Java SE Basics

SDET Program

Outline

- Java Basic Concepts
  - Object, class, instance
  - Variables, data types
  - Generic, Wrapper Class
  - Reference & Dereference (.)
- Syntax
  - Component
  - Operator
  - Flow Control
- Some Keywords in Java
  - Structure definitions (class, interface, and enum)
  - Access Modifier (public, private, protected, and default)
  - Other Keywords (static and final)

Java Basic Concepts

- Object, class, instance
- Variables, data types
- Generic, Wrapper Class
- Reference & Dereference (.)

Objects

- Java is an OOP language
    - **O**bject-**O**riented **P**rogramming — printed as: **OOP**: **O**bject-**O**riented **P**rogramming
- Object = data + operations
    - **Fields**: data
    - **Methods**: operations on data
- For example, a car
    - Fields: brand, year, location, speed
    - Methods: `drive()`, `brake()`, `accelerate()`

- 4 Pillars of OOP will be discussed tomorrow

Variable

- A variable is a data container, a named memory location capable of storing data.
- Object variables refer to objects
    - Created with the **new** keyword.
        - `ArrayList<Integer> list = new ArrayList<>();`

- We can also store data in simple variables
    - Represent data only
    - No associated methods
    - **i.e., Primitive variables**

Primitive Variables

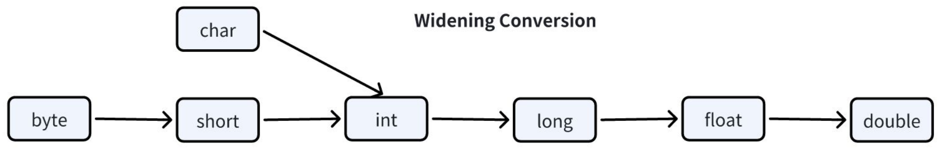
Data Type	Default Value	Size	Range
byte	0	8	-128 to 127 (inclusive)
short	0	16	-32,768 to 32767 (inclusive)
int	0	32	-2,147,483,648 to 2,147,483,647 (inclusive)
long	0L	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	0.0F	32	1.401298464324817e^{-45f} to 3.4028234663852886e^{38f}
double	0.0D	64	4.9406564584124654e^{-324} to 1.7976931348623157e^{308}
char	‘\u0000’	16	0 to 65535
boolean	false	Not Defined	`true` or `false`

Literals (Constants)

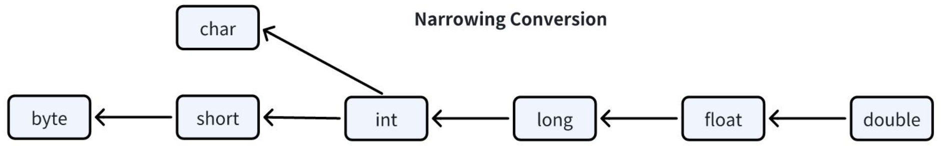
- Integral literals
  - int: 1, +5, -10
  - long: 200L, 1_000_000_000_000L
  - (optional)oct: 007, 09 (invalid)
  - (optional)hex: 0x7fff
  - (optional)bin: 0b1101_1000
- Floating-point literals
  - double: 1e9, 3.0, 3.0d
  - float: 3.0e-1f, 5F
- Char literals: ‘A’, ‘\u0000’
- String literals: “Hello World”
- Boolean literals: true, false
- Null literals: null

Conversion & Casting

- Conversion (widening casting)
  - Performed automatically (implicitly)
  - A smaller box can be placed in a bigger box and so on



- Casting (narrowing casting)
  - A bigger box has to be placed in a smaller box
  - Casting is **not implicit** in nature
  - `int i = (int)(8.0 / 3.0)`
  - Casting will lose precision



Conversion & Casting

```java
int i = 5 / 2;

int m = (int)(5.0 / 2.0);
double n = (int)5.0 / 2.0;

int k = (int) 2147483648.0f; //Integer.MAX_VALUE = 2147483647

char c = (char)75; //ASCII: 75 → K
```

Creating Variables

- Declaration: defining the type and name of a variable or object
    - `int a;`
    - `ArrayList<Integer> list;`
- Initialization: assigning an initial value to a variable
    - `a = 100;`
    - `list = new ArrayList<Integer>();`
- Instantiation: creating an object using the **`new`** keyword
    - Combining declaration and initialization
        - `int a = 100;`
        - `List<Integer> list = new ArrayList<>();`

Wrapper Class

- A wrapper class is a class whose object wraps or contains a primitive data type
- When we create an object from a wrapper class
    - It contains a field
        - In this field, we can store a corresponding primitive data type
            - `Byte`
            - `Short`
            - `Integer`
            - `Long`
            - `Double`
            - `Character`
            - `Boolean`
- Usage: `ArrayList<Integer>`

Class & Instance

- A class is a blueprint or template for creating objects
- A class definition contains the block of code that includes:
    - Fields
    - Constructors
    - Getters
    - Setters
    - Methods

```java
class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  public void hi() {
    System.out.println("Hi, my name is " + name);
  }
}
```

Class & Instance

- An instance (object) is a concrete occurrence of classes created in memory
- How to create a new instance?
    - Instantiation
        - `Person p = new Person();`

Generics

- “Type as parameter”
  - Write once, use in different cases

- For example, if we want to design a class called Box
  - We can put something into the box, or get it out
  - How would you design the Box class?

Generics - Example

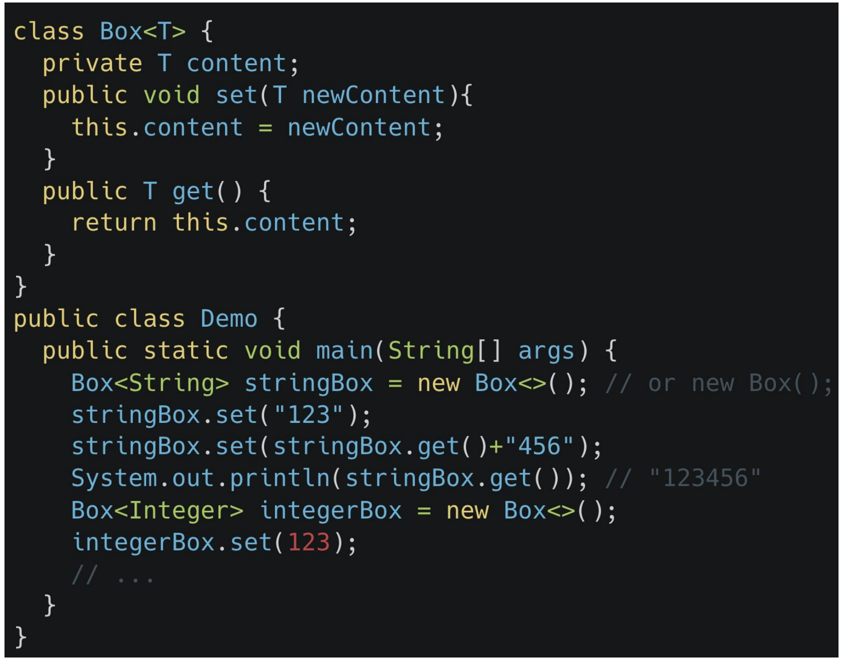
- Design a box class: we can put *something* into the box, or get it out
- Data + Operations
    - Data: an object
    - Operations: set(), get()

```java
class Box {
  private Object content;
  public void set(Object newContent){
    this.content = newContent;
  }
  public Object get() {
    return this.content;
  }
}
public class Demo {
  public static void main(String[] args) {
    Box b = new Box();
    b.set("123");
    b.set(new Box());
  }
}
```

```java
class StringBox {
  private String content;
  public void set(String newContent){
    this.content = newContent;
  }
  public String get() {
    return this.content;
  }
}
public class Demo {
  public static void main(String[] args) {
    Box b = new Box();
    b.set("123");
    b.set(b.get()+"456");
    System.out.println(b.get()); // "123456"
  }
}
```

Generic - Example

- Both of the implementations have its drawback
- Here is where generics comes into play



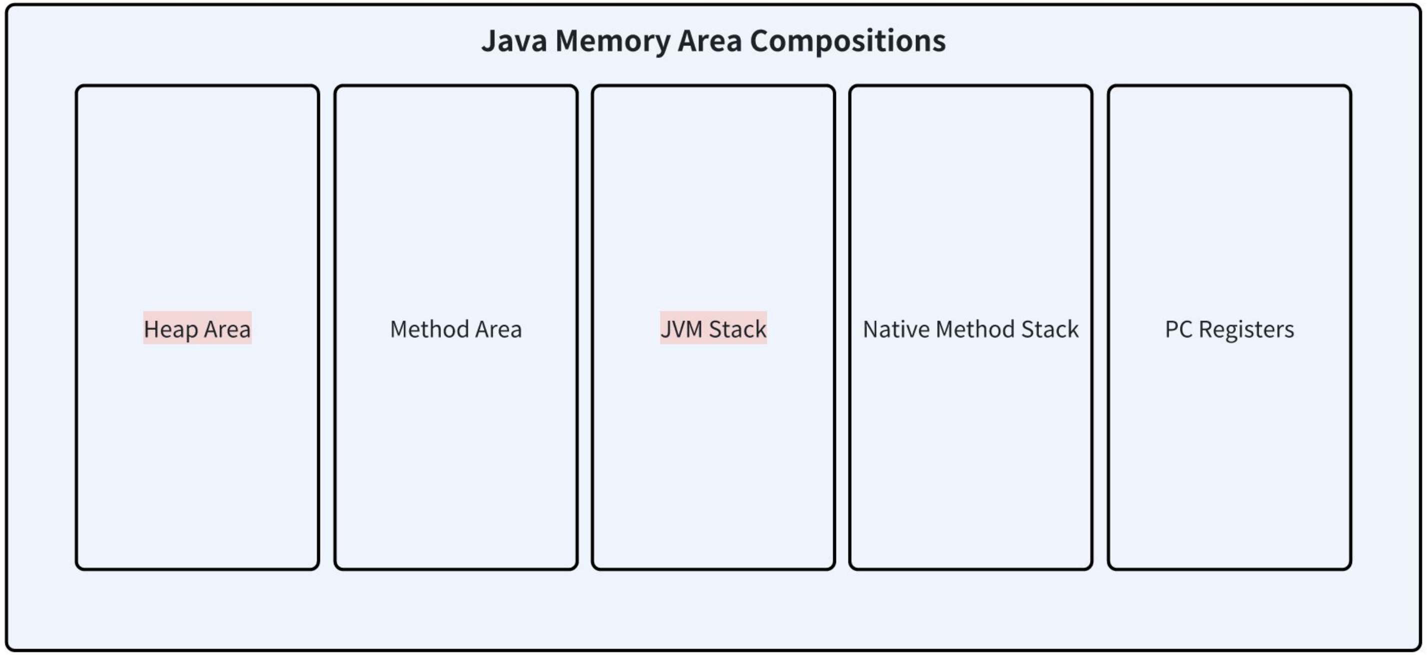
```java
class Box<T> {
  private T content;
  public void set(T newContent){
    this.content = newContent;
  }
  public T get() {
    return this.content;
  }
}
public class Demo {
  public static void main(String[] args) {
    Box<String> stringBox = new Box<>(); // or new Box();
    stringBox.set("123");
    stringBox.set(stringBox.get()+"456");
    System.out.println(stringBox.get()); // "123456"
    Box<Integer> integerBox = new Box<>();
    integerBox.set(123);
    // ...
  }
}
```

String

- A String is a sequence of characters
- In Java, String objects are immutable
    - Once they are created, they cannot be changed
- String Pool
    - A collection of Strings which are stored in the heap memory

Java Memory

- Runtime Data

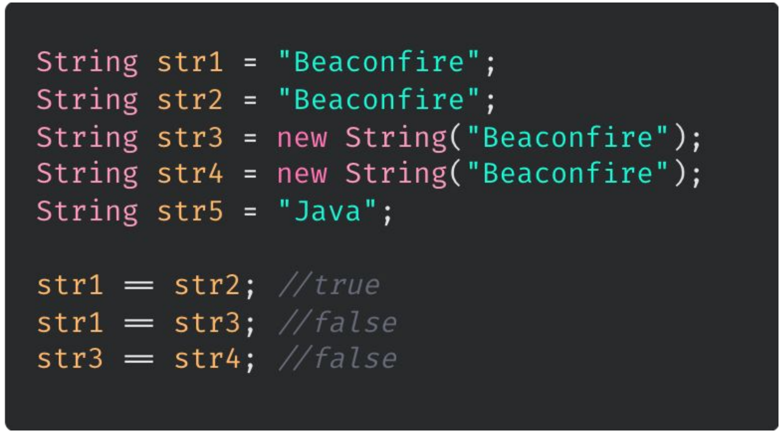


Heap & Stack

- Heap
  - Dynamic memory allocation for **Objects**
    - Everything we **new**
  - Objects can be globally accessed
- Stack
  - Used for static memory allocation and the execution of a thread
    - Contains primitive values declared in methods
    - References to objects that are in the heap

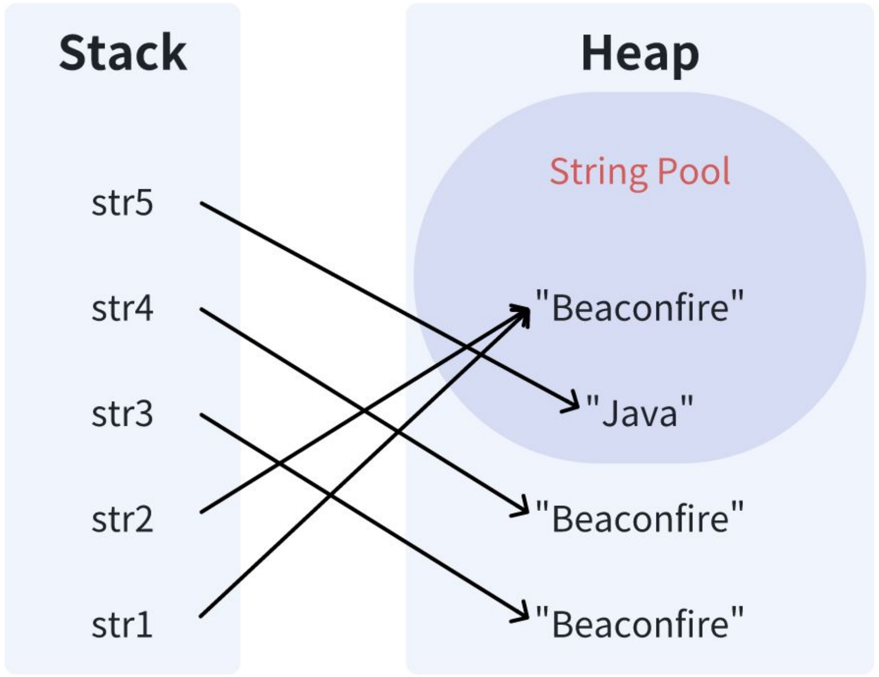
Stack Memory & Heap Space

- String Pool
  - Save memory
  - Reusability
    - No need to create a new String if one already exists



```java
String str1 = "Beaconfire";
String str2 = "Beaconfire";
String str3 = new String("Beaconfire");
String str4 = new String("Beaconfire");
String str5 = "Java";

str1 == str2; //true
str1 == str3; //false
str3 == str4; //false
```

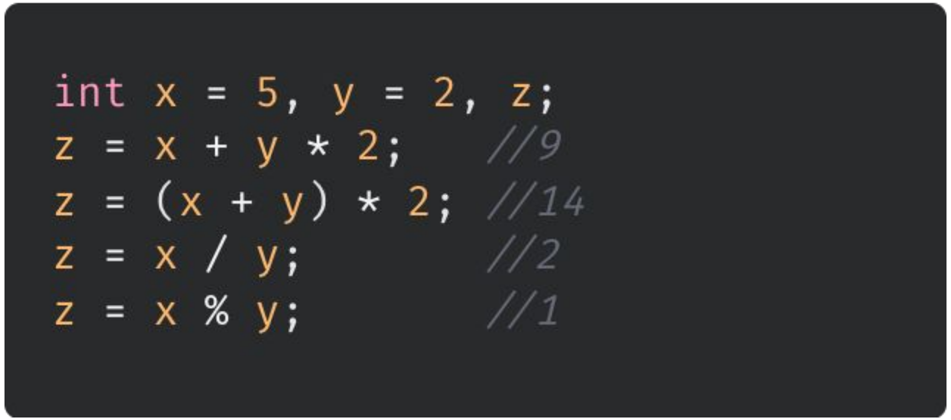


Comment

- Java supports 3 types of comments
  - `/* text */`
    - The compiler ignores everything from /* to */
  - `// text`
    - The compiler ignores everything from // to the end of the line
  - `/** Documentation */`
    - This is a documentation comment and in general it's called doc comment
    - The JDK javadoc tool uses doc comments when preparing automatically generated documentation

Operator

- Arithmetic operations in Java
  - Precedence: `(*, /, %) > (+, -)`
  - Parentheses ( )
    - Evaluate the innermost parenthesized expression first, and work your way out through the levels of nesting
  - No { } or [ ] in terms of arithmetic operations in Java



```
int x = 5, y = 2, z;
z = x + y * 2;    //9
z = (x + y) * 2;  //14
z = x / y;        //2
z = x % y;        //1
```

Assignment Operators

Operator	Example	Meaning
`+=`	`x += 1;`	`x = x + 1;`
`-=`	`x -= 1;`	`x = x - 1;`
`*=`	`x *= 5;`	`x = x * 5;`
`/=`	`x /= 2;`	`x = x / 2;`
`%=`	`x %= 10;`	`x = x % 10;`

Increment & Decrement Operators

- Increment
  - Pre-Increment: `y = ++x;`
  - Post-Increment: `y = x++;`
- Decrement
  - Pre-Decrement: `y = --x;`
  - Post-Decrement: `y = x--;`

Expression	Initial Value of x	Final Value of x	Final Value of y
`y = ++x;`	4	5	5
`y = x++;`	4	5	4
`y = --x;`	4	3	3
`y = x--;`	4	3	4

Logical Operator

Operator	Description	Example	Result
`&&`	Logical AND: true if both operands are true, otherwise false	`true && false`	`false`
`||`	Logical OR: true if at least one operand is true	`true || false`	`true`
`!`	Logical NOT: inverts the value of a boolean	`!true`	`false`
`^`	Logical XOR: true if operands are different	`true ^ false`	`true`
`&`	Bitwise AND: can also be used as logical AND	`true & false`	`false`
`|`	Bitwise OR: can also be used as logical OR	`true | false`	`true`

Short-Circuit: `if (array == null || array.length == 0) {}`

Relational Operators

Operator	Result
==	Equal To
! =	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

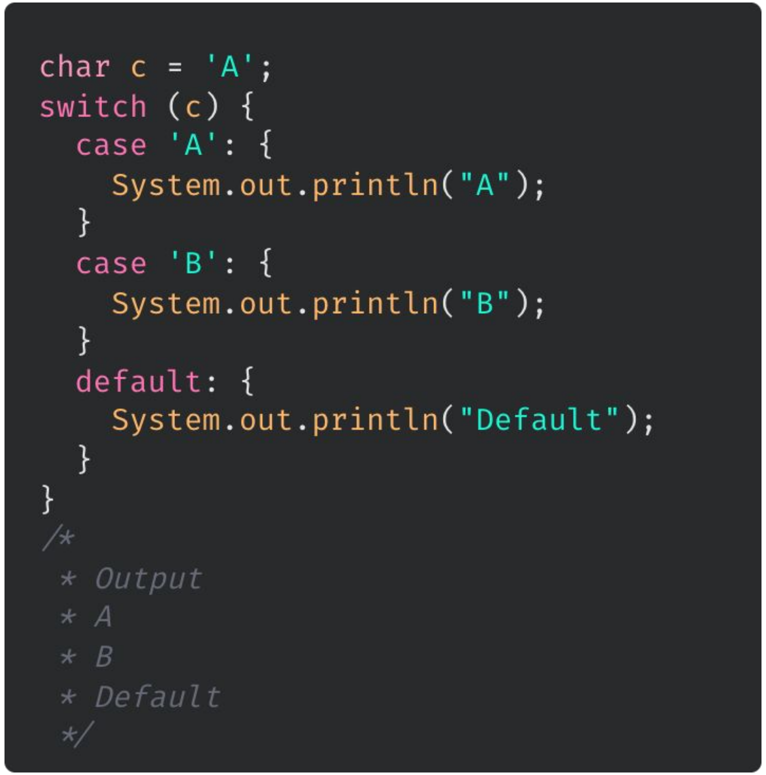
Member Access Operator

- Dot `.`
  - Used to access members (methods, variables) of a class or an object
  - Also known as dereference operator
- `NullPointerException`
  - When dereference null
  - `array.length`

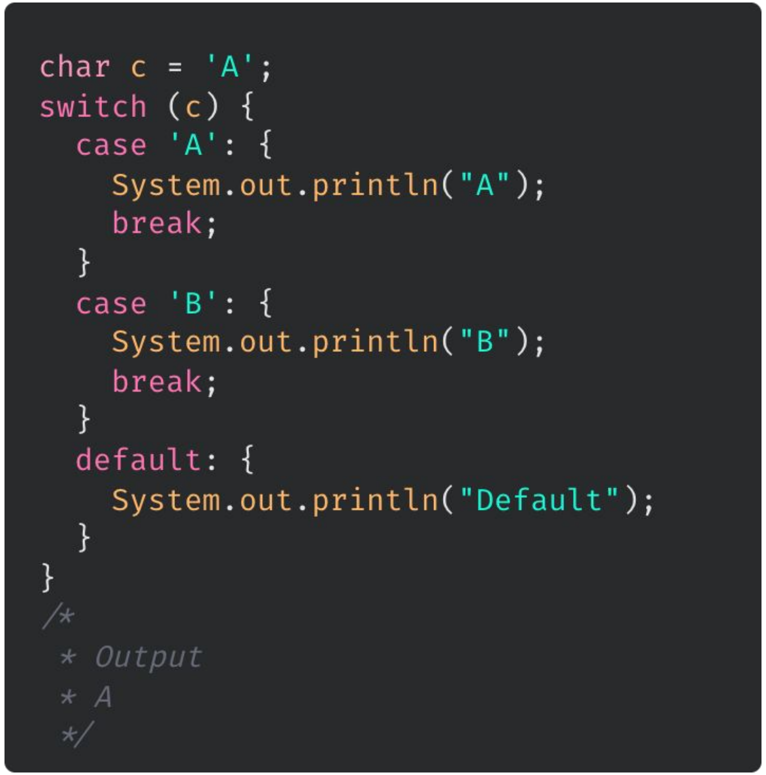
Flow Control

- Selection Statements
  - `if`
  - `switch`
- Iteration Statements
  - `while`
  - `do-while`
  - `for`
  - Nested loops
- Jump Statements
  - `break`
  - `continue`
  - `return`

Switch

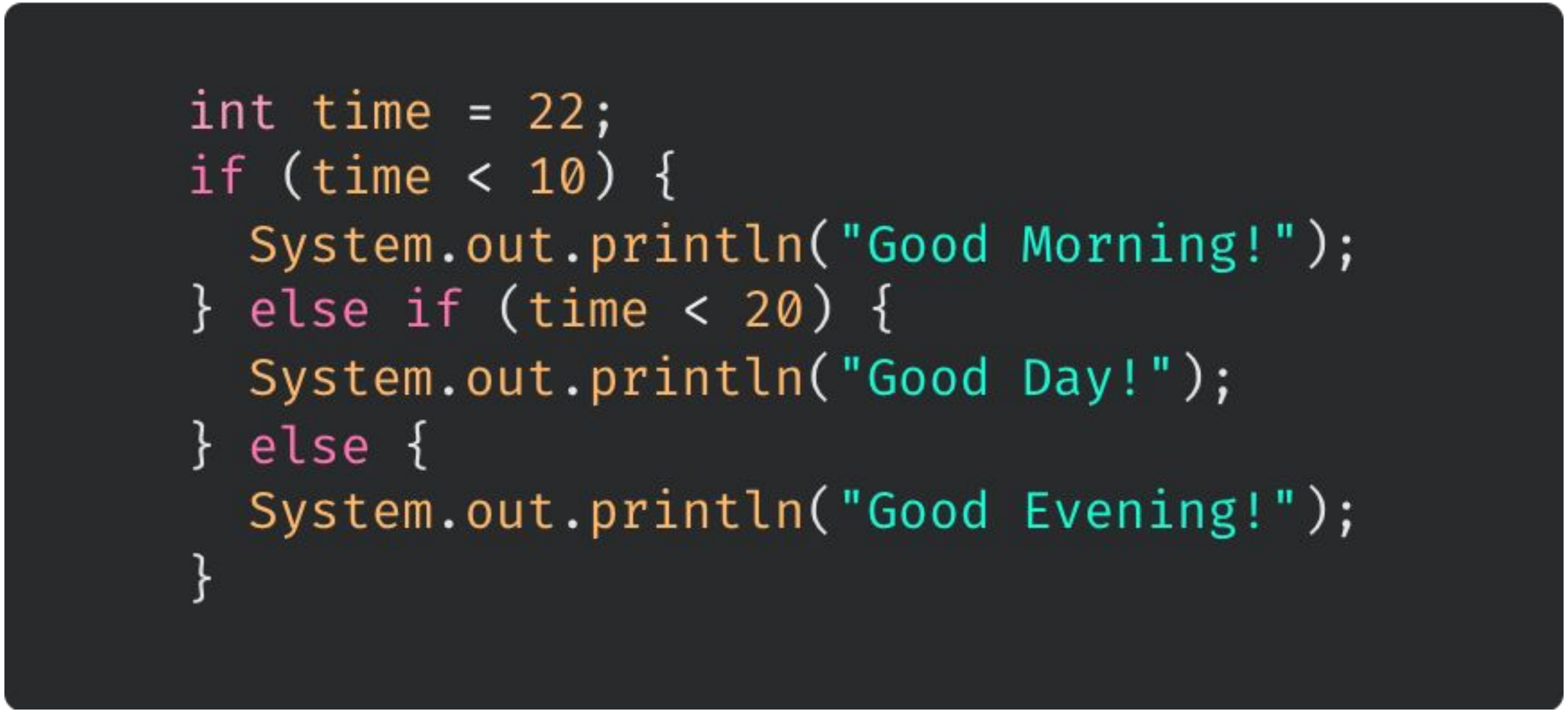


```java
char c = 'A';
switch (c) {
  case 'A': {
    System.out.println("A");
  }
  case 'B': {
    System.out.println("B");
  }
  default: {
    System.out.println("Default");
  }
}
/*
 * Output
 * A
 * B
 * Default
 */
```



```java
char c = 'A';
switch (c) {
  case 'A': {
    System.out.println("A");
    break;
  }
  case 'B': {
    System.out.println("B");
    break;
  }
  default: {
    System.out.println("Default");
  }
}
/*
 * Output
 * A
 */
```

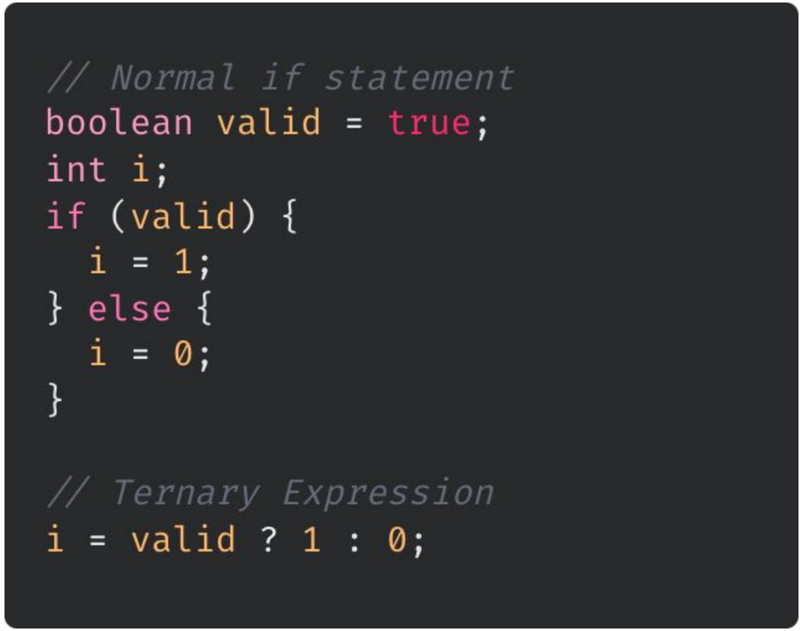
If



```java
int time = 22;
if (time < 10) {
  System.out.println("Good Morning!");
} else if (time < 20) {
  System.out.println("Good Day!");
} else {
  System.out.println("Good Evening!");
}
```

Ternary Expression

- `variable = (condition) ? expressionTrue : expressionFalse;`



```
// Normal if statement
boolean valid = true;
int i;
if (valid) {
  i = 1;
} else {
  i = 0;
}

// Ternary Expression
i = valid ? 1 : 0;
```

Iteration Statement

- While

```
// Syntax
while (condition) {
  // statements to keep executing while condition is true
}

// Example
while (n > 100) {
  n = n + 1;
}
```

Iteration Statement

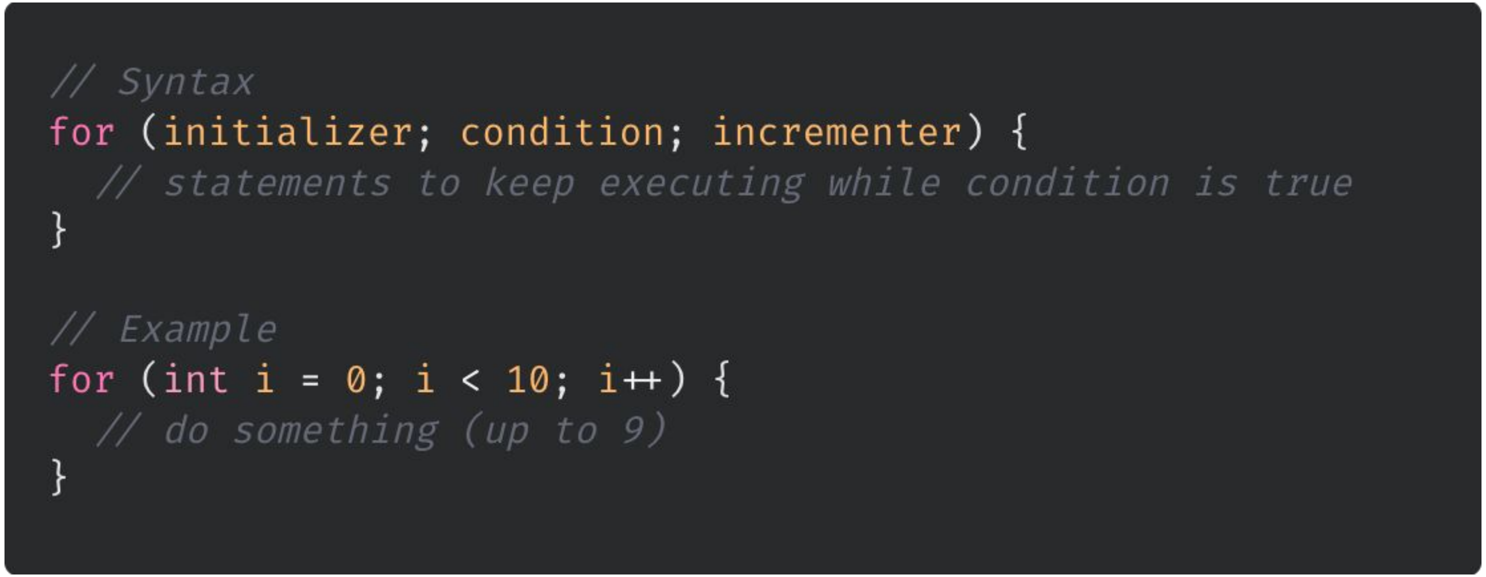
- Do-While

```
// Syntax
do {
  // Statements to keep executing while condition is true
  // It first executes the statement and then evaluates the condition
} while (condition);

// Example
do {
  System.out.println("n = " + n);
  n--;
} while (n > 0);
```

Iteration Statement

- For

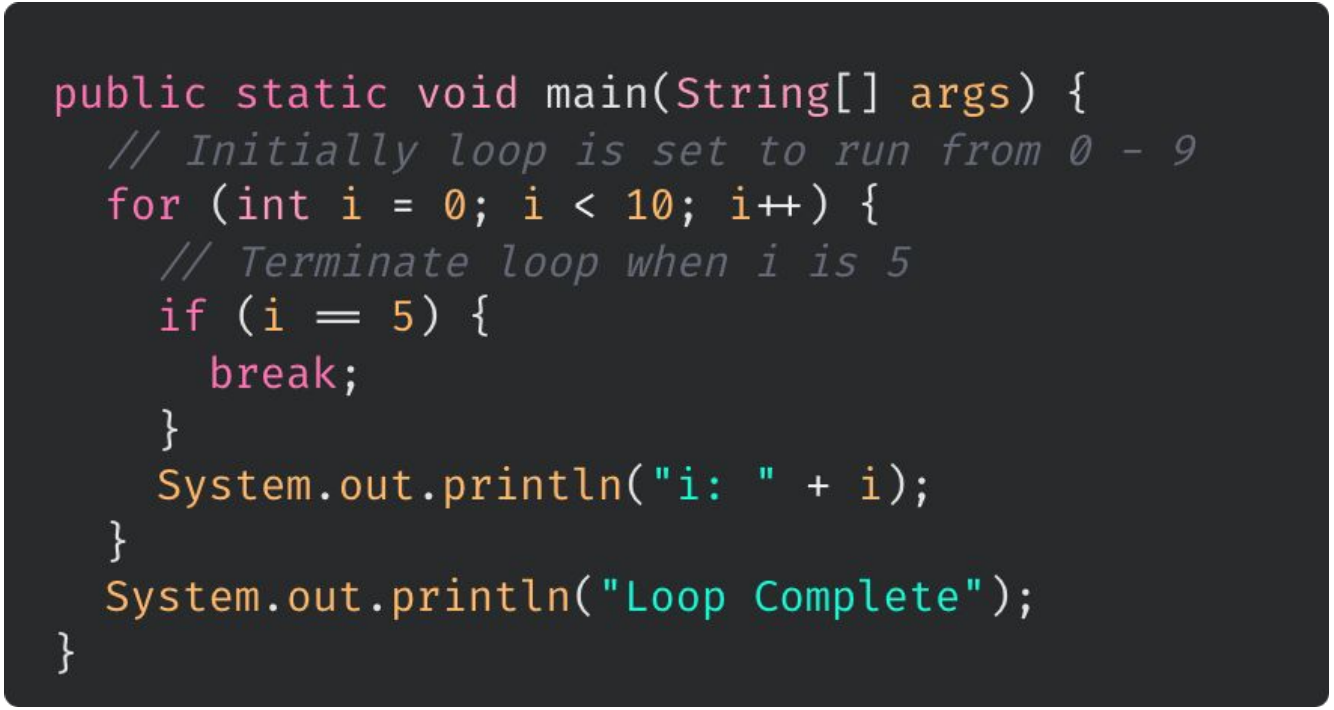


```
// Syntax
for (initializer; condition; incrementer) {
  // statements to keep executing while condition is true
}

// Example
for (int i = 0; i < 10; i++) {
  // do something (up to 9)
}
```

Jump Statement

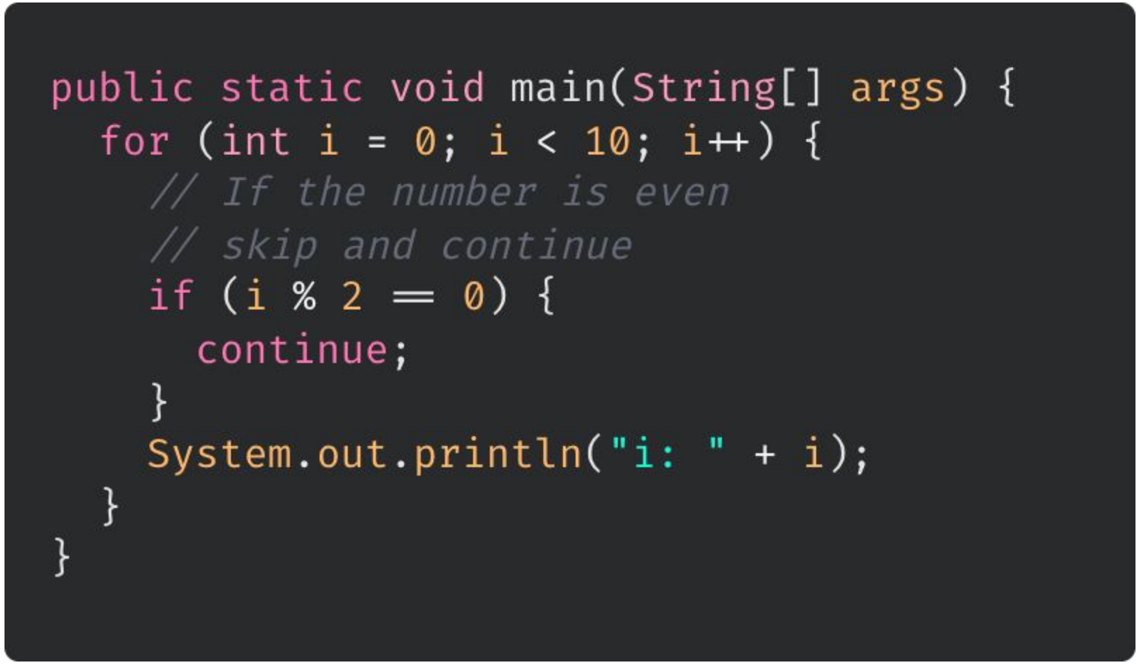
- Break



```java
public static void main(String[] args) {
  // Initially loop is set to run from 0 - 9
  for (int i = 0; i < 10; i++) {
    // Terminate loop when i is 5
    if (i == 5) {
      break;
    }
    System.out.println("i: " + i);
  }
  System.out.println("Loop Complete");
}
```

Jump Statement

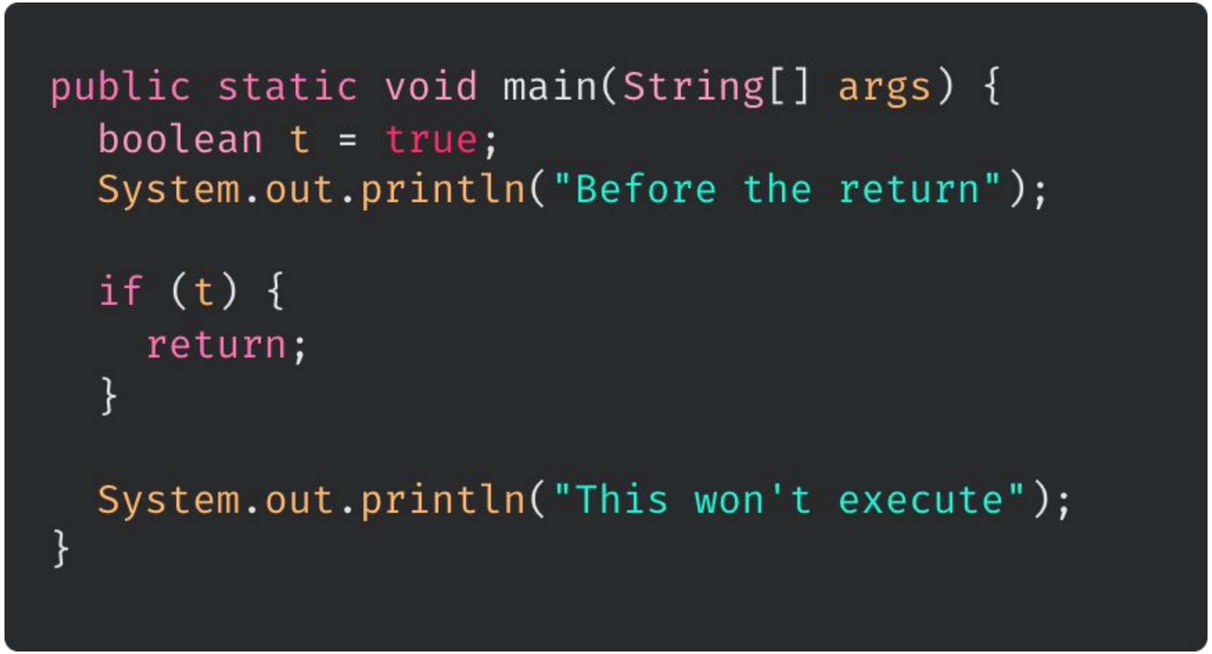
- Continue



```
public static void main(String[] args) {
  for (int i = 0; i < 10; i++) {
    // If the number is even
    // skip and continue
    if (i % 2 == 0) {
      continue;
    }
    System.out.println("i: " + i);
  }
}
```

Jump Statement

- Return



```java
public static void main(String[] args) {
  boolean t = true;
  System.out.println("Before the return");

  if (t) {
    return;
  }

  System.out.println("This won't execute");
}
```

Keywords

- Structure Definitions
  - Class
  - Interface
  - Enum
- Modifier
- Static
- Final

Modifier

- Access Modifier
    - Specify accessibility of:
        - Field
        - Method
        - Constructor
        - Class
- Non-Access Modifier
    - Static
    - Abstract
    - Final

Access Modifiers

- Determine access rights for the class and its members
  - Public
  - Private
  - Protected
  - Default

Access Modifier Scope

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	X
Default	✓	✓	X	X
Private	✓	X	X	X

Non-Access Modifiers

- **Static**
- Final
- Abstract

Non-Access Modifiers

- Static
  - Class
  - Compile time or early binding
- Non-Static
  - Object
  - Runtime or dynamic binding

Class Variable vs. Instance Variable

- What’s the difference between the following statements?
  - `public int x;`
  - `public static int x;`

Class Variable vs. Instance Variable

```java
public class VariableDemo {
  static int staticVariable = 0;
  int instanceVariable = 0;

  public static void main(String[] args) {
    System.out.println(staticVariable); // 0

    // instance variable can only be accessed through Object reference
    System.out.println(instanceVariable);

    VariableDemo object1 = new VariableDemo();
    System.out.println(object1.instanceVariable);

    object1.staticVariable = 1;
    object1.instanceVariable = 1;

    VariableDemo object2 = new VariableDemo();

    // each object has its own copy of instance variable
    System.out.println(object2.instanceVariable);

    // common to all object of a class
    System.out.println(object2.staticVariable);
  }
}
```

Class Variable vs. Instance Variable

Class Variables	Instance Variables
Class variables are declared with keyword static	Instance variables are declared without static keyword
Class variables are common to all instances of a class. These variables are shared between the objects of a class	Instance variables are not shared between the objects of a class. Each instance will have their own copy of instance variables
As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another	As each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object
Class variables can be accessed using either class name or object reference	Instance variables can be accessed only through object reference

Static Method vs. Non-Static Method

- What is the difference between the following statements?
    - `public int getX();`
    - `public static int getX();`

Static Method vs. Non-Static Method

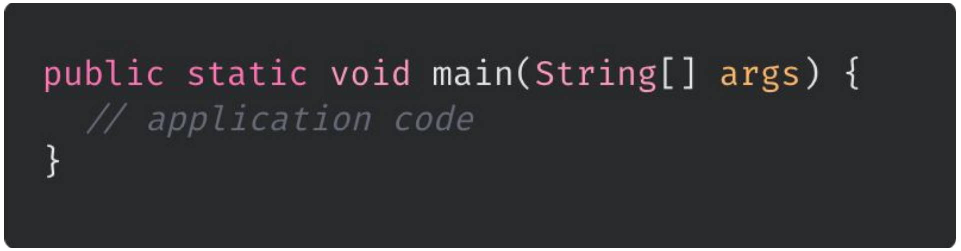
	Static Method	Non-Static Method
Access instance variables?	X	✓
Access static class variables?	✓	✓
Call static class methods?	✓	✓
Call non-static instance methods?	X	✓
Use the object reference this?	X	✓

Final

- The final keyword can be used to make a class, method or variable immutable
- Final variable
    - Once a final variable is assigned a value, it becomes a constant and can no longer be changed
- Final method
    - Once a method is made final, it cannot be overridden
- Final class
    - Once a class is made final, it cannot be extended

Main Method

- `main()` is a method
  - `public`: main can be called from outside the class
  - `static`: main can be called by the JVM without instantiating an object
  - `void`: main does not return a value



```java
public static void main(String[] args) {
  // application code
}
```

Deep Copy vs. Shallow Copy

- In OOP languages like Java, object copying is creating an exact copy of an existing object
- **Shallow Copy**
  - If you modify the copied object
    - Original object will be modified as well
  - The shallow copy points to the same reference as the original object
  - To perform shallow copy, we use the default `clone()` method
- **Deep Copy**
  - If you modify the copied object
    - Original object will not be modified
  - To do a deep copy, we use the new keyword and copy over the values one by one

Deep Copy vs. Shallow Copy

```java
public class ShallowCopy {
  private int[] copy;

  // Shallow copying an object using the default
cloning process
  public ShallowCopy(int[] copy) {
    this.copy = copy;
  }

  public void printArray() {
    System.out.println(Arrays.toString(copy));
  }
}

int[] original = { 1, 6, 9 };
ShallowCopy shallowCopy = new
ShallowCopy(original);
shallowCopy.printArray(); // {1, 6, 9}
original[0] = 3;
shallowCopy.printArray(); // {3, 6, 9}
```

```java
public class DeepCopy {
  private int[] copy;

  // Modified constructor to make a deep copy
  public DeepCopy(int[] original) {
    copy = new int[original.length];
    for (int i = 0; i < original.length; i++) {
      copy[i] = original[i];
    }
  }

  public void printArray() {
    System.out.println(Arrays.toString(copy));
  }
}

int[] original = { 1, 6, 9 };
DeepCopy deepCopy = new DeepCopy(original);
deepCopy.printArray(); // {1, 6, 9}
original[0] = 3;
deepCopy.printArray(); // {1, 6, 9}
```

Thank You!