Programming with Java

Banu Prakash C

Objectives

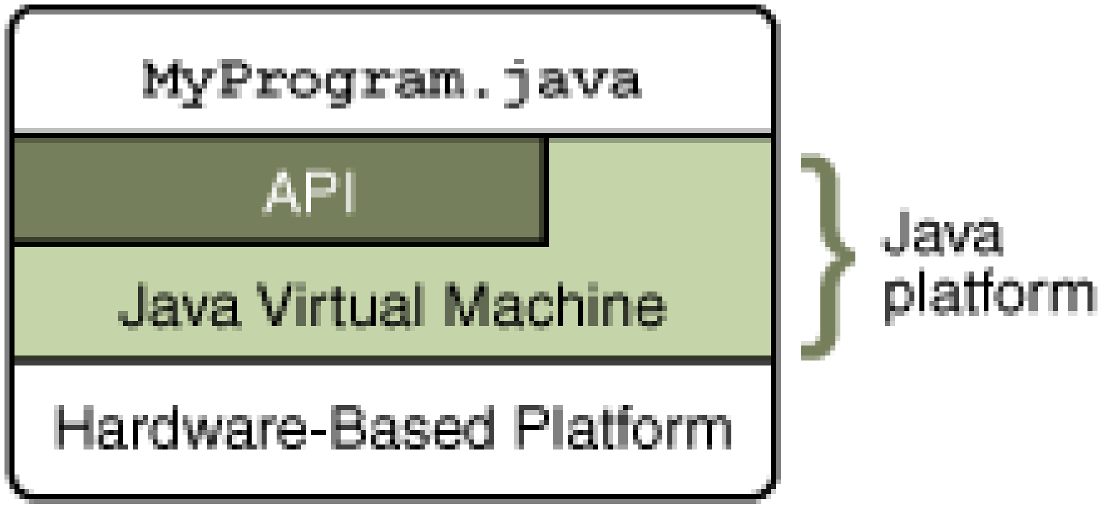
- Understand Java Technology
- Get the complete picture of Java
- Understand Features of Java
- Understand different Java environments
- Understand the Java architecture
- J2SE, J2EE, and J2ME Platforms
- Understand variables, data types and identifiers
- Understand Java language constructs
- Applications vs. Applets

Java Technology

- Java technology is both a platform and a programming language.
- A *platform* is the hardware or software environment in which a program runs.
- Most platforms can be described as a combination of the operating system and underlying hardware.
- The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

Java Platform

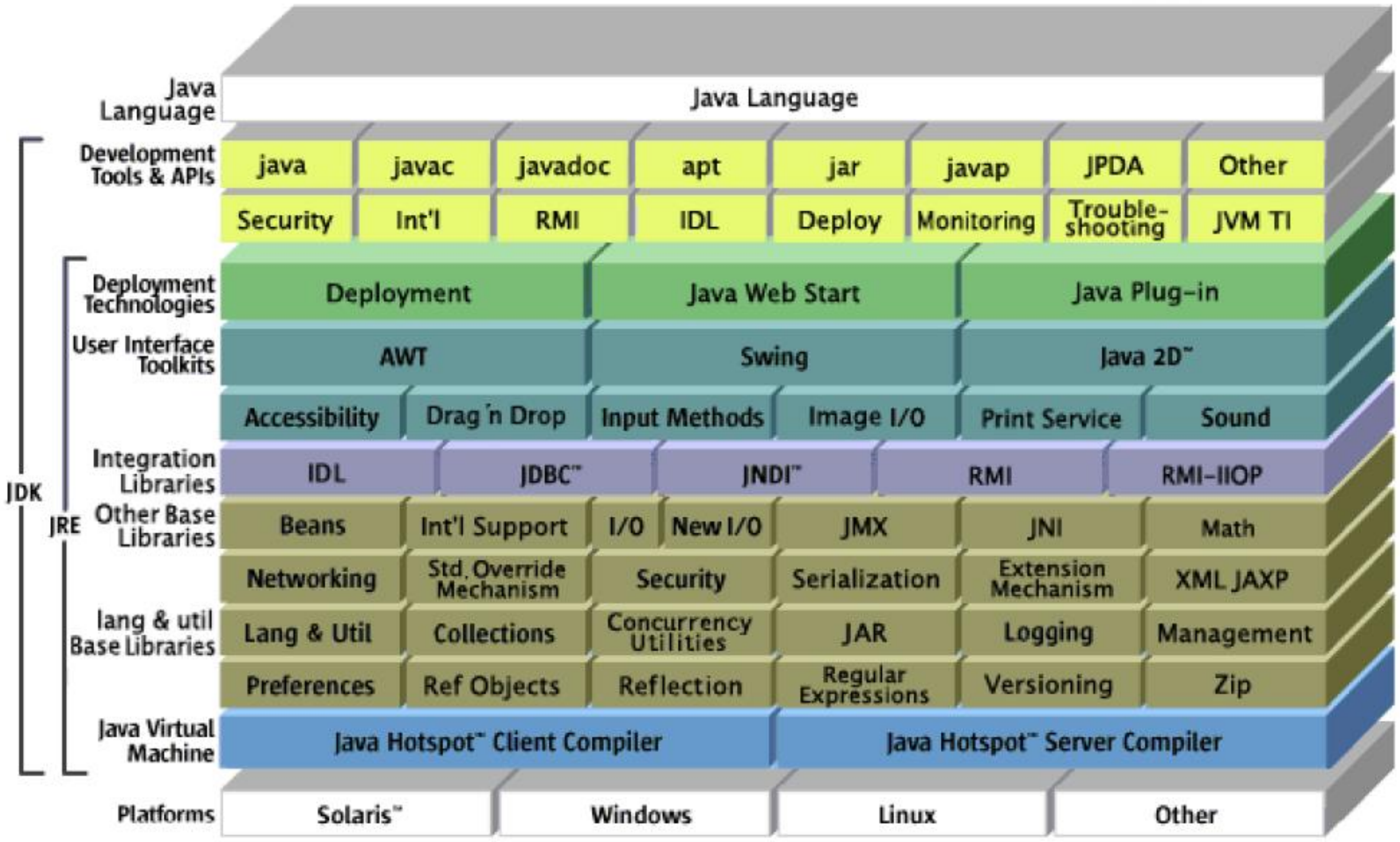
- The Java platform has two components:
  - The *Java Application Programming Interface* (API)
  - The *Java Virtual Machine*



Java Platform

- **Application Programming Interface (API)**:
  - An application programming interface (API) is a particular set of rules and specifications that software programs can follow to communicate with each other.
  - The Java API offers a wide collection of useful classes ready for use in your own applications.

Java Complete Picture



Features of Java

- **Simple**
  - Although Java resembles C++, it omits many of C++'s more confusing features, including the ones most likely to cause problems for beginners: pointers, operator overloading, multiple inheritance, and templates
- **Object-Oriented**
- **Distributed**
- **Robust**
  - ***No pointers.*** Although Java uses pointers internally, no pointer operations are made available to programmers. There are no pointer variables, arrays can't be manipulated via pointers, and integers can't be converted into pointers.
  - ***Garbage collection.*** Thanks to automatic garbage collection, there's no chance of a program corrupting memory via a dangling pointer.
  - ***Strict type checking.*** Java's type checking is much stricter than that in C or C++. In particular, casts are checked at both compile time and run time. As a bonus, type checking is repeated at link time to detect version errors.
  - ***Run-time error checking.*** Java performs a number of checks at run time, including checking that array subscripts are within bounds.

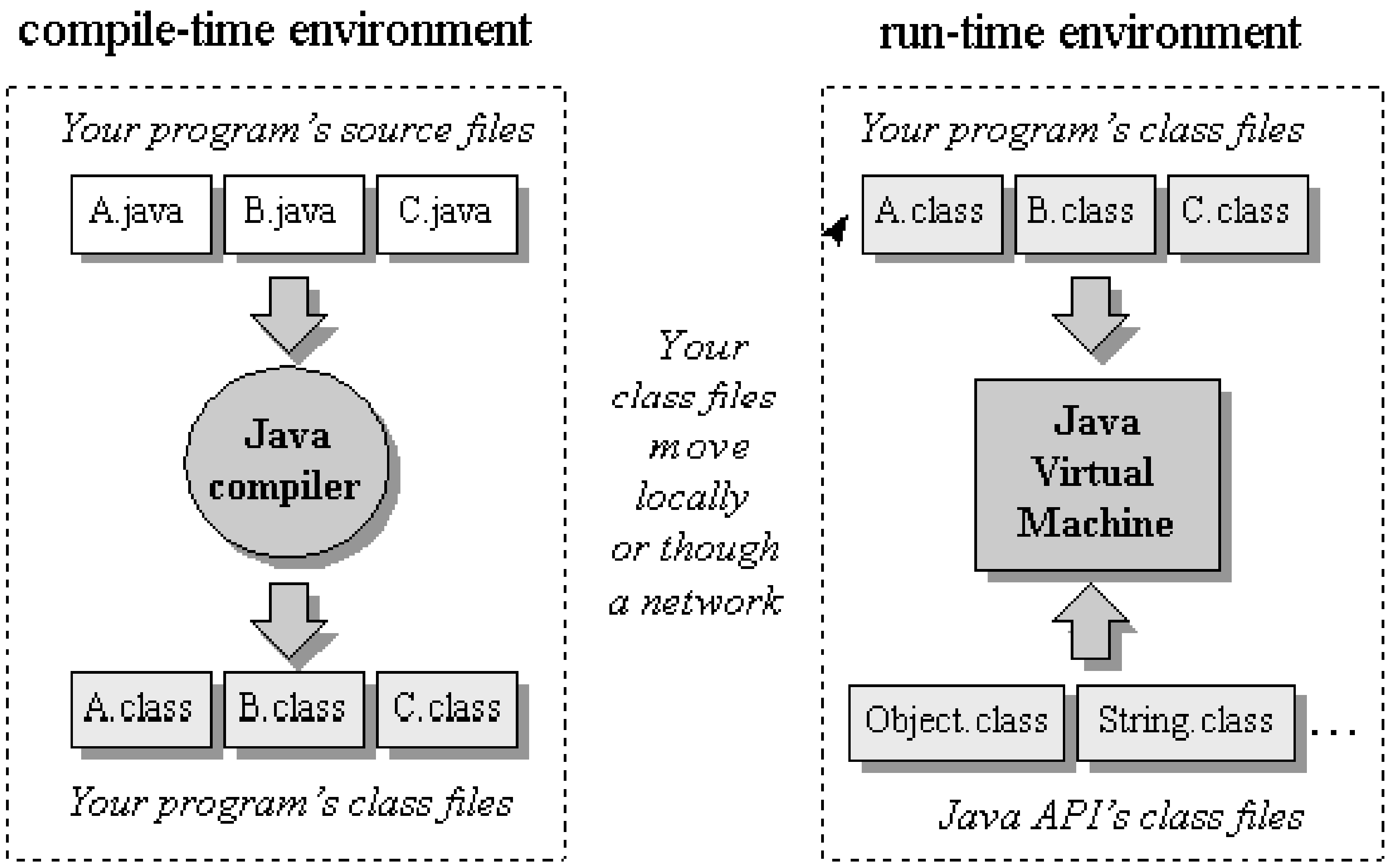
Features of Java

- **Secure**
  - Java programs are designed to be *secure* (safe against malicious attack). Java's run-time system performs checks to make sure that programs transmitted over a network have not been tampered with.
  - The code produced by the Java compiler is checked for validity, and the program is prevented from performing unauthorized actions. For example, an applet that's been downloaded from a Web page can't access files on the local computer.
  - Moreover, the nature of Java makes it hard to write viruses and other kinds of malicious programs. A program that can't access memory locations via pointers will find it hard to do much damage

Features of Java

- **Architecture-Neutral**
  - The Java language is completely architecture-neutral. As a result, programs written in Java will run on any platform that supports the Java run-time system
- **Portable**
  - Java achieves portability by completely defining all aspects of the language, leaving no decisions to the compiler writer.
  - Consider the issue of types. Most programming languages don't define the exact ranges of types, allowing for variations based on the computer's architecture. Java, on the other hand, completely defines the ranges and properties of all types.
- **Multithreaded**
- **Dynamic**

Java Environment



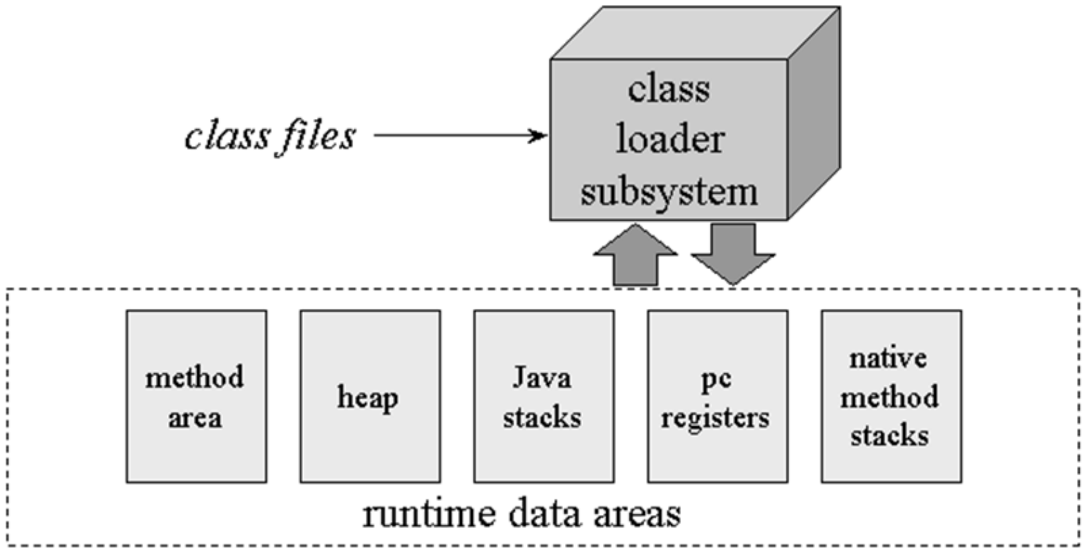
ByteCode

- Bytecode is an intermediate form of java programs. Bytecode consists of optimized set of instructions that are not specific to processor.

- We get bytecode after compiling the java program using a compiler [javac].

- The bytecode is to be executed by java runtime environment which is called as Java Virtual Machine (JVM).

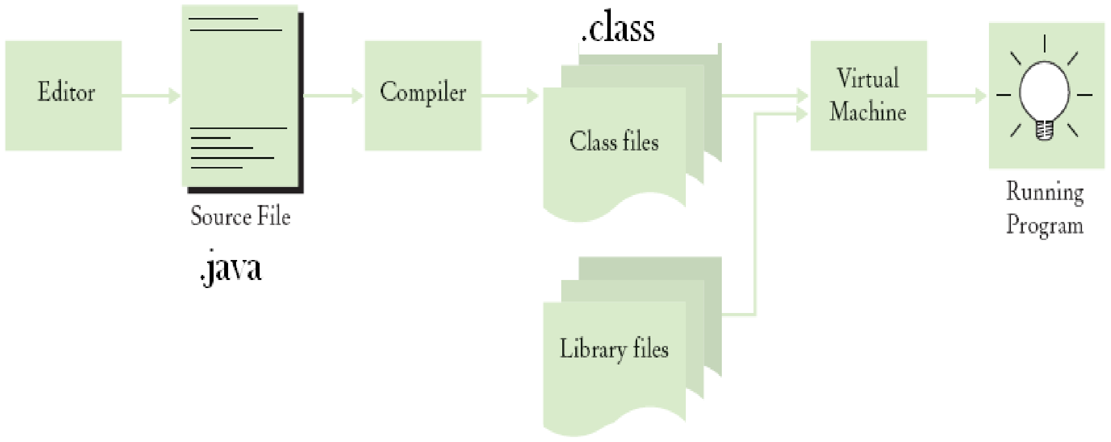
Java Runtime Environment



- The Class loader loads all the classes (byte code) : classes from program and from Java API.
- For every class the type information is stored in Method Area.
- As the program runs the virtual machine places all the objects in heap area.
- The java method invoked will be placed on the stack .
- PC (program counter) indicates the next instruction to execute.

Java Execution Process

- Java Source code having an extension "*.java" will be compiled using Java Compiler.
- The Compiled byte code has ".class" extension.
- The byte code and library files (Java API) will be loaded into JVM by class loader.
- The execution of byte code takes place in JVM



JIT Just-In-Time Compilation

- The JVM executes .class or .jar files, by either interpreting it or using a just-in-time compiler (JIT).
- The JIT compiler is enabled by default, and is activated when a Java method is called.
- The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run.
- When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.
- When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time.
- In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold.
- Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all

Java HotSpot Compiler

- HotSpot is the Just-In-Time compiler that is part of JVM, Standard Edition.
- HotSpot – Optimize only when required.
  - A program starts off being interpreted.
  - A Profiler collects run-time info in the background
  - After a while, a set of hot spots are identified.
  - A Thread is launched to compile the methods in the hot spots
    - Execution of the program is *not* blocked.
    - “Take your time!” – fully optimized
    - Take advantage of the late compilation by using run-time info.
  - Once a method is compiled it need not be interpreted.
  - Performance
    - 2-3 times faster than JITs
    - Comparable to C++.

Java Platforms

- Java SE
    - Java Platform, Standard Edition (Java SE) lets you develop and deploy Java applications on desktops and servers, as well as in today's demanding embedded environments.
- Java ME
    - Java Platform, Micro Edition (Java ME) provides a robust, flexible environment for applications running on mobile and other embedded devices: mobile phones, personal digital assistants (PDAs), TV set-top boxes, and printers.
- Java EE
    - Java Platform, Enterprise Edition (Java EE) provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications. Java EE extends the Java SE

Application vs. Applet

- Applications - Java programs that run directly on your machine.
  - Applications must have a *main()*.
  - Java applications are compiled using the **javac** command and run using the **java** command
- Applets - Java programs that can run over the Internet.
  - The standard client/server model is used when the Applet is executed.
  - The server stores the Java Applet, which is sent to the client machine running the browser, where the Applet is then run.
  - Applets do not require a *main()*
  - An Applet also generally embedded within an HTML file .
  - Java Applets are also compiled using the **javac** command, but are run either with a browser or with the **appletviewer** command.

Java source file structure

declaration order

1. Package declaration

Used to organize a collection of related classes.

2. Import statement

Used to reference classes and declared in other packages.

3. Class declaration
A Java source file can have several classes but only one public class is allowed.

```
/*
 * Created on June 1, 2011
 *
 * First Java Program
 */
package com.banu.sample;
import java.lang.System;
import java.lang.String;

/**
 * @author Banu Prakash
 */
public class Example{

    public static void main(String[] args) {
          // print a message
          System.out.println("Welcome to Java!");
    }
}
```

Java source file structure

Class

- Every Java program includes at least one class definition. The class is the fundamental component of all Java programs.

```
/*
 * Created on June 1, 2011
 *
 * First Java Program
 */
package com.banu.sample;
import java.lang.System;
import java.lang.String;

/**
 * @author Banu Prakash
*/
public class Example{

    public static void main(String[] args) {
          // print a message
          System.out.println("Welcome to Java!");
    }
}
```

Java source file structure

Comments

1. Block Comment

```
/*
 * insert comments here
 */
```

2. Documentation Comment

```
/**
  * insert documentation
  */
```

3. Single Line Comment

```
// insert comments here
```

The compiler ignores comments.

```
/*
 * Created on June 1, 2011
 *
 * First Java Program
 */
package com.banu.sample;
import java.lang.System;
import java.lang.String;

/**
 * @author Banu Prakash
*/
public class Example{

    public static void main(String[] args) {
          // print a message
          System.out.println("Welcome to Java!");
    }
}
```

Setting up the environment and executing

- Video: HelloWorld_example.swf
  - Hello World Java application illustrating the basic java file structure.
  - illustrates steps to carry out to run this application:
    - Setting path and classpath.

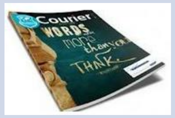
Variables

- A variable is a storage location and an associated symbolic name which contains information, a value.

- A programmer can imagine a variable as being a box into which information can be placed, and the shape of the box (the variables *type*) determines what kind of information it can store

Variables

- **Real world example of a variable:**
  - Imagine that you are a postal worker. Your job is to sort mail into rows of boxes

100 Main Street	101 Main Street	102 Main Street	Variables
			contents

  - Each box has an address and a space in which to put the mail.
  - Every day the contents of the boxes change as you load and unload the boxes. On Day 1 the 100 Main Street might contain a magazine. On Day 2 to the same address it could be Internet-Broadband bill.
  - The contents of the box vary frequently, but the addresses themselves do not vary. 100 Main Street is always 100 Main Street, no matter what its contents are.

Variables

- **Programming Example:**
  - Circle's circumference : **c = 2πr** [C for "circumference" and "*r*" for "radius"]
  - Given the radius "r" a non-negative number, we get to figure out what the circumference "c" is. Both "r" and "c" are variables.
  - The value of "π" is not going to change, hence "π" is a constant.

  - Identify the variables in Quadratic Formula:

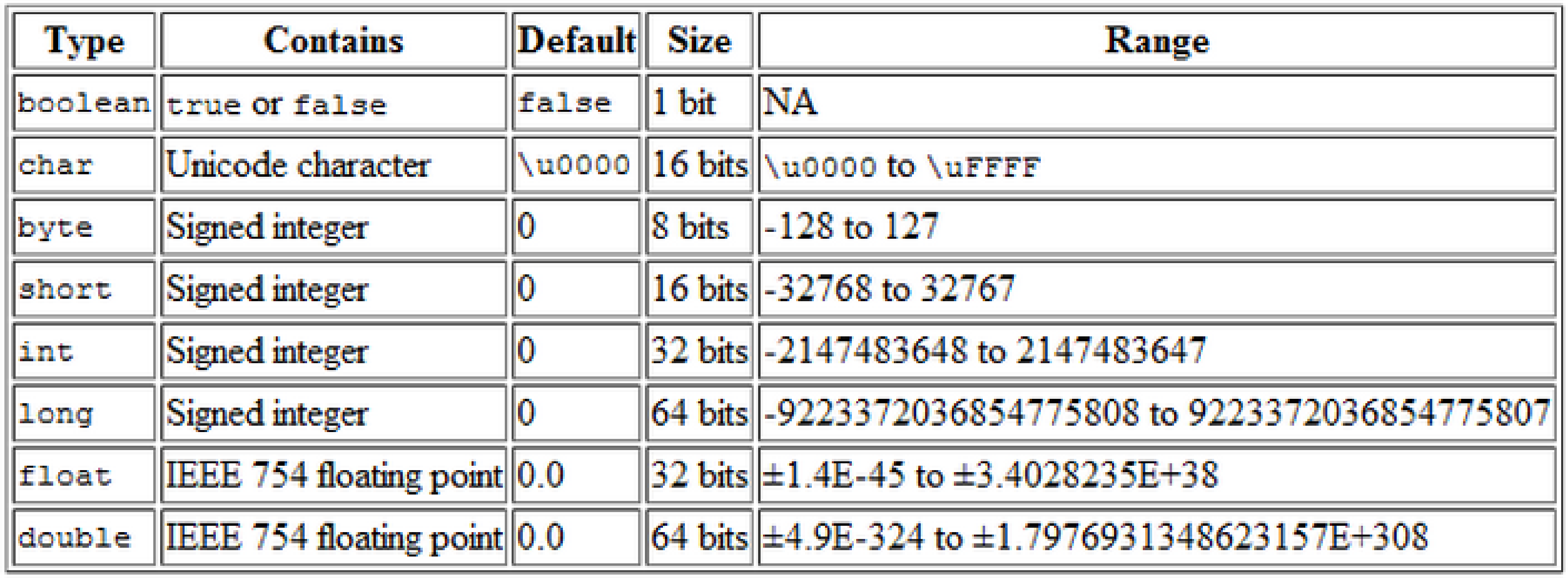
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Data types

- Data Types in Java Programming Language are classified into two main groups
  - Primitive Data Types
  - Reference Data Types
- Primitive Data Types:
    - Predefined by the Java language
    - Reserved keyword
    - Primitive values do not share state with other primitive values.
    - All Primitive Data Types have respective Wrapper Classes.

Data types

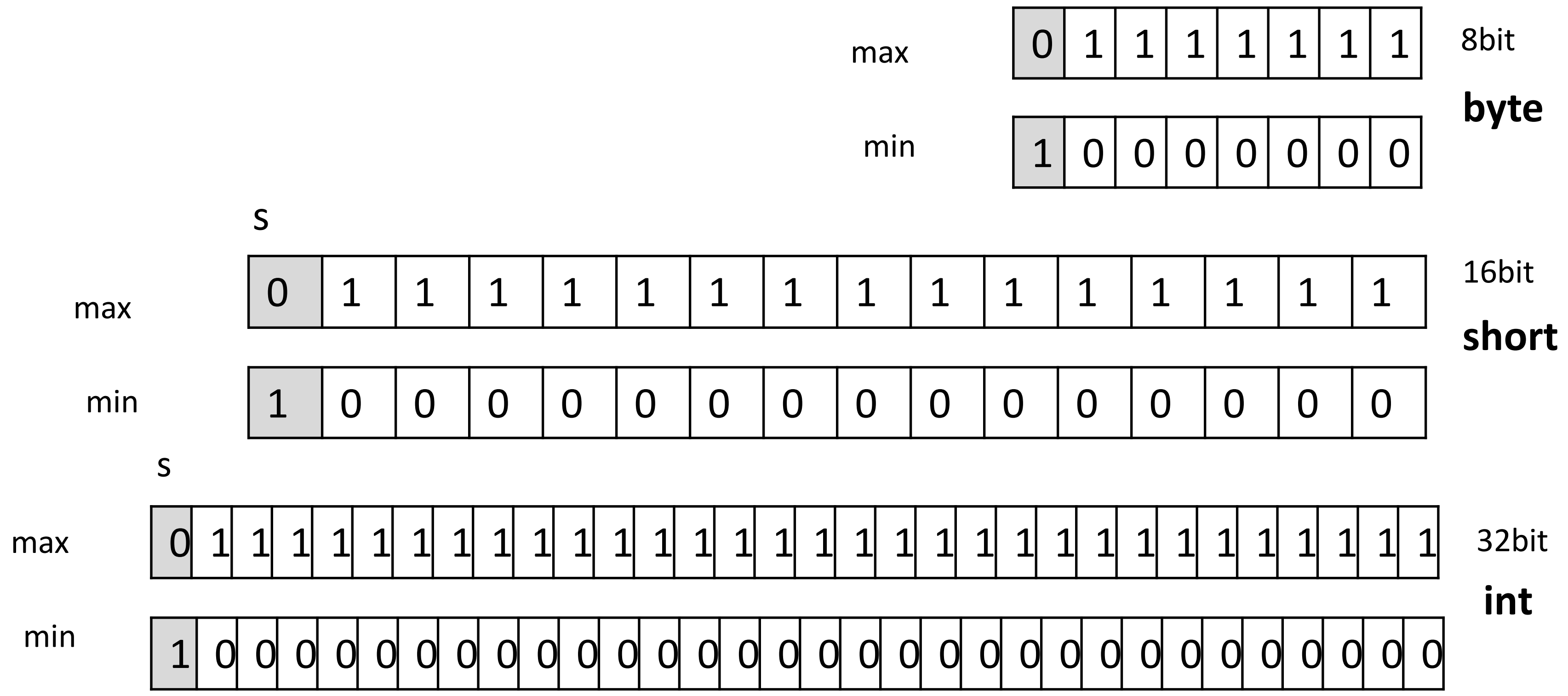
- Primitive data types



Type	Contains	Default	Size	Range
`boolean`	`true` or `false`	`false`	1 bit	NA
`char`	Unicode character	`\u0000`	16 bits	`\u0000` to `\uFFFF`
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

Data types

- Integer data types



S - > Sign bit

Data types

- The float/double data type

float	double
Variables of this type can have values from -3.4E38 (-3.4 * 1038) to +3.4E38 (+3.4 * 1038) and occupy 4 bytes in memory. Values are represented with approximately 7 decimal digits accuracy.	Variables of this type can have values from -1.7E308 (-1.7 * 10308) to +1.7E308 (+1.7 * 10308) and occupy 8 bytes in memory. Values are represented with approximately 17 decimal digits accuracy.

Data types

- The boolean type:
  - The boolean data type is used for **logical values**.
  - The boolean data type can have two possible values : **true or false**.
  - The boolean is the type **returned by all relational operators**
  - The boolean is the type required **by the conditional expressions** used in control statements such as if and for.

- The char data type:
  - Character is **16 bits wide in Java**.
  - Java uses **Unicode to represent characters**.
  - The range of a char is **0 to 65,536**.

Identifiers

- Identifiers are tokens that represent names of variables, methods, classes, etc.

  Examples of identifiers are: Hello, main, System, out.

- Java identifiers are case-sensitive.

  This means that the identifier **Hello** is not the same as **hello.**

- Identifiers must begin with either a letter, an underscore “_”, or a dollar sign “$”.

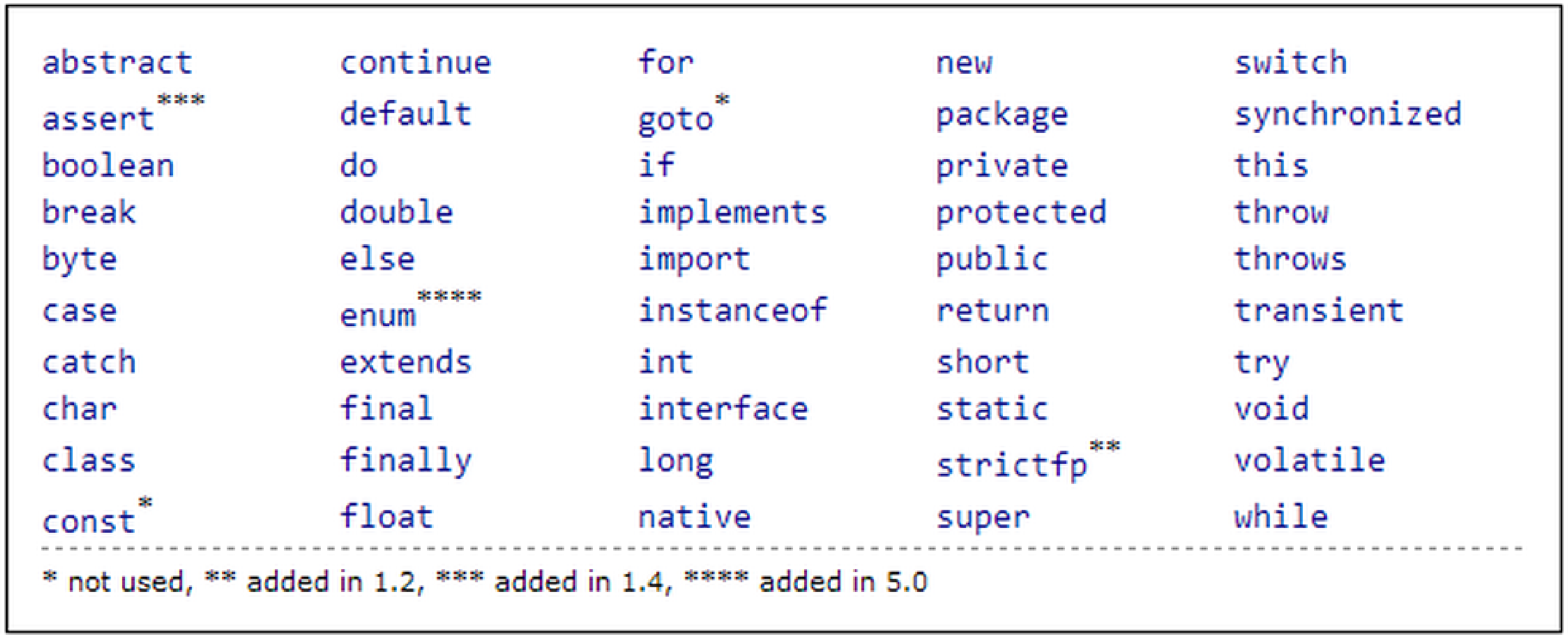
- Letters may be lower or upper case. Subsequent characters may use numbers 0 to 9.

Can you identify legal identifiers from the list?

- MyVariable
- _9pins
- $myvariable
- My Variable
- 9pins
- Mindtree_&_Associates
- testing1-2-3
- _myvariable
- This_is_an_insanely_long_variable_name_that_just_keeps_going_and_going

Java Keywords (Reserved words)

- Keywords are the built-in vocabulary of a language.
- In Java all keywords are lowercase
- You cannot use keywords as **identifiers**, (names you make up and then use to describe classes, objects, methods, and variables



abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used, ** added in 1.2, *** added in 1.4, **** added in 5.0

Java Literals

A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation

- Integer literals in our programs:
  - Decimal
    - example: 12
  - Hexadecimal
    - example: 0xC
  - Octal
    - example: 014
- Double Literals (Floating Point):
  - 583.45 (standard),
  - 5.8345e2 (scientific)
- Float Literals:
  - 583.45f
- Boolean
  - Boolean literals have only two values, true or false.
- Character literal
  - the letter a, is represented as ‘a’.
  - ‘\n’ for the newline
- String literals
  - “Hello World”.

Variables and data types

- Primitive declarations with assignments:
  - int x;
  - x = 456;
  - byte b = 22;
  - boolean exists = true;
  - double salary = 52345.22;
  - char ch = ‘J’;
  - float temperature = 94.5f;
  - int y = x;

Operators

- Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.
- Operators with higher precedence are evaluated before operators with relatively lower precedence
- When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first
- All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operators

Precedence	Operator	Description	Associativity
1	() [ ] .	Method invocation Array access Class member selection	Left-to-right
2	++ --	Postfix increment and decrement	Left-to-right
3	++ -- + - ! ~ (type) val new	Prefix increment and decrement Unary plus and minus Logical Not and bitwise NOT Type cast Class instance or array creation	Right-to-left

Operators

Precedence	Operator	Description	Associativity
4	* / %	Multiplication, division, and modulus (remainder)	Left-to-right
5	+ -	Addition and subtraction	
6	<< >> >>>	Bitwise left shift, signed right shift and unsigned right shift	
7	< <= > >= instanceof	Relational “less than” and “less than or equal to” Relational “greater than” and “greater than or equal to” Type comparison	
8	== !=	Relational “equal to” and “not equal to”	
9	& ^ |	Bitwise AND, XOR and OR	
10	&& ||	Logical AND and OR	

Operators

Precedence	Operator	Description	Associativity
11	*c* ? *t* : *f*	Ternary condition	Left-to-right
12	= += -= *= /= %= <<= >>= &= ^= !=	Assignment	Right-to-left

String

- Strings are objects designed to represent a sequence of characters
- Java allows programmers to use Strings in a similar manner as other common programming languages.
- Creation of a String through assignment does not require the use of the new operator, for example
    - String name= “Banu Prakash";
    - String language = "Java";
- String objects can be created through constructors. The following constructors are supported:
- public **String**()
- public **String**(String value)
- public **String**(char[ ] value)

String

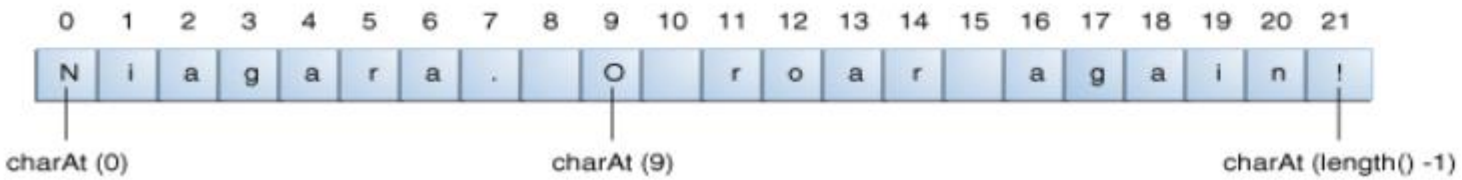
- **Comparing Strings**
  - Java provides a variety of methods to compare String objects, including:
    - public int **compareTo**(String str)
    - Compares the current String object to str, and returns **0** only if the two strings contain the same sequence of characters (case sensative).
    - A negative value is returned if the current String is lower in the Unicode set than the String str.
    - A positive value is returned if the current String is higher in the Unicode set than the String str.

String

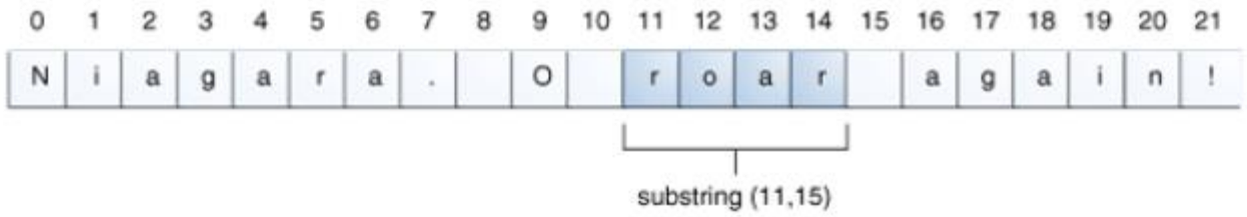
- **Comparing Strings**
  - public boolean **equals**(Object obj)
    - Compares the current String object with obj and returns true if obj is another String containing the same sequence of characters; false is returned otherwise.
  - public boolean **equalsIgnoreCase**(String str)
    - Performs a case-insensitive comparison of the current String object with str and returns true if str is another String containing the same sequence (ignoring case) of characters; false is returned otherwise.
- Use of the **==** operator only tests whether two String object references refer to the same object (memory space). The **==** operator does not test whether the contents of the two Strings are equal.

String

- Other useful String methods:
- public char **charAt**(int index)
  - Returns the character at the specified index. The index may range from 0 to length() - 1.
  - String str = "Niagara. O roar again!";



-
- public String **substring**(int beginIndex, int endIndex)
  - Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.



String

- public int **length**()
  - Returns the length of the current String. The length is equal to the number of 16-bit Unicode characters in the String.

    String s = new String(“MindTree”);

    System.out.println(s.length ()) ; // 8

- public int indexOf(int ch)
  public int lastIndexOf(int ch)
- public int indexOf(String str)
  public int lastIndexOf(String str)
    - Returns the index of the first (last) occurrence of the specified character/String.
  - public int indexOf(int ch, int fromIndex)
    public int lastIndexOf(int ch, int fromIndex)
  - public int indexOf(String str, int fromIndex)
    public int lastIndexOf(String str, int fromIndex)
    - Returns the index of the first (last) occurrence of the specified character/string, searching forward (backward) from the specified index.

Reference

- Refer:
  - Java Language Basics
    - http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html
  - For more information on the String class
    - http://docs.oracle.com/javase/6/docs/api/java/lang/String.html
  - For information on StringBuffer class
    - http://docs.oracle.com/javase/6/docs/api/java/lang/StringBuffer.html
  - For information on StringBuilder class
    - http://docs.oracle.com/javase/6/docs/api/java/lang/StringBuilder.html

Scope of a variable.

Local Variable

- Declared inside methods.
- *str* is local to the method *main*

Class Variable

- Declared outside methods but inside class. Denoted by `static` keyword.

Instance Variables

- Declared outside methods but inside class.

```
package com.banu.sample;

public class Example{
    /**
     * @param args
     */
    public static void main(String[] args) {

          String str = "This is a local variable";

    }
}

class Employee {
    private static int totalCount = 0;


    private String firstName;
    private int age;

}
```

Formatting Strings

- Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.

```
String fs;
fs = String.format("The value of the float "
    + "variable is %f, while "
    + "the value of the "
    + "integer variable is %d, "
    + " and the string is %s",
    floatVar, intVar, stringVar);
System.out.println(fs);
```

Formatting Strings

- System.*out.printf("Formatted output|%6d|%8.3f|%.2f%n",* 2004, Math.*PI, 1234.0354);*
  - Output (default Locale):
  - Formatted output| 2004| 3.142|1234.04

- System.*out.printf(Locale.FRENCH,*

  "Formatted output|%6d|%8.3f|%.2f%n",

  2004, Math.*PI, 1234.0354);*
  - Output (FRENCH Locale):
  - Formatted output| 2004| 3,142|1234,04

Formatting Strings

- // Argument Index: 1$, 2$, ...

```
String fmtYMD = "Year-Month-Day: %3$s-%2$s-%1$s%n";
String fmtDMY = "Day-Month-Year: %1$s-%2$s-%3$s%n";
System.out.printf(fmtYMD, 7, "March", 2004);
System.out.printf(fmtDMY, 7, "March", 2004);
```

Output:

- Year-Month-Day: 2004-March-7
- Day-Month-Year: 7-March-2004

- The optional *argument_index* is a decimal integer indicating the position of the
- argument in the argument list. The first argument is referenced by "1$", the second by
- "2$", and so on.

Widening Type Conversion

- Suppose a value of one type is assigned to a variable of another type.
  - Type1 t1;
  - Type2 t2 = t1;

  - Above statement is valid if
    - two types are compatible
    - destination type is larger then the source type

  - Example:
    - int i = 10;
    - double d = i;

Narrowing Type Conversion

- Narrowing Type Conversion happens when
  - two types are compatible
  - destination type is smaller then the source type

  - Example:

  - Java will not carry out type-conversion:

    ```
    int i = 120;
    ```
    ~~byte b = i;~~

  - Instead, we have to rely on manual type-casting:

    ```
    int I = 120;
    byte b = (byte) i;
    ```

Narrowing Type Conversion

- Example

```
int i = 257;
byte b = (byte) i;
System.out.println("Conversion of int to byte.");
System.out.println("i and b " + i + " ," + b); // i and b: 257, 1

double d = 123.456;
System.out.println("\ndouble to int.");
int value = (int) d;
System.out.println("d and value: " + d + " ," + value);
                                        // d and value: 123.456 ,123
```

Conditional Flow Control

- **IF STATEMENT**
  - In certain solutions, it may be necessary to write code to perform various actions based on certain criteria. To perform these actions, Java provides certain decision-making statements such as the if statement

if-statement has the form:

`if( boolean_expression)`
`      statement;`
	`if( boolean_expression ) {`
	`      statement1;`
	`      statement2;`
	`}`

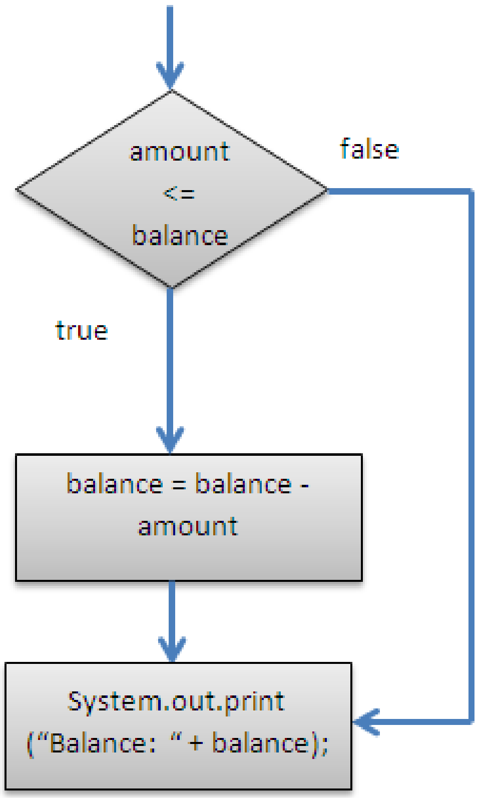
The if Statement

Example:

```
   int balance = 10000;
   int amount ;

    Scanner in = new Scanner(System.in);
    System.out.println("Enter Amount: ");
    double amount = in.nextDouble();

    if (amount <= balance) {
               balance = balance - amount;
      }
    System.out.print(“Balance:  “ + balance);
```



The if-else-if statement

Example:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter magnitude on the Richter scale");
double richter = scanner.nextDouble();
String r = “”;

if (richter >= 7.0)
        r = "Many buildings destroyed";
else if (richter >= 6.0)
        r = "Many buildings considerably damaged, some collapse";
else if (richter >= 4.5)
        r = "Damage to poorly constructed buildings";
else if (richter >= 3.5)
        r = "Felt by many people, no destruction";
else if (richter >= 0)
        r = "Generally not felt by people";
else
        r = "Negative numbers are not valid";

System.out.println("for richter scale " + richter + " result : " + r);
```

Switch Statement

- Switch statements provides a better alternative than if-else-if when the execution follows several branches depending on the value of an expression
- Syntax:

```
switch (expression) {
        case value1:
                        statement1;
                        break;
        case value2:
                        statement2;
                        break;
        case value3:
                        statement3;
                        break;
        default: statement;
}
```

Switch Statement

- Example:

```
int month = 4;  String season;
switch (month)  {
        case 12:
        case 1:
        case 2: season = "Winter"; break;
        case 3:
        case 4:
        case 5: season = "Spring"; break;
        case 6:
        case 7:
        case 8: season = "Summer"; break;
        case 9:
        case 10:
        case 11: season = "Autumn"; break;
        default: season = “unknown";
}
```

Answer this

- If y has the value 5 what will be the value of the variable y after the following piece of code is executed?

```
if (y > 0)
        y += 2;
```

- If p has the value 3 and max has the value 5, what will be the value of the variable max after the following piece of code is executed?

```
if (p < max) max = p;
```

- If x has the value 5.0 what will be the value of the variable count after the following piece of code is executed?

```
count = 0;
if (x < 0.0)
        negsum = negsum + x;
        count = count + 1;
```

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

- Java provides the following iteration statements:
  - while loop
  - do while loop
  - for loop
  - Enhanced for loop [for-each]

The while loop

- Syntax:

```
while (boolean_expression)
statement;
```

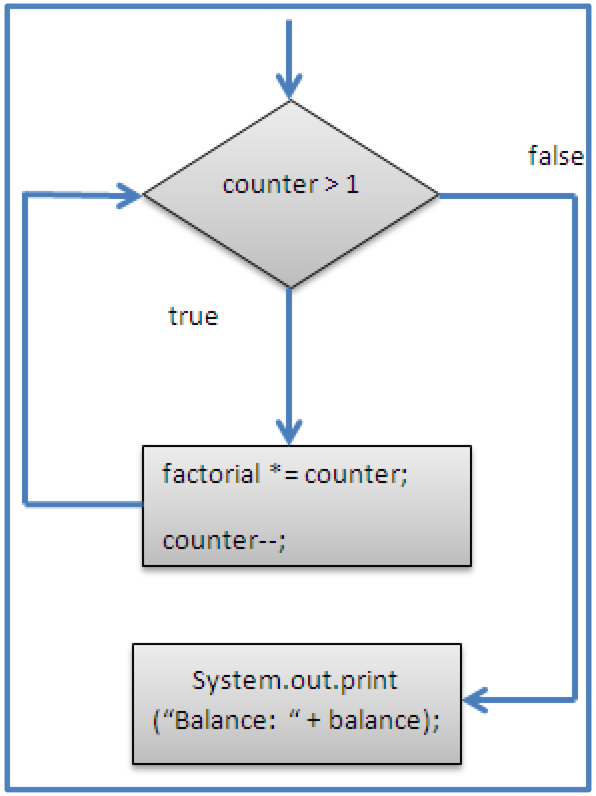
```
while (boolean_expression) {
        statement;
}
```

- Semantics:
  - repeat execution of statement until expression becomes false
  - expression is always evaluated before statement
  - if expression is false initially, statement will never get executed

The while loop example

- **Example:** while loop to calculate the factorial of the number 5

```
int counter = 5;
long factorial = 1;
while (counter > 1) {
      factorial *= counter;
      counter-- ;
    }
System.out.println(factorial);
```



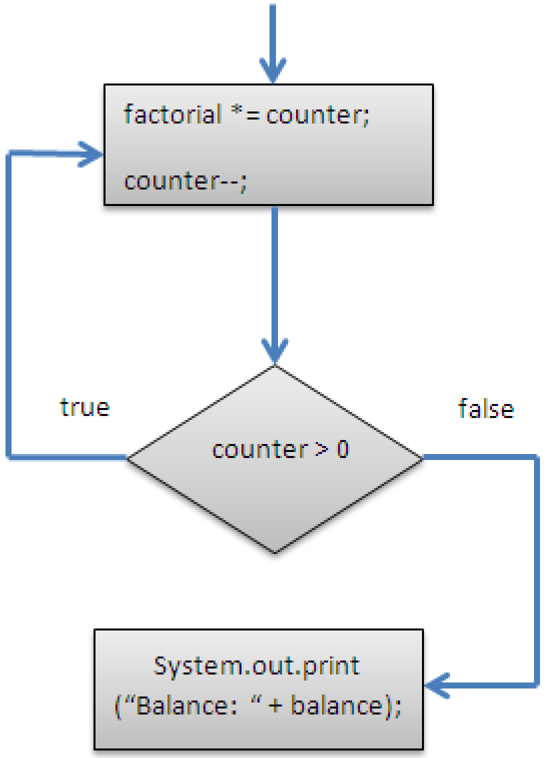
The do while loop

- If a component statement has to be executed at least once, the do-while statement is more appropriate than the while statement
- Syntax:

```
do  {
        statement;
 } while (condition);
```

- Example:

```
int counter = 5;
int factorial = 1;
do {
        factorial *= counter;
        counter--;
} while (counter > 0);
System.out.println(factorial);
```



The for loop

- When iterating over a range of values, for statement is more suitable to use then while or do-while

- Syntax:

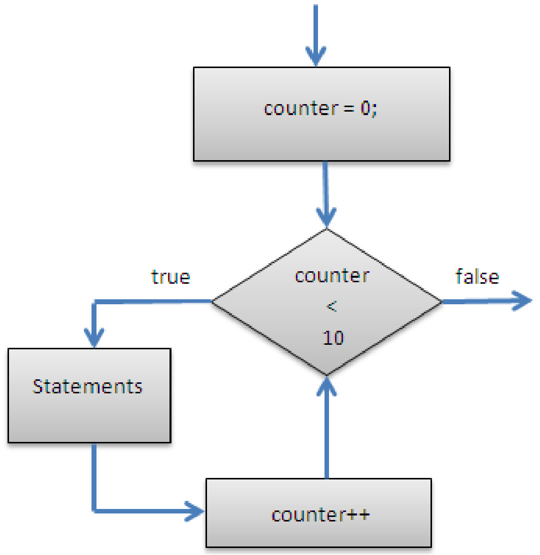
```
for (initialization; termination; increment)
        statement;
```

  - initialization statement is executed once before the first iteration
  - termination expression is evaluated before each iteration to determine when the loop should terminate
  - increment statement is executed after each iteration

The for loop

- The for statement may include declaration of a loop control variable:

```
for (int counter = 0;
              counter < 10;
              counter++) {
 Statements;
}
```



- The variable “counter” does not exist outside the for statement

The for loop example

- Find if the given number is a prime number.

```
int num = 14;
boolean isPrime = true;
for (int i=2; i < num/2; i++) {
        if ((num % i) == 0) {
                isPrime = false;
                break;                  //exit an iterative statement
        }
}
if (isPrime) {
        System.out.println("Prime");
}
else {
        System.out.println("Not Prime");
}
```

The For-Each Loop

- A for each-construct was introduced in JDK 1.5.0.
- It is also referred to as the "Enhanced for Loop"
  - Semantics:

```
for each item in collection
        do something to item
```

- The following method returns the sum of the values in an int array:

```
// Returns the sum of the elements of array arr
int sum(int [] arr) {
        int total = 0;
        for (int value : arr) {
                total += value;
        }
        return total;
}
```

Arrays

- Arrays are the simplest and most common type of structured data.
- An Array contains a group of items that are all of the same type and that are directly accessed through the use of an array index.

- One-Dimensional array:
  - A *one-dimensional array* is, essentially, a list of like-typed variables.
  - To create an array, you first must create an array variable of the desired type.
  - The general form of a one dimensional array declaration is

    type var_name[ ];

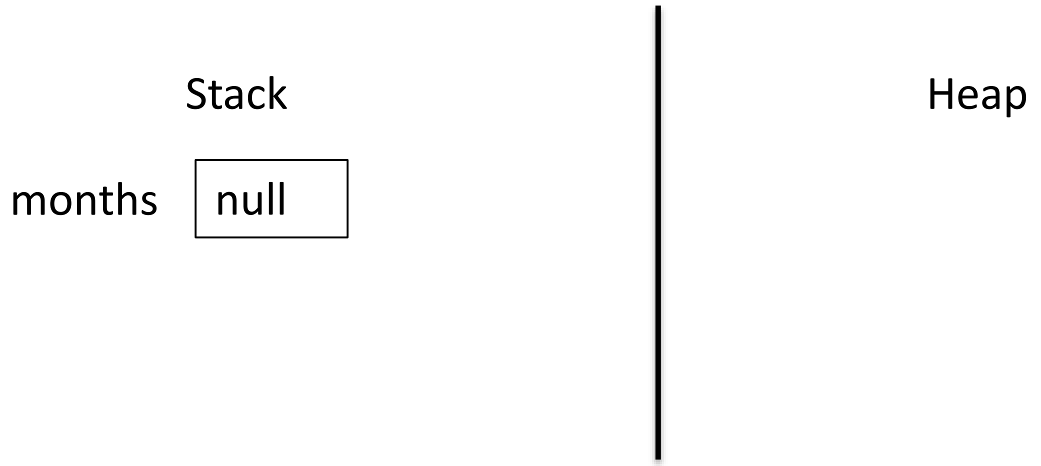
  - For example, the following declares an array named **months** with the type “array of int”:

    int months[];

Arrays

int months[];

- Although this declaration establishes the fact that **months** is an array variable, no array actually exists.
- In fact, the value of **months** is set to **null**, which represents an array with no value



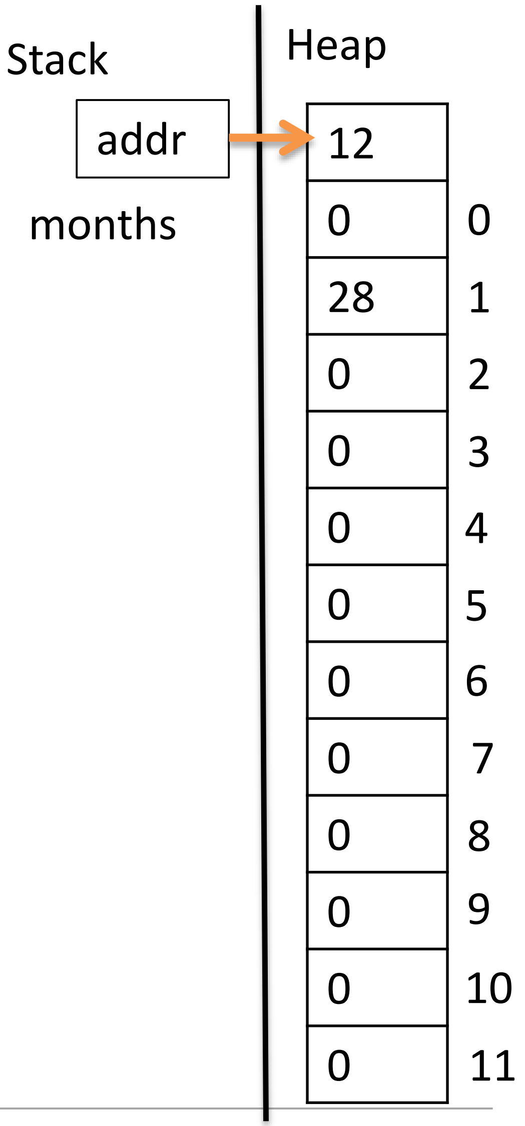
Arrays

- To link **months** with an actual, physical array of integers you must allocate one using **new** and assign it to **months**.
- This example allocates a 12-element array of integers and links them to **months**.

  int months = new int[12];

- You can access a specific element in the array by specifying its index within square brackets.
- For example, this statement assigns the value 28 to the second element of **months**.
- months [1] = 28;

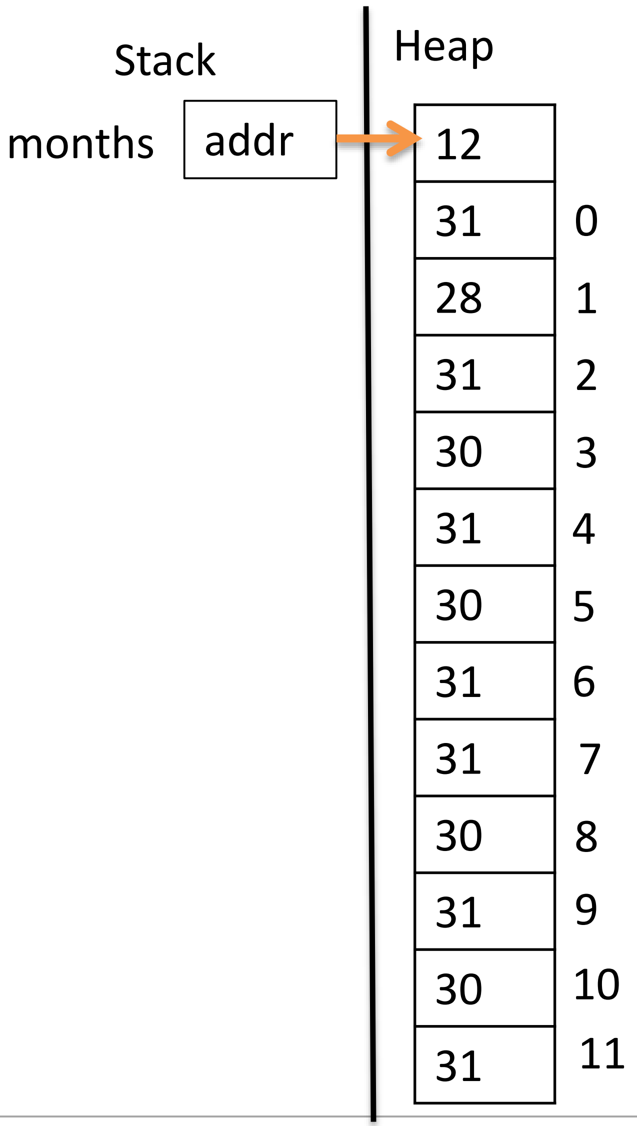
addr: address of physical array of integers



Arrays

- Example:

```
int months[] = new int[12];
months[0] = 31;
months[1] = 28;
months[2] = 31;
months[3] = 30;
months[4] = 31;
months[5] = 30;
months[6] = 31;
months[7] = 31;
months[8] = 30;
months[9] = 31;
months[10] = 30;
months[11] = 31;
System.out.println("April has " + months[3] + " days.");
```



Exiting Loops early

- break
  - The *break* statement causes a loop to terminate through the normal exit channel; the program resumes execution at the first statement following the loop.

- continue
  - The *continue* statement rather than causing a loop exit, causes the program to skip the loop body and continue executing at the beginning of the next iteration of the loop.
  - A *continue* statement is shorthand for an *if-else* clause that would prevent the rest of the loop from being executed.

Exiting Loops early

- Example:

```
int i = 0;
while(i < 100) {
        if(i == 10) {
                break; // terminate loop if i is 10
        }
        System.out.println("i: " + i);
        i++;
}
System.out.println("Loop complete.");
```

Continue example

```
for(int i=0; i<10; i++) {
    System.out.print(i + " ");
    if (i%2 == 0) {
        continue;
        }
    System.out.println("");
}
```

- Output :

```
0 1
2 3
4 5
6 7
8 9
```

Using continue with a label.

```
outer:
    for (int i=0; i< 5; i++) {
        for(int j=0; j< 5; j++) {
            if(j > i) {
                System.out.println();
                continue outer;
            }
            System.out.print(" " + (i * j));
        }
    }
```

Output:

```
 0
 0 1
 0 2 4
 0 3 6 9
 0 4 8 12 16
```

Variable Parameter Lists (“varargs”)

- VarArgs
  - Assume you want to invoke a method with variable number of arguments.
  - The option we had on hand in Java 1.4 or earlier was to pass an array. Let’s consider a simple example as shown below:

```
 public static int findMax(int[ ] values) {
   int max = values[0];
                  for(int value: values) {
                                  if (value> max) {
                                                   max = value;
                                  }
                  }
   return max;
 }
```

- We can invoke this method by passing an array with different number of values as
- illustrated below:
  - findMax(**new int[]** {31, 27, 2,29, 8});
  - findMax(**new int[]**{88, 12, 87, 223, 1, 3, 6});
- In order to invoke a method that takes variable number of arguments, we had to bundle the parameters into an array

Variable Parameter Lists (“varargs”)

- The varargs concept in Java 5 does not suffer from these issues. It is not only elegant, it also is very type safe

```
public static int findMax(int… values) {                // change int[] to int…
  int max = values[0];
                for(int value: values) {
                              if (value> max) {
                                           max = value;
                              }
                }
  return max;
}
```

- A type followed by … is simply a syntax–it’s nothing but an array
- For the new code example, now I can call the modified max method as follows:
- findMax(new int[] {1, 7, 2, 9, 8});
- findMax(new int[]{8, 12, 87, 23, 1, 3, 6, 9, 37});
- findMax(1, 7); // is compiled into: findMax(new int[] {1, 7});
- findMax(8, 12, 87, 23, 1, 3, 6); // is compiled into: findMax(new int[] {8, 12, 87, 23, 1, 3, 6});

Enumerated Types

- An enumerated type defines a *finite set of symbolic names and their values*.
- Standard approach is the int *enum pattern* (or the analogous String *enum pattern*):

```
public class MachineState {
        public static final int BUSY = 1;
        public static final int IDLE = 0;
        public static final int BLOCKED = -1;
        //...
}
```

```
public class Machine {
        int state;
        public void setState(int state) {
                this.state = state;
        }
        //...
}
```

- Machine machine = new Machine();
- **machine.setState(MachineState.BUSY); // (1) Constant qualified by class name**
- **machine.setState(1); // Same as (1)**
- **machine.setState(5); // Any int will do.**
- **System.out.println(MachineState.BUSY); // Prints "1", not "BUSY".**

Enumerated Types

- **Disadvantages of the int Enum Pattern**
- Not typesafe.
  - Any int value can be passed to the setState() method.
- Uninformative textual representation.
  - Only the value can be printed, not the name.
  - Constants compiled into clients.
  - Clients need recompiling if the constant values change
- Cannot be used in switch statement

Typesafe Enum Construct

- The enum construct provides support for enum types:

  enum MachineState { BUSY, IDLE, BLOCKED }

  enum Day {

  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

  }

  - Keyword enum is used to declare an *enum type*.
  - Overcomes all the disadvantages of the int enum pattern, and is more powerful

Enumerated Types

- **Enum Constructors**
  - Each constant declaration can be followed by an argument list that is passed to the constructor of the enum type having the matching parameter signature.

```
public enum Meal {
    BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);
    Meal(int hh, int mm) {
        this.hh = hh;
        this.mm = mm;
}
    private int hh; private int mm;
    public int getHour() { return this.hh; }
    public int getMins() { return this.mm; }
}
```

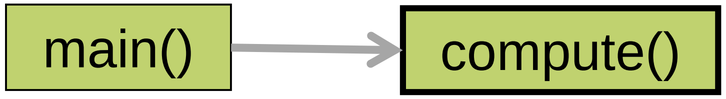
Enumerated Types

- **Enums in a switch statement**
- The switch expression can be of an enum type, and the case labels can be enum constants of this enum type.

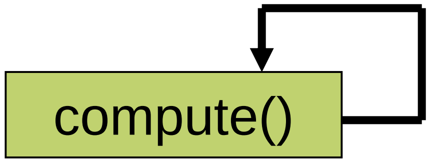
```
MachineState state = machine.getState();
  switch(state) {
            case BUSY:
            System.out.println(state + ": Try later."); break;
            case IDLE:
            System.out.println(state + ": At your service."); break;
            case BLOCKED:
            System.out.println(state + ": Waiting on input."); break;
  }
```

Recursion

- "Normally", we have methods that call other methods.
  - For example, the `main()` method calls the compute`()` method.



- The idea of calling one function from another immediately suggests the possibility of a function calling *itself*.
- This function-call mechanism is known as *recursion*.



Recursion

- Why use Recursive Methods?
  - some problems are more easily solved by using recursive methods for example:
    - Traversing through a directory or file system.
    - Traversing through a tree of search results.
    - Some sorting algorithms recursively sort data
- Rules of Recursion
  - **Base case**: You must always have some base case which can be solved without recursion
  - **Making Progress**: For cases that are to be solved recursively, the recursive call must always be a case that makes progress toward the base case

Recursion

- Example
  - Computing factorials are a classic problem for examining recursion.
  - A factorial is defined as follows:

    n! = n * (n-1) * (n-2) …. * 1;
  - For example:

    1! = 1 (Base Case)

    2! = 2 * 1 = 2

    3! = 3 * 2 * 1 = 6

    4! = 4 * 3 * 2 * 1 = 24

    5! = 5 * 4 * 3 * 2 * 1 = 120

Recursion

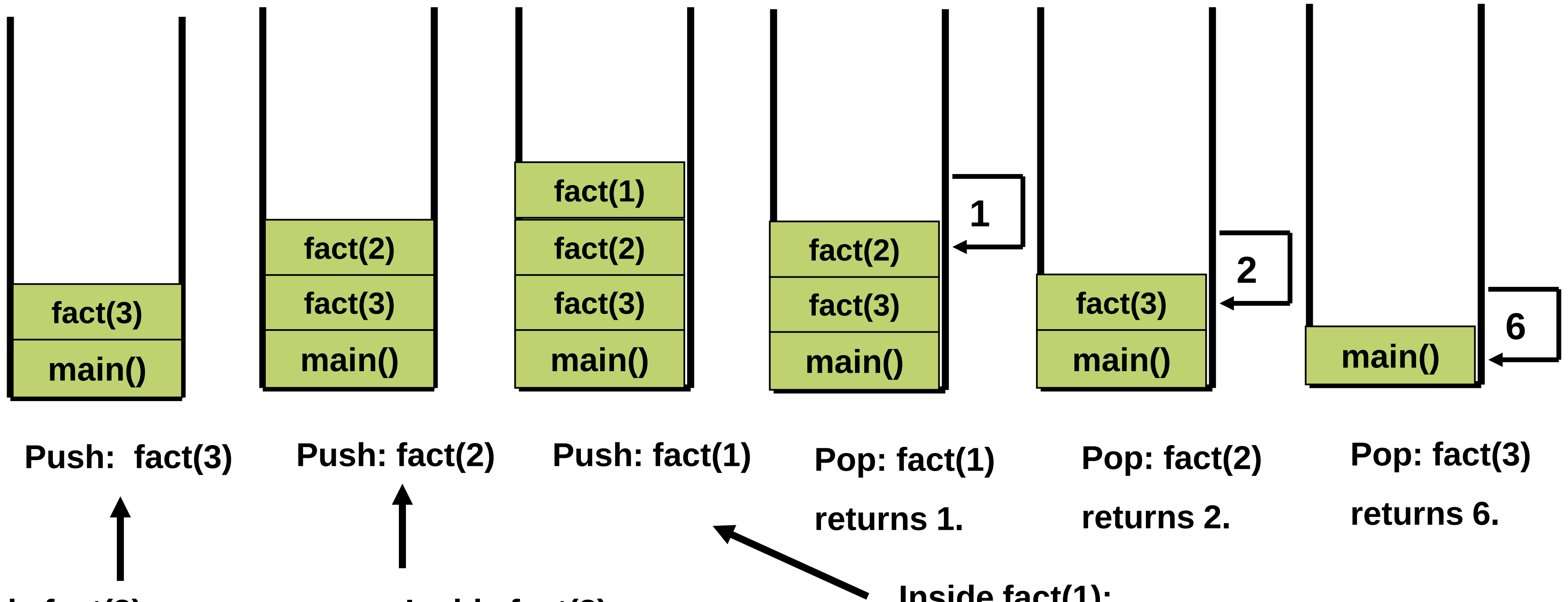
```
public class Example
{
        public static void main (String args[ ])      {
                System.out.println (fact(3));
        }
   }

   public static int fact(int number)        {
        if (( number == 1) || (number == 0))
                return 1;
        else
                return (number * fact(number-1));
   }
}
```

Base Case.

Making progress

Recursion



Inside fact(3):

if (number <= 1) return 1;

else return (3 * fact (2));

Inside fact(2):

if (number <= 1) return 1;

else return (2 * fact (1));

Inside fact(1):

if (number <= 1) return 1;

else return (1 * fact (0));

Recursion

- **Pitfalls of recursion.**
  - *Missing base case:* If you call this function, it will repeatedly call itself and never return.
  - *Excessive space requirements.* If a function calls itself recursively an excessive number of times before returning, the space required by Java for this task may be too much. Can lead to StackOverflowError.