

Module 4

Converting Design to Code in JAVA

Java's History

- James Gosling and Sun Microsystems, 1992
- Initially “Oak”
- Java, May 20, 1995
- Aimed to develop platform independent language
- To be used in consumer electronic devices
- HotJava
 - The first Java-enabled Web browser

Why Java?

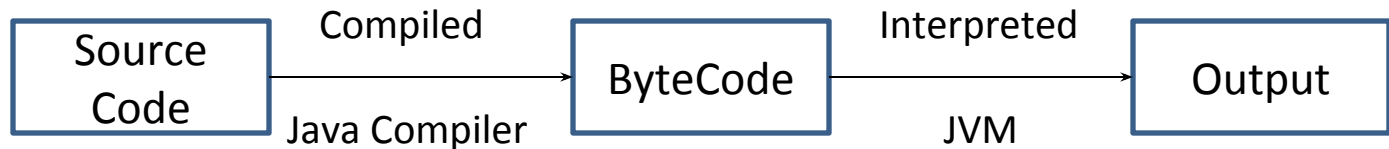
- Enables Users
 - To develop and deploy applications on the Internet
 - For servers, desktop computers, hand- held devices
- Java is a general purpose programming language.
- Java is the Internet programming language.

Java, Web, and Beyond

- Java can be used to develop Web applications.
 - Java Applets
 - Java Servlets and JavaServer Pages
- Java can also be used to develop applications for hand-held devices such as Palm and cell phones
- Why Java is important to the Internet
 - Java applets and applications
 - Security – no virus infection
 - Portability- portable code

Java's Magic : The Byte-Code

- Java is compiled as well as interpreted language
- Highly optimized set of instructions - executed by JVM
- Provides portability- need only JVM to execute
- Provides Security- all control of JVM



Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java is partially modeled on C++, but greatly simplified and improved. Some people refer to Java as "C++--" because it is like C++ but with more functionality and fewer negative aspects.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java is inherently object-oriented. Although many object-oriented languages began strictly as procedural languages, Java was designed from the start to be object-oriented. Object-oriented programming (OOP) is a popular programming approach that is replacing traditional procedural programming techniques.

One of the central issues in software development is how to reuse code. Object-oriented programming provides great flexibility, modularity, clarity, and reusability through encapsulation, inheritance, and polymorphism.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Distributed computing involves several computers working together on a network. Java is designed to make distributed computing (e.g. *Web Services*) easy. Since networking capability is inherently integrated into Java, writing network programs is like sending and receiving data to and from a file.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

You need an interpreter to run Java programs. The programs are compiled into the Java Virtual Machine code called bytecode. The bytecode is machine-independent and can run on any machine that has a Java interpreter, which is part of the Java Virtual Machine (JVM).

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java compilers can detect many problems that would first show up at execution time in other languages.

Java has eliminated certain types of error-prone programming constructs found in other languages.

Java has a runtime exception-handling feature to provide programming support for robustness.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java implements several security mechanisms to protect your system against harm caused by stray programs.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Write once, run anywhere

With a Java Virtual Machine (JVM), you can write one program that will run on any platform.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Because Java is architecture neutral, Java programs are portable. They can be run on any platform without being recompiled.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Because Java is architecture neutral, Java programs are portable (moveable). They can be run on any platform without being recompiled.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Multithread programming is smoothly integrated in Java, whereas in other languages you have to call procedures specific to the operating system to enable multithreading.

Characteristic of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java was designed to adapt to an evolving environment. New code can be loaded on the fly without recompilation. There is no need for developers to create, and for users to install, major new software versions. New features can be incorporated transparently as needed.

JDK Versions

- JDK 1.0 (Jan. 1996)
- JDK 1.1 (Feb. 1997)
- JDK 1.2 (Dec. 1998)
- JDK 1.3 (May 2000)
- JDK 1.4 (Feb. 2002)
- JDK 1.5 (Sep. 2004)
- JDK 1.6 (Dec. 2006)
- JDK 1.7 (July 2011)
- JDK 1.8 (Mar. 2014)
- JDK 9 (Sep. 2017)
- JDK 10 (Mar. 2018)
- JDK 11 (Sep. 2018)
- JDK 12 (Mar. 2019)
- JDK 13 (Sep. 2019)
- JDK 14 (Mar. 2020)
- JDK 15 (Sep. 2020)

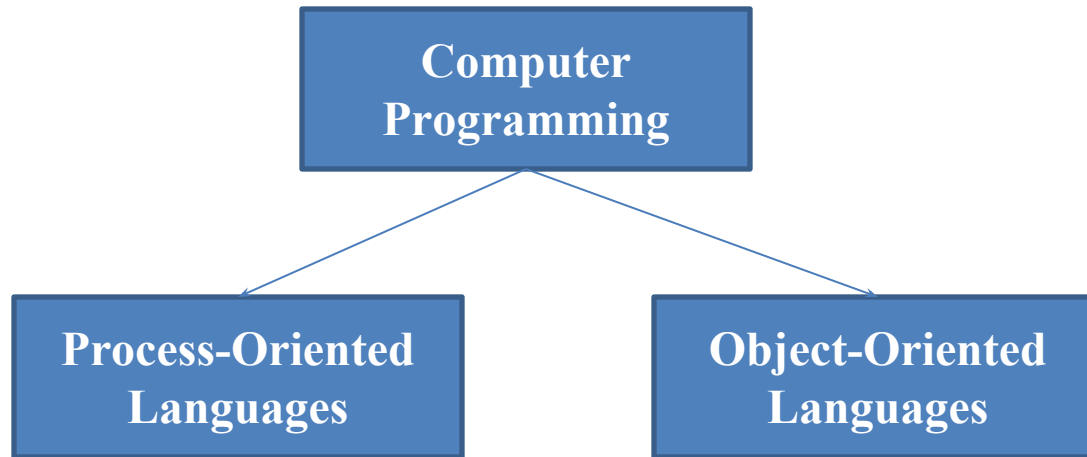
JDK Editions

- Java Standard Edition (J2SE)
 - J2SE can be used to develop client-side standalone (independant) applications or applets.
- Java Enterprise Edition (J2EE)
 - J2EE can be used to develop server-side applications such as Java servlets and Java ServerPages.
- Java Micro Edition (J2ME).
 - J2ME can be used to develop applications for mobile devices such as cell phones.

Java IDE Tools

- Borland JBuilder
- NetBeans Open Source by Sun
- Sun ONE Studio by Sun Microsystems
- Eclipse Open Source by IBM

Object-Oriented Programming



- ✓ **What is Happening**
- ✓ **Series of linear steps**
- ✓ **Code acting on data**

- ✓ **Who is being affected**
- ✓ **Well defined interfaces**
- ✓ **Data controlling access to code**

Object-Oriented Programming

1. Abstraction

- Data hiding
- Hierarchical classification
- Manage complexity by breaking the system into more manageable pieces

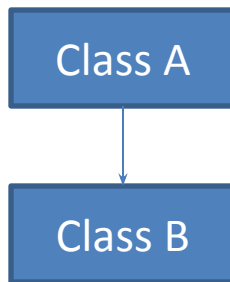
2. Encapsulation

- “wrap the data”
- Binds code and data
- Class, objects, methods
- Public and private interfaces

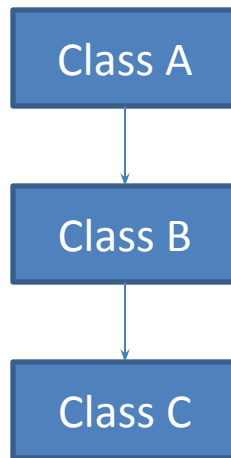
Object-Oriented Programming

3. Inheritance

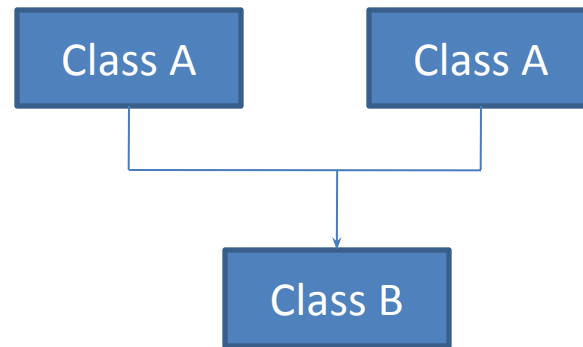
- Process by which one object acquires the properties of another object
- Object need only define those qualities that make it unique within its class
- Inherit its general properties from its parent
- Different types of inheritance



Single Level



Multi Level



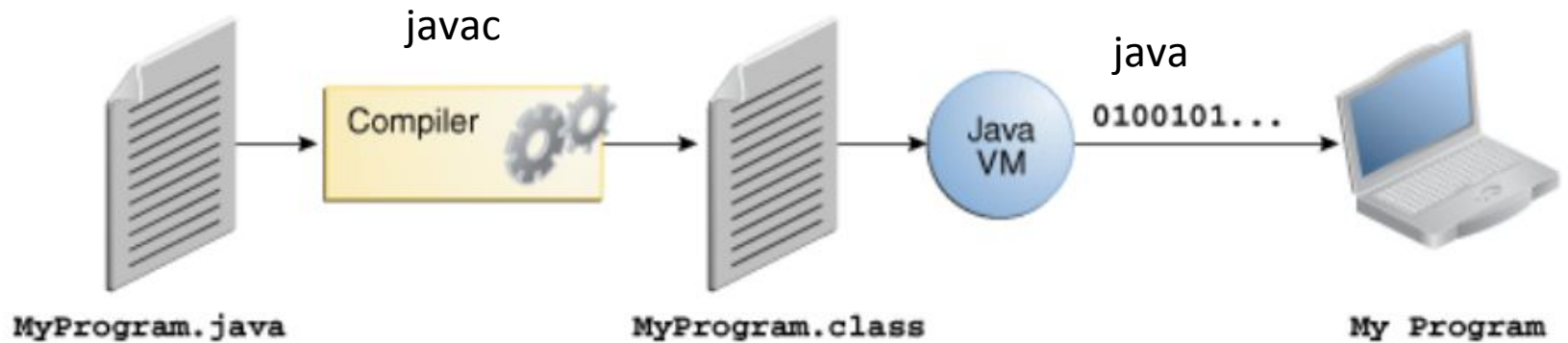
Multiple Inheritance

Object-Oriented Programming

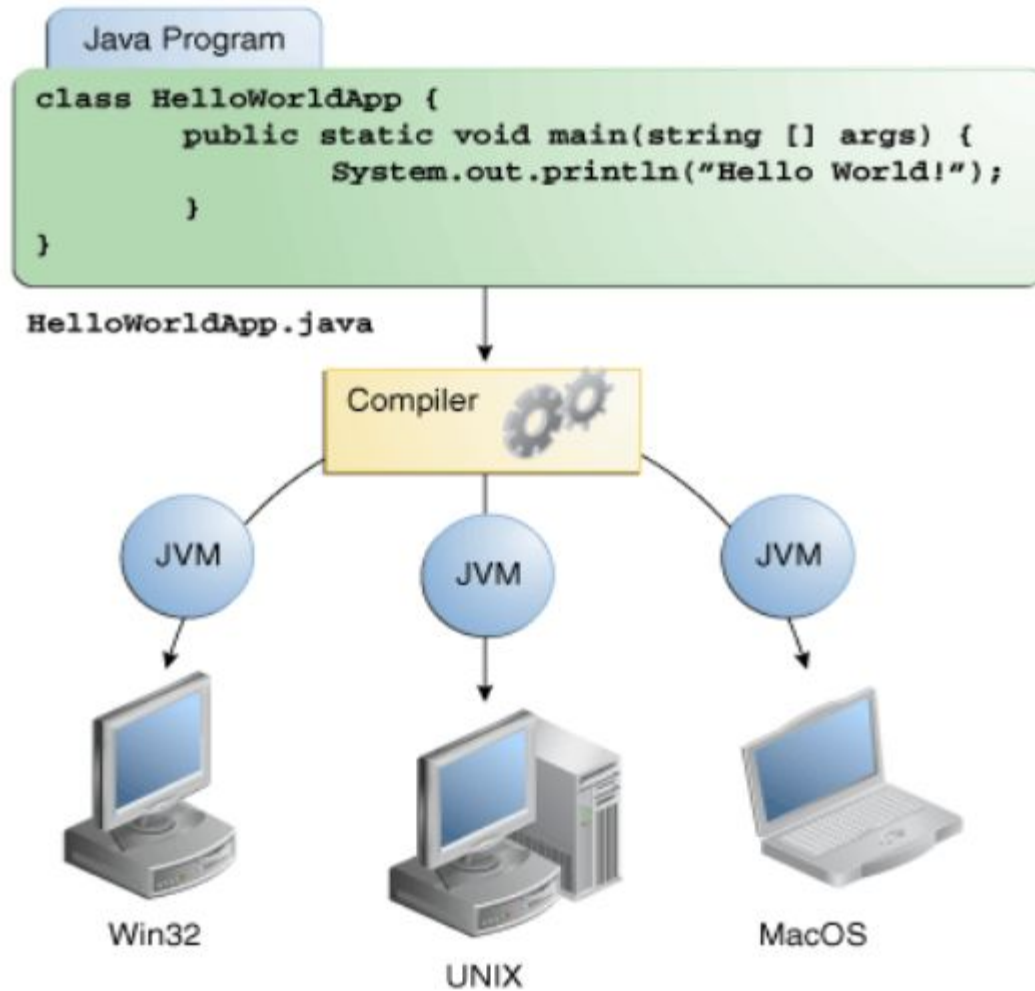
4. Polymorphism

- “one interface multiple methods”, many forms
- Possible to design a generic interface to a group of related activities
- Reduce complexity
- Compiler selects the specific action as it applies to each situation

Java Program



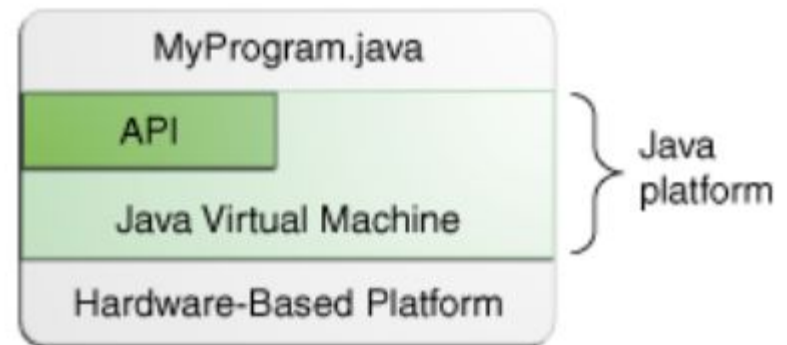
Java Program



The Java Platform

- A *platform* is the hardware or software environment in which a program runs. For ex. Microsoft Windows, Linux, Solaris OS, and Mac OS.
- The Java platform has two components:
 - The *Java Virtual Machine*
 - The *Java Application Programming Interface (API)*

The API is a large collection of ready-made software components that provide many useful capabilities



"Hello World!" for Microsoft Windows

- **Checklist**
 - The Java SE Development Kit 8 (JDK 8)
 - A text editor
- **Creation of first program**
 - Create a source file
 - Compile the source file into a .class file
 - Run the program

A Simple Java Program (*welcome.java*)

```
/**
 * The HelloWorldApp class implements an
 * application that simply prints "Hello World!"
 * to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //
        Display the string.

    }
}

> javac Welcome.java
> java HelloWorldApp 1 2 3 4 5
```

A closer look

- Source code comments
- Class definition
- Main method

Comments

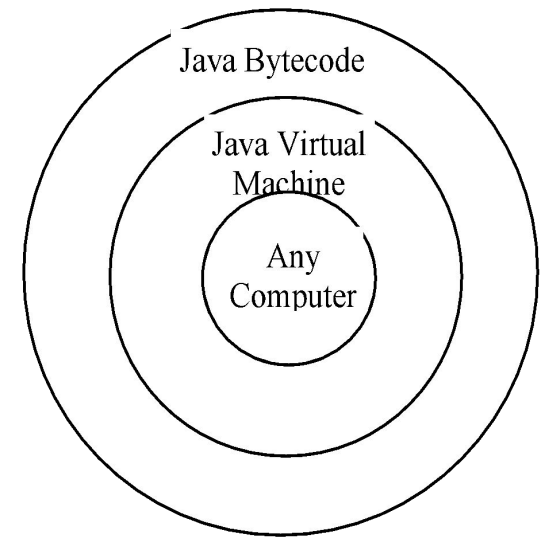
- Ignored by the compiler
- 3 type:
 - `/* text */` ☐ The compiler ignores everything from `/*` to `*/`
 - `/** documentation */` ☐ generally used by Javadoc
 - `// text` ☐ The compiler ignores everything from `//` to the end of the line.

Compiling JAVA Source Code

- Compile the welcome program by executing the compiler, **javac**
C:\> javac welcome.java
- The **javac** compiler creates a file called **welcome.class** - bytecode version of the class

- Use java interpreter , called **java** to run the program
- The *bytecode* can run on any computer with a Java Virtual Machine

C:\>java welcome



Trace a Program Execution

Enter main method

```
//This program prints Welcome to Java!  
public class Welcome  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Trace a Program Execution

Execute statement



```
//This program prints Welcome to Java!  
public class Welcome  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Trace a Program Execution

```
//This program prints Welcome to Java!  
public class Welcome  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Welcome to Java!");  
    }  
}
```



print a message to the console

Anatomy of a Java Program

- Comments
- Reserved words
- Modifiers
- Statements
- Blocks
- Classes
- Methods
- The main method

Comments

- The contents of a comment are ignored by the compiler.
- Java supports three styles of comments:
 - Single-line comment — starts with two slashes(//)
 - when the compiler sees //, it ignores all text after // in the same line
 - Multiline comment — enclosed between /* and */
 - when compiler sees /*, it scans for the next */ and ignores any text between /* and */
 - Documentation comment — use to produce an HTML file.
 - enclosed between /** and */

Reserved words

- Reserved words or keywords are words that have a specific meaning to the compiler and cannot be used for other purposes in the program.
- Example - public, static, void etc.

(Visibility or Access) Modifiers

- Java uses certain reserved words called modifiers that specify the properties of the data, methods, and classes and how they can be used.
- Examples of modifiers are public and static.
- Other modifiers are private, final, abstract, and protected.

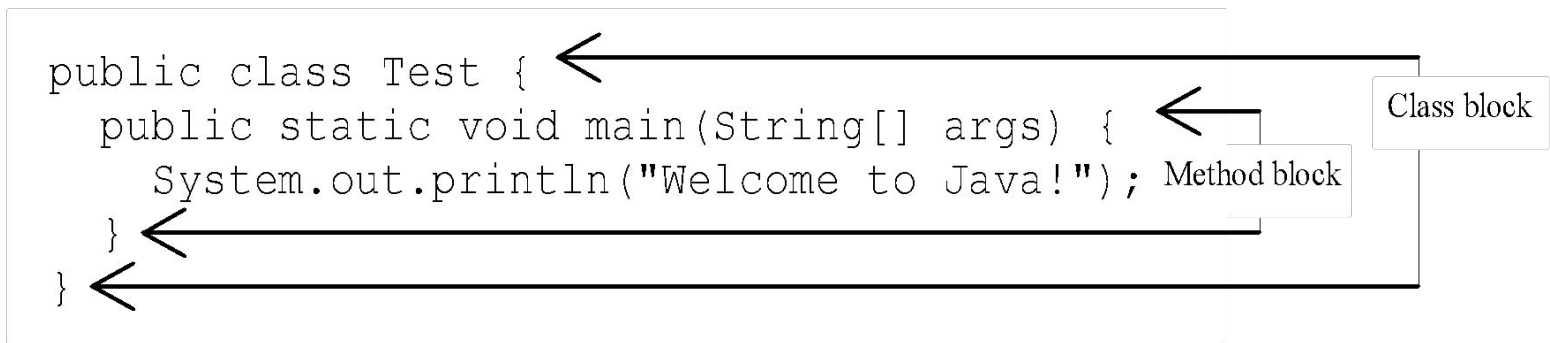
public > protected > package (default) > private

Statements

- A statement represents an action or a sequence of actions.
- The statement `System.out.println("Welcome to Java!")` in the program is a statement to display the greeting "Welcome to Java!"
- Every statement in Java ends with a semicolon (;).

Blocks

- A pair of braces in a program forms a block that groups components of a program.



Classes

- The class is the essential Java construct.
- A class is a template or blueprint for objects.
- Keyword **class** is used to declare a new class.
- The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

Methods

- A collection of statements that performs a sequence of operations and can be reuse with different arguments.
- There is no need to write all the statements again and again.

Using a single line call statement to execute a collection of statements.

main Method

- The main method provides the control of program flow. The Java interpreter executes the application by invoking the main method.
- The main method looks like this:

```
public static void main(String[] args)  
{  
    // Statements;  
}
```

Contd...

- All Java applications begin execution by calling **main()**
- **public** – an access specifier. Public member can be accessed by code outside the class. `main()` is declared public, since it must be called outside of its class when the program is started.
- **static** – allows `main` to be called without having to instantiate a particular instance of the class. `main()` is called by java interpreter before any objects are made.

Contd...

- **void** – simply tells the compiler that `main()` does not return a value
- **String args[]** – parameter for `main()` method. It declares an array of instances of the class `String` that stores character strings. `args` receives any command line arguments when program is executed.

The Java Keywords

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

Command Line Arguments

- These are the parameters that are supplied to the application program at the time of invoking it for execution.
 - *Public static void main (String args[])*
- Any arguments provided in the command line are passed to the array **args**.

C:> *Java example hello world*

- Assigned to the array **args** as follows:
 - **Args[0]** = *hello*
 - **Args[1]** = *world*

Constants

Refer to fixed values that do not change during the execution of a program.

- **Integer Constants:** sequence of digits

123 037 0X2

- **Real Constants:** numbers containing fractional parts

0.0083 -0.75 435.36

- **Single character Constants:** single character

'5' 'A' ';'

- **String Constants:** sequence of characters

"Hello World" "?+;;;....#"

- **Backslash character (Symbolic) Constants:** used in output methods

'\n' '\b' '\t' '\'' '\\'' '\\'

Symbolic Constants

- Some constants may appear repeatedly in a number of places in the program.
- These constants can be defined as a symbolic name
- Make a program:
 - easily modifiable
 - More understandable


final type symbolic name = value

*ex: final float **PI** = 3.14159*

Variables

- An identifier that denotes a storage location used to store a data value.
- May take different values at different times during the execution of the program
- ***Naming Conventions:*** may consists of alphabets, digits, underscore and dollar with following conditions
 - Must not begin with a digit
 - Uppercase and Lowercase are distinct.
 - Should not be a keyword
 - Space is not allowed

Scope of Variables

 Classified into three kinds:

- **Instance Variables**

- Created when the objects are instantiated
- Take different values for each object

- **Class Variables**

- Global to a class
- Belong to the entire set of objects that class creates
- Only one memory location is created

- **Local Variables**

- Used inside methods or inside program blocks.
- Blocks are defined between opening brace { and a closing brace }
- Visible to program only from the beginning to the end of the block.

Data Types

 Data type specify the size and type of values

Java defines eight simple types of data. These can be put into four groups:

1. Integers
2. Floating-point numbers
3. Characters
4. Boolean

Data Types contd...

Integers

Name	Width(Size)	Range
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-2^{64-1} to $2^{64-1} - 1$

Floating-point

Name	Width(Size)	Range
float	32 bits	1.4e-045 to 3.4e+038
double	64 bits	4.9e-324 to 1.8e+038

Data Types contd...

Characters

- Data type used to store character is **char**.
- It requires 16 bits.
- Range of a char is 0 to 65536 (no negative char)

Boolean

- Used to test a particular condition
- It can take only two values: **true** and **false**
- Uses only 1 bit of storage
- All comparison operators return boolean value

Control Statements

- Control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic.
- Java contains following types of control statements
 - Selection statements
 - Iteration statements
 - Branching statements

Selection Statements

- **If statement**

- This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed.

Syntax:

```
if(conditional_expression)
{
    <statements>;
    ...;
    ...;
}
```

Example: If $n\%2$ evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence **"This is even number"** is printed on the screen.

```
int n = 10;
if(n%2 == 0){
    System.out.println("This is even number");
}
```

Selection Statements

- **If-else Statement**

- The **"if-else"** statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression)
{
    <statements>;
    ...;
    ...;
}
else{
    <statements>;
    ....;
    ....;
}
```

Example: If $n\%2$ doesn't evaluate to 0 then else block is executed. Here $n\%2$ evaluates to 1 that is not equal to 0 so else block is executed. So **"This is not even number"** is printed on the screen.

```
int n = 11; if(n%2 == 0){
    System.out.println("This is even number");
}
else{
    System.out.println("This is not even
number");
}
```

Selection Statements

- **Switch Statement**

Syntax:

```
switch(control_expression)
{
    case expression 1:
        <statement>;
    case expression 2:
        <statement>;
    ...
    ...
    case expression n:
        <statement>;
    default:
        <statement>;
} //end switch
```

Example: Here expression "day" in switch statement evaluates to 5 which matches with a case labeled "5" so code in case 5 is executed that results to output **"Friday"** on the screen.

```
int day = 5; switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thrusday"); break;
    case 5: System.out.println("Friday"); break;
    case 6: System.out.println("Saturday"); break;
    case 7: System.out.println("Sunday"); break;
    default: System.out.println("Invalid entry");
             break;
}
```

Iteration Statements

- **while loop statements**

- It executes a block of code or statements till the given condition is true. The expression must be evaluated to a boolean value.

Syntax:

```
while(expression){  
    <statement>;  
    ...;  
    ...;  
}
```

Example: prints values 1 to 10 on the screen.

```
int i = 1;//print 1 to 10  
while (i <= 10){  
    System.out.println("Num " + i);  
    i++;  
}
```

Iteration Statements

- **do-while loop statements**
 - First the **do** block statements are executed then the condition given in **while** statement is checked. So in this case, even the condition is false in the first attempt, do block of code is executed at least once.

Syntax:

```
do{  
<statement>;  
...;  
...;  
}while (expression);
```

Example: prints values 1 to 10 on the screen.

```
int i = 1;do{  
    System.out.println("Num: " + i);  
    i++;  
}while(i <= 10);
```

Iteration Statements

- **for loop statements**

Syntax:

for (initialization; condition; increment or decrement)

```
{  
    <statement>;  
    ...;  
    ...;  
}
```

initialization: The loop is started with the value specified.

condition: It evaluates to either 'true' or 'false'. If it is false then the loop is terminated.

increment or decrement: After each iteration, value increments or decrements.

Example: prints values 1 to 10 on the screen.

```
for (int num = 1; num <= 10;  
    num++)  
{ System.out.println("Num: "  
    + num);  
}
```

Branching Statements

- **Break statements**

- The break statement is a branching statement that contains two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for) . It also terminates the switch statements.

- **Syntax:**

- break; // breaks the innermost loop or switch statement.
 - break label; // breaks the outermost loop in a series of nested loops.

Branching Statements

- **Break statements**

Using break to exit a loop

```
Class breakloop
{
public static void main( String a[])
{
    for(int i=0;i<100;i++)
    {
        if(i==10) break;
        System.out.println("i:"+i);
    }
    System.out.println("loop complete");
}
}
```

Using break as form of goto

```
Class Break{
public static void main( String a[]){
boolean t=true;
First: {
    second: {
        Third: {
            System.out.println("before break);
            If(t) break second;
            System.out.println("this won't execute);
        }
        System.out.println("this won't execute);
    }
    System.out.println("this is after second
block");}}}}
```


Branching Statements

- **Continue statements**

- This is a branching statement that are used in the looping statements (while, do-while and for) to skip the current iteration of the loop and resume the next iteration .

- **Syntax:**
 continue;

Example:

```
Class Continue{  
public static void main( String a[]){  
    for(int i=0;i<10;i++){  
        System.out.println(i+ "");  
        if(i%2==0) continue;  
        System.out.println(" ");  
    }  
}}
```

Branching Statements

- **return**
 - return statement can be used to cause execution to branch back to the caller of the method.

Example:

```
Class Return{  
    public static void main( String a[]){  
        boolean t= true;  
        System.out.println("before the return");  
        if(t) return;  
        System.out.println("This won't execute ");  
    }  
}
```

Thank you !!