum overview

Enums vs Classes

An enum can have attributes and methods. Just like a class.

Contrary to the class, enum constants can be public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it can not extend other classes (but it can implement interfaces).

When To Use Enums?

You should use enum types any time you need to represent a fixed set of constants.
Use enums when you have values that are not going to change, like months, days, colors, deck of cards, etc.

Java Programming: Step by Step from A to Z

Dates and Times

Dates & Times Overview

In **Java** there are two ways to represent and manipulate date and time.

The old way when we use the following classic Date and Calendar APIs (before Java **8** this was the only possibility): java.util.Date, java.util.Calendar, java.util.GregorianCalendar

and

The new way. In Java **8** a new series of date and time APIs are created in the new java.time package.

We will learn the new way, but we'll look into the classic, old way too, so your knowledge will be complete and also because you can run into this in many older applications. This will help you in the future if you need to read these older codes.

Dates & Times Overview

Main differences between the two solutions, the disadvantages of the old way:

- not thread safe : unlike String, Date is mutable, so it's not thread safe. New solution is thread safe.

- less operations : in old API there are only a few date operations but the new API provides us with many of these.

- in the old solution the Calendar class is not very user-friendly or intuitive in Java. (e.g.: Month indexes were **0** based instead of **1** based, which was confusing. Etc.)

Backward Compatibility

A toInstant() method is added to the original Date and Calendar objects and can be used to convert them to the new Date/Time API.

Conclusion

Use new java.time API when writing new code or refactoring the old one.

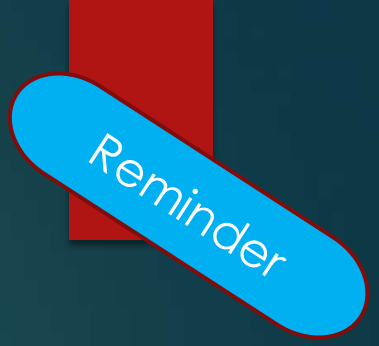
Java Programming: Step by Step from A to Z

MORE ABOUT

Conditional Statements

The nested if statement

Conditional Statements overview



As we learned earlier, in programming it's often required to execute a certain section of code based upon whether the specified condition is true or false (which is known only during the run time). For such cases, conditional statements are used.

- if statement
- if-else statement
- if-else-if statement
- Switch statement

Conditional Statements overview

Reminder

if statement

```
if(condition){
      //code executed if condition is true
}
```

if-else statement

```
if(condition){
      //code executed if condition is true
}else{
      //code executed if condition is false
}
```

If-else-if statement

```
if(condition1){
      //code executed if condition1 is true
}else if(condition2){
      //code executed if condition1 is false and
                        condition2 is true
}
else if(condition3){
      //code executed if condition1 is false and
                        if condition2 is false and
                          condition3 is true
}
...
else{
      //code executed if all the conditions are false
}
```

Hint! Previously in:
First Steps in Java,
lecture 20,
„Decisions: the 'if' statement"

Conditional Statements overview

Reminder

Switch statement

```
switch(expression){
case value1:
 //code executed if value1 matched the expression;
 break;  //optional
case value2:
 //code executed if value2 matched the expression;
 break;  //optional
.....

default:
 code executed if none of the cases are matched;
}
```

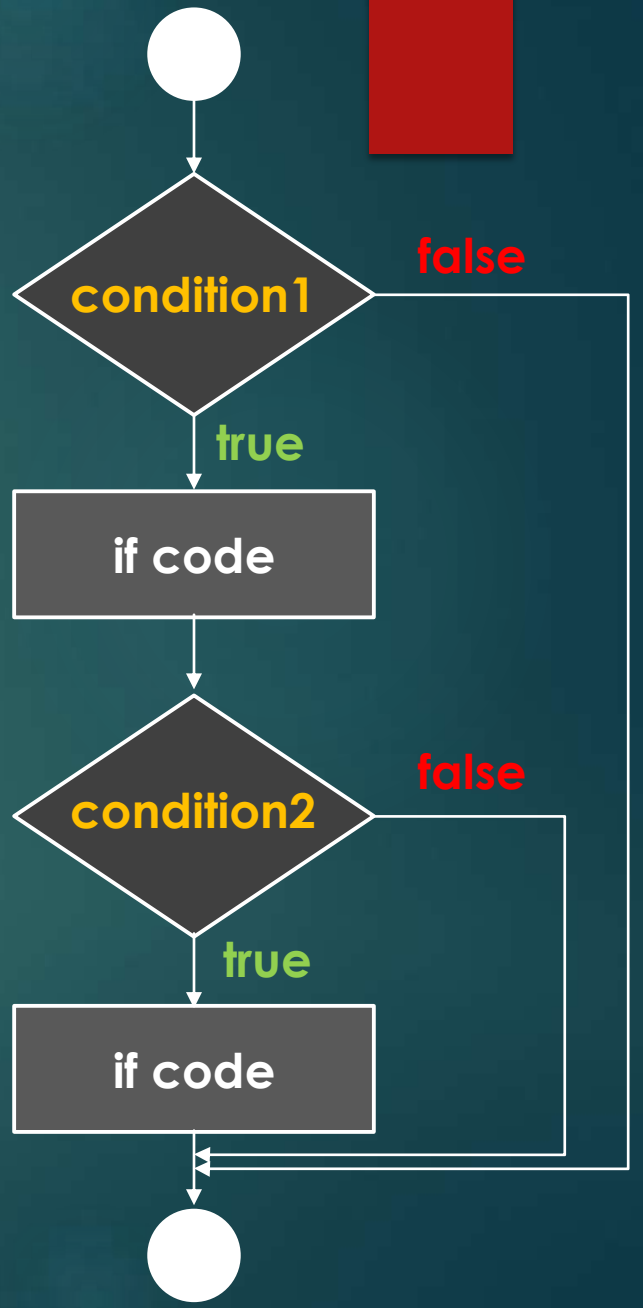
Hint! Previously in: First Steps in Java, lecture 20, „Switch"

Nested if Statements

There is a possibility to nest if-else statements which means you can use one if or else if statement inside another if or else if statement. In other words the nested if statement means an if statement inside an if statement.

Nested if statement

```
if(condition1){
    //code executed if condition1 is true
        if(condition2){
            //code executed if condition1 is true and
                                    condition2 is true
    }
}
```



Nested if Statements

Can we make a triple nested statement? The answer is yes we can. But don't forget that you can often use logical operators as a simpler solution to avoid too much nesting to increase the code readability.

Nested if statement

```
if(condition1){
    //code executed if condition1 is true
        if(condition2){
            //code executed if condition1 is true and
                                    condition2 is true
                if(condition3){
                    //code executed if condition1 is true and
                                                condition2 is true
                                                condition3 is true
        }
    }
}
```

Java Programming: Step by Step from A to Z

MORE ABOUT Loops

The nested loop

Loops overview

Reminder

A loop statement allows us to execute a statement or group of statements multiple times. It repeatedly loops until a particular condition is satisfied.

As we have seen before there are **three types of loops** and **two types of loop control statements** in java.

- for loop (and for-each loop)
- while loop
- do while loop

- break statement
- continue statement

Loops overview

Reminder

for loop

```
for (int i = 0; i < 4; i++){
        System.out.println(i);
}
```

for-each loop

```
String[] colours = {"red", "green", "blue ", " yellow"};
for (String s : colours){
        System.out.println(s);
}
```

Hint! Previously in:
First Steps in Java,
lecture 17,
„For loop"

Loops overview

Reminder

while loop

```
int i=0;
while (i <= 3){
        System.out.println(i);
        i++;
}
```

do while loop

```
int i=0;
do {
        i++;
        System.out.println(i);
}
while (i>100);  //false
```

Hint! Previously in:
First Steps in Java,
lecture 18,
„While loop"

Loops overview

Reminder

break

```
for (int i = 0; i < 10; i++){
    if (i == 4) {
        break;
    }
    System.out.println(i);
}
```

continue

```
for (int i = 0; i < 5; i++){
    if (i == 2) {
        continue;
    }
    System.out.println(i);
}
```

Hint! Previously in:
First Steps in Java,
lecture 19,
„Break and continue"

Nested loops

The placing of one loop inside the body of another loop is called nesting. In other words if a loop exists inside the body of another loop, it's called nested loop. In this case the outer loop takes control of the number of complete repetitions of the inner loop.

```
for (int i = 0; i < 5; i++){
    for (int j = 0; j < i; j++){
        System.out.println(*);
    }
    System.out.println(); //for new line
}
```

Nested for loop

Nested for-each loop

```
String[] arrayString = {"up", "down"};
int[] arrayNum = {1, 2};
for (String s : arrayString){
    for (Integer i : arrayNum){
        System.out.println(s + " " + i);
    }
}
```

Optional Labels for loops

We can have a name for each Java for loop. To do so, we use label before the for loop. It is useful if we have a nested for loop so that we can break/continue the exact loop we want. We will see example for the usage of labels and for nested loops in the next lecture.

Nested for loop with labels

```
OUTER_LOOP:  for (int i = 0; i < 5; i++){
    INNER_LOOP: for (int j = 0; j < i; i++){
                    System.out.println(*);
    }
    System.out.println(); //for new line
}
```

Java Programming: Step by Step from A to Z

MORE ABOUT Arrays

Array Methods

Arrays overview

Reminder

As we learned earlier an array is a container object that stores a fixed number of elements from the same data type. Each element can be referred to by an index. Arrays are zero based and the index of the first element is zero.

Creating (one-dimensional) Arrays

```
String[] names = {"Adam", "Kevin", "Joe"};
```

or

```
String[] names = new String[3];
names[0] = "Adam";
names[1] = "Kevin";
names[2] = "Joe";
```

Loop through the Array with for-each loop

```
for (String n : names) {
  System.out.println(n);
}
```

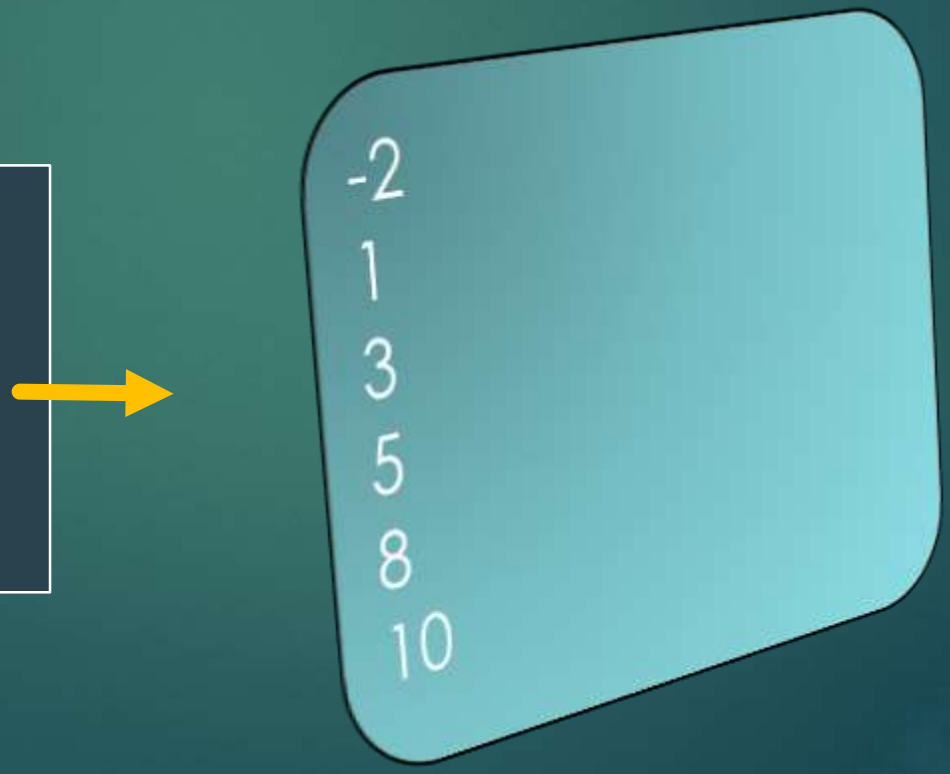
Hint! Previously in: First Steps in Java, lecture 17, „Arrays"

Array methods

Reminder

The Arrays class contains various methods for working with arrays. These methods can be used for modifying, sorting, copying, or searching data. We already learned about one of these methods earlier and that was the sort method.

```
int[] numbers = {1, 5, 3, -2, 10, 8};

Arrays.sort(numbers);

for(Integer s : numbers){
    System.out.println(s);
}
```



Hint! Previously in: First Steps in Java, lecture 18, „Sorting arrays"

Array methods

In the next lectures, among other things, we will see more of these Array methods but is good to know, that there are lots of useful methods for Array class.

As always you can look around on the website of Oracle for more information on the Array class methods.

Java API Documentation → **https://docs.oracle.com/en/java/**

Java Programming: Step by Step from A to Z

Multidimensional Arrays

Multidimensional Arrays

In Java, you can declare an array of arrays known as multidimensional array. That means components of a multidimensional array are also arrays.

Two dimensional array: looks like a matrix. To access the elements of a two-dimensional array, you use two indexes (e.g): a [0][2]

```
int[][] a = new int[3][4];
```

This 2D array can store 3*4 = 12 elements.

	Column 1	Column 1	Column 3	Column 4
Row 1	a [0, 0]	a [0, 1]	a [0, 2]	a [0, 3]
Row 2	a [1, 0]	a [1, 1]	a [1, 2]	a [1, 3]
Row 3	a [2, 0]	a [2, 1]	a [2, 2]	a [2, 3]

Three dimensional array: Here we declared a three dimensional array of size 2 levels * 3 rows * 4 columns. You can think of this array as a cube. Each element requires three indexes to access (e.g.): a [0][1][3]

```
int[][][] a = new int[2][3][4];
```

This 3D array can store 3*4*2 = 24 elements.

Creating two-dimensional array

Arrays beyond 2 dimensions are rarely used, so that's why we'll focus on two-dimensional arrays instead of 3D arrays. Anyway with the knowledge of two dimensional arrays you will succeed with more than two dimensional arrays too. Let's see an example for two-dimensional array creating:

```
int[][] a = new int[3][4];
```

We creat a 2D array without initialization. This is a so called Indirect Method of Declaration.

Notice that the initial base value of the elements are zero.

	Column 1	Column 1	Column 3	Column 4
Row 1	**0** a [0, 0]	**0** a [0, 1]	**0** a [0, 2]	**0** a [0, 3]
Row 2	**0** a [1, 0]	**0** a [1, 1]	**0** a [1, 2]	**0** a [1, 3]
Row 3	**0** a [2, 0]	**0** a [2, 1]	**0** a [2, 2]	**0** a [2, 3]

Creating two-dimensional array

Arrays beyond 2 dimensions are rarely used, so that's why we'll focus on two-dimensional arrays instead of 3D arrays. Anyway with the knowledge of two dimensional arrays you will succeed with more than two dimensional arrays too. Let's see an example for two-dimensional array creating:

```
int[][] a = new int[3][4];
```

Number of elements inside Arrays

Number of Arrays

Notice that the initial base value of the elements are zero.

We creats a 2D array without initialization. This is so called Indirect Method of Declaration.

	Column 1	Column 1	Column 3	Column 4
Row 1	0 a [0, 0]	0 a [0, 1]	0 a [0, 2]	0 a [0, 3]
Row 2	0 a [1, 0]	0 a [1, 1]	0 a [1, 2]	0 a [1, 3]
Row 3	0 a [2, 0]	0 a [2, 1]	0 a [2, 2]	0 a [2, 3]

Creating two-dimensional array

Arrays beyond 2 dimensions are rarely used, so that's why we'll focus on two-dimensional arrays instead of 3D arrays. Anyway with the knowledge of two dimensional arrays you will succeed with more than two dimensional arrays too. Let's see an example for two-dimensional array creating:

```
int[][] a = new int[3][4];

a[0][0]=4;
a[0][1]=2;
a[0][2]=3;
a[0][3]=12;
a[1][0]=5;
a[1][1]=7;
a[1][2]=6;
a[1][3]=9;
a[2][0]=11;
a[2][1]=8;
a[2][2]=1;
a[2][3]=10;
```

Number of elements inside Arrays

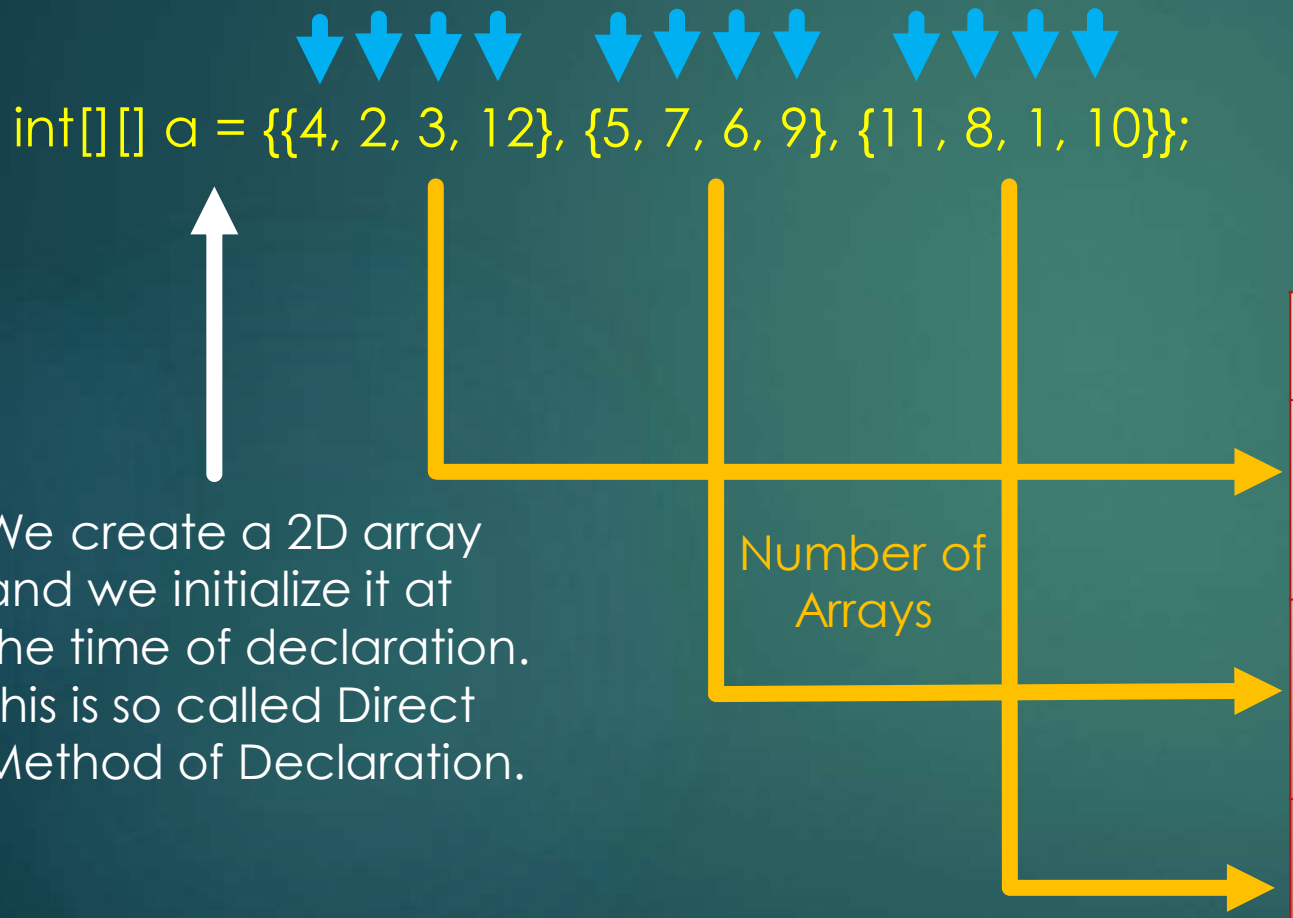
Number of Arrays

Second step: We initialize the array with some data. This is assignment initialization. The indexes are in the square brackets.

	Column 1	Column 1	Column 3	Column 4
Row 1	**4** a [0, 0]	**2** a [0, 1]	**3** a [0, 2]	**12** a [0, 3]
Row 2	**5** a [1, 0]	**7** a [1, 1]	**6** a [1, 2]	**9** a [1, 3]
Row 3	**11** a [2, 0]	**8** a [2, 1]	**1** a [2, 2]	**10** a [2, 3]

Creating two-dimensional array

Let's take a look to another, more used technique how we can create the same two-dimensional array.



Number of elements inside Arrays

	Column 1	Column 1	Column 3	Column 4
Row 1	**4** a [0, 0]	**2** a [0, 1]	**3** a [0, 2]	**12** a [0, 3]
Row 2	**5** a [1, 0]	**7** a [1, 1]	**6** a [1, 2]	**9** a [1, 3]
Row 3	**11** a [2, 0]	**8** a [2, 1]	**1** a [2, 2]	**10** a [2, 3]

Irregular two-dimensional array

Irregular (also called „Jagged") array is a multidimensional array where member arrays have different sizes. For example, we can create a 2D array where the first array contains 4 elements, but the second array contains only 2 elements.

4 — 2 — 3 — elements

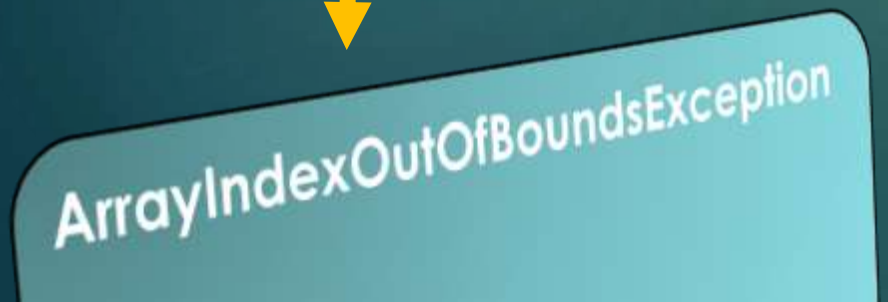
```
int[][] a = {{4, 2, 3, 12}, {5, 7}, {11, 8, 1}};
```

We can creat a Jagged 2D array with this technique.

Number of Arrays

Warning! If you try to access a non-existent index you'll get an error message.

```
System.out.println(a[1][3]);
```



	Column 1	Column 1	Column 3	Column 4
Row 1	**4** a [0, 0]	**2** a [0, 1]	**3** a [0, 2]	**12** a [0, 3]
Row 2	**5** a [1, 0]	**7** a [1, 1]		
Row 3	**11** a [2, 0]	**8** a [2, 1]	**1** a [2, 2]	

4
2
3

Java Programming: Step by Step from A to Z

Varargs

Varargs overview

The varargs is a short-form for ***variable arguments***. It allows a method to accept zero or muliple arguments and it stores these arguments using an array under the hood.

```
public static void exampleMethod(int ... x) {
    // exampleMethod body
}
```

The **...** syntax tells the compiler that we used varargs and these arguments will be stored in an array referred to by **x**.

```
exampleMethod();                    //zero argument
exampleMethod(23);                  //one argument
exampleMethod(23, 44, 99, 100);     //four arguments
```

After that we can call our exampleMethod with an arbitrary number of arguments.

Varargs overview

Don't forget that varargs are arrays so we need to work with them just like we'd work with a normal array.

```
public static void exampleMethod(int ... x) {
        // exampleMethod body
}
```

In this example, we have defined the data type of **x** as **int**. So it can take only integer values.

The number of arguments can be found out using **x.length**, the way we find the length of an array in Java.

Varargs overview

While using varargs you must follow some rules otherwise program code won't compile.

A method can have variable length parameters with other parameters too.

```
public static void exampleMethod(String s, Float f, int ... x) {
        // exampleMethod body
}
```

But varargs parameter should be written last in the parameter list of the method declaration.

```
public static void exampleMethod(int ... x, Float f, String s) {
        // exampleMethod body
}
```

Compile time error

Each method can only have one varargs parameter.

```
public static void exampleMethod(String … s, int ... x) {
        // exampleMethod body
}
```

Compile time error

Java Programming: Step by Step from A to Z

ArrayLists

ArrayList overview

Arrays are of a fixed length. After arrays are created, they cannot grow or shrink. In contrast of this an ArrayList is a re-sizable array, also called a dynamic array (it changes size at runtime as needed).
It grows its size to accommodate new elements and shrinks the size when the elements are removed.

- ArrayList just like arrays, allows you to retrieve the elements by their index.
- ArrayList allows duplicate and null values.
- ArrayList is an ordered collection. It maintains the insertion order of the elements.
- ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

Old (before JDK 1.5) way of creating an ArrayList:

```
ArrayList al = new ArrayList();
```

create an ArrayList

```
ArrayList al = new ArrayList(10);
```

create an ArrayList containing a specific number of slots

ArrayList overview

There is another, more modern, generic way to create an ArrayList.

```
ArrayList al = new ArrayList();
```

Old (before JDK 1.5) way of creating an ArrayList

```
ArrayList<String> al = new ArrayList<String>();
```

New, generic way of creating an ArrayList.

```
ArrayList<String> al = new ArrayList<>();
```

Starting in Java 7, you can even omit that type from the right side.

The main difference is that, in the new way of creation the ArrayList is forced to have the only specified type of objects in it. If you try to add another type of object, it gives compile time error.

This is what makes the containers type-safe.

About generics

Simplistically: Generics forces the programmer to store specific type of objects. We will learn about generics later at the Generics chapter.

ArrayList overview

Elements in an ArrayList are actually objects. For primitive types you have to use wrapper classes: Integer for int, Boolean for boolean, Double for double, Character for char, etc .

```
ArrayList<Integer> myNumbers = new ArrayList<>();
```

ArrayList class is non synchronized. If multiple threads try to modify an ArrayList at the same time, then the final outcome will be non-deterministic.

The ArrayList class implements the List interface.

That's why we could write this:

```
List<String> exampleList = new ArrayList<>();
```

But this is incorrect:

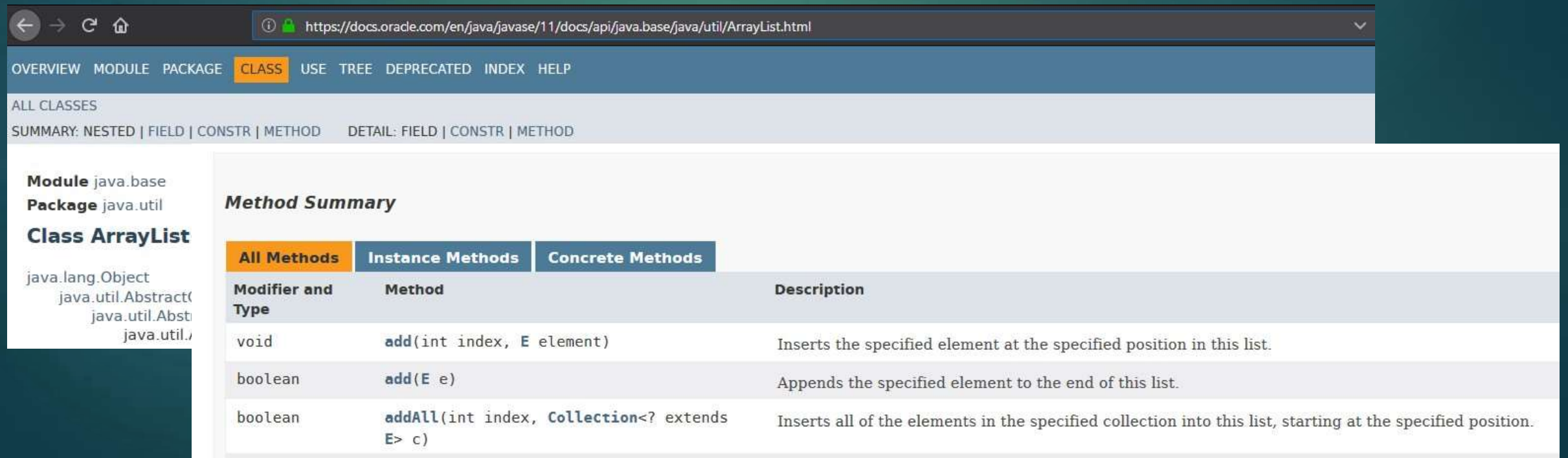
```
ArrayList<String> exampleList = new List<>();
```

ArrayList methods

ArrayList has many methods. In the next lectures, among other things, we will see these.

As always you can look around on the website of Oracle to check ArrayList class methods.

Java API Documentation → **https://docs.oracle.com/en/java/**



Java Programming: Step by Step from A to Z

Generics

Generics overview

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Type parameters provide a way for you to re-use the same code with different inputs. The inputs to type parameters are types. Advantages of Generics:

Type-safety: Forced to have the only one specified type of object.

```
ArrayList list = new ArrayList();
```

```
ArrayList<String> list = new ArrayList<>();
```

Type casting is not required: With Generics we don't need to typecast the object.

```
ArrayList list = new ArrayList();
list.add("Hello Java!");
String s = (String) list.get(0);  // Typecasting
```

```
ArrayList <String> list = new ArrayList<>();
list.add("Hello Java!");
String s = list.get(0);
```

Compile-Time Checking: It is checked at compile time so no problem will occur at runtime. Better programming practice.

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello Java!");
list.add(21); //Compile Time Error
```

Generic method

We can also write generic methods that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
Integer[] arrayInt = {10, 20, 30};
Character[] arrayChar = {'A', 'B', 'C'};
printArray(arrayInt);
printArray(arrayChar);
```

```
static void printArray(Integer[] integer){
        for(Integer i : integer) {
        System.out.println(i);
        }
}
static void printArray(Character[] character){
        for(Character c : character) {
        System.out.println(c);
        }
}
```

With method overloading you have to make different methods for different input types.

```
static <T>void printArray(T[] t){
        for(T item: t) {
        System.out.print(item);
}
```

<>: The „Diamond" operator

T: The generic type parameter

t: Instance of generic type T.

With generic one method is enough, it can be called with different types of arguments.

Generic naming conventions

A type parameter can be named anything you want. The convention is to use single uppercase letters to make it obvious that they aren't real class names. The following are common letters to use:

```
static <T>void printArray(T[] t){
    for(T item: t) {
    System.out.print(item);
}
```

Generic type parameter

E for an element

K for a map key

V for a map value

N for a number

T for a generic data type

S, **U**, **V**, and so forth for multiple generic types

Generic method with two types

A Generic method can deal with more than one generic type, where this is the case, all generic types must be added to the method signature.

method with one generic type

```
static <T>void printArray(T[] t){
      for(T item: t) {
      System.out.print(item);
}
```

<>: The „Diamond" operator
T: The generic type parameter
t: Instance of generic type T

method with two generic types

```
static <T, S>void printArray(T[] t, S[] s){
      for(T item: t) {
            for(S item2: s) {
      System.out.print(item +"-"+ item2 + ",");
      }
}
```

<>: The „Diamond" operator
T, S: The generic types parameters
t, s: Instances of generic type T and S

Generic class

A class that can refer to any type is known as generic class. A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section. The type parameter section of a generic class can have one or more type parameters separated by commas.

```
public class ClassName<T> {
    private T contents;
}
```

```
public class ClassName<T, S> {
    private T contents;
    private S contents2;
}
```

The **T** (and **S**) type indicates that it can refer to any type (like Integer , String, Student etc.). The type you specify for the class, will be used to store and retrieve the data.

When you instantiate the class, you tell the compiler what **T** (and **S**) should be for that particular instance

Generic wildcard

Wildcards are represented by the question mark "?" and they are used to refer to an unknown type.

If we write **<? extends Number>**, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class (subclass) object. This is called an **upper bounded** wildcard.

```
public static void countSomething(List<? extends Number> list) {
        // more code
}
```
upper bounded wildcard

Wildcards can also be specified with a **lower bound**, where the unknown type has to be a supertype of the specified type. If we write **<? super Integer>** it means any superclass (and all its parents) of Integer e.g. Number, Object.

```
public static void countSomething(List<? super Integer> list) {
        // more code
}
```
lower bounded wildcard

The unbounded wildcard type is called a list of unknown type. **<?>** It's useful when writing a method that can be implemented using functionality provided in Object class. When methods don't depend on the type parameter.

```
public static void print(List<?> list) {
        // more code
}
```
unbounded wildcard

Java Programming: Step by Step from A to Z

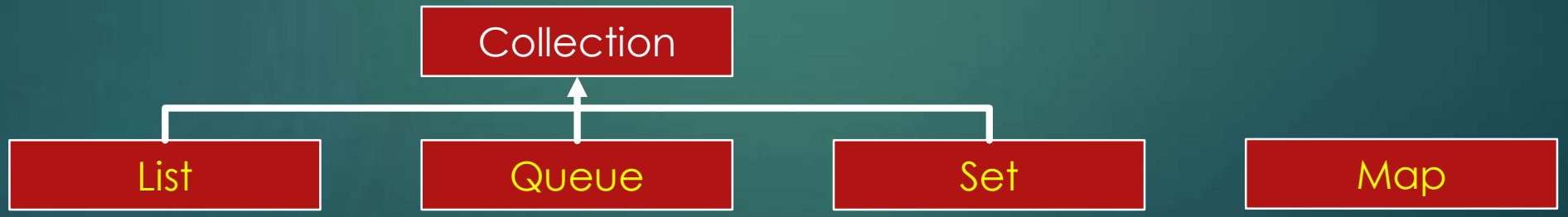
Collection Framework

Collection framework overview

A collection is a group of objects contained in a single object. The Java Collections Framework is a set of Interfaces, Classes and Algorithms for storing collections. Some of the benefits:

- reduced development effort by using core collection classes rather than implementing our own collection classes.
- code quality is enhanced with the use of well tested collections framework classes.
- reusability and interoperability

The Java Collections Framework includes four main types of data structures: lists, sets, queues, and maps. The Collection interface is the parent interface of **List**, **Set**, and **Queue**. The **Map** interface does not extend Collection.

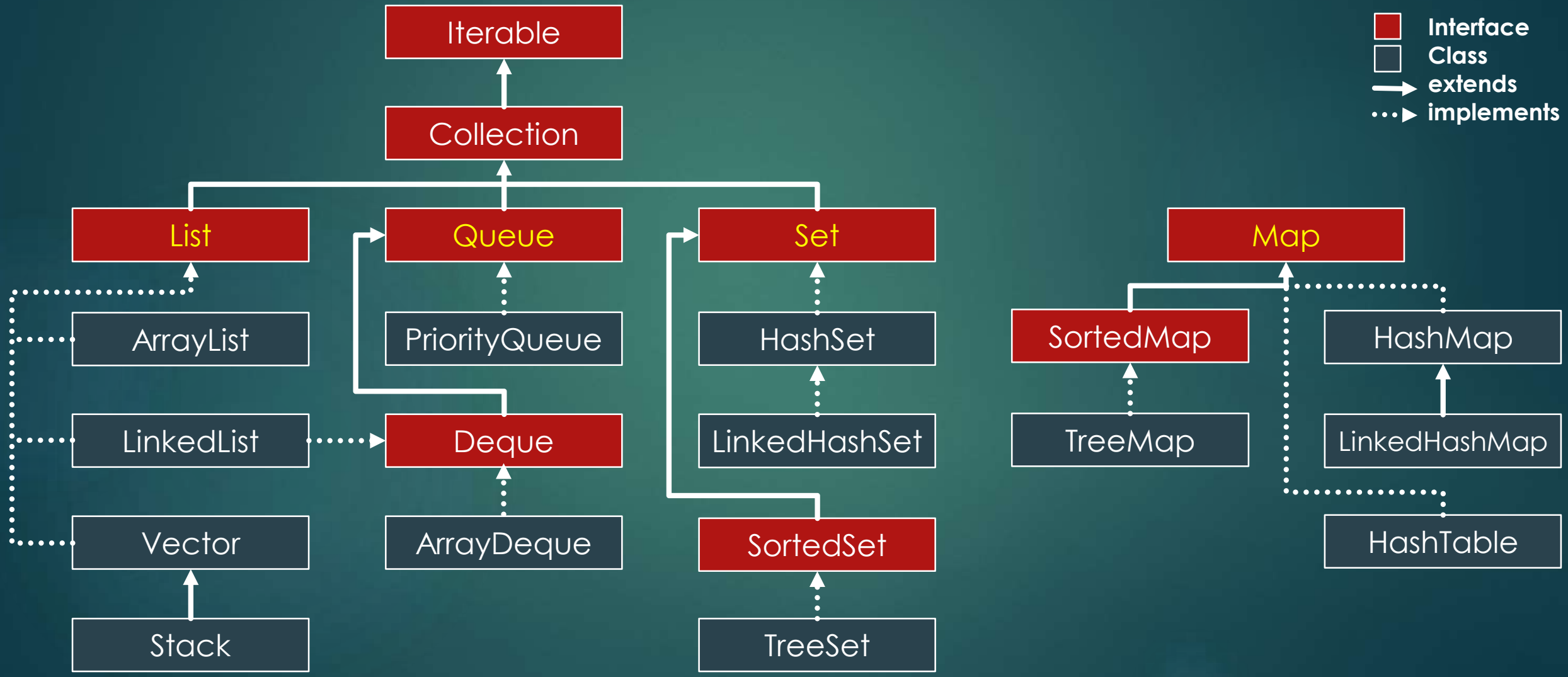


Although maps are not collections in the proper use of the term, but they are fully integrated with collections and it is considered part of the Java Collections Framework.

Collection framework overview

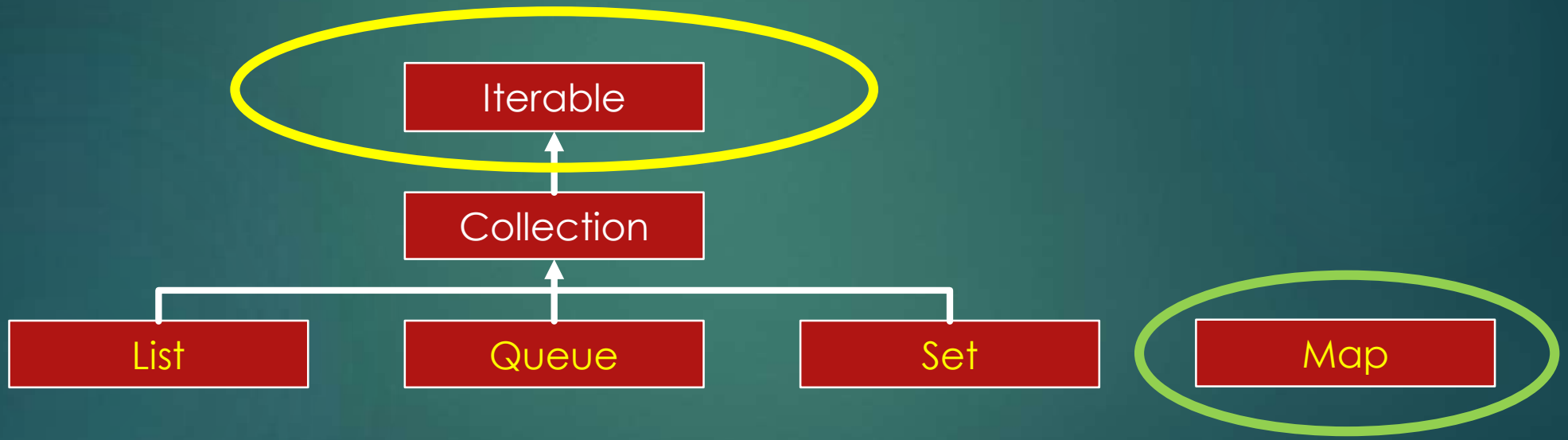
List: A list is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an int index.
Queue: A queue is a collection that orders its elements in a specific order for processing.
Set: A set is a collection that does not allow duplicate entries.
Map: A map is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

Collection framework hierarchy



Iterable interface

When Collection Framework was created it gave a powerful ability to acquire data from collections automatically called Iterability (ability to iterate through the collection) by extending **Iterable interface** to Collection. It's very useful, because of this we can use an iterator method or a foreach loop to any kind of Collections.



Iterating ability can only be applied to one dimesional collections like lists, queues, sets but **not to maps**. Why? Because Iterator uses the indexes of elements and Map doesn't support indexes but key-value structure, Map can't be iterated and hence can't extends Collection which further extends Iterable.

Collection framework's useful methods

Another advantage of the Collection interface that it contains useful methods for working with lists, sets, and queues. These methods can be familiar to you, because we already talked about these in our ArrayList's method section. What is very good in these (and from now on you will know too) that they can be used not only for ArrayLists but also for other lists, sets, and queues.

The **add()** method inserts a new element into the Collection and returns whether it was successful. boolean add(E element)
The **remove()** method removes a single matching value in the Collection and returns whether it was successful. boolean remove(Object object)
The **isEmpty()** and **size()** methods look at how many elements are in the Collection. boolean isEmpty() & int size()
The **clear()** method provides an easy way to discard all elements of the Collection. void clear()
The **contains()** method checks if a certain value is in the Collection. boolean contains(Object object)

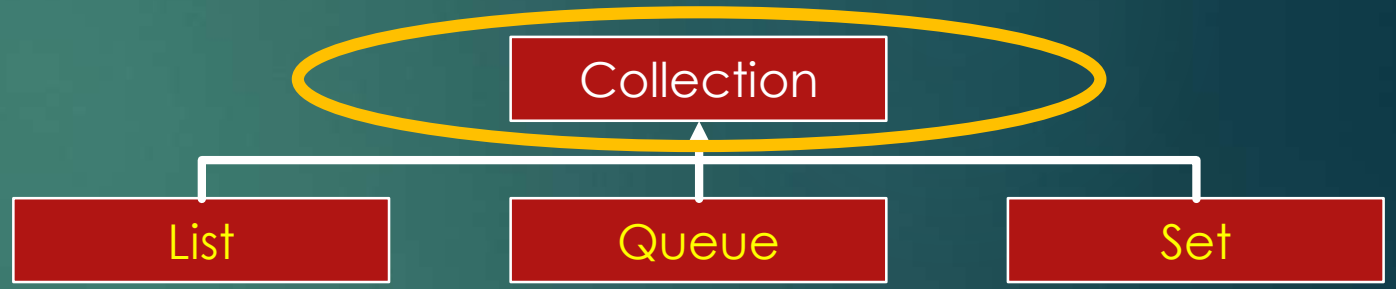
Hint! Previously in:
Java programing: Step by Step from A to Z, lecture 46, „ArrayList methods"

Collection vs Collections

Before we look at the different Collection Classes in practice in the next chapters, here is an interesting but also confusing tricky interview question about the Collection Framework as a closing.

What are the differences between Collection and Collections in Java?

As we have seen **Collection** is a root level **interface** of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface.



Collections is a utility **class** in java.util package. It consists of only static methods which are used to operate on objects of type Collection. For example **sort(List<T> list)** method to sort the collection, and so on.

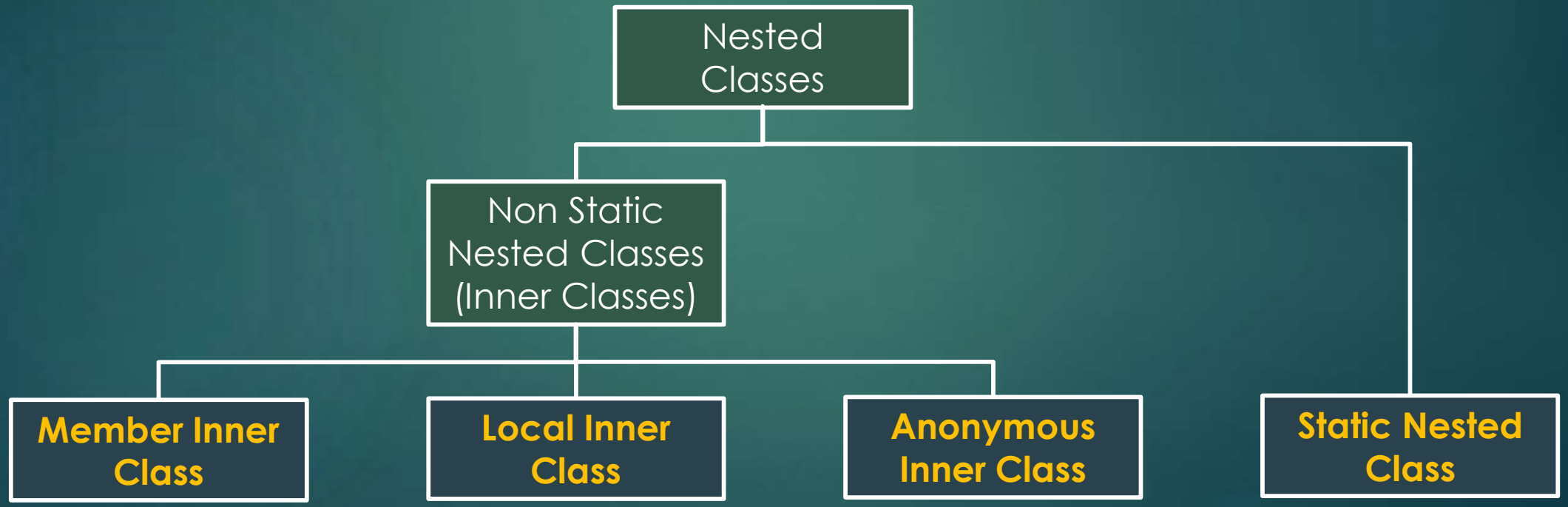
Java Programming: Step by Step from A to Z

Nested Classes

Nested classes overview

A Nested Class is a class that is defined within another class. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, create more readable, maintainable code and it requires less code to write (Code Optimization).

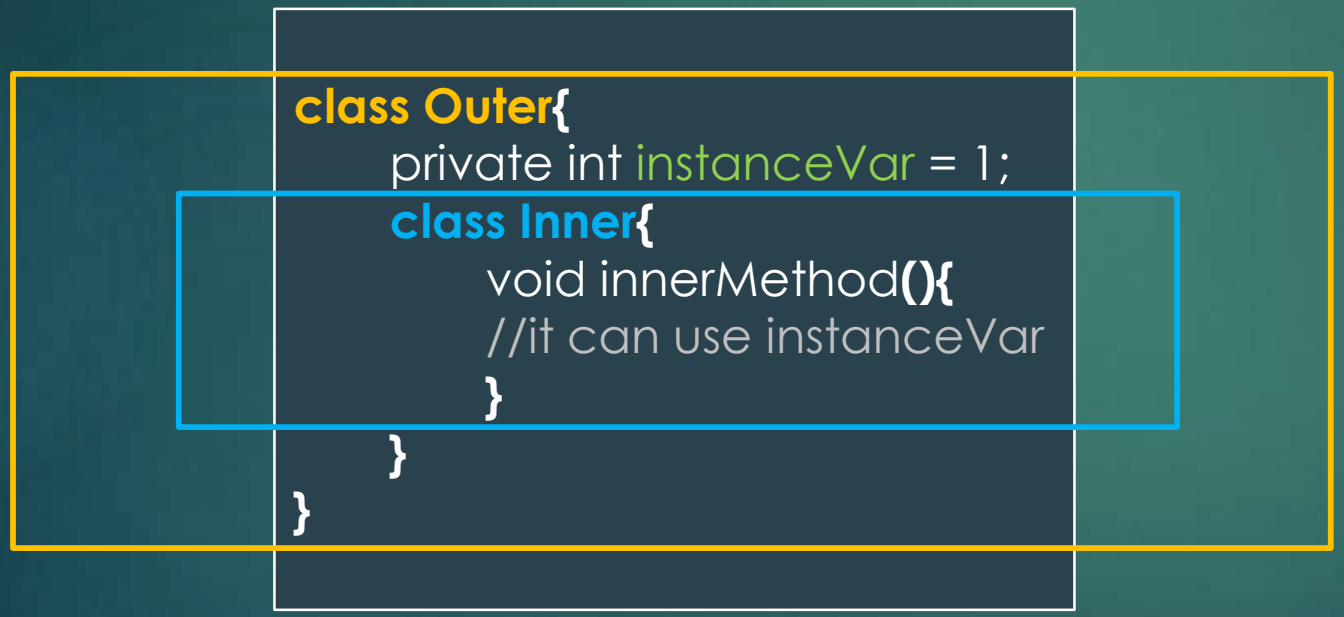
There are four types of nested classes (A nested class that is not static is called an inner class):



Member Inner Class

Member Inner Class is a non-static class that is created inside a class but outside a method.

Member Inner class is accessing the instance variable member of the Outer class.



If you want to instantiate Inner class, first you must have to create the instance of Outer class.

```
Outer o = new Outer();

Outer.Inner i = o.new Inner();
i.innerMethod();
```

After that an instance of Inner class is created inside the instance of Outer class and with this, you can access the method inside Inner class.

Local Inner Class

Local Inner Class is a non-static class that is created inside a class but (and this is the difference from the previous Member Inner Class) inside a method.

Local Inner class is accessing the instance variable member of the Outer class.

```
class Outer{
    private int instanceVar = 1;
    void outerMethod(){
        class Inner{
            void innerMethod(){
             //it can use instanceVar
            }
        }
        Inner i = new Inner();
        i. innerMethod();
    }
}
```

You can instantiate Inner class inside of the method which contains the Inner Class. This way you can access the inner method.

```
Outer o = new Outer();
o.outerMethod();
```

After that if you create an instance of Outer class, you can call the method which contains the Inner Class.

Static Nested Class

A static nested class is a static class that is defined at the same level as static variables. It's a nested class which is a static member of the outer class.

Static Nested class is accessing the static variable member of the Outer class. But keep in mind that it cannot access non-static instance variables and methods of the outer class.

```
class Outer{
    private static int staticVar = 1;
    static class Nested{
        void nestedMethod(){
        //it can use staticVar
        }
    }
}
```

Nested class can be accessed without instantiating the Outer class, using other static members.

```
Outer.Nested i = new Outer.Nested();
i. nestedMethod();
```

After that you can access the method inside Nested class.

Anonymous Inner Class

It is an inner class without a name. It should be used if you have to override a method of a class or an interface. We can have an anonymous inner class that extends a class (may be abstract or concrete) or that implements an interface.

```
abstract class AClass{
    abstract void absMethod();
}
```

```
public interface AInterface{
    void intMethod();
}
```

```
AClass aClass = new AClass() {
    @Override
    void absMethod() {
    //code here
    }
};
aClass.absMethod();
```

```
AInterface aInterface = new AInterface() {
    public void intMethod() {
    //code here
    }
};
aInterface.intMethod();
```

Java Programming: Step by Step from A to Z

Exceptions

Exceptions overview

Reminder

Java uses exceptions to handle errors and other exceptional events. When an error occurs during the code execution Java will normally stop and generate an error message. The technical term for this is that Java will **throw an exception** (show an error). One of the aims of the Exception Handling mechanism is to handle the runtime errors so that normal flow of the application can be maintained.

```
int num = 22 / 0; // may throw exception

System.out.println("rest of the code");
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at com.udemy.app.App.main(App.java:11)
```

```
try {
        int num = 22 / 0; // may throw exception
}
// handling the exception
catch (ArithmeticException e) {
        System.out.println(e);
}
System.out.println("rest of the code");
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Exceptions overview

Reminder

Try-catch and finally block

The **try** statement allows you to define a block of code to be tested for errors while it is being executed. The try block must be followed by either catch or finally.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block. It can be followed by finally block later.

The **finally** block is used to execute the important code of the program. It is executed whether an exception is handled or not.

```
try {
        // code that may throw an exception
}
catch (Exception e) {
        // handling the exception
}
finally {
        // finally block is always executed
        // whether exception is handled or not
}
```

Hint! Previously in:
First Steps in Java,
lecture 23,
„Exceptions"

Exceptions overview

Reminder

Throw and throws keyword

The **throw** keyword is used to throw an exception. This statement is used together with an exception type. e.g. ArithmeticException, ClassNotFoundException, ArrayIndexOutOfBoundsException, etc.

```
static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException(„Not valid age");
    }
    //more code
```

The **throws** keyword indicates what exception type may be thrown by a method.

```
static void checkAge(int age) throws ArithmeticException {
    if (age < 18) {
      throw new ArithmeticException(" Not valid age ");
    }
    //more code
```

Hint! Previously in:
First Steps in Java,
lecture 24,
„Throws and throw"

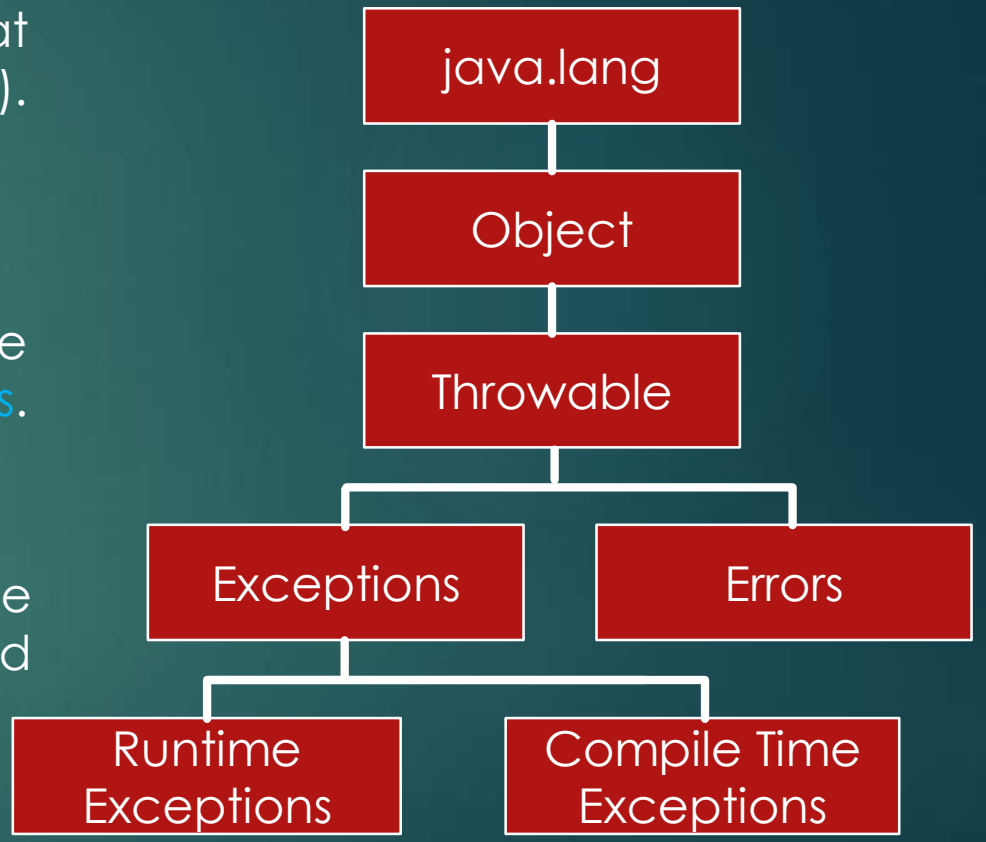
Exceptions overview - Hierarchy

Reminder

A **checked exception** is checked by the compiler at compilation-time (also known as compile time exceptions). The programmer should handle these exceptions.
E.g.: IOException, SQLException, etc.

An **unchecked exception** is an exception that occurs at the time of execution. These are also called as runtime exceptions.
E.g.: ArithmeticException, NullPointerException, etc.

Errors. Problems arise beyond the control of the user or the programer. Errors are generated to indicate errors generated by the runtime environment.
E.g: OutOfMemoryError, VirtualMachineError, etc.



Exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Multiple exceptions

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

```
try {
        // code that may throw an exception
} catch (ArithmeticException e) {
        // handling the exception
} catch (NumberFormatException e) {
        // handling the exception
}
```

Nested try block

Exception handling can be nested. This can become necessary when methods are used in a catch or finally block that also throws exceptions.

```
// Outer try-catch block
try {
        // code that may throw an exception

    // Inner try-catch block
    try {
            // code that may throw an exception
    } catch(RuntimeException rtex) {
            // handling the exception
    }

} catch(Exception ex) {
  // handling the exception
}
```

Exception Handling with Method Overriding

An overriding method (the method of child class) can throw any unchecked exceptions, regardless of whether the overridden method (method of parent, super class) throws exceptions or not.

However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw those checked exceptions, which have less scope than the exception(s) declared in the overridden method.

Simply put, subclasses can throw fewer checked exceptions than their superclass, but not more.

In the next chapters we will look at the Multiple Exceptions, the Nested try block and Exception Handling with Method Overriding in practice.

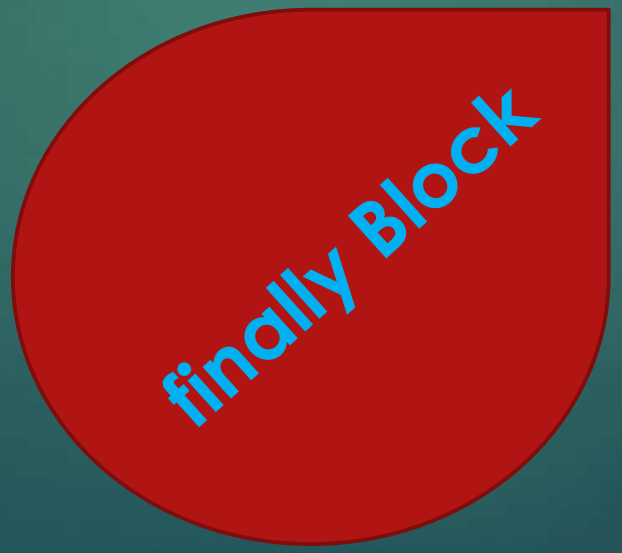
Java Programming: Step by Step from A to Z

Difference between final, finally and finalize

final, finally and finalize

In this chapter, we're going to take an overview of three already known Java keywords: final, finally and finalize. While these keywords resemble each other each has a very different meaning in Java. Many people are confused about this topic, so it's worth summarizing it.

Helps you memorize and differentiate these words if you match these to phrases which express what they are about.



final

final is a keyword

```
class FinalExample{
        public static void main(String[] args){
        final int i = 10;
        i = 20; // Compile Time Error
        }
}
```

The variable declared as final should be initialized only once and cannot be changed.

Java classes declared as final cannot be extended.

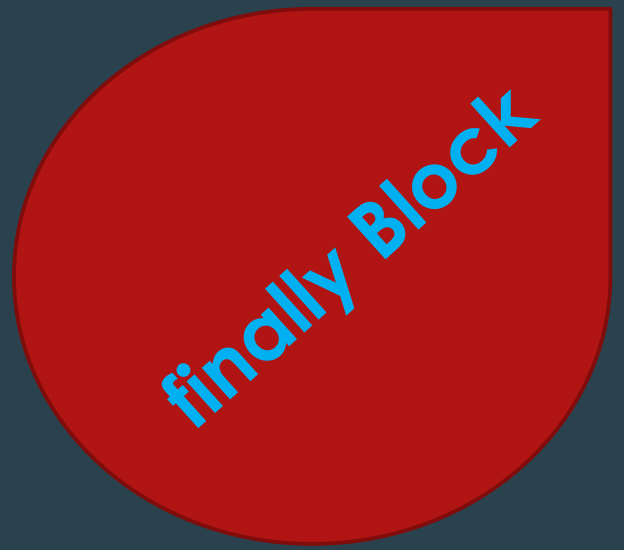
Methods declared as final cannot be overridden.

Putting final in front of a field, a variable or a parameter means that once the reference has been assigned then it cannot be changed.

finally

```
try {
        // code that may throw an exception
}
catch (Exception e) {
        // handling the exception
}
finally {
        // finally block is always executed
        // whether exception is handled or not
}
```

finally is a block



The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

But finally is useful for more than just exception handling - it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

finalize

```
class FinalizeExample{
        public void finalize(){
                System.out.println("finalize called");
        }
        public static void main(String[] args){
                FinalizeExample f = new FinalizeExample();
                f = null;
                System.gc();
        }
}
```

finalize is a method

Before an object is garbage collected, the runtime system calls its finalize() method.

You can write system resources release code in finalize() method before getting garbage collected.

As finalize is a method and not a reserved keyword, so we can call finalize method explicitly, then it will be executed just like normal method call but object won't be deleted/destroyed.

Java Programming: Step by Step from A to Z

OOP Overview

Deeper into Object-Oriented Programming

Object-Oriented Programming overview

Reminder

As we learned earlier Object-Oriented Programming (OOP) is a methodology to design a program using classes and objects. OOP concepts make it possible to reuse code without creating security risks or making a Java program less readable. It allows us to create specific interactions between objects.

A class can be defined as a blueprint from which you can create an individual object. The object is an entity that has state and behavior. An Object can be defined as an instance of a class.

As we already seen in our "First steps in Java" course there are four main principles of OOP:

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

OOP overview - Inheritance

Reminder

When one object takes all the properties and behaviors of a parent object, it is known as inheritance. When you inherit from an existing class, you can reuse methods and fields of the parent (super) class in the child (sub) class.

The objectives : It provides **Code Reusability**, and **Code Readability**.

Inheritance

```
public class ParentClass {
        // methods and fields of the parent class
        protected String fieldName = "Xyz";
        public void methodName() {
                // more code
        }
}
```

```
public class ChildClass extends ParentClass {
        // reuse methods and fields of the parent class
        // new methods and fields in current child class
}
```

Hint! Previously in: First Steps in Java, lecture 25, „Inheritance"

OOP overview - Polymorphism

Reminder

This concept refers to the ability to perform a certain action in different ways. It can take two forms.
Method overriding occurs when the child class overrides a method of its parent.
Method overloading happens when various methods with the same name are present in a class.

The objectives : It provides **Code Flexibility** and **Better Implementation of Inheritance**.

Run Time Polymorphism
Method overriding

```
public class ParentClass {
    protected void methodName() {
     // more code
    }
}
```

```
public class ChildClass extends ParentClass
{
    @Override
    protected void methodName() {
    // more code
    }
}
```

Complie Time Polymorphism
Method overloading

```
public void jump() {
    // more code
}
public void jump(int num) {
    // more code
}
```

Hint! Previously in: First Steps in Java, lecture 28, „Polymorphism"

OOP overview - Abstraction

Reminder

It hides complexity of data and shows only the relevant information, gives flexibility to programmers to change the implementation of the abstract behaviour. There are two ways to achieve abstraction:
Abstract classes: Partial abstraction (0-100%)
Interfaces: Total abstraction (100%)

The objectives : It provides **Flexibility of Implementation**, **Code Reusability** and **Multiple Inheritance.**

Abstraction
Abstract class

```
abstract class AbClass {
    abstract void abstractMethod();
}
```

Abstraction
Interface

```
interface InterfaceName{
    void interfaceMethod();
}
```

Hint! Previously in:
First Steps in Java,
lecture 29,
„Abstraction"

Hint! Previously in:
First Steps in Java,
lecture 31,
„Interfaces"

OOP overview – Encapsulation

Reminder

Binding or wrapping code and data together into a single unit are known as encapsulation. It restricts direct access to data members (fields) of a class. You can implement encapsulation in Java by keeping the fields (class variables) private and providing public getter and setter methods to each of them.

The objectives : It provides **Code Readability** and **Data Protection**.

Encapsulation

```
public class EncapsulatedClass{
        // field set to private: restricted access
        private String name;

        // getter
        public String getName() {
        return name;
        }
        // setter
        public void setName(String name) {
        this.name = name;
        }
}
```

Hint! Previously in: First Steps in Java, lecture 30, „Encapsulation"

Deeper into OOP

Because these concepts in Java are the main ideas behind Java's Object Oriented Programming it's the key to a better understanding. That's why we will go deeper into these in the next chapters. We will look at the following:

Inheritance:
After we already know Single-level Inheritance we will look at other types of Inheritance as Multilevel Inheritance, Hierarchical Inheritance, Multiple Inheritance and Hybrid Inheritance.

Polymorphism:
We will see Polymorphism with Multilevel Inheritance and Runtime Polymorphism with Data Members.

Encapsulation:
We will examine Advanced Encapsulation, Encapsulation with Mutable Classes and JavaBeans.

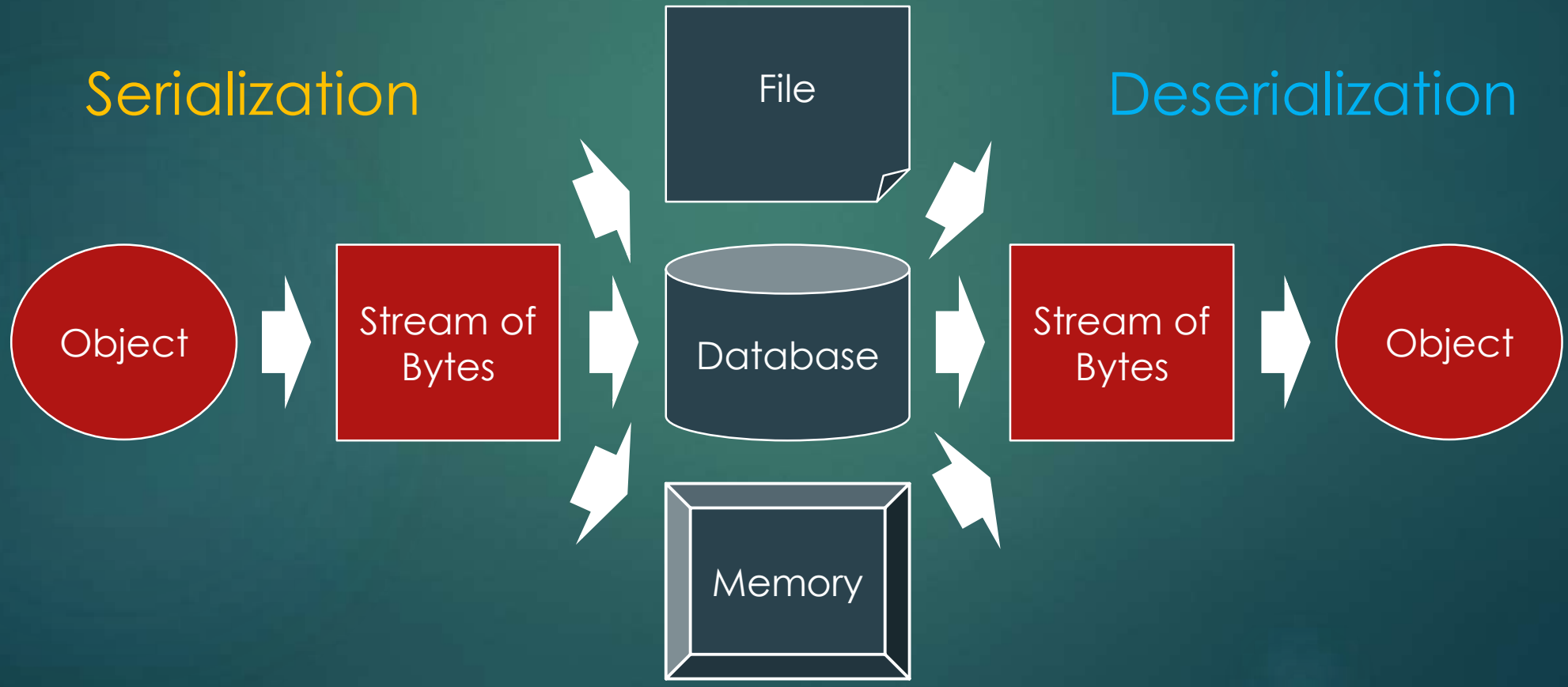
Abstraction:
We will see Default Methods in Interfaces and the Nested or Inner Interface in another Interface or in Class.

Java Programming: Step by Step from A to Z

Serialization

Serialization overview

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to **persist the object** and to **travel object's state** on the network (which is known as marshaling).



Serialization overview

Only the objects of those classes can be serialized which are implementing java.io.Serializable interface. Serializable has no data member and method, so it's a marker interface. It is used to "mark" java classes so that objects of these classes may get certain capability.

```
import java.io.Serializable;

public class Example implements Serializable {
    //more code
}
```

Serialization overview

Some „good to know" things about Serialization:

During the serialization process only non-static data members are saved, static data members not.

If you don't want to save the value of a non-static data member you can make it transient and that means it won't be saved via the serialization process.

If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice versa it isn't true.

If a class has a reference to another class, all the references must be Serializable otherwise the serialization process will not be performed. In such case, NotSerializableException is thrown at runtime.

When an object is deserialized the constructor of the object is never called.

Java programing: Step by Step from A to Z

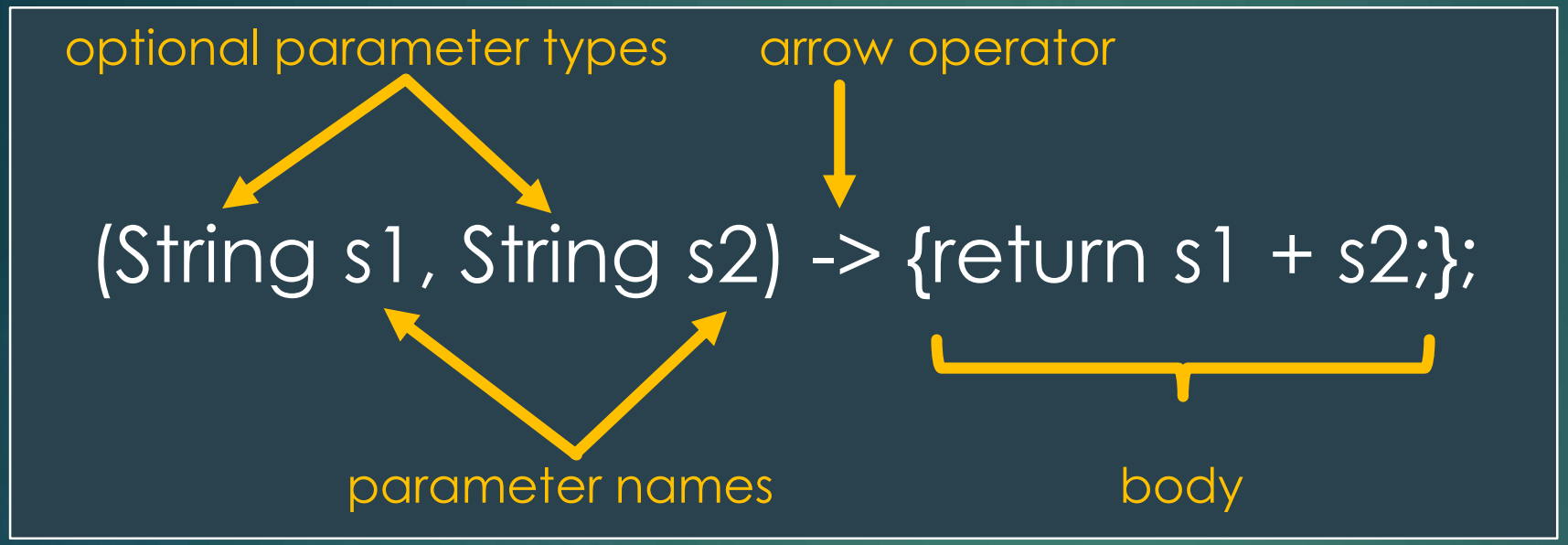
Lambda Expression & Functional Interface

Lambda Expression overview

The Lambda expression was added in Java 8 and it's a block of code that gets passed around, like an anonymous method. It provides a clear and concise way to represent single method (functional) interface using an expression. You can use lambda expressions only with functional interfaces (an interface that contains only one abstract method). Advantages :

- A function that can be created without belonging to any class.
- Less coding.
- A lambda expression can be passed around as if it was an object and executed on demand.

Lambda Expression Syntax



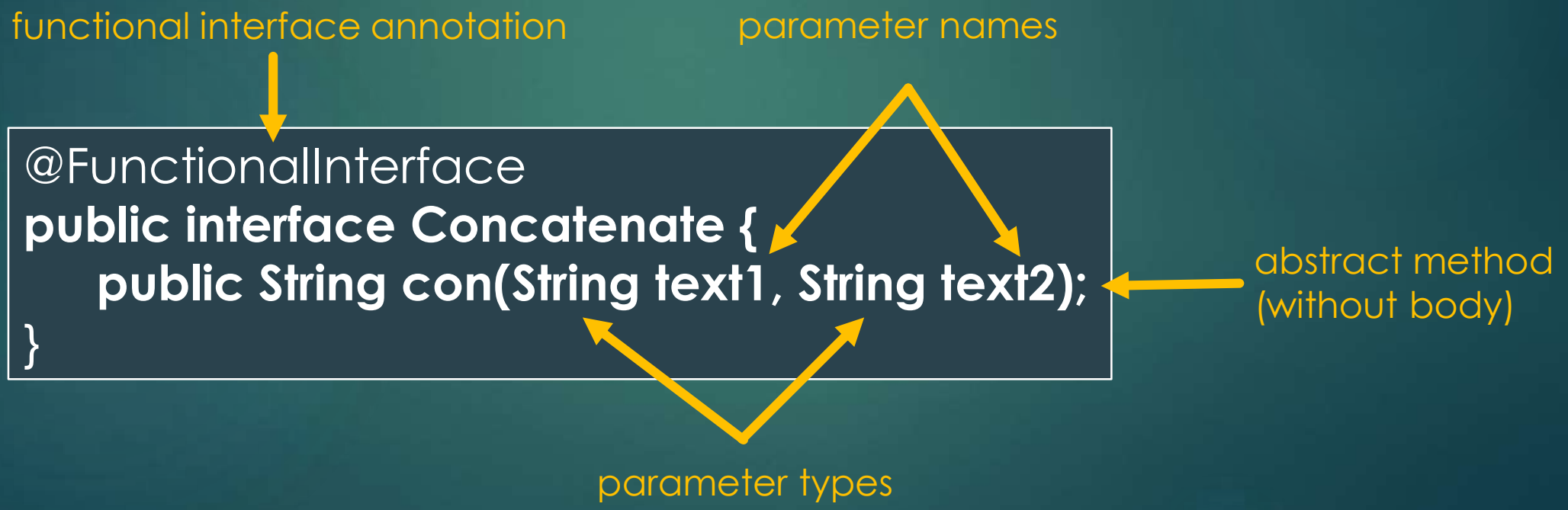
Lambda syntax,
including optional parts

```
(s1, s2) -> s1 + s2;
```

Lambda syntax
omitting optional parts

Functional interface

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. Lambda expressions can be used to represent the instance of a functional interface.

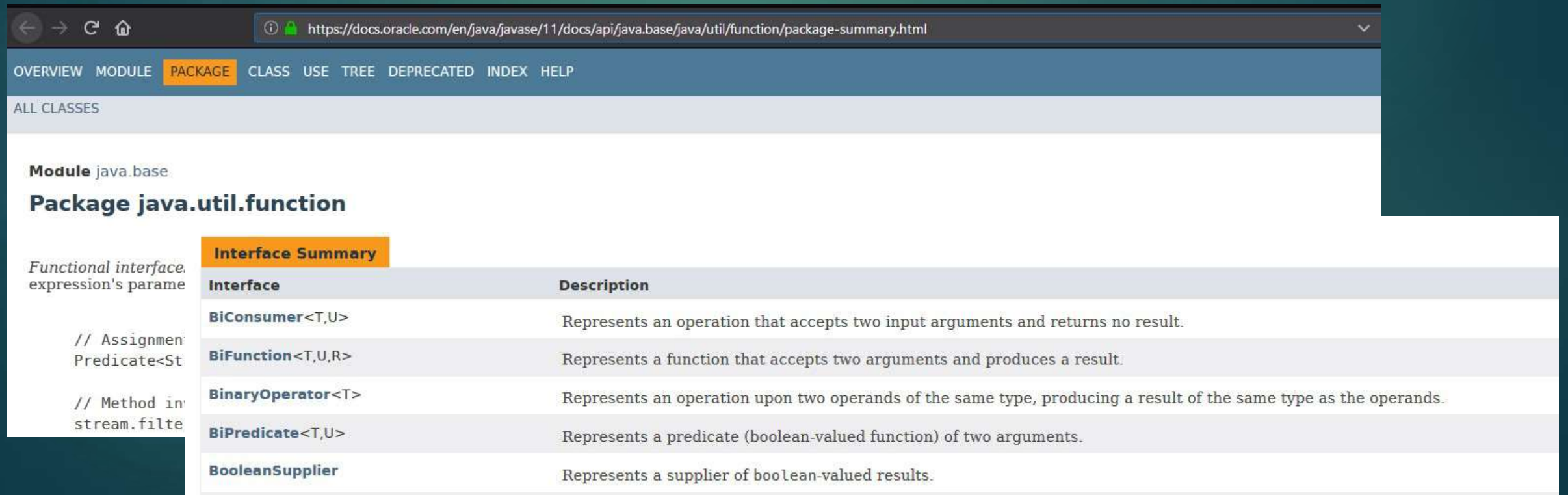


Predefined-Functional Interfaces

As you will see in the next lectures you can define your own custom functional interfaces, but useful to know that Java provides predefined functional interfaces too to deal with functional programming by using lambda and method references. These predefined functional interfaces are placed in java.util.function package.

You can look around at any time on the website of Oracle for these.

Java API Documentation → **https://docs.oracle.com/en/java/**



Java programing: Step by Step from A to Z

Method Reference

Method Reference overview

We can use lambda expressions to implement functional interfaces as we have seen in the previous lectures, but the lambda expressions are not the only mechanisms we can use for this purpose.

Sometimes a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

In other words a method reference is the shorthand syntax for a lambda expression that executes just one method.

Method references always utilize the :: operator.

Method Reference overview

There are four kinds of method references:

Kind	Syntax
Reference to a Static Method	ContainingClass::staticMethodName
Reference to an Instance Method of a Particular Object	containingObject::instanceMethodName
Reference to an Instance Method of an Arbitrary Object of a Particular Type	ContainingType::methodName
Reference to a Constructor	ClassName::new

Method Reference overview

Here are some examples for Lambda versus Method reference solutions. The results are the same but in many cases, the Method reference solution has easier syntax.

LAMBDA		METHOD REFERENCE
str -> str.length()		String::length
(str, i) -> str.substring(i)		String::substring
(str, s2, s3) -> str.replaceAll(s2, s3)		String::replaceAll

However, we can't use Method reference to replace all kinds of lambda expressions since they have some limitations. We will see examples of these in the following lectures. Until then the golden rule that needs to be remembered is that:

If the task can be accomplished with both methods use that which is more readable in that case.

Java Programming: Step by Step from A to Z

Optional

Optional overview

Java 8 has introduced a new class **Optional** in java.util package. It is a public final class and used to deal with **NullPointerException** which can crash your code, and is very hard to avoid without using too many null checks.

It's a **wrapper class** that contains an optional value, meaning it can either contain an **object** or a null value (**empty**). Therefore, it is possible to manipulate null values as if they were normal instances without necessarily having to perform a null check at every step.

An empty Optional

Contains an object of type Car

Optional overview

The Optional class includes various utility methods to explicitly deal with the cases where a value is present or absent.

However, the advantage compared to null references is that the Optional class forces you to think about the case when the value is not present. As a consequence, you can prevent unintended null pointer exceptions.

It is important to know that the intention of the Optional class is not to replace every single null reference. Instead, its purpose is to help design more-comprehensible code so that by just reading the signature of a method, you can tell whether you can expect an optional value. This forces you to actively unwrap an Optional to deal with the absence of a value.

Optional overview

Examples of creating optional objects:

An empty Optional

```
Optional<Car> car = Optional.empty();
```

Optional with not null value

```
Car car = new Car();
Optional<Car> c = Optional.ofNullable(car);
```

The "ofNullable()" method ensures that if the car was null, the result would be an empty Optional instead of null.

Examples of other useful utility methods:

For example if a value is present in an Optional container object isPresent() will return true and get() will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as orElse() (return a default value if value not present) and ifPresent() (execute a block of code if the value is present).

map vs flatMap

map: If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result.

flatMap: If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional. Its purpose is to apply the transformation function on the value of an Optional (just like the map operation does) but then flatten the resulting multi-level Optional into a single one.

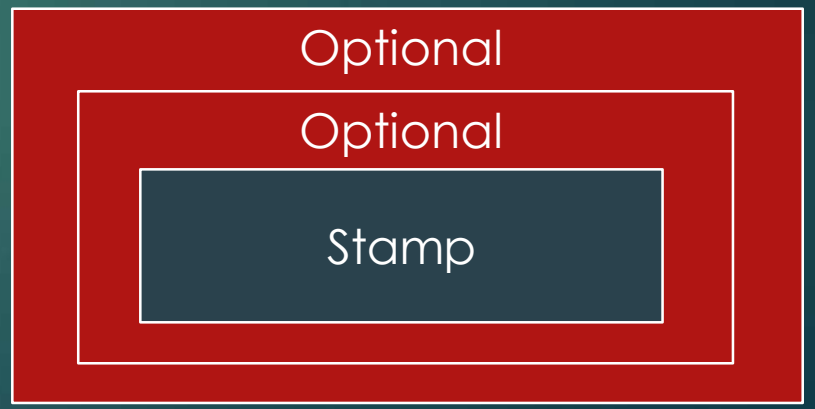
map vs flatMap

```
String result = person.map(Person::getPassport)
                      .map(Passport::getStamp)
                      .map(Stamp::getCountryOfStamp)
                      .orElse("Unknown");
```

Compile time error: The type Passport does not define getStamp(Optional<Passport>) that is applicable here

Person::getPassport result is Optional <Optional <Passport>>

As a result, the call to getStamp() is invalid because the outermost Optional contains as its value another Optional, which of course doesn't support the getStamp() method.



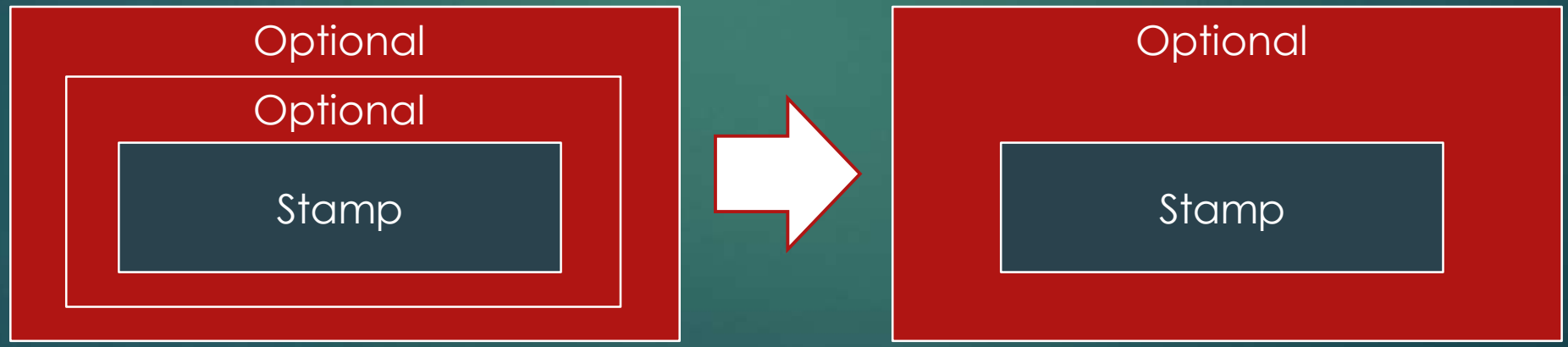
map vs flatMap

```
String result = person.flatMap(Person::getPassport)
                      .flatMap(Passport::getStamp)
                      .map(Stamp::getCountryOfStamp)
                      .orElse("Unknown");
```

flatMap flatten the resulting multi-level Optional into a single one

The first flatMap ensures that an Optional<Password> is returned instead of an Optional<Optional<Password>>, and the second flatMap achieves the same purpose to return an Optional<Stamp>. Note that the third call just needs to be a map() because getCountry() returns a String rather than an Optional object.

with maps



with flatMaps

Golden rule:

Use map if the function returns the object you need or flatMap if the function returns an Optional.

Java Programming: Step by Step from A to Z

Stream

Stream overview

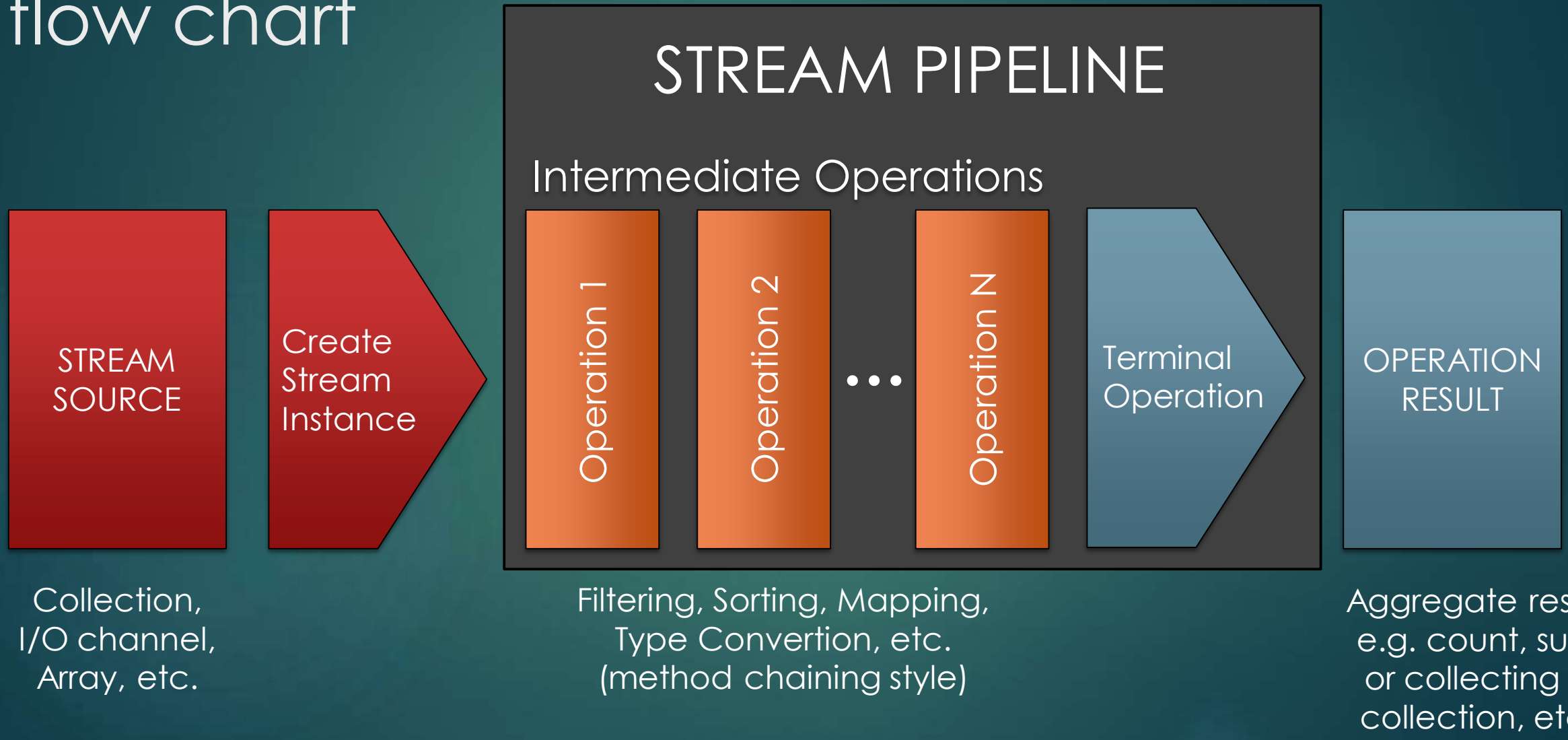
Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

You can process data in a declarative (solve problems without requiring the programmer to specify an exact procedure to be followed) way similar to SQL statements. Stream lets the developer leverage multicore architecture without the need to write any specific code for it.

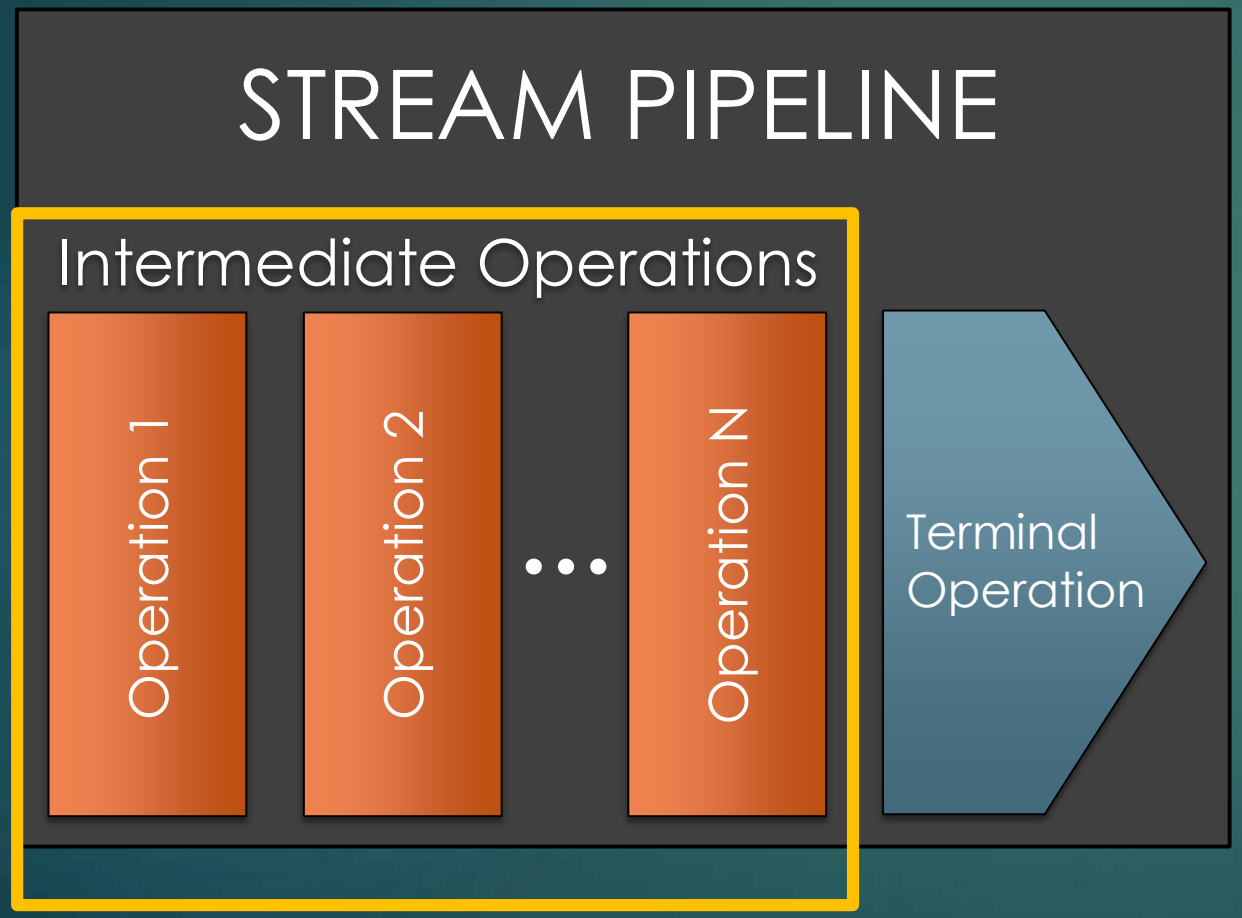
A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels. Stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.

Operations performed on a stream does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, but this does not change the original, source collection.

Stream flow chart

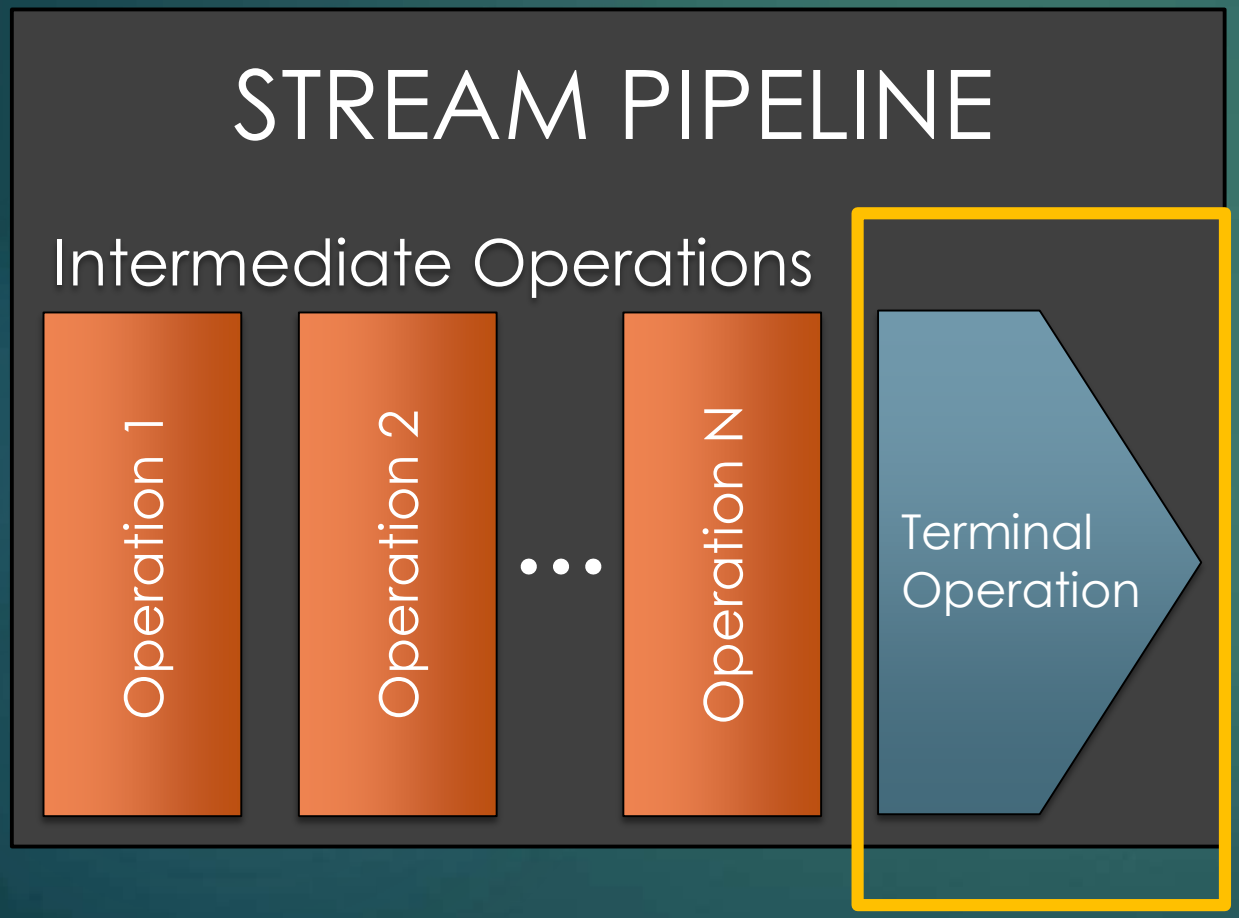


Stream's intermediate operations



filter()
map()
flatMap()
distinct()
sorted()
peek()
limit()
skip()

Stream's terminal operations



toArray()
collect()
count()
reduce()
forEach()
forEachOrder
ed()
min()
max()
anyMatch()
allMatch()
noneMatch()
findAny()
findFirst()

Stream example

```
List<String> names = Arrays.asList("Alan", "James", "Kevin", "Joe");
List<String> filteredNames = names.stream().filter(name -> name.startsWith("J")).collect(Collectors.toList());
```

```
names.forEach(System.out::println);
```

```
filteredNames.forEach(System.out::println);
```

stream: Returns a sequential stream considering collection as its source.

filter: The filter method is used to select elements based on a specified criteria.

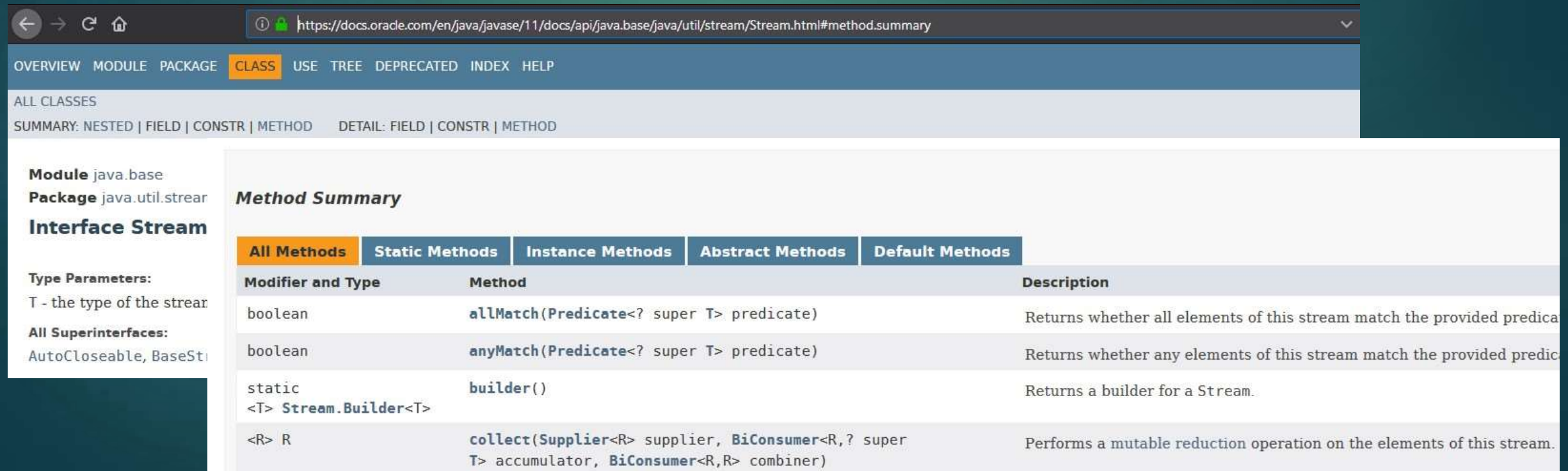
collect: The collect method is used to return the result of the intermediate operations performed on the stream.

Stream methods

In the next lectures, among other things, we will see more of these methods but it is good to know, that there are lots of useful methods for the Stream Interface.

You can look around at any time on the website of Oracle for these methods.

Java API Documentation → **https://docs.oracle.com/en/java/**



Java Programming: Step by Step from A to Z

Parallel Stream

Parallel Stream overview

The Stream API enables developers to create the parallel streams that can take advantage of multi-core architectures and enhance the performance of Java code. In a parallel stream, the operations are executed alongside each other and there are two ways to create a parallel stream.

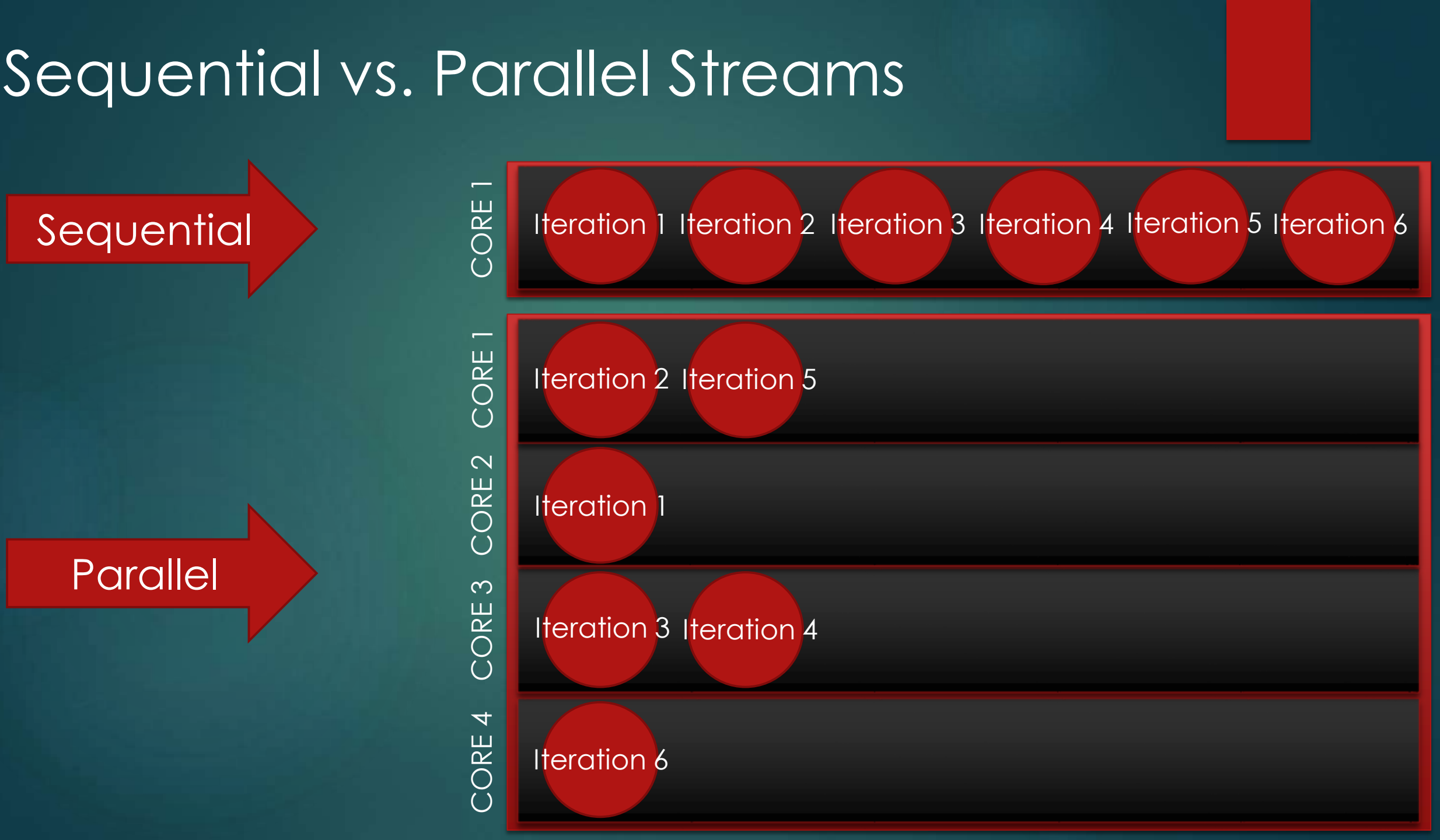
On a collection use parallelStream() method.

On a stream use parallel() method.

Use Parallel Streams only with stateless, associative and non-interfering operations.

A stateless operation is an operation in which the state of one element does not affect another element.
An associative operation is an operation in which the result is not affected by the order of operands.
A non-interfering operation is an operation in which data source is not affected.

Sequential vs. Parallel Streams



When to use Parallel Stream?

You need to consider parallel Stream if and only if:

You have a large dataset to process

You are actually suffering from performance issues

You need to make sure that all the shared resources between threads are synchronized properly otherwise it might produce unexpected results

Java Programming: Step by Step from A to Z

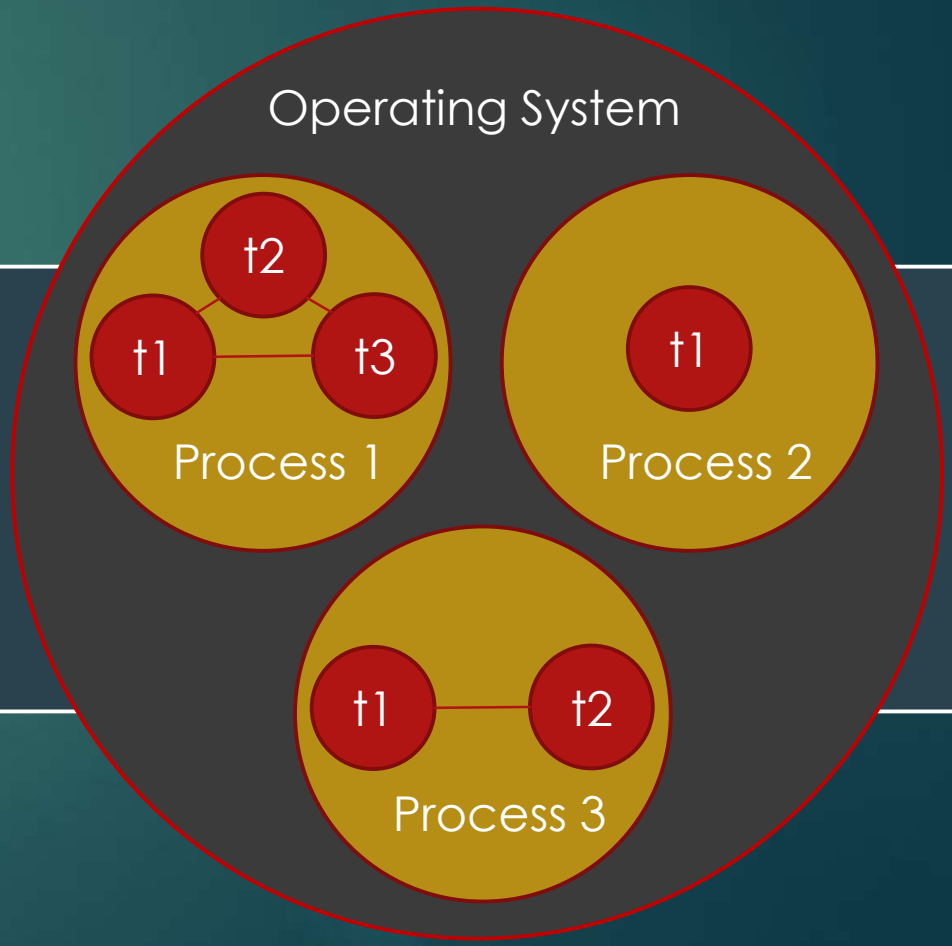
Multithreading

Multithreading overview

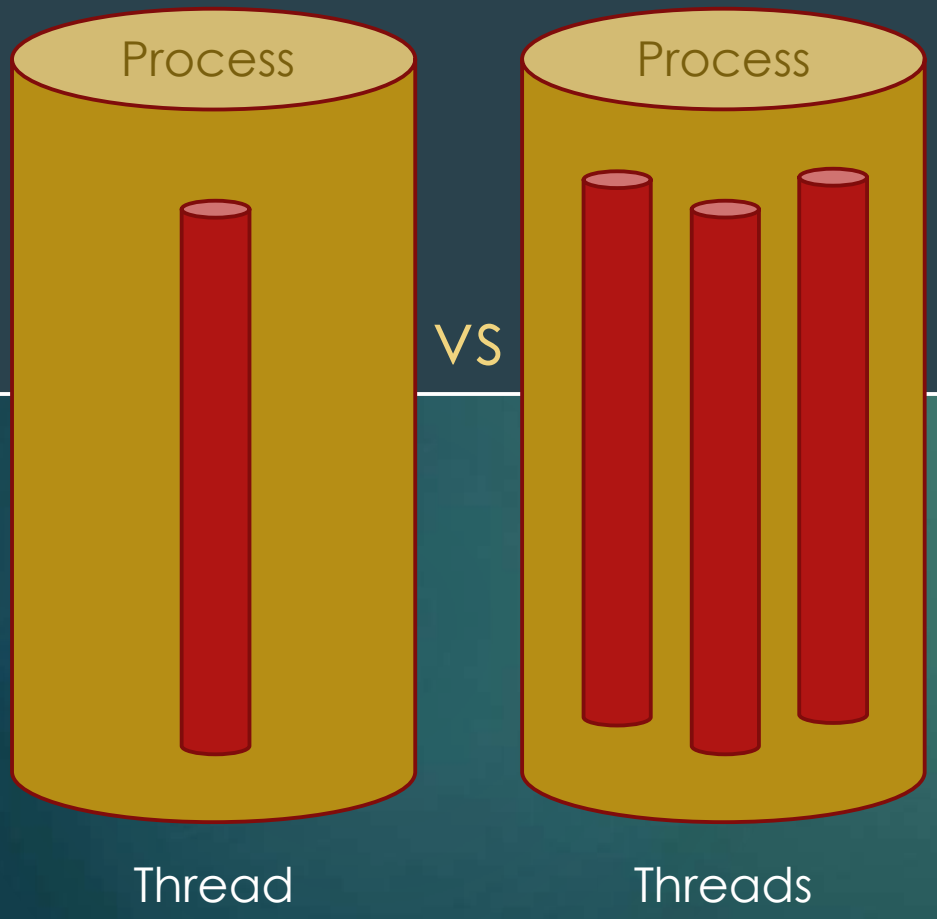
Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for making optimal use of the available resources specially when your computer has multiple CPUs. Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

t1

A thread is a lightweight sub-process, the smallest unit of processing and it's executed inside the process. Each of the threads can run in parallel and there is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

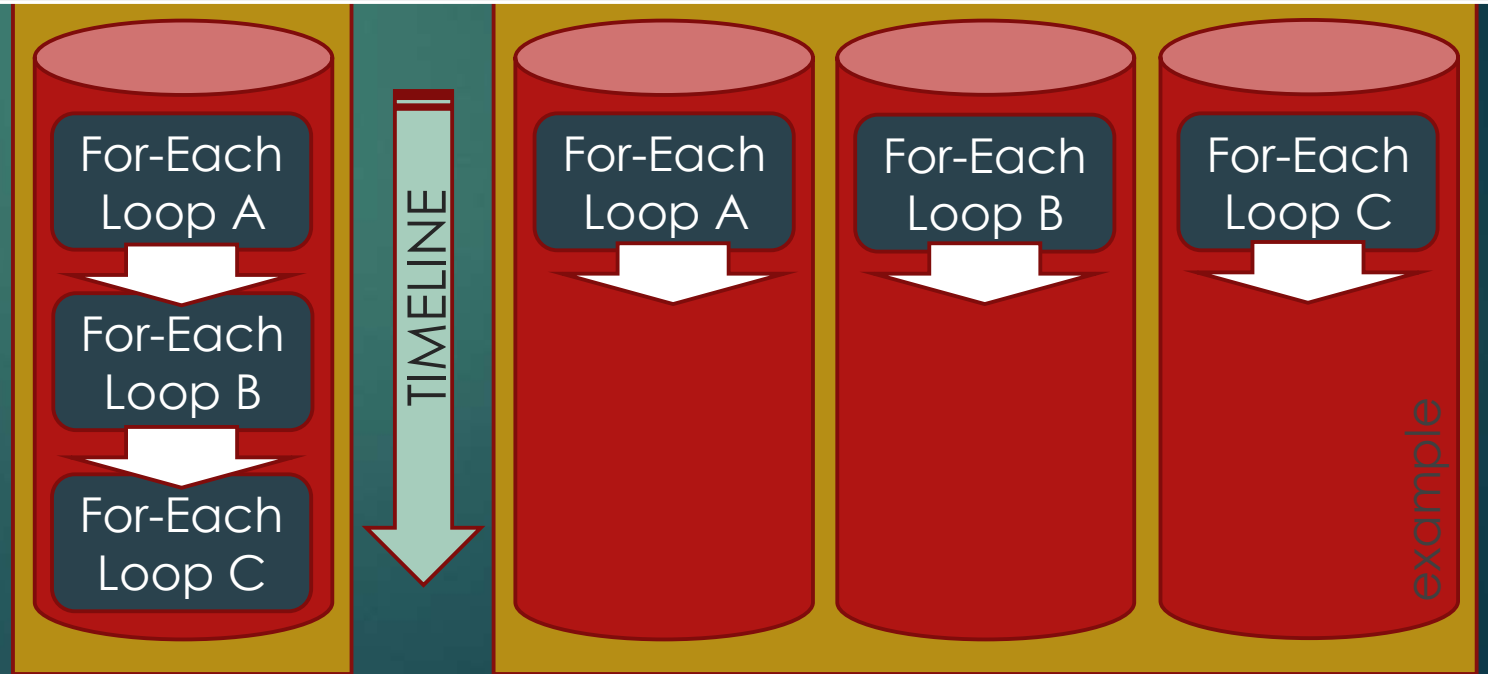


Advantages of Multithreading



Advantages of Multithreading

It allows multiple operations at once, as they are not dependent on each other, thus not blocking the user.
It saves time as multiple operations are possible.
They are independent thus making the functionality better.

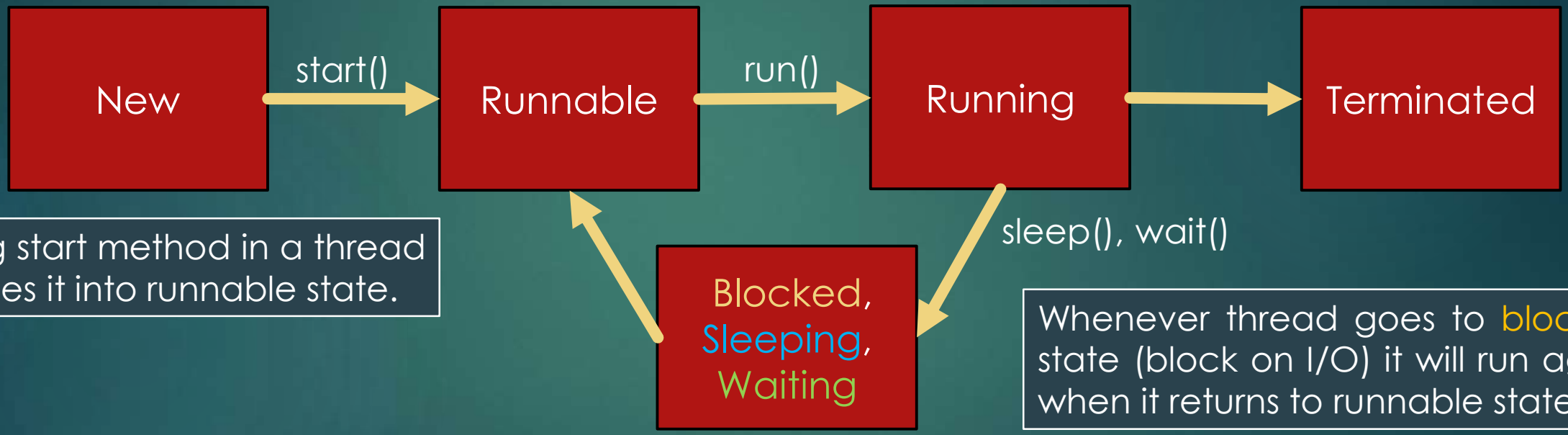


Life cycle of a thread in Java

Thread's constructor creates thread in new state.

Thread Scheduler of JVM runs the thread as soon as the processor becomes available.

Thread terminates as soon as run method ends.



Calling start method in a thread changes it into runnable state.

Whenever thread goes to blocked state (block on I/O) it will run again when it returns to runnable state.

If sleep() method is called, the thread will go to sleeping state, it changes into runnable state again after sleep time is over.

If the wait() method is called, the thread will go to waiting state, it goes to runnable state after it gets notification through the notify or notifyAll method.

How to create a thread

There are two ways to create Threads:

By implementing the Runnable Interface

```
class ThreadExample implements Runnable {
    public void run() {
        // more code
    }
}
```

```
Thread object = new Thread(new ThreadExample());
object.start();
```

By extending Thread class

```
class ThreadExample extends Thread {
    public void run() {
        // more code
    }
}
```

```
ThreadExample object = new ThreadExample();
object.start();
```

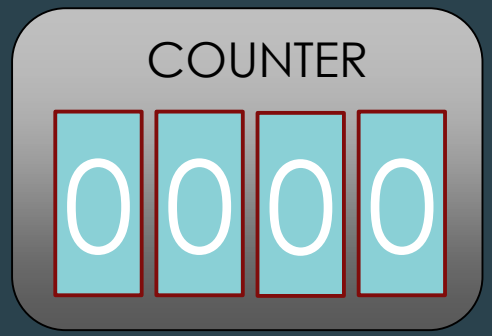
Java programing: Step by Step from A to Z

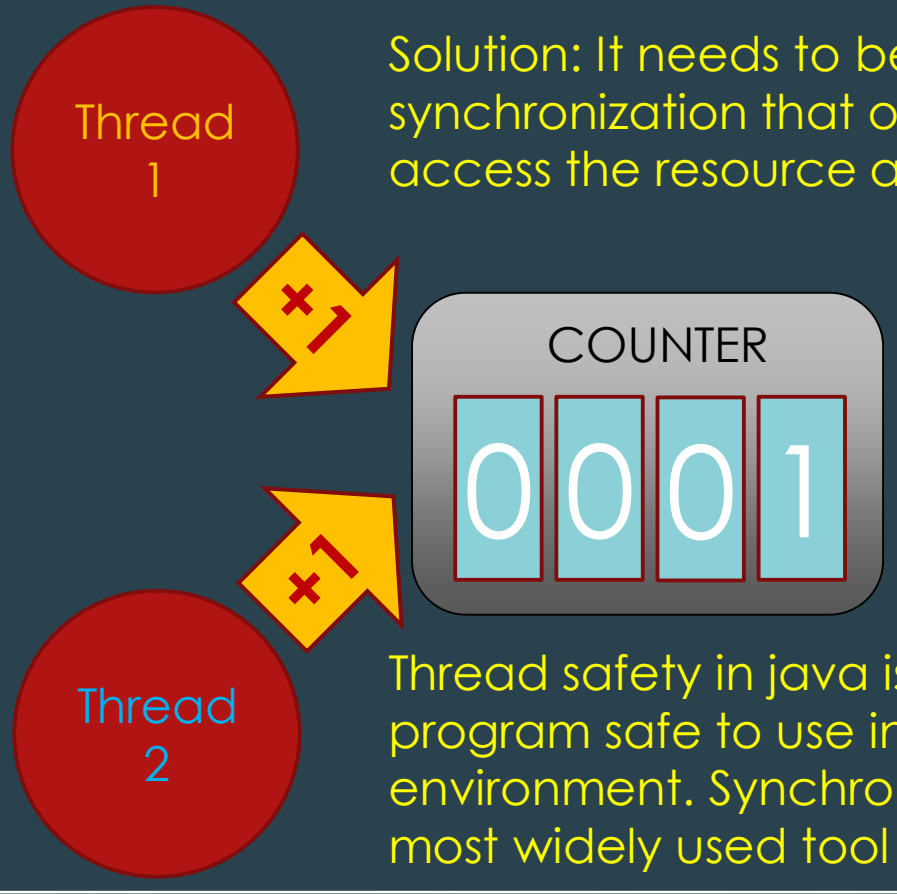
Synchronization

Synchronization overview

In a multi-threaded environment multiple threads created from the same Object share object variables and this can lead to data inconsistency when the threads are used to read and update the shared data. The reason for data inconsistency is because updating any field value is not an atomic process, it requires three steps; first to read the current value, second to do the necessary operations to get the updated value and third to assign the updated value to the field reference.

from Thread 1 perspective at current time
- read current value from the counter: 0
- command: increase value with 1
- set the counter's new value to 1 (0+1)





Solution: It needs to be made sure by some synchronization that only one thread can access the resource at a given point of time.

incorrect result value (1) is possible instead of the correct value (2)

from Thread 2 perspective at current time
- read current value from the counter: 0
- command: increase value with 1
- set the counter's new value to 1 (0+1)

Thread safety in java is the process to make our program safe to use in multithreaded environment. Synchronization is the easiest and most widely used tool for thread safety in java.

The Java Monitor

A monitor is mechanism to control concurrent access to an object. This prevents multiple threads accessing the monitored (synchronized) section at the same time. One will start, and the monitor will prevent the other from accessing the region before the previous one finishes. Only one thread at a time may hold a lock on a monitor.

1. Threads enter to acquire lock. (Entry Set)

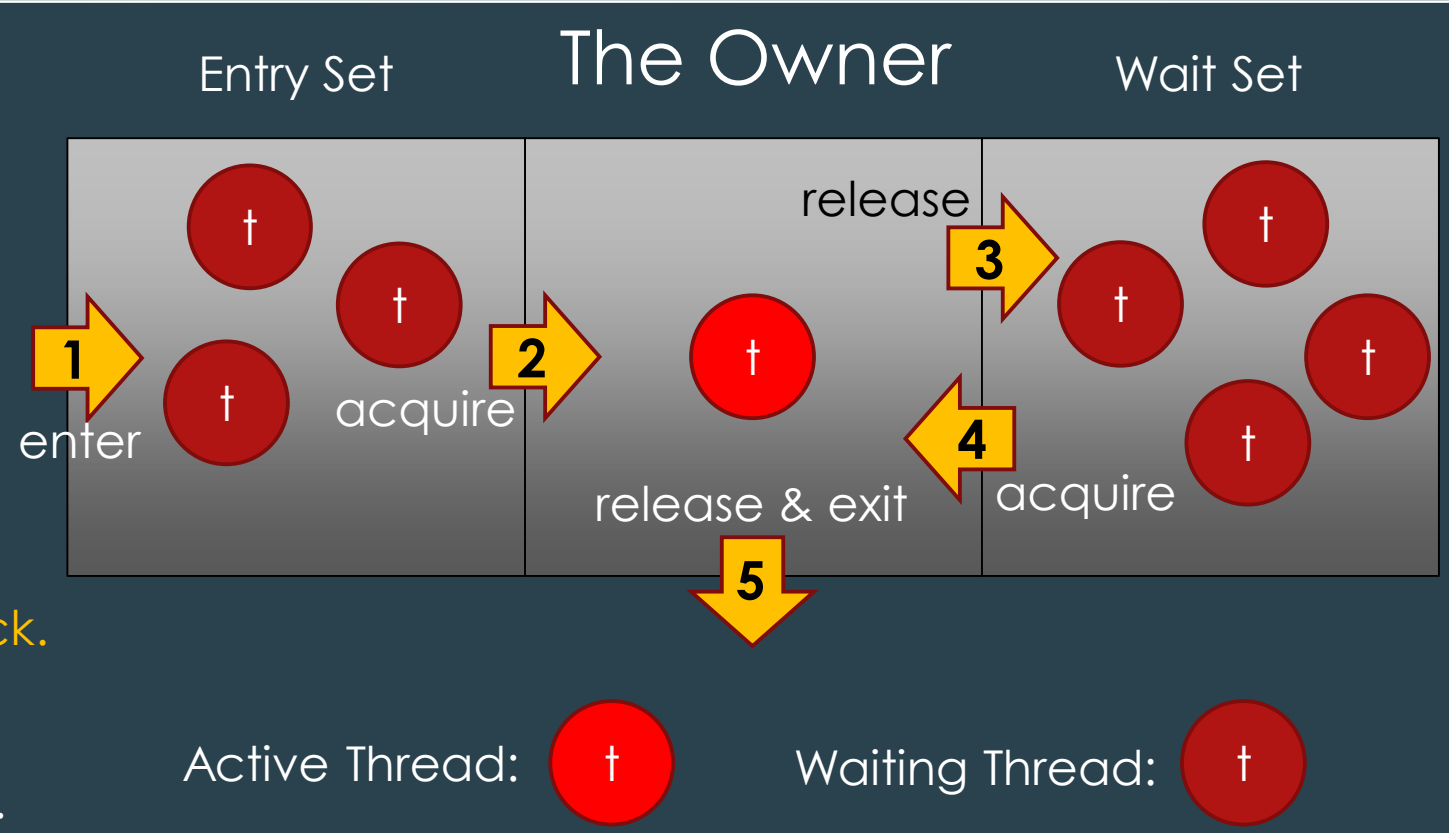
2. Lock is acquired by thread. (Owner)

3. Now thread goes to waiting state (Wait Set) if you call wait() method on the object. Otherwise it releases the lock and exits. (5) If you call notify() or notifyAll() method it wakes up the thread (threads).

4. Now the thread is available to acquire lock.

5. After completion of the task, thread releases the lock and exits the monitor state.

The operation of the Java Monitor



Java synchronized keyword

Synchronized in Java is an implementation of the Monitor concept and JVM guarantees that synchronized code will be executed by only one thread at a time. Java keyword synchronized is used to create this and internally it uses locks on Object or Class to make sure only one thread is executing the synchronized code.

There are different ways in java to create synchronized code:

by synchronized method

```
synchronized void methodName(int n){
    //synchronized method
}
```

by static synchronization

```
synchronized static void methodName(int n){
    //static synchronization
}
```

by synchronized block

```
synchronized(this){
    //synchronized block
}
```

In the next chapters we will see these in practice.