Introduction to Java Programming

Brief History of Java

- Java was created by Sun Microsystems in **May 1995.**
- A team - **that was called the Green Team** - was assembled and lead by **James Gosling**.
- Platform and OS **Independent** Language.
- **Free** License; cost of development is brought to a minimum.

Brief History of Java

From mobile phones to handheld devices, games
and navigation systems to e-business solutions, **Java is everywhere!**

Java can be used to create:

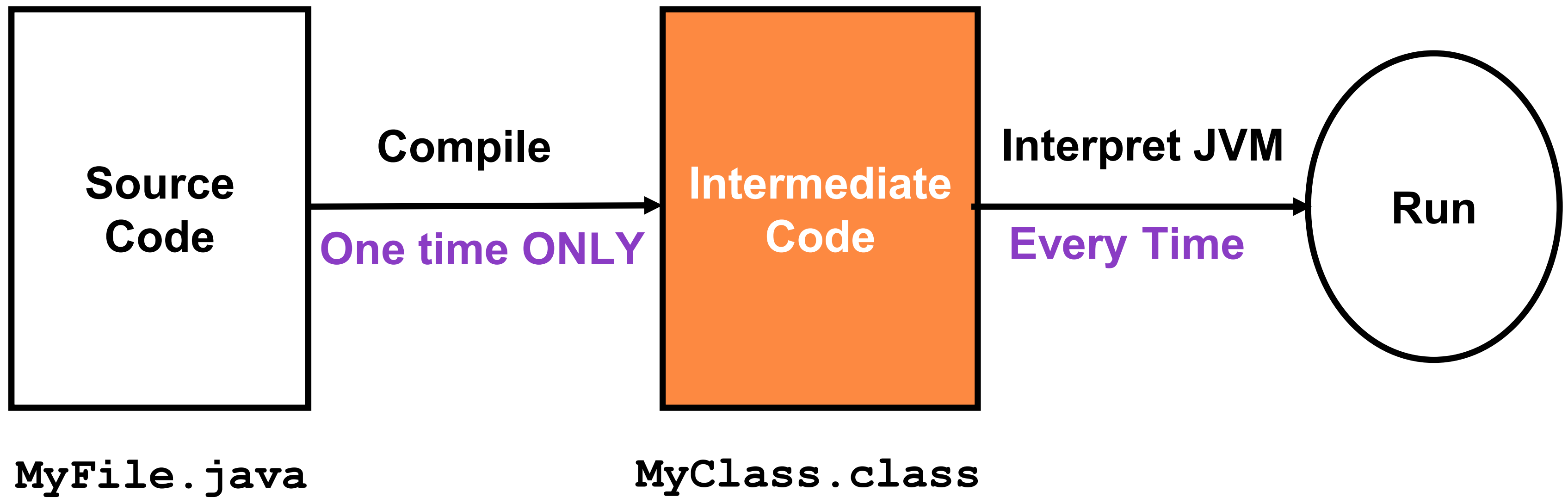
- Desktop Applications,
- Web Applications,
- Enterprise Applications,
- Mobile Applications,
- Smart Card Applications.
- Embedded Applications (Sun SPOT- Raspberry Pi)

Java Features

- Java is easy to learn!
- Syntax of C++
- Dynamic Memory Management (Garbage Collection)
- No pointers

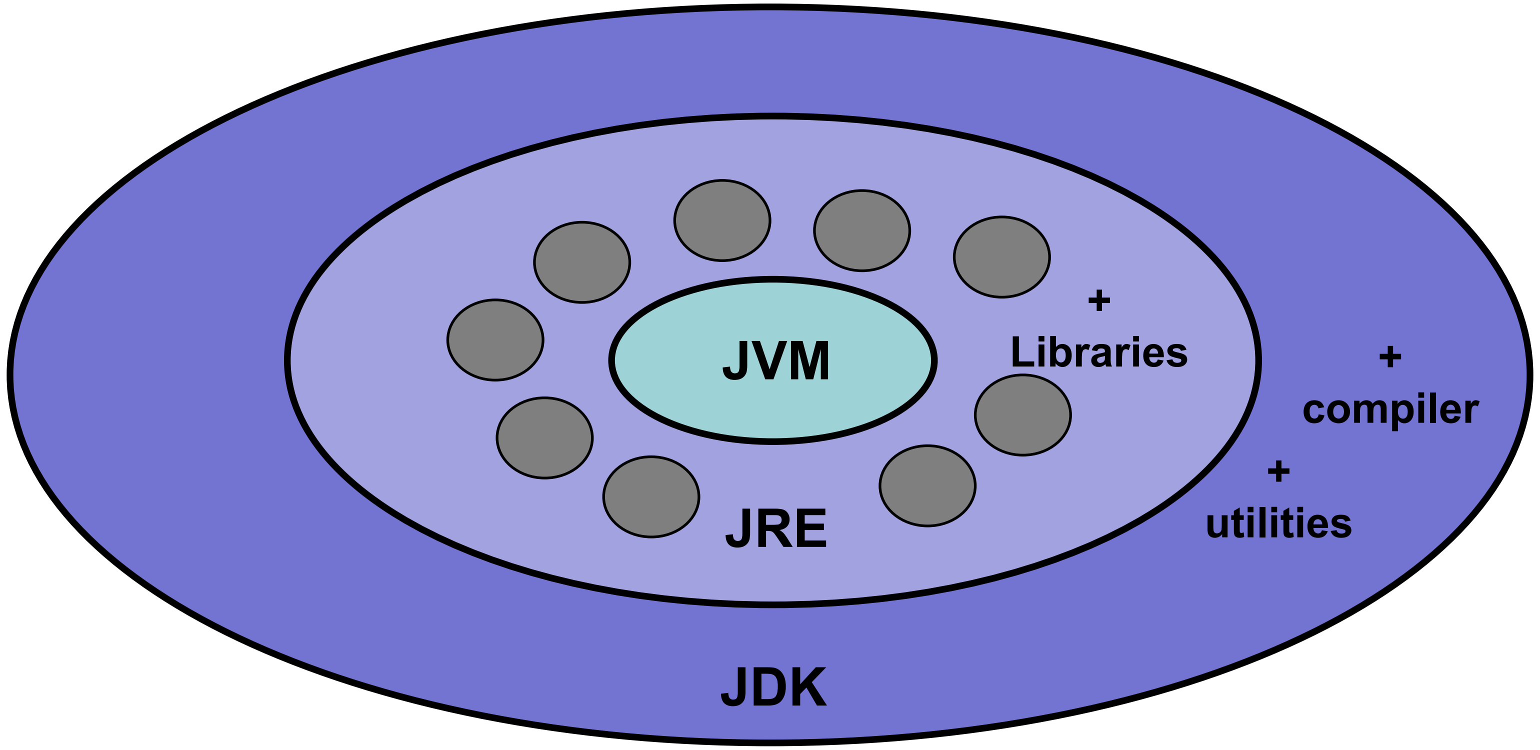
Java Features cont'd

- Java is both, compiled and interpreted



Java Features Cont'd

- Java depends on dynamic linking of libraries



Java development Kit (JDK)

Java Features cont'd

- Java is fully Object Oriented
  - Made up of Classes.
  - No multiple Inheritance.
- Java is a multithreaded language
  - You can create programs that run multiple threads of execution in parallel.
    - Ex: GUI thread, Event Handling thread, GC thread
- Java is networked
  - Predefined classes are available to simplify network programming through Sockets(TCP-UDP)

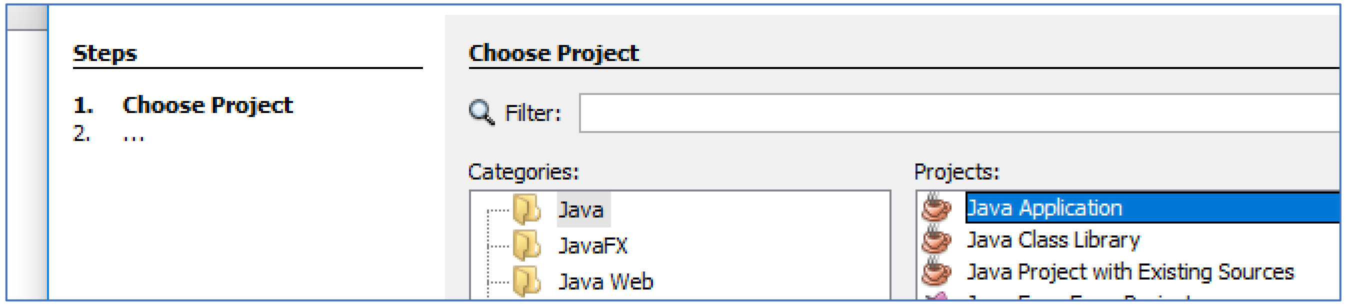
Preparing your environment

1. Download and Install the JDK. Here https://www.oracle.com/eg/java/technologies/javase/javase8-archive-downloads.html

2. Download and Install NetBeans or Apache NetBeans. Here: https://tinyurl.com/43dn5676

3. Open NetBeans

NetBeans



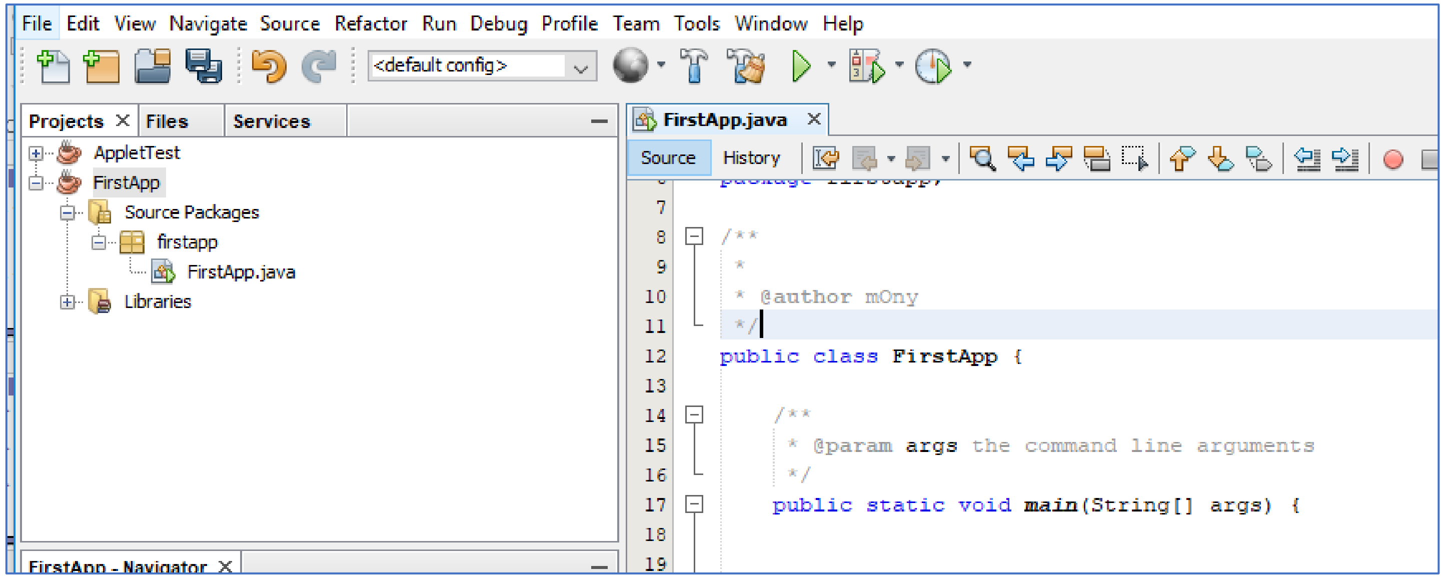
Steps

1. **Choose Project**
2. ...

Choose Project

Filter:

Categories:	Projects:
Java	Java Application
JavaFX	Java Class Library
Java Web	Java Project with Existing Sources



```
package firstapp;

/**
 *
 * @author mOny
 */
public class FirstApp {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

```

First Java Application

```
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println(“Hello Java”);
    }
}
```

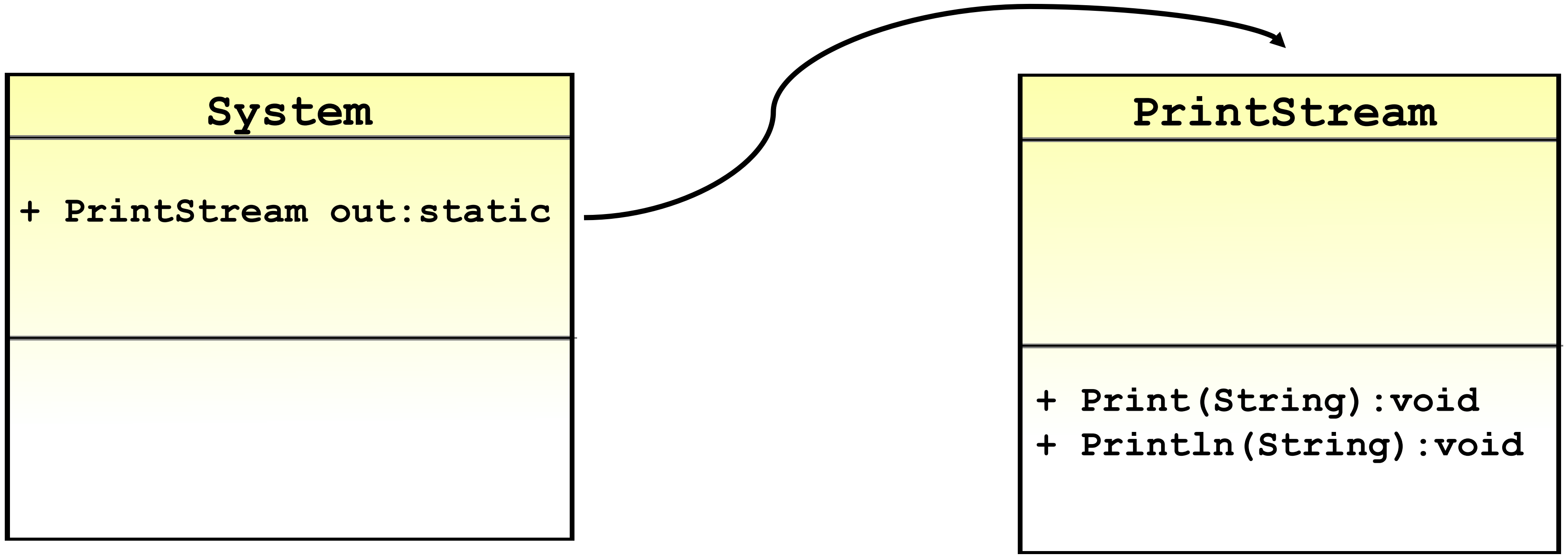
File name: `hello.java`

First Java Application cont'd

- The `main()` method:
  - Must return void.
  - Must be static.
    - because it is the first method that is called by the Interpreter (`HelloWorld.main(..)`) even before any object is created.
  - Must be public to be directly accessible.
  - It accepts an array of strings as parameter.
    - This is useful when the operating system passes any command arguments from the prompt to the application.

System.out.println("Hello");

- **out** is a static reference that has been created in class **System**.
- **out** refers to an object of class **PrintStream**. It is a ready-made stream that is attached to the standard output (i.e. the screen).



Variables
& Data Types
& Operators

Variables

- **What is a variable?**
  - **A variable refers to something that can change.**
    - Variables can be initiated with a value.
    - The value can be changed.
    - A variable holds a specific type of data.

```
String firstName = "Ahmed";
```

Variables Types

- Some of the types of values a variable can hold:
  - `String` (example: "Hello")
  - `int` (examples: -10, 0, 2, 10000)
  - `double` (examples: 2.00, 99.99, -2042.09)
  - `boolean` (true or false)

- If uninitialized, variables have a default value:
  - `String` → "" (the empty string)
  - `int` → 0
  - `double` → 0.0
  - `boolean` → false

Identifiers

- An identifier is the name given to a feature (**variable**, **method**, or **class**).
- An identifier can begin with either:
  - a letter,
  - $, or
  - underscore.
- Subsequent characters may be:
  - a letter,
  - $,
  - underscore, or
  - digits.

Variables declaration and initialization

- Syntax:

```
<access-modifier>* <type> identifier [= value];
```

- Examples:

```
String customer;

String name, city;

String address = "123 Main street";

String country = "Egypt", city = "Cairo";
```

Uses variables

- Holding data used within a method:

```
String name = "Ahmed";
double price = 12.35;
boolean outOfStock = true;
```

- Assigning the value of one variable to another:

```
String name = name1;
```

- Representing values within a mathematical expression:

```
total = quantity * price;
```

- Printing the values to the screen:

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

Data types

- Data types can be classified into two types:

Primitive

Boolean	`boolean`	1 bit	(true/false)
Integer	`byte`	1 B	$(-2^{7} \rightarrow 2^{7}-1)$ $(-128 \rightarrow +127)$
	`short`	2 B	$(-2^{15} \rightarrow 2^{15}-1)$ (-32,768 to +32,767)
	`int`	4 B	$(-2^{31} \rightarrow 2^{31}-1)$
	`long`	8 B	$(-2^{63} \rightarrow 2^{63}-1)$
Floating Point	`float`	4 B	Standard: IEEE 754 Specification
	`double`	8 B	Standard: IEEE 754 Specification
Character	`char`	2 B	unsigned Unicode chars $(0 \rightarrow 2^{16}-1)$

Reference

Arrays
Classes
Interfaces

Literals

- A literal is any value that can be assigned to a primitive data type or String.

boolean	true false	
char	‘a’ …. ’z’ ‘A’ …. ‘Z’	
	‘\u0000’ …. ‘\uFFFF’	
	‘\n’ ‘\r’ ‘\t’	
Integral data type	15	Decimal (int)
	15L	Decimal (long)
	017	Octal
	0XF	Hexadecimal
Floating point data type	73.8	double
	73.8F	float
	5.4 E-70	5.4 * 10^{-70}
	5.4 e+70	5.4 * 10^{70}

Wrapper Classes

- Each primitive data type has a corresponding wrapper class.

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

Wrapper Classes

- There are three reasons that you might use a wrapper class rather than a primitive:

1. (Optional) As an argument of a method that expects an object.
2. To use constants defined by the class,
   - such as **MIN_VALUE** and **MAX_VALUE**,
     that provide the upper and lower bounds of the data type.
3. (Object Methods) To use class methods for
   - converting values to and from other primitive types,
   - converting to and from strings.
4. Collections
   - Storing primitive types in collections **ONLY** by using their corresponding wrapper classes

Wrapper Classes

- They have useful methods that perform some general operation, for example:

`primitive parseXXX(String)`	→	convert String to primitive
`Wrapper valueOf(String)`	→	convert String to Wrapper

```
String five = "5";
int intFive = Integer.parseInt(five);
Integer integerFive = Integer.valueOf(five);
```

Reference Data types: Classes

- General syntax for creating an object:

```
MyClass myRef;              // just a reference
myRef = new MyClass();      // construct a new object
```

- Or on one line:

```
MyClass myRef = new MyClass();
```

- An object is garbage collected when there is no reference pointing to it.

Reference Data types: Classes

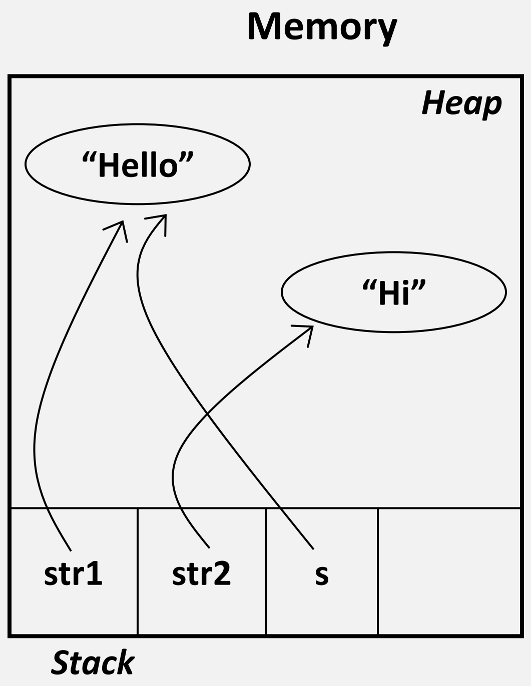
```
String str1;                      // just a null reference
str1 = new String("Hello");       // object construction
String str2 = new String("Hi");

String s = str1;            //two references to the same object

str1 = null;

s = null;         // The object containing "Hello" will
                  // now be eligible for garbage collection.

str1.anyMethod();           // ILLEGAL!
                            //Throws NullPointerException
```



Operators

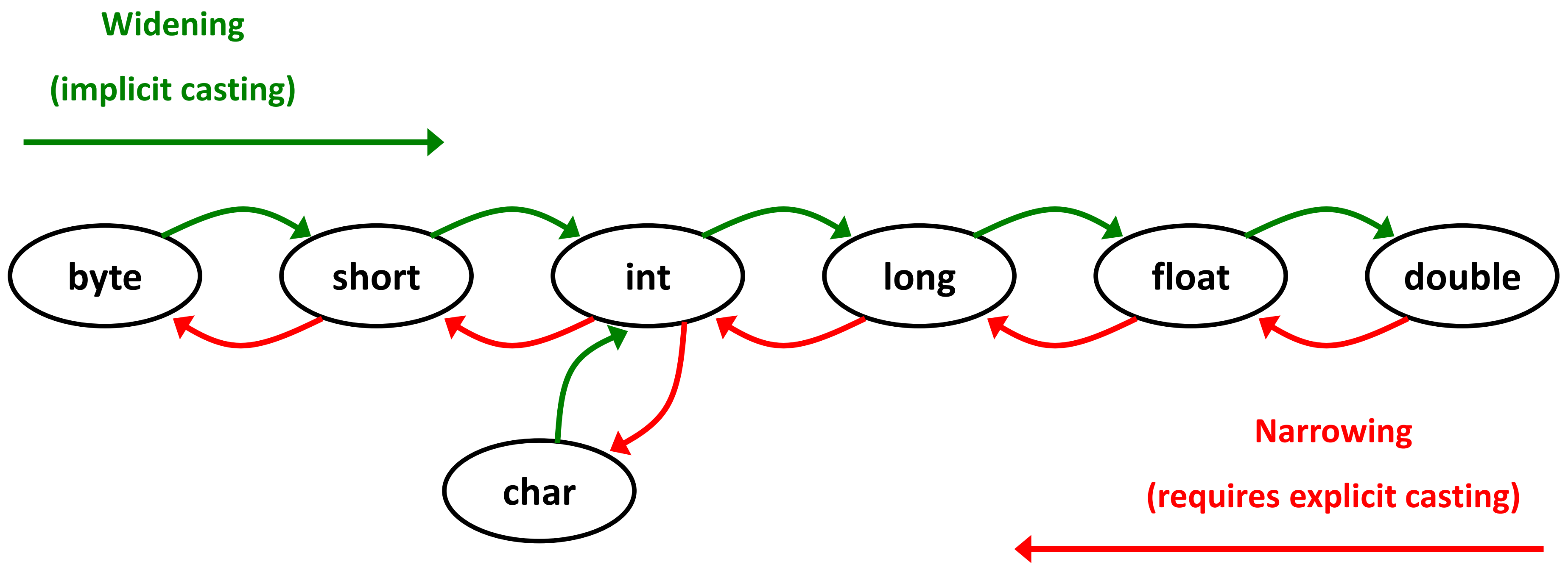
Operators are classified into the following categories:

- Unary Operators.
- Arithmetic Operators.
- Assignment Operators.
- Relational Operators.
- Shift Operators.
- Bitwise and Logical Operators.
- Short Circuit Operators.
- Ternary Operator.

Operators

- Unary Operators:

+	-	++	--	!	~	( )
positive	negative	increment	decrement	boolean complement	bitwise inversion	casting



Operators

- Arithmetic Operators:

+	-	*	/	%
add	subtract	multiply	division	modulo

- Assignment Operators:

=	+=	-=	*=	/=	%=	&=	|=	^=

- Relational Operators:

<	<=	>	>=	==	!=	Instanceof

Operations must be performed on homogeneous data types

Operators

Bitwise and Logical Operators:

&	**|**	**^**
AND	OR	XOR

Short Circuit Operators:

&&	**||**
(condition1 AND condition2)	(condition1 OR condition2)

Operators

- Ternary Operator:

```
condition ? true statement : false statement
```

```
int y = 15;
int z = 12;
int x = y < z ? 10 : 11;
```

$\rightarrow$

```
if(y<z)
    x=10;
else
    x=11;
```

Using Arrays & Strings

What is Array?

- An Array is a collection of variables of the **same data type**.
- Each element can hold a **single item**.
- Items can be **primitives** or **object references**.
- The length of the array is determined when it is created.

Declaring an Array

- General syntax for creating an array:

```
Datatype[] arrayIdentifier;                     // Declaration
arrayIdentifier = new Datatype [size];          // Construction
```

- Or on one-line, hard coded values:

```
Datatype[] arrayIdentifier = { val1, val2, val3, val4 };
```

- To determine the size (number of elements) of an array at runtime, use:

```
                   arrayIdentifier.length
```

Declaring an Array

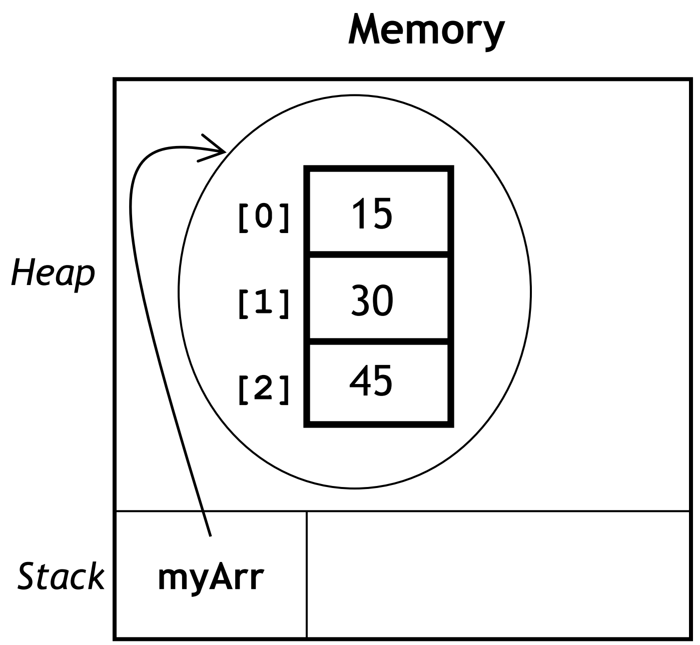
- **Example1:** Array of Primitives:

```
int[] myArr;
myArr = new int[3];

myArr[0] = 15;
myArr[1] = 30;
myArr[2] = 45;

System.out.println(myArr[2]);

myArr[3] = … ;          // ILLEGAL!
                        //Throws ArrayIndexOutOfBoundsException
```



Declaring an Array

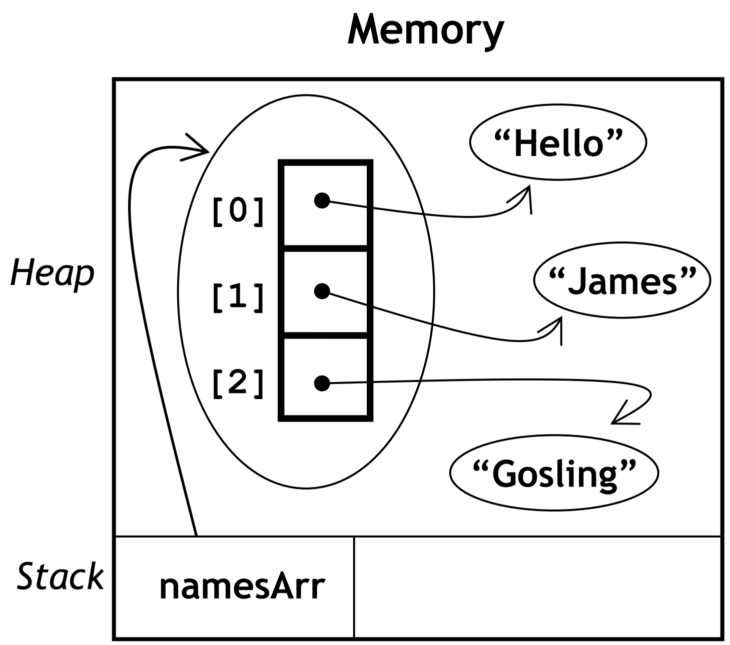
- **<u>Example2:</u>** Array of Object References:

```
String[] namesArr;
namesArr = new String[3];

namesArr[0].anyMethod()      // ILLEGAL!
                             //Throws NullPointerException


namesArr[0] = new String("Hello");
namesArr[1] = new String("James");
namesArr[2] = new String("Gosling");

System.out.println(namesArr[1]);
```



String Operations

- Although String is a reference data type (class),
  - it may figuratively be considered as the 9th data type because of its special syntax and operations.
  - Creating String Object:

```
String myStr1 = new String("Welcome");
String sp1 = "Welcome";
String sp2 = " to Java";
```

  - Testing for String equality:

```
if(myStr1.equals(sp1))

if(myStr1.equalsIgnoreCase(sp1))

if(myStr1 == sp1)
// Shallow Comparison (just compares the references)
```

Strings Operations

```
String myStr1 = new String("Welcome");
String sp1 = "Welcome";
String sp2 = " to Java";
```

- The ‘+’ and ‘+=‘ operators were overloaded for class String to be used in concatenation.

```
String str = myStr1 + sp2;        // “Welcome to Java”
str += " Programming";            // “Welcome to Java Programming”
str = str.concat(" Language");    // “Welcome to Java Programming Language”
```

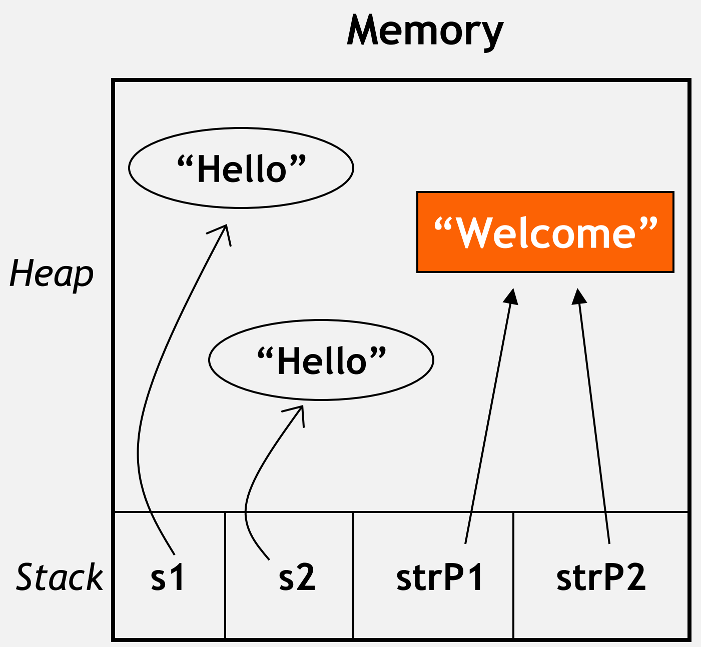
- Objects of class String are immutable
  - you can’t modify the contents of a String object after construction.

- Concatenation Operations always return a new String object that holds the result of the concatenation. The original objects remain unchanged.

String Pool

```
String s1 = new String("Hello");
String s2 = new String("Hello");

String strP1 = "Welcome";
String strP2 = "Welcome";
```



- String objects that are created without using the "new" keyword are said to belong to the "String Pool".

String Pool

- String objects in the pool have a special behavior:
  - If we attempt to create a fresh String object with exactly the same characters as an object that already exists in the pool (case sensitive), then no new object will be created.
  - Instead, the newly declared reference will point to the existing object in the pool.
- Such behavior results in a better performance and saves some heap memory.
- Remember: objects of class String are immutable.

`StringBuilder` Class

`StringBuilder` provides a **mutable** alternative to `String`. `StringBuilder`:

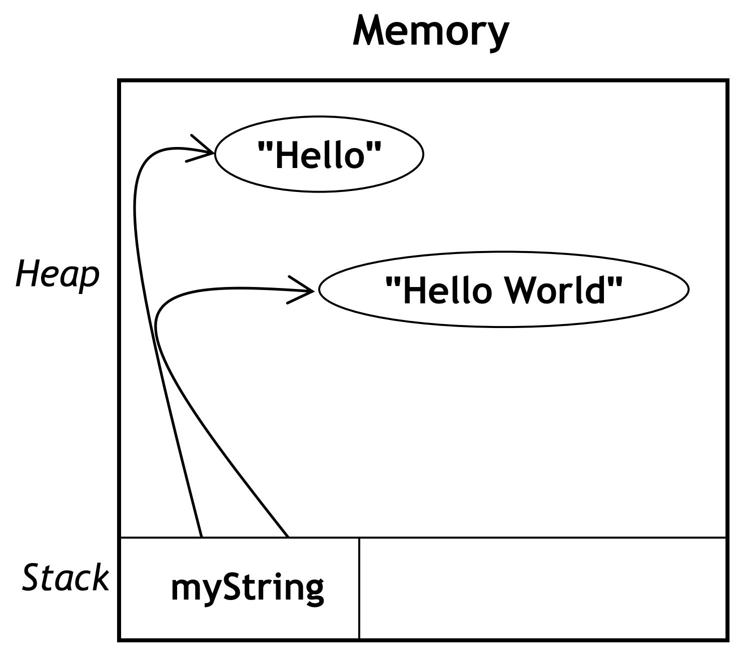
- Is instantiated using the **`new`** keyword
- Has many methods for manipulating its value
- Provides better performance because it is **mutable**
- Can be created with an initial capacity

`String` is still needed because:

- It may be safer to use an **immutable** object
- A method in the API may require a string
- It has many more methods not available on `StringBuilder`

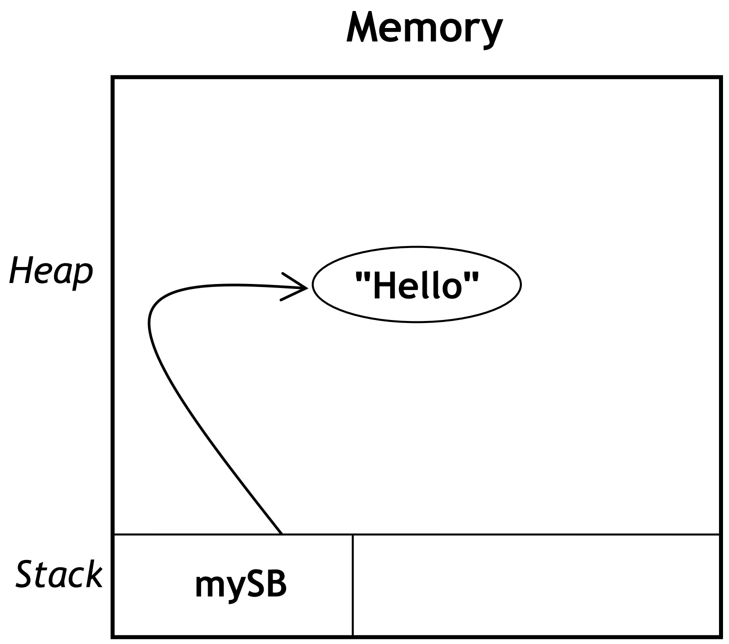
`StringBuilder` Advantages over `String` for Concatenation (or Appending)

```
String myString = "Hello";
myString = myString + " World";
```



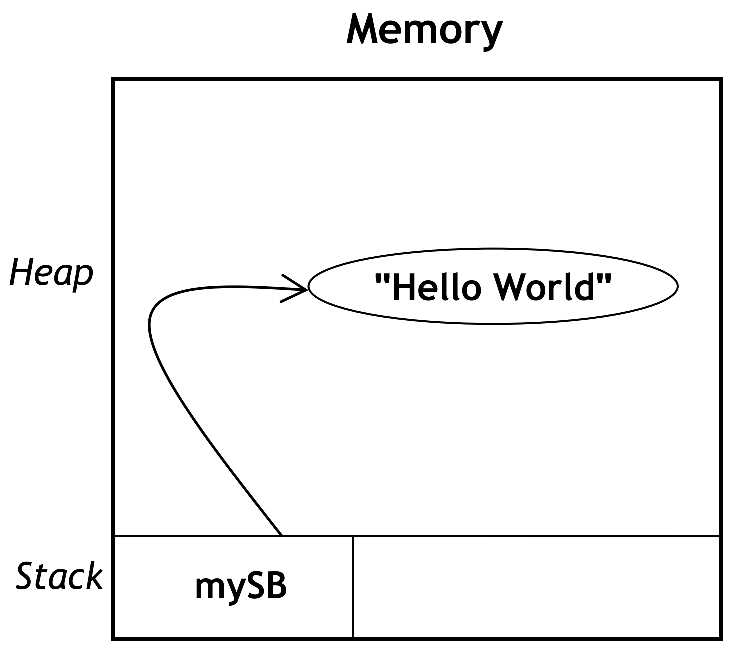
`StringBuilder` Declare, Instantiate, and Append

```
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



StringBuilder Declare, Instantiate, and Append

```
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



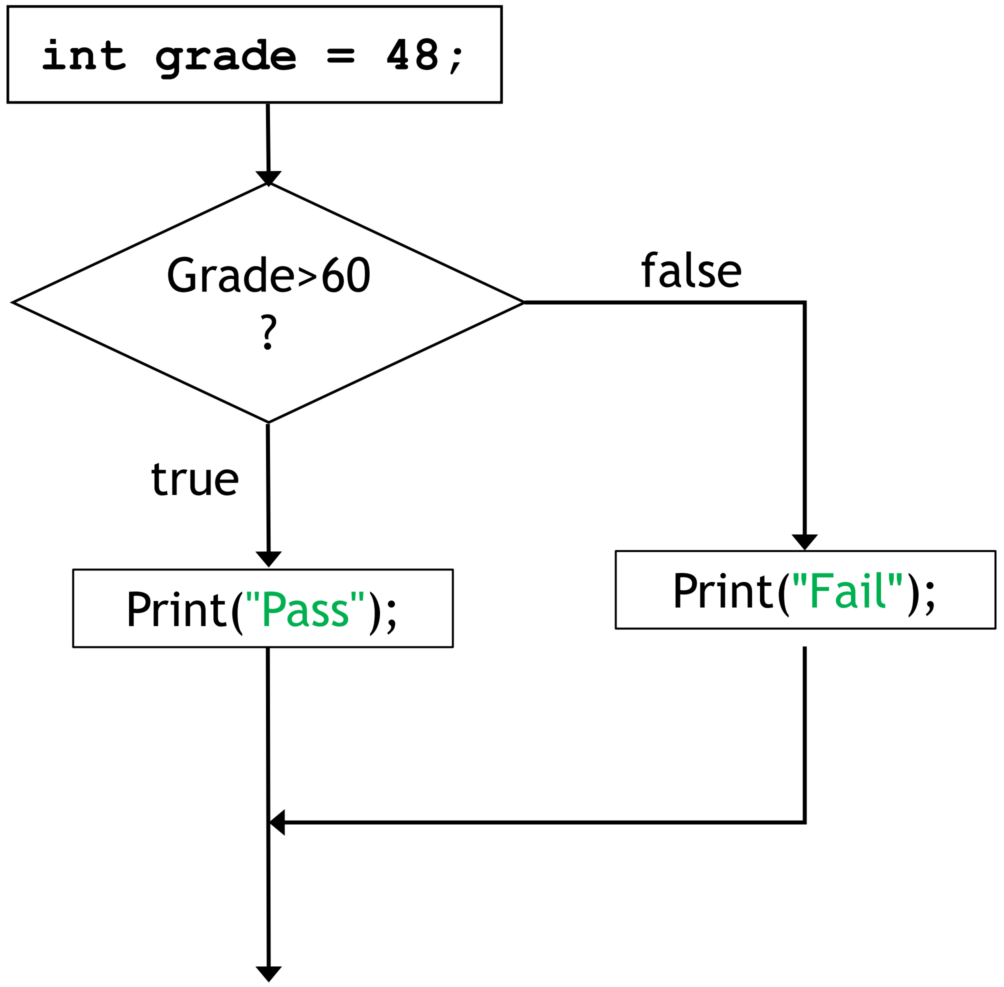
Controlling Program Flow

Flow Control: Branching - if, else

- The if and else blocks are used for binary branching.
- **Syntax:**

```
if(boolean_expr)
{
      …
      …    //true statements
      …
}
[else]
{
      …
      …    //false statements
      …
}
```

if, else Example



```
int grade = 48;
if(grade > 60)
  System.out.println("Pass");
else
{
  System.out.println("Fail");
}
```

Flow Control: Branching - switch

- The switch block is used for multiple branching.
- **Syntax:**

```
switch(myVariable){
        case value1:
           …
           …
           break;
        case value2:
           …
           …
           break;
        default:
           …
}
```

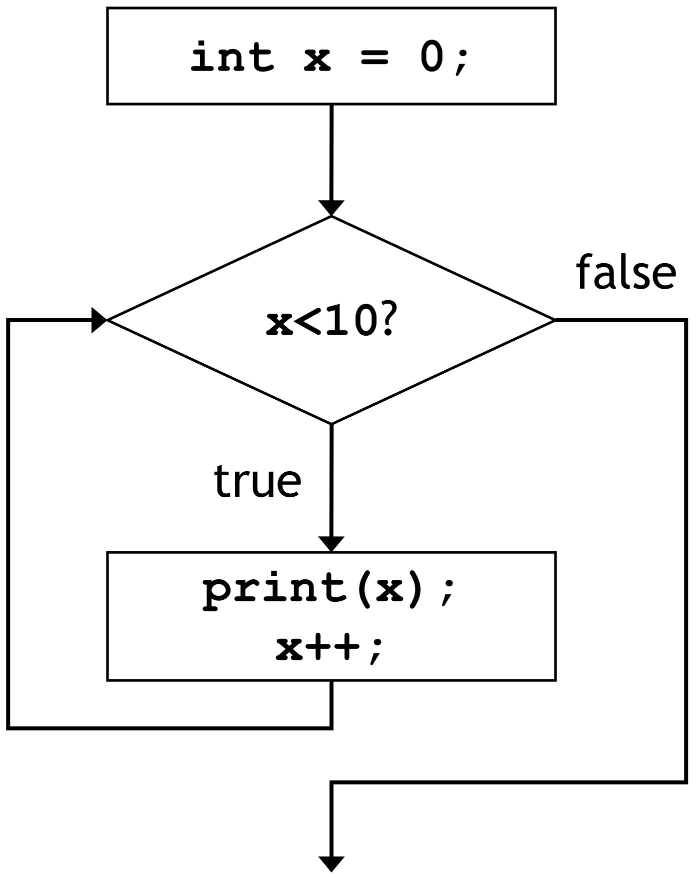
- byte
- short
- int
- char
- enum
- String "Java 7"

Flow Control: Iteration – while loop

- The **while loop** is used when the **termination condition occurs unexpectedly** and is checked at the beginning.
- **Syntax**:

```
while (boolean_condition)
{
        …
        …
        …
}
```

while loop Example



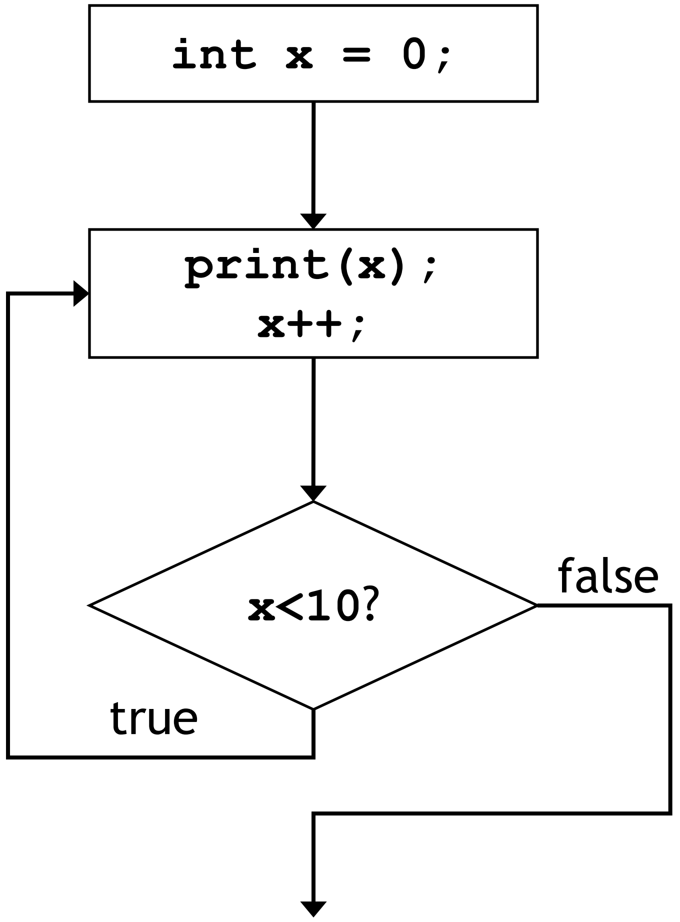
```
int x = 0;
while (x<10) {
    System.out.println(x);
    x++;
}
```

Flow Control: Iteration – do..while loop

- The **do..while loop** is used when the **termination condition occurs unexpectedly and is checked at the end (execute once at least)**.
- **Syntax:**

```
do
{
        …
        …
        …
}
while(boolean_condition);
```

do..while loop Example



```
int x = 0;
do{
    System.out.println(x);
    x++;
} while (x<10);
```

Flow Control: Iteration – for loop

- The for loop is used when the number of iterations is **predetermined**.
- **Syntax:**

```
for (initialization ; loop_condition ; step)
{
        …
        …
        …
}
```

```
for (int i=0 ; i<10 ; i++)
{
        …
        …
}
```

Flow Control: Iteration – Enhanced for loop

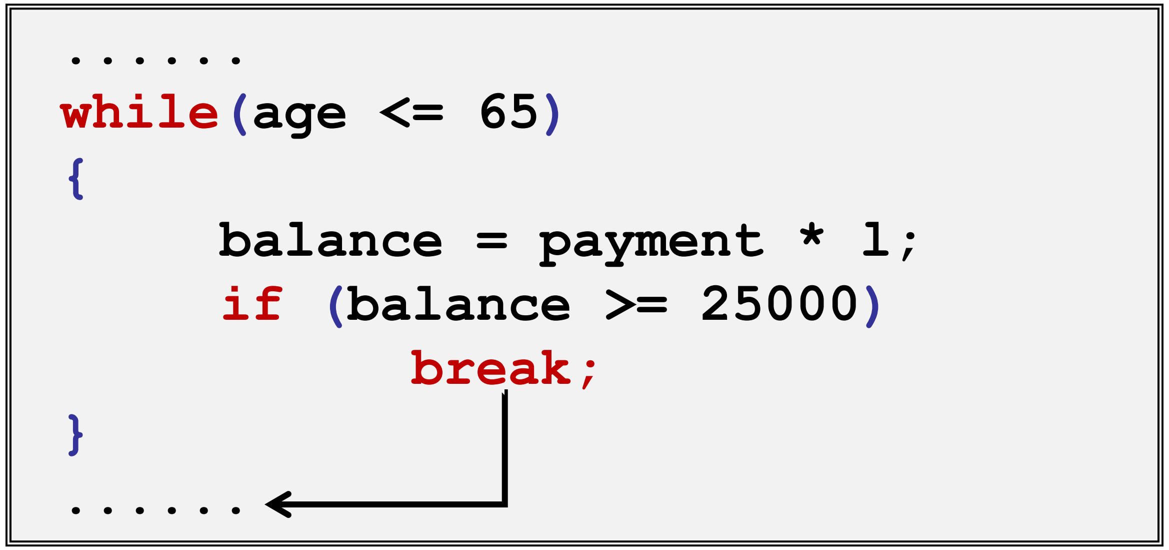
- **The first element:**
  - is an identifier of the same type as the iterable_expression

- **The second element:**
  - is an expression specifying a collection of objects or values of the specified type.
- The enhanced loop is used when we want to iterate over **arrays** or **collections.**

```
for (type identifier : iterable_expression)
{
        // statements
}
```

The `break` statement

- The break statement can be used in **loops** or **switch**.
- It transfers control to the first statement after the loop body or switch body.



```
......
while(age <= 65)
{
     balance = payment * l;
     if (balance >= 25000)
           break;
}
......
```

The `continue` statement

- The continue statement can be used **ONLY** in loops.
- Abandons the current loop iteration and jumps to the next loop iteration.

```
......
for(int year=2000; year<= 2099; year++){
     if (year % 100 == 0)
           continue;
}
......
```

Comments in Java

- To comment a single line:

```
// write a comment here
```

- To comment multiple lines:

```
/*  comment line 1
    comment line 2
    comment line 3 */
```

Lab Assignment

- Create a simple non-GUI Application that prints out the following text on the command prompt: **Hello Java**

- Create a simple program to determine if a given number is even or odd.

- Create a non-GUI Application that accepts a well-formed IP Address in the form of a string and cuts it into separate parts based on the dot delimiter.

  Input: 163.121.12.30

  Output:

  ```
  163
  121
  12
  30
  ```

Java SE8 Documentation

- Overview (Java Platform SE 8) (oracle.com)