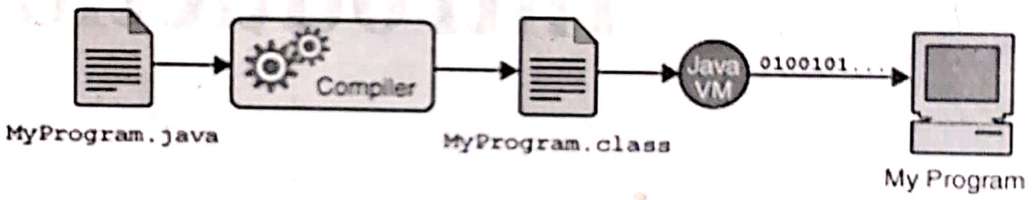
Java technology is both a programming language and a platform.

The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

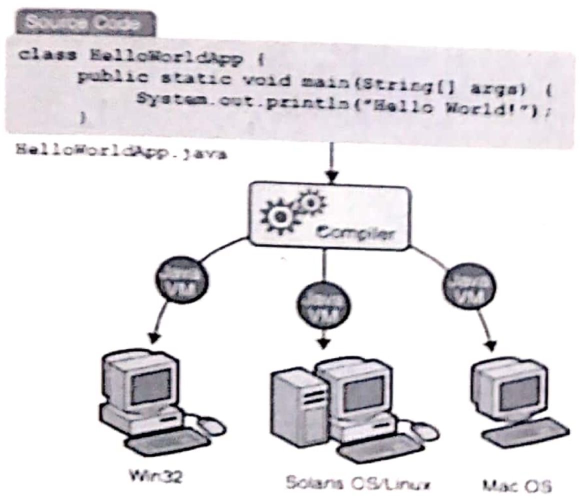
Simple	Architecture neutral
Object oriented	Portable
Distributed	High performance
Multithreaded	Robust
Dynamic	Secure

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to your processor; it instead contains *bytecodes* the machine language of the Java Virtual Machine[1] (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris ™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java HotSpot virtual machine, perform additional steps at runtime to give your application a performance boost. This include various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.



```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
```

The Java Platform

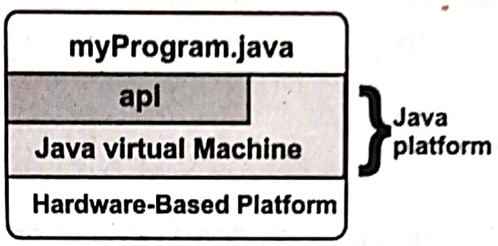
A *platform* is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The *Java Virtual Machine*
- The *Java Application Programming Interface* (API)

You've already been introduced to the Java Virtual Machine; it's the base for the Java platform and is ported onto various hardware-based platforms.

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.



The API and Java Virtual Machine insulate the program from the underlying hardware.

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

What Can Java Technology Do?

The general-purpose, high-level Java programming language is a powerful software platform. Every full implementation of the Java platform gives you the following features:

- **Development Tools**: The development tools provide everything you'll need for compiling, running, monitoring, debugging, and documenting your applications. As a new developer, the main tools you'll be using are the javac compiler, the java launcher, and the javadoc documentation tool.
- **Application Programming Interface (API)**: The API provides the core functionality of the Java programming language. It offers a wide array of useful classes ready for use in your own applications. It spans everything from basic objects, to networking and security, to XML generation and database access, and more.
- **Deployment Technologies**: The JDK software provides standard mechanisms such as the Java Web Start software and Java Plug-In software for deploying your applications to end users.
- **User Interface Toolkits**: The Swing and Java 2D toolkits make it possible to create sophisticated Graphical User Interfaces (GUIs).
- **Integration Libraries**: Integration libraries such as the Java IDL API, JDBC^{TM} API, Java Naming and Directory Interface^{TM} ("J.N.D.I.") API, Java RMI, and Java Remote Method Invocation over Internet Inter-ORB Protocol Technology (Java RMI-IIOP Technology) enable database access and manipulation of remote objects.

Data Types

Java is a strongly typed language. This means that every variable must have a declared type. There are eight primitive types in Java. Four of them are integer types; two are floating-point number types; one is the character type char, used for characters in the Unicode encoding, and one is a boolean type for truth values.

Integers

The integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four integer types :
int 4 bytes -2,147,483,648 to 2,147,483, 647 (just over 2 billion)
short 2 bytes -32,768 to 32,767
long 8 bytes -9,223,372,036,854,775,808L to 9,223,372,036,854,775,807L
byte 1 byte -128 to 127

In most situations, the int type is the most practical. If you want to represent the number of inhabitants of our planet, you'll need to resort to a long. The byte and short types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Under Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code. This alleviates a major pain for the programmer who wants to move software from one platform to another, or even between operating systems on the same platform. In contrast, C and C++ programs use the most efficient integer type for each processor. As a result, a C program that runs well on a 32-bit processor may exhibit integer overflow on a 16-bit system. Since Java programs must run with the same results on all machines, the ranges for the various types are fixed.

Long integer numbers have a suffix Example (for example, 4000000000L). Hexadecimal numbers have a prefix 0x (for example, 0xCAFE). Octal numbers have a prefix 0. For example, 010 is 8. Naturally, this can be confusing, and we recommend against the use of octal constants.

Floating-Point Types

The floating-point types denote numbers with fractional parts. There are two floating-point types :
float 4 bytes approximately ±3.40282347E+38F (67 significant decimal digits)
double 8 bytes approximately ±1.79769313486231570E+308 (15 significant decimal digits)

The name double refers to the fact that these numbers have twice the precision of the float type. (Some people call these double-precision numbers.) Here, the type to choose in most applications is double. The limited precision of float is simply not sufficient for many situations. Seven significant (decimal) digits may be enough to precisely express your annual salary in dollars and cents, but it won't be enough for your company president's salary. The only reason to use float is in the rare situations in which the slightly faster processing of single-precision numbers is important, or when you need to store a large number of them. Numbers of type float have a suffix F, for example, 3.402F. Floating-point numbers without an F suffix (such as 3.402) are always considered to be of type double. You can optionally supply the D suffix such as 3.402D.

All floating-point computations follow the IEEE 754 specification. In particular, there are three special floating-point values:
positive infinity
negative infinity

NaN (not a number)
to denote overflows and errors. For example, the result of dividing a positive number by 0 is positive infinity. Computing 0/0 or the square root of a negative number yields NaN.

Boolean

The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

Char

The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Variables

In Java, every variable has a type. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:
double salary;
int vacationDays;
long earthPopulation;
char yesChar;
boolean done;
Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement.
The rules for a variable name are as follows:
A variable name must begin with a letter, and must be a sequence of letters or digits. Note that the terms "letter" and "digit" are much broader in Java than in most languages. A letter is defined as 'A"Z', 'a"z', '_', or any Unicode character that denotes a letter in a
language. For example, German users can use umlauts such as ' ä' in variable names; Greek speakers could use a p. Similarly, digits are '0"9' and any Unicode characters that denote a digit in a language. Symbols like '+' or '©' cannot be used inside variable names, nor can spaces. All characters in the name of a variable are significant and case is also significant. The length of a variable name is essentially unlimited.
You can have multiple declarations on a single line
int i, j; // both are integers
However, we don't recommend this style. If you define each variable separately, your programs are easier to read.

Assignments and Initializations

After you declare a variable, you must explicitly initialize it by means of an assignment statementyou can never use the values of uninitialized variables. You assign to a previously declared variable using the variable name on the left, an equal sign (=), and then some Java expression that has an appropriate value on the right.
int vacationDays; // this is a declaration
vacationDays = 12; // this is an assignment
Here's an example of an assignment to a character variable:
char yesChar;
yesChar = 'Y';
One nice feature of Java is the ability to both declare and initialize a variable on the same line.
For example:
int vacationDays = 12; // this is an initialization
Finally, in Java you can put declarations anywhere in your code. For example, the following is valid code in Java:
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // ok to declare variable here

Constants (Final Keyword)

In Java, you use the keyword final to denote a constant. For example,

```
public class Constants
        {
        public static void main(String[] args)
        {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeter: "
        + paperWidth * CM_PER_INCH + " by "
        + paperHeight * CM_PER_INCH);
        }
        }
```

The keyword final indicates that you can assign to the variable once, then its value is set once and for all. It is customary to name constants in all upper case.
It is probably more common in Java to want a constant that is available to multiple methods inside a single class. These are usually called class constants. You set up a class constant with the keywords static final. Here is an example of using a class constant:

```
public class Constants2
{
public static final double CM_PER_INCH = 2.54;;
public static void main(String[] args)
{
double paperWidth = 8.5;
double paperHeight = 11;
System.out.println("Paper size in centimeter: "
+ paperWidth * CM_PER_INCH + " by "
+ paperHeight * CM_PER_INCH);
}
}
```

Operators

The usual arithmetic operators + * / are used in Java for addition, multiplication, and division. The / operator denotes integer division if both integers, and floating-point division otherwise. Integer remainder (that is, the denoted by %. For example, 15 / 2 is 7, 15 % 2 is 1, and 15.0 / 2 is 7.5.
Note that integer division by 0 raises an exception, whereas floating-point division an infinite or NaN result.
You can use the arithmetic operators in your variable initializations:

```
int n = 5;
int a = 2 * n; // a is 10
```

There is a convenient shortcut for using binary arithmetic operators in an example,

```
x += 4;
```

is equivalent to

```
x = x + 4;
```

(In general, place the operator to the left of the = sign, such as *= or %=.)

Increment and Decrement Operators

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: x++ adds 1 to the current value of the variable x, and x-- subtracts 1 from it. For example, the code subtracts 1 from it. For example, the code

```
int n = 12;
N++;
```

changes n to 13. Because these operators change the value of a variable, they cannot be applied to numbers themselves. For example, 4++ is not a legal statement.
There are actually two forms of these operators; you have seen the "postfix" form of the operator that is placed after the operand. There is also a prefix form, ++n. Both change the value of the variable by 1. The difference between the two only appears when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

We recommend against using ++ inside other expressions as this often leads to confusing code and annoying bugs.
(Of course, while it is true that the ++ operator gives the C++ language its name, it also led to the first joke about the language. C++ haters point out that even the name of the language contains a bug: "After all, it should really be called ++C, since we only want to use a language after it has been improved.")

Relational and boolean Operators

Java has the full complement of relational operators. To test for equality you use a double equal sign, ==. For example, the value of 3 == 7 is false.
Use a != for inequality. For example, the value of 3 != 7 is true.
Finally, you have the usual < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators.

Java, following C++, uses && for the logical "and" operator and || for the logical "or" operator. As you can easily remember from the != operator, the exclamation point ! is the logical negation operator. The && and || operators are evaluated in "short circuit" fashion. This means that when you have an expression like:
A && B
once the truth value of the expression A has been determined to be false, the value for the expression B is not calculated. For example, in the expression
x != 0 && 1 / x > x + y // no division by 0
the second part is never evaluated if x equals zero. Thus, 1 / x is not computed if x is zero, and no divide-by-zero error can occur.
Similarly, if A evaluates to be true, then the value of A || B is automatically true, without evaluating B.
Finally, Java supports the ternary ?: operator that is occasionally useful. The expression
condition ? e1 : e2
evaluates to e1 if the condition is true, to e2 otherwise. For example,
x < y ? x : y
gives the smaller of x and y.

Bitwise Operators

When working with any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are:
& ("and") | ("or") ^ ("xor") ~ ("not")
These operators work on bit patterns. For example, if n is an integer variable, then
int fourthBitFromRight = (n & 8) / 8;
gives you a one if the fourth bit from the right in the binary representation of n is one, and a zero if not. Using & with the appropriate power of two lets you mask out all but a single bit.

Core

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order tha they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then and if-then-else Statements

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes(){
    if (isMoving){ // the "if" clause: bicycle must moving
        currentSpeed--; // the "then" clause: decrease current speed
    }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes(){
    if (isMoving) currentSpeed--; // same as above, but without braces
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes(){
    if (isMoving) {
        currentSpeed--;
    }else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
    public static void main(String[] args) {
            int testscore = 76;
            char grade;

            if (testscore >= 90) {
                grade = 'A';
            }else if (testscore >= 80) {
                grade = 'B';
            }else if (testscore >= 70) {
                grade = 'C';
            }else if (testscore >= 60) {
                grade = 'D';
            }else {
                grade = 'F';
            }
            System.out.println("Grade = " + grade);
        }
    }
```

The output from the program is:

```
  Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

Multiple Selections the switch Statement

The if/else construct can be cumbersome when you have to deal with multiple selections with many alternatives. Java has a switch statement that is exactly like the switch statement in C and C++, warts and all.

```
String input = JOptionPane.showInputDialog
("Select an option (1, 2, 3, 4)");
int choice = Integer.parseInt(input);
switch (choice)
        {
        case 1:
        . . .
        break;
        case 2:
        . . .
        break;
        case 3:
        . . .
        break;
        case 4:
        . . .
        break;
        default:
        // bad input
        . . .
        break;
        }
```

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The while statement evaluates expression, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
    // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
}while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        }while (count <= 11);
    }
}
```

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The initialization expression initializes the loop; it's executed once, as the loop begins.
- When the termination expression evaluates to false, the loop terminates.
- The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

Classes and Objects
Classes

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior

```
Public class Student {

    // the Student class has two fields
    public int id;
    public String name;

    // the student class has methods
    public void setId(int newValue) {
        id = newValue;
    }

    public void setName(String newValue) {
        name = newValue;
    }
}
```

You can also add modifiers like public or private at the very beginningso you can see that the opening line of a class declaration can become quite complicated. The modifiers public and private, which determine what other classes can access Student class, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the extends and implements keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as public, private, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can `implement more than one interface.
5. The class body, surrounded by braces, {}.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarationsexcept that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Objects

A typical Java program creates many objects, which as you know, interact by invoking methods Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small program, called CreateObjectDemo, that creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        //Declare and create a point object
        //and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        //display rectOne's width, height, and area
        System.out.println("Width of rectOne: " +
              rectOne.width);
        System.out.println("Height of rectOne: " +
              rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        //set rectTwo's position
        rectTwo.origin = originOne;

        //display rectTwo's position
        System.out.println("X Position of rectTwo: "
              + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
              + rectTwo.origin.y);

        //move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: "
              + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
              + rectTwo.origin.y);
    }
}
```

Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

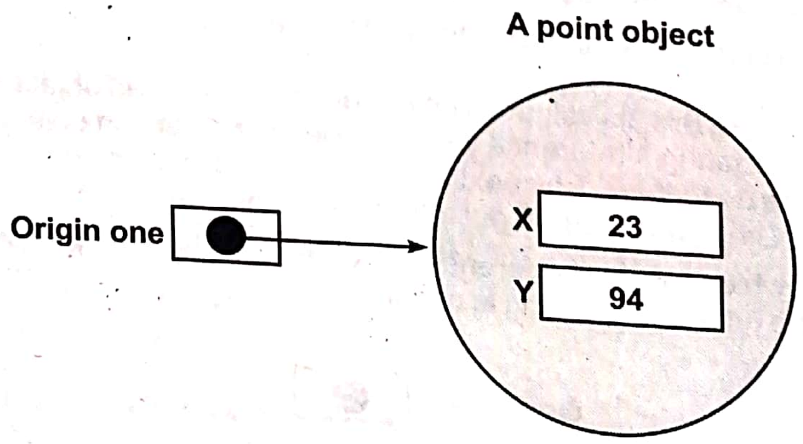
The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.
The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:
Point originOne = new Point(23, 94);
The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:
int height = new Rectangle().height;
This statement will be discussed in the next section.

Initializing an Object

Here's the code for the Point class:

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:
Point originOne = new Point(23, 94);
The result of executing this statement can be illustrated in the next figure:



Here's the code for the Rectangle class, which contains four constructors:

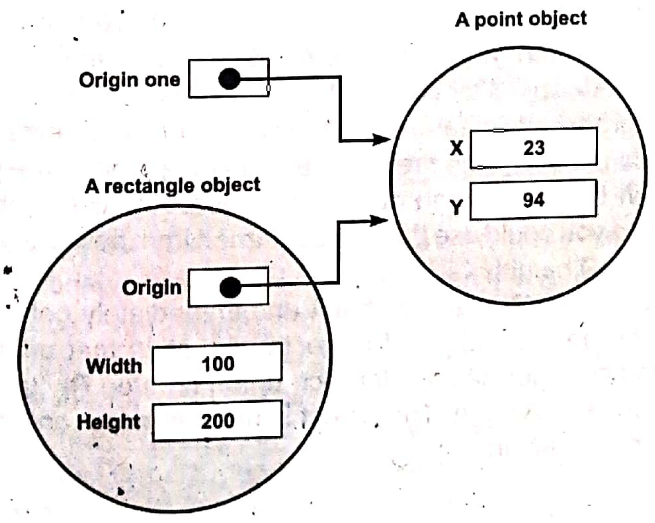
```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;
```

```
        //four constructors
        public Rectangle() {
                origin = new Point(0, 0);
        }
        public Rectangle(Point p) {
                origin = p;
        }
        public Rectangle(int w, int h) {
                origin = new Point(0, 0);
                width = w;
                height = h;
        }
        public Rectangle(Point p, int w, int h) {
                origin = p;
                width = w;
                height = h;
        }

        //a method for moving the rectangle
        public void move(int x, int y) {
                origin.x = x;
                origin.y = y;
        }

        //a method for computing the area of the rectangle
        public int getArea() {
                return width * height;
        }
}
```

Each constructor lets you provide initial values for the rectangle's size and width, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the same Point object an object can have multiple references to it, as shown in the next figure:



Core

Using the this Keyword

Within an instance method or a constructor, this is a reference to the current object the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

The most common reason for using this keyword is because a field is shadowed by a method or constructor parameter.
For example, the Point class was written like this

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the second constructor shadows one of the object's fieldsinside the constructor x is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

Using this with a Constructor

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation. Here's another Rectangle class, with a different implementation from the one in the Objects section.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
```

Static Fields and Methods

In all sample programs that you have seen, the main method is tagged with the static modifier. We are now ready to discuss the meaning of this modifier.

Static Fields

If you define a field as static, then there is only one such field per class. In contrast, each object has its own copy of all instance fields. For example, let's suppose we want to assign a unique identification number to each employee. We add an instance field id and a static field nextId to the Employee class:

```
class Employee
{
. . .
private int id;
private static int nextId = 1;
}
```

Now, every employee object has its own id field, but there is only one nextId field that is shared among all instances of the class. Let's put it another way. If there are one thousand objects of the Employee class, then there are one thousand instance fields id, one for each object. But there is a single static field nextId. Even if there are no employee objects, the static field nextId is present. It belongs to the class, not to any individual object.

Static Methods

Static methods are methods that do not operate on objects. For example, the pow method of the Math class is a static method. The expression:
Math.pow(x, y)
computes the power xy. It does not use any Math object to carry out its task. In other words, it has no implicit parameter.
In other words, you can think of static methods as methods that don't have a this parameter.
Because static methods don't operate on objects, you cannot access instance fields from a static method. But static methods can access the static fields in their class. Here is an example of such a static method:

```
public static int getNextId()
{
return nextId; // returns static field
}
```

To call this method, you supply the name of the class:

Core

Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobilestop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning Satellite) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know how the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

In the Java programming language, an interface is a reference type, similar to a class, that can contain only constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiatedthey can only be implemented by classes or extended by other interfaces. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures
   int turn(Direction direction,   // An enum with values RIGHT, LEFT
            double radius, double startSpeed, double endSpeed);
   int changeLanes(Direction direction, double startSpeed, double endSpeed);
   int signalTurn(Direction direction, boolean signalOn);
   int getRadarFront(double distanceToCar, double speedOfCar);
   int getRadarRear(double distanceToCar, double speedOfCar);
        ......
   // more method signatures
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar {

   // the OperateCar method signatures, with implementation --
   // for example:
   int signalTurn(Direction direction, boolean signalOn) {
      //code to turn BMW's LEFT turn indicator lights on
      //code to turn BMW's LEFT turn indicator lights off
      //code to turn BMW's RIGHT turn indicator lights on
```

```
        //code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes
    // not visible to clients of the interface

}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard Application Programming Interface (API). APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secretin fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

Interfaces and Multiple Inheritance

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance (inheritance is discussed later in this lesson), but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

Defining an Interface

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1,
                                  Interface2, Interface3 {

    // constant declarations
    double E = 2.718282;  // base of natural logarithms

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
    }
```

The clone() Method

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:
aCloneableObject.clone();
Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as
protected Object clone() throws CloneNotSupportedException
-- or --
public Object clone() throws CloneNotSupportedException
if you are going to write a clone() method to override the one in Object.
If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.
The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.
For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independentto decouple them, you must override clone() so that it clones the object and ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

The equals() Method

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object references are equalthat

The finalize() Method

The Object class provides a callback method, finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothingyou can override finalize() to do cleanup, such as freeing resources.
The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

The getClass() Method

You cannot override getClass.
The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's class is "
                obj.getClass().getSimpleName());
}
```

The Class class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

String greeting = "Hello world!";

In this case, "Hello world!" is a string literala series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its valuein this case, Hello world!.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = {'h', 'e', 'l', 'l', 'o', '.'};
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays hello.

The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

String Length

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

Concatenating Strings

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the + operator, as in

```
"Hello," + " world" + "!"
```

which results in

```
"Hello, world!"
```

Creating Format Strings

You have seen the use of the printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

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

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

you can write

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

Converting Strings to Numbers

Frequently, a program ends up with numeric data in a string objecta value entered by the user, for example.

The Number subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each provide a class method named valueOf that converts a string to an object of that type. Here is an example, ValueOfDemo , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

```
public class ValueOfDemo {
   public static void main(String[] args) {

      //this program requires two arguments on the command line
      if (args.length == 2) {
                               //convert strings to numbers
         float a = (Float.valueOf(args[0]) ).floatValue();
         float b = (Float.valueOf(args[1]) ).floatValue();

         //do some arithmetic
         System.out.println("a + b = " + (a + b) );
         System.out.println("a - b = " + (a - b) );
         System.out.println("a * b = " + (a * b) );
         System.out.println("a / b = " + (a / b) );
         System.out.println("a % b = " + (a % b) );
      }else {
         System.out.println("This program requires two command-line arguments.");
      }
   }
}
```

Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```
int i;
String s1 = "" + i; //Concatenate "i" with an empty string;
                    //conversion is handled for you.
or
String s2 = String.valueOf(i);  //The valueOf class method.
```

Each of the Number subclasses includes a class method, toString(), that will convert its primitive type to a string. For example:

Manipulating Characters in a String

The String class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Getting Characters and Substrings by Index

You can get the character at a particular index within a string by invoking the charAt() accessor method. The index of the first character is 0, while the index of the last character is length()-1. If you want to get more than one consecutive character from a string, you can use the substring method.
The substring Methods in the String Class Method Description String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character + 1. String substring(int beginIndex) Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

Other Methods for Manipulating Strings

Here are several other String methods for manipulating strings:

Other Methods in the String Class for Manipulating Strings

Method	Description
String[] split(String regex) String[] split(String regex, int limit)	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."
CharSequence subSequence(int beginIndex, int endIndex)	Returns a new character sequence constructed from beginIndex index up until endIndex - 1.
String trim()	Returns a copy of this string with leading and trailing white space removed.
String toLowerCase() String toUpperCase()	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

Searching for Characters and Substrings in a String

The following table describes the various string search methods.

The Search Methods in the String Class

Method	Description
int indexOf(int ch) int lastIndexOf(int ch)	Returns the index of the first (last) occurrence of the specified character.
Int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
int indexOf(String str) int lastIndexOf(String str)	Returns the index of the first (last) occurrence of the specified substring.
int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
Boolean contains(CharSequence s)	Returns true if the string contains the specified character sequence.

Note: CharSequence is an interface that is implemented by the String class. Therefore, you can use a string as an argument for the contains() method.

Replacing Characters and Substrings into a String

The String class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have removed from a string with the substring that you want to insert.

The String class does have four methods for replacing found characters or substrings, however. They are:

Method	Description
String replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String replace(CharSequence target, CharSequence replacement)	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.
String replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement.

Exception Handling

Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred. This Chapter discusses how to use exceptions appropriately in your programs and designs. Look to this companion article for a tutorial on the nuts and bolts of what exceptions are and how they work in the Java language and virtual machine.

When a method encounters an abnormal condition (an exception condition) that it can't handle itself, it may throw an exception, Throwing an exception is like throwing a beeping, flashing red ball to indicate there is a problem that can't be handled where it occurred. Somewhere, you hope, this ball will be caught and the problem will be dealt with. Exceptions are caught by handlers positioned along the thread's method invocation stack. If the calling method isn't prepared to catch the exception, it throws the exception up to its calling method, and so on. If one of the threads of your program throws an exception that isn't caught by any method along the method invocation stack, that thread will expire. When you program in Java, you must position catchers (the exception handlers) strategically, so your program will catch and handle all exceptions from which you want your program to recover.

Exception classes

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable. Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw. A small part of this family is shown in Figure 1.

As you can see in Figure 1, Throwable has two direct subclasses, Exception and Error. Exceptions (members of the Exception family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread. Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

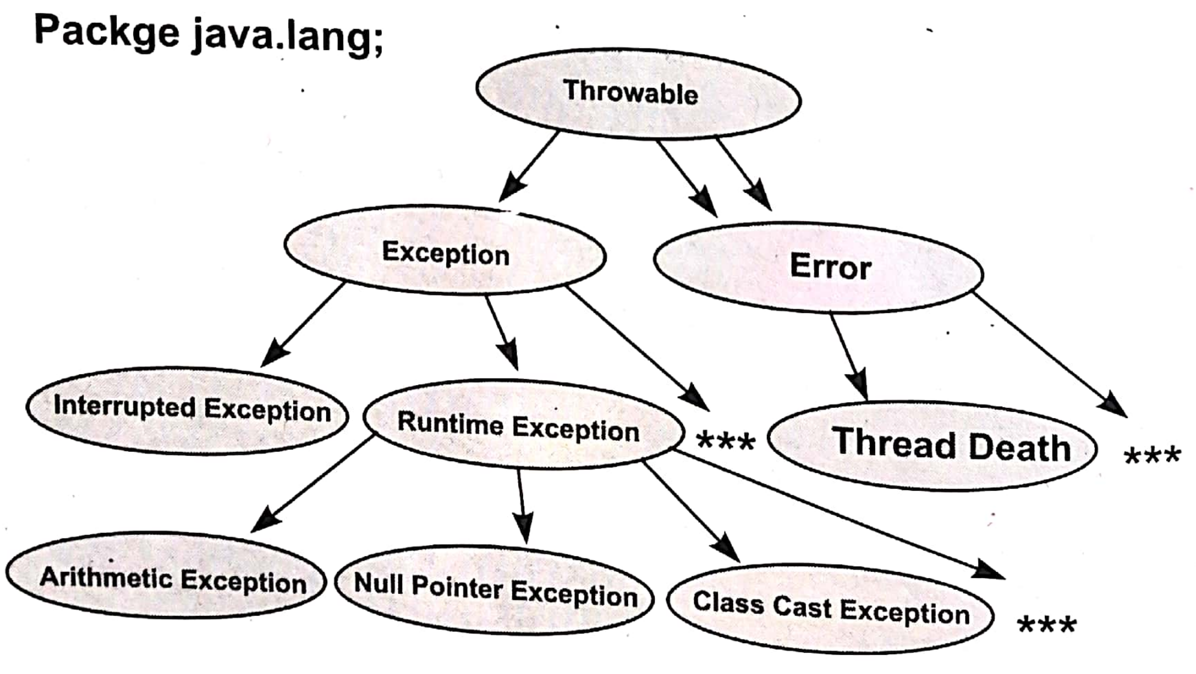


Figure 1. A partial view of the Throwable family

In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of the Throwable family. In general, however, the throwable classes you define should extend class Exception. They should be "exceptions." The reasoning behind this rule will be explained later in this article.

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine. For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

Other times, however, you will want to convey more information about the abnormal condition than a class from java.lang will allow. Usually, the class of the exception object itself indicates the type of abnormal condition that was encountered. For example, if a thrown exception object has class IllegalArgumentException, that indicates someone passed an illegal argument to a method. Sometimes you will want to indicate that a method encountered an abnormal condition that isn't represented by a class in the Throwable family of java.lang.

As an example, imagine you are writing a Java program that simulates a customer of a virtual café drinking a cup of coffee. Consider the exceptional conditions that might occur while the customer sips. The class hierarchy of exceptions shown in Figure 2 represents a few possibilities.

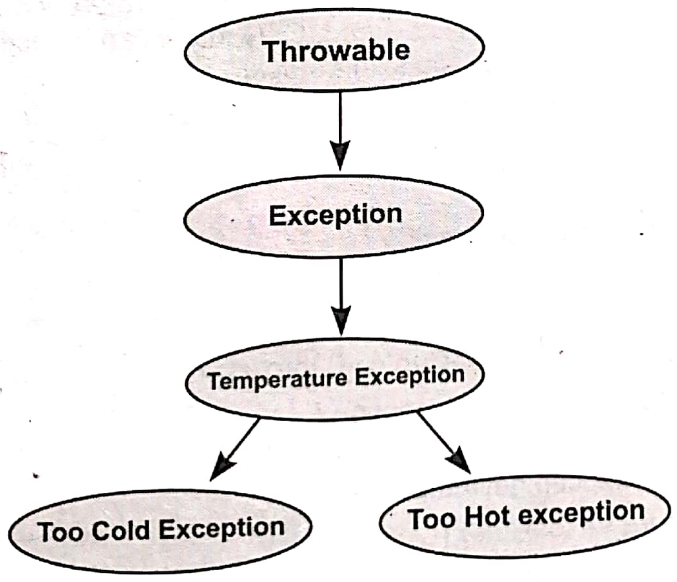


Figure 2. Exception hierarchy for coffee sipping

If the customer discovers, with dismay, that the coffee is cold, your program could throw a TooColdException. On the other hand, if the customer discovers that the coffee is overly hot, your program could throw a TooHotException. These conditions could be exceptions because they are (hopefully) not the normal situation in your café. (Exceptional conditions are not necessarily rare, just outside the normal flow of events.) The code for your new exception classes might look like this:

```
// In Source Packet in file except/ex1/TemperatureException.java
class TemperatureException extends Exception {
}


// In Source Packet in file except/ex1/TooColdException.java
class TooColdException extends TemperatureException {
}
```

```
// In Source Packet in file except/ex1/TooHotException.java
class TooHotException extends TemperatureException {
}
```

This family of classes, the TemperatureException family, declares three new types of exceptions for your program to throw. Note that each exception indicates by its class the kind of abnormal condition that would cause it to be thrown: TemperatureException indicates some kind of problem with temperature; TooColdException indicates something was too cold; and TooHotException indicates something was too hot. Note also that TemperatureException extends Exception -- not Throwable, Error, or any other class declared in java.lang.

Throwing exceptions

To throw an exception, you simply use the throw keyword with an object reference, as in:

```
throw new TooColdException();
```

The type of the reference must be Throwable or one of its subclasses.

The following code shows how a class that represents the customer, class VirtualPerson, might throw exceptions if the coffee didn't meet the customer's temperature preferences. Note that Java also has a throws keyword in addition to the throw keyword. Only throw can be used to throw an exception. The meaning of throws will be explained later in this article.

```
// In Source Packet in file except/ex1/VirtualPerson.java
class VirtualPerson {

    private static final int tooCold = 65;
    private static final int tooHot = 85;

    public void drinkCoffee(CoffeeCup cup) throws
        TooColdException, TooHotException {

        int temperature = cup.getTemperature();
        if (temperature <= tooCold) {
            throw new TooColdException();
        }
        else if (temperature >= tooHot) {
            throw new TooHotException();
        }
        //...
    }
    //...
}

// In Source Packet in file except/ex1/CoffeeCup.java
class CoffeeCup {
    // 75 degrees Celsius: the best temperature for coffee
    private int temperature = 75;
    public void setTemperature(int val) {
            temperature = val;
        }
        public int getTemperature() {
            return temperature;
        }
        //...
    }
```

Catching exceptions

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The try block places a fence around a bit of code that is under the watchful eye of the associated catchers. If the bit of code delimited by the try block throws an exception, the associated catch clauses will be examined by the Java virtual machine. If the virtual machine finds a catch clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that catch clause.

As an example, consider a program that requires one argument on the command line, a string that can be parsed into an integer. When you have a String and want an int, you can invoke the parseInt() method of the Integer class. If the string you pass represents an integer, parseInt() will return the value. If the string doesn't represent an integer, parseInt() throws NumberFormatException. Here is how you might parse an int from a command-line argument:

```
// In Source Packet in file except/ex1/Example1.java
class Example1 {
    public static void main(String[] args) {

        int temperature = 0;
        if (args.length > 0) {
            try {
                temperature = Integer.parseInt(args[0]);
            }
            catch(NumberFormatException e) {
                System.out.println(
                    "Must enter integer as first argument.");
                return;
            }
        }
        else {
            System.out.println(
                "Must enter temperature as first argument.");
            return;
        }

        // Create a new coffee cup and set the temperature of
        // its coffee.
```

Core

```
            CoffeeCup cup = new CoffeeCup();
            cup.setTemperature(temperature);

            // Create and serve a virtual customer.
            VirtualPerson cust = new VirtualPerson();
            VirtualCafe.serveCustomer(cust, cup);
        }
    }
```
Here, the invocation of parseInt() sits inside a try block. Attached to the try block is a catch clause that catches NumberFormatException:
```
    catch(NumberFormatException e) {
        System.out.println(
            "Must enter integer as first argument.");
        return;
    }
```
The lowercase character e is a reference to the thrown (and caught) NumberFormatException

Scanned with CamScanner

The finally Clause

When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource that only it knows about and if that resource must be cleaned up. One solution is to catch and rethrow all exceptions. But this solution is tedious because you need to clean up the resource allocation in two places, in the normal code and in the exception code.

Java has a better solution, the finally clause:

```
Graphics g = image.getGraphics();
try
{
code that might
throw exceptions
}
catch (IOException e)
{
show error dialog
}
finally
{
g.dispose();
}
```

This program executes the code in the finally clause whether or not an exception was caught. This means, in the example code above, the program will dispose of the graphics context under all circumstances.

Defining your own exception classes

Now you know how to write exception handlers for those exception objects that are thrown by the runtime system, and thrown by methods in the standard class library.

It is also possible for you to define your own exception classes, and to cause objects of those classes to be thrown whenever an exception occurs. In this case, you get to decide just what constitutes an exceptional condition.

For example, you could write a data-processing application that processes integer data obtained via a TCP/IP link from another computer. If the specification for the program indicates that the integer value 10 should never be received, you could use an occurrence of the integer value 10 to cause an exception object of your own design to be thrown.

Choosing the exception type to throw

Before throwing an exception, you must decide on its type. Basically, you have two choices in this regard:

- Use an exception class written by someone else, such as the myriad of exception classes defined in the Java standard library.
- Define an exception class of your own.

Choosing a superclass to extend

If you decide to define your own exception class, it must be a subclass of **Throwable**. You must decide which class you will extend.

The two existing subclasses of **Throwable** are **Exception** and **Error**. Given the earlier description of **Error** and its subclasses, it is not likely that your exceptions would fit the **Error** category. (In concept, errors are reserved for serious hard errors that occur deep within the system.)

```
class MyException extends Exception{
 MyException(String message){//constr
  super(message);
 }//end constructor
}//end MyException class
```

Threads

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

Defining a Thread

In Java, a thread is defined as a special class in the java.lang package. The thread class implements an interface called the "Runnable" interface, which defines a single abstract method called "run".

```
// in the java.lang.Runnable file you will find the following interface
package java.lang;
public interface Runnable {
public void run(); // just like a pure virtual function in C++
}
```

Since we are defining an interface, run is implicitly a "abstract" method. As we have seen, interfaces in Java define a set of methods with NO implementation. Some class must "implement" the Runnable interface and provide an implementation of the run method, which is the method

Core

that is started by a thread. Java provides a "Thread" class, that implements the Runnable interface, but does not implement the run

```
public class ConcurrentReader implements Runnable {
...
public void run() {/* code here executes concurrently with caller */ }
...
}
```

To start this thread you need to first create an object of type Thread , bind it to a new Thread object, and then start it. Calling start creates the thread stack for the thread, and then invoked the run() method as the first procedure on that new thread stack.

```
ConcurrentReader readerThread = new ConcurrentReader();
Thread t = new Thread(readerThread); // create thread using a Runnable object
readerThread.start();
```

The java.lang.Thread class has a constructor that takes an object of type Runnable:

```
Thread(Runnable object); // must provide an object that implements run
```

Alternatively, we can define a subclass of the class Thread directly.

```
class ConcurrentWriter extends Thread {
public void run() {
//you provide the code here to run as a separate thread of control
}
}
```

To start this thread you just need to do the following:

```
ConcurrentWriter writerThread = new ConcuirrentWriter();
writerThread.start(); // start calls run()
```

java.lang.Thread

```
public class Thread implements Runnable {
private char name[];
private Runnable target;
...
public final static int MIN_PRIORITY = 1;
public final static int NORM_PRIORITY = 5;
public final static int MAX_PRIORITY = 10;
private void init(ThreadGroup g, Runnable target, String name) {...}
public Thread() {init(null, null, "Thread-" + nextThreadNum()); }
public Thread(Runnable target) {
init(null, target, "Thread-" + nextThreadNum());
}
public Thread(Runnable target, String name) {init(null, target, name); }
public synchronized native void start();
public void run() {
if (target != null) {
target.run();
}
}
...
}
```

java.lang.Thread

```
public class Thread implements Runnable {
...
public static native Thread currentThread();
public static native void yield();
Public static native void sleep(long millis) throws InterruptedException;
```

A File Has Many Names

A File object contains the file name string used to construct it. That string never changes throughout the lifetime of the object. A program can use the File object to obtain other versions of the file name, some of which may or may not be the same as the original file name string passed to the constructor.

Suppose a program creates a File object with the constructor invocation

File a = new File("xanadu.txt");

The program invokes a number of methods to obtain different versions of the file name. The program is then run both on a Microsoft Windows system (in directory c:\java\examples) and a Solaris system (in directory /home/cafe/java/examples). Here is what the methods would return:

Method Invoked	Returns on Microsoft Windows	Returns on Solaris
a.toString()	xanadu.txt	xanadu.txt
a.getName()	xanadu.txt	xanadu.txt
b.getParent()	NULL	NULL
a.getAbsolutePath()	c:\java\examples\xanadu.txt	/home/café/java/examples/xanadu.txt
a.getCanonicalPath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt

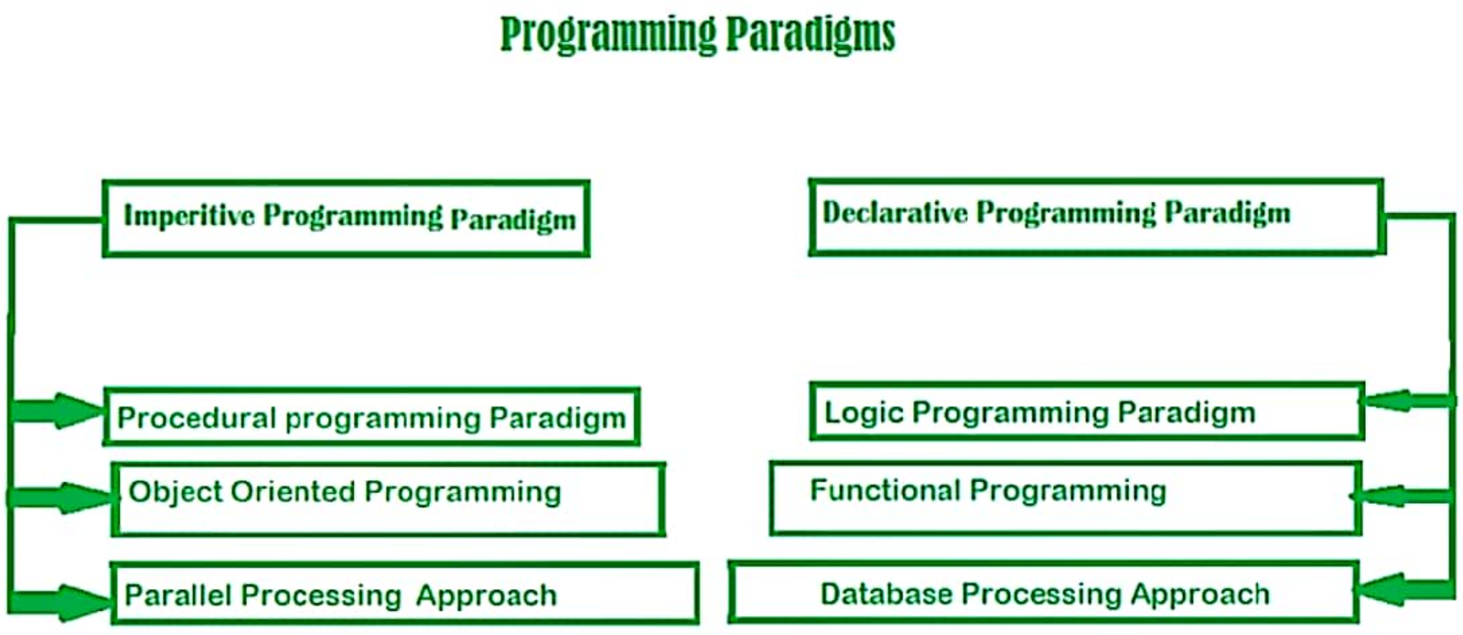
Then the same program constructs a File object from a more complicated file name, using File.separator to specify the file name in a platform-independent way.

File b = new File(".." + File.separator + "examples" + File.separator + "xanadu.txt");

Although b refers to the same file as a, the methods return slightly different values:

Introduction of Programming Paradigms

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfil each and every demand. They are discussed below:


Programming Paradigms
Imperitive Programming Paradigm
Declarative Programming Paradigm
Procedural programming Paradigm
Object Oriented Programming
Parallel Processing Approach
Logic Programming Paradigm
Functional Programming
Database Processing Approach

1. Imperative programming paradigm:

It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.

Advantage:

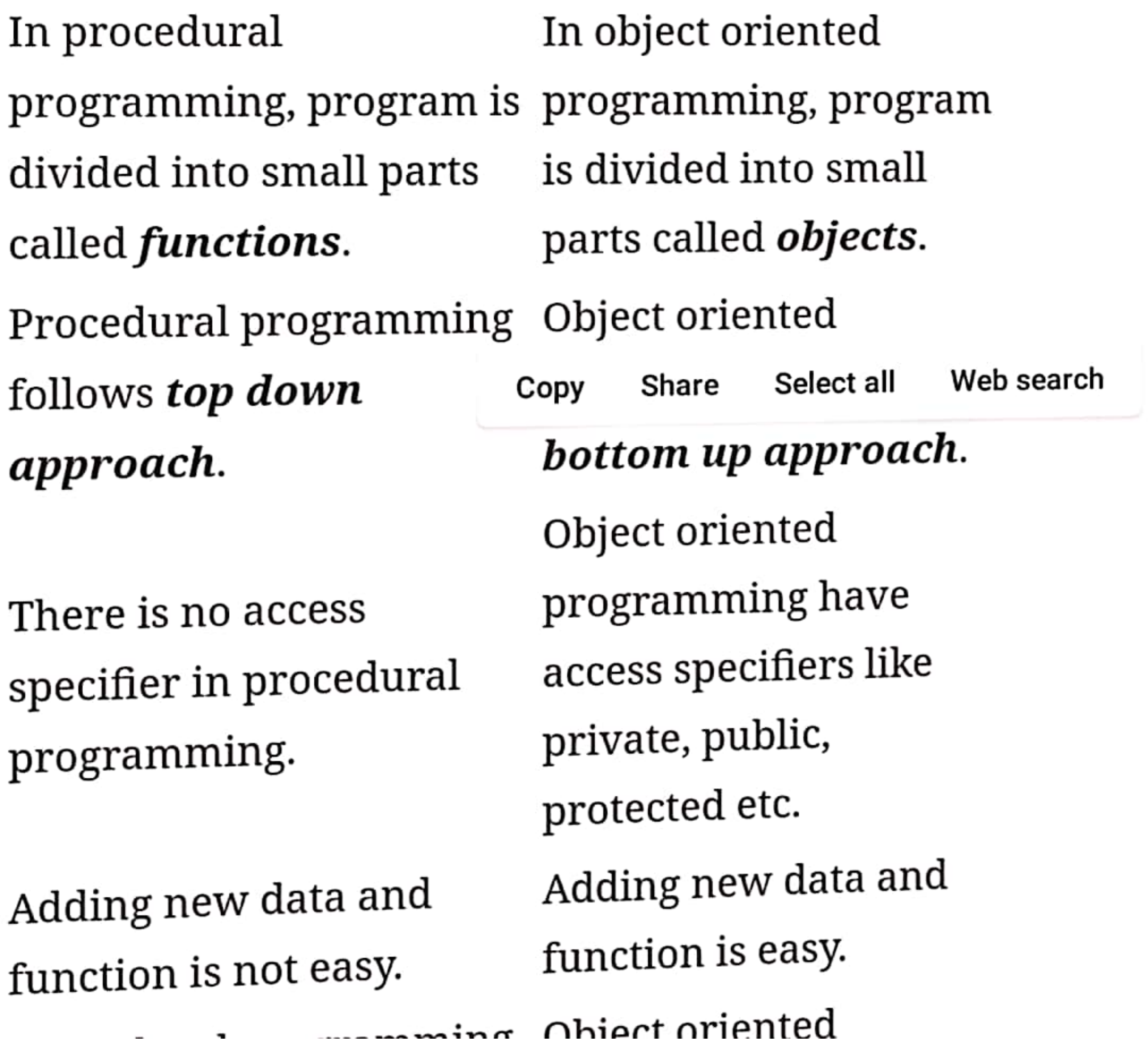
1. Very simple to implement
2. It contains loops, variables etc.

- **Procedural programming paradigm –**
  This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

1) Monolithic Programming Approach
2) Procedural Programming Approach
3) Structured Programming Approach
4) Object Oriented Programming Approach

Monolithic Programming Approach: In this approach, the program consists of sequence of statements that modify data. All the statements of the program are Global throughout the whole program. The program control is achieved through the use of jumps i.e. goto statements. In this approach, code is duplicated each time because there is no support for the function. Data is not fully protected as it can be accessed from any portion of the program. So this approach is useful for designing small and simple programs. The programming languages like ASSEMBLY and BASIC follow this approach.

Procedural Programming Approach: This approach is top down



In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented [illegible] ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
[illegible]	Object oriented

Benefits of OOP

- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them
- Secure, protects information through encapsulation

text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.

- **AI Expert System:** These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System
- **Office automation System:** These include formal as well as informal electronic systems that primarily concerned with information sharing and communication to and from people inside and outside the organization. OOP also help in making office automation principle.
- **Neural networking and parallel programming:** It addresses

[illegible] approximation of complex-time

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access (Refer this for details).
2. **class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Object

It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

Java

```
// Class Declaration

public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
                   int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }

    // method 3
    public int getAge()
    {
        return age;
    }

    // method 4
    public String getColor()
    {
        return color;
    }

    @Override
    public String toString()
    {
        return("Hi my name is "+ this.getName()+
               ".\nMy breed,age and color are "
               this.getBreed()+"," + this.getAge
               " "+ this.getColor());
```

```
            return("Hi my name is "+ this.getName()+
                   ".\nMy breed,age and color are "
                   this.getBreed()+"," + this.getAge
                   ","+ this.getColor());
        }

        public static void main(String[] args)
        {
            Dog tuffy = new Dog("tuffy","papillon",
            System.out.println(tuffy.toString());
        }
    }
```

Output:

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

- This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiate the constructors based on the number and the type of the arguments. The constructor in the *Dog* class takes four arguments. The following statement provides “tuffy”,”papillon”,5,”white” as values for those arguments:

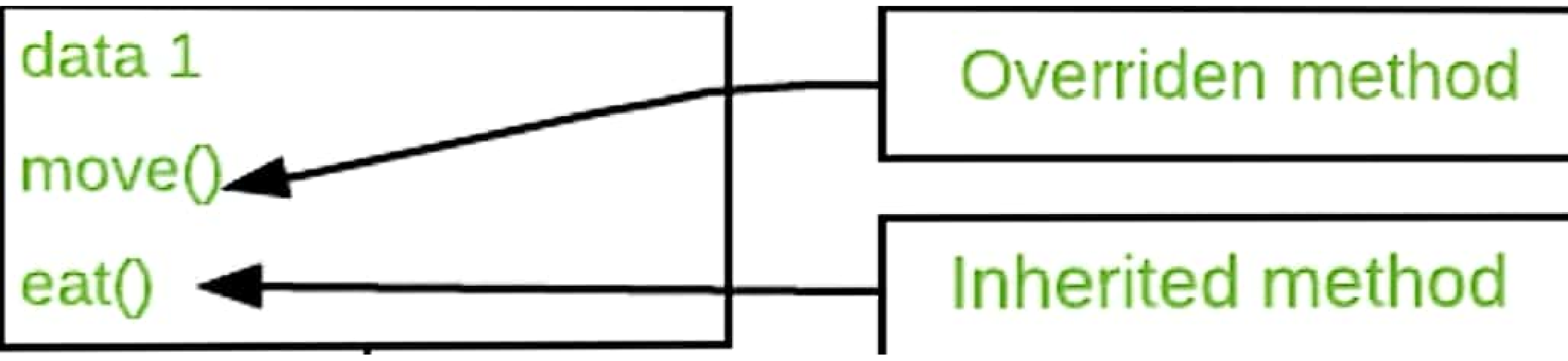
```
Dog tuffy = new Dog("tuffy","papillon",5, "white");
```

only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

```
// Java program to illustrate the concept of Abstra

abstract class Shape {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have a constructor
    public Shape(String color)
    {
        return Math.PI * Math.pow(
    }

    @Override
    public String toString()
    {
        return "Circle color is "
            + super.color
            + "and area is : "
            + area();
    }
}

class Rectangle extends Shape {

    double length;
    double width;

    public Rectangle(String color,
                     double length,
                     double width)
    {

        // calling Shape constructor
        super(color);
               println("Rectangle constructor c
```

Difference between Abstraction and Encapsulation:

Abstraction	Encapsulation
Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.


data 1
move()
eat()
Overriden method
Inherited method

```
    void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
```

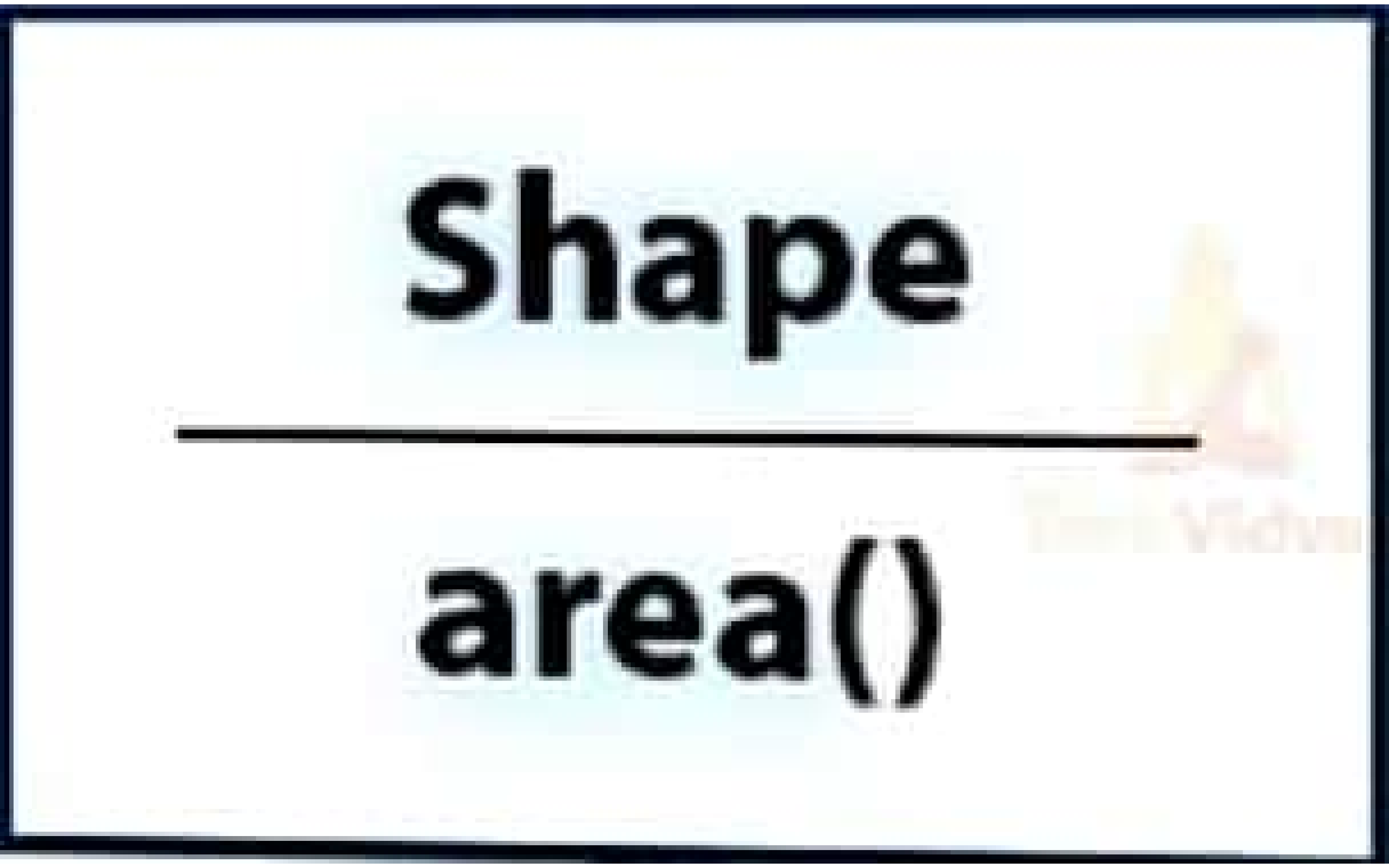
It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

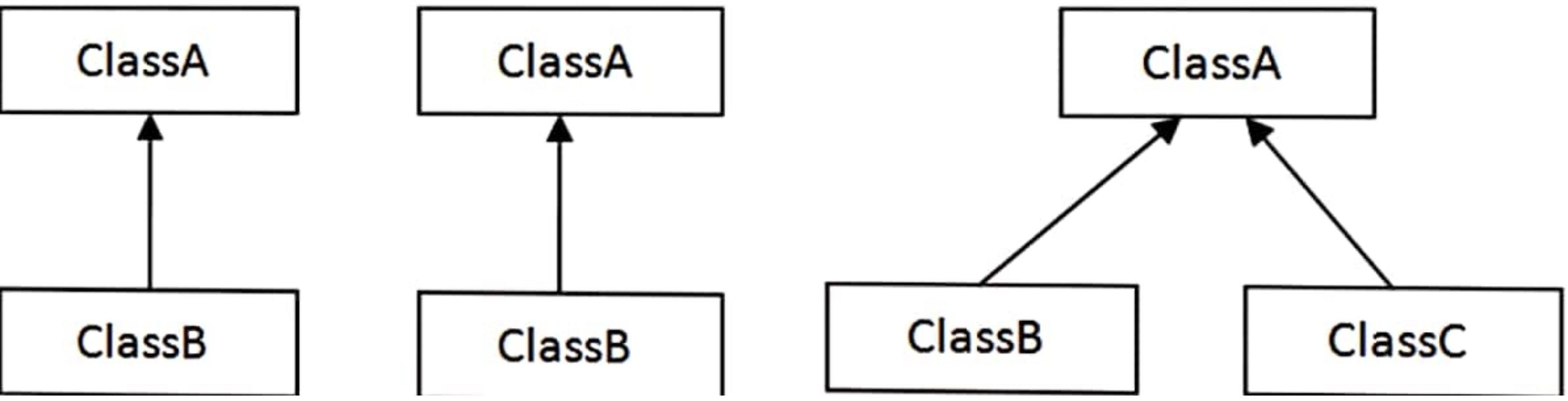
The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.


Shape
area()


ClassA
ClassB
ClassA
ClassB
ClassA
ClassB
ClassC

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetari
```

Now, the Deer class is considered to be polymorphic since this