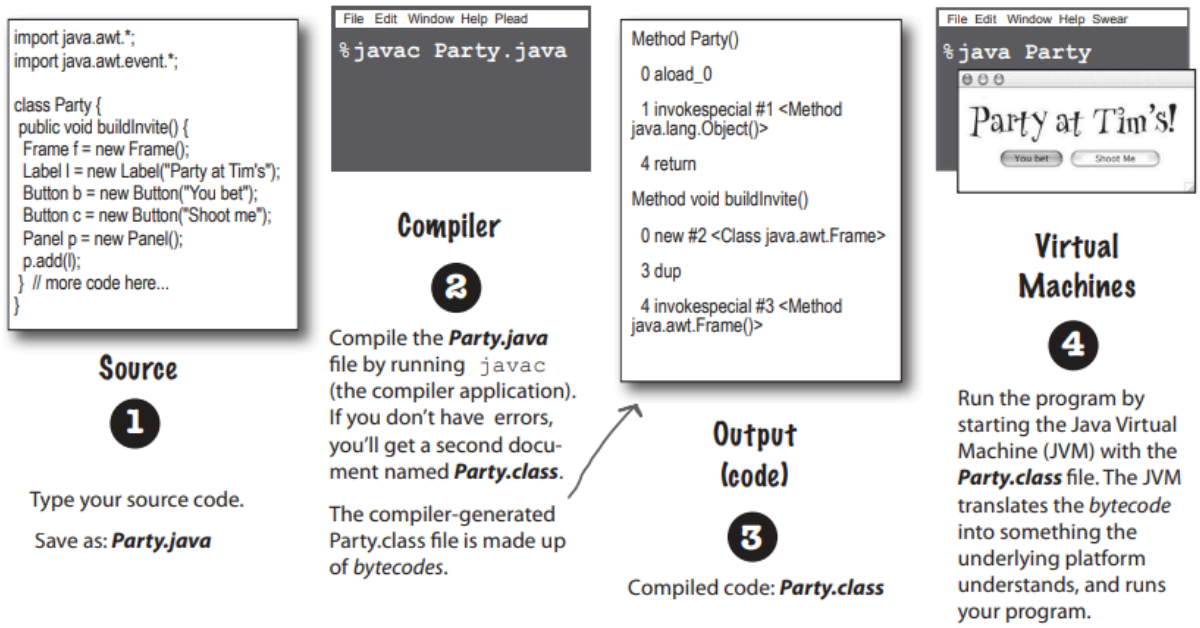
Head First Java Notes

Chapter 1 → Breaking the Surface

The way Java works

You'll have a source code file, compile it using the javac compiler (any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform independent), and then run the compiled bytecode on a Java virtual machine.



History, Speed and Memory usage

Java was initially released, on January 23, 1996 by James Gosling. Java is famous for its backward compatibility, so old code can run quite happily on new JVMs.

When Java was first released, it was slow. But soon after, the HotSpot VM was created, as were other performance enhancers. While it's true that Java isn't the fastest language out there, it's considered to be a very fast language—almost as

fast as languages like C and Rust, and much faster than most other languages out there.
Java has a magic super-power—the JVM. The Java Virtual Machine can optimize your code while it's running, so it's possible to create very fast applications without having to write specialized high-performance code.
Compared to C and Rust,
Java uses a lot of memory.

The Compiler and JVM battle over the question, "Who's more important?"

JVM

What, are you kidding? HELLO. I am Java. I'm the one who actually makes a program run. The compiler just gives you a file, but the file doesn't do anything unless I'm there to run it.

A programmer could just write bytecode by
hand, and I'd take it. You might be out of a job
soon, buddy.

Compiler

Excuse me, but without me, what exactly would you run? There's a reason Java was designed to use a bytecode compiler. If Java
were a purely interpreted language, where—at
runtime—the virtual machine had to translate
straight-from-a-text-editor source code, a Java program would run at a ludicrously glacial pace.

While it is true but a programmer writing bytecode by hand is like painting pictures of your vacation instead of taking photos—sure, it's an art, but most people prefer to use their time differently.

So, what do you actually do?

Remember that Java is a strongly typed language, and that means I can't allow variables to hold data of the wrong type. This is a crucial safety feature, and I'm able to stop the vast majority of violations before they ever get to you.

But some still get through! I can throw ClassCastExceptions and sometimes I get people trying to put the wrong type of thing in an array that was declared to hold something else.

Yes, there are some datatype exceptions that can emerge at runtime, but some of those have to be allowed to support one of Java's other important features—dynamic binding. At runtime, a Java program can include new objects that weren't even known to the original programmer, so I have to allow a certain amount of flexibility. But my job is to top anything that would never—could never—succeed at runtime.

OK. Sure. But what about security?

Excuse me, but I am the first line of defense, as they say. The datatype violations I previously described could wreak havoc in a program if they were allowed to manifest. I am also the one who prevents access violations, such as code trying to invoke a private method, or change a method that—for

Whatever. I have to do that same stuff too, though, just to make sure nobody snuck in after you and changed the bytecode before running it.

security reasons—must never be changed. I stop people from touching code they're not meant to see, including code trying to access another class critical data.

Of course, but as I indicated previously, if I didn't prevent what amounts to perhaps 99% of the potential problems, you would grind to a halt.

Chapter 2 → A Trip to Objectville

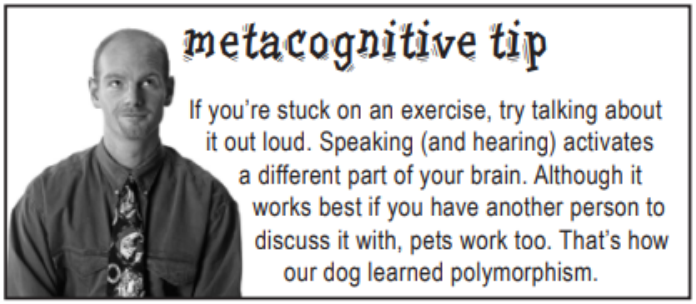
What do you like about OO?

"Object-oriented programming lets you extend a program without having to touch previously tested, working code. (not messing around with code I've already tested, just to add a new feature)"

"It helps me design in a more natural way. Things have a way of evolving."

"I like that the data and the methods that operate on that data are together in one class."

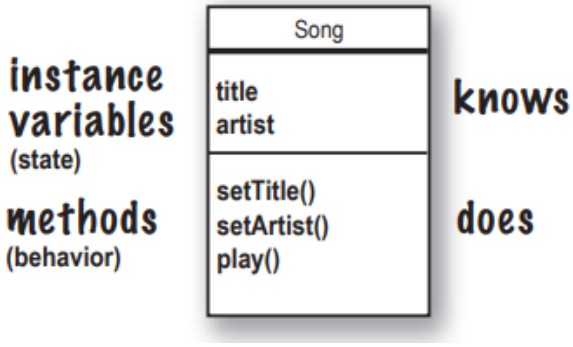
"Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later."



When you design a class, think about the objects that will be created from that class type. Think about:

- things the object knows
- things the object does

Things an object knows about itself are called instance variables. They represent an object's state (the data) and can have unique values for each object of that type.
Things an object can do are called methods. When you design a class, you think about the data an object will need to know about itself, and you also design the methods that operate on that data.
So objects have instance variables and methods, but those instance variables and methods are designed as part of the class.



What's the difference between a class and an object?

💡 **A class is not an object (but it's used to construct them)**

A class is a blueprint for an object. It tells the virtual machine how to make an object of that particular type. Each object made from that class can have its own values for the instance variables of that class.

Chapter 3 → Know Your Variables

Declaring a variable

Variables come in two flavors: primitive and object reference.
Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating-point numbers.

> Eg. byte x = 7;

Primitive Types

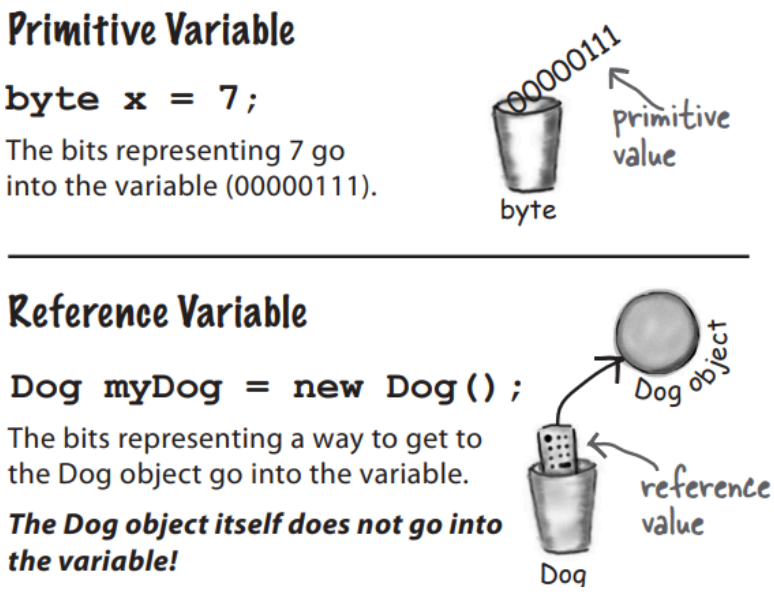
Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32 bits	varies
double	64 bits	varies

An object reference variable holds bits that represent a way to access an object. There is actually no such thing as an object variable. There's only an object reference variable.

Think of a Dog reference variable as a Dog remote control. You use it to get the object to do something (invoke methods).

```
Eg. Dog myDog = new Dog();
```

Although a primitive variable is full of bits representing the actual value of the variable, an object reference variable is full of bits representing a way to get to the object.
You use the dot operator (.) on a reference variable to say, “use the thing before the dot to get me the thing after the dot.” For example:
myDog.bark();
means, “use the object referenced by the variable myDog to invoke the bark() method.” When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.
An object reference is just another variable value. Only this time, the value is a remote control.



You really don’t want to spill that...

Be sure the value can fit into the variable.

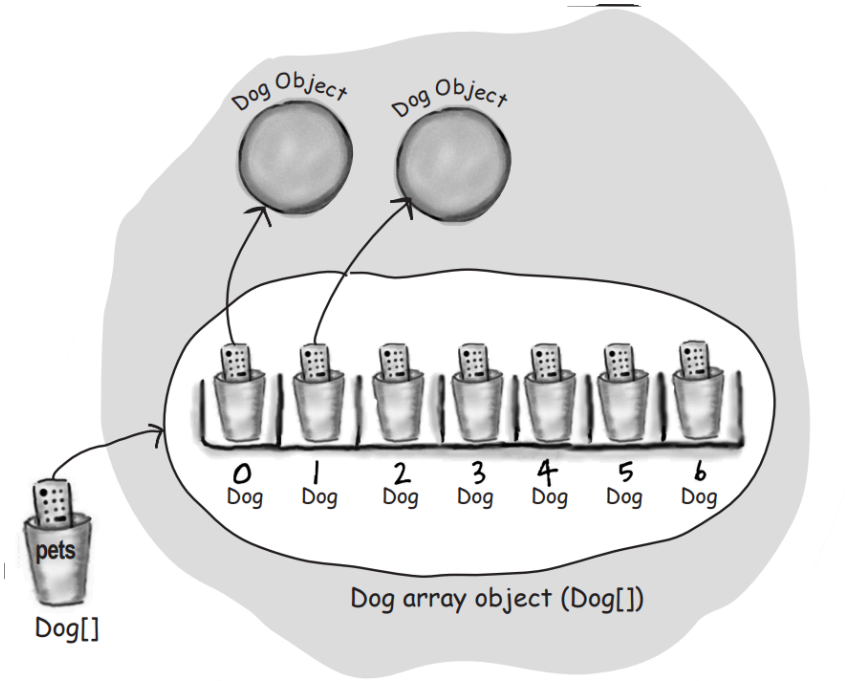
```
int x = 24;
```

```
byte b = x;  //won't work!!
```

Why doesn't this work? After all, the value of x is 24, and 24 is definitely small enough to fit into a byte. You know that, and we know that, but all the compiler cares about is that you're trying to put a big thing into a small thing, and there's the possibility of spilling. Don't expect the compiler to know what the value of x is, even if you happen to be able to see it literally in your code.

Arrays

Arrays are always objects, whether they're declared to hold primitives or object references.



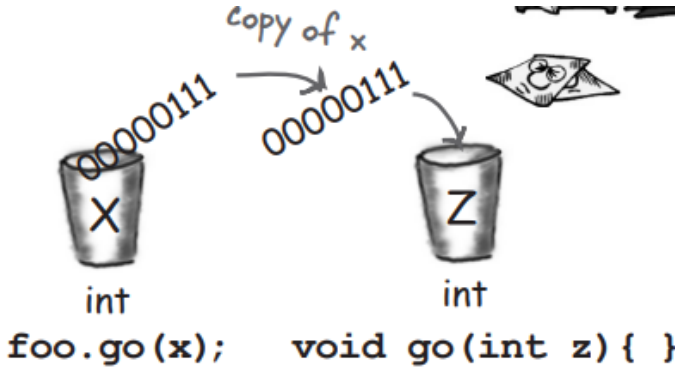
It also behaves same for the array that holds primitives

Strings are a special type of object. You can create and assign them as if they were primitives (even though they're references).

```
Eg. String name = "Avish";
```

Chapter 4 → How Objects Behave

Java is pass-by-value



Java is pass by value means pass by copy. The bits in x are copied, and the copy lands in z. Changing the value of z inside the method doesn't change the value of x!

Encapsulation

Make your instance variable private and provide public getters and setters for access control.

```
class GoodDog {
 private int size;

 public int getSize() {
    return size;
 }
 public void setSize(int s) {
    size = s;
 }

 void bark() {
    if (size > 60) {
       System.out.println("Wooof! Wooof!");
    } else {
       System.out.println("Ruff! Ruff!");
```

```
    }
   }
}
```

Declaring and initializing instance variables & difference b/w instance and local variables

Instance variables are declared inside a class and local variables are declared within a method.

You don’t have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variable get null.

Local variables do not get a default value! The compiler complains if you try to use a local variable before the variable is initialized.

Comparing Variables

Use == to compare two primitives or to see if two references refer to the same object.

```
//primitives
int a = 3;
byte b = 3;
System.out.println(a == b);  //true

// references
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
System.out.println(a == b);  //false
System.out.println(a == c);  //true
```

Use the equals() method to see if two different objects are equal.

```
String s1 = new String("Avish");
String s2 = new String("Avish");
String s3 = "Avish";
String s4 = "Avish";

// Compare using ==
System.out.println(s1 == s2);      // ❌ false (different objects)

System.out.println(s3 == s4);      // ✅ true (same reference from String pool)

// Compare using equals()
System.out.println(s1.equals(s2));  // ✅ true (same content)

System.out.println(s3.equals(s4)); // ✅ true
```

Chapter 5 → Extra-Strength Methods

Test code for SimpleStartup class

```
int [] locationCells
int numOfHits
String checkYourself(int guess)
void setLocationCells(int[] loc)
```

SimpleStartup have the above instance variables and methods. Then ask yourself, “If the checkYourself() method were implemented, what test code could I write that would prove to me the method is working correctly? The test code is given below -

```
public class SimpleStartupTestDrive {
 public static void main(String[] args) {
    SimpleStartup dot = new SimpleStartup();

    int[] locations = {2, 3, 4};
```

```
    dot.setLocationCells(locations);

    int userGuess = 2;
    String result = dot.checkYourself(userGuess);

    String testResult = "failed";
    if (result.equals("hit")) {
       testResult = "passed";
    }
    System.out.println(testResult);
 }
}
```

The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do.
As soon as your implementation code is done, you already have test code just waiting to validate it. Ideally, write a little test code, then write only the implementation code you need in order to pass that test. Then write a little more test code and write only the new implementation code needed to pass that new test. At each test iteration, you run all the previously written tests to prove that your latest code additions don't break previously tested code.

Bullet Points:

▼ Your Java program should start with a high-level design.

▼ Typically you'll write three things when you create a new class:
- prep code
- test code
- real (Java) code

▼ Prep code should describe what to do, not how to do it. Implementation comes later.

▼ Use the prep code to help design the test code. Write test code before you implement the methods.

▼ The concept of writing the test code first is one of the practices of Test-Driven Development (TDD), and it can make it easier (and faster) for you to write your

code.

Chapter 6 → Using the Java Library

Bullet Points:

▼ An ArrayList resizes dynamically to whatever size is needed. It grows when objects are added, and it shrinks when objects are removed.

▼ To put something into an ArrayList, use add(). To remove something from an ArrayList use remove(). To find out where something is (and if it is) in an ArrayList, use indexOf(). To find out if an ArrayList is empty, use isEmpty(). To get the size (number of elements) in an ArrayList, use the size() method.

▼ You declare the type of the array using a type parameter, which is a type name in angle brackets.
Example: ArrayList<Button> means the ArrayList will be able to hold only objects of type Button (or subclasses of Button).

▼ Although an ArrayList holds objects and not primitives, the compiler will automatically “wrap” (and “unwrap” when you take it out) a primitive into an Object and place that object in the ArrayList instead of the primitive.

▼ Classes are grouped into packages.

▼ A class has a full name, which is a combination of the package name and the class name. Class ArrayList is really java.util.ArrayList.

▼ To use a class in a package other than java.lang, you must tell Java the full name of the class. You get the java.lang package sort of “pre-imported”.

▼ Java 9 (and later versions) introduced the Java Module System, which means that the JDK is now split into modules. These modules group together related packages. This can make it easier to find the classes that interest you, because they’re grouped by function.

▼ Short-Circuit Operators `&&` and `||` evaluate boolean expressions efficiently by checking only what's necessary: `&&` stops if the left side is false, and `||` stops if the left side is true. This helps avoid issues like `NullPointerException` when checking conditions on possibly null references—for example, `if (refVar != null && refVar.isValidType())`.

▼ Non-Short-Circuit Operators `&` and `|` always evaluate both sides, which is useful in bitwise operations but less efficient in boolean logic.