BOOK 1

2018

HARTCODE ACADEMY PUBLISHING

THE JAVA CHRONICLES

SOUND BITES

WHAT IS JAVA?

OOP BASICS

ABSTRACTION

ENCAPSULATION

STRINGS

OPERATORS

POLYMORPHISM

INHERITANCE

VARIABLES

JAVA SOUND BITES

IT IS NOT JAVASCRIPT!

They are only syntactically similar.

OPEN-SOURCED

Way too big for us to cover all aspects in one session. There are different ways to use Java, different versions of Java, and thousands of libraries and API's to utilize.

JAVA IS MASSIVE AND DENSE

BASED ON C AND C++ PROGRAMMING LANGUAGES

Find out how you're going to USE Java and use that knowledge to determine how you should LEARN Java.

CREATED IN 1995

Final is not final in Java

Final has four different meanings in Java:

final class: the class cannot be extended

final method: the method cannot be overridden

final field: the field is a constant

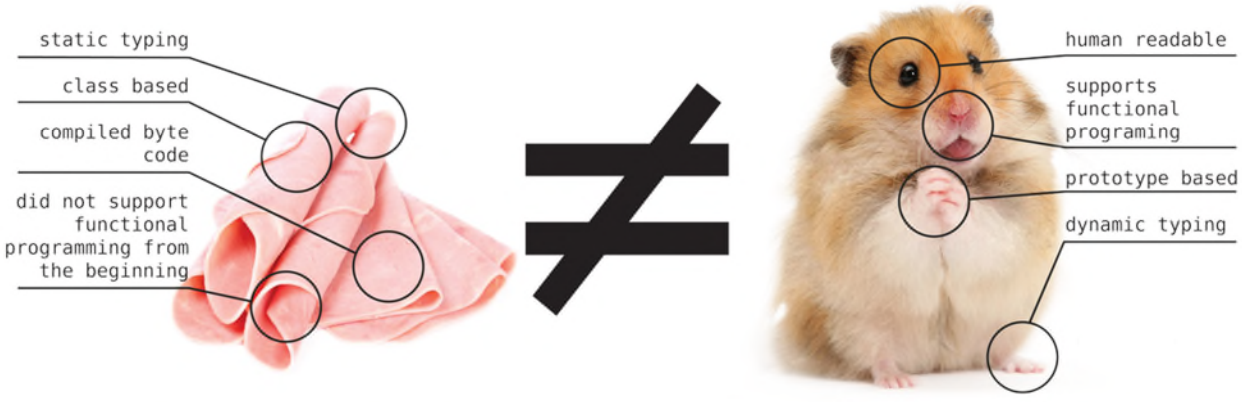
final variable: the value of the variable cannot be changed once assigned

REMEMBER!

JAVA IS TO

{JAVASCRIPT}

AS HAM IS TO A HAMSTER

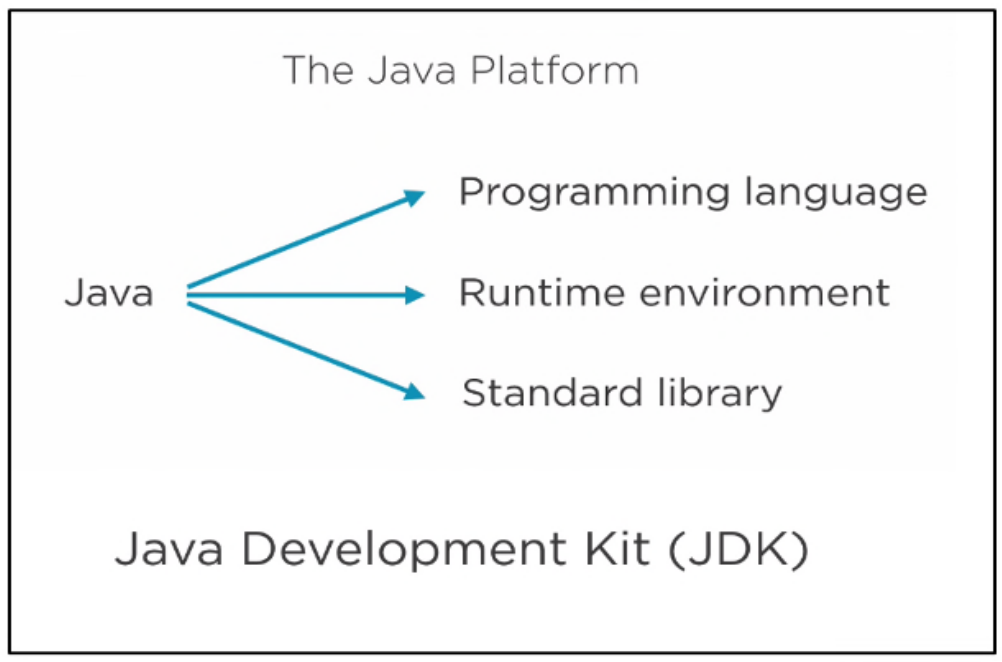


PROGRAMMING LANGUAGE?

In general, when people refer to Java, they are usually talking about the programming language.

Java is actually made up of the programming language, the *runtime environment*, and the *standard library*.

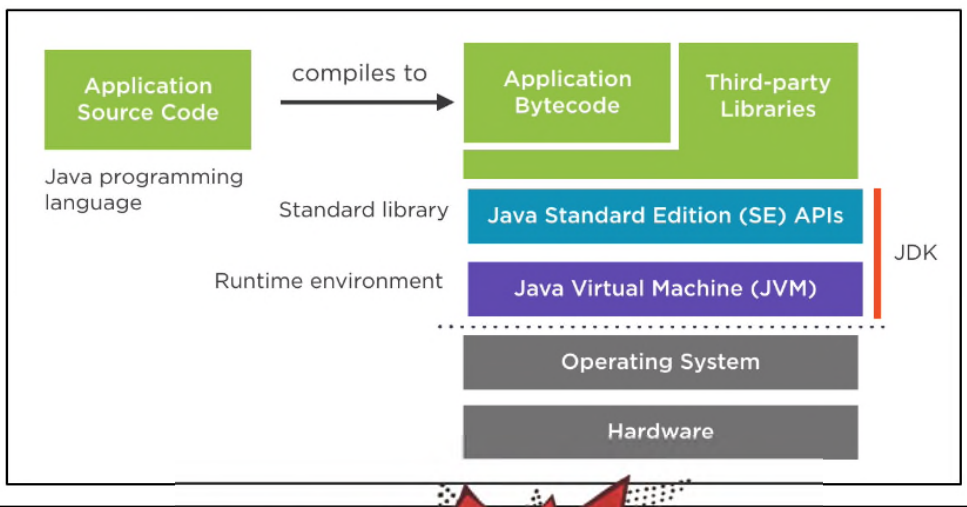
The *runtime environment* and the *standard library* are what make up the *Java Development Kit* and are critical to making Java *platform independent.*



SO, WHAT DOES PLATFORM INDEPENDENT ACTUALLY MEAN??

You only need to write one set of code which can then be run on different platforms. Java accomplishes this by compiling the *application source code* into *application bytecode*.

This gets combined with any libraries and APIs which the *Java Virtual Machine* then sends to the operating system. The operating system is able to convert it into code that is compatible with the system and hardware of the device attempting to run the Java program.



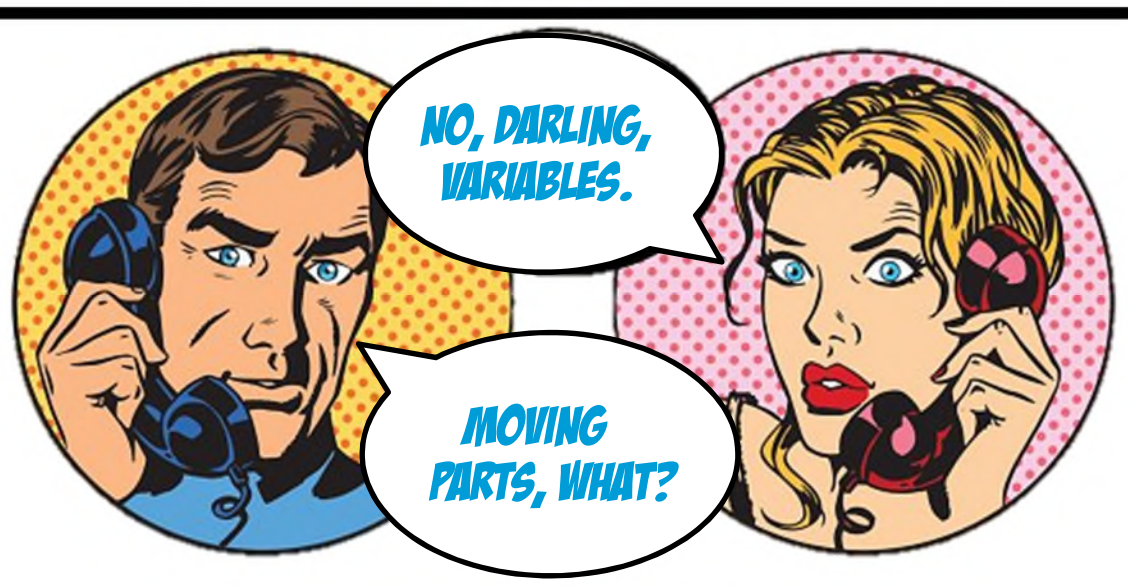
Make sure you have a JDK installed on your machine!

Next, get yourself an IDE that has the features you're looking for.

What the heck is an IDE, you ask??

An IDE is an *Integrated Development Environment*. It's a program that allows you to write, debug, compile, and run Java code.

Common examples are Eclipse, MyEclipse, NetBeans, IntelliJ, and VSCode.



STRONGLY TYPED AND MODIFIABLE

PRIMITIVE DATA TYPES:

- ✓ Data types BUILT INTO the language
  - Foundation of ALL other types
- ✓ Four categories:
  - Integer
  - Floating point
  - Character
  - Boolean

NAMING CONVENTIONS:

- ✓ Rules allow the use of LETTERS, NUMBERS, $ and _
  - Typically, you will only use letters and numbers
- ✓ The first character CANNOT be a NUMBER
- ✓ Written in camelCase

INTEGER TYPES:

Four different types, but the only real difference is the size of storage they take up which affects the range of values that can be stored in them.

Type	Size (bits)	Min Value	Max Value	Literal Format
byte	8	-128	127	0
short	16	-32768	32767	0
int	32	-2147483648	2147483647	0
long	64	-9223372036854775808	9223372036854775807	0L

```
byte numberOfEnglishLetters = 26;
short feetInAMile = 5283;
int milesToSun = 92960000;
long nationalDebt = 18100000000000L;
```

FLOATING POINT TYPES:

A bit more complicated, but main points are:

- Stores values containing a fractional portion
- Supports positive, negative, and zero values

Type	Size (bits)	Smallest Positive Value	Largest Positive Value	Literal Format
float	32	1.4×10^{-45}	3.4×10^{38}	0.0f
double	64	4.9×10^{-324}	1.7×10^{308}	0.0 or 0.0d

Any number with a decimal and no additional notation will be assumed to be a double!

```
float milesInAMarathon = 26.2f;
double atomWidthInMeters= 0.0000000001d;
```

CHARACTER & BOOLEAN TYPES:

The **char** type stores a single Unicode character.

```
char regularU = 'U';
```

Single quotes!

Want to use a character that isn't on your keyboard?

Use Unicode code points!

- ✓ \u followed by the 4-digit hex value
  char accentedO = '\u00D3'; Ó

The **boolean** type stores true/false values

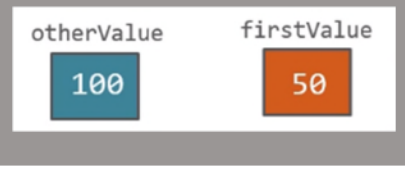
```
boolean iLoveJava = true;
```

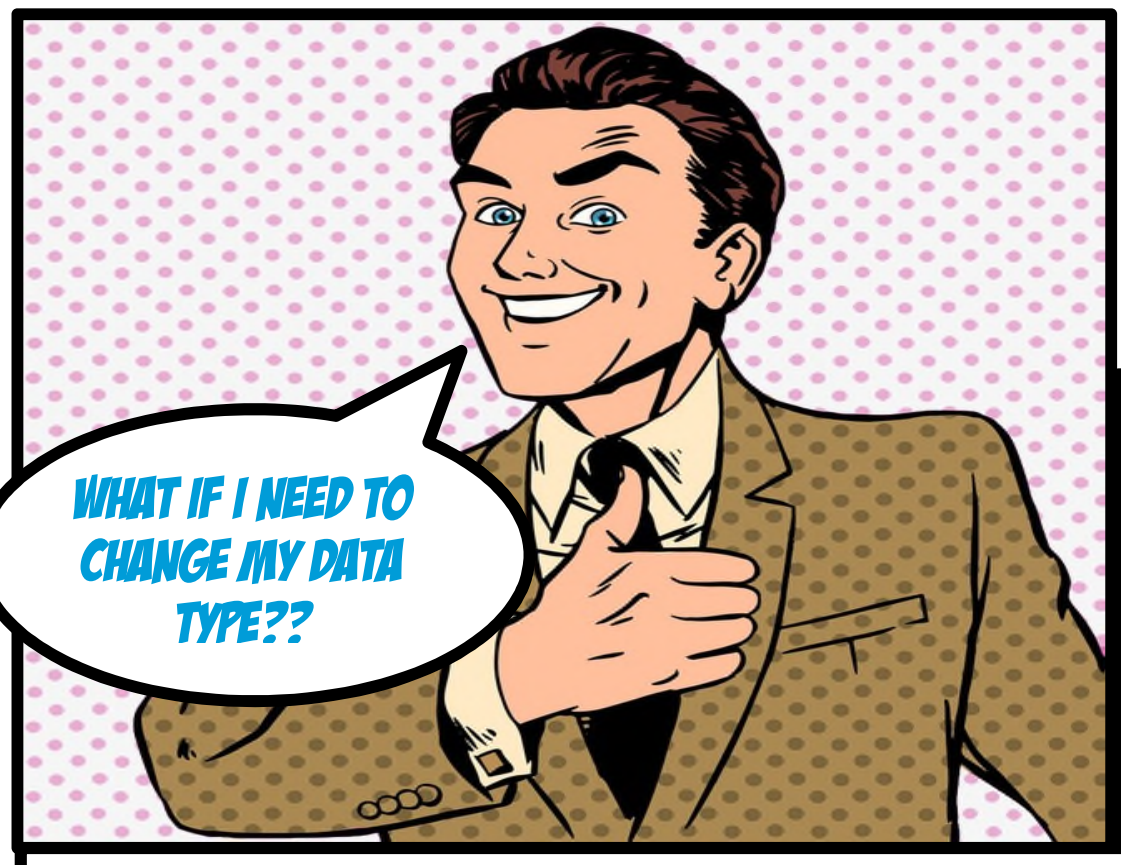
THINGS TO KNOW:

Primitive data types are stored **by-value**

- This means that you can assign one value:
  int firstValue = 100;
- Then assign another variable to equal that same value:
  int otherValue = firstValue;
- And what you'll wind up with is two copies of the value 100

- If you then change the value of firstValue:
  firstValue = 50;
- otherValue will retain the original value of 100 because that is the value stored in that variable





ENTER...TYPE CONVERSION:

There are two types of type conversion:

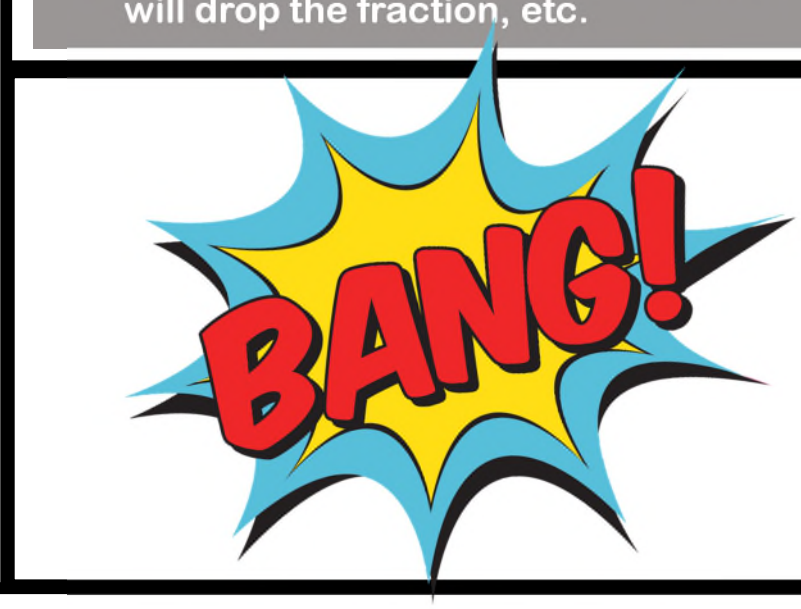
- ✓ Implicit
- ✓ Explicit

EXPLICIT TYPE CONVERSION:

- ✓ These conversions are not automatic and only happen when we call them.
- ✓ This means they can both *widen* and *narrow*
- ✓ Be mindful when narrowing!
  - Casting from a floating point to an int will drop the fraction, etc.

IMPLICIT TYPE CONVERSION:

- ✓ Widening conversions are automatic
  - byte to short
  - short to int
  - int to long
- ✓ Happens when there are mixed integer sizes
  - Uses the largest integer in the equation
  - Adding a short to a long, the short will be implicitly cast into a long
- ✓ Happens when there are mixed floating point sizes
  - Uses double
  - Adding a float to a double, the float will be implicitly cast into a double
- ✓ Happens when mixing integer and floating point
  - Will cast all integers into the largest floating point in the equation
  - Adding a long to a float, the long will be implicitly cast into a float



BASIC MATH OPERATORS:

Operator	Meaning	Example	Result
+	Addition	10 + 2	12
-	Subtraction	10 – 2	8
*	Multiplication	10 * 2	20
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 2	0
++	Increment	a++ (consider a = 10)	11
--	Decrement	a-- (consider a = 10)	9
+=	Addition Assignment	a += 10 (consider a = 10)	20
-=	Subtraction assignment	a -= 10 (consider a = 10)	0
*=	Multiplication assignment	a *= 10 (consider a = 10)	100
/=	Division assignment	a /= 10 (consider a = 10)	1
%=	Modulus assignment	a %= 10 (consider a = 10)	0

PREFIX / POSTFIX OPERATORS:

++ increments value by 1

-- decrements value by 1

As prefix applies operation before returning value

```
int myVal = 5;
System.out.println(++myVal);   6
System.out.println(myVal);     6
```

As postfix applies operation after returning value

```
int myVal = 5;
System.out.println(myVal++);   5
System.out.println(myVal);     6
```

BUT! WHAT ABOUT STRINGS??

IN JAVA, STRINGS AREN'T ACTUALLY A DATA TYPE, THEY ARE A CLASS.

SINCE THEY ARE A CLASS, ANY REFERENCE TO STRING SHOULD BE CAPITALIZED.

STRING CLASS:

The String class stores a sequence of Unicode characters

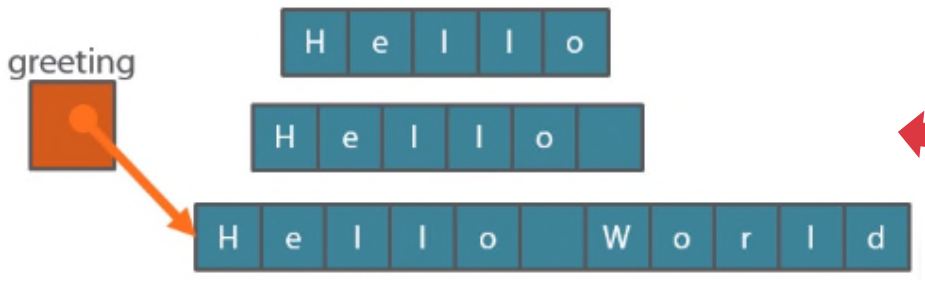
Stored using UTF-16 encoding

- Literals are enclosed in double quotes (" ")
- Values can be concatenated using + and +=
- String objects are immutable

```
String name = "Jim";
```

```
System.out.println("Hi " + name);
```

```
String greeting = "Hello";
greeting += " ";
greeting += "World";
```



Once a String value has been assigned, it can never be changed without a whole new instance of the variable being created.

- This is SUPER INEFFICIENT!

Use the StringBuilder class for instances like this!

MEANWHILE...

OBJECT-ORIENTED PROGRAMMING:

Java, like a lot of other programming languages, is object-oriented.

Object orientation boils down to the way that we separate and group things.

OBJECTS ARE NOT JUST DATA!

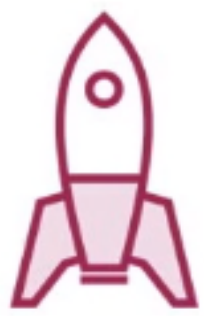
They are data and functionality with behavior and code that perform tasks on that data

Each object almost becomes a self-contained miniature program, able to manage both its own data and its own behavior.

Spaceship

```
// data
positionX
positionY
shieldLevel
color
name

// behavior
fly()
fireMissile()
explode()
...
```



Understanding Objects

Begin with non-technical language

Think about what your program needs to do, in plain English, and then look at the nouns.

"A customer can begin a new order
by adding an item to the shopping cart"

Potential objects: customer, order, item, shopping cart...

For a media player: album, track, playlist...

For a game: spaceship, enemy, asteroid, missile...

OO TERMINOLOGY:

Class	Object
How we define an object	The object itself
"The Blueprint"	"The house made from the blueprint"
"The Recipe"	"The dish made from the recipe"
One class can be used to make multiple objects	
The class comes first!	

DEFINING A CLASS:

```
class Lamp {
  boolean isOn;

  void turnOn() {
     isOn = true;
  }

  void turnOff() {
    isOn = false;
  }

  void displayLightStatus() {

      System.out.println("Light on? " + isOn);
  }
}
```

THERE IS NO POINT IN DEFINING A CLASS UNLESS YOU PLAN TO MAKE AN OBJECT, SO...

CREATING AN OBJECT AKA INSTANTIATION

DECLARE AND CONSTRUCT:

```
// Declare 3 instances of the class Circle, c1, c2, and c3
Circle c1, c2, c3;  // They hold a special value called null
// Construct the instances via new operator
c1 = new Circle();
c2 = new Circle(2.0);
c3 = new Circle(3.0, "red");

// You can Declare and Construct in the same statement
Circle c4 = new Circle();
```

BASIC IDEA:

Once a class has been defined, you can create, or instantiate, multiple objects (instances) based on that class.

Also, most of the objects you instantiate, you won't have to make the class.

In most real world programming environments, there are a huge amount of classes already defined and available for use:

- Dates
- Times
- Image
- Sound Players
- Graphics
- Audio
- Encryption
- User Interfaces

WANT TO BE PRODUCTIVE AS AN OOP PROGRAMMER?

DON'T RE-INVENT THE WHEEL, FIND WHAT IS AVAILABLE AND TAP INTO IT!

DOT OPERATOR?

Q. How do I access a class's methods?

A. Using the dot operator!

1. First, identify the instance you are interested in.

2. Second, use the dot operator (.) to reference the desired member variable or method.

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

    // create a new object from that class
    Lamp lamp1 = new Lamp();
    //call methods
    lamp1.turnOn();
    lamp1.displayLightStatus();

    // instantiate another object
    Lamp lamp2 = new Lamp();
    //call methods
    lamp2.turnOn();
    lamp2.displayLightStatus();

  }
}
```

The ClassObjectsExample class has a main() method and can be executed.

You would save this as "ClassObjectsExample.java", compile it, run it, and be able to study the output.

MEANWHILE...

THERE ARE FOUR MAIN OBJECT-ORIENTED PROGRAMMING CONCEPTS IN JAVA.

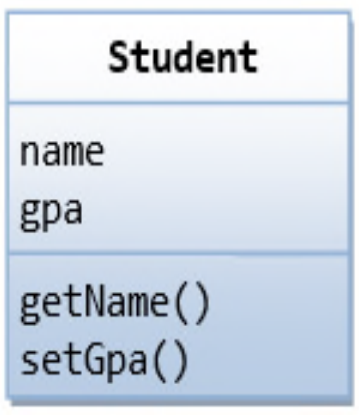
1: ENCAPSULATION

The practice of keeping fields within a class private, then providing access to them via public methods. It keeps the data and code safe within the class itself. This way, we can re-use objects without allowing open access to the data.

Think of it this way:

A class encapsulates the name, static attributes, and dynamic behaviors into a "3-compartment box". You can then seal up the "box" and allow others to use it over and over again without worrying about someone "breaking" a "working box".

Seal the box using a private access control modifier.



A what?

An *access control modifier* is used to *control the visibility* of a class, a member variable, or a member method within a class.

1. public: The class/variable/method is accessible and available to *all* the other objects in the system.

2. private: The class/variable/method is accessible and available *within this class only*.

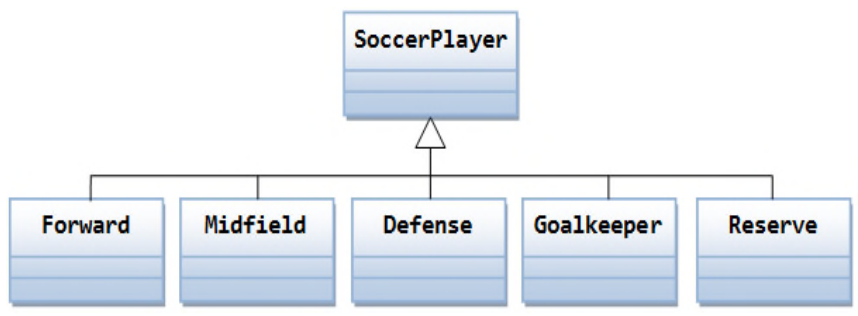
Access to the member variable is granted via public assessor methods, e.g., getName() and setGPA().

This follows the principle of *information hiding*: objects communicate with each other by using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The details are encapsulated within the class. Information hiding facilitates reuse of the class.

2: INHERITANCE

Inheritance allows us to avoid duplication and reduce redundancy.

Think of all the classes in a program belonging to a hierarchy:



The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies.

- ✓ A class in the lower hierarchy is called a *subclass* (or *derived, child, extended class*).
- ✓ A class in the upper hierarchy is called a *superclass* (or *base, parent class*).

Remember: a subclass is not a "subset" of a superclass. Subclasses are "supersets" of a superclass.
Why? Because a subclass inherits all the variables and methods of the superclass AND it extends the superclass by providing more variables and methods.

In Java, a subclass is defined by using the keyword "extends", e.g.

```
class Goalkeeper extends SoccerPlayer {......}
class MyApplet extends java.applet.Applet {.....}
class Cylinder extends Circle {......}
```

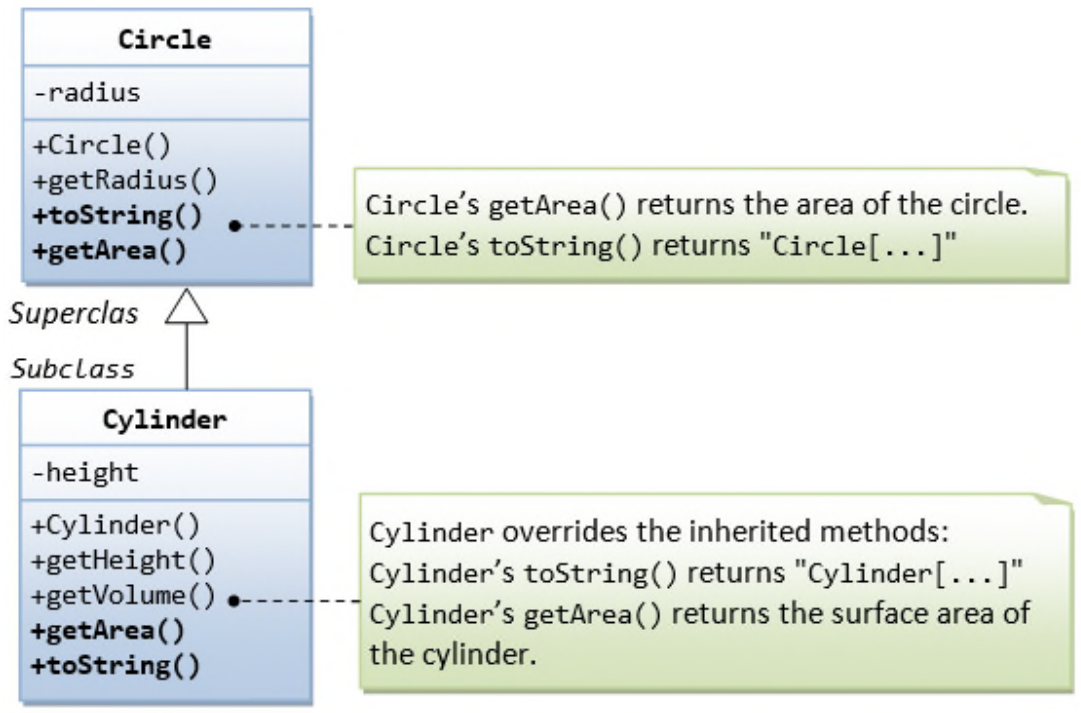
3: POLYMORPHISM

A subclass instance possesses all the attributes and operations of its superclass, right? This means that a subclass object can do whatever its superclass can do! Therefore, when a superclass instance is expected, it can be substituted by a subclass instance.

In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - this is called *substitutability*.

Once substituted, we can invoke methods defined in the superclass; we cannot invoke methods defined in the subclass.

However, if the subclass overrides inherited methods from the superclass, the subclass's version will be executed, instead of the superclass's version.



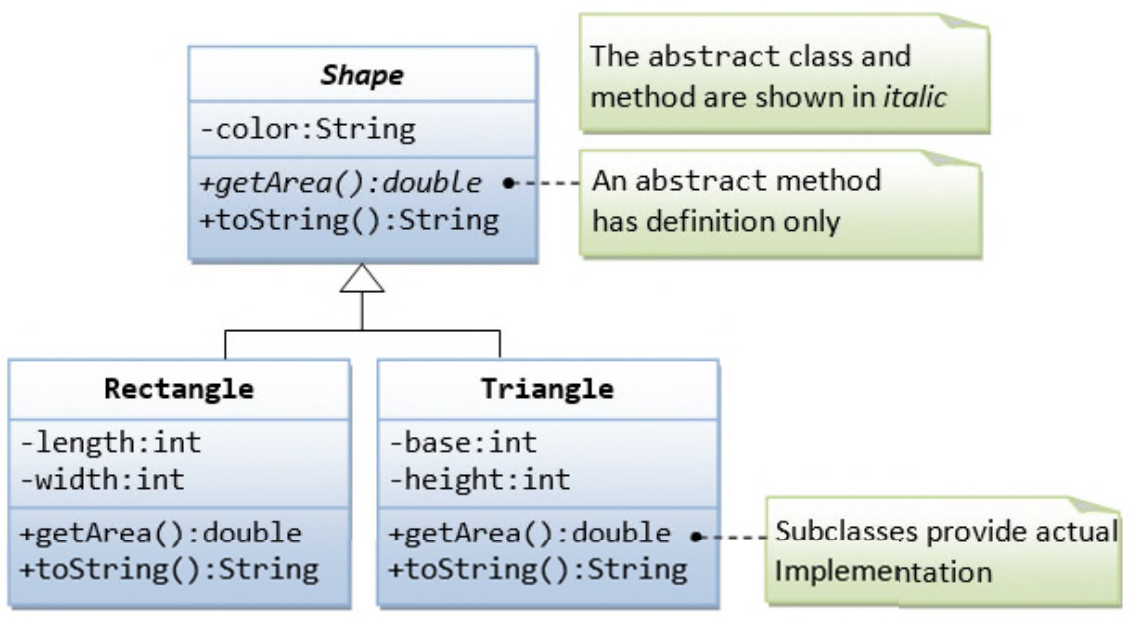
4: ABSTRACTION

Abstraction handles complexity by hiding unnecessary details from the user using Abstract and Interface classes.

This allows the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

Any objects in the real world, like your coffee machine, or classes in your current software project, that hide internal details provide an abstraction.

These abstractions make it a lot easier to handle complexity by splitting them into smaller parts.



WAIT A MINUTE... THIS SOUNDS AN AWFUL LOT LIKE ENCAPSULATION!

Well, you're right, in a way. The easiest way to tell the two apart is with this simple explanation:

Encapsulation: Information hiding.

Abstraction: Implementation hiding.