**Lecture Notes #2**

**Translation**

At the lowest level, hardware executes machine language, so higher level languages need applications to translate to machine language.

**Language Translator Target**

Machine code Hardware

Assembly Assembler Machine code

High level language Compiler Machine code

High level language Compiler Assembly

Java source javac compiler Java byte code

Java byte code JVM Machine code

So for Java, the situation is slightly more complicated. The javac compiler does not convert .java files directly to machine code.

An article on compilers vs. interpreters: <https://www.geeksforgeeks.org/compiler-vs-interpreter-2/>

A Java application is able to run on its own.

A Java applet requires a web browser to run.

Java is not related to javascript. Javascript is a name produced by a marketing department to take advantage of the popularity of Java.

[Software Engineering](https://en.wikipedia.org/wiki/Software_engineering)

The Software Life Cycle:

* Requirements Analysis (What problem are we solving, and are we solving the right problem?)
* Design (UML Diagrams or other design ‘on paper’)
* Coding (Should be fairly easy if the design is precise)
* Testing (Static, Dynamic, White Box, Black Box, unit testing, end-to-end testing)
* Debugging (Debuggers, Log files)
* Operation
* Maintenance (May include enhancements as well as bug fixes)

It is generally good practice to incorporate testing into as many of the above phases as possible.

We will be discussing Java from now on.

A class is like a pattern to produce ‘things’.

An object is a ‘thing’ that we produce using the pattern defined in one (and only one) class.

We say that an object is an instance of a class. (instance is a noun)

When we create an object of a class using the **new** reserved word, we instantiate that object. (instantiate is a verb)

We can produce as many objects of a class as we would like. (until we run out of memory!)

You must first define a class before you can start producing objects from that class.

A message can be sent to a class or object. This is done by calling a method. “Sending a message” means the same thing as “calling a method”.

Class methods operate on classes. Instance methods operate on objects.

Data fields: An instance data value, or field, is used when there is a possibility that this field can differ across the objects of the class. Each object in the class has its own storage for the value.

A class data value should be used to store data that is the same for all objects in the class. (e.g. A maximum value for all members, the count of the number of objects created, etc.) There is only one storage location for a class data value.

Objects are created from a class by using the **new** operator. The **new** operator allocates the space needed for the object that is created.

UML Diagrams

|  |  |
| --- | --- |
| The class Bicycle | Bicycle |
| An Object named redSchwinn  (the underline identifies this as an object) | redSchwinn |
| An object named redSchwinn,  Of class Bicycle | redSchwinn: Bicycle |

Let’s say I have 2 methods for the above:

getFrontWheelDiameter()

getMaxWheelDiameter()

Which should be a class method? An instance method?

Each object in class Bicycle should store a unique front wheel diameter as an instance data value. The Bicycle class itself might store a maximum possible front wheel diameter as a class data value.

Inheritance

A subclass can inherit behavior and data from a superclass. This relationship can also be described as descendant-ancestor or derived class – base class.

An inheritance diagram:

Vehicle

Motorcycle

Car

The arrowheads point from the subclass to the superclass. The superclass is always at the top of an inheritance diagram.

Inheritance is not limited to one level. In fact, every class in Java ultimately inherits from the base class **‘Object’**.

**Naming *rules* for Java identifiers:**

Identifiers are names that we assign to classes, objects, methods, variables, and constants.

Identifiers must consist of upper and lower case letters, digits, ‘\_’, and ‘$’, and begin with a non-digit.

Your Java code will not compile if it does not follow the naming *rules*.

**Naming *conventions* for Java identifiers:**

Classes begin with an uppercase letter, and are usually nouns.

Objects begin with a lowercase letter and are usually nouns.

Methods begin with a lowercase letter and are usually verbs.

(See the textbook for more detail on the naming conventions)

Your Java code will compile if you violate the naming conventions, but it is not socially acceptable to do so. Other Java programmers will have more difficulty understanding your code, they will not want to help you debug it, and they will think that you are very uncool.

**State of memory diagrams:**

* When an object is named, no space has been allocated for it yet. We say that the object has been **declared**:

JFrame window1;

* When space is allocated using the **new** reserved word, a **reference** to the object is created that is tied to the name that has been declared:

window1 = new JFrame();

(a reference is another name for the address of the object’s location in the computer’s memory)

* You can have more than one reference to the same object:

window1 = new JFrame();

JFrame window2;

window2 = window1;

References can be reassigned, but it is possible for the allocated memory to be “lost”.

Consider this situation:

JFrame window3 = new JFrame();

JFrame window4 = new JFrame();

window4 = window3;

When the 3rd line is executed, both window3 and window4 reference the same memory location. The reference to the memory originally allocated for window4 is lost forever.

Java will take care of such “orphaned” memory through its built-in “**garbage collector**”. The garbage collector process runs occasionally, searching for objects in memory that no longer have any reference to them, and returns them to the system for re-use.

Some popular computer languages do not have garbage collection, and poor programming practice can result in “**memory leaks**” where the memory used by an application can grow and grow during execution, resulting in a slow or eventually unusable system.