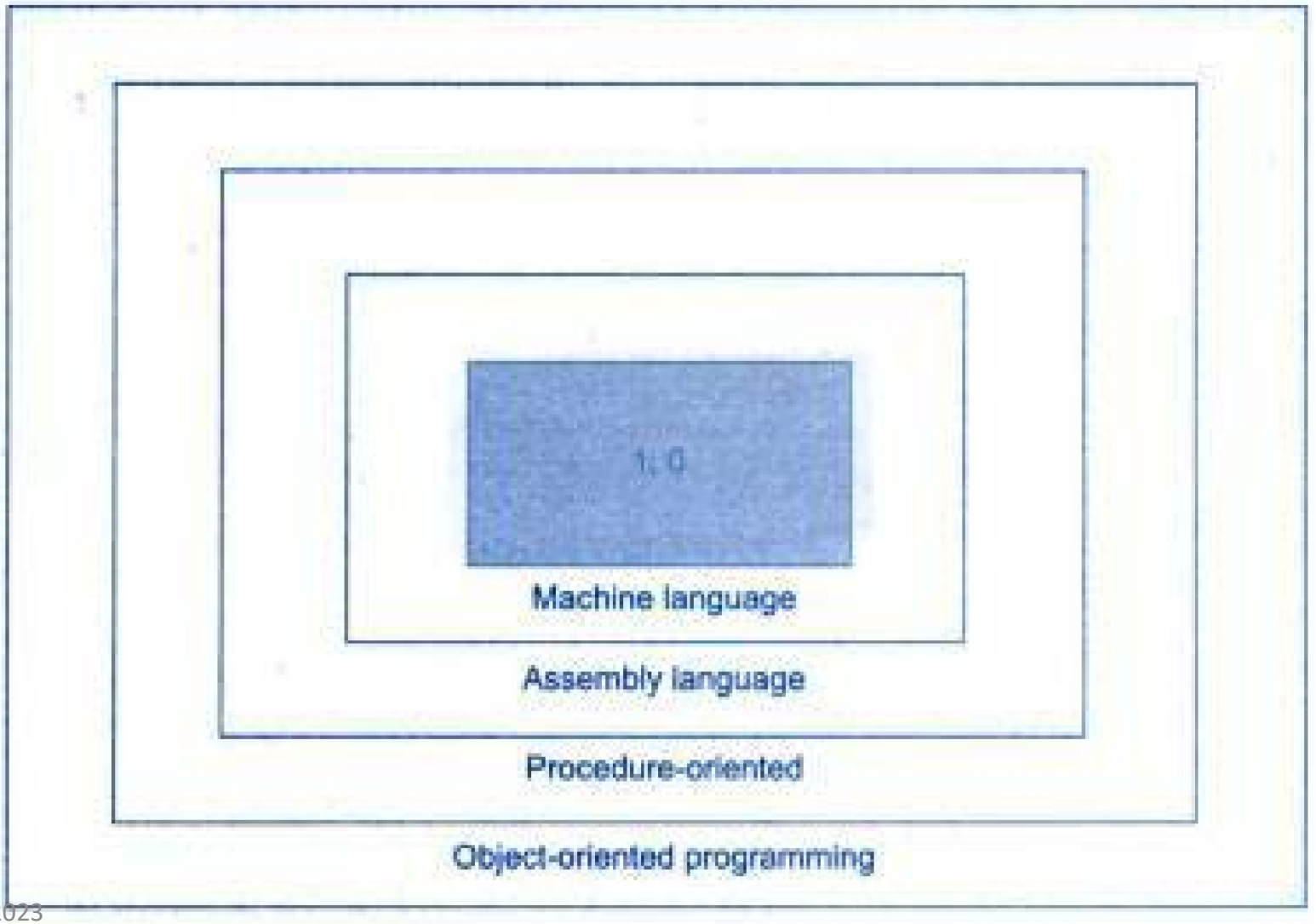
Java Programming Language (BCA 324)

Layers of Computer Software



- Computer language innovation and development occurs for two fundamental reasons:

1. To adapt to changing environments and uses
2. To implement refinements and improvements in the art of programming

Java is related to C++, which is a direct descendant of C.

Much of the character of Java is inherited from these two languages.

From C, Java derives its syntax.

Many of Java's object-oriented features were influenced by C++.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.

It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995.

- Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++.”
- Java was influenced by C++, it is not an enhanced version of C++.
- Java was not designed to replace C++. Java was designed to solve a certain set of problems.
- C++ was designed to solve a different set of problems.
- Both will coexist for many years to come.
- Java’s influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java.
- There are, of course, differences between Java and C#, but the overall “look and feel” of these languages is very similar.

How Java Changed the Internet

1. **Java Applets:**

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser.

If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.

Applets are intended to be small programs.

They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server.

In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs.

For example, when you read your e-mail, you are viewing passive data.

Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program.

2. Security:

As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code.

At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.

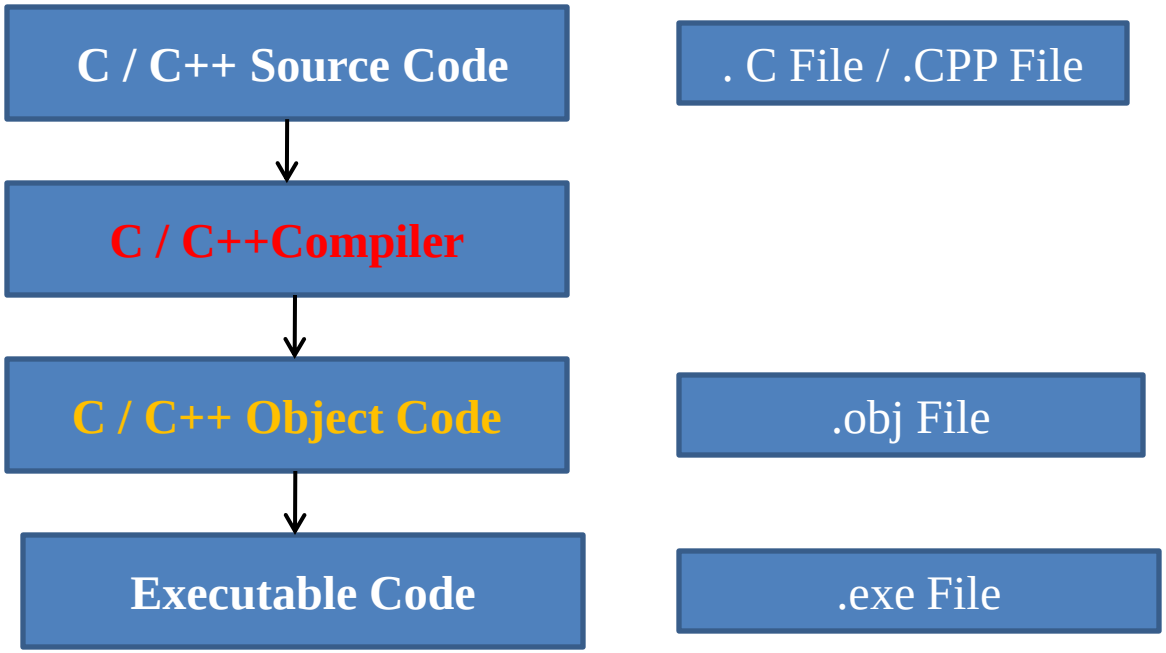
Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

3. Portability :

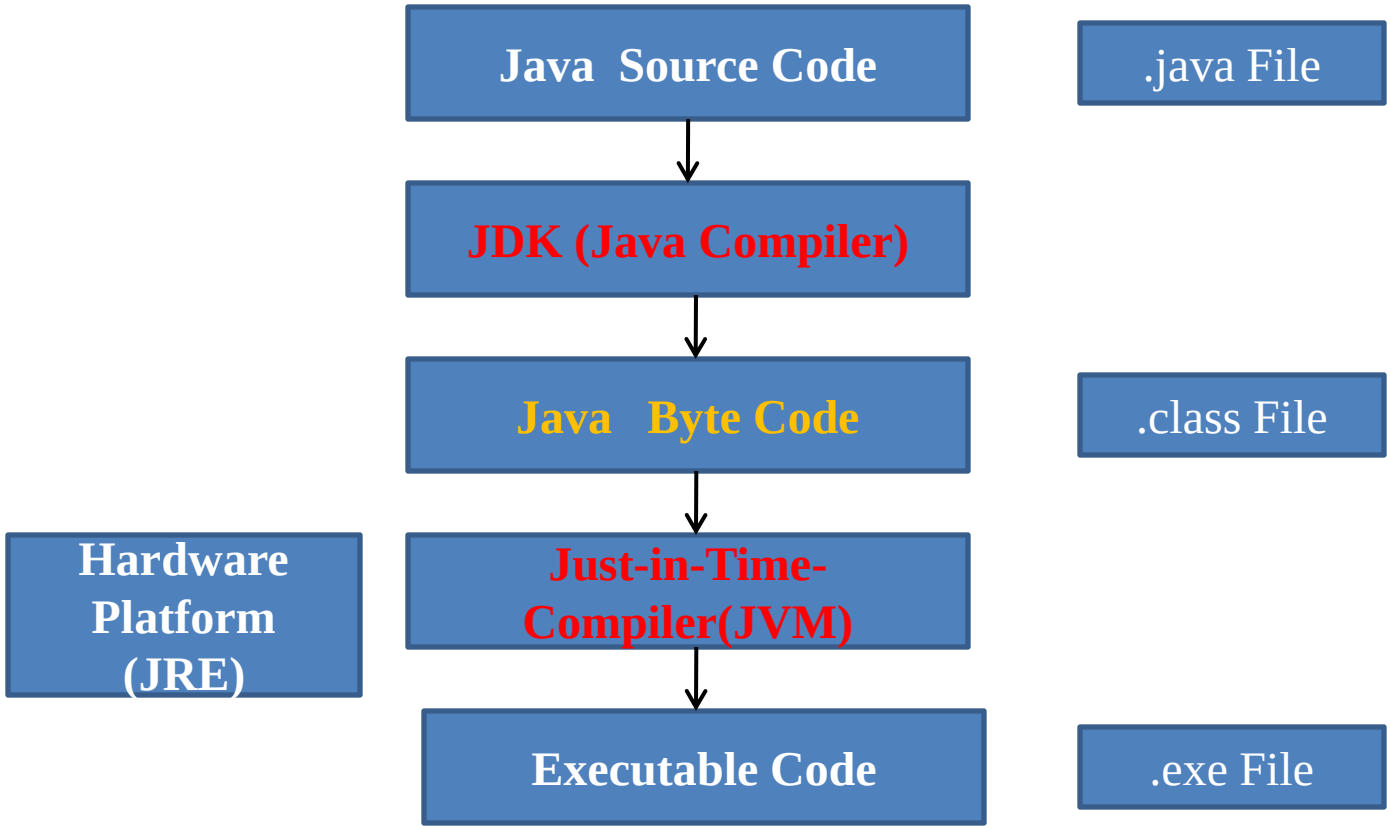
Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.

It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers.

C / C++ Program Execution ?



Java Program Execution ?



Java's Magic: The Bytecode

- *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).*
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- Although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.
- In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code.
- Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.
- It is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time.
- Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation.
- The remaining code is simply interpreted.

Servlets: Java on the Server Side

- A servlet is a small program that executes on the server.
- Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.
- Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.
- Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable.
- Thus, the same servlet can be used in a variety of different server environments.
- The only requirements are that the server support the JVM and a servlet container.

The Java Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

1. Simple:

Java was designed to be easy for the professional programmer to learn and use effectively.

Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

2. Object-Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language.

The object model in Java is simple and easy to extend

3. Robust:

Java is a strictly typed language, it checks your code at compile time.

However, it also checks your code at run time.

Two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors).

Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.)

Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs.

Java helps in this area by providing object-oriented exception handling.

4. Multithreaded :

Java was designed to meet the real-world requirement of creating interactive, networked programs.

To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

5. Architecture-Neutral :

No guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.

Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.

Goal was “write once; run anywhere, any time, forever.”

6. Interpreted and High Performance:

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.

This code can be executed on any system that implements the Java Virtual Machine.

7. Distributed :

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.

Java also supports *Remote Method Invocation (RMI).*

This feature enables a program to invoke methods across a network.

8. Dynamic:

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

- All computer programs consist of two elements: code and data.
- Furthermore, a program can be conceptually organized around its code or around its data.
- That is, some programs are written around “what is happening” and others are written around “who is being affected.”
- These are the two paradigms that govern how a program is constructed.
- The first way is called the *process-oriented model.*
- This approach characterizes a program as a series of linear steps (that is, code).
- The process-oriented model can be thought of as *code acting on data. Ex:* C employ this model.
- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized as *data controlling access to code*

OOP Principles

1. **Abstraction:**

2. Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.

In Java, the basis of encapsulation is the class.

A *class* defines the structure and behavior (data and code) that will be shared by a set of objects.

Each object of a given class contains the structure and behavior defined by the class.

For this reason, objects are sometimes referred to as *instances of a class.*

Thus, a class is a logical construct; an object has physical reality.

The data defined by the class are referred to as *member variables* or *instance variables.*

The code that operates on that data is referred to as *member methods* or just *methods.*

***3.* Inheritance:**

Inheritance is the process by which one object acquires the properties of another object.

This is important because it supports the concept of hierarchical classification.

This means that it is possible to design a generic interface to a group of related activities.

4. Polymorphism:

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions.

The concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”

A Simple Program : Example.java

```
class Example
    {
    public static void main(String args[])
            {
           System.out.println("This is a simple Java program.");
           }
    }
```

In Java, a source file is officially called a *compilation unit.* It is a text file that contains one or more class definitions.

The Java compiler requires that a source file use the **.java** filename extension.

In Java, all code must reside inside a class.

By convention, the name of that class should match the name of the file that holds the program.

Compiling the Program:

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line:

C:\>javac Example.java

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program.

The Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute.

Run :

To actually run the program, you must use the Java application launcher, called **java**.

To do so, pass the class name **Example** as a command-line argument, as shown here:

C:\>java Example

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension.

This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file.

When you execute **java** , it will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

Comment line in Java

Java supports three styles of comments.:

1. **Single-line comment :** A *single-line comment* begins with a // and ends at the end of the line.

Ex: // Addition of two numbers

2. **Multiline comment:** This type of comment must begin with **/*** and end with ***/**.

Anything between these two comment symbols is ignored by the compiler.

Ex: **/*** Addition of two numbers ***/**

3. **Documentation comment :**

class Example {

- This line uses the keyword **class** to declare that a new class is being defined.
- **Example** is an *identifier* that is the name of the class.

public static void main(String args[])

- This line begins the **main( )** method.
- All Java applications begin execution by calling **main( )**.
- The **public** keyword is an *access specifier*, which allows the programmer to control the visibility of class members.
- When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.
- In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.
- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class.
- This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made.
- The keyword **void** simply tells the compiler that **main( )** does not return a value.

- In **main( )**, there is only one parameter.
- **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**.
- Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.
- applets—Java programs that are embedded in web browsers—you won't use **main( )** at all, since the web browser uses a different means of starting the execution of applets.

System.out.println("This is a simple Java program.");

- The built-in **println( )** method. In this case, **println( )** displays the string which is passed to it.
- **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.
- The **println( )** statement ends with a semicolon.
- All statements in Java end with a semicolon.

System.out.println("This is num: " + num);

- In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output.
- (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedes it.
- Using the **+** operator, you can join together as many items as you want within a single **println( )** statement.
- The **print( )** method is just like **println( )**, except that it does not output a newline character after each call.

Command-Line Arguments

- Sometimes you will want to pass information into a program when you run it.
- This is accomplished by passing command-line arguments to **main( )**.
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**.
- The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.
- All command-line arguments are passed as strings.
- You must convert numeric values to their internal forms manually

Java Input

- The **Scanner class** is used to get user input, and it is found in the **java.util package**.

Method	Description
nextInt()	**Reads a int value from the user**
nextFloat()	**Reads a float value from the user**
nextByte()	**Reads a byte value from the user**
nextDouble()	**Reads a double value from the user**
nextBoolean()	**Reads a boolean value from the user**
nextLine()	**Reads a String value from the user**
nextShort()	**Reads a short value from the user**
nextLong()	**Reads a long value from the user**

Input Using Scanner

```
 import java.util.Scanner;
class B
 {
   public static void main(String args[]) throws java.io.IOException
      {
         Scanner a = new Scanner(System.in);
         System.out.println("\n Enter Your name:");
         String name = a.nextLine();
         System.out.println("\nYour name is :" + name);
      }
}
```

Input Using BufferedReader

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
 class C
  {
    public static void main(String args[]) throws java.io.IOException
       {
   BufferedReader a = new BufferedReader(new  InputStreamReader(System.in));
     System.out.println("\n Enter Your Name:");
     String name = a.readLine();
     System.out.println("\n Enter a number:");
     int num1 = Integer.parseInt(a.readLine());
     System.out.println("\nYour name is :" + name);
     System.out.println("\nNumber You entered is :" + num1);
      }  }
```

Java Keywords

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

- Java is a strongly typed language.
- **First**, every variable has a type, every expression has a type, and every type is strictly defined.
- **Second**, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.
- The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not.

Primitive Data Types

Group	Type / Size	Represents	
Intergers	byte (1 byte) short (2 bytes) int (4 bytes) long (8 bytes)	whole-valued signed numbers	
Floating-point number	float (4 bytes) double (8 bytes)	numbers with fractional precision	
Characters	char (2 bytes)	symbols in a character set, like letters and numbers.	
Boolean	boolean	true/false values.	

Integers

- Java defines four integer types: **byte**, **short**, **int**, and **long**.
- All of these are signed, positive and negative values.
- Java does not support unsigned, positive-only integers.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

1. byte :

- The smallest integer type is **byte**.
- This is a signed 8-bit type that has a range from –128 to 127.
- Variables of type **byte** are especially useful when you're working with a stream of data from a network or file.
- They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the **byte** keyword.
- Ex: byte b, c;

2. short : short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type.

Ex: short a, b;

3. int :

- The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case.
- The reason is that when **byte** and **short** values are used in an expression they are *promoted* to **int** when the expression is evaluated.
- Therefore, **int** is often the best choice when an integer is needed.
- Ex: int a, b;

4. long :

- **long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value.
- The range of a **long** is quite large.
- This makes it useful when big, whole numbers are needed.
- Ex: long a, b;

Floating Point

- Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.
- There are **two kinds** of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.
-

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

5. float :

- The type **float** specifies a *single-precision* value that uses 32 bits of storage.
- Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small.
- Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.
- Ex: float a, b;

6. double:

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.
- All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. Ex: double a, b;

7. char:

- In Java, the data type used to store characters is **char**.
- Java uses Unicode to represent characters.
- *Unicode* defines a fully international character set that can represent all of the characters found in all human languages.
- It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- For this purpose, it requires 16 bits.
- Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.
- the ASCII character set occupies the first 127 values in the Unicode character set.

- Ex: char a, b;

8. Boolean:

- Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**.

Literals

- **1. interger literals:**
- Any whole number value is an integer literal.
- all decimal values, meaning they are describing a base 10 number.
- Octal values are denoted in Java by a leading zero.
- A hexadecimal constant with a leading zero-x, (**0x** or **0X**).
-

2. Floating-Point Literals:

- Floating-point numbers represent decimal values with a fractional component.
- They can be expressed in either **standard or scientific notation**.
- ***Standard notation*** consists of a whole number component followed by a decimal point followed by a fractional component.
- ***Scientific notation*** uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.
- The exponent is indicated by an ***E*** **or** ***e*** followed by a decimal number, which can be positive or negative.
- Floating-point literals in Java default to **double** precision.
- To specify a **float** literal, you must append an ***F*** **or** ***f*** to the constant. You can also explicitly specify a **double** literal by appending a ***D*** **or** ***d***.
- The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits.

3. Boolean Literals:

- There are only two logical values that a **boolean** value can have, **true** and **false**.
- The values of **true** and **false** do not convert into any numerical representation.
- The **true** literal in Java does not equal 1, nor does the **false** literal equal 0.
- In Java, they can only be assigned to variables declared as **boolean**, or used in expressions with Boolean operators.

4. Character literals:

- Characters in Java are indices into the Unicode character set.
- They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators.
- A literal character is represented inside a pair of single quotes.
- For octal notation, use the backslash followed by the three-digit number. For example, *'\141'* is the letter *'a'*.
- For hexadecimal, you enter a backslash-u (**\u**), then exactly four hexadecimal digits. For example, *'\u0061'*

5. String literals:

- String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.
- Java strings is that they must begin and end on the same line.
- There is no line-continuation escape sequence as there is in some other languages.
- C/C++, strings are implemented as arrays of characters.
- However, this is not the case in Java. Strings are actually object types.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

VARIABLES

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.
- In Java, all variables must be declared before they can be used.
-

The Scope and Lifetime of Variables

- A block defines a *scope*.
- Thus, each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program.
- It also determines the lifetime of those objects. Many other computer languages define two general categories of scopes: global and local.
- However, these traditional scopes do not fit well with Java's strict, object-oriented model.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- The scope defined by a method begins with its opening curly brace.
- However, if that method has parameters, they too are included within the method's scope.

- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Scopes can be nested.
- The outer scope encloses the inner scope.
- This means that objects declared in the outer scope will be visible to code within the inner scope.
- However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.
- Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
- Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left.

- To assign a value of one type to a variable of another type, if the two types are compatible, then Java will perform the conversion automatically.
- You must use a *cast*, which performs an explicit conversion between incompatible types.
- **Automatic Conversions:**
- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a *widening conversion* takes place.

Casting Incompatible type:

- A *narrowing conversion*, since we are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast.
- A *cast* is simply an explicit type conversion.
- Syntax: (target-type) value ;
- *target-type* specifies the desired type to convert the specified value to.
- If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.
- If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

Automatic Type Promotion in Expressions

The Type Promotion Rules :

- First, all **byte**, **short**, and **char** values are promoted to **int**.
- Then, if one operand is a **long**, the whole expression is promoted to **long**.
- If one operand is a **float,** the entire expression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

ARRAY

- *Syntax: array-var* = new *type*[*size*];
- int month_days[] = new int[12];
- Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array.
- That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate.
- The elements in the array allocated by **new** will automatically be initialized to zero.
- In Java all arrays are dynamically allocated.
- All array indexes start at zero.

- Arrays can be initialized when they are declared.

int m[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

- The process is much the same as that used to initialize the simple types.
- An *array initializer* is a list of comma-separated expressions surrounded by curly braces.
- The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use **new**.
- The Java run-time system will check to be sure that all array indexes are in the correct range.
- If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Multidimensional Arrays

- In Java, *multidimensional arrays* are actually arrays of arrays.
- int 2D[ ][ ] = new int[4][5];
- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.
- It is possible to initialize multidimensional arrays.
- To do so, simply enclose each dimension's initializer within its own set of curly braces.
- double m[ ][ ] = {
  { 0*0, 1*0, 2*0, 3*0 },
  { 0*1, 1*1, 2*1, 3*1 },
  { 0*2, 1*2, 2*2, 3*2 },
  { 0*3, 1*3, 2*3, 3*3 }
  };

Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:
  type[ ] *var-name;*
- Here, the square brackets follow the type specifier, and not the name of the array variable.
- int al[ ] = new int[3];
  int[ ] a2 = new int[3];
- char 2D[ ][ ] = new char[3][4];
  char[ ][ ] 2D = new char[3][4];
- This alternative declaration form offers convenience when declaring several arrays at the same time.
- **int[ ] nums, nums2, nums3; // create three arrays**
  creates three array variables of type **int**. It is the same as writing
  int nums[ ], nums2[ ], nums3[ ]; // create three arrays
- The alternative declaration form is also useful when specifying an array as a return type for a method.

- **String:** String defines an object.
- The **String** type is used to declare string variables. You can also declare arrays of strings.
- A quoted string constant can be assigned to a **String** variable.
- A variable of type **String** can be assigned to another variable of type **String**.
- **Pointer :** Java does not support or allow pointers.
- Java does not support pointers that can be accessed and/or modified by the programmer.)
- Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer.

Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- The operands of the arithmetic operators must be of a numeric type.
- You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.
- The modulus operator, **%**, returns the remainder of a division operation.
- It can be applied to floating-point types as well as integer types.
- Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**.
- These operators act upon the individual bits of their operands.
- All of the integer types (except **char**) are signed integers.
- This means that they can represent negative values as well as positive ones.
- Java uses an encoding known as *two's complement,* which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.
- To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

left shift : The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.

- It has this general form: **value << num**
- Here, *num* specifies the number of positions to left-shift the value in *value*.
- That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by *num*.
- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2.
- If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative.

- **right shift :**The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here: **value >> num**
- Here, *num* specifies the number of positions to right-shift the value in *value*.
- That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by *num*.
- Each time you shift a value to the right, it divides that value by two—and discards any remainder. You can take advantage of this for high-performance integer division by 2.
- You must be sure that you are not shifting any bits off the right end.
- if you shift –1 right, the result always remains –1, since sign extension keeps bringing in more ones in the high-order bits.

Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Logical Operators

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Boolean Logical Operators

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

CONTROL STATEMENTS

1. Selection Statements:

- if
- if – else
- nested if – else
- if – else if ladder

2. switch statement

- nested switch statement

3. iteration statements:

- while
- do – while
- for
- for – each

4. Jump Statements: break, continue & return;

for - each

- The for-each style of **for** is also referred to as the *enhanced* **for** loop.

  for (*type itr-var : collection*) *statement-block*

- Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by *collection*.
- There are various types of collections that can be used with the **for**.
- With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*.
- The loop repeats until all elements in the collection have been obtained.
- Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.
- Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

- Its iteration variable is “read-only” as it relates to the underlying array.
- An assignment to the iteration variable has no effect on the underlying array.
- In other words, you can’t change the contents of the array by assigning the iteration variable a new value.

CLASS

- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- As such, the class forms the basis for object-oriented programming in Java.
- Any concept you wish to implement in a Java program must be encapsulated within a class.
- A class defines a new data type.
- Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template* for an object, and an object is an *instance* of a class.

- The data, or variables, defined within a **class** are called *instance variables.*
- The code is contained within *methods.*
- Collectively, the methods and variables defined within a class are called *members* of the class.
- In most classes, the instance variables are acted upon and accessed by the methods defined for that class.
- Thus, as a general rule, it is the methods that determine how a class' data can be used.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

- The general form of a class does not specify a **main( )** method.
- Java classes do not need to have a **main( )** method.
- You only specify one if that class is the starting point for your program.
- Further, applets don’t require a **main( )** method at all.
-

Declaring Objects

Syntax: **class-name object-name;**

object-name = new class-name( );

or **class-name object-name = new class-name( );**

- Obtaining objects of a class is a **two-step** process.
- **First, you must declare a variable of the class type.**
- This variable does not define an object.
- Instead, it is simply a variable that can *refer* to an object.
- **Second, you must acquire an actual**, **physical copy of the object and assign it to that variable.**
- You can do this using the **new** operator.
- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable.
- Thus, in Java, all class objects must be dynamically allocated.

class-name object-name; // declare reference to object

object-name = new class-name( ); // allocate memory object

- An object reference is similar to a memory pointer.
- The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers.
- Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

object-name = new class-name( );

- Here, *object-name* is a variable of the class type being created.
- The *class-name* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- However, if no explicit constructor is specified, then Java will automatically supply a default constructor.

- **new** allocates memory for an object during run time.
- However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur.
-

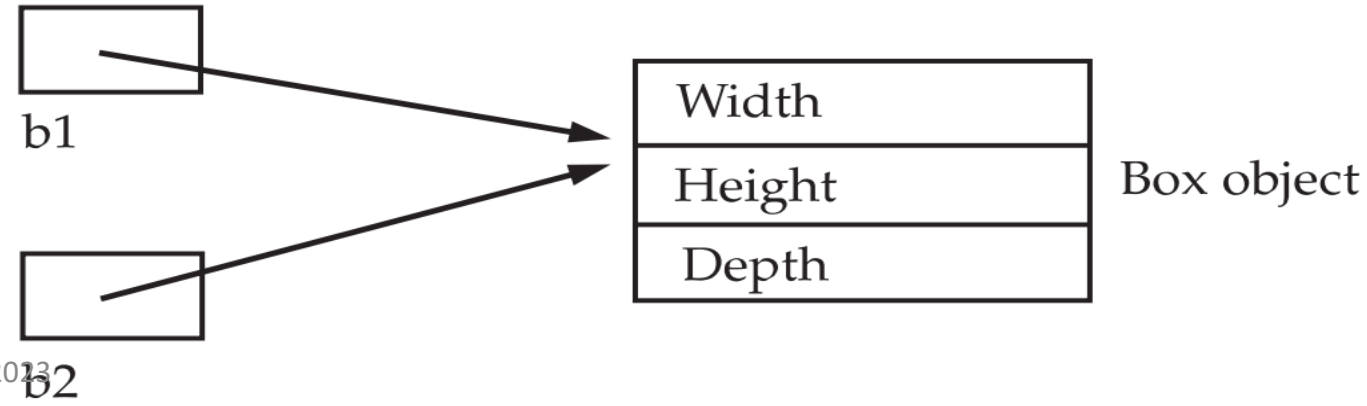
Distinction between a Class and an Object

- A class creates a new data type that can be used to create objects.
- That is, a class creates a logical framework that defines the relationship between its members.
- When you declare an object of a class, you are creating an instance of that class.
- Thus, a class is a logical construct.
- An object has physical reality. (That is, an object occupies space in memory.)

Assigning Object Reference Variables

Box b1 = new Box();
Box b2 = b1;

- **b1** and **b2** will both refer to the same object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object.
- It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
- When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.



CONSTRUCTOR

- Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they have no return type, not even **void**.
- This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Default Constructor:

Box mybox1 = new Box();

- **new Box( )** is calling the **Box( )** constructor.
- The default constructor automatically initializes all instance variables to zero.

Parameterized Constructor:

Box mybox1 = new Box(10, 20, 15);

- The values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.
- Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

Constructor Overloading:

this keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword.
- **this** can be used inside any method to refer to the *current* object.
- That is, **this** is always a reference to the object on which the method was invoked.

```
Box(double w, double h, double d)
 {
  this.width = w;
  this.height = h;
  this.depth = d;
  }
```

Instance Variable Hiding:

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- It is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding.

```
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;  }
```

METHOD OVERLOADING

- When two or more methods within the same class that share the same name and parameter declarations are different, the methods are said to be *overloaded*, and the process is referred to as *method overloading*.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.
- Java will employ its automatic type conversions only if no exact match is found.
- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.

Using Objects as Parameters

- One of the most common uses of object parameters involves constructors.
- Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as a parameter.

PASSING ARGUMENTS MECHANISM

- There are **two** ways that a computer language can pass an argument to a subroutine.
- ***call-by-value:*** This approach copies the *value* of an argument into the formal parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- ***call-by-reference:*** In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine.

PASSING OBJECT AS REFERENCE

- When a primitive type is passed to a method, it is done by use of call-by-value.
- Objects are implicitly passed by use of call-by-reference.
- When you create a variable of a class type, you are only creating a reference to an object.
- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference.
- Changes to the object inside the method do affect the object used as an argument.

STATIC

- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**.
- **main( )** is declared as **static** because it must be called before any objects exist.
- Instance variables declared as **static** are, essentially, global variables.
- When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- **They can only call other static methods.**
- **They must only access static data.**
- **They cannot refer to ‘this’ or ‘super’ in any way.**
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.
- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator.
- if you wish to call a **static** method from outside its class : ***classname.method( )***
- Here, *classname* is the name of the class in which the **static** method is declared.
- A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class.

final

- A variable can be declared as **final**.
- Doing so prevents its contents from being modified.
- This means that you must initialize a **final** variable when it is declared.
- **final int FILE_NEW = 1;**
  final int FILE_OPEN = 2;

- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis.
- Thus, a **final** variable is essentially a constant.
- The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

NESTED CLASS

- It is possible to define a class within another class; such classes are known as *nested classes*.
- The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested.
- However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- It is also possible to declare a nested class that is local to a block.

- There are **two types** of nested classes: ***static* and *non-static***.
- A static nested class is one that has the **static** modifier applied.
- Because it is static, it must access the members of its enclosing class through an object.
- That is, it cannot refer to members of its enclosing class directly.
- Because of this restriction, static nested classes are seldom used. The most important type of nested class is the *inner* class.
- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

- An inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.
- Inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope.
- You can define a nested class within the block defined by a method or even within the body of a **for** loop.
- **USE :** While nested classes are not applicable to all situations, they are particularly helpful when handling events.

Command-Line Arguments

- Sometimes you will want to pass information into a program when you run it.
- This is accomplished by passing command-line arguments to **main( )**.
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**.
- The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.
- All command-line arguments are passed as strings.
- You must convert numeric values to their internal forms manually

INHERITANCE

- A class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass.
- Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- Java does not support the inheritance of multiple super classes into a single subclass.
- No class can be a superclass of itself .
- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- A class member that has been declared as private will remain private to its class.
- It is not accessible by any code outside its class, including subclasses.

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- Once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes.
- Each subclass simply adds its own unique attributes.

A Superclass Variable Can Reference a Subclass Object

- It is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
- There will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms.
- The **first** calls the superclass' constructor.
- The **second** is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors(1st Use)

- A subclass can call a constructor defined by its superclass by use of the following form of **super**:
  super(arg-list);
  Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.
- When a subclass calls **super( )**, it is calling the constructor of its immediate superclass.
- Thus, **super( )** always refers to the superclass immediately above the calling class.
- This is true even in a multileveled hierarchy.
- Also, **super( )** must always be the first statement executed inside a subclass constructor.

A Second Use for super

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.

  super.*member*

- Here, *member* can be either a method or an instance variable.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

When Constructors Are Called?

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.
- Constructors are executed in order of derivation because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass.
- Therefore, it must be executed first.

METHOD OVERRIDING

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.
- Method overriding occurs *only* when the names and the type signatures of the two methods are identical.
- If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- A superclass reference variable can refer to a subclass object.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.
- When different types of objects are referred to, different versions of an overridden method will be called.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.
- These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.

Syntax: abstract *type name(parameter-list);*

- Any class that contains one or more abstract methods must also be declared abstract.

- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
- Such objects would be useless, because an abstract class is not fully defined.
- Also, you cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

Using final with Inheritance

- The keyword **final** has three uses.
- **First**, it can be used to create the equivalent of a named constant.
- **Second**, Using final to Prevent Overriding : To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- Methods declared as **final** cannot be overridden.
- Since **final** methods cannot be overridden, a call to one can be resolved at compile time.
- This is called *early binding.*
- **Third**, Using final to Prevent Inheritance Sometimes you will want to prevent a class from being inherited
- .To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations

PACKAGES

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- The package is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are only exposed to other members of the same package.
- This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

- Include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.
- **package *pkg*;**
  Here, *pkg* is the name of the package.
-

- Java uses file system directories to store packages.
- For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.
- The **package** statement simply specifies to which package the classes defined in a file belong.
- It does not exclude other classes in other files from being part of that same package.
- You can create a hierarchy of packages.
- The general form of a multileveled package statement is :
  package *pkg1*[.*pkg2*[.*pkg3*]];
- A package declared as package **java.awt.image;**
  needs to be stored in **java\awt\image** in a Windows environment.
- You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

- Packages are mirrored by directories.
- Java run-time system know where to look for packages that you create?
- The answer has three parts:
- **First**, by default, the Java run-time system uses the current working directory as its starting point.
- Thus, if your package is in a subdirectory of the current directory, it will be found.
- **Second**, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- **Third**, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

- package **MyPack**
- In order for a program to find **MyPack,** one of three things must be true.
- Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.
- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to* **MyPack**.
- For example, in a Windows environment, if the path to **MyPack** is C:\MyPrograms\Java\MyPack
- Then the class path to **MyPack** is : **C:\MyPrograms\Java**
- Simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory.
- **Execution : java packagename.filename**

Execution of Package File

- **Compile the File : File is present in the path C:\Users\ASUS\Desktop\A**
- **javac FileName.java**
- Ex: D.java File present in the path **C:\Users\ASUS\Desktop\A\D.java** and Package name is “A”
- C:\Users\ASUS\Desktop\A> javac D.java
- **Run the File: javac -d directory javafilename**
- **javac -d . D.java Or javac -d E:/abc D.java (to save in location of “abc” folder of “E” drive)**
- Ex: C:\Users\ASUS\Desktop\A>javac -d . D.java
- **javac –d .. demo.java** **.** Represents current directory **..** Represents parent directory
- This forces the compiler to create the “D" package.
- The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.
- **Note:** The package name should be written in lower case to avoid conflict with class names.

- When a package name is not specified, a class is in the default package (the current working directory) and the package itself is given no name. Hence you were able to execute assignments earlier.
- While creating a package, care should be taken that the statement for creating package must be written before any other import statements
- Ex: // not allowed

  import package p1.*;

  package p3;
- //correct syntax

  package p3;

  import package p1.*;

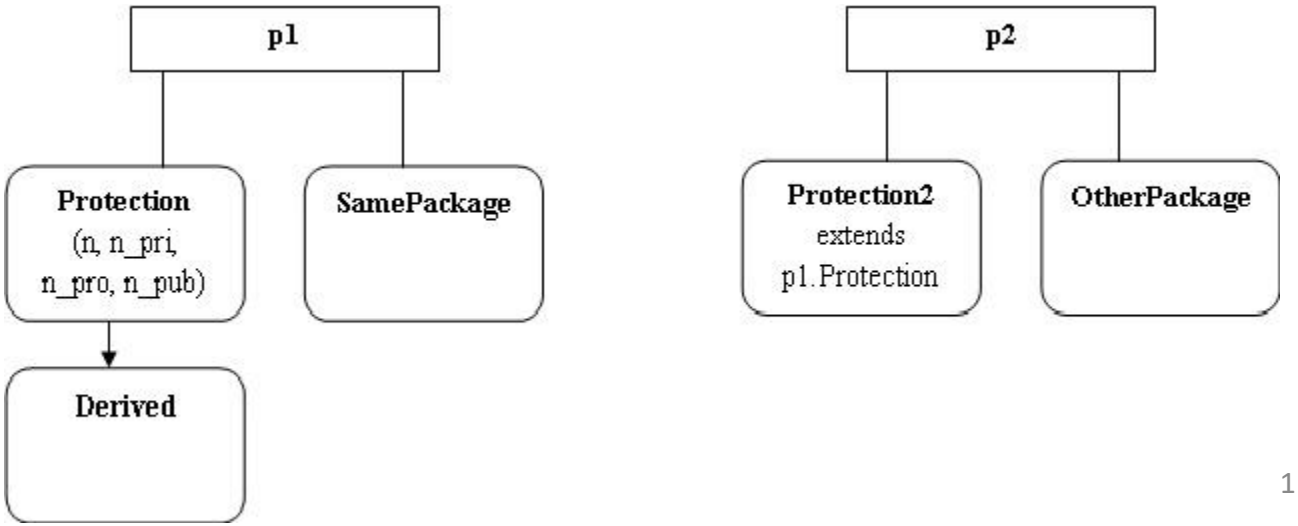
ACCESS PROTECTION

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- The class is Java's smallest unit of abstraction.
- Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

CLASS MEMBER ACCESS

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes



Importing Packages

- Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
- For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- import *pkg1*[.*pkg2*].(*classname*|*);
  Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

- Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.
- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- **import java.lang.*;**
- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy.
- import java.util.*;

class MyDate extends Date {
}
The same example without the import statement looks like this:
class MyDate extends java.util.Date {
}

Interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface**.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- However, each class is free to determine the details of its own implementation.
- Interfaces are designed to support dynamic method resolution at run time.

Defining an Interface

- access interface name

```
{
 return-type method-name1(parameter-list);
 return-type method-name2(parameter-list);
 type final-varname1 = value;
 type final-varname2 = value;
 // ...
 return-type method-nameN(parameter-list);
 type final-varnameN = value;
 }
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other code.
- In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- Notice that the methods that are declared have no bodies.
- They end with a semicolon after the parameter list.
- They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations.
-

- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- They must also be initialized.
- All methods and variables are implicitly **public.**

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- **class *classname* [extends *superclass*] [implements *interface* [,*interface*...]]**

  {
  // class-body
  }
- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.
- When you implement an interface method, it must be declared as **public**.

KEY POINTS

An interface is similar to a class in the following ways :

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

KEY POINTS

- We can’t instantiate an interface in java. That means we cannot create the object of an interface.
- Interface provides full abstraction as none of its methods have body.
- While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- Interface cannot be declared as private, protected.
- All the interface methods are by default **abstract and public.**
- Variables declared in interface are **public, static and final** by default.
- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

KEY POINTS

- Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final.
- An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.
- A **class** can implement any **number of interfaces**.
- If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.
- A class cannot implement two interfaces that have methods with same name but different return type.
- Variable names conflicts can be resolved by interface name.

Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, a exception is an run-time error.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.
- **Advantages**: **Firs**t, it allows you to fix the error.
- **Second**, it prevents the program from automatically terminating.

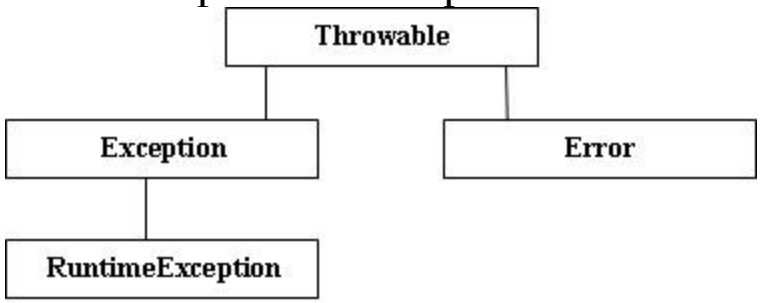
- General Form of an Exception handling block:

try
{
// block of code to monitor for errors
}
catch (*ExceptionType1 exOb*)
{
// exception handler for *ExceptionType1*
}
catch (*ExceptionType2 exOb*)
{
// exception handler for *ExceptionType2*
}
// ...
finally
{
// block of code to be executed after try block ends
}

Here, ***ExceptionType*** is the type of exception that has occurred.

Exception Types

- All exception types are subclasses of the built-in class **Throwable**.
- Thus, **Throwable** is at the top of the exception class hierarchy.



- **Exception** This class is used for exceptional conditions that user programs should catch.
- This is also the class that you will subclass to create your own custom exception types.
- There is an important subclass of **Exception**, called **RuntimeException**.
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions

- class A

```
{
  public static void main(String args[])
   {
      int d = 0;
      int a = 42 / d;
    }
  }
```

The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:
java.lang.ArithmeticException: / by zero at A.main(A.java:6)

Notice how the class name, **A**; the method name, **main**; the filename, **A.java**; and the line number, **6**, are all included in the simple stack trace.

Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

Exception inside a Method

- class A

```
{
static void subroutine()
  {
    int d = 0;
    int a = 10 / d;
   }
public static void main(String args[])
 {
A.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at A.subroutine(A.java:4)
at A.main(A.java:7)
```

As we can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4.

Design Mechanism

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch.
- A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements).
- The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

- class A
  {
  public static void main(String args[])
  {
  int d, a;
  try { // monitor a block of code.
  d = 0;
  a = 42 / d;
  System.out.println("This will not be printed.");
  } catch (ArithmeticException e) { // catch divide-by-zero error
  System.out.println("Division by zero.");
  }
  System.out.println("After catch statement.");
  }
  }

  This program generates the following output:

  Division by zero.

  After catch statement.

Displaying a Description of an Exception

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception.
- You can display this description in a **println( )** statement by simply passing the exception as an argument.
- Ex:(EXCEPTION-1, EXCEPTION-2)The program is run, each divide-byzero error displays the following message:
  Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.
- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

Nested try Statements

- A **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.
-

throw

- Syntax: **throw *ThrowableInstance*;**
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are **two ways** you can obtain a **Throwable** object: using a parameter in a **catch** clause, or creating one with the **new** operator.
- Ex: **throw new NullPointerException("demo");**
  Here, **new** is used to construct an instance of **NullPointerException**.
- Many of Java's builtin run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.
- When the **second form** is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to **print( )** or **println( )**.
- It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

throws

- A **throws** clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the **throws** clause.
- If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a **throws** clause:

  type method-name*(*parameter-list*) throws *exception-list
  {
  // body of method
  }

  Here, ***exception-list*** is a comma-separated list of the exceptions that a method can throw.

Different between throw & throws

- **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.
- **throws** keyword is in the function signature. It is used when the function has some statements that can lead to some exceptions.
- **throw** keyword is used to throw an exception explicitly. It can throw only one exception at a time.
- **throws** keyword can be used to declare multiple exceptions, separated by comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then.
- Syntax of **throw** keyword includes the instance of the Exception to be thrown.
- Syntax of **throws** keyword includes the class names of the Exceptions to be thrown.

finally

- **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The **finally** clause is optional.
- However, each **try** statement requires at least one **catch** or a **finally** clause.

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- In the language of Java, these are called ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions.
- The checked exceptions defined in **java.lang** are those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.
- These are called ***checked exceptions***.
- Java defines several other types of exceptions that relate to its various class libraries.

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Checked Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating Your Own Exception Subclasses

- Create your own exception types to handle situations specific to your applications.
- define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).
- Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own.
- It does, of course, inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

The Methods Defined by Throwable

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

- **Exception( ) :** creates an exception that has no description.
- **Exception(String *msg*) :** specify a description of the exception.

MULTITHREAD

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking: process based and thread-based.
- *Process-based* multitasking is the feature that allows your computer to run two or more programs concurrently.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
- In processbased multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously.
- For instance, a text editor can format text at the same time that it is printing .
- Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces.
- Inter-process communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight.
- They share the same address space and cooperatively share the same heavyweight process.
- Inter-thread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java.
- Multitasking threads require less overhead than multitasking processes.

- Processes are heavyweight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight.
- They share the same address space and cooperatively share the same heavyweight process.
- Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
-

The Java Thread Model

- Single-threaded systems use an approach called an ***event loop*** with ***polling***.
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- In general, in a singled-threaded environment, when a thread ***blocks*** (that is, suspends execution) because it is waiting for some resource, the entire program stops running.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Thread State

- Threads exist in several states.
- A thread can be ***running***. It can be *ready to run* as soon as it gets CPU time.
- A running thread can be ***suspended***, which temporarily suspends its activity.
- A suspended thread can then be ***resumed***, allowing it to pick up where it left off.
- A thread can be ***blocked*** when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately.
- Once terminated, a thread cannot be resumed.

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- A thread's priority is used to decide when to switch from one running thread to the next.
- This is called a ***context switch***. The rules that determine when a context switch takes place are simple:
- *A thread can voluntarily relinquish control*. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread*. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called ***preemptive multitasking.***

Synchronization

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
- The **Thread** class defines several methods that help manage threads.
-

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

- When a Java program starts up, one thread begins running immediately.
- This is usually called the ***main thread*** of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:
  • It is the thread from which other "child" threads will be spawned.
  • Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.
- To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

- Its general form is :
  static Thread currentThread( )
- This method returns a reference to the thread in which it is called.
- Once you have a reference to the main thread, you can control it just like any other thread.

sleep( ) Method

- The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds.
- i) **static void sleep(long *milliseconds*)** throws InterruptedException
  The number of milliseconds to suspend is specified in *milliseconds*.
  This method may throw an **InterruptedException**.
- **ii) static void sleep(long *milliseconds*, int *nanoseconds*)** throws InterruptedException.
- This second form is useful only in environments that allow timing periods as short as nanoseconds.
- Methods that are members of the **Thread** class and are declared like this:
  final void setName(String *threadName*)
  final String getName( )
  Here, ***threadName*** specifies the name of the thread.

Creating a Thread

- We create a thread by instantiating an object of type **Thread**.
- Java defines two ways in which this can be accomplished:
  - 1. You can implement the **Runnable** interface.
  - 2. You can extend the **Thread** class, itself.
-

1. Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**.
- To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:
  public void run( )
- Inside **run( )**, you will define the code that constitutes the new thread.
- **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program.
- This thread will end when **run( )** returns.

- After you create a class that implements **Runnable**, we will instantiate an object of type **Thread** from within that class.
- **Thread** defines several constructors.
- Thread(Runnable *threadOb*, String *threadName*)
- In this constructor, ***threadOb*** is an instance of a class that implements the **Runnable** interface.
- This defines where execution of the thread will begin. The name of the new thread is specified by ***threadName***.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**.
- In essence, **start( )** executes a call to **run( )**.

Key Points

- The main thread must be the last thread to finish running.
- If the main thread finishes before a child thread has completed, then the Java run-time system may “hang.”

2. Extending Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

Key Points – better way ?

- The **Thread** class defines several methods that can be overridden by a derived class.
- Of these methods, the only one that *must* be overridden is **run( )**.
- This is, of course, the same method required when you implement **Runnable**.
- If you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.

main Thread to Finish Last?

- **Solution 1:** This is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread.
- Disadvantage: How can one thread know when another thread has ended?
- **Two ways exist to determine whether a thread has finished.**
- **i) First**, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:
  final boolean isAlive( )
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running.
- It returns **false** otherwise.
- **ii)** While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**:
  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.

Applet

- ***Applets*** are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.
- The applet does not have a **main( )** method.
- Unlike Java programs, applets do not begin execution at **main( )**.
- An applet begins execution when the name of its class is passed to an applet viewer or to a network browser.
- The applet class is contained in the java.applet package.
- Applet contains several methods that give you detailed control over the execution of your applet.
- In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Types of Applets

- **1. Based on Applet class:**
- These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all).
- This style of applet has been available since Java was first created.
- It is used when only a very simple user interface is required.

- **2. based on the Swing class Japplet:**
- Swing applets use the Swing classes to provide the GUI.
- Swing offers a richer and often easier-to-use user interface than does the AWT.
- Thus, Swing-based applets are now the most popular.
- Because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **Japplet.**
- Both AWT- and Swing-based applets are valid.

Basics of Applets

- All applets are subclasses (either directly or indirectly) of **Applet**.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.
- **appletviewer**, provided by the JDK.
- But you can use any applet viewer or browser you like.
- Execution of an applet does not begin at **main( )**.
- Actually, few applets even have **main( )** methods.
- Instead, execution of an applet is started and controlled with an entirely different mechanism.
- Output to your applet's window is not performed by **System.out.println( )**.
- Rather, in non-Swing applets, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in a console application.

- To use an applet, it is specified in an HTML file.
- One way to do this is by using the APPLET tag. (The OBJECT tag can also be used, but Sun currently recommends the APPLET tag.
- The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file.
- To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag.
- /*

  <applet code="MyApplet" width=200 height=60>
  </applet>
  */

Applet Compilation and Run

- **Source Code : Applet-1.html**
- **Compilation:** compile in the same way that you have been compiling programs.
- Run: there are two ways in which you can run an applet:
- 1. Executing the applet within a Java-compatible web browser.
- 2. Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.
- C:\>appletviewer RunApp.html

Applet Class

- **Applet** provides all necessary support for applet execution, such as starting and stopping.
- It also provides methods that load and display images, and methods that load and play audio clips.
- **Applet** extends the AWT class **Panel**.
- In turn, **Panel** extends **Container**, which extends **Component**.
- These classes provide support for Java's window-based, graphical interface.
- Thus, **Applet** provides all of the necessary support for window-based activities.

Methods Defined by Applets

Method	Description
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*.

Method	Description
void showStatus(String *str*)	Displays *str* in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after **init( )** when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls **start( )**.

Methods Defined by Applets

Method	Description
AudioClip getAudioClip(URL *url*, String *clipName*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url* and having the name specified by *clipName*.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url*.
Image getImage(URL *url*, String *imageName*)	Returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.
Locale getLocale( )	Returns a **Locale** object that is used by various locale-sensitive classes and methods.
String getParameter(String *paramName*)	Returns the parameter associated with *paramName*. **null** is returned if the specified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose.
void init( )	Called when an applet begins execution. It is the first method called for any applet.

Methods Defined by Applets

boolean isActive( )	Returns **true** if the applet has been started. It returns **false** if the applet has been stopped.
static final AudioClip newAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object.
void play(URL *url*)	If an audio clip is found at the location specified by *url*, the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**.
void resize(int *width*, int *height*)	Resizes the applet according to the dimensions specified by *width* and *height*.
final void setStub(AppletStub *stubObj*)	Makes *stubObj* the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser.

Applet Architecture

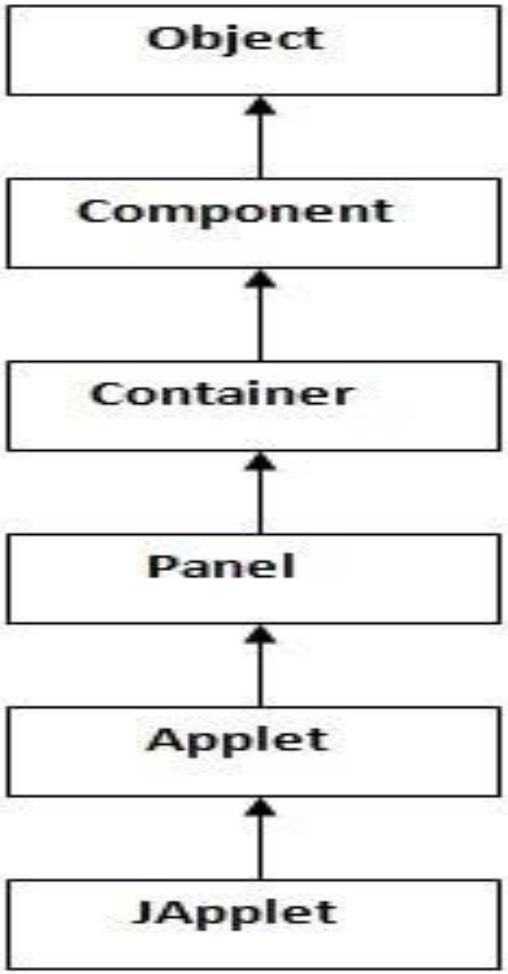
- An applet is a window-based program.
- As such, its architecture is different from the console-based programs.
- First, applets are event driven.
- An applet resembles a set of interrupt service routines.
- An applet waits until an event occurs.
- The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet.
- Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point.
- For the most part, your applet should not enter a “mode” of operation in which it maintains control for an extended period.
- Instead, it must perform specific actions in response to events and then return control to the run-time system.
-

- In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window).
- Second, the user initiates interaction with an applet.
- The user interacts with the applet as he or she wants, when he or she wants.
- These interactions are sent to the applet as events to which the applet must respond.
- For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated.
- If the user presses a key while the applet's window has input focus, a keypress event is generated.
- Applets can contain various controls, such as push buttons and check boxes.
- When the user interacts with one of these controls, an event is generated.

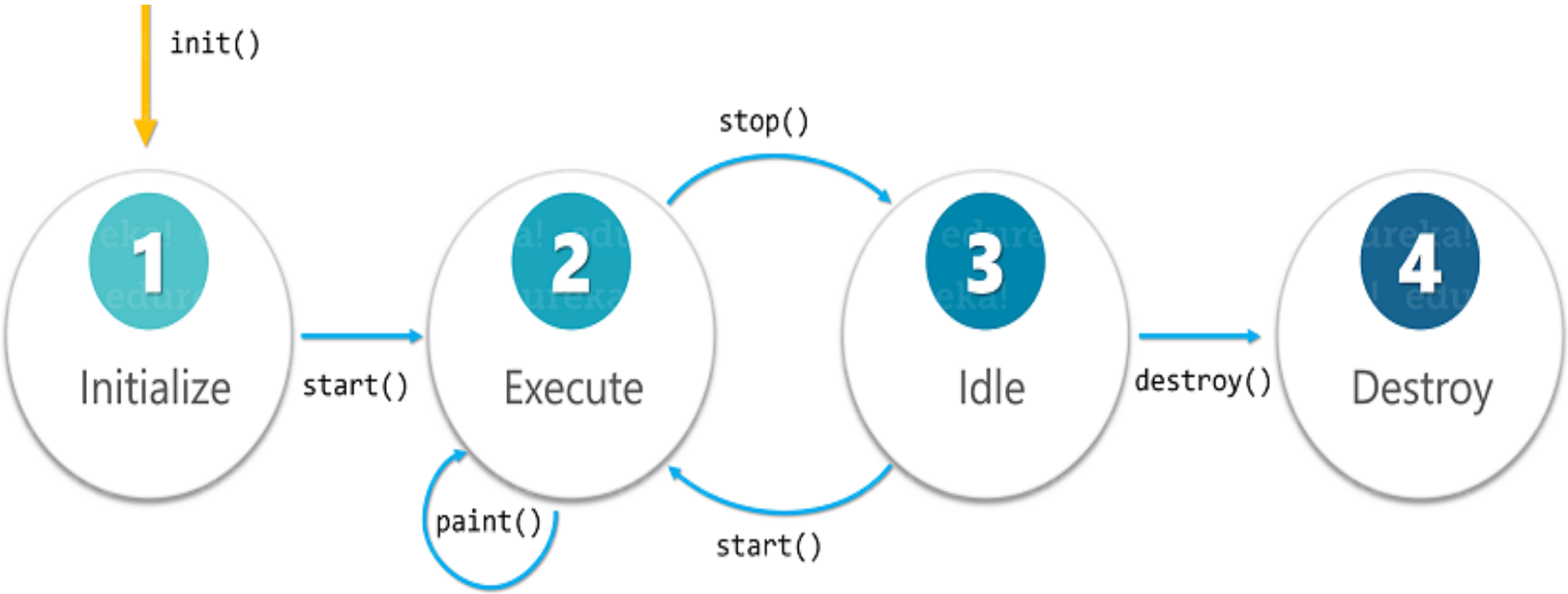
- Override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**.
- Default implementations for all of these methods are provided.
- Applets do not need to override those methods they do not use.
- However, only very simple applets will not need to define all of them.
- AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class.
- This method is called when the applet's output must be redisplayed.

- Although this skeleton does not do anything, it can be compiled and run.
- When run, it generates the following window when viewed with an applet viewer:

Hierarchy of Applet



APPLET LIFE CYCLE



Applet Initialization and Termination

- When an applet begins, the following methods are called, in this sequence:
  1. **init( )**
  2. **start( )**
  3. **paint( )**
- When an applet is terminated, the following sequence of method calls takes place:
  1. **stop( )**
  2. **destroy( )**

- **1. init( ) :** The **init( )** method is the first method to be called. This is where you should initialize variables.
- This method is called only once during the run time of your applet.
- **2. start( ) :** The **start( )** method is called after **init( )**.
- It is also called to restart an applet after it has been stopped.
- Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen.
- So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

- **3. paint( ) :** The **paint( )** method is called each time your applet's output must be redrawn.
- This situation can occur for several reasons.
- The applet window may be minimized and then restored.
- **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
- The **paint( )** method has one parameter of type **Graphics**.
- This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.
- This context is used whenever output to the applet is required.

- **4. stop( ) :** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example.
- When **stop( )** is called, the applet is probably running.
- You should use **stop( )** to suspend threads that don't need to run when the applet is not visible.
- You can restart them when **start( )** is called if the user returns to the page.
- **5. destroy( ):** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory.
- At this point, you should free up any resources the applet may be using.
- The **stop( )** method is always called before **destroy( )**.

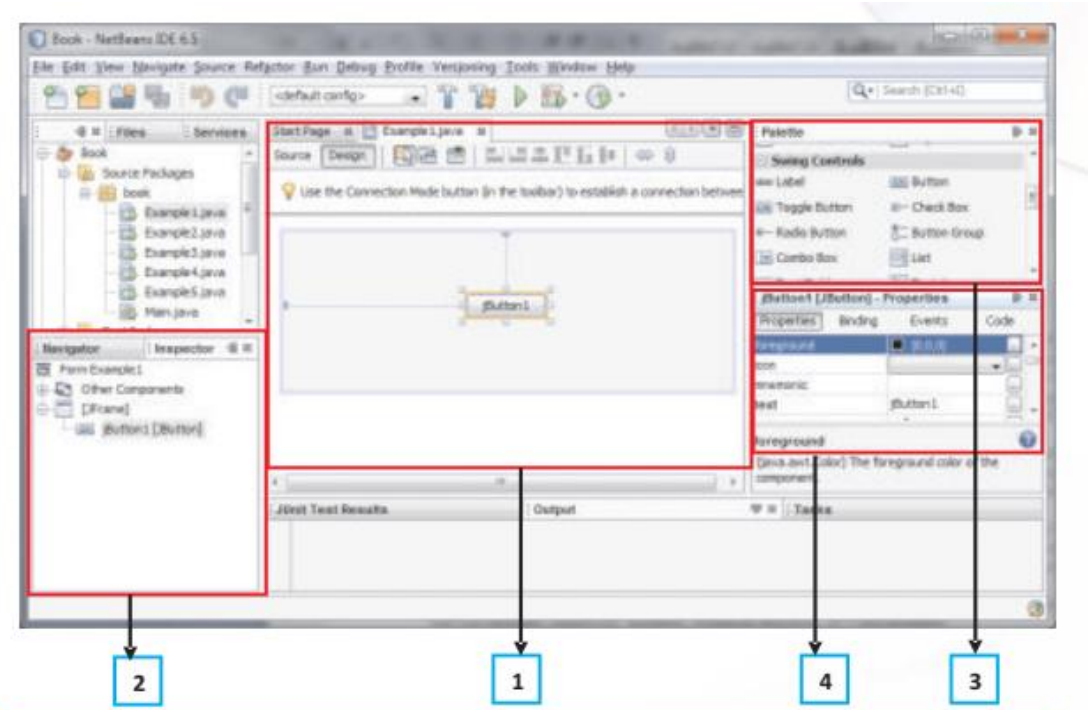
- Overriding update( ): In some situations, your applet may need to override another method defined by the AWT, called **update( )**.
- This method is called when your applet has requested that a portion of its window be redrawn.
- The default version of **update( )** simply calls **paint( )**.
- In general, overriding **update( )** is a specialized technique that is not applicable to all applets.

- Applets are displayed in a window, and AWT-based applets use the AWT to perform input and output.
- To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class.
- Typically, it is called from within either **update( )** or **paint( )**.
- It has the following general form: **void drawString(String *message*, int *x*, int *y*)**
- Here, *message* is the string to be output beginning at *x*,*y*.
- In a Java window, the upper-left corner is location 0,0.
- The **drawString( )** method will not recognize newline characters.
- If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin.

- These methods are defined by **Component**, and they have the following general forms:
  void setBackground(Color *newColor*) : To set the background colour of an applet's window.
  void setForeground(Color *newColor*) : To set the foreground colour of an applet's window.
  Here, *newColor* specifies the new color.
- setBackground(Color.green);
  setForeground(Color.red);
- You can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively.
- Color getBackground( )
  Color getForeground( )
- The class **Color** defines the constants.

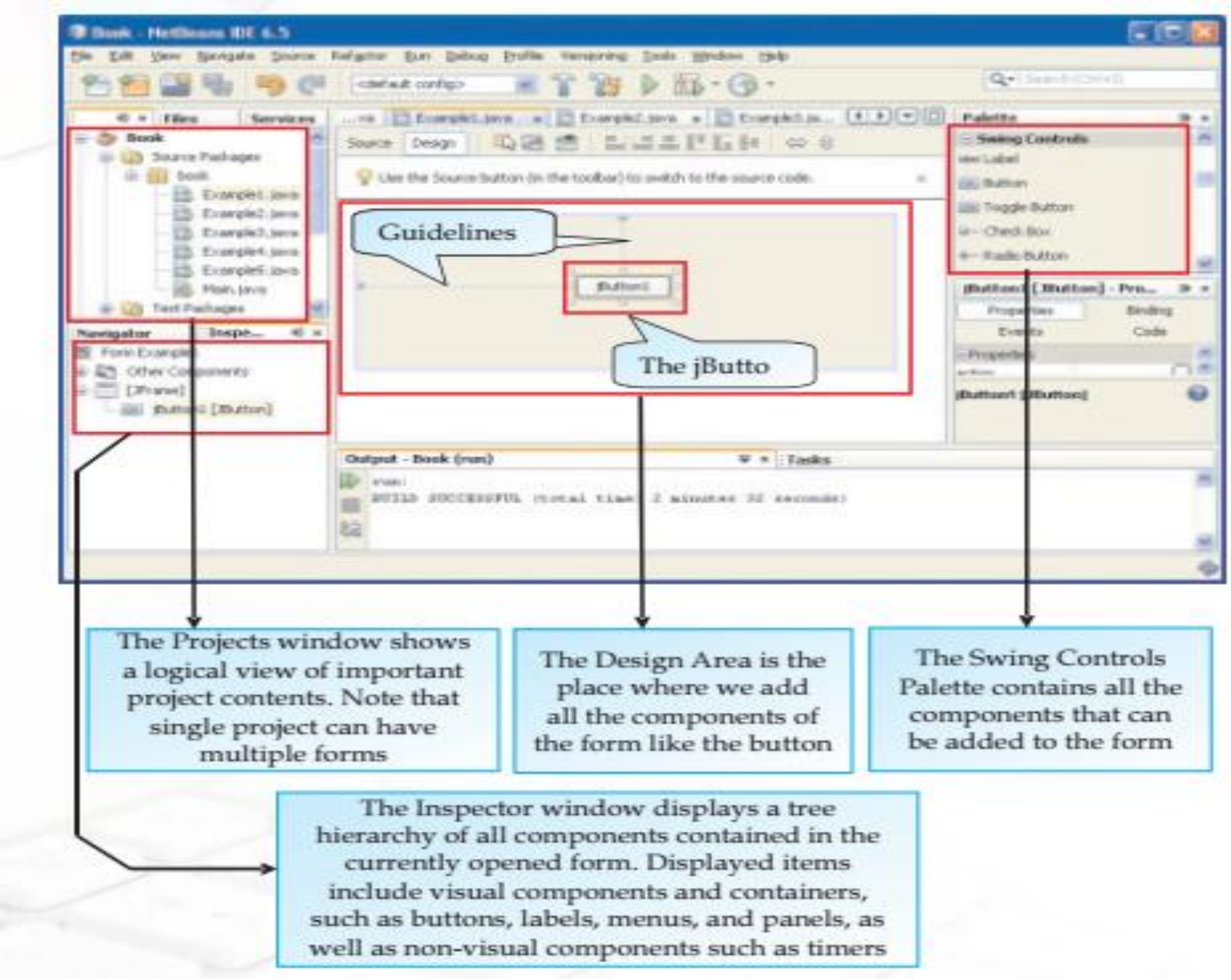
Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white

Netbeans GUI Interface

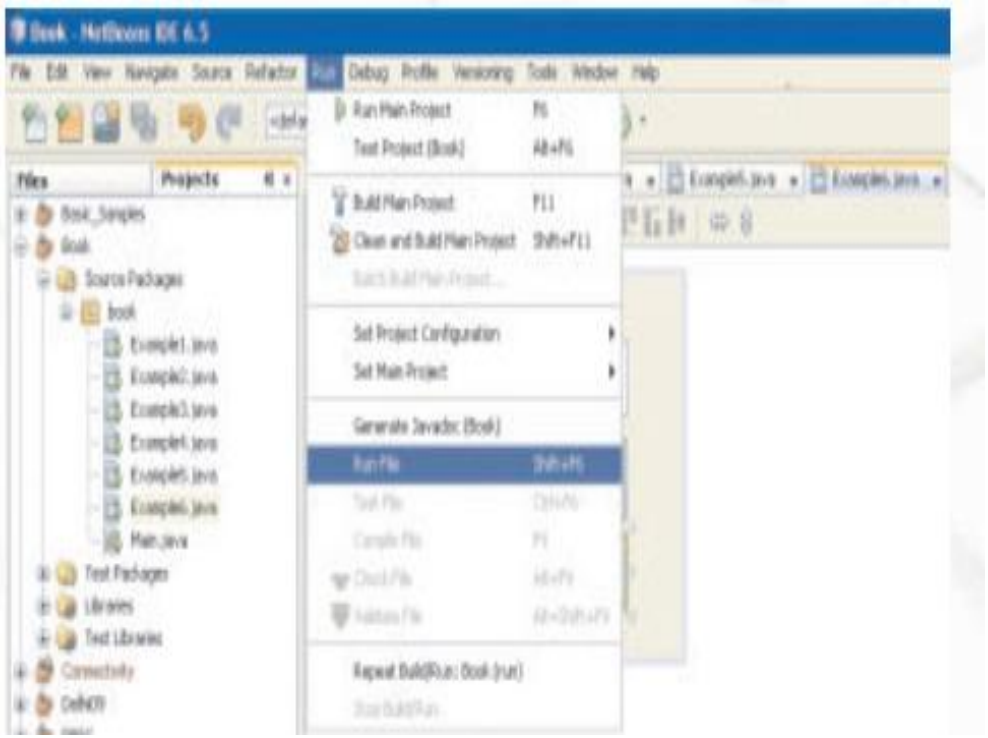


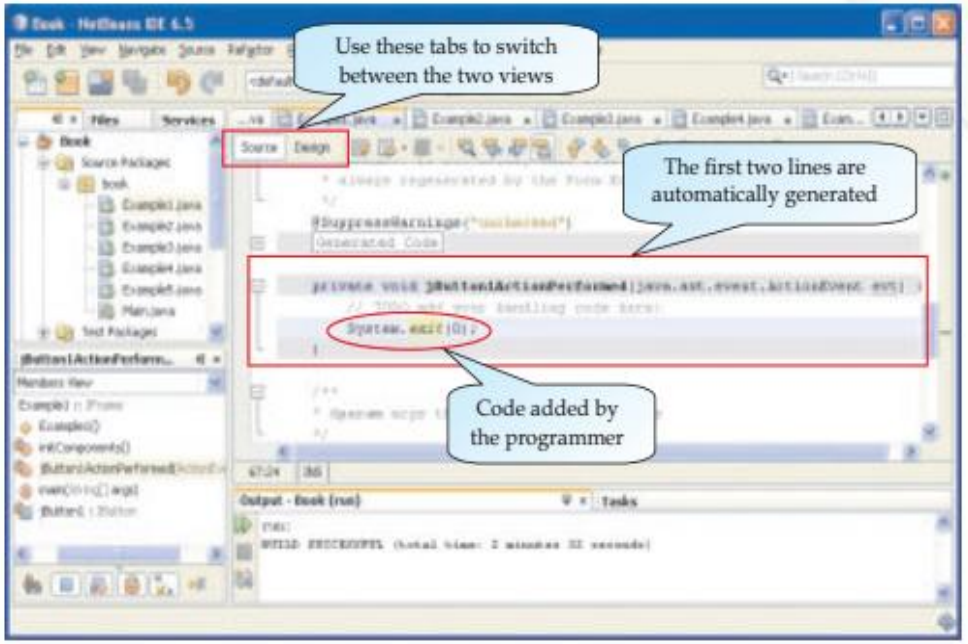
Components: 1. Design Area 2. Inspector Window 3. Palette 4. Properties Window

Different Type of Windows

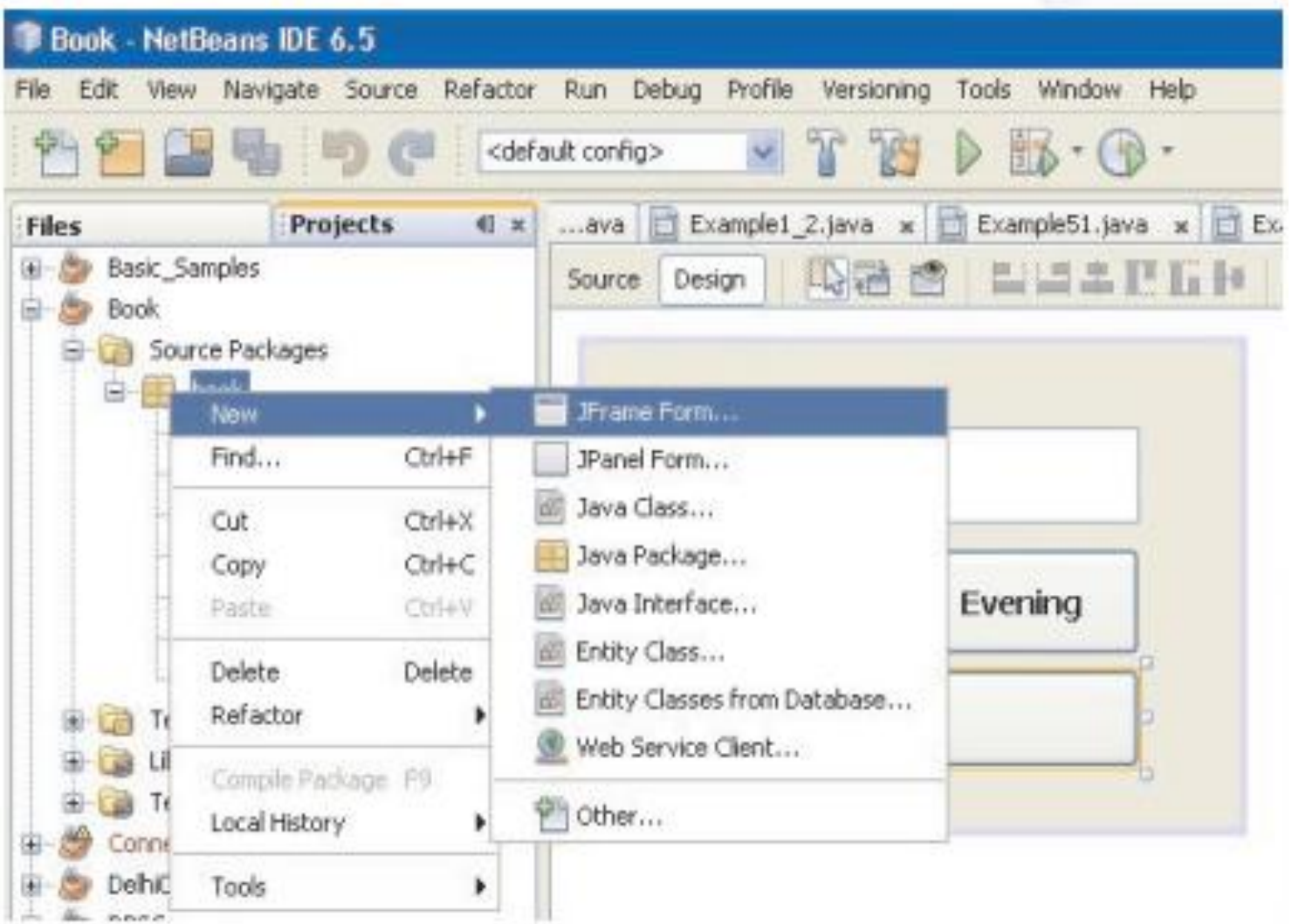


Executing a File




Use these tabs to switch between the two views
The first two lines are automatically generated
Code added by the programmer

Adding a New JFrame Form



Class	Objects	Data Members (Properties)	Methods
JTextField	*jTextField1*	• *Text* • *Editable* • *Enabled* • *ToolTipText*	• *setText()* • *getText()* • *setEditable()* • *setEnabled()* • *setToolTipText()*
JLabel	*jLabel1*	• *Text* • *Enabled* • *ToolTipText*	• *setText()* • *getText()* • *setEnabled()* • *setToolTipText()*
JTextArea	*jTextArea1*	• *Columns* • *Editable* • *Font* • *lineWrap* • *Rows* • *wrapStyleWord* • *toolTipText*	• *isEditable()* • *isEnabled()* • *getText()* • *setText()*
JButton	*jButton1*	• *Background* • *Enabled* • *Font* • *Foreground* • *Text* • *Label*	• *getText()* • *setText()*

JCheckBox	*jCheckBox1*	• *Button Group* • *Font* • *Foreground* • *Label* • *Selected* • *Text*	• *getText()* • *setText()* • *isSelected()* • *setSelected()*
JRadioButton	*jRadioButton1*	• *Background* • *Button Group* • *Enabled* • *Font* • *Foreground* • *Label* • *Selected* • *Text*	• *getText()* • *setText()* • *isSelected()* • *setSelected()*
JPasswordField	*jPasswordField1*	• *Editable* • *Font* • *Foreground* • *Text* • *Columns* • *toolTipText*	• *setEnabled()* • *setText()* • *getText()* • *isEnabled()*
JComboBox	*jComboBox1*	• *Background* • *ButtonGroup* • *Editable* • *Enabled*	• *getSelectedItem()* • *getSelectedIndex()* • *setModel()*

		• Font • Foreground • Model • SelectedIndex • SelectedItem • Text	
JList	jList1	• Background • Enabled • Font • Foreground • Model • SelectedIndex • SelectedItem • SelectionMode • Text	• getSelectedValue()
JTable	jTable1	• model	• addRow() • getModel()

Check Box

Method	Description
getText()	Returns the text displayed by the checkbox
setText(String s)	Sets the text displayed by the check box to the String value specified in parenthesis.
isSelected()	Returns the state of check box - true if selected else returns false.
setSelected()	Sets the state of the button - true if the button is selected, otherwise sets it to false.

List

- SINGLE implies that List box will allow only a single value to be selected.
- SINGLE_INTERVAL implies that List box allows single continuous selection of options using shift key of keyboard (i.e. values which occur in succession).
- MULTIPLE_INTERVAL implies that List box allows multiple selections of options using ctrl key of keyboard.

Syntax:

```
jList.isSelectedIndex(int num)
```

```
if (jList1.isSelectedIndex(0)==true)
```

Method	Description
getSelectedValue()	Returns the selected value when only a single item is selected. If multiple items are selected then it returns the first selected value. Returns null in case no item is selected
isSelectedIndex(int index)	Returns true if specified index is selected.

Combo Box

1. getSelectedIndex() - This method is used to return the index of the selected item. If an item is selected only then will the getSelectedIndex method return a value else it returns -1. The syntax of this method is given below:

   Syntax:

   ```
   jComboBox.getSelectedIndex()
   ```

2. getSelectedItem() - This method is used to return the selected item. The syntax of this methos is given below:

   Syntax:

   ```
   jComboBox.getSelectedItem()
   ```

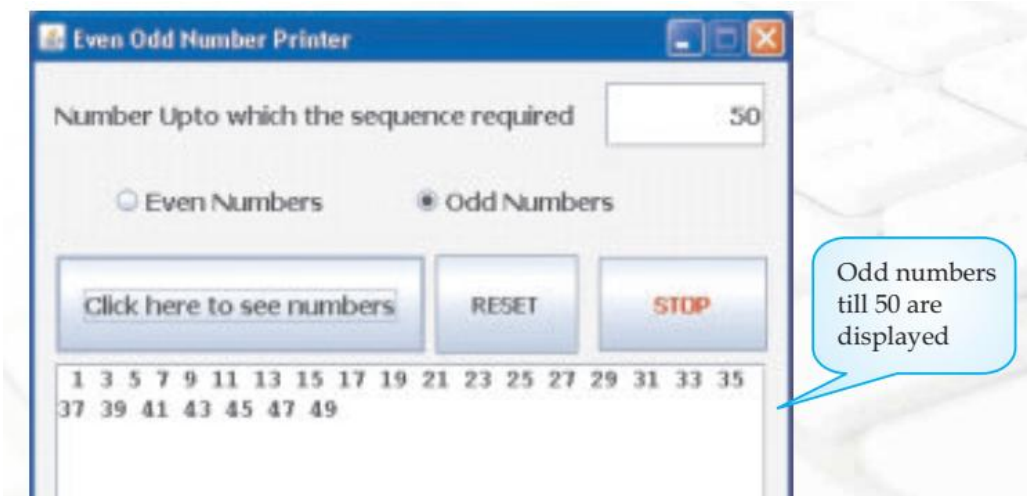
```
if (jComboBox1.getSelectedIndex()==0)
```

- Checks whether the item stored at the first position is selected or not using the getSelectedIndex() method

```
if (jComboBox1.getSelectedIndex()==0)
  JOptionPane.showMessageDialog(this, jComboBox1.getSelectedItem()+
      " - Known as Silicon Valley of India");
```

Text Area

```
int LastNumber=Integer.parseInt(jTextField1.getText());
for (int I=1;I<=LastNumber;I++)
  jTextArea1.setText(jTextArea1.getText()+
                                  " "+Integer.toString(I));
```




```
int LastNumber=Integer.parseInt(jTextField1.getText());
if (jRadioButton1.isSelected())  //Even Numbers required
{
 for (int I=2;I<=LastNumber;I+=2)
   jTextArea1.setText(jTextArea1.getText()+
                              " " +Integer.toString(I));
}
else if (jRadioButton2.isSelected())//Odd Numbers required
{
  for (int I=1;I<=LastNumber;I+=2)
    jTextArea1.setText(jTextArea1.getText()+
                              " " +Integer.toString(I));
}
else
  JOptionPane.showMessageDialog(this,
              "Click to select [Even] or [Odd] Option");
```

JDBC

- Steps for Java and MySql Connection
- 1. Import the Package
- 2. Load the Driver
- 3. Register the Driver
- 4. Establish the Connection
- 5. Create the Statement
- 6. Execute the Query
- 7. Process the Result
- 8. Close the Connection

Connecting Java Program with MySQL Database

```
import java.sql.*;                          //Importing java.sql package

        public class JavaMySQLConnect
{

            public static void main(String[] args)
{
                try {
                        Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/studentdb", "root", "root");
                                                                    //Establishing connection
                    System.out.println("Connected With the database successfully");
                                                                    //Message after successful connection

            } catch (SQLException e) {

                    System.out.println("Error while connecting to the database");
                                        //Message if something goes wrong while conneting to the database

                            }
            }

        }
```

- Java code to connect to MySQL database in NetBeans.
- import **java.sql** package so that you will be able to perform JDBC operations like creating and executing sql queries as it includes all the classes and interfaces to perform JDBC operations.
- Establish a connection using **DriverManager.getConnection(String URL)** and it returns a **Connection** reference.
- In **String URL** parameter you have to write like this **jdbc:mysql://localhost:3306/studentdb", "root", "root"** where,
  - **jdbc** is the API.
  - **mysq**l is the database.
  - **localhost** is the name of the server in which MySQL is running.
  - **3306** is the port number.
  - **studentdb** is the database name. If your database name is different, then you have to replace this name with your database name.
  - The first **root** is the username of the MySQL database. It is the default username for the MySQL database. If you have provided a different username in your MySQL database, then you have to provide that username.
  - The second **root** is the password that you give while installing the MySQL database. If your password is different, then provide that password at that place.
- SQL Exception might occur while connecting to the database, So you need to surround it with the **try-catch** block.

Inserting Data into The MySQL Database using Java

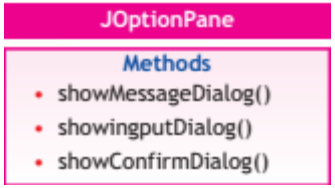
- Steps to insert data into the table.
- Create a PreparedStatement object and use SQL INSERT query to add data to the table.
- Specify the values for each parameter through the PreparedStatement object.
- Execute the query by calling the executeUpdate() method of the PreparedStatement object.

```
import java.sql.*;

  public class JavaMySQLConnect {

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

              Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/studentdb", "root", "root");//Establishing connection
              System.out.println("Connected With the database successfully");
              //Creating PreparedStatement Object
              PreparedStatement preparedStatement =connection.prepareStatement("insert into student values(?,?,?,?)");

              //Setting values for each parameter
              preparedStatement.setString(1,"1");
              preparedStatement.setString(2,"Mehtab");
              preparedStatement.setString(3,"Computer");
              preparedStatement.setString(4,"Ranchi");

              //Executing Query
              preparedStatement.executeUpdate();
              System.out.println("Data inserted Successfully");


      } catch (SQLException e) {

              System.out.println("Error while connecting to the database");

                  }
              }

              }
```

Class	Objects	Data Members (Properties)	Methods
JTextField	*jTextField1*	• *Text* • *Editable* • *Enabled* • *ToolTipText*	• *setText()* • *getText()* • *setEditable()* • *setEnabled()* • *setToolTipText()*
JLabel	*jLabel1*	• *Text* • *Enabled* • *ToolTipText*	• *setText()* • *getText()* • *setEnabled()* • *setToolTipText()*
JTextArea	*jTextArea1*	• *Columns* • *Editable* • *Font* • *lineWrap* • *Rows* • *wrapStyleWord* • *toolTipText*	• *isEditable()* • *isEnabled()* • *getText()* • *setText()*
JButton	*jButton1*	• *Background* • *Enabled* • *Font* • *Foreground* • *Text* • *Label*	• *getText()* • *setText()*

JCheckBox	jCheckBox1	• Button Group • Font • Foreground • Label • Selected • Text	• getText() • setText() • isSelected() • setSelected()
JRadioButton	jRadioButton1	• Background • Button Group • Enabled • Font • Foreground • Label • Selected • Text	• getText() • setText() • isSelected() • setSelected()
JPasswordField	jPasswordField1	• Editable • Font • Foreground • Text • Columns • toolTipText	• setEnabled() • setText() • getText() • isEnabled()
JComboBox	jComboBox1	• Background • ButtonGroup • Editable • Enabled	• getSelectedItem() • getSelectedIndex() • setModel()

		• Font • Foreground • Model • SelectedIndex • SelectedItem • Text	
JList	jList1	• Background • Enabled • Font • Foreground • Model • SelectedIndex • SelectedItem • SelectionMode • Text	• getSelectedValue()
JTable	jTable1	• model	• addRow() • getModel()


JOptionPane
Methods
showMessageDialog()
showingputDialog()
showConfirmDialog()