Fundamentals of Object Oriented Programming

Dr. Ayesha Hakim

MODULE 1

Module No.	Unit No.	Contents	No of Hrs.	CO
1	**Fundamentals of Object oriented Programming**		10	CO 1
	1.1	Introduction, Basic Program Construction, Procedural Programming Approach, Structured Programming Approach, Modular Programming Approach, OOP Approach		
	1.2	Objects and classes, Data abstraction and Encapsulation, Inheritance and Polymorphism, Runtime polymorphism, Static and Dynamic Binding, Exceptions, Reuse, Coupling and Cohesion, Object Oriented Features of Java and C++.		
	1.3	**C++ Programming Basics:** Namespace Fundamentals, using, The standard Namespace, Data types, Input with cin, Output Using cout, Type Conversion: Automatic Conversions, Casts		
	1.4	Loops and Decision making statements, Functions, Function overloading		

A Survey of Programming Techniques

- Unstructured programming
- Procedural programming
- Modular programming
- Object-oriented programming

Unstructured programming

- Only one Program i.e. main Program
- All the sequences of commands or statements in one program called main Program
- E.g. Fortran, assembly language, old Basic

Unstructured programming. The main program directly operates on global data.

Structured Programming

- Also called as **Procedural** Programming
- For each task procedure is created
- The procedures are called from main

Figure 2.2: Execution of procedures. After processing flow of controls proceed where the call was made.

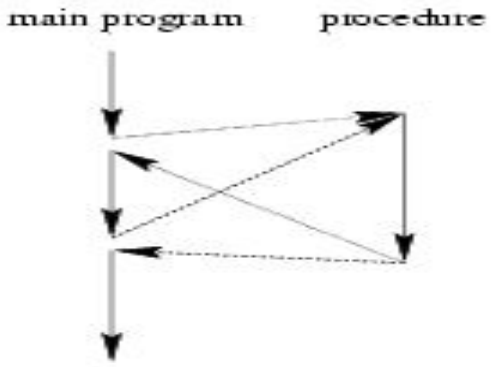
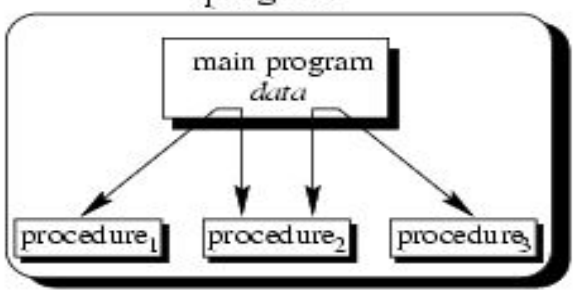


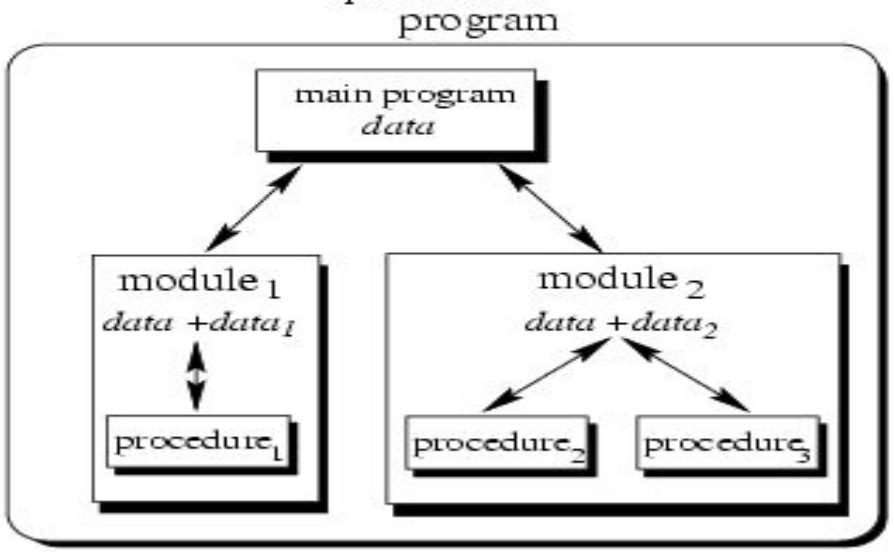
Figure 2.3: Procedural programming. The main program coordinates calls to procedures and hands over appropriate data as parameters.



Modular programming

- Procedures with some common functionality are grouped together into separate modules
- Program is categorized into several smaller modules
- Each module can have its own data

Figure 2.4: Modular programming. The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.



Object-oriented programming

- Works on objects which is considered smallest unit of the object oriented languages
- Focuses more on data rather than Procedures

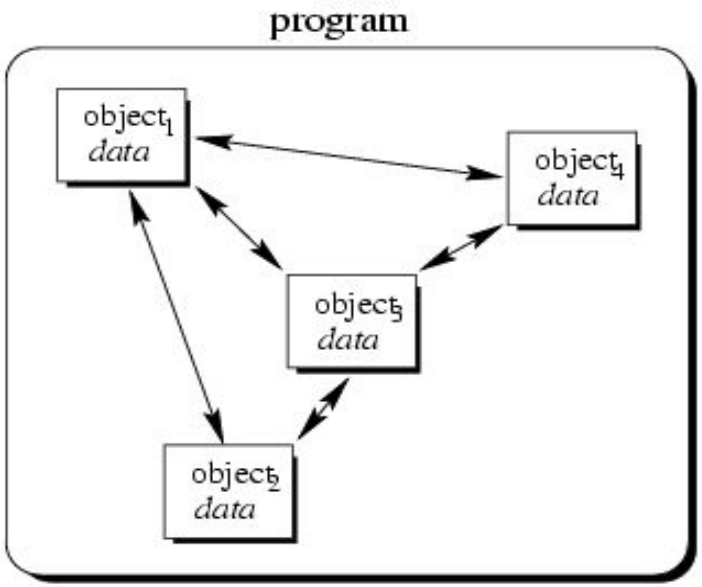


Figure 2.6: Object-oriented programming. Objects of the program interact by sending messages to each other.

Example

- **Unstructured Programming:**

```
#include
#include

void main()
{
   int a,b,c;
   clrscr();
   cout << "Enter the first number";
   cin >> a;
   cout << "Enter the second number";
   cin >> b;
   c=a+b;
   cout << "The sum is:" << c;
   getch();
}
```

EXAMPLE

- **Procedural Programming:**

```
#include
#include
int add(int,int);
void main()
{

   int a,b,c;
   clrscr();
   cout << "Enter the first number";
   cin >> a;
   cout << "Enter the second number";
   cin >> b;
   c=add(a,b);
   cout<<"The sum is:" << c;
   getch();
}
int add(int x,int y)
{
   int z=x+y;
   return z;
}
```

EXAMPLE

- **Object Oriented programming**

```
#include
#include

class Addition
{
  int a,b,c;
  public:
  void read()
  {
    cin >> a;
    cin >> b;
  }
  void add()
  {
    c=a+b;
  }
  void display()
  {
    cout << "The sum is:" << c;
  }
};
```

```
void main()
{
  Addition obj;   //object creation
  cout << "Enter the numbers";
  obj.read();
  obj.add();
  obj.display();
  getch();
}
```

Limitations of procedural programming

- Poor real world model.
- No importance to data.
- No privacy.
- No true reuse.
- Functions and data should be treated equally.

Procedural v/s object oriented programming

Feature	Procedure oriented Programming	Object oriented Programming
Divided Into	In POP Program is divided into small parts called **functions**	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**
Access Specifiers	POP does not have any access specifier	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Object oriented features

INTRODUCTION

- C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented programming.
- C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features
- Developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey

C++ Class Definition

- It is template, format or blueprint
- Also can be said as user defined data Type
- A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces.
- Class definition must be followed by a semicolon
- Class contains data members and member function

```
class Box{
        public:
        double length;   // Length of a box
   double breadth;  // Breadth of a box
   double height;   // Height of a box
     };
```

C++ Objects

- A class provides the blueprints for objects
- An object is created from a class
- Object is variable of class type
- Objects are declared the same way the variables are declared

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

- The public data members of objects of a class can be accessed using the direct member access operator (.)

```
#include <iostream>
using namespace std;
class Box{
public:
      double  length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
  int main( ){
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type
  Box   double volume = 0.0;     // Store the
  volume of a box here
  // box 1 specification   Box1.height = 5.0;
   Box1.length = 6.0;
  Box1.breadth = 7.0;
  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;   //
  volume of //box 1
  volume = Box1.height *
  Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " <<
  volume <<endl;    // volume of
  box 2
  volume = Box2.height *
  Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " <<
  volume <<endl;   return 0;}
```

Members of class

- A member function of a class is a function that has its definition or its prototype within the class
- Members are accessed using dot operator(.)

```
class Box
{
public:
  double length;          // Length of a box
  double breadth;         // Breadth of a box
      double height;          // Height of a box
  double getVolume(void);   // Returns box volume
  // member function
};
```

data member

Members of class(continued...)

- Member functions can be defined within the class definition or separately using **scope resolution operator, ::**

```
class Box{
   public:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
     double getVolume(void){
          return length * breadth * height;
     }
};// class end
```

Members of class(continued...)

- If you like you can define same function outside the class using **scope resolution operator**, :: as follows:

```
double Box::getVolume(void){
  return length * breadth * height;
}
```

Data encapsulation

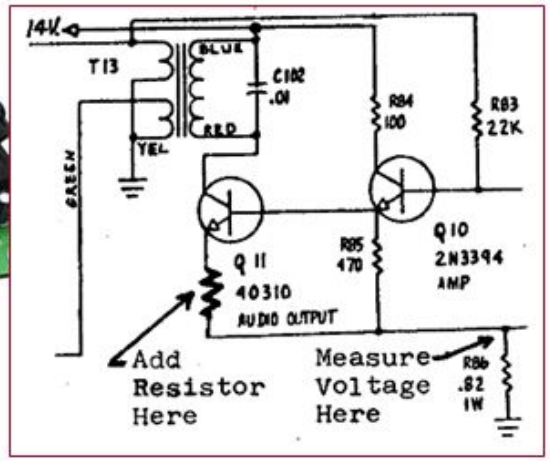
- The wrapping up of data and function into a single unit (called class) is known as encapsulation. Data and encapsulation is the most striking feature of a class.

- OOP encapsulates data (attributes) and functions (behavior) into packages called **objects**

- E.g. class encapsulates data members and member functions

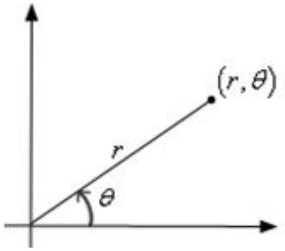
Encapsulation

- **encapsulation**: Hiding implementation details from clients.
  - Encapsulation enforces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



Benefits of encapsulation

- Abstraction between object and clients
- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates (r, ϑ) with the same methods.



- Can constrain objects' state (**invariants**)
  - Example: Only allow `Accounts` with non-negative balance.
  - Example: Only allow `Dates` with a month from 1-12.

Data abstraction

- Abstraction refers to the **act of representing essential features without including the background details or explanation**

- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes.

- E.g TV
  - a television separates its internal implementation from its external interface and we can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

Information hiding

- **Data hiding** is one of the important features of Object Oriented Programming which allows **preventing the functions of a program to access directly the internal representation of a class type**
- The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body
- private, and protected are called access specifiers/modifiers.
- A class can have multiple public, protected, or private labeled sections
- The default access for members and classes is private

Information hiding(continued)

```
class Base {

   public:   // public members go here

   protected:   // protected members go here

   private:   // private members go here

};
```

Information hiding(continued)

- A **public** member is accessible from anywhere outside the class but within a program

- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- By default all the members of a class would be private

- A **protected** member variable or function is very similar to a private member but it provides one additional benefit that they can be accessed in child classes which are called derived classes

INHERITANCE

- **Inheritance** is the process by which objects of one class acquired the properties of objects of another classes.
- In OOP, the concept of inheritance provides the idea of reusability
- Can add additional features to an existing class without modifying it

INHERITANCE

- **inheritance**: Forming new classes based on existing ones.

  - a way to share/**reuse code** between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
  - gets a copy of every field and method from superclass
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.

INHERITANCE SYNTAX

```
public class name extends superclass {
```

§Example:

```
public class Lawyer extends Employee {
  ...
}
```

- By extending `Employee`, each `Lawyer` object now:

§receives a copy of each method from `Employee` automatically

§can be treated as an `Employee` by client code

§

- Lawyer can also replace ("override") behavior from Employee.

POLYMORPHISM

- Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form.
- Ability for the same code to be used with different types of objects and behave differently with each.
- The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.
- Using a single function name to perform different type of task is known as **function overloading**.

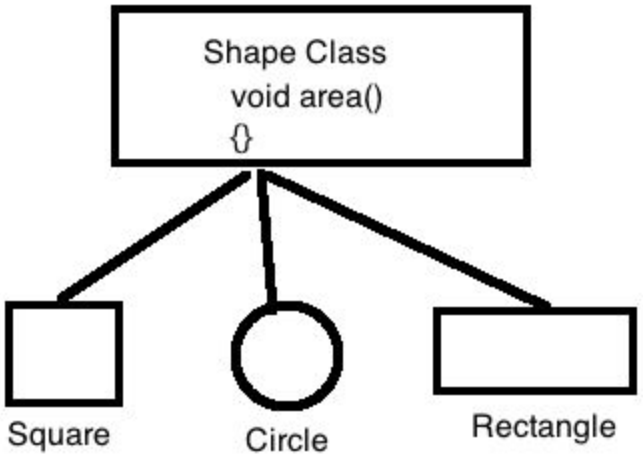


Fig . Polymorphism

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use **new().**

```
class Shape
{
    public void draw() {
    }
}
class Square extends Shape {
   public void draw() {      // <-- overridden method
   }
   .    // other methods or variables declaration
 }
class Circle extends Shape {
   public void draw() {      // <-- overridden method
   }
    .    // other methods or variables declaration
}
class Shapes {
   public static void main(String[] args) {
      Shape a = new Square();       // <-- upcasting Square to Shape
      Shape b = new Circle();       // <-- upcasting Circle to Shape
      a.draw();          // draw a square
      b.draw();          // draw a circle
   }
}
```

Static binding works during compile time and performs better.
Static binding can be applied using function overloading or operator overloading.

POLYMORPHIC PARAMETERS

- You can pass any subtype of a parameter's type.

```
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee e) {
    System.out.println("pay  : " + e.getSalary());
    System.out.println("vdays: " + e.getVacationDays());
    System.out.println("vform: " + e.getVacationForm());
    System.out.println();
}
```

OUTPUT:

```
pay  : 50000.0   pay  : 50000.0
vdays: 15        vdays: 10
vform: pink      vform: yellow
```

Runtime polymorphism in java

Runtime polymorphism, also known as the **Dynamic Method Dispatch**, is a process that **resolves a call to an overridden method at runtime.** The process involves the **use of the reference variable of a superclass** to call for an overridden method.

The method to be called will be determined based on the object being referred to by the reference variable.

Benefits of Runtime Polymorphism in Java

The primary advantage of runtime polymorphism is enabling the class to offer its specification to another inherited method. This implementation transfer from one method to another can be done without altering or changing the codes of the parent class object. Also, the call to an overridden method can be resolved dynamically during runtime.

What is Method Overriding?

Runtime polymorphism in Java can be carried out only by method overriding. **Method overriding** occurs when **objects possess the same name, arguments, and type as their parent class but have different functionality.** When a child class has such a method in it, it is called an overridden method.

What is Upcasting?

When an overridden method of child class is called through its parent type reference in Java, it is called **upcasting**. In this process, the object type represents the method or functionality that will be invoked. This decision is made during the **runtime**, and hence, the name run time polymorphism is given to the process.

Run time polymorphism is also known as **Dynamic Method Dispatch** as the **method functionality is decided dynamically at run time** based on the object.

It is also called “**Late Binding”** as the process of binding the method with the object occurs late after compilation.

Rules to be Followed When Executing Run Time Polymorphism

Below is a set of essential rules in run time polymorphism:

- Both the child and the parent class should have the **same method names.**
- Child and the parent class methods should have the **same parameter.**
- It is not possible to override the private methods of a parent class.
- It is not possible to override static methods.

EXAMPLE

```
class Shape{
void draw(){System.out.println("creating...");}
}
class square extends Shape{
void draw(){System.out.println("creating square...");}
}
class Triangle extends Shape{
void draw(){System.out.println("creating triangle...");}
}
class Pentagon extends Shape{
void draw(){System.out.println("creating pentagon...");}
}
```

```
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Square();
s.draw();
s=new Triangle();
s.draw();
s=new Pentagon();
s.draw();
}
}
```

Dynamic binding works during run time & is more flexible.

Reuse Concept

- Code reusability, an important feature of Object-Oriented Programming (OOP), is **enabled through inheritance, polymorphism, and information hiding.**
- Reusability in OOP this allows developers to **save time and effort** by not having to write the same code over again. Reusability also helps to **reduce the amount of code** that needs to be written and maintained, making programs more efficient and easier to maintain.

object oriented features of C++ and Java

COMPARISION	C	C++	JAVA
INHERITANCE	Not Supported in C.	Single, Mutli level & Multiple inheritance.	Single, Multilevel & Hierarchical inheritance. Multiple not Permitted.
POLYMORPHISM	Not Supported in C.	Method Overloading & Operator Overloading are allowed.	Method overloading is allowed but not operator overloading.
MULTI THREADING	No built in Support for Multithreading.	No built in Support for Multithreading.	Java provides built in support for Multithreading.
Header files	Supported. Header file has .h extension.<stdio.h>	Supported . <iostream>	Not Supported.
Compilation	Compiler	Compiler	Compiler & Interpreter.

Metrics	C	C++	Java
Programming Paradigm	Procedural language	Object-Oriented Programming (OOP)	Pure Object Oriented
Origin	Based on assembly language	Based on C language	Based on C and C++
Developer	Dennis Ritchie in 1972	Bjarne Stroustrup in 1979	James Gosling in 1991
Translator	Compiler only	Compiler only	Interpreted language (Compiler + interpreter)
Platform Dependency	Platform Dependent	Platform Dependent	Platform Independent
Code execution	Direct	Direct	Executed by JVM (Java Virtual Machine)
Approach	Top-down approach	Bottom-up approach	Bottom-up approach
File generation	.exe files	.exe files	.class files
Pre-processor directives	Support header files (#include, #define)	Supported (#header, #define)	Use Packages (import)
keywords	Support 32 keywords	Supports 63 keywords	50 defined keywords
Datatypes (union, structure)	Supported	Supported	Not supported
Inheritance	No inheritance	Supported	Supported except Multiple inheritance
Overloading	No overloading	Support Function overloading (Polymorphism)	Operator overloading is not supported
Pointers	Supported	Supported	Not supported
Allocation	Use malloc, calloc	Use new, delete	Garbage collector
Exception Handling	Not supported	Supported	Supported
Templates	Not supported	Supported	Not supported
Destructors	No constructor neither destructor	Supported	Not supported

S.N.	Parameters	C	C++	JAVA
1.	Language Type	procedure oriented programming	object oriented programming	object oriented programming
2.	Developer	Dennis Ritchie	Bjarne Stroustrup	James Gosling
3.	Language Level	Middle level language	High level language	High Level language
4.	Building Block	Function driven	Object driven	Object and Class driven
5.	Extensions	.c	.cpp	.java
6.	Platform	Dependent	Independent	Independent
7.	Comment Style	/* */	//single line, /* */ for multi line	same as C++
8.	Keywords	32	50	63
9.	Dynamic Variable	not support(int x = 5)	not support(int x = 5)	not support(int x = 5)
10.	Data Security	not secure	secure(less than java)	fully secured(hidden)

11.	Coding Difference (Print Hello)	`#include<stdio.h>`
		`int main()`
		`{`
		`printf("Hello");`
		`return 0;`
		`}`
		Output:- Hello
		Note:- C is mostly used to develop system software.
			`#include <iostream>`
			`using namespace std;`
			`int main()`
			`{`
			`cout << "Hello";`
			`return 0;`
			`}`
			Output:- Hello
			Note:- C++ is also secured and used to solve many real time of problem.
				`class HelloWorld {`
				`public static void main(String[] args) {`
				`System.out.println("Hello");`
				`}`
				`}`
				Output:- Hello
				Note:- Java is very secured and rubust language that's why it's very popular than others languages.

EXCEPTION HANDLING

- Exception is an event that occurs during the execution of a program and that disrupts the normal flow of instructions.

- Java exceptions are specialized events that indicate something bad has happened in the application, and the application either needs to recover or exit.

Why handle Java exceptions?

Java exception handling is important because it helps maintain the normal, desired flow of the program even when unexpected events occur.

If Java exceptions are not handled, programs may crash or requests may fail. This can be very frustrating for customers and if it happens repeatedly, you could lose those customers.

The worst situation is if your application crashes while the user is doing any important work, especially if their data is lost.

To make the user interface robust, it is important to handle Java exceptions to prevent the application from unexpectedly crashing and losing data. There can be many causes for a sudden crash of the system, such as incorrect or unexpected data input. For example, if we try to add two users with duplicate IDs to the database, we should throw an exception since the action would affect database integrity.

Tracking Exceptions in Java

Exceptions will happen in your application, no matter how well you write your program. You cannot predict the runtime environment entirely, especially that of your customer's.

The standard practice is to record all events in the application log file. The log file acts as a time machine helping the developer (or analyst) go back in time to view all the phases the application went through and what led to the Java exception.

For small-scale applications, going through a log file is easy. However, enterprise applications can serve thousands or even millions of requests per second. This makes manual analysis too cumbersome because errors and their causes can be lost in the noise.

This is where error monitoring software can help by grouping duplicates and providing summarized views of the top and most recent Java errors. They can also capture and organize contextual data in a way that's easier and faster than looking at logs.

How to handle Java exceptions?

This is accomplished using the keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

- You **try** to execute the statements contained within a block of code. *(A block of code is a group of one or more statements surrounded by braces.)*
- If you detect an exceptional condition within that block, you **throw** an exception object of a specific type.
- You **catch** and process the exception object using code that you have designed.
- You optionally execute a block of code, designated by **finally**, which needs to be executed whether or not an exception occurs. *(Code in the **finally** block is normally used to perform some type of cleanup.)*

How to handle exceptions in Java

The `try-catch` is the simplest method of handling exceptions. Put the code you want to run in the `try` block, and any Java exceptions that the code throws are caught by one or more `catch` blocks.

This method will catch any type of Java exceptions that get thrown.

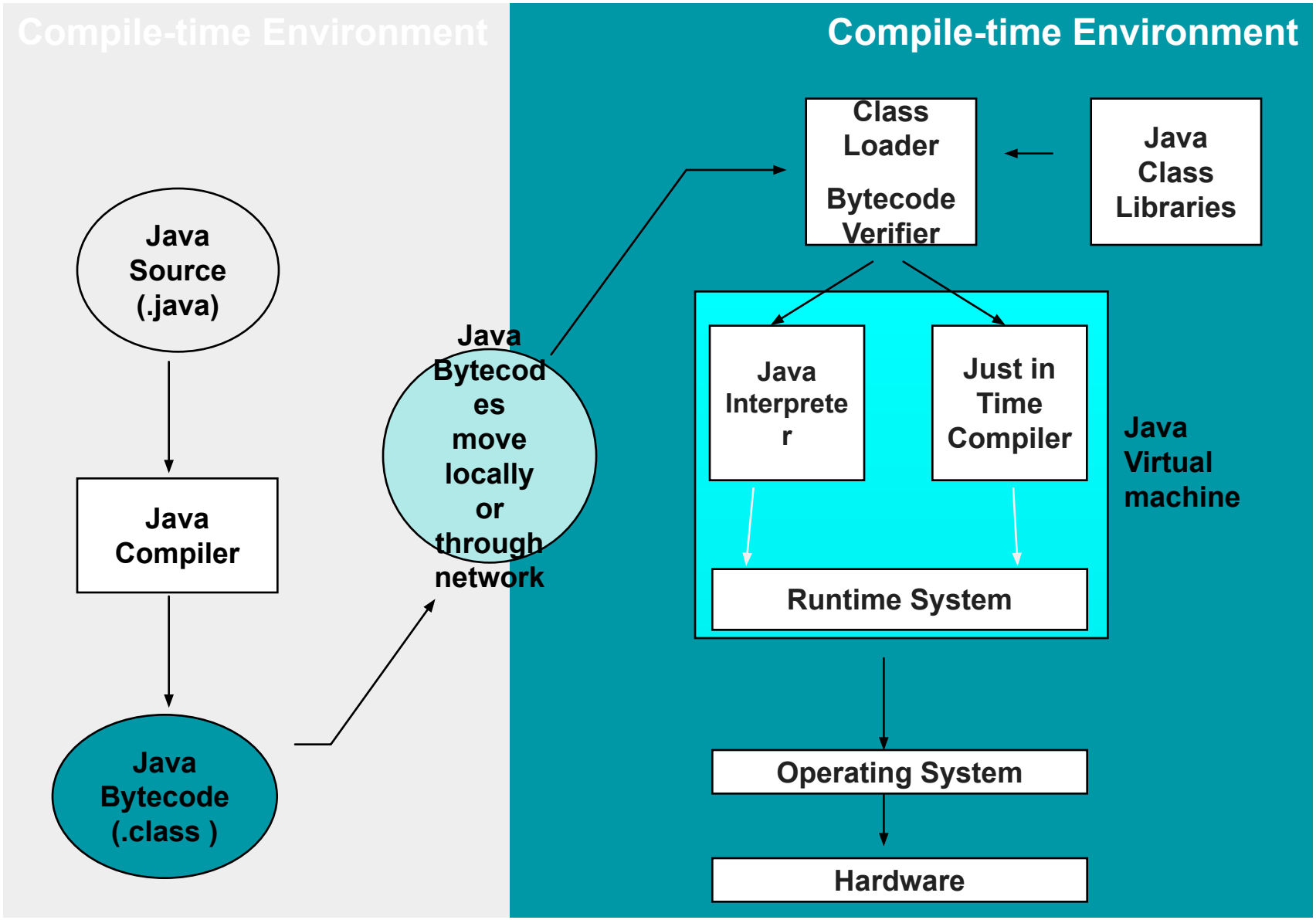
```
try {
  // block of code that can throw exceptions
} catch (Exception ex) {
  // Exception handler

  // Push the handled error into Rollbar
  rollbar.error(ex, "Hello, Rollbar");
}
```

Note: You can't use a `try` block alone. The try block should be immediately followed either by a `catch` or `finally` block.

Java - General

- Java is:
  - platform independent programming language
  - similar to C++ in syntax
- Java has some interesting features:
  - automatic type checking,
  - automatic garbage collection,
  - simplifies pointers; no directly accessible pointer to memory,
  - simplified network access,
  - multi-threading!

How it works…!


Compile-time Environment
Compile-time Environment
Class Loader
Bytecode Verifier
Java Class Libraries
Java Source (.java)
Java Compiler
Java Bytecode (.class)
Java Bytecodes move locally or through network
Java Interpreter
Just in Time Compiler
Java Virtual machine
Runtime System
Operating System
Hardware

How it works…!

- Java is independent only for one reason:
  - Only depends on the Java Virtual Machine (JVM),
  - code is compiled to *bytecode*, which is interpreted by the resident JVM,
  - JIT (just in time) compilers attempt to increase speed.

Java - Security

- Pointer denial - reduces chances of virulent programs corrupting host,
- Applets even more restricted -
  - May not
    - run local executables,
    - Read or write to local file system,
    - Communicate with any server other than the originating server.

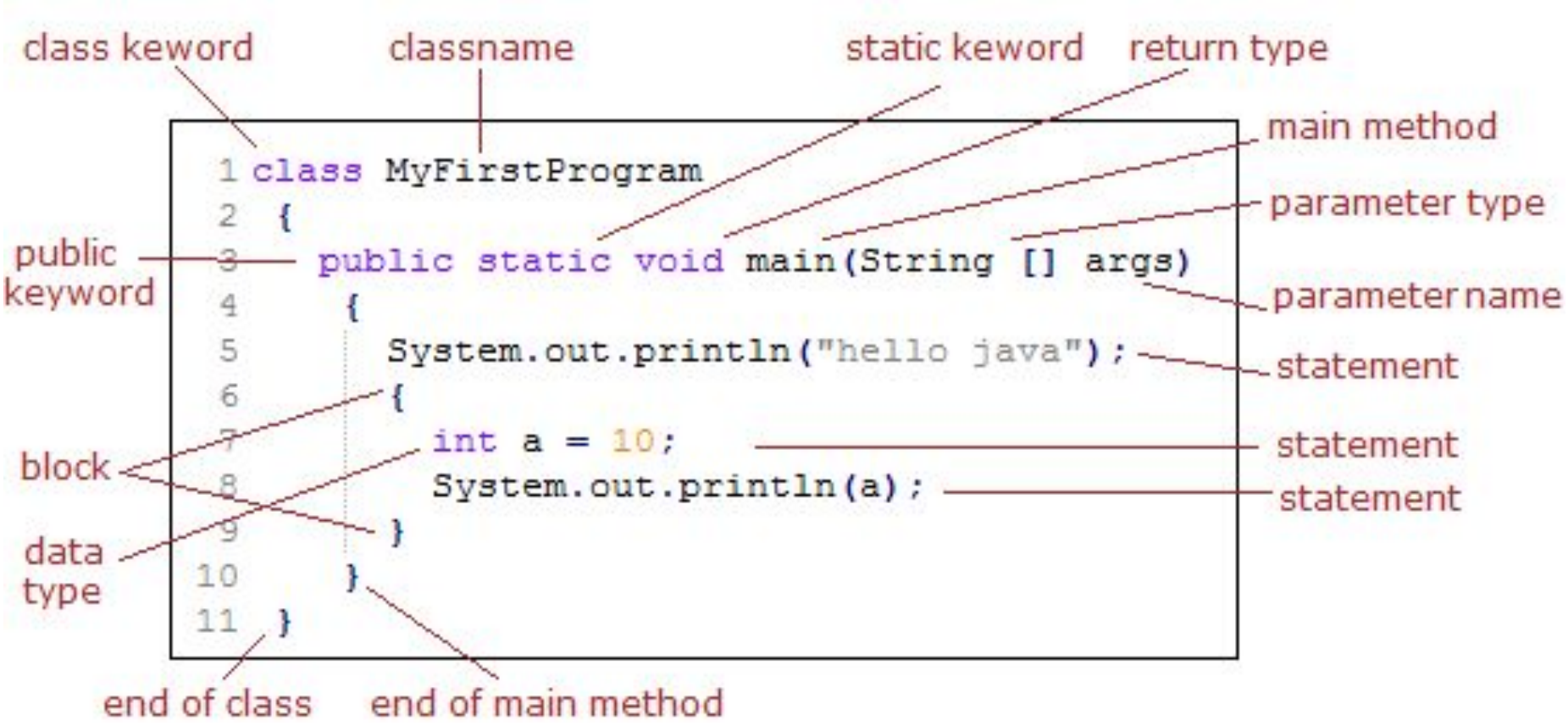
Object-Oriented

- Java supports:
  - Polymorphism
  - Inheritance
  - Encapsulation
- Java programs contain nothing but definitions and instantiations of classes
  - Everything is encapsulated in a class!

Java Advantages

- Portable - Write Once, Run Anywhere
- Security has been well thought through
- Robust memory management
- Designed for network programming
- Multi-threaded (multiple simultaneous tasks)
- Dynamic & extensible (loads of libraries)
  - Classes stored in separate files
  - Loaded only when needed

Basic Java Syntax



Primitive Types and Variables

- boolean, char, byte, short, int, long, float, double etc.
- These basic (or primitive) types are the only types that are not objects (due to performance issues).
- This means that you don't use the new operator to create a primitive variable.
- Declaring primitive variables:

```
float initVal;
int retVal, index = 2;
double gamma = 1.2, brightness
boolean valueOk = false;
```

Initialisation

- If no value is assigned prior to use, then the compiler will give an error
- Java sets primitive variables to zero or false in the case of a boolean variable
- All object references are initially set to null
- An array of anything is an object
  - Set to null on declaration
  - Elements to zero false or null on creation

Declarations

```
int index = 1.2;          // compiler error
boolean retOk = 1;        // compiler error
double fiveFourths = 5 / 4;   // no error!
float ratio = 5.8f;       // correct
double fiveFourths = 5.0 / 4.0; // correct
```

- 1.2f is a float value accurate to 7 decimal places.
- 1.2 is a double value accurate to 15 decimal places.

Assignment

- All Java assignments are right associative

```
int a = 1, b = 2, c = 5
a = b = c
System.out.print(
“a= “ + a + “b= “ + b + “c= “ + c)
```

- What is the value of a, b & c
- Done right to left: a = (b = c);

Basic Mathematical Operators

- * / % + - are the mathematical operators
- * / % have a higher precedence than + or -

```
double myVal = a + b % d - c * d / b;
```

- Is the same as:

```
double myVal = (a + (b % d)) -
               ((c * d) / b);
```

Statements & Blocks

- A simple statement is a command terminated by a semi-colon:

  ```
  name = “Fred”;
  ```

- A block is a compound statement enclosed in curly brackets:

  ```
  {
   name1 = “Fred”; name2 = “Bill”;
  }
  ```

- Blocks may contain other blocks

Flow of Control

- Java executes one statement after the other in the order they are written
- Many Java statements are flow control statements:

Alternation: if, if else, switch

Looping: for, while, do while

Escapes: break, continue, return

If – The Conditional Statement

- The if statement evaluates an expression and if that evaluation is true then the specified action is taken

```
if ( x < 10 ) x = 10;
```

- If the value of x is less than 10, make x equal to 10
- It could have been written:

```
if ( x < 10 )
x = 10;
```

- Or, alternatively:

```
if ( x < 10 ) { x = 10; }
```

Relational Operators

==	Equal (careful)
!=	Not equal
>=	Greater than or equal
<=	Less than or equal
>	Greater than
<	Less than

If… else

- The if … else statement evaluates an expression and performs one action if that evaluation is true or a different action if it is false.

```
if (x != oldx) {
  System.out.print(“x was changed”);
}
else {
  System.out.print(“x is unchanged”);
}
```

Nested if … else

```
if ( myVal > 100 ) {
  if ( remainderOn == true) {
     myVal = mVal % 100;
  }
  else {
   myVal = myVal / 100.0;
  }
}
else
{
  System.out.print("myVal is in range");
}
```

else if

- Useful for choosing between alternatives:

```
if ( n == 1 ) {
  // execute code block #1
}
else if ( j == 2 ) {
  // execute code block #2
}
else {
  // if all previous tests have failed, execute
  code block #3
}
```

A Warning…

WRONG!

```
if( i == j )
     if ( j == k )
      System.out.print(
         "i equals k");
     else
      System.out.print(
      "i is not equal
      to j");
```

CORRECT!

```
if( i == j ) {
  if ( j == k )
  System.out.print(
       "i equals k");
}
else
  System.out.print("i
  is not equal to
  j");// Correct!
```

The switch Statement

```
switch ( n ) {
  case 1:
   // execute code block #1
   break;
  case 2:
   // execute code block #2
   break;
   default:
   // if all previous tests fail then
   //execute code block #4
   break;
}
```

The for loop

- Loop n times

```
for ( i = 0; i < n; n++ ) {
  // this code body will execute n times
  // i from  0 to n-1
}
```

- Nested for:

```
for ( j = 0; j < 10; j++ ) {
  for ( i = 0; i < 20; i++ ){
   // this code body will execute 200 times
  }
}
```

while loop

```
while(response == 1) {
  System.out.print( "ID =" +
  userID[n]);
  n++;
  response = readInt( "Enter ");
}
```

What is the minimum number of times the loop is executed?

What is the maximum number of times?

do {… } while loops

```
do {
  System.out.print( “ID =” + userID[n] );
  n++;
  response = readInt( “Enter ” );
}while (response == 1);
```

What is the minimum number of times the loop is executed?

What is the maximum number of times?

Break

- A break statement causes an exit from the innermost containing while, do, for or switch statement.

```
for ( int i = 0; i < maxID, i++ ) {
  if ( userID[i] == targetID ) {
   index = i;
   break;
  }
} // program jumps here after break
```

Continue

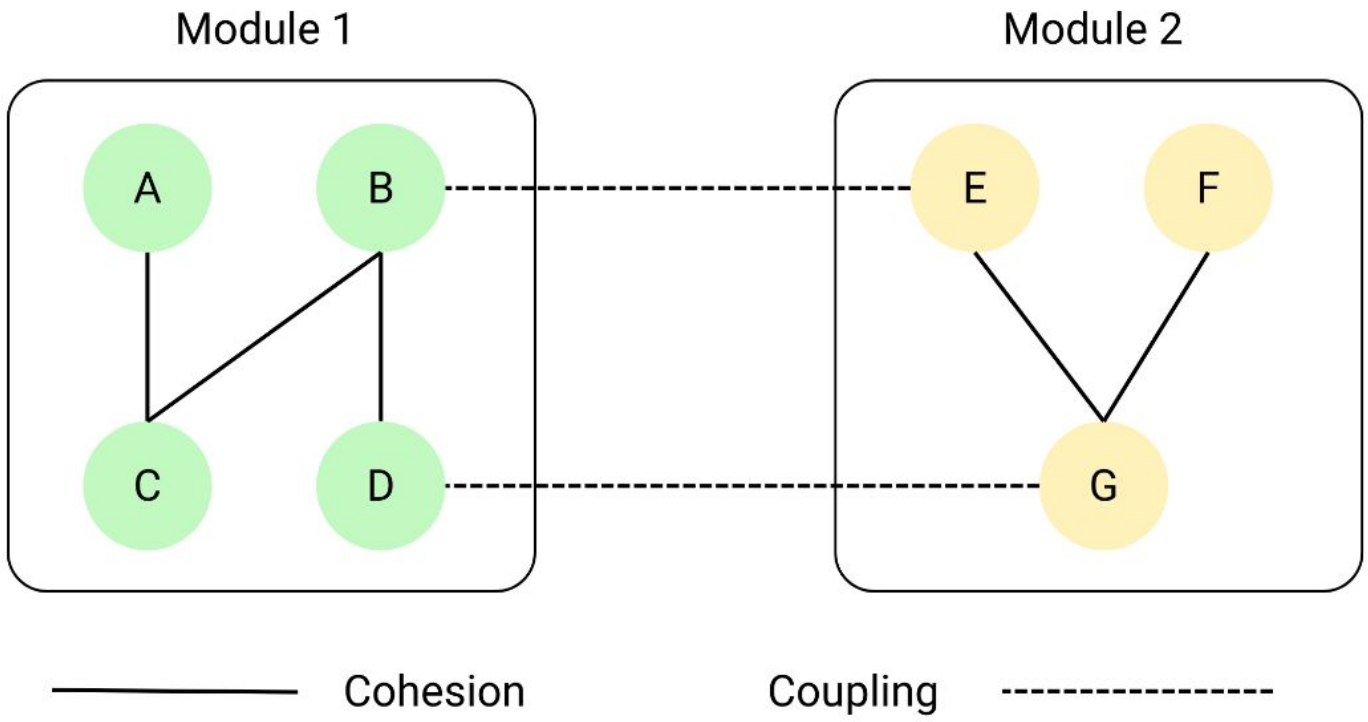
- Can only be used with while, do or for.
- The continue statement causes the innermost loop to start the next iteration immediately

```
for ( int i = 0; i < maxID; i++ ) {
  if ( userID[i] != -1 ) continue;
  System.out.print( "UserID " + i + " :" +
  userID);
}
```

Cohesion and Coupling in Object Oriented Programming (OOPS)

As software developers, we may have heard of the concepts of Cohesion and Coupling in Object-Oriented Programming. These two concepts are used to measure our code quality and ensure it is maintainable and scalable. Let's understand the meaning of both terms.

- **Cohesion** defines how effectively elements within a module or object work together to fulfil a single well-defined purpose. A high level of cohesion means that elements within a module are tightly integrated and working together to achieve the desired functionality.
- **Coupling** defines the degree of interdependence between software modules. Tight coupling means that modules are closely connected and changes in one module can affect others. On the other hand, loose coupling means that modules are independent and changes in one module have minimal impact on others.



So, it's best practice in OOPS to achieve loose coupling and high cohesion because it helps us to create classes or modules that are more flexible and less prone to break when changes are made.

What is the meaning of Low Cohesion?

In low cohesion, a module or object within a system has multiple responsibilities that are not closely related i.e. a single module or object performs a diverse range of independent tasks. As a result:

- This makes our code difficult to understand.
- This can lead to confusion and make it challenging to modify or update the code.

For example, let's consider a class "StudentRecord" that has following responsibilities: 1) Maintaining student information 2) Calculating student grades 3) Printing student transcripts 4) Sending email notifications to students.

```java
public class StudentRecord {
    private String name;
    private String id;
    private String address;
    private double[] grades;

    public StudentRecord(String name, String id, String address, double[] grades) {
        this.name = name;
        this.id = id;
        this.address = address;
        this.grades = grades;
    }

    public double calculateAverageGrade() {
        double sum = 0;
        for (double grade : grades) {
            sum += grade;
        }
        return sum / grades.length;
    }

    public void printTranscript() {
        System.out.println("Student Transcript");
        System.out.println("Name: " + name);
        System.out.println("ID: " + id);
        System.out.println("Address: " + address);
        System.out.println("Average Grade: " + calculateAverageGrade());
    }

    public void sendEmailNotification(String emailAddress) {
        // code to send email notification
    }
}
```

In this case, StudentRecord class has multiple unrelated responsibilities like maintaining student information, calculating grades, printing transcripts, and sending emails. These all are distinct tasks that could be handled by separate classes or modules. So this code becomes more difficult to understand and maintain. How? Let's think!

- Suppose we want to add a new field to the student data. This may affect or break the code that calculates average grade or sends email notifications or prints the transcript. If there are a lot of other unrelated responsibilities, then we need to do a lot of work to identify breaking changes. Idea is simple: If a class has multiple reasons to change then it has multiple reasons to break as well.
- If we want to use StudentRecord class in a different context, we would need to copy entire class, including all of its responsibilities.

How can make this code more cohesive? Let's solve this problem by understanding the idea of high cohesion.

What is the meaning of High Cohesion?

In OOPS, high cohesion is an idea where each module or component within a system has a single, well-defined responsibility. In other words, each module or component is highly focused on a specific task and has all the necessary information and resources to perform that task effectively.

So to make the code more cohesive, we can use the idea of Single Responsibility Principle, which states that a class should have only one reason to change. In other words, we should refactor the code so that each responsibility is encapsulated in a separate class.

Based on this idea, let's solve the problem of low cohesion in previous example. We can divide the responsibilities of the original "StudentRecord" class into four separate classes: Student, GradeCalculator, TranscriptPrinter, and EmailNotifier.

```java
public class Student {
    private String name;
    private String id;
    private String address;
    private double[] grades;

    public Student(String name, String id, String address, double[] grades) {
        this.name = name;
        this.id = id;
        this.address = address;
        this.grades = grades;
    }

    public String getName() {
        return name;
    }

    public String getId() {
        return id;
    }

    public String getAddress() {
        return address;
    }

    public double[] getGrades() {
        return grades;
    }
}
```

```
public class GradeCalculator {
    public static double calculateAverage(Student student) {
        double sum = 0;
        double[] grades = student.getGrades();
        for (double grade : grades) {
            sum += grade;
        }
        return sum / grades.length;
    }
}

public class TranscriptPrinter {
    public static void print(Student student) {
        double[] grades = student.getGrades();
        System.out.println("Student Transcript");
        System.out.println("Name: " + student.getName());
        System.out.println("ID: " + student.getId());
        System.out.println("Address: " + student.getAddress());
        System.out.println("Average Grade: " + GradeCalculator.calculateAverage(student));
    }
}

public class EmailNotifier {
    public static void sendNotification(String emailAddress, Student student) {
        // code to send email notification
    }
}
```

Now, this separation of concerns makes the code easier to understand, reuse and maintain, as each class has a well-defined purpose and responsibility. This also helps us to find and fix bugs easily, and understand how different components work together.

What is the meaning of Tight coupling?

In OOPS, tight coupling is a situation when classes or modules have many dependencies on each other.

- Tight coupling makes it difficult to modify or extend our modules without affecting the other.
- This can lead to an increased risk of bugs and unintended side effects.

Here is an example of tight coupling in Java:

```
class Car {
    private Engine engine;

    public Car() {
        this.engine = new Engine();
    }

    public void start() {
        engine.start();
    }

    public void stop() {
        engine.stop();
    }
}

class Engine {
    public void start() {
        // Some implementation
    }

    public void stop() {
        // Some implementation
    }
}
```

In this example, Car class has a direct reference to the Engine class, and it also calls start() and stop() methods on it. This creates a tight coupling because if we want to modify or extend Engine class, we also need to consider how it will affect Car class. For example:

- If we change the method signatures of start or stop methods in Engine class, we also need to change corresponding method calls in Car class.
- Suppose, we want to add specific implementations of Engine like ElectricEngine or a HybridEngine, then things get more complex further!

How can make this code less coupled? Let's solve this problem by understanding the idea of loose coupling.

What is the meaning of Loose Coupling?

In OOPS, loose coupling is a situation when classes or modules have minimal dependencies on each other i.e. changes in one class or module are unlikely to affect the other.

- This makes our code modular and reduces the risk of introducing bugs.
- This makes our code easier to maintain and extend over time.

To avoid tight coupling, we can use design patterns like Dependency Injection or use interfaces to decouple dependencies between classes.

For example, to solve the problem of tight coupling in above code, we can add the Engine interface and create a ConcreteEngine class that implements the Engine interface. Here Engine interface defines methods that Car class needs to use, but it doesn't specify exactly how these methods are implemented.

```
interface Engine {
    void start();
    void stop();
}

class ConcreteEngine implements Engine {
    public void start() {
        // Some implementation
    }

    public void stop() {
        // Some implementation
    }
}

class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }

    public void stop() {
        engine.stop();
    }
}
```

This will help us to create multiple concrete implementations of the Engine interface, each with its own unique behavior. For example, we can also create additional implementations of the Engine interface, such as ElectricEngine class or a HybridEngine class.

- Now Car class depends on the Engine interface, not on a specific implementation of the Engine. When we create an instance of the Car class, we pass in an instance of the Engine interface. This can be an instance of any implementation like ConcreteEngine, ElectricEngine, or HybridEngine.
- This also helps us to create different Car instances with different types of engines, without having to modify the Car class itself.

By decoupling Car and Engine classes, we can make each class easier to modify or extend without affecting the other. For example, if we want to add new methods to the Engine interface, we can do so without having to modify the Car class. If we want to change the behavior of the ConcreteEngine class, we can do so without having to modify the Car class.

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

Best practices to achieve High Cohesion in OOPS

- Each class should have a single responsibility and all its methods and data should be related to that responsibility.
- We should encapsulate data and methods that belong together within the same class. This makes the class easier to maintain and understand.
- We should aim to design system as a collection of small, reusable modules, each with its own high cohesion. This makes it easier to identify and fix any problems, as well as to extend and reuse the code.
- We should identify and use proper design patterns to maintain high cohesion.

How to Check Cohesion?

To check the cohesion of a class that you have created, you can follow these steps:

1. Identify the responsibilities of the class. What is the main purpose of the class, and what tasks does it perform?
2. Analyze the methods in the class. Do they all contribute to the same purpose, or do they perform unrelated tasks?
3. Check if the methods in the class use the same instance variables or share common functionality. Do they rely on each other to complete their tasks?
4. Determine if there is any duplicated code, or if methods perform similar functions but with different parameters.
5. Finally, evaluate if the class has a clear and single responsibility.

If the methods in the class are closely related to the same responsibility or purpose, use the same instance variables, share common functionality, and there is no duplicated code or irrelevant methods, then the class has high cohesion.

If the methods in the class are unrelated to each other, have different instance variables, and do not share common functionality, then the class has low cohesion.

Best practices to achieve Loose Coupling in OOPS

- By defining interfaces, we can ensure that objects communicate with each other in a well-defined manner, rather than having direct references to concrete implementations.
- By using dependency injection, we can create objects with necessary dependencies they require to function, rather than creating objects with their own dependencies.
- We can use design patterns, such as facade or adapter pattern to provide a unified interface for interacting with multiple objects. This helps to further decouple objects and make the system more modular.
- We should use encapsulation to limits the visibility of data and methods, and avoids exposing implementation details. By doing so, we reduce number of dependencies between objects and make it easier to modify or extend the code without affecting other parts of the system.

How to Check Coupling?

For coupling, it is important to consider the level of interdependence between two modules. To check coupling in an OOP application, you can try the following:

1. Look at the classes used by a particular class. If the class is dependent on many other classes, this may indicate a high degree of coupling.
2. Count the number of method calls made between classes. A large number of method calls can indicate a high degree of coupling.

Advantages of High Cohesion in OOPS

Let's take an example of a house. Each room in the house has a specific purpose and all furniture, fixtures, and decorations in that room are related to that purpose. This makes it easy for us to understand what each room is for and what we can find in those rooms. It also makes it easier for us to maintain and fix things if something breaks, as we know exactly what belongs in that room.

Similarly, when we write code with high cohesion, each class or module is like a room in the house. It has a single, well-defined responsibility, and all its methods and data are related to that responsibility. This makes it easy for other developers to understand what the class is for, what it does, and how it works. It also makes it easier for them to maintain and fix the code if they need to, as they can easily identify the source of any problems.

Advantages of Loose Coupling in OOPS

Let's again take the example of building a house. Just like a software system, a house is made up of different parts that need to work together. When these parts are connected in a loosely coupled way, it's easier to make changes or additions to one part without affecting the others.

For example, if we want to add a new room to our house, we don't want to tear down the entire house. Similarly, in software, if we want to add a new feature to our code, we don't want to rewrite the entire system. So loose coupling allows us to make changes to one part of the code without affecting the rest, just like adding a room to our house.

Finally, low coupling also reduces the risk of unintended side effects. If each room in a house is connected in a clear and well-defined way, it's easier to understand how they all work together. In software, low coupling makes it easier to understand the relationships between objects, reducing the risk of bugs and making it easier to maintain over time.

Conclusion

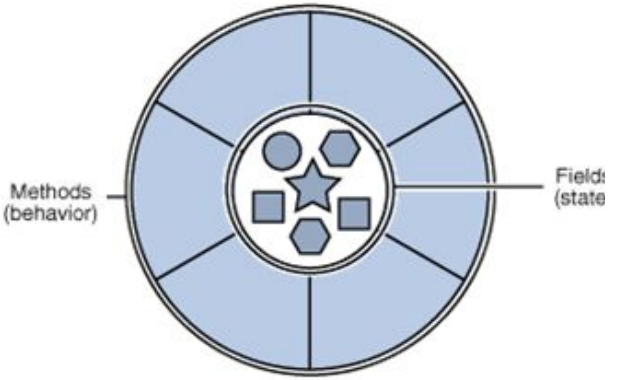
Cohesion and coupling are interdependent concepts in Object-Oriented Programming (OOP). The level of one can impact the level of the other. High cohesion is often accompanied by loose coupling. When the elements within a module are tightly related to each other and serve a single, well-defined purpose, they will have limited interaction and dependence on other modules. This results in a loose coupling between the module and other modules.

On the other hand, tight coupling can be an indicator of low cohesion. When elements from two modules are heavily dependent on each other, it suggests that the elements are spread across both modules and lack a clear, well-defined purpose. This results in low cohesion between the modules.

So in a well-designed OOP system, we should aim to achieve high cohesion and low coupling. This will help us to enhance the overall maintainability, scalability, and readability of the code. Enjoy learning, Enjoy OOPS!

Objects

- **object:** An entity that encapsulates data and behavior.
  - *data*: variables inside the object
  - *behavior*: methods inside the object
    - You interact with the methods; the data is hidden in the object.



- Constructing (creating) an object:

```
Type objectName = new Type(parameters);
```

- Calling an object's method:

```
objectName.methodName(parameters);
```

Classes

- **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

– **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

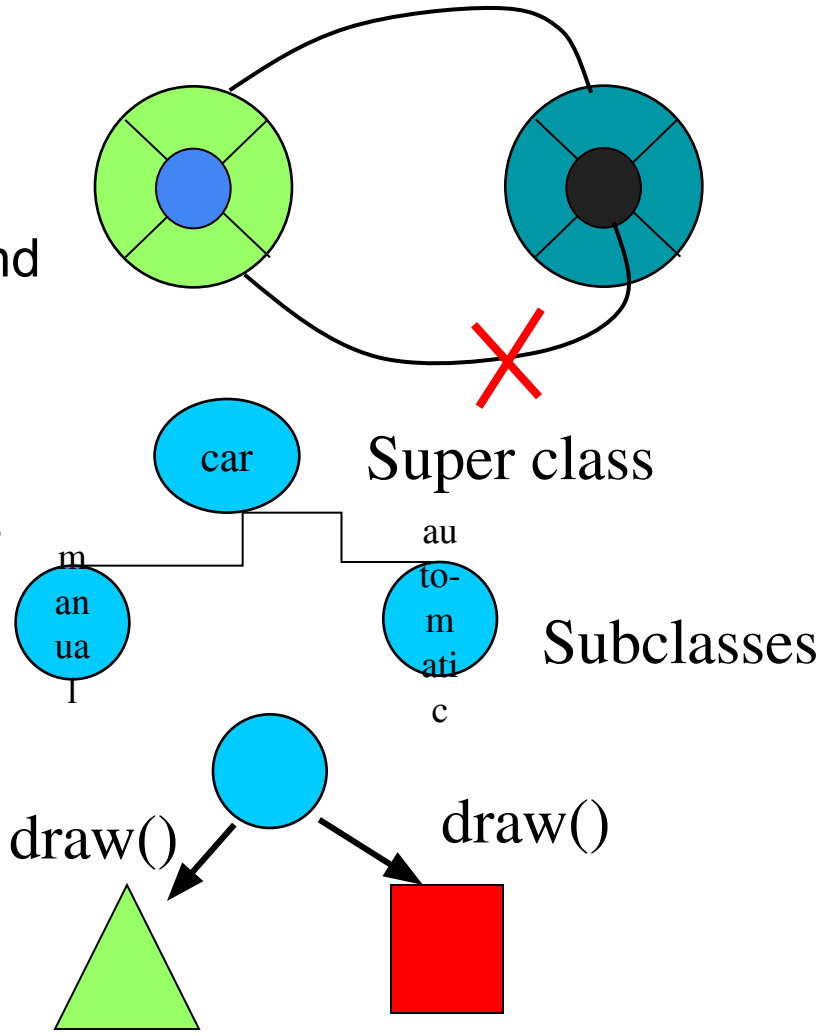
  – **abstraction**: Separation between concepts and details.
  Objects and classes provide abstraction in programming.

Classes ARE Object Definitions

- OOP - object oriented programming
- code built from objects
- Java these are called ***classes***
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name

The three principles of OOP

- Encapsulation
  - Objects hide their functions (**methods**) and data (**instance variables**)
- Inheritance
  - Each **subclass** inherits all variables of its **superclass**
- Polymorphism
  - Interface same despite different data types



Simple Class and Method

```
Class Fruit{
    int grams;
    int cals_per_gram;

    int total_calories() {
     return(grams*cals_per_gram);
    }
}
```

Methods

- A method is a named sequence of code that can be invoked by other Java code.
- A method takes some parameters, performs some computations and then optionally returns a value (or object).
- Methods can be used as part of an expression statement.

```
public float convertCelsius(float tempC) {
   return( ((tempC * 9.0f) / 5.0f) + 32.0 );
  }
```

Method Signatures

- A method signature specifies:
  - The name of the method.
  - The type and name of each parameter.
  - The type of the value (or object) returned by the method.
  - The checked exceptions thrown by the method.
  - Various method modifiers.
  - *modifiers type name (parameter list) [throws exceptions]*

```
public float convertCelsius (float tCelsius ) {}
public boolean setUserInfo ( int i, int j, String name ) throws
    IndexOutOfBoundsException {}
```

Public/private

- Methods/data may be declared ***public*** or ***private*** meaning they may or may not be accessed by code in other classes …
- Good practice:
  - keep data private
  - keep most methods private
- well-defined interface between classes - helps to eliminate errors

Using objects

- Here, code in one class creates an instance of another class and does something with it …

```
Fruit plum=new Fruit();
int cals;
cals = plum.total_calories();
```

- ***Dot operator*** allows you to access (public) data/methods inside Fruit class

Java Development Kit (JDK)

- javac - The Java Compiler
- java - The Java Interpreter
- jdb - The Java Debugger
- appletviewer -Tool to run the applets

- javap - to print the Java bytecodes
- javaprof - Java profiler
- javadoc - documentation generator
- javah - creates C header files


JDK
java, javac, jdb, appletviewer, javah, javaw
jar, rmi.....
JRE
Class Loader, Byte Code Verifier
Java API, Runtime Libraries
JVM
Java Interpreter
JIT
Garbage Collector
Thread Sync.....

Some Salient Characteristics of Java

- Java is ***platform independent:*** the same program can run on any correctly implemented Java system
- Java is ***object-oriented:***
  - Structured in terms of ***classes***, which group data with operations on that data
  - Can construct new classes by ***extending*** existing ones
- Java designed as
  - A ***core language***
  - A rich collection of ***commonly available packages***
- Java can be embedded in Web pages

Java Processing and Execution

- Begin with Java ***source code*** in text files: **`Model.java`**
- A Java source code compiler produces Java ***byte code***
  - Outputs one file per class: **`Model.class`**
  - May be standalone or part of an IDE
- A ***Java Virtual Machine*** loads and executes class files
  - May compile them to native code (e.g., x86) internally

Compiling and Executing a Java Program

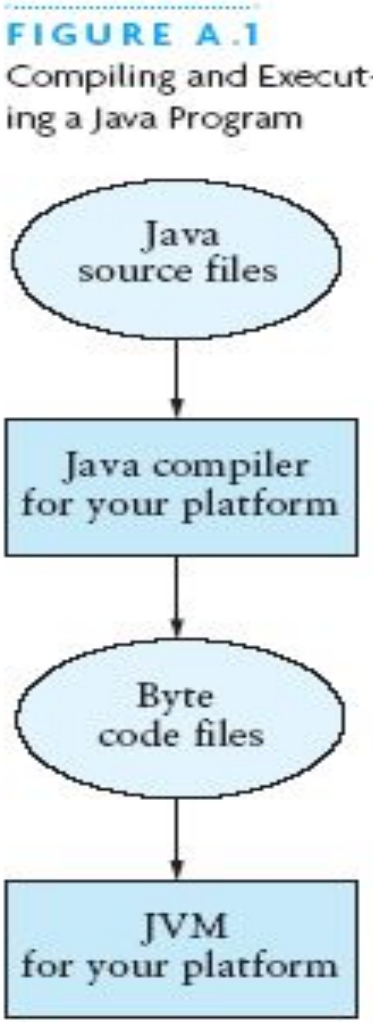


FIGURE A.1
Compiling and Executing a Java Program

Classes and Objects

- The ***class*** is the unit of programming
- A Java program is a ***collection of classes***
  - Each class definition (usually) in its own **`.java`** file
  - *The file name must match the class name*
- A class describes ***objects (instances)***
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics
- These characteristics are:
  - ***Data fields*** for each object
  - ***Methods*** (operations) that do work on the objects

Grouping Classes: The Java API

- API = *Application Programming Interface*
- Java = small core + extensive collection of packages
- A ***package*** consists of some related Java classes:
  - Swing: a GUI (graphical user interface) package
  - AWT: Application Window Toolkit (more GUI)
  - util: utility data structures
- The ***import*** statement tells the compiler to make available classes and methods of another package
- A ***main*** method indicates where to begin executing a class (if it is designed to be run as a program)

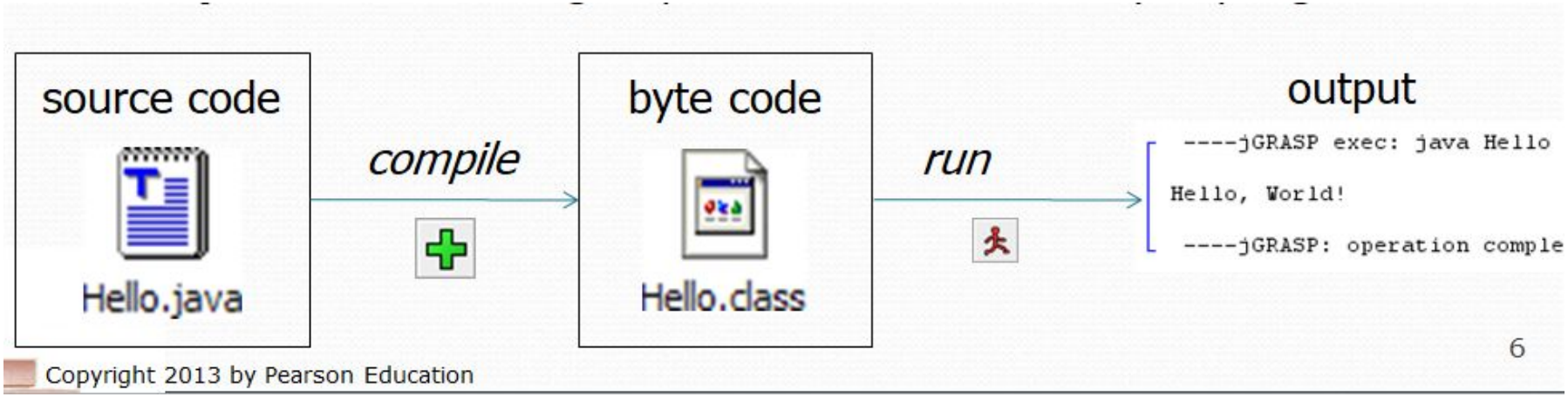
A Little Example of **`import`** and **`main`**

```
import javax.swing.*;
    // all classes from javax.swing
public class HelloWorld {  // starts a class
  public static void main (String[] args) {
  // starts a main method
  // in: array of String; out: none (void)
  }
}
```

- **`public`** = can be seen from any package
- **`static`** = not “part of” an object

Processing and Running `HelloWorld`

- **`javac HelloWorld.java`**
  - Produces **`HelloWorld.class`** (byte code)
- **`java HelloWorld`**
  - Starts the JVM and runs the **`main`** method



Names & Identifiers

- You must give your program a name.

```
public class GangstaRap {
```

- Naming convention: capitalize each word (e.g. `MyClassName`)
- Your program's file must match exactly (`GangstaRap.java`)
  - includes capitalization (Java is "case-sensitive")

Keywords in Java

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

Syntax

- **syntax**: The set of legal structures and commands that can be used in a particular language.
  - Every basic Java statement ends with a semicolon `;`
  - The contents of a class or method occur between `{` and `}`

- **syntax error** (**compiler error**): A problem in the structure of a program that causes the compiler to fail.

  Examples:
  - Missing semicolon
  - Too many or too few `{ }` braces
  - Illegal identifier for class name
  - Class and file names do not match
  - ...

Syntax error example

```
1  public class Hello {
2      pooblic static void main(String[] args) {
3          System.owt.println("Hello, world!")_
4      }
5  }
```

- Compiler output:

```
Hello.java:2: <identifier> expected
    pooblic static void main(String[] args) {
           ^
Hello.java:3: ';' expected
}
^
2 errors
```

  - The compiler shows the line number where it found the error.
  - The error messages can be tough to understand!

References and Primitive Data Types

- Java distinguishes two kinds of entities
  - Primitive types
  - Objects
- Primitive-type data is stored in primitive-type variables
- Reference variables store the *address of* an object

Primitive Data Types

- Represent numbers, characters, boolean values
- Integers: byte, short, int, and long
- Real numbers: float and double
- Characters: char

Primitive Data Types

Data type	Range of values
`byte`	-128 .. 127 (8 bits)
`short`	-32,768 .. 32,767 (16 bits)
`int`	-2,147,483,648 .. 2,147,483,647 (32 bits)
`long`	-9,223,372,036,854,775,808 .. ... (64 bits)
`float`	$+/-10^{-38}$ to $+/-10^{+38}$ and 0, about 6 digits precision
`double`	$+/-10^{-308}$ to $+/-10^{+308}$ and 0, about 15 digits precision
`char`	Unicode characters (generally 16 bits per char)
`boolean`	True or false

Operators

1. subscript `[ ]`, call `( )`, member access `.`
2. pre/post-increment `++ --`, boolean complement `!`, bitwise complement `~`, unary `+ -`, type cast `(type)`, object creation `new`
3. `* / %`
4. binary `+ -` (`+` also concatenates strings)
5. signed shift `<< >>`, unsigned shift `>>>`
6. comparison `< <= > >=`, class test `instanceof`
7. equality comparison `== !=`
8. bitwise and `&`
9. bitwise or `|`

Operators

11. logical (sequential) and **`&&`**
12. logical (sequential) or **`||`**
13. conditional **`cond ? true-expr : false-expr`**
14. assignment **`=`**, compound assignment **`+= -= *= /= <<= >>= >>>= &= |=`**

Java Control Statements

- A group of statements executed in order is written
  - `{ stmt1; stmt2; ...; stmtN; }`
- The statements execute in the order 1, 2, ..., N
- Control statements alter this sequential flow of execution

Java Control Statements (continued)

TABLE A.4
Java Control Statements

Control Structure	Purpose	Syntax

`if ... else`	Used to write a decision with *conditions* that select the alternative to be executed. Executes the first (second) alternative if the *condition* is true (false).	`if (condition) {`
		`  ...`
		`} else {`
		`  ...`
		`}`

`switch`	Used to write a decision with scalar values (integers, characters) that select the alternative to be executed. Executes the *statements* following the *label* that is the *selector* value. Execution falls through to the next `case` if there is no `return` or `break`. Executes the statements following `default` if the *selector* value does not match any *label*.	`switch (selector) {`
		`  case label : statements; break;`
		`  case label : statements; break;`
		`  ...`
		`  default : statements;`
		`}`

`while`	Used to write a loop that specifies the repetition *condition* in the loop header. The *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited.	`while (condition) {`
		`  ...`
		`}`

`for`	Used to write a loop that specifies the *initialization*, repetition *condition*, and *update* steps in the loop header. The *initialization* statements execute before loop repetition begins; the *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited. The *update* statements execute after each iteration.	`for (initialization; condition; update) {`
		`  ...`
		`}`

Java Control Statements (continued)

TABLE A.4 (continued)

Control Structure	Purpose	Syntax

`do ... while`	Used to write a loop that specifies the repetition *condition* after the loop body. The *condition* is tested after each iteration of the loop and, if it is true, the loop body is repeated; otherwise, the loop is exited. The loop body always executes at least one time.	`do {`
		`  ...`
		`while (`*condition*`) ;`

Methods

- A Java method defines a group of statements as performing a particular operation
- **`static`** indicates a ***static*** or ***class*** method
- All method arguments are ***call-by-value***
  - Primitive type: *value* is passed to the method
  - Method may modify local copy ***but*** will not affect caller's value
  - Object reference: *address of object* is passed
  - Change to reference variable does not affect caller
  - ***But*** operations can affect the object, visible to caller

The Class `Math`

TABLE A.5
Class `Math` Methods

Method	Behavior
`static` *numeric* `abs(`*numeric*`)`	Returns the absolute value of its *numeric* argument (the result type is the same as the argument type).
`static double ceil(double)`	Returns the smallest whole number that is not less than its argument.
`static double cos(double)`	Returns the trigonometric cosine of its argument (an angle in radians).
`static double exp(double)`	Returns the exponential number *e* (i.e., 2.718 ...) raised to the power of its argument.
`static double floor(double)`	Returns the largest whole number that is not greater than its argument.
`static double log(double)`	Returns the natural logarithm of its argument.
`static` *numeric* `max(`*numeric*`,` *numeric*`)`	Returns the larger of its *numeric* arguments (the result type is the same as the argument types).
`static` *numeric* `min(`*numeric*`,` *numeric*`)`	Returns the smaller of its *numeric* arguments (the result type is the same as the argument type).
`static double pow(double, double)`	Returns the value of the first argument raised to the power of the second argument.
`static double random()`	Returns a random number greater than or equal to 0.0 and less than 1.0.
`static double rint(double)`	Returns the closest whole number to its argument.
`static long round(double)`	Returns the closest `long` to its argument.
`static int round(float)`	Returns the closest `int` to its argument.
`static double sin(double)`	Returns the trigonometric sine of its argument (an angle in radians).
`static double sqrt(double)`	Returns the square root of its argument.
`static double tan(double)`	Returns the trigonometric tangent of its argument (an angle in radians).
`static double toDegrees(double)`	Converts its argument (in radians) to degrees.
`static double toRadians(double)`	Converts its argument (in degrees) to radians.

Escape Sequences

- An escape sequence is a sequence of two characters beginning with the character \
- A way to represents special characters/symbols

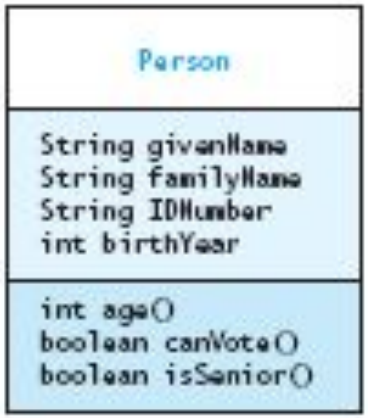
TABLE A.6
Escape Sequences

Sequence	Meaning
`\n`	Start a new output line
`\t`	Tab character
`\\`	Backslash character
`\"`	Double quote
`\'`	Single quote or apostrophe
`\u`*dddd*	The Unicode character whose code is *dddd* where each digit *d* is a hexadecimal digit in the range `0` to `F` (`0`–`9`, `A`–`F`)

Defining Your Own Classes

- *Unified Modeling Language* (UML) is a standard diagram notation for describing a class

FIGURE A.6
Class Diagram for Person

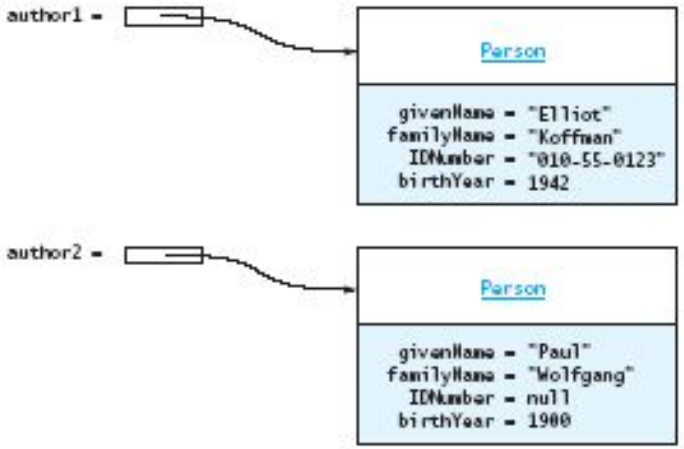


Field *signatures*: type and name

Method *signatures*: name, argument types, result type

Class name

FIGURE A.7
Object Diagrams of Two Instances of Class Person



Field *values*

Class name

Defining Your Own Classes (continued)

- The modifier **`private`** limits access to just this class
- Only class members with **`public`** visibility can be accessed outside of the class* (* but see **`protected`**)

TABLE A.11
Default Values for Data Fields

Data Field Type	Default Value
`int` (or other integer type)	0
`double` (or other real type)	0.0
`boolean`	`false`
`char`	`\u0000` (the smallest Unicode character: the null character)
Any reference type	`null`

Input/Output using Streams

- An **`InputStream`** is a sequence of characters representing program input data
- An **`OutputStream`** is a sequence of characters representing program output
- The console keyboard stream is **`System.in`**
- The console window is associated with **`System.out`**

Summary

- A Java program is a collection of classes
- The JVM approach enables a Java program written on one machine to execute on any other machine that has a JVM
- Java defines a set of primitive data types that are used to represent numbers, characters, and boolean data
- The control structures of Java are similar to those found in other languages
- The Java **`String`** and **`StringBuffer`** classes are used to reference objects that store character strings

Summary (continued)

- Be sure to use methods such as **`equals`** and **`compareTo`** to compare the *contents* of **`String`** objects
- You can declare your own Java classes and create objects of these classes using the **`new`** operator
- A class has data fields and instance methods
- Class **`JOptionPane`** can be used to display dialog windows for data entry and message windows for output
- The stream classes in package **`java.io`** read strings from the console and display strings to the console, and also support file I/O


The End!