OBJECT ORIENTED PROGRAMMING (JAVA) (CST8284)

PROF. SANDRA IROAKAZI

WEEK 2

A Review of Object Oriented Programming (OOP) Concepts

Week 1 Learning Outcomes

- Discuss Object Oriented technology
- Revise **concepts** of object oriented programming paradigm
- Review **Java** as an object oriented programming language
- Describe a typical Java development **environment**

Review of Programming Paradigms

The process of **designing** and **implementing** computer programs is known as **computer programming.**

- **Procedural** programming
  - composed of set of procedures and code for each procedure
- **Structured** programming
  - Easier to understand, correct and modify

Review of Programming Paradigms (2)

- **Object Oriented Programming *(OOP)***
  - Paradigm in which real-world objects are viewed as separate **entities** having their own **state** that can be changed only using built in **procedures** that are called **methods**
- **Some Benefits**
  - Easy code reuse
  - Extension (inheritance)
  - Encapsulation and information hiding

Review of Programming Paradigms (3)

- A **major** difference between object-oriented programming paradigm and other paradigms:
  - Provides the possibility for extension (inheritance). And the reuse of code

Object-Oriented Analysis and Design (OOAD)

How will you **create** the code (program instructions) for your programs (*product*)?

- Requirements analysis: determine your project's requirements (**what** the system should do)
- Design: develop a design that satisfies the requirements (**how** the system should do it)
- Review the **design carefully** (you and other software professionals) before writing any code

Object-Oriented Analysis and Design (OOAD) (2)

- Analyzing and designing your system from an object-oriented point of view is called an **object-oriented-analysis-and-design (OOAD) process**
- Languages like **Java** are object-oriented
- **Object-oriented programming (OOP)** allows you to implement an object-oriented design as a working system

Objects

- Objects are essentially **reusable** software components

- **For example:** date objects, time objects, audio objects, video objects, automobile objects, people objects, etc.

Objects (2)

- Almost any **noun** can be represented as a software object in terms of:
  - **Attributes** (example name, color and size)
  - **Behaviors** (example eating, making sounds)

Attributes: color, name, size

Behaviors: eating, sounds

https://www.vectorstock.com/royalty-free-vector/different-faces-of-wild-animals-vector-18132970

Main Concepts of Object-Oriented Programming

- **Abstraction** – Process of **hiding** some details from the user and showing just the essential information

- **Encapsulation -** The process of **binding** together the data (variables) and code that acts on the data **(methods)** as a single unit.

  - Variables of a class will be **hidden** from other classes, and can be accessed only through the **methods** of their current class.

Main Concepts (2)

- **Inheritance -** the process of class acquiring the properties (methods and fields) of another.
  - The inheriting **class** is referred to as a **child class** or **subclass**, whereas the class providing the properties to be **inherited** is referred to as the **parent class** (**superclass)**

- **Polymorphism**
  - Poly – **many**, morphism – **forms**
  - Ability for a data or message to be processed in more than one form

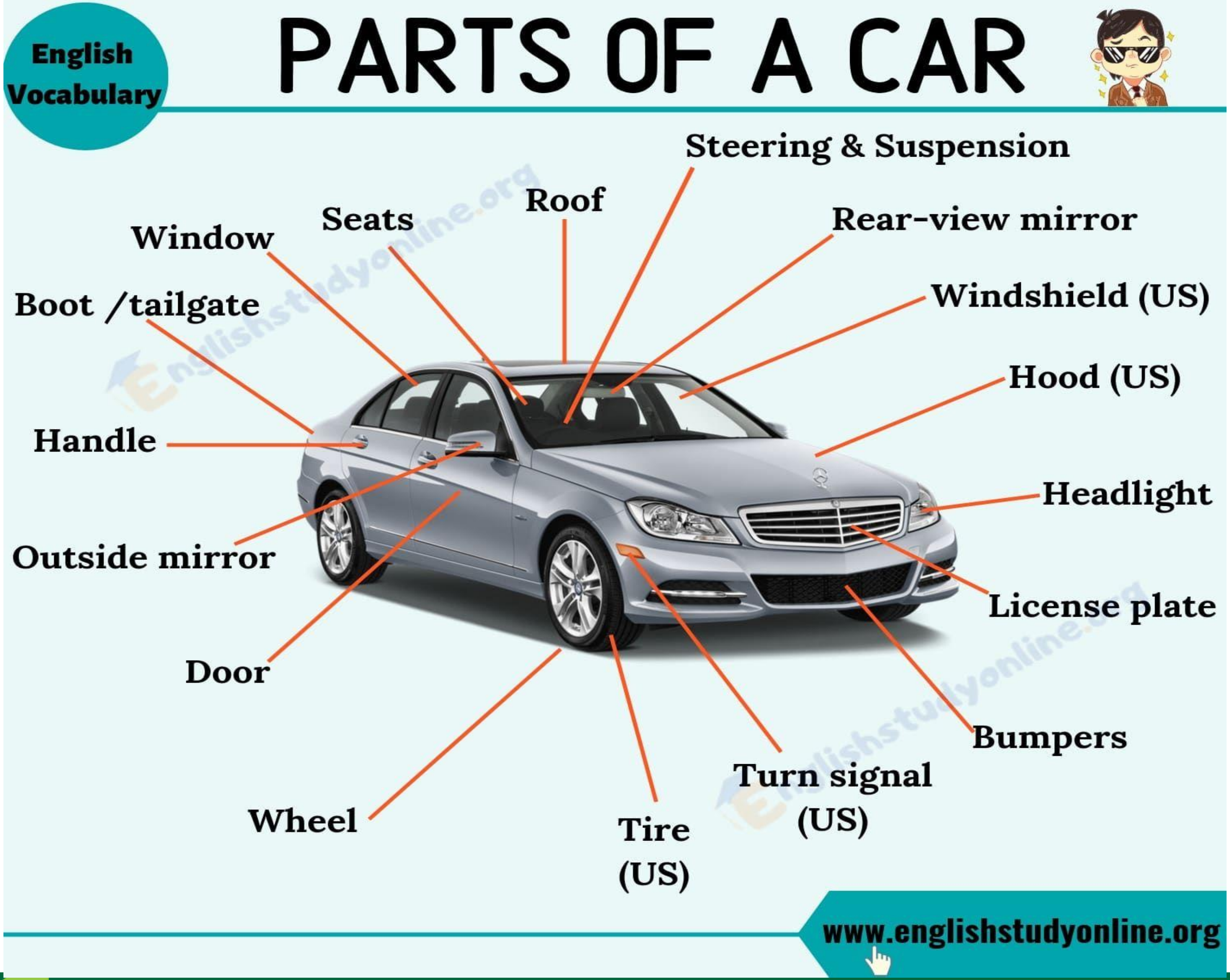
Object (2) - An Automobile Example

- **Automobile as an Object**
  - Suppose you want to drive a car and make it **go faster** by pressing its **accelerator pedal**
  - Before you can drive a car, someone has to **design** it
  - A car typically begins as an **engineering drawings**, similar to the **blueprints** that describe the design of a house

https://www.dreamstime.com/royalty-free-stock-photo-car-blueprint-image28443535

Object (3) - An Automobile Example

- Drawings include the design for an **accelerator pedal**.
- Pedal **hides** from the driver the complex mechanisms that actually make the car go faster
- Brake pedal **hides** the mechanisms that slow the car
- Steering wheel **hides** the mechanisms that turn the car.



ALGONQUIN COLLEGE

Objects (4) - An Automobile Example

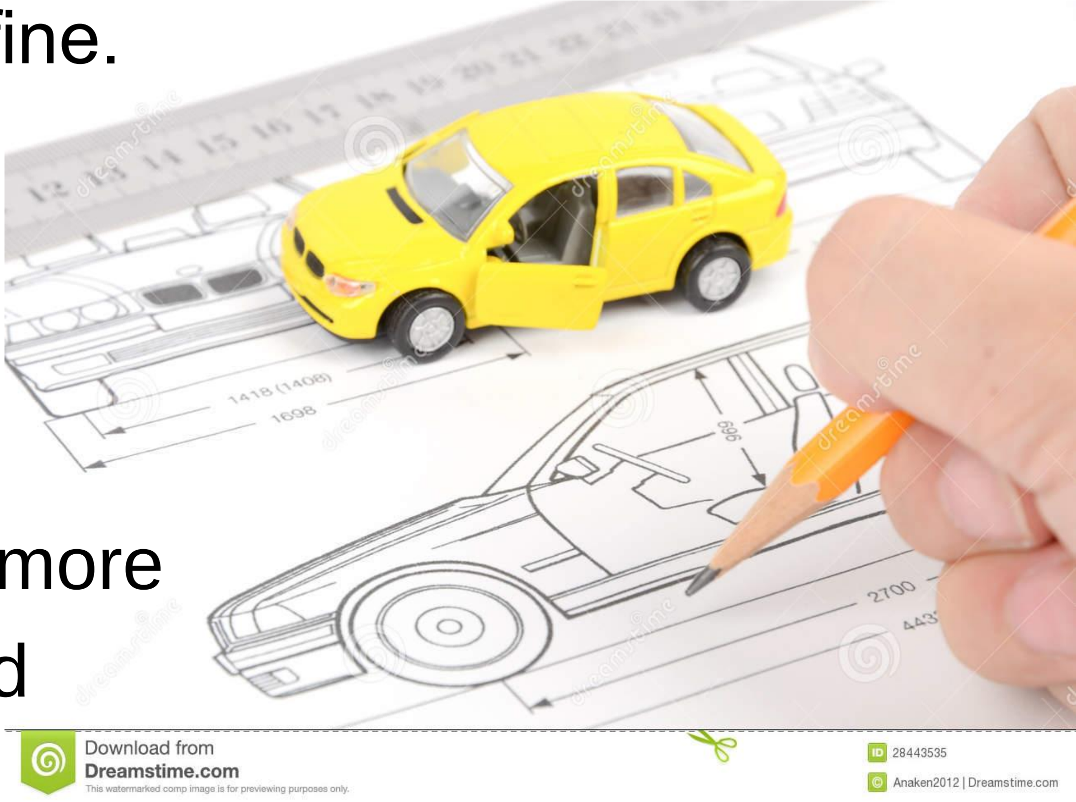
- With little or no knowledge of how **engines**, **braking** and **steering** systems work, you can easily drive a car
- But a car must be **built** first from the engineering drawings that describes it
- A completed car has an **actual** accelerator pedal to make it go faster
- The driver must **press** the pedal to accelerate the car.

Methods and Classes

- Performing a task in a program requires a **method**
- The **method** holds the program statements that **perform** its tasks
- **Hides** these statements from its user
- In Java, program units called a **class** are created to hold the set of **methods** that perform the class’s tasks

Instantiation

- You need to build an **object** of a class **before** a program can perform the tasks that the class’s methods define.



- An **object** is referred to as an **instance** of its class.

- A class has one or more special methods called a **constructor**

Instantiation (2)

- An object is instantiated (new instance of the class) through a **constructor** (**created**)
- **Properties** and **methods** in a **class** are called **class fields** and **class methods**
- **Properties** and **methods** in an **object** are referred to as **instance properties** and **instance methods**

ALGONQUIN COLLEGE

Garbage Collection

- Objects occupy space in memory
- Once they're no longer needed, they should be deleted from memory.
- In Java, **garbage collection** occurs automatically.

Reuse

- Car’s engineering drawings can be **reused many** times to build many cars
- A **class** can be reused **many** times to build **many objects**

Picture from: motoringresearch.com

Reuse (2)

- **Reuse saves time:** using existing classes in building new classes and programs saves time and effort

- **Reuse can create reliable and effective systems**: by reusing existing classes and components that have undergone extensive **testing, debugging** and **performance** tuning

- **Reusable classes** are important to the software revolution started by object technology.

Messages and Method Calls

- **Messages** can be sent to an object
- Each message is implemented as a **method call** which informs a method of the object to **perform** its task.

Attributes and Instance Variables

- A **car** has **attributes**
  - Color, number of doors, amount of gas in its tank, its current speed, record of total miles driven, etc.
- The car's attributes are represented in its design in the engineering diagrams.
  - **Each** car maintains its **own** attributes.
  - **Each** car knows how much gas is in **its own** gas tank, **but not** in the tanks of **other** cars.

Attributes and Instance Variables (2)

- An **object** has **attributes** that it carries along as it is used in a program

- Specified as part of the **object's class**.

- Attributes are **specified** by the class's **instance variables**

Attributes and Instance Variables (3)

EXAMPLE:

- A bank-account object has a **balance attribute** that represents the amount of money in the account

- **Each bank-account** object knows the balance in its account, but **not the balances** of the **other** accounts in the bank

Encapsulation and Information Hiding

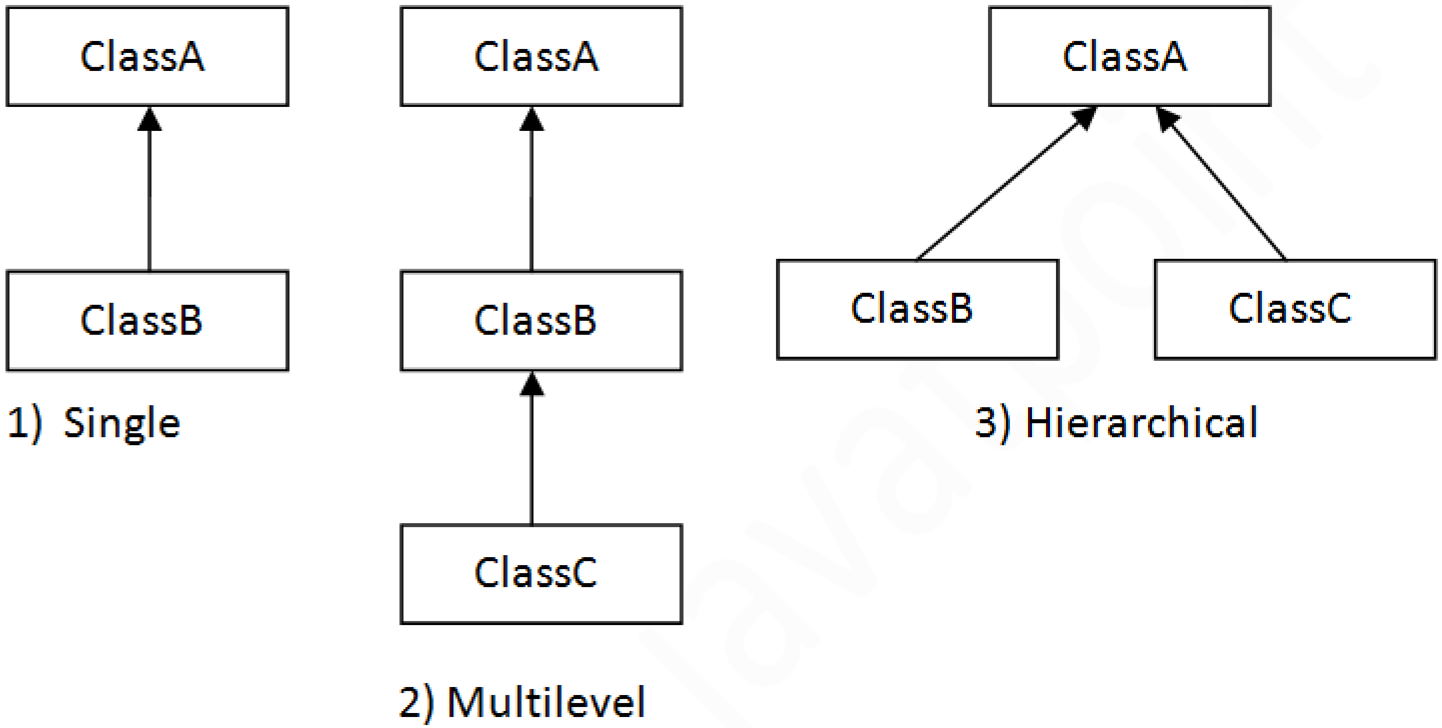
- Classes (and their objects) **encapsulate**
  - encase their **attributes** and **methods**
- Objects may communicate with one another
  - Not allowed to know **how** other objects are implemented
- Implementation details can be **hidden** within the objects themselves (declared **private**)
- **Information hiding** is crucial to good software engineering.

Inheritance

- A new **class** of objects can be created conveniently by **inheritance**
- The new class is called the **subclass** and starts with the characteristics of an existing class called the **superclass**
- Subclass can **customize** the characteristics of the superclass and add **unique** characteristics of its own

Inheritance (2)

- Therefore, the automobile example:
  - An object of class **“convertible”** certainly ***is an*** object of the more **general** class **“automobile,”**
  - But, more specifically, the roof can be raised or lowered


ClassA
ClassB
1) Single
ClassA
ClassB
ClassC
2) Multilevel
ClassA
ClassB
ClassC
3) Hierarchical

Inheritance (3)

- **A new class** is created by **acquiring** an existing class's members and may be enhancing them with new or modified capabilities

- **Can save time** during program development by basing new classes on existing proven and debugged high-quality software.

- **Increases** the likelihood that a system will be implemented and maintained effectively.

Inheritance (4)

- A **subclass** is **more specific** than its superclass and represents a **more specialized group** of objects

- The **subclass** exhibits the **behaviors** of its **superclass** and can **add behaviors** that are specific to the subclass

  - This is the **reason** inheritance is sometimes referred to as **specialization.**

Polymorphism

- The **capability** of a method to do different things based on the object that it is acting upon.

- Feature that allows us to perform an action in different ways.

Example of concept

```
public class Food{
	...
	public void taste(){
		System.out.println(“Food has taste");
	}
}
```

But, we want some **spicy taste!**

Example of concept (2)

- As a generic class, we cannot give **Food** a **specific** implementation of taste().
- Instead we gave a **generic** message
- Assuming we have one **subclass** of the Food class, named **Peppers** that **extends** the Food class.

- Then, we can implement the same **method** in the following way:

Example of concept (3)

```
public class Peppers extends Food{
…
	@override
	public void taste(){
		System.out.println(“Spicy”);
	}
}
```

Gives a different way to implement taste()

Interfaces

- **Interfaces** are collections of **related methods** that typically enable you to tell objects **what** to do, but not **how** to do it.

- **Automobile example:** a "basic-driving-capabilities" **interface** consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car **what** to do.

- Knowing how to use this interface for turning, accelerating and braking, you can drive **many types** of cars, even though manufacturers may implement these systems differently.

Interfaces (2)

- A class **implements** zero or more interfaces, each of which can have one or more methods
- Different car manufacturers can implement the same capabilities in their cars **differently**:
  - Thus, classes may implement an interface's methods **differently**.

The UML (Unified Modeling Language)

- The Unified Modeling Language (UML) is the most widely used graphical scheme for modeling object-oriented systems.

Java

- Using Java, you can write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called "write once, run anywhere."

- Java drew the attention of the business community because of the phenomenal interest in the Internet.

Java (2)

- Used to **develop** large-scale enterprise applications, to **enhance** the functionality of web servers, to provide applications for consumer devices, to develop robotics software and for many other purposes.

- Key language for developing Android smartphone and tablet apps.

- Most widely used **general-purpose** programming language

Java (3)

Java Class Libraries

- Rich collections of existing **classes** and **methods**
- Also referred to as **Java APIs (Application Programming Interfaces)**

Writing your own Java API

Even though Java has rich collections of existing **classes** and **methods** (APIs) isn't it advisable to **write** your own?

- If yes, why?
- If No, why?

A Typical Java Development Environment

- Normally there are **five phases**
  - edit
  - compile
  - load
  - verify
  - Execute

A Typical Java Development Environment (2)

- **Phase 1** consists of **editing** a file with an **editor program**
  - Type a Java program (**source code**) into an editor
  - Make any necessary **corrections**
  - **Save** the program
  - Java source code files are given a name ending with the **.java extension** indicating that the file contains Java source code

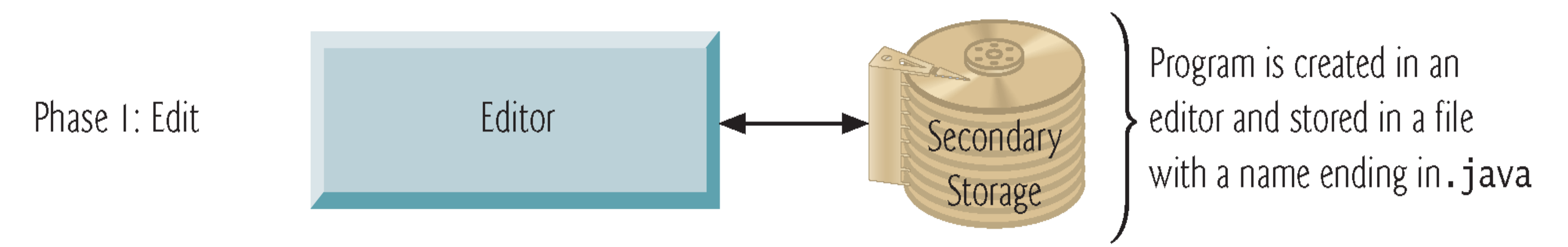


Fig. 1.6 | Typical Java development environment—editing phase.

A Typical Java Development Environment (2)

- Two **editors** widely used on Linux systems are **vi** and **emacs**
- Windows provides **Notepad**
- macOS provides **TextEdit**
- Many **freeware** and **shareware** editors are also available online, including
  - Notepad++ (http://notepad-plus-plus.org)
  - EditPlus (http://www.editplus.com)
  - TextPad (http://www.textpad.com)
  - jEdit (http://www.jedit.org) and more.

A Typical Java Development Environment (2)

- **Integrated development environments (IDEs)** provide tools that support the software development **process**, such as editors, debuggers for locating **logic errors** that cause programs to execute incorrectly and more.
- The most popular **Java IDEs** are:
  - Eclipse (http://www.eclipse.org)
  - IntelliJ IDEA (http://www.jetbrains.com)
  - NetBeans (http://www.netbeans.org)

A Typical Java Development Environment (3)

- **Phase 2:** Compiling a Java Program into Bytecodes:
  - Use the command javac (the Java compiler) to compile a program. For example, to compile a program called `Welcome.java`, you type:
    - `javac Welcome.java`
  - If the program compiles, the compiler produces a .class file called `Welcome.class` that contains the compiled version.

.

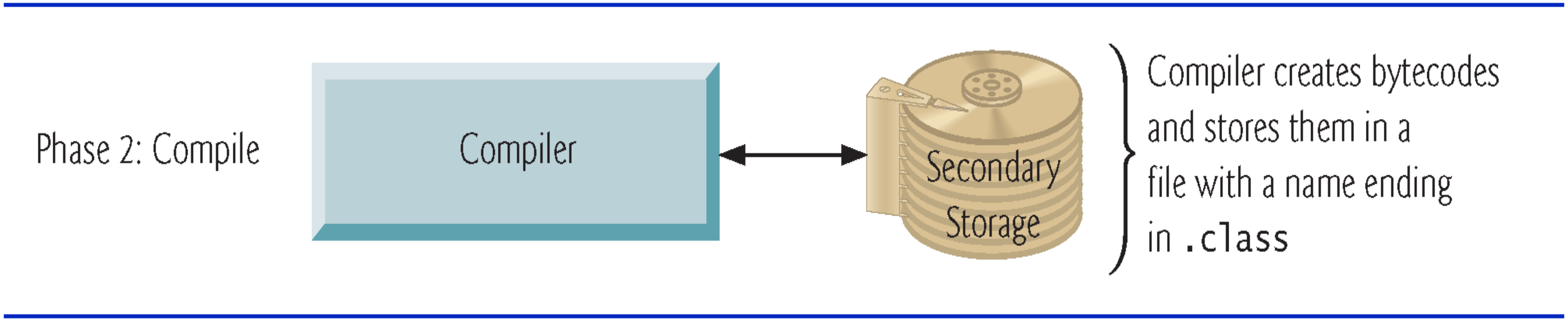


Fig. 1.7 | Typical Java development environment—compilation phase.

Common Error when using javac

Think back to your observations in Lab 1, what possibly could go wrong while using javac?

A Typical Java Development Environment (4)

- **Java compiler** translates Java source code into bytecodes that represent the tasks to execute.
- The **Java Virtual Machine (JVM)—**a part of the JDK and the foundation of the Java platform—executes bytecodes.
- **Virtual machine (VM)**—a software application that simulates a computer
  - Hides the underlying operating system and hardware from the programs that interact with it.
- If the same VM is implemented on many computer platforms, applications written for that type of VM can be **used on all** those platforms.

A Typical Java Development Environment (5)

- Bytecode instructions are **platform independent**
- Bytecodes are **portable**
  - The same bytecode instructions can execute on any platform containing a JVM that understands the version of Java in which the bytecode instructions were compiled.
- The JVM is invoked by the **java command**. For example, to execute a Java application called `Welcome`, you can type:
  - `java Welcome`

A Typical Java Development Environment (6)

- **Phase 3:** Loading a Program into Memory
  - The JVM places the program in memory to execute it—this is known as loading
  - Class loader takes the `.class` files containing the program’s bytecodes and transfers them to primary memory
  - Also loads any of the `.class` files provided by Java that your program uses
- The `.class` files can be loaded from a disk on your system or over a network

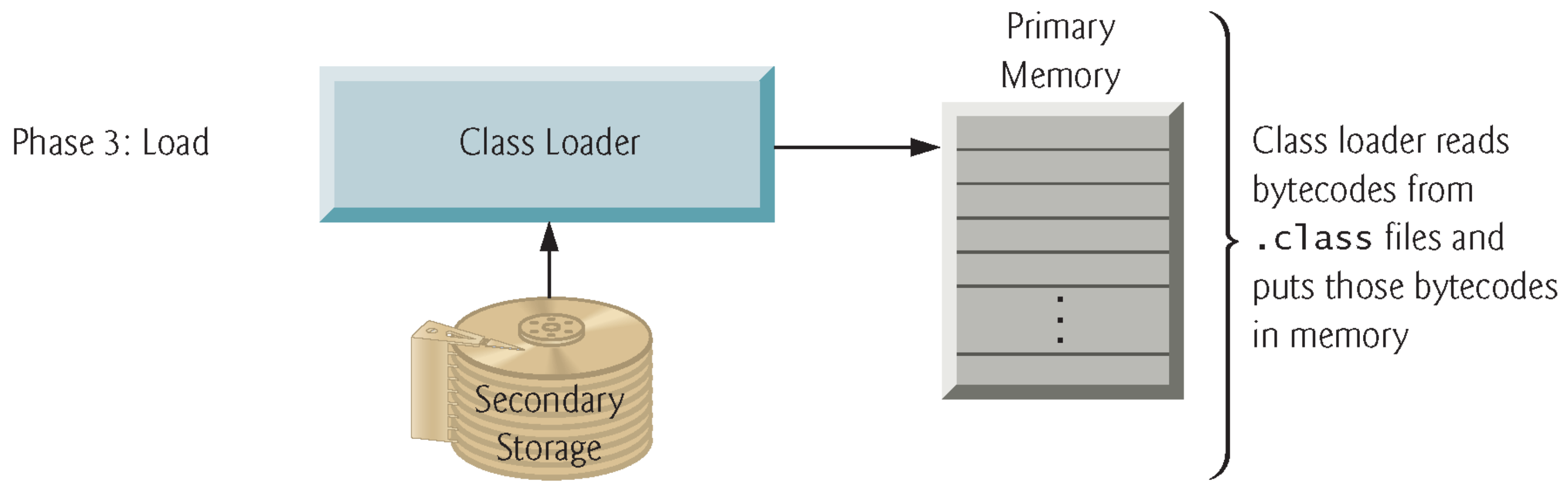


Fig. 1.8 | Typical Java development environment—loading phase.

A Typical Java Development Environment (7)

- **Phase 4: Bytecode Verification**
  - As the classes are loaded, the **bytecode verifier** examines their bytecodes
  - Ensures that they're valid and do not violate Java's security restrictions.
- Java enforces **strong security** to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might).

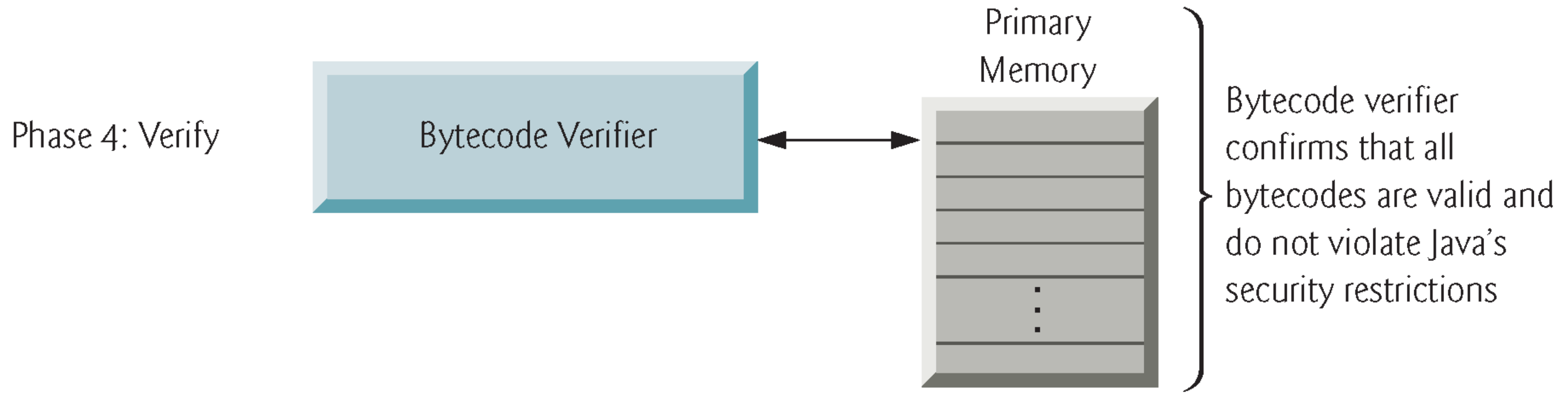


Fig. 1.9 | Typical Java development environment—verification phase.

A Typical Java Development Environment (8)

- **<u>Phase 5:</u> Execution**
  - The JVM **executes** the program's bytecodes
  - JVMs typically execute bytecodes using a combination of interpretation and **so-called <u>just-in-time (JIT) compilation</u>**
  - Analyzes the bytecodes as they are interpreted
  - A **just-in-time (JIT) compiler**, such as Oracle's **Java HotSpot™ compiler** that translates the bytecodes into the underlying computer's **machine language**

A Typical Java Development Environment (9)

- When the JVM encounters these compiled parts again, the faster machine-language code executes

- Java programs go through **two** compilation **phases:**

  - One - source code **is translated** into **bytecodes** (for portability across JVMs on different computer platforms)

  - Two - during execution, the **bytecodes** are **translated** into **machine language** for the actual computer on which the program executes

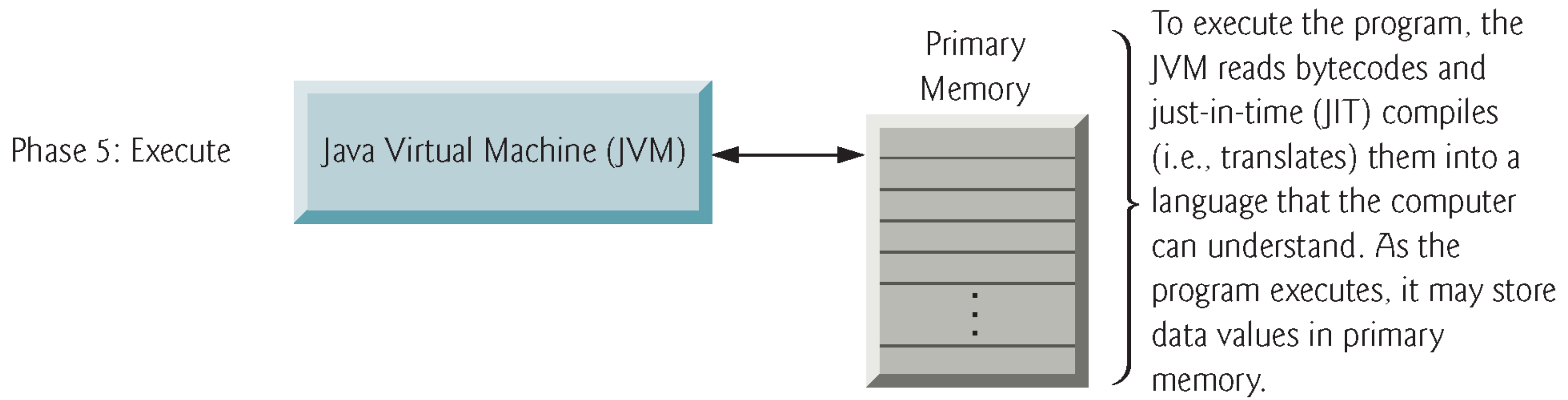


Fig. 1.10 | Typical Java development environment—execution phase.

Common Programming Errors

- Division by zero errors occur as a program runs – **runtime** or **execution-time** errors
  - **Fatal** runtime errors cause programs to **terminate** immediately **without** having successfully performed their jobs
  - **Nonfatal** runtime errors allow programs to run to **completion,** but provide **incorrect** results

References

- Slides for this lecture are taken from the textbook for this course and:

Java How to Program, Early Objects Plus MyProgrammingLab with Pearson eText -- Access Card Package, 11/E. Author: Deitel ISBN: 9780134800271

Finalizing…

- Remember to complete any **outstanding lab** or assessments
- Continue your study this week by revising the corresponding sections from the course **textbook**
- The **hybrid** section for the week has been released. Do well to review it as it will be part of your lab and other assessments